Overview

The objective of this project is to build the game of Boggle, which is just a word hunt game. Building a game like this requires efficient data storage and querying, as well as a user friendly interface.

Goals

Our personal goals for this project were to develop a game completely from scratch. The previous labs had a given UI, but this lab gave us free reign to implement this game however we liked.

Solution Design

GameDictionary:

We store the words in our dictionary as a trie. This requires building a tree with nodes containing strings and edges signifying appending a character to the parent. For example, if the node with value “he” has an edge with the value “y”, we know that the child resulting from taking that edge has value “hey”. At initialization, we start with a root of value “”, and calling loadDictionary adds words to our trie by iterating through the loaded words.txt file and lowercasing them for case insensitivity.

We initially stored the children of each node as an array of size 26 (with index 0 representing ‘a’, 1 representing ‘b’, etc.), but realized that it was wasting memory, which could lead to scenarios such as a Java heap space error. So we switched to a HashMap where the key is the character of the edge and the value is the child node. This operated in O(1) time and allowed for non alphabetic characters to be accounted for, but when creating our iterator, we wanted to

iterate in alphabetical order which isn't simple with a HashMap. This led us to switching to a TreeMap, making the iteration very easy but sacrificing some time complexity. However, we do assume we are working only with ASCII characters, which means that there are a maximum of 128 possible edges for each node. Since this upper bound is a constant (and also small which is unrelated but nice), running a O(log n) complexity TreeMap on it will still be constant, so we can treat its operations as constant time for later operations. Also note we store the keys of each TreeMap in reverse order since for the DictionaryIterator, the stack must be pushed also in reverse order.

The process of adding a word starts at the root and goes through each character in the word sequentially. For each character, the current node moves to the child that has the edge that has the same value as such character. If that edge doesn't exist, then we create the edge with its child node and continue the traversal. Once the end of the word is reached, we signify that the final node is a full word by setting the internal boolean validWord of the node to true. We cannot just look at leaf nodes since some valid words are prefixes of one another (ex. he and hey), and we cannot assume all nodes are valid words since some are just prefixes required to reach the final word (ex. "az" is created when adding "azure" but is not a valid word). Notice that the adding of a word is just a find query that adds edges when it reaches leaves it needs to continue iterating on.

With the loading and adding of words being handled in loadDictionary, we also needed to implement isPrefix() and contains(). These are just searching for a word in the dictionary, and if it's found then we return true, and false otherwise. However, contains has the extra condition that its interval boolean validWord must be true as well. The process of searching for the word is just a find query, like adding the word, so it has a similar implementation and complexity. The only

difference is that in the case that the edge it needs to take doesn't exist, we end and return false, so it is faster in practice.

In most cases, these find-based operations run in O(w) time, where w is the length of the word that is being added. We chose the trie specifically for this time complexity as well as because its implementation fits the problem statement of the dictionary cleanly.

The dictionary has one more method it needs to implement, and that is an iterator. We created a DictionaryIterator class to represent this iterator. At its core, it is doing an iterative depth-first traversal of the trie using a stack (like we talked about in class). At initialization, it is passed the root, which is pushed to the stack. Calling the next value just means that it will pop the top node of the stack and add all of its children in reverse order from the TreeMap (see TreeMap reasoning above). If the popped node is a valid word (node.validWord is true), then we find the next item in this alphabetical iterator and return it. Because of this property, we know that there must be a valid word we have not reached on in the iterator if the stack is not empty. This is because a node only exists if its a prefix (or equal to) a valid word, so if a node is in the stack it either has a node in its subtree that is a valid word or it is a valid word itself. This makes hasNext pretty simple: !stack.empty(), which is an O(1) operation. Since this is a depth first search, we have to iterate through all n nodes. And since the popping of a stack for each node is O(1) and the adding of all its children is also O(1) because it has a constant maximum of 128 children (see ASCII in TreeMap reasoning above), the total iteration takes O(n) time.

<u>GameManager:</u>

- newGame: Initializes game specifics (board, dictionary, size, num players and their scores, etc.) to default or specified values. Checks if inputs are valid (ex. size > 0) and if

they aren’t then it sets them to their default values and informs the user of this mismatch. In the case of using newGame from the UI, it will reprompt the user. Initializing the board is getting all non-empty cubes in the cube file, and shuffling the cubes, and choosing a random face from the first $size^2$ cubes.

- getBoard: returns board
- addWord: makes word lower case, checks if long enough and a valid word in dictionary and on board, checks if anyone has guessed the word already, updates player scores and lastAddedWord. Checks if in dictionary using contains(), and checks if word is on board using a depth-first search (DFS) starting at each square on the board. The DFS keeps track of what order it visits squares using a 2D visited array, which not only allows it to not visit the same square twice, but if the word is found on the board, it also tells us what points correspond to that word and in what order they were reached (for lastAddedWord). This is done by setting the location in visited to its index in the word when it first reaches it, but after all its children are called in the DFS, it resets back to 0.
- getLastAddedWord: returns last added word computed in addWord (if not computed yet, defaults to null)
- setGame: Sets board to input 2D array and resets player data after checking if its the right size, if it's not it informs the user and doesn’t update the board
- setSearchTactic: updates stored search tactic to passed value
- getScores: returns array of player scores
- getAllWords: Runs the current search tactic (if not valid search tactic informs user and returns empty output). If it's SEARCH_DICT, it iterates through all the words in the dictionary (see dictionary iterator above) and if the word is long enough and is found on

the board (see DFS in addWord) then it adds it to the hash set of all possible words. If it's SEARCH_BOARD, it runs a DFS (like addWord) starting at each possible starting spot in the board, but instead of trying to find a word, it sequentially adds characters by moving through the board, creating a word. If the word is contained in the dictionary and it's long enough then it's added to the hash set of all possible words. We also check at each step if the current word is a prefix in the dictionary, since if it's not then there is no reason to keep iterating through the DFS.

- Since SEARCH_DICT iterates through all n words in the dictionary, and checking if a word exists on the board requires going through b^2 starting spots and at most reaching w spots (b = size of board, w = length of longest word). Technically, you can reach more than w spots since there are cases where you can build prefixes of a word in multiple ways, so you can go down a valid path that doesn't actually find the word. However, in a standard game of Boggle that is based on words in the English dictionary and randomized starting boards, the number of such similar paths on average is extremely small, call it c. This gives approximately $O(nb^2wc)$ time complexity. This is a rough estimation only used to compare with the other tactic.
- On the other hand SEARCH_BOARD starts at b^2 spots but splits into 8 paths and does an $O(w)$ check to see if a word is contained or prefixed in the dictionary (see dictionary complexity above) each time. However, not all of these 8 paths continue because we check if the current word is a prefix every time, meaning that they terminate early. So let's call the amount that it splits into on average p, where we know $p > 1$. In most English cases p is small but if the dictionary has a

large amount of random strings of characters without structure then p can get closer to 8, which makes it exponential w.r.t to w (which in the same situation will also likely be large). However, sticking with normal cases, this gives approximately $O(b^2p^ww)$ time complexity. Again, this is a rough estimation only used to compare with the other tactic.

- This clearly shows that for smaller dictionaries (making n small), SEARCH_DICT is better because nc is better than p^w but when we have larger dictionaries (and also on smaller boards since p decreases for the amount of items we have already seen) then nc is much slower than p^w so SEARCH_BOARD is faster. Also if all the words in the dictionary are small, then we find that the w terms are negligible so nc is slower than p^w making SEARCH_BOARD also faster in this case. However, these are only estimates on time complexities, in practice it's likely that both methods are faster than their respective approximations.

User Interface:

Because it was more simple and it would be easier to scale the board output for different sizes, we decided to create a text-based user interface. As a result, our solution design effectively boiled down to performing a sequence of prompts and events. To start out, we prompt the user for all the parameters (words file, cube file, board size, and number of players). With each parameter we error-check in a similar way by ensuring that it meets the necessary conditions, and if it doesn't then we display the error message and prompt the user to input the parameter again. Once they are all valid, we check if the conditions are met to play a game. Specifically, we store

in a HashMap the player number and a boolean as to whether or not they are still guessing. Then, if there exists at least one player who is still guessing we will carry out the game for those player(s).

We prompt the user for their guess, then update their score accordingly. If their guess is invalid or already guessed on the board, then we ask the user to enter a new word. The UI will then interact with GameManager to add the word and display the chosen word's letters as capitalized on the board. The user might also want to terminate any future turns for the game, so if they enter "!" (our keyword to terminate their turn) as their guess, then the map will change the boolean value at the player number's key. We repeat this process for the remainder of player's still guessing. After all players are done guessing, the final scores are displayed, all the possible words are shown, and we ask the user if they would like to play again. If they would like to play again, we do this entire procedure again and reset/re-prompt for all the variables.

In regards to our error checking methodology, the process was the same for most of the parameters, as we would first verify that it is of the right type if it needs to be (if we are expecting an integer, we ensure that it isn't a string for instance). We then verify that the input will make sense logically, so for the number of players, we force the user to input an integer greater than 0. The UI accomplishes this by sending the information to GameManager, which will display the associated error message and temporarily make the value for the parameters a default value for safety reasons in case GameManager is used at the current state.

Assumptions

We make the assumption that our code will run in sufficient time and memory, meaning that with extremely large dictionaries in specific unique structures (see scope) we don't terminate

our program after a lot of time or memory was used so some errors can arise in those scenarios. Also, we generally assume that the user is either interacting with GameManager mostly through Boggle.java or using GameManager and GameDictionary properly, but even if they aren't we handled most of these error-causing scenarios.

Scope of Solution

As previously mentioned, the issue with our scope would be its time complexity in some unique scenarios. For example, we assume the dictionary is only passed ASCII characters and not hypothetically Unicode characters, because there are only 128 possible ASCII characters but over one million Unicode characters. If all Unicode characters are added would make our trie significantly slower because of the O(log n) complexity of the tree map, so we would have likely stuck with a HashMap if that is the case. Additionally, we also assume the dictionary consists of valid English words (or something with structure that is non-exhaustive of some length of string combinations) for the complexity of both searches. If the input words.txt file was really long and in this form, then SEARCH_DICT's constant *c* will increase greatly, but more significantly the SEARCH_BOARD method will increase exponentially.

Quality of Solution

Our code uses many helper methods to make it clear what each method is doing. We commented our code and structured it to make it as readable as possible, though in GameManager we do recognize that there are two different DFS methods that could get confusing with one another. Also understanding how we used the 2D visited array might be confusing for those that aren't familiar with DFS.

Interesting Results

An interesting result we found was with how the board finds palindromes. For example, if “kayak” is on the board, then in what order are the points returned? For our implementation, we start at the top left and move right and then down, but if we went down and then right it could give a different result (or any other configuration, but those two are the most common). This can be used to get an idea of the internal workings of other people’s code without actually decompiling anything. An example of a board that would show this is:

A K A

K Y A

A A Y

The order of the returned points signifies how their search is actually working, which we thought was a pretty cool way to get insight into code you can’t directly see.

Problems Encountered

The peer review helped us find that the points we returned from getLastAddedWord were not in the right order, which was because visited was a boolean value that was stored if we saw it or not (and not in the order in which we actually saw it). By updating it to an integer array, our problem was solved. The peer review also gave us an instance where the search_board and search_dictionary gave different outputs. Specifically, we were shown that search_dictionary finds a few more words than search_board. We saw that those values were valid so the issue was

with search_board. After some debugging, we realized in searchAllWordsDFS for search_board, we were adding the character to the word after adding it to our set and not before, so some instances were actually found but not reflected in our output. We also found that if we tried iterating over the dictionary without loading it, then its hasNext would return true because we were pushing the root even if it didn't have children (an edge case we didn't consider initially).

Testing

As we were able to easily verify that the output state was correct, we utilized both blackbox testing and whitebox testing. We utilized automated testing primarily for the internals with GameManager and GameDictionary and used manual testing for the user interface, specifically to catch edge cases, as it was simple and easy to spot errors in our code. We also used a lot of these tests to check other groups' programs for our peer reviews.

Generated Board Environment

```
char[][] board = {
    {'p', 'a', 'a', 'a'},
    {'a', 'l', 'a', 'a'},
    {'a', 'a', 'a', 'a'},
    {'a', 'a', 'a', 'n'}
};
```

Automated Blackbox Unit Testing

- The first test we performed was ensuring that board equality would hold true. When setting the current game board to our own generated board and then calling getBoard(), the expected result returned would be an identical board to the one we generated. Thus, traversing through each element in our board should be the exact same as the one received by getBoard() at the same position.
- Another test we performed was testing if addWord() and getLastAddedWord() are functioning properly. Using the same generated board as above, we compared if adding a

valid word and then getting the points of that word on the board using getLastAddedWord() would indeed match the expected points on the board. In this case of our generated board, adding the word “plan” would be expected to return a list of points across the diagonal.

- To test the search tactics and verify that they are working, we also use a generated board and predetermined the valid words possible on it. After setting the search tactic to either search the dictionary or search the board, we traversed through all the words possible on the board from getAllWords() and compared to see if they matched up with the words we predetermined to be on the board.
- The last blackbox test we had was to verify that our addWord() and getLastAddedWord() did not accidentally have multiple points for the same characters. Using the generated board above, we ensured that the number of occurrences of each character would match its expected value. For example, in the generated board environment, the list of points should only have a point to one “a” character when the word “plan” is added.

Automated Whitebox Unit Testing

- Testing the dictionary’s loadDictionary() and contains() methods, we verified that when loading in all the words from words.txt that it would contain all the lines in the file. We used Scanner to traverse through the file and compared each line one by one to see if they were equal or not. If they weren’t equal, then we knew that something with our loadDictionary() or contains() would be incorrect.

- To test the board prefixes, we traversed through all the words in the dictionary and then subsequently found each of the word's substrings starting from index 0. Then, we checked if each substring was indeed included by seeing if isPrefix() returned true.
- The last whitebox test we performed was testing the dictionary's iterator. Specifically, we kept a count of the number of times the iterator would run if it were traversing through all the words in the dictionary. We then compared that number to the number of lines in the dictionary to determine whether they were equal. If they were not equal, we would know that the iterator did not reach every word in the dictionary.

Manual Testing

- To validate that our user interface was working as expected for most cases, we performed basic sanity checks throughout the development process. This involved playing the game as normal and making sure that the appropriate result would happen. For instance, this meant that when a word was guessed, the right letters would be highlighted, the next player would go, and the player's score would update accordingly.
- After verifying that our code worked with basic cases there were a few edge cases we had to consider with the user interface
    - Input Validation: since we ask the user to input parameters for file names, size, and the number of players, it is important that we check for invalid inputs. These could be more explicit like when the expected response is an integer and the user enters a string, or more subtle like when they input a board size too big that the cubes file can't generate enough configurations. We tested all possible combinations of these and ensured that they handled all situations safely.

- Word Guessing: when a player guesses a word, the word needs to not be already entered before and not too short. We tested circumstances when these conditions were true, and made sure that they made appropriate responses correctly. We also verified that capitalized words work as well.

Social Impact

Dictionaries are used heavily in content moderation. Specifically, whoever might be trying to filter out certain content would use a dictionary to store inappropriate language and block out any results using that language. Companies like Google and Facebook use this methodology to create a safer environment for its users (Parker). While Boggle's use of a dictionary is not the exact same, it still relies on the same principles of searching for content contained in the loaded dictionary. Speaking from personal experience, we both participated in HackTX 2022 and utilized this methodology to filter out content. As our project was a decentralized search engine, we prioritized user privacy and safety and decided that using a dictionary to filter inappropriate content would be one way to do this.

However, there are some limitations to a solution like this. As language evolves and new terms are used, the dictionary will also need to adapt to these changes. However, this becomes incredibly tedious and difficult to do manually for enormous amounts of words. Not to mention that the dictionary also has to recognize which words are considered inappropriate and which ones are permitted. Despite the challenges that exist with doing it manually, Machine learning and Artificial Intelligence can provide a solution to this issue. One of the methods of doing this involves having an artificial neural network that will undergo "training through exposure to real-world content, both appropriate and inappropriate" (Catrinescu). With huge amounts of

training data, the AI will be able to detect any toxicity using information about the amount of text and other patterns (Catrinescu). In this way, the AI has clear upgrades over the dictionary in that it can improve continually and handle more subtle inappropriate language. While a content-filtering system will never be perfect and there will always exist content that bypasses the filters, dictionaries have nevertheless proven to be an efficient and simple means of achieving some semblance of moderation.

Pair Programming

- 10/18: 2 hours of A driving, 2 hours of B driving
- 10/20: 2 hours of A driving, 2 hours of B driving
- 10/21: 1 hour of A driving, 1 hour of B driving
- 10/25: 1 hour of peer review work together
- 10/26: 1 hour of peer review work together
- 10/27: 1 hour of A driving, 1 hour of B driving
- 10/28: 2 hours of A driving, 4 hours of B driving
- 10/29: 7 hours of both working on separate topics

We really enjoyed working together on the last three labs, so it's a little sad we can't pair programs anymore. Overall, pair programming was a great experience that we both learned a lot from.

Works Cited

Catrinescu, Vlad. “Machine Learning for Content Filtering – Winning the Battle against Harassment and Trolling.” *Cognillo*, 30 January 2020, https://www.cognillo.com/blog/machine-learning-for-content-filtering-winning-the-battle-against-harassment-and-trolling/. Accessed 29 October 2022.

Parker, James. “Bad Words List and Page Moderation Words List for Facebook.” *Free Web Headers*, https://www.freewebheaders.com/bad-words-list-and-page-moderation-words-list-for-facebook/. Accessed 29 October 2022.