Programming Assignment 5 - Boggle

Steven Cheng & Kyutae Sim

October 2020

1 Introduction

Boggle is a word game where 16 cubes, each with one letter on every face, are rolled and arranged into a 4x4 board. Players will then guess various words whose letters are adjacent on the board, and are awarded based on the length of the words. In our computer program adaptation, the player plays against a computer player, who will guess all words the player missed on the board.

For this assignment, our goals included creating a visually appealing GUI that would make it much easier for a human player to view the board and interact with the game. We also used this as an opportunity to understand more about various ways in which strings can be stored and manipulated. Finally, with the peer testing component of this project, we sought to create well-rounded tests that would provide useful feedback to any arbitrary Boggle project, regardless of specific implementations or design choices.

2 Solution

2.1 Design

Our solution design can be broken down into the 3 main components we were required to implement - the UI, the gameManager, and the gameDictionary. Generally, these were designed to be relatively independent of design choices, but still required certain assumptions (detailed in 2.2) in order to interact properly.

2.1.1 GUI

For our GUI, we wanted to prominently display the board, provide a clear display of the words and scores, and intuitive controls for playing the game. Thus, we chose the following design:

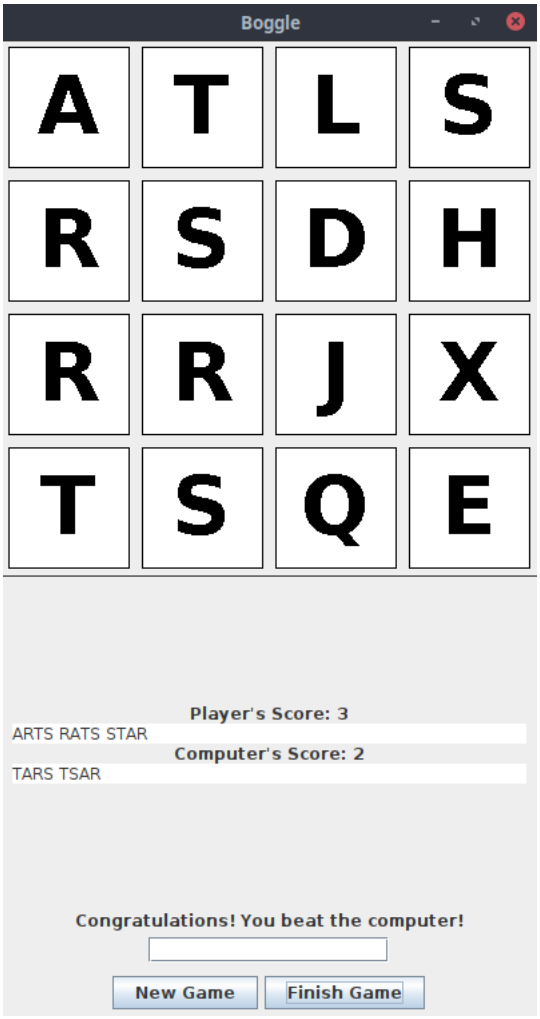

Figure 1: Our GUI after a game has been finished.

The board is displayed on the top of our GUI, and highlights correctly guessed words. Additionally, lines are drawn to connect letters together (see Figure 2). In the middle, the current player's score and list of words are displayed. Once all players finish, the computer's score and list of words will be added and the winner will be displayed. At the bottom is where the player actually interacts with the game. The player guesses words using the text field or finish their turn with the button.Above the text field is the provided feedback to the player for various error or success messages. A menu is also opened upon pressing new game that enables configuration of the game values, though it is filled with helpful defaults.

Figure 2: A successful guess by the player.

2.1.2 GameManager

For our GameManager, our design choices were relatively straightforward. The design of the required functionalities of GameManager are listed below, but the specific implementation details can be found in Section 3.1.2.

- **Setting, Resetting, and Getting Game State Values.** Internally, GameManager stores the board, every player's scores and guessed words, as well as the location of the last added word. Actions such as addWord and setGame will update these values, while copies of the internal values will be returned.

- **Handling Cubes.** To simulate rolling the cubes and arranging them, we first randomly ordered the cubes, and then selected a random letter from the 6 possible ones. The cubes are stored as Strings, since the only functionality that is really necessary is simply having access to all the characters to randomly pick one.

- **Searching the Board.** There are two ways the GameManager can search the board. The first is a word-driven search, where the computer searches the board for a specific word. This is used both for the addWord and the SEARCH_DICT tactic. The other search is an open-ended all words search, where the computer will search for all valid words. This search is used for the SEARCH_BOARD tactic.

2.1.3 GameDictionary

For our GameDictionary, there were many various designs we could have made, but few could satisify the general runtime requirements expected (For the runtime discussion, let N be the number of strings and L be the length of the longest string - though L can be considered a constant for most practical purposes). Most of our attention was focused on the following structures:

- **Indexed List.** The most obvious choice was to simply add everything to a list, and iterate through the list to check for the contains/isPrefix methods. However, this iteration would take $O(NL)$ time ($O(N)$ to iterate through the strings, and roughly $O(L)$ to check if the word is equal or a prefix). This also takes roughly $O(NL)$ memory, which is perfectly acceptable (since this is equivalent to just storing all the strings and nothing else).

  However, the list is by far the easiest to implement - the iterator would only require storing the index and incrementing it when appropriate (with $O(1)$ runtime for calling next), and the searches are straightforward and intuitive. When combined with Java's ArrayList and useful functions such as startsWith(), the list was by far the easiest to implement.

- **Sorted Indexed List.** We noticed that, in essence, the indexed list is performing a sequential search over the words. However, if we sorted the list after initially reading it in, we could now perform binary searches to achieve a runtime of $O(L \log N)$ for the contains method. If we wanted to apply binary search to the isPrefix method, we also considered creating a another list that would contain all the prefixes and apply the same process as above - sort the list initially, and then binary search to check for prefixes. This would achieve a runtime of $O(L \log NL)$ (since there are $O(NL)$ prefixes), but increase the memory cost to $O(NL^2)$. Note that introducing the sorts also adds a preprocessing step with runtime $O(NL \log N)$ for the normal words and $O(NL^2 \log NL)$ for the prefixes.

While this improves the runtime of the indexed list, it comes at a cost of being harder to implement (and more memory costly for the prefixes). However, using built-in Java methods such as Arrays.binarySearch makes this process easier. The iterator is still just as easy to implement, though.

- **Set.** We also considered using a set. Both ordered and unordered sets have Java implementations (TreeSet, HashSet), which would make them incredibly easy to implement. Their contains method also satisfies the runtime requirement, since it is $O(1)$ for HashSet and $O(\log N)$ for a TreeSet.

  However, we realized that implementing other portions of GameDictionary would be hard, if not impossible with sets. To implement isPrefix, we would still need to iterate over every string, yielding a runtime of $O(NL)$ for both kinds of sets (since iterating over a TreeSet is $O(N)$). Additionally, the iterator would be impossible to implement for a HashSet without directly using its iterator, since there is no concept of "order" other than the HashSet's internal hashed values. For the TreeSet, we could store our current element and use the higher method to find the lexicographically next element. However, this call to higher adds an (on average) a $O(\log N)$ factor to calling next().

- **Trie.** The last data structure we considered was a trie. Through the nature of a trie, both contains and isPrefix can be implemented in $O(L)$ time by simply traversing down the trie. In terms of memory, the trie has roughly $O(NL)$ nodes as well, meaning it is theoretically still equivalent to the efficient storage of an indexed list. Since it has $O(NL)$ nodes, the iterating through a trie also takes $O(NL)$ time total (with $O(L)$ runtime per next call).

  However, the trie is by far the most tedious to implement. Java does not provide any trie structure in its standard library, so every part of the trie would need to be implemented ourselves. Moreover, though the memory use is theoretically optimal, there is still a high overhead due to each node needing to map characters (edges) to their children. The iterator would also be hard to implement, as we would need to implement our own tree walk algorithm.

Ultimately, we ended up using the trie. When considering the required runtimes, only the sorted list and trie qualify (assuming L is a constant and can be removed). Between these two, we chose the trie since its only major drawback was the tedious implementation and we simply wanted to learn more about it.

2.2 Assumptions

While Boggle seems straightforward at a glance, there were many assumptions that had to be made due to the nature of GameManager, GameDictionary, and the GUI being in different classes (and assumed to be entirely independent of one another).

- **Case Sensitivity.** In general, it is not specified whether case matters at any of these steps. In the end, we decided to make all portions of this case insensitive. For the GUI, this would mean the player could guess their word in any case they wanted (eVEn LikE tHiS If ThEy REalLy WaNteD). For the GameManager and GameDictionary, they would also accept both kinds of words. The files words.txt and cubes.txt would allow both kinds of letters, leading to some interesting boards consisting of all lowercase letters.

- **Boggle Rules.** The majority of the Boggle rules were kept in the UI, such as how the computer should play, what order the players play in, etc. The only exception to this was the bare minimums required to run GameManager - specifically, the score calculation and players being unable to guess words they had already guessed. The main implication of this is that while addingWords and the internal game state changes act under Boggle rules, getAllWords will return any word that is both in the dictionary and on the board regardless of whether it is greater than 3 characters long.

- **Invalid Parameters.** For invalid parameters, we would choose do nothing (and throw an error) if this is the case. This is to handle, say, if the GUI accidentally passes bad parameters to the GameManager - an exception will be thrown, which we can then catch and forward to the user.

- **Loading the Dictionary.** It was not clear when and where the dictionary should be loaded. Therefore, we assumed that the dictionary passed into gameManager is already loaded (or at least loaded before the dictionary is ever used). Additionally, it is assumed that the Dictionary will not be reloaded in the middle of the game, since already guessed words would suddenly become invalid if this would happen.

- **File Specifics.** We assumed that both words.txt and cubes.txt were allowed to have any kinds of characters, if one theoretically wanted to play a game where valid words were prime numbers or something. We also assumed that each cube can only have 6 faces - no more, no less.

- **getLastAddedWord Reset.** While it would be convenient for getLastAddedWord to return null after an invalid guess (since this would automatically remove the highlight), we assumed that getLastAddedWord should still return the points of the last sucessfully added word, as this is what we assumed this is what was required by the BoggleGame docs.

3 Discussion

3.1 Implementation

3.1.1 Boggle GUI

Our GUI was implemented using Java Swing. Most of the implementation was just utilizing the Swing components to create the UI design we intended. To make the UI actually function as a game of Boggle, however, we used the methods provided by GameManager to define values and provide special error messages.

Upon running Boggle, a new GameManager and GameDictionary are created. The GameDictionary file is also loaded initially. Everytime newGame is called, the game parameters as well as the GameDictionary are passed into the newGame method, and any errors are reflected in the UI. Guessing words passes a guess to the addWord method, while clicking Finish will switch the current player to the next one (or cause the computer to play its turn).

For displaying the UI, getBoard and getLastAddedWord are used to display the word, highlight letters, and draw the lines between the cells. Note that to achieve the effect demonstrated in Figure 2, the lines would be drawn first, and then the board would be drawn over it. The scores are retrieved from the getScore method as well.

For the guessed words list we had to store these separately in the Boggle GUI. There were no accessor methods to view the internal lists within the GameManager, and we needed to both display this list for the player and use it to determine which words the computer can guess.

3.1.2 GameManager

For storing the internal state of GameManager, we store the board as a character matrix, the list of guessed words as an array of HashSets, and the player scores as an array of ints. Additionally, we store a list of points for the locations of each letter of the last successful word, as well storing the board size, number of players, and the Random object used to roll the cubes.

For the required methods, their details are listed below. Note that all the string or character comparisons are case insensitive.

- **newGame.** In the new game method, we first validate all the given parameters, throwing IllegalArgumentExceptions when they are not valid. Afterwards, we instantiate the guessed words and score arrays. In addition, we also load cubes from a text file and create the board (discussed below).

- **loadCubes.** loadCubes is a utility method we created that handles reading from the cubes file and put each cube into an ArrayList. loadCubes also handles validating the cube file, checking if there are enough cubes and each cube has exactly 6 characters.

- **createBoard and roll.** createBoard and roll are utility methods we created that handle using those cubes to create a board and roll the cubes. createBoard first shuffles the cubes using Collections.shuffle() and using the cube at index $4i + j$ to put in board[i][j]. Then we "roll" each cube using roll(), a method that randomly picks one of the 6 possible characters using a Random object, to determine the character that goes there.

- **setGame.** For setGame, we use a nested for loop to deep copy the given board. In addition, all scores are reset to 0 and the word lists are all cleared.

- **getBoard.** Returns a deep copy of the internal board.

- **addWord.** We first validate the given arguments, checking if the player number is in bounds and throwing an IllegalArgumentException if it isn't. Then we check whether the word was already found or if it less than 4 characters long, returning 0 if so. Otherwise, we use the findWord method (detailed below) to attempt to find the word within the board. If this is successful, we update the current player's score and guessed word list, returning the score for this word. Otherwise, we return 0.

- **findWord.** Our findWord method is actually an overloaded method. The first method only takes the word to be searched as a parameter. It sets up a "visited" matrix and iterates over the board, starting the search at each spot on the board (implemented in the other findWord method). If the word is found, it will use the values from the visited matrix to get the location of this word, and returns true. Otherwise, it returns false.

  The visited matrix is an int matrix that stores whether the recursive search visited a certain spot on the board. -1 means that the value has not been visited, while anything else corresponds to what index that letter corresponded to in the searched word. That is, if the word "FOUR" appeared in the very top row of the board, you would see the top row of visited as $[0, 1, 2, 3]$.

  The findWord search method takes as parameters the indices of the the board we are currently searching at, the word we are trying to find, the current index of the word we are trying to match, and the visited matrix. It returns a boolean to say whether or not the word has been found.

  It starts by checking if we have found the word (returning true if we have) and then checking to make sure we aren't revisiting cells or out of bounds (returning false if so). If the current letter we are at does not match whatever is at word[index], we also return false (since this already fails to make the word).

If none of the above checks fail, then the findWord has already found some prefix of the word, and needs to continue searching. First, we set the visited matrix to the current index. Then we loop over the 3x3 area centered around the current cell, continuing the findWord search on each of these (since these are all valid adjacent letters). Note that, even though we revisit the current cell, this call instantly terminates because we have already visited it. If any of the findWord calls return true, then we instantly return true as well. This has the effect of "freezing" the visited matrix when we have found the word, allowing us to compute the location of the words as described earlier. If none of the findWord calls are successful, we change the visited matrix back to -1 and return false.

- **getLastAddedWord.** Simply returns the points we have computed and found from findWord() and addWord().

- **getAllWords.** The two search tactics have implementations discussed below. Both of these add the words to a TreeSet to be returned, which was an arbitrary choice we made to keep the output sorted nicely and not contain duplicates. Any standard data structure would have worked, such as a HashSet or ArrayList, assuming that the dictionary did not repeat any words. Additionally, we uppercased all words since this would provide consistency between SEARCH_BOARD, which depends on the cases found on the board, and SEARCH_DICT, which depends on the cases found in the dictionary file.

  - **SEARCH_BOARD.** To search the board for all words, we used a similar method to the findWord() recursion (implemented in the findAllWords) for searching all words on the board. The only difference is that rather than storing a target word and the current index we're at in the word, we store our current prefix and always check that this is a valid prefix within the dictionary. If this prefix is a dictionary word, we add also add it to our list. While this doesn't have the intense optimizations that are available when we have a target word, we still prune the search whenever the current prefix doesn't match any prefixes in the dictionary.
  - **SEARCH_DICT.** This search tactic simply iterates through the dictionary and adds the word to our list if found on our board using the findWord() method. We also make sure that the last added word points are not modified by this.

- **setSearchTactic.** If the SearchTactic passed in the parameter is neither of the tactics we use, we set the tactic to the default. Otherwise, we set the tactic to the tactic passed in the parameter.

- **getScores.** Returns a copy of the scores array where we stored all the players' scores.

3.1.3 GameDictionary

The GameDictionary implementation can be broken into the TrieNode and TrieIterator classes.

Each TrieNode has a map that maps character edges to the associated next TrieNode, a pointer to its parent TrieNode, and a boolean signifying whether this node is an "end" node (i.e. a dictionary word ended at this node). To construct the trie, a root node is first instantiated (with null as its parent). Then, we call the root's add method with each string in the dictionary. The add method traverses down the trie according to the word's letters, adding new edges and TrieNodes whenever needed. Once it reaches the end, it marks that node as an end node (setting its boolean to true). This is all done upon a call to loadDictionary.

To check if a word is a prefix or contained within the trie, we use the isPrefix and contains methods in TrieNode. To check if a word is in the trie, we can simply traverse down the trie according to that word and see if we end up at an end node. If we attempted to traverse down an edge that did not exist, we can just return false, since the edge would exist if the word was in the dictionary. The prefix check is similar, but we return true regardless of if we end at an end node - being able to traverse some part of the way through a trie means that this path was part of the path to a whole word, and thus is a valid prefix.

To iterate over the dictionary, our TrieIterator performs a modified version of a preorder traversal, where it only stops on end nodes. It keeps track of the current "prefix" based on the edges traversed, and returns this prefix as the string once an end node is reached.

To actually implement this, we kept track of the current node, the depth of that node, and a modified form of the prefix that would sometimes have an extra letter attached. This "extra letter" was used to store what the last traversed edge was at some node, so we could determine what should be traversed to next. To make this process easier, we created a method advanceNode that moves to the next node. It works as follows:

- If there are no children for this node, decrement the depth and set the current node to its parent. Do not delete the last character of the prefix, since we need this to determine which edge to traverse to next.

- If there is no extra character on the prefix (i.e. the length of the prefix matches the depth), then this must be the first time we are at this node. Traverse to the first child, increment the depth, and add the letter of the edge we just traversed to the prefix.

- Otherwise, we have already visited this node before, and need to determine the correct subtree to visit next. The extra character on the prefix tells us the edge to the subtree we just visited, so we pick the character in the map that follows the extra character, and traverse to that node (updating currentNode, incrementing depth, and changing the last character on the prefix to whatever edge we just traversed).

- If there is no next character, then we need to go backwards. Current node is set to its parent, and the depth is decremented. Note that we also remove the last character on the prefix, since this extra character is no longer useful to us (it was associated with specifically which edge was traversed at this node).

For the next method of the TrieIterator, we keep advancing the node until we hit an end node. However, we only want to consider an end node the first time we reach it. Thus, in addition to checking if a node is an end node, we also need to make sure this is the first time we have visited it. We can do this by checking if the length of the prefix equals the depth, since this would mean that no extra character exists (aka no previously traversed edge exists). To implement hasNext, we just need to check if the currentNode is null, since the traversal will move to root's parent after it is done (and root's parent is null). If we are on the last word, however, this will return "true", despite the fact that there are no more words after the last word. To fix this, our iterator will find the next node in advance. Specifically, calling next actually returns the prefix associated with the previous node, and it advances the internal node to the next word for the next call. This way, when next returns the last word in the dictionary, the internal node finishes the rest of the traversal and ends at null.

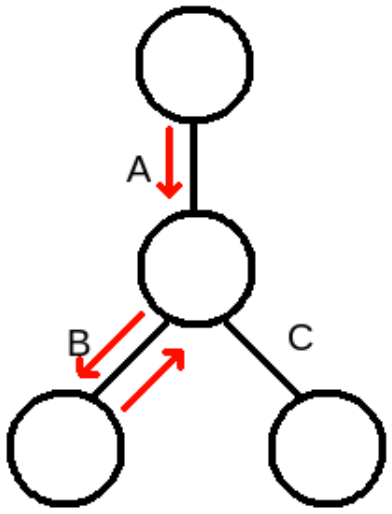

Figure 3: If TrieIterator has already traversed according to the red arrows, its current prefix would be AB, but its current depth would be 1. The extra B tells the TrieIterator to use the C edge, rather than go back down to the B edge again.

3.2 Special Cases

For Boggle, there are a surprising amount of special cases that needed to be considered. Some special cases were only resolvable through assumptions, and are listed both here and under assumptions.

- **Case Sensitivity.** As mentioned in assumptions, we assumed case does not matter for comparisons. The implementation discussion talks about how exactly this is handled, since the behavior changes for different methods.
- **Invalid Parameters.** In general, we handle invalid parameters by throwing an IllegalArgumentException and not changing game state. This also allows us to catch these Exceptions and display their messages to the user.
- **Bad Cube Files.** If the cubes file does not contain enough cubes (or the cubes are malformed according to our assumptions), we also throw an IllegalArgumentException.
- **Invalid SearchTactic.** The SearchTactic enum seemingly only takes on one of 2 values, both of which are valid. However, calling setSearchTactic(null) is still possible. In this case, we use the default SearchTactic instead.
- **Bad SetGame Boards.** If the board passed into setGame is not square, we throw an IllegalArgumentException and do not change the board state.
- **Pointer Issues.** When calling getBoard and setGame, we make sure to deep copy everything to avoid potential pointer issues. Specifically, these issues can occur if the matrix passed into setGame / the matrix retrieved by getBoard are modified after the method call. If the deep copy never occured, these would change internal game states.
- **SetGame with Irregular Board Sizes.** For non square boards, this action should not be allowed (and throws IllegalArgumentException). However, if the board is square but does not match the original size, we need to update the internal size variable to match the new board's size.

3.3 Interesting Results

There were not many "interesting" results - mostly, the computer player would pick up infinitely more words than we could hope to guess. However, on two occasions, we were able to beat the computer, which are shown below.

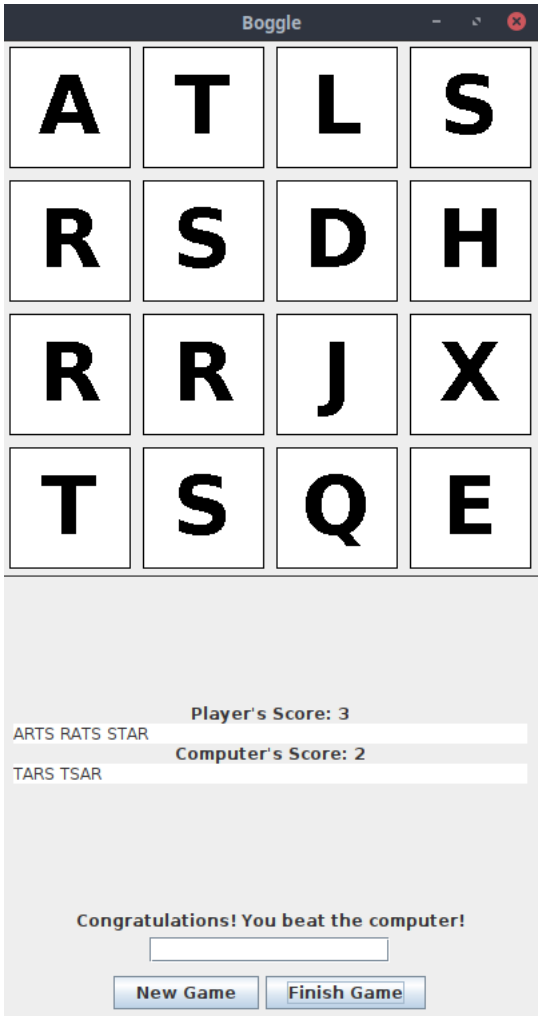

Figure 4: A board with only 5 words total.

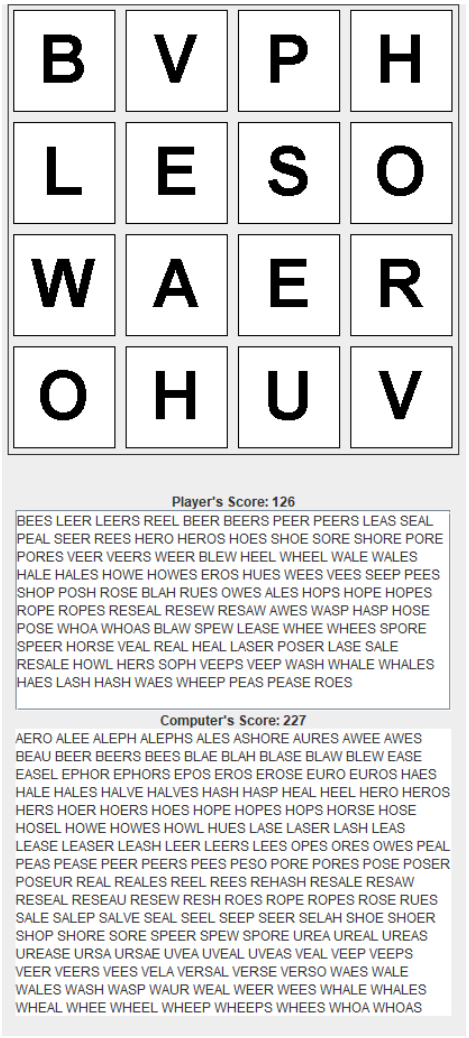

Figure 5: The results of an hour of thinking. This was when the computer still guessed words that were used by the human, so the final score is actually 126 to 101.

4 Testing

These tests are more comprehensive and covered almost everything we can think of, since we used these for testing both our and others code. A brief explanation of the files we created can be found in section 4.4.

4.1 Overall Gameplay

In order to test the program rigorously, we tried several of cases categorized into the following:

- **Board Sizes.** When given the ability to input the size of the board, we try various sizes as well as invalid sizes such as non-numbers.
- **Player Number.** When given the ability to input the number of players, we try various numbers as well as non-numbers.
- **Dictionary Words.** Tests dictionary words during the gameplay that are either: correct according to the rules, shorter than 4 characters, on the board but not connected, on the board but uses one or more cube multiple times, not on the board at all, or previously guessed words.
- **Non-dictionary Words.** Tests non-dictionary words that are on the board and not on the board.
- **Invalid Input.** Tests some edge cases of invalid inputs during the gameplay such as an empty string, uppercase strings, lowercase strings, strings with mix of uppercase and lowercase characters, and strings that consist non-letter characters.

In all of these test cases, we simply make sure the game behaves correctly as expected. Expected behaviors include giving appropriate error messages, handling invalid inputs correctly, updating scores accurately, as well as showing the computer's word list at the end of the game play. In addition, we make sure that program is able to acknowledge valid words that adhere to the rules and tell them apart from invalid words or invalid inputs.

4.2 GameManager

Most methods were tested in their own JUnit file, though some similar ones were grouped together. The list of tests can be found under each of the subheadings. For the tests that are related to valid actions, there are correct outputs that we can verify. For the edge case / invalid input tests, however, these tests are more to determine if these cases are even handled rather than validating a specific output.

4.2.1 newGame and getBoard (GameManagerNewGameTest)

The tests can be divided into 3 groups:

- **Player Number Tests.** These checked how GameManager reacted to various player numbers, including invalid tests. For the valid ones, we checked if getScore returned an array of all zeroes with the same length as the number of players.
- **Board Size Tests.** These checked how GameManager reacted to various board sizes, including invalid tests. Bigger cube files were provided for bigger boards.
- **Board Correctness Tests.** These checked if GameManager created valid boards given our custom cube files, since these were designed to make validating boards easy. TestCubes 1 and 4 allowed us to make sure each cube was being used once, while 2 and 3 checked to make sure that only valid characters were being used.

4.2.2 addWord and getScores (GameManagerAddWordTest)

To control the board and dictionaries, we either used testCubes6 with the normal dictionary or testCubes5 with testDic1. This way, we can guarantee that "COCO" is always a valid word for testCubes6 and any sequence of "B" is valid for testCubes5. The tests can be divided into 4 groups:

- **Case Sensitivity Check.** These checked if we were awarded points for adding COCO, coco, and CoCo. These also doubled as basic point checks.
- **Point Check.** Using testCubes5, we tested every sequence of B's on a 10x10 board to see if each word gave the correct number of points.
- **Player Point Check.** Using testCubes5, we added the string BBBBB to various players, and checked if the getScore array matched which players we added the string for.

4.2.3 getLastAddedWord (GameManagerGetLastAddedWordTest)

The getLastAddedWord tests consisted of hardcoded matrices that were passed into the gameManager. Each matrix contained the word APPLE in varying directions, with the rest of the letters as Z. We checked that the returned list of points exactly matched the location of APPLE for each matrix.

4.2.4 setGame (GameManagerSetGameTest)

The setGame tests consisted of a variety of edge case testing:

- **Correctness Check.** A simple check to see if the board is being set to what we gave it.

- **Reset Check.** These tested use testDic1 and a board of all Bs to add the string BBBB to a player. Then they call setGame and check if the guessedWords and the scores are reset.
- **Pointer Check.** These tested if there were pointer issues related to setGame or getBoard, as described earlier.
- **Non-square Board Check.** These tested how boards that were not square were handled when passed into setGame.
- **Different Sized Board Check.** These tested how boards that were square but did not match the original board size were handled when passed into setGame.

4.2.5 getAllWords and setSearchTactic (GameManagerGetAllWordsTest and GameManagerSetSearchTacticTest)

These tests checked if all words were found for the 3 search tactics (explicitly SEARCH_BOARD, explicitly SEARCH_DICT, and the default) for the following cases:

- 2x2 board of all Bs, with testDic2 (B, BB, BBB, BBBB)
- 2x2 board of all As, with testDic2 (no words)
- 2x2 checkerboard of B's/A's, with testDic2 (B, BB)
- 10x10 board with stripes of B's, with testDic2 (any sequence of B with length 1...10)

Additionally, we tested if setSearchTactic(null) followed by getAllWords() still successfully retrieved words.

4.3 GameDictionary

For our GameDictionary tests, we tested the following on each testDic file (as well as words.txt):

- loadDictionary runs successfully
- Every word in the dictionary file returns true when passed into contains
- Every prefix of a word in the dictionary file returns true when passed into isPrefix
- The HashSet created by adding each word from the file exactly equals the HashSet created by adding each word given by iterating through the GameDictionary.

4.4 Custom Cube and Dictionary Files

The cube files and dictionary files we made separately so as to make the testing more efficient are described in the table below.

Cube Files

File name	Description
longercube.txt	Cube with 8 characters instead of 6
lowercase.txt	Random strings that are all lower-case
randomchar.txt	Random characters that aren't letters
shortercube.txt	Cubes with 4 characters instead of 6
testcubes1.txt	Cubes with the same letter for all 6 characters (AAAAAA....PPPPPP)
testcubes2.txt	Same cubes of "ABCDEF"
testcubes3.txt	Same cubes of "ABCDEF" for a 6 by 6 board (36 cubes)
testcubes4.txt	Same as testcubes1.txt but repeated 6 times for a 6 by 6 board
testcubes5.txt	126207 cubes of "BBBBBB"
testcubes6.txt	8 cubes each of "CCCCCC" and "OOOOOO"

Dictionary Files

File name	Description
testDic1.txt	Words consisting only b's ranging from length 1 to length 100
testDic2.txt	Random words that start with 'a'
testDic3.txt	"repeated" repeated 147600 times
testDic4.txt	Random strings
testDic5.txt	Upper case dictionary

5 Pair Programming

5.1 Time Log

Date	Time	Description
10/15/20	23:30 - 25:30	Whiteboarding and planning together
10/17/20	16:30 - 19:00	Steven driving, Kyutae navigating
10/17/20	00:00 - 02:00	Kyutae driving, Steven navigating
10/17/20	02:00 - 04:00	Steven driving, Kyutae navigating
10/19/20	01:00 - 02:00	Steven driving, Kyutae navigating
10/19/20	02:00 - 03:00	Kyutae driving, Steven navigating
10/22/20	23:00 - 25:00	Whiteboarding and planning together (peer review testing)
10/24/20	19:00 - 21:00	Steven driving, Kyutae navigating (Testing)
10/24/20	22:00 - 00:00	Kyutae driving, Steven navigating (Testing)
10/25/20	13:00 - 17:00	Peer review testing together

5.2 Discussion

For this pair programming assignment, our pair programming experience was relatively standard. The work was entirely done through the driver/navigator method, largely just implementing the things we had planned earlier, where the navigator was mostly checking for bugs and verifying that the code works as we had intended. For creating and using our testing harness, it was useful to have the navigator double check that the tests we were creating did not rely on assumptions or implementation details specific to our project.