

Matthew deWet and Christopher Wiley  
HWK 5  
October 31, 2006

#### Overview:

This program is an exercise in applying specialized data structures in order to handle a large amount of data. The data structure we used to store the data is supposed to make search operations as quick as possible while taking up a reasonable amount of space. Another point of this assignment was to use recursion to search both the data structure we choose and the board. A smaller point was to minimize the number of recursive calls by trimming the recursive tree. Oh, and the situation we're writing the logic for is supposed to be a boggle game. Unfortunately, the specification was so vague and/or incorrect about the rules of boggle that it's clear the actual emphasis of the assignment was on the implementation of the data structure and the search algorithms.

#### Implementation (Game Dictionary):

The handout almost said out loud that the data structure was supposed to be some form of a tree. In order to get  $O(P)$  for the `isPrefix` method, we decided to store the words in a tree of characters. This allows us to take advantage of the massive amount of overlap in between words. In fact, the maximum depth of our tree is always the maximum length of the tree. It's interesting to note that since the length of prefix is almost always some trivially small number, and that number is always independent of  $N$ , that `isPrefix` can be said to run in  $O(1)$  time. At least it brings a warm glow to our hearts.

Specifically, each node in our tree has the value of the character it represents, a reference to its parent, and an array of twenty-six references to its children. Naturally, each of those nodes corresponds to a letter in the alphabet. Each node also has a boolean variable, indicating whether that node represents an ending of a word. In addition, to minimize the number of times we have to search to see if a node has children, we store a boolean indicating whether it has at least one child.

On reflection, it may be the case that actually storing the value of the character the node represents is a waste of space, because each node's value is also established by its position in the reference array of its parent. We may be able to take advantage of the fact that the parent actually therefore knows the value of its child. Unfortunately, to simplify our code, we do not take advantage of this fact.

The method `isPrefix`, in our implementation, just returns the boolean result of a private method `prefixSearch`. This helper method just goes down the tree until it runs out of characters and has found a node with children, indicating one or more words has the characters above the current node as a prefix. If the recursion runs into a null somewhere along the path, it knows it has found an invalid prefix. It's important to note, that a word is not actually a prefix. For instance, if the dictionary included only the word "tuberculosis," "tuberculosi" is a prefix, whereas "tubercolosis" is not a prefix. Thus there are valid words which are not prefixes.

The public method `nextWord` uses `findNext` to find the next node in alphabetical order which is a word ending. The method starts at a word ending and checks four cases.

Either the current node:

Is null, because the whole tree was traversed and there are no more endings. (return null)

Has a child before all the others which is an ending. (return the child)

Has children it hasn't visited who are not endings. (return itself on the leftmost child)

Has no children, (return itself on the parent with an indication that it has been visited)

It is unclear from the specification if we are to reset the dictionary iterator ourselves to begin with or not. Since our reset function is an  $O(1)$  operation, we went ahead and did it ourselves. It cannot hurt for the user to do it again, and if they forget, its all set up for them.

#### Dictionary Analysis:

As noted on the wiki, our tree does compress the dictionary at the cost of an obscene number of objects. Specifically, it has been found that the dictionary and subsections of the dictionary obey the general rule that there are around twice as many nodes in the tree as there are words in the dictionary. As we have yet to run into problems with memory it appears this is acceptable. Analysis of actual memory usage appears to be somewhat futile, since we use 17 object references and 2 booleans, neither of which has a clear size in bits. (And yes, the boolean, despite its intuitive 1 bit size, is actually implementation specific according to JAVA certification test prep books.) Instinctively though, its fair to say we use probably more memory per node. So in reality, we're using far more than twice as much memory than just storing the Strings. This sounds absolutely ridiculous except that its very conducive to having terrific run times and it hasn't hurt anyone to date.

#### More analysis (Game Dictionary):

The advantage of using a tree of characters to represent words is that there is actually no searching done on the tree. It is very easy to determine, given a word, whether or not the word is on the tree or not. Lookup is a very dumb sort of operation which takes advantage of the fact that all words are assumed to be lower case and comprised of the alphabetic characters a-z. The only comparisons done are to determine whether or not the needed next node is null or not. Another advantage is that the tree is by definition always as balanced as it could be. The root has no input on the balance of the tree. The disadvantage is that a large number of very small objects are used to represent this structure. Depending on the VM, this gigantic group of objects could be very incredibly larger than a simple binary tree of Strings. Actually traversing the tree is actually no longer an  $O(1)$  operation however, because instead of just going to the next left child, the program has to traverse over a part of the tree. The relative importance of isPrefix to the users of the dictionary made our implementation more excellent.

#### Implementation (Game Manager):

The most difficult part of the implementation of the GameManager class was understanding just barely documented design of the interface. The comments actually proved to be more helpful than the handout, and the comments didn't actually describe boggle, but rather some very loose guidelines which could be construed to be boggle. As such, we took some liberties in our implementation.

For one, we keep track of scores, playing status, and words picked in a BogglePlayer class. This class does not specify which words are valid and which are not, but does control how much a word is worth. This is a bit confusing, but convenient. For better results, document better. This particular implementation admittedly does not follow good OO conventions very well, but is convenient.

The big calls in this part of the assignment were the recursive calls to search the board and how those calls were trimmed down. In general, it made sense to try every possible test to test word validity before calling a

recursive board search, because the board search had loosely exponential time in proportion to the size of the board, whereas every other check was independent of both the size of the dictionary and of the board.

There are essentially two type of board searches. One starts with a string, and returns where that string can be found in the board, and the other returns all valid words in the board. Human players use the former and computers use the latter.

The search which starts with the String and works toward the path iterates over the board and calls a recursive method at each position.

The method has four cases.

The letter at the current index does not match the letter on the board, return null

The letter matches, and the word is done, return a list containing the current point

One of the calls to around the current position returns a list of points, add my point

No paths go through this point, return null

There is a special case that if a point is visited twice, it is not a valid path, we catch this.

The search for valid words can be done two ways. Either you take all possible words on the board and check them against the dictionary, or you take the entire dictionary and check it against the board.

The board check takes quite a bit less time due to our highly excellent trimming of the recursive tree. The dictionary search, while still not taking an incredibly great ammount of time, tends to be slower by a factor of around 7 for a 4 by 4 board with words.txt.

This constant falls with an increasing board size, but even with a 16 by 16 board, the dictionary search runs slower. The reason for this is that no matter how well the dictionary search is trimmed, eventually, the board has to be searched, and then the 16 by 16 board will eat up time like having two girlfriends and an addiction to World of Warcraft. This is a major reason why the videogamer clique doesn't tend to intersect with the player clique. I would recommend at a maximum perhaps a game like Starcraft and one reliable partner.

The board search is also faster because of our ungodly excellent isPrefix method. This trims the number of calls by an inordinately amazing number.

Theoretically, with a large enough board or a small enough dictionary, the dictionary search would be the optimum solution. Reallistically, with a decent dictionary and a 4 by 4 board, the board search wins hands down. This is because,  $N$ , the number of words in the dictionary is large enough that the  $O(N*9^L)$  time of the dictionary search ( $L$  being the average length of the words) tends to be bigger than the  $O(D^2*9^L*L)$  time of the board search ( $D$  being the length of one side of the board). These non  $N$  variables tend to be very very small, very often in the single digits.

Assumptions:

We assumed that once one player adds a word, no other players can add it.

We assumed a number of things when initiallizing the board.

--each cube row must have exactly 6 characters

--there will be no spaces

--uppercase/lowercase issues are irrelevant

--there will be only alphabet characters

--there must be at least as many cubes as spots in the board

--extra cubes are ok

We assumed its ok to reset the iterator when we load the dictionary.

Since there was no mention of throwing errors in the sketchy comments, we don't throw any. In a couple places, the directions just say to return if there is an error. This tends to promote strange behavior when errors are thrown further down in the program, so to compromise, we print an error message to System.err but do not throw an error. Our greatest desire is for a project to actually explain what to do when invalid input is found.

#### Testing:

Once again, we used a highly informal testing method which could be described as white box. We tend to look for areas in our code which could potentially screw up, and use System.out's to check their status. It also is very helpful to comment afterward, which forces us to go over the code one more time and check for weird things. A lot of encapsulation breaks were caught in this way. To test the most complicated things, we used cases where we knew the answers, such as very small grids or grids with all z's and one word to test our ability to handle certain cases.

#### We checked:

- duplicate words are ignored
- all words in the file can be found in the dictionary
- boardSearches always are complete
- there is not clear and easy way to get a non square board into the game
- both searches return the same stuff
- numerous small tests which revealed errors in our logic, but can't be remembered

#### Code Not Our Own:

We used all the premade interfaces, data, and any other information that was given to us that we forgot. Arguably, this is a nearly infinite list since there is very little original thought left in the world.

#### Collaboration:

Wiley gave in to temptation and talked about what kind of tree was to be used in the project with Forrest. Forrest at that point had already implemented his tree and everything was excellent because we had already implemented ours as well. Wiley is still waiting to see if the shadowy forces of the draconian standard of scholastic plagiarism come out and bite him. It is yet unclear to Wiley what exactly is not forbidden.

Log: We each worked about eight to ten hours. We tend to share the keyboard fairly equally although we take long turns at it. We tend to program in multiple short burts, and talk about how we're going to do something before we undertake it. Matt has a supply of small white boards which we use to excellent effect when planning algorithms.