***Case Study and Lab:* Boggle**

*Final Specification*

By Jessica Peng, Alanna Zhou, Anvitha Aluri

**Overview**

This case study considers the plan and implementation of a computerized Boggle application. Boggle, a board game of alphabetical letters, is played by users with the objective of connecting the letters and finding as many words as possible. In the physical board game, letters in the English alphabet are randomly shaken to fall upon a grid. Players then write down a list of words that the letters strung together could form, and scores are given among players based on the complexity of the words found.

**Rules of Boggle**

* A player shakes a 4 by 4 grid of 16 cubic dice, with 4 faces each having a letter
* The player searches for words that can be made from the letters of sequentially adjacent or neighboring cubes within 3 minutes
* Words must have at least 3 letters
* One letter cube cannot be used more than once in a word
* Single and plural forms of words are counted as separate words

In the past few years, the age of gaming has brought a graphical user interface of Boggle into mobile and web applications.

In this project we will program our own digitized Boggle game based on Duke University’s Boggle project outline: <https://www.cs.duke.edu/courses/compsci201/fall12/assignments/boggle/>

Having a computerized Boggle allows the player to play against a computer machine that can find all possible words, unlike the physical board game where it would be very difficult to find all the possible words to see how the player compares. Essentially, it can serve as a more interesting approach to mastering the game as the computer can generate more words than any human could possibly generate efficiently.

The documentation can be found in the Boggle project package as “docs.”

**Structural Design**

**Interfaces**

|  |  |
| --- | --- |
| **Interfaces** | **About** |
| IAutoPlayer.java | Describes a component that are required of a computer player. The class outlines the necessary and to be implemented method findAllValidWords that is able to find all of the valid words on a Boggle board. |
| IBoardMaker.java | Describes a component that are required of a computer player. The class outlines the necessary and to be implemented method makeBoard that is able to generate a board of randomized letter cubes. |
| ILexicon.java | The specification for a Lexicon, which makes sure that a word found on a Boggle board is indeed a valid word in the English language. This class requires that Lexicon be a class that can transform a text file into a traversable data structure. |
| IPlayer.java | Describes a human player, by specifying certain components like score and name getters, etc. |
| IPlayerView.java | Outlines what a human player should see during a game of Boggle. It requires the methods showWord and showError. |
| IWordOnBoardFinder.java | Outlines what a word finder must have in order to find words on a board; it requires a cellsForWord method that returns a list of BoardCell objects that correspond to each letter of the word to be confirmed on the board. |

**Abstract Classes**

|  |  |
| --- | --- |
| **Abstract Class** | **About** |
| AbstractAutoPlayer.java | Describes a general computer player to be implemented - the backbone of the AI. Implements IAutoPlayer. Adds on a getter method for a player’s name. |
| AbstractPlayer.java | Describes a general human player to be implemented. The class outlines the need for the list of found words, a score keeper (BoggleScore), and a score. |

**Classes**

|  |  |
| --- | --- |
| **Computer Player Classes** | **About** |
| BadAutoPlayer.java | An auto player that does nothing so that the project can compile until the BoardFirstAutoPlayer class is written. |
| LexiconFirstAutoPlayer.java | A computer player that traverses a lexicon and checks if each word in the lexicon is able to pass any word finder class’ cellsForWord method. |
| BoardFirstAutoPlayer.java | The best computer player that finds words on a board efficiently by using recursion. |

|  |  |
| --- | --- |
| **Word Finder Classes** | **About** |
| BadWordOnBoardFinder.java | A word finder that returns nothing so that the project can compile until the GoodWordOnBoardFinder class is written. |
| GoodWordOnBoardFinder.java | Confirms that a given word found by a human player or a computer player is actually on the Boggle board being played. It returns the list of BoardCell objects that identifies where the word has been found on the Boggle board. |

|  |  |
| --- | --- |
| **Boggle Board Classes** | **About** |
| BoggleBoard.java | Describes a boggle board by relating the “faces,” or letters, to a given row and column on the board (which is essentially an array). The size can be specified, but the board must be a square. |
| BoggleBoardFactory.java | Describes how a randomized boggle board is generated. It ensures that every cube on the boggle board has been randomly chosen through Cube, and randomly placed through its own class. |
| BoggleMain.java | Runs the Boggle game in its single main method. It initializes all the components of the game by choosing a word finder, a computer player, and a lexicon to construct. |
| BoggleScore.java | Has a sole method that returns a score given the word on the Boggle board. The score calculator is generous to longer words. |
| Cube.java | Part of a BoggleBoardFactory implementation, which is able to generate random faces for the board. |
| BoggleGUI.java | Describes the Graphical User Interface of the game. |

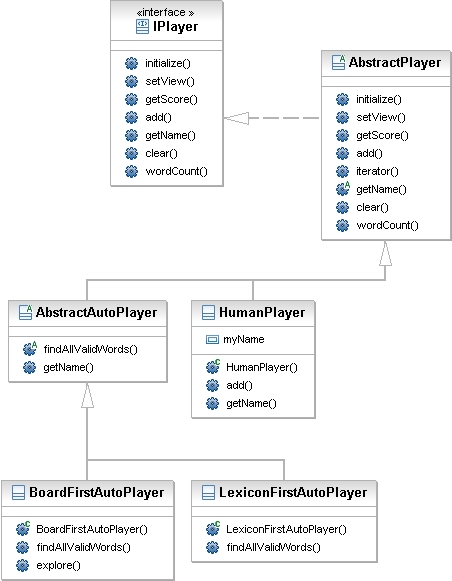
|  |  |
| --- | --- |
| **Lexicon Classes** | **About** |
| SimpleLexicon.java | Traverses a text file to see if the word exists. |
| TrieLexicon.java | Goes beyond SimpleLexicon by traversing a text file using a Map<Character, Node> so that it can find a word quicker. |
| BinarySearchLexicon.java | Uses a binary tree to manage the text file so that a word can be found using a Binary Search. |
| CompressedTrieLexicon.java | Uses logic on how certain letters in a word are grouped together to compress the TrieLexicon’s nodes (which goes from including just one letter to groups of letters). This saves space. |

**Testing**

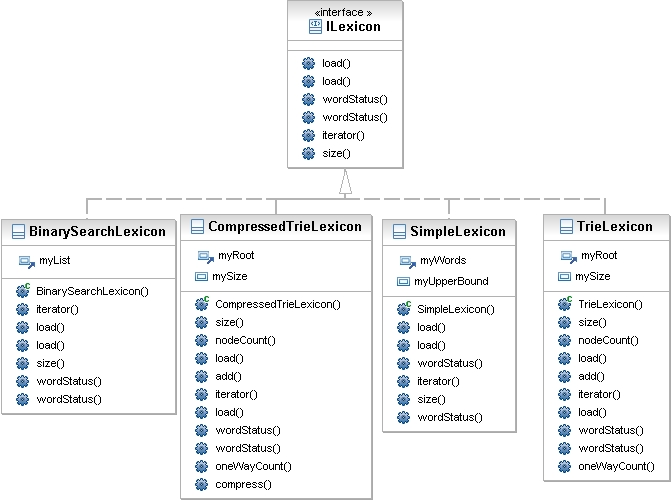
|  |  |
| --- | --- |
| **Tests** | **About** |
| JUnitTestAutoPlayer.java | Ensures that the computer autoplayer finds corner words, non corner words, doesn’t identify words not on the board, and doesn’t find words with duplicate cells. |
| JUnitTestLexicon.java | Constructs a lexicon (simple, binary, trie, compressedtrie) and checks for its correct implementation and whether it returns the correct word status (word, not a word, or prefix). |
| JUnitTestWordFinder.java | Ensures that corner words are found and repeats are not considered found. |
| LexiconBenchmark.java | Runs a given lexicon for statistical purposes to show which lexicon is the most efficient, specifically which is the fastest and the smallest. |
| BoggleStats.java | Runs a given lexicon and computer auto player for statistical purposes to show which combination of the two is the most efficient. |

**Object Oriented Design**

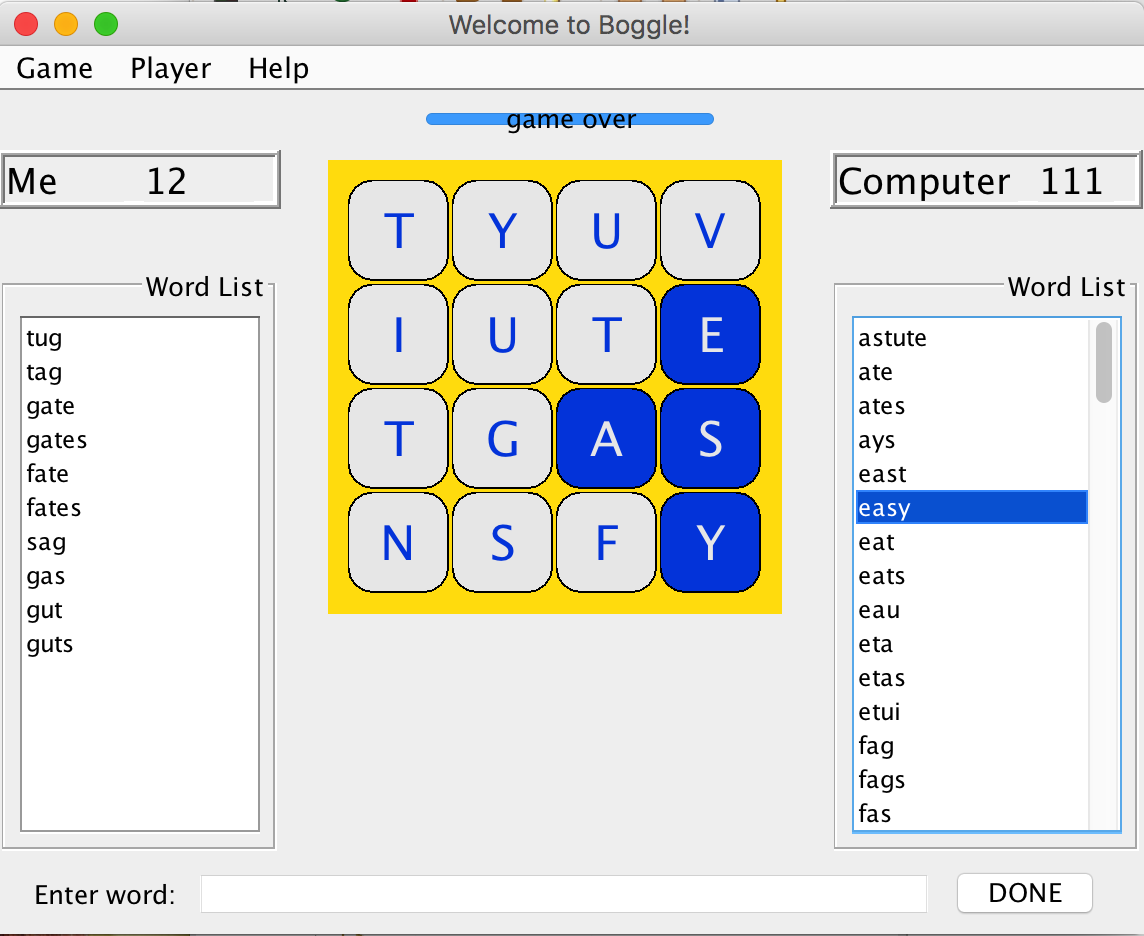
**Player Classes**



**Lexicon Classes**



**GUI Design**



**Visual of Boggle’s Graphical User Interface**

Above is what the general Boggle game looks like. The design is simple, user-friendly, and intuitive to use. “Enter word:” at the bottom signifies that the right text field is where the user should enter their list of words. The “Done” button automatically ends the game and shows the list of words that the computer player found.

**Player Panel**

The left side of the screen shows the player’s name, player’s score, and the player’s list of words found. As the player plays, the score increments based on the length of the word found. Words are added to the list only when the GoodWordOnBoardFinder class returns the list of BoardCells that confirms the word indeed exists on the board.

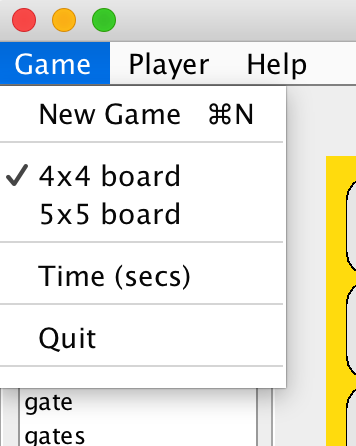
**Computer Panel**

The right side of the screen shows the computer player’s score and the computer player’s list of words found, which uses the class BoardFirstAutoPlayer to find words on the board and combine it with the functionality of a lexicon to ensure that the word exists in the English dictionary.

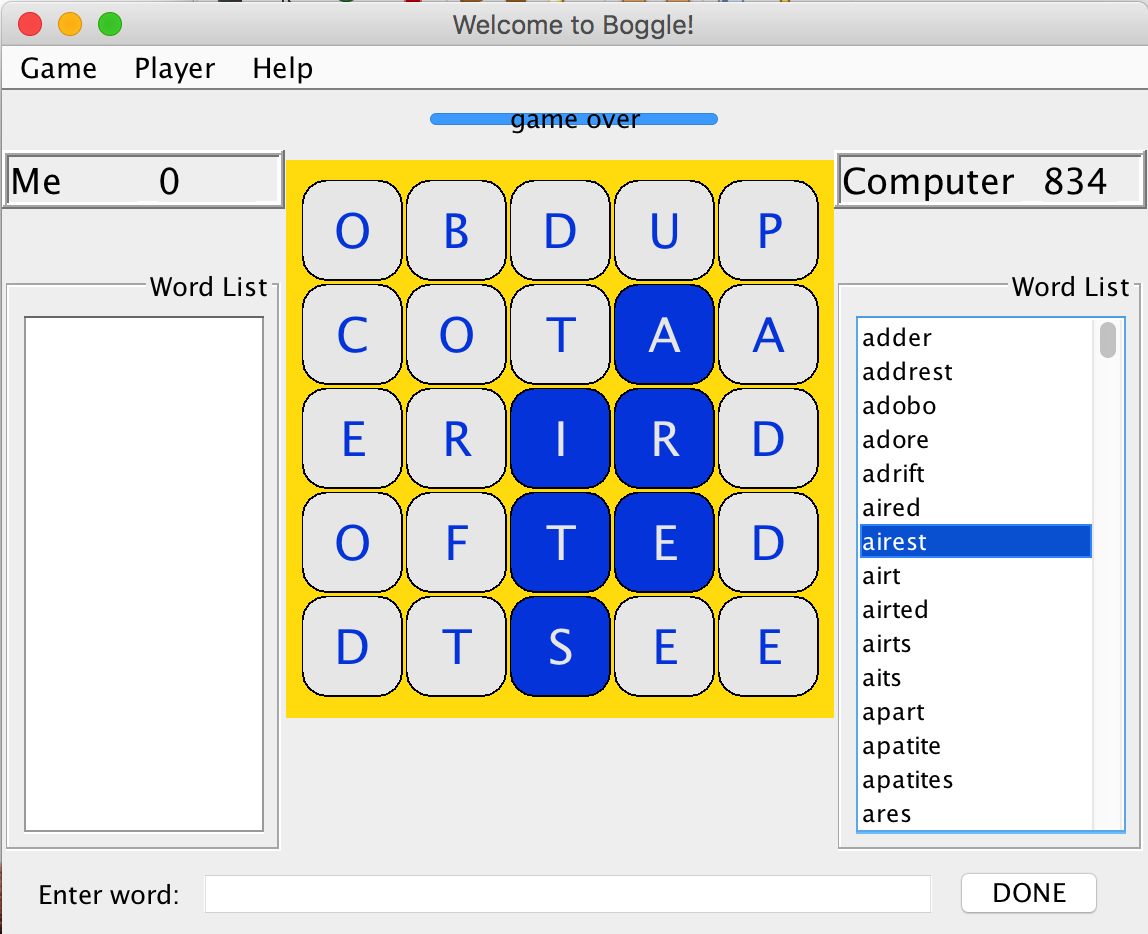
**Menu Bar**

The top of the screen is the menu bar, where the user can do many things:

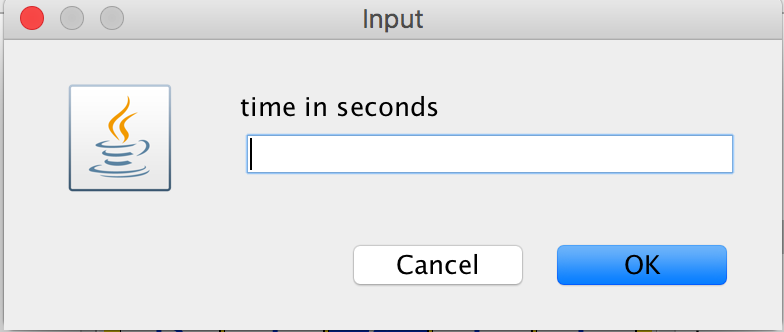
**Game**



* New Game allows the player to start over again, but with a completely newly randomized Boggle Board
* 4x4 and 5x5 board allows the player to specify the dimensions of the board to be played. A 5x5 board for example, would look like this:

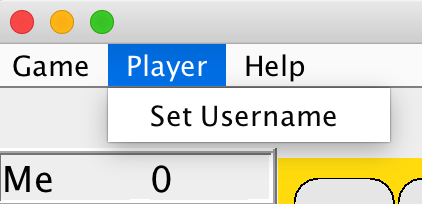


* Time (secs) allows the player to set the timer of the game. The default time for the user to complete the game is 60 seconds.

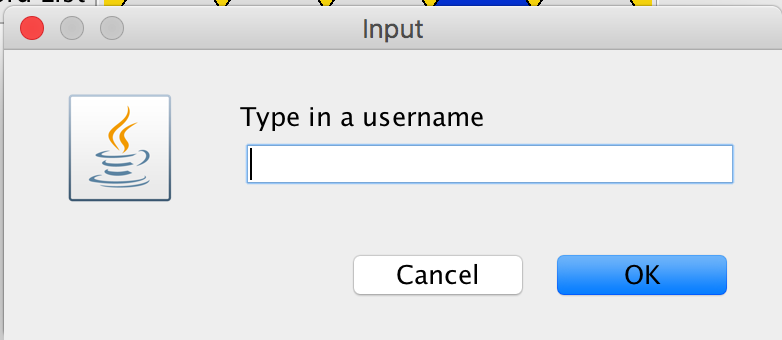


* Quit allows the user to quit the application.

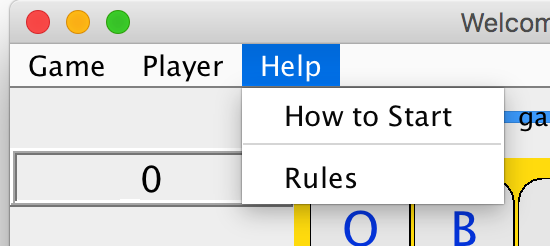
**Player**



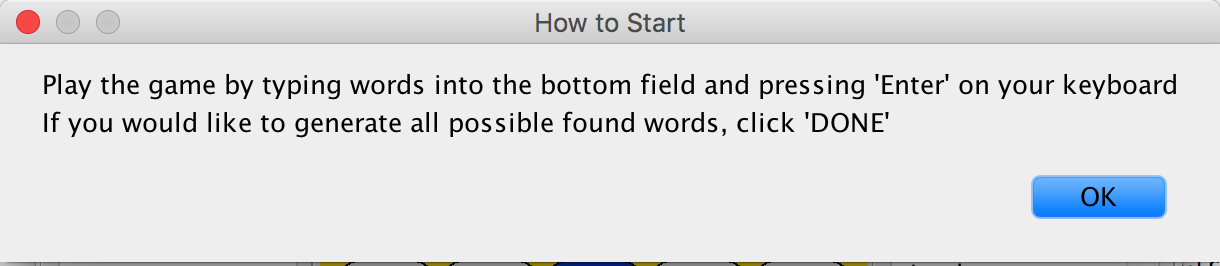
* Set Username allows the player to change the username from the default “Me” to a desire name.



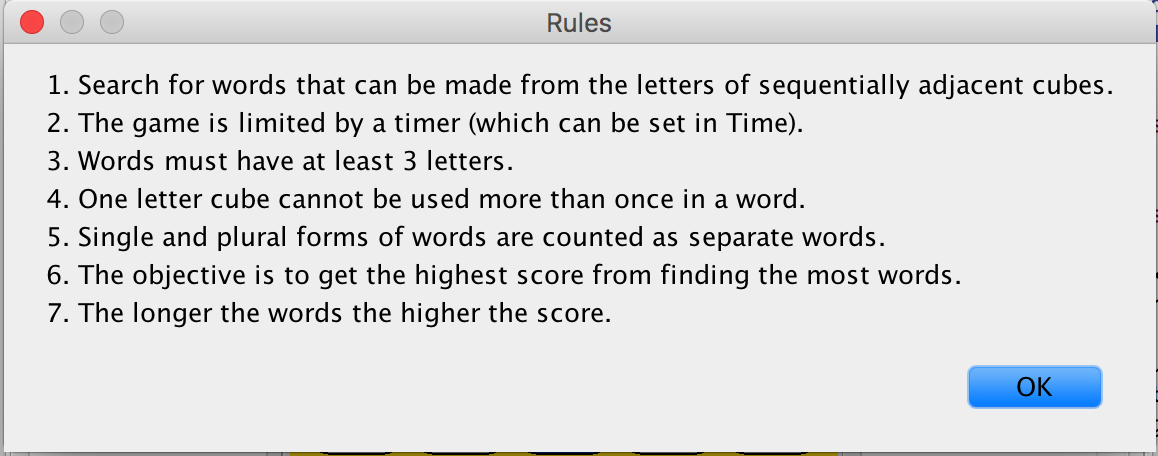
**Help**



* How to Start tells the user how to initially use the Boggle Game interface



* Rules tells the user the rules of the Boggle game

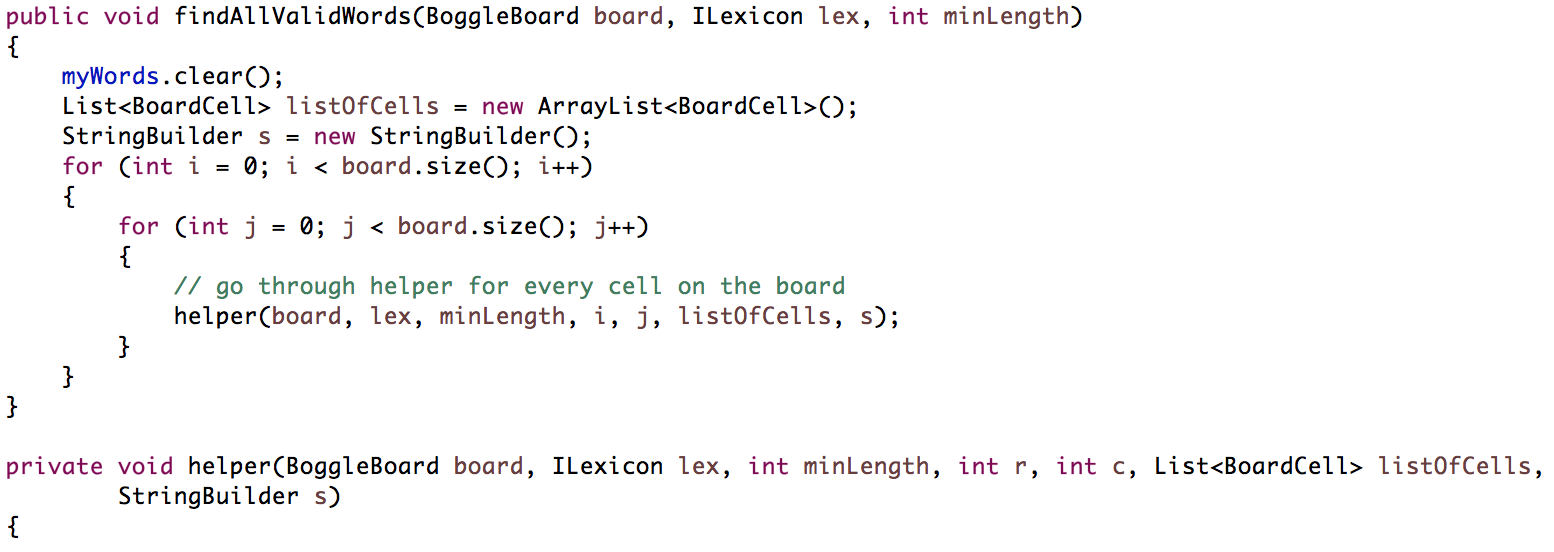


**Detailed Design**

**Auto Player**

The BoardFirstAutoPlayer is designed to find all the potential words on a board. In this class, there is a recursive helper method that finds all the words at a specified row and column. When first called, the string built from the search so far is the empty string: "". The current cube (known as a BoardCell object) on the board, if legal and not used in the search so far, is added to the end of the string built so far. If the string is a word, the word is added to the collection of found words by calling the inherited add(..) method. If the string is either a word or the prefix of a word in the lexicon then the search is continued by calling the helper method for each adjacent cube with the string built so far. If the string is not a prefix (or a word) then the search is cut-off at this point and the recursion will backtrack (essentially to the point where the last possible word/prefix was formed). Because of the backtracking code, the code must not reuse a board cell once it has been used in the current search.

Here is an example of how the code and the helper method can be implemented:



**Word Finder**

The GoodWordOnBoardFinder correctly finds where a given word is on the board. We implement and write the helper method cellsForWord, which uses recursive backtracking search to find the word. The Boggle board and word to be found is passed into cellsForWord for the method to return a list of BoardCell objects that notes where each letter of the word is on the Boggle board. cellsForWord starts off the recursion, and the helper method that it calls (named helper) continues the backtracking recursion. helper ensures that recursion stops based on a series of conditions:

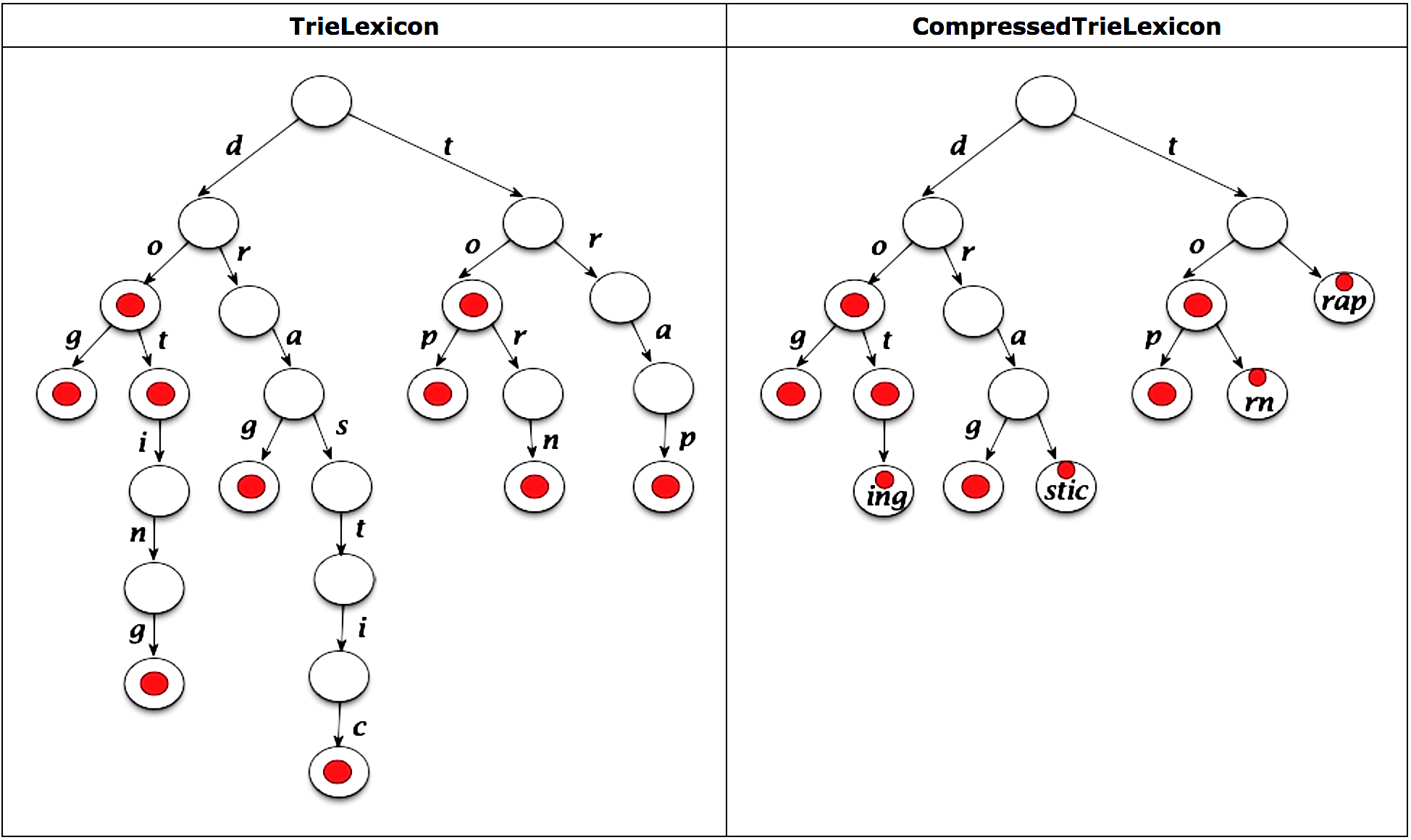
* if the index of the word is greater than the word’s length, return true because it means that the word has already been found
* if the row or col to be checked is out of bounds, return false
* if the list of BoardCell objects to be returned already has the BoardCell that was found to be matched with the letter at the index given of the word, return false

These conditions ensure that the list of BoardCell objects returned are valid for the GUI to highlight the correct cells on the board based on the word found by the player or the computer auto player. This system is also the basis for how the computer auto player detects words on the board (which must pass through lexicon), because once the computer finds the words, this method is the one that gives the GUI the list of BoardCell objects to highlight for the user to see.

**Lexicons**

In order to confirm that a word is a valid word, a lexicon is used. It is a list of words, similar to a dictionary except without their respective definitions. Essentially the lexicon object is cross checking the found word with a text file to ensure that not just any word is able to score the player points. There are multiple implementations of the Lexicon interface, and our Boggle game happens to use the CompressedTrieLexicon. We discuss BinarySearchLexicon in addition to demonstrate that there is an alternative to the CompressedTrieLexicon that can be used to identify words.

The CompressedTrieLexicon implements a lexicon based on a *compressed trie* data structure. In the class, we remove nodes with only one child, effectively compressing the tree. To do this, a compress method inside CompressedTrieLexicon with a helper recursive compress method runs through all the nodes by going from parent node to child nodes contained in a map of the parent node. It then determines the last node, or leaf of a branch and adds it to an arraylist of nodes. Then, another loops is run through this arraylist which checks each of these leaf nodes and goes up from there to begin compressing all nodes with only one child. The nodes are compressed by taking the information, or characters, out from the child node and adding it to the parent node. The CompressedTrieLexicon is both time and space efficient than the TrieLexicon. The result is shown in the picture below. The red dots indicate the path from root to node is a word.



**Diagram of *TrieLexicon* vs. correctly implemented *CompressedTrieLexicon***

Another type of Lexicon that can be used as opposed to CompressedTrieLexicon is the BinaryTreeLexicon. The BinarySearchLexicon also implements the ILexicon, but does its own take by using a Binary Search to confirm that words found by a player or a computer player are indeed valid words from the dictionary. The Binary Search is recursive and is used in conjuction with word finder classes, such as GoodWordOnBoardFinder to ensure that although the word may exist on the board, the word must also be found in the text files of words. That is how we confirm that the words are valid words - using a text file, similar to a dictionary but without word definitions.

When the board is searching for a word entered by the user, a recursive algorithm is run through to find if it is a real word. In this case, the WordStatus method is very important. If the letters enter form a real word, LexStatus.WORD is returned. If it is not a word, a method call is first initiated to determine if it could possibly be a prefix. If it is a prefix,LexStatus.Prefix is returned. If it is not a word or a prefix, then LexStatus.NOTWORD is returned. In a way, these status variables returned are a type of boolean that allows the board to return whether what the user entered was a real word or not.