**Cracking the Crossword**

**Problem Statement**

Crossword puzzles are a whimsical pastime which many enjoy due to their challenging nature. While solving crosswords is great fun for crossword enthusiasts, the process of generating them algorithmically is a similarly interesting and perplexing conundrum for computer scientists. After all, someone needs to make the puzzles!

Here’s the challenge: given an N x N grid consisting minimally of blank tiles and possibly also containing unusable tiles and tiles which must contain a specific, predetermined character, how can we generate all valid crossword puzzle solutions which would match the board using English words found in the dictionary? Can this be accomplished in a reasonable amount of time, and if so, how can we do it? How can we represent the board, the dictionary, and the computation as efficiently as possible?

**Dictionary Setup**

A naïve approach to storing the dictionary is to simply maintain it as a sorted array or ArrayList. The MyDictionary class is a concrete implementation of this idea which we will consider in our analysis as a contrast to more sophisticated strategies. We will consider also the De La Brandais trie and the multiway trie as alternative implementations. I encapsulated all these choices into a single Crossword class to make it simple and easy to compare the options.

Crossword.java uses the first command line argument to decide how to store the dictionary information from “dict8.txt,” which contains an alphabetical listing of English dictionary words. If it is “DLB”, it makes a De La Brandais representation of the file, if it is “MWT”, it makes a multi-way trie representation, otherwise it defaults to using a MyDictionary object. Since my program calls dictionary methods through a DictInterface reference variable, it works the same way (though perhaps more or less efficiently) regardless of which data structure implementation the user chooses. Unfortunately, there are severe performance limitations inherent to the MyDictionary implementation. Once a solution to the crossword is found, the program uses the *instanceof* keyword to check which type of dictionary is being used and halt execution in the case of MyDictionary. This way, one program can be used to test all implementations of the dictionary in a reasonable amount of real time.

**Board Representation Setup**

How should we keep track of the state of the board as the crossword solutions are generated? It makes sense to keep a copy of the unaltered board as an instance variable to avoid retrieving information from a file over and over. Furthermore, we need some way to represent the words we are generating as solutions to the crossword so that they can be checked against the dictionary. Though it would be possible to do this with String objects, we incur a heavy price if we alter the Strings incrementally because String is an immutable class in Java. An entirely new object must be created to adjust a String, which is a needless waste of resources. How, then, can we track the board state in a sensible way?

The second command line argument to Crossword.java is the input file’s name. If the file exists, the program first reads in the size of the crossword puzzle board. Then, it reads the board characters into a 2D character array. I decided to make these both in the global scope to simplify the parameter lists of my program’s other methods. It stores a flag of whether it encounters any wall characters while building the board for use in a later optimization. In addition, it creates two N-length arrays of type StringBuilder—one for the board’s rows and one for its columns. Together they exist to hold my solutions as it recursively determines them character by character. Unlike the String class, StringBuilder can make this adjustment without making a new object instance. Each time a new character is added to the logical position at board[i][j], the StringBuilders can equivalently represent this by appending that character to the StringBuilder in the rows array at index i AND to the StringBuilder in the columns array at index j.

**Verifying a StringBuilder**

How can we determine if a given StringBuilder has hit a dead end? This problem is made more complex because of the existence of the wall tiles. Our method must be able to verify a row or column that represents more than one word or prefix at a time. While I could have chosen to store rows/columns with more than one part separately, I chose to keep them all together as one StringBuilder. Our method also needs to check if parts of strings are words and not just prefixes in positions other than just the ends of rows and columns. For boards containing no walls, there is no need for either consideration. We can simply check if we’ve found a word (not just a prefix) upon reaching the end of the row/column by testing against the global board size N.

For boards which do contain walls, the logic is slightly more complex. I used a loop to find the most recently added word or prefix (following a wall or board edge) and verify that just that piece was a valid word or prefix. To do this, I relied on the DictInterface’s searchPrefix method which takes a starting and ending index to parse out the parts of the StringBuilder I wanted to verify. There is no need to recheck earlier parts of the StringBuilder because they have already been verified if they do not belong to the current word. I also checked for full word status when I placed a wall after a non-wall character.

**The Backtracking Workhorse** The solveTile method is a recursive backtracking method which solves the board. It takes two parameters: the current row position and the current column position. It returns an integer value which allows the algorithm to communicate to prior calls that we wish to backtrack more than one tile.

Like any recursive algorithm, we first need at least one base case. The algorithm checks first if the current row equals N. If it does, the board is full of valid characters and I’ve located a complete a solution. The MyDictionary implementation ends execution after locating the first solution. If we’re using a DLB or MWT implementation of the DictInterface, the previous call will resume where it left off and continue to find any remaining solutions. When all options are exhausted, the recursion terminates.

Otherwise, there’s still work left to do. We check the next character on the 2D array board to see what path to follow. We’ll call this c. If c is a letter or a wall, we know that any solutions to the board must use c as is. Alternatively, if c is a plus sign, we can place any letter of the alphabet in this space. Since we will have to test every option to be sure our exhaustive search is correct, this is best accomplished by simply looping through all 26 options. I stored an array literal of the lowercase alphabet as a global variable to make this simpler. The procedure is the same as for walls and specified letters, except that we try 26 characters in sequence before giving up and returning instead of only one.

If we find a letter which works for both the current column and the current row, we have placed a character that might lead to a solution and so we recurse. Either we recurse down the row if there is room there, or else recurse to the first column of the next row. See the figure below for the order of tile visitation in a 4 x 4 board. The loop counter is stored as part of the activation record that the runtime stack provides us, so we can easily pick up where we left off if later recursive calls hit a dead end.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** |
| **0** | **1** | **2** | **3** | **4** |
| **1** | **5** | **6** | **7** | **8** |
| **2** | **9** | **10** | **11** | **12** |
| **3** | **13** | **14** | **15** | **16** |

We want to keep going forward as long as possible, but sometimes we reach an impasse. When all recursive calls finish and we are out of characters left to try, or if we can’t find a valid option for the current tile, we must backtrack. This is the pruning part of the algorithm where we peel off huge pieces of our exponential search space. It makes no sense to keep pushing blindly down a dead-end path. We remove c from the StringBuilders and return. Perhaps we can make progress by altering a prior choice.

How far back should we go? Ideally, we’d like to backtrack so long as we know no forward progress can be made to avoid wasting time, but we can’t do this unless we’re certain that it won’t make us miss any valid solutions. There is one scenario which permits us to backtrack an entire row at a time rather than just one tile. This occurs whenever no possible options for the current tile lead to a legitimate word/prefix for the current column. If the current tile breaks the column no matter what it is, we can revert to the previous row in the same column and save time. This is a lossless heuristic which can potentially trim a substantial number of execution paths, particularly for boards which include tiles which must be used as-is.

See the figures on the next page for an illustration. I have provided examples which showcase the algorithm in the middle of execution in situations which require backtracking. On the left, *none* of the possible 26 characters we can choose will make a valid 4-letter word for column 1. As a result, we can backtrack the entire row. Contrast this with the example on the right, where I have inserted an as-is tile “I” at position [2][3]. We clearly must backtrack here because DUMI is not a valid word. However, SAI is a valid prefix to a word this time (e.g., said) and so we’ll backtrack only one tile rather than the whole row.





How is the backtracking implemented in the code? If we want to backtrack to just the previous tile, we return 0. If we want to backtrack the whole row, we pass how many extra tiles to backtrack (calculated based on the board size). We check the recursive calls for nonzero return values. If we still have more tiles to backtrack, return a value one smaller to keep track of our progress. The final return value is 0, indicating we are returning normally and will continue looking for solutions.

**Asymptotic Runtime Analysis**

As the size of the board increases, the runtime for crossword generation skyrockets. How bad is the worst case? For a square grid of length N, there are N2 tiles on the board. For each tile, we’ve got to try potentially as many different options as there are letters in our alphabet. In our case this is 26, but let’s generalize and call this M. Without considering any dictionary implementations or lookups at all, we can see we have an exponential search space sized **M^(N2)**. We can decrease this search space considerably by implementing a good pruning heuristic, but this doesn’t change our overall asymptotic runtime. Still, this makes the difference between a problem I can solve today and a problem that would take months to solve, provided the problem size is not too large.

**MyDictionary**

We need to factor in the specific dictionary implementation to finish out the analysis. Let’s start with MyDictionary. The first thing we do is build a sorted ArrayList. This is something we only do once, so it’ll end up as a lower order term which is dwarfed by the exponential board generation problem and we’ll throw it out in our asymptotic analysis, even in a tilde approximation. What *will* matter to us is how the dictionary’s structure will affect the lookup performance, since we will utilize this functionality at least once per recursive call.

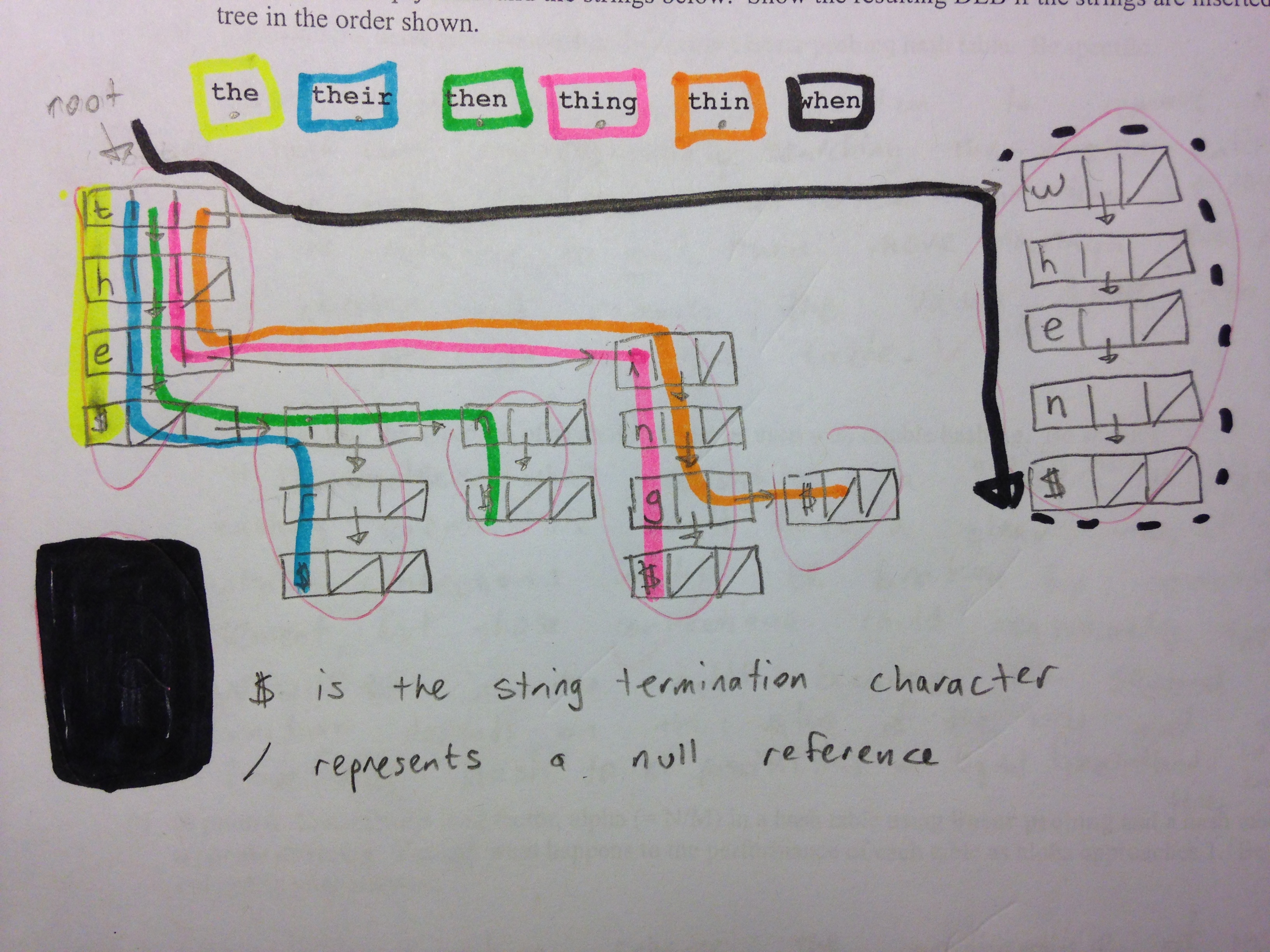
For every option for every square, we search the dictionary twice to verify the current row and the current column. In big O analysis we’ll drop the constant factor 2. The MyDictionary implementation performs a sequential search of the dictionary every time we call the searchPrefix method. Let’s call the size of the dictionary D. We know that the dictionary is sorted, which helps us quite a bit because we can stop earlier than the end of the list on search misses. This gives us a fractional coefficient for our linear search but will unfortunately still scale proportionally to D in the limit. There’s another concern here as well. Every time we check our word against the current dictionary value, we’re checking character by character because we need to assess if we have a valid prefix. Let’s call the number of characters in the key K. We know for sure that K <= N because it wouldn’t fit on the board otherwise, and we could feasibly restrict K to values below length of the longest word in the dictionary also depending on how we implement our query. We end up with a rough approximation which suggests that *each* search of the MyDictionary scales at Θ(KD). That’s a substantial factor to multiply with our already huge runtime. Alas, we end up with a final runtime for the crossword algorithm using the MyDictionary of:

**Θ (KD \* (M)^(N2))**

**The DLB**

How does the DLB compare? The De La Brandais trie stores each character of each key in a discrete node. Each node has a single pointer to its siblings, which represent the other possibilities for the letters in this position, and to its children, which represent the possible suffixes we can build with the current prefix.

Below is an illustration of a DLB structure which was created by adding the words the, their, then, thing, thin, and when, in that order. Each rectangle represents a node in the DLB. The leftmost field in the rectangle is the letter the node is storing, the middle box represents the child reference, and the rightmost box represents the sibling reference. Each color corresponds to a word and showcases the path representing that word’s representation in the DLB.



We perform searchPrefix by sequential searching through all the siblings at the current level to locate each character of the key. If that character is not present at the current level, the key is not found in the dictionary (and there are no words with it as a prefix, either). If it is found, we proceed down that node’s child pointer and check the next character. If we exhaust all the characters in the key, we check for a special string terminator which signifies that the key is a real word in the dictionary. Typically, this is a node with a sentinel character (shown as ‘$’ above) or a boolean attribute field in the node. If the string terminator is not found, the key is a prefix but not a valid word. If we find the string terminator, we have found a valid word. We test to see if there are any words beyond this as well (which would vary in detail slightly depending on how we handle the terminator character). If there are paths past the termination of the word, we indicate that we’ve found a word AND prefix, otherwise we indicate that this is a valid word but not a valid prefix. Search misses are resolved faster in the DLB than hits, because we don’t need to traverse all the levels of the tree.

What does all of this buy us in runtime? Building the DLB is going to get left out in the asymptotic analysis because it’s a one-time Θ(MK) process. The search can proceed through as many levels as characters in the longest possible key, which we’re calling K. We also might take as long as M steps at each level because of our sequential search through each possible character of the alphabet. That would make searching the DLB Θ(KM). However, we probably wouldn’t use a DLB trie unless our data is very sparse over the set of possible keys, as is the case with the dictionary words in the key space of all possible permutations of K characters. We can probably safely assume that search is roughly Θ(K) *as it pertains to this problem*, much like the multiway trie, because most levels won’t contain anywhere near M options. This is a major improvement for us because **the size of the dictionary doesn’t matter**. The longest word in English is only 42 characters long, so Θ(K) is *way* better for us than Θ(KD) for the same task. That gives us an overall result for the crossword algorithm using the DLB of:

**Θ (K \* (M)^(N2))**

This is still a terrible runtime, but at least we haven’t artificially inflated it with bad design decisions. Large values of N will still make the problem intractable. Even so, it’s a very significant savings over the other implementation because the size of the dictionary is considerable. Our dictionary contains 17,271 words. Many real dictionaries contain over 100,000 words, which would further increase the edge a trie provides. My empirical analysis shows that the DLB isn’t 17000x faster in practice, but we can see places where it’s over 1000x more performant. 1000x faster is a big enough difference that I could feasibly complete some of the test cases with the DLB that I wouldn’t bother to complete with MyDictionary because there’s just not enough real-world time to wait for it to finish. One possible reason the DLB isn’t fully D times faster than the MyDictionary is that I’m dropping a scalar which is larger than one from the DLB runtime and a fractional constant from the MyDictionary runtime. It may also reflect that linked structures (like the DLB) are not as performant as contiguous structures (like the MyDictionary) on many real architectures because they are not cache-friendly. Lastly, for boards with many solutions, we may not get as bad of performance as we would expect with the MyDictionary when we are only interested in finding the first solution to the board because many of the words we are searching for appear very early in the dictionary.

**The Multiway Trie**

As we discussed in lecture, the advantage of using a de la Brandais tree over a traditional multiway trie is that it saves a substantial amount of memory if the tree is sufficiently sparse. We ceded that it came at a cost of speed. I was curious how much faster tailoring a multiway trie to the DictInterface would be compared to the DLB, so I implemented it in MWT.java. My implementation uses a private inner class MWNode which stores two boolean values, whether the current node terminates a valid word and whether the current node has children. The latter is set as the trie is built, and I included it to prevent checking all the child pointers every time I want determine if a word is a valid prefix. The child pointers are stored as an array of size 26 because these are the only values we’re interested in for this application.

The MWT is clearly faster than the DLB, as my test data shows. It completed the test files roughly two or three times faster than the DLB, on average. The MWT is faster because the sequential searches through the siblings at each character position in the DLB cost us time, whereas the MWT has random access to its children. Since we’re dealing with exponential runtimes here, two or three times faster probably matters to us a lot. The price we pay here is that the MWT is extremely wasteful with memory. That said, we can probably afford to pay this price on a modern machine for this application. Though the English dictionary is large, its size is bounded for practical purposes.

Algorithm Performance

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TEST FILE | MyDictionary | DLB (first solution) | DLB (all solutions) | MWT (first solution) | MWT (all solutions) |
| 3A | <1 second | <1 second | 2 seconds | <1 second | 1 second |
| 3B | <1 second | <1 second | <1 second | <1 second | <1 second |
| 4A | 1 second | <1 second | 7 minutes 12 seconds | <1 second | 2 minutes, 38 seconds |
| 4B | 1 second | <1 second | 30 minutes | <1 second | 14 minutes 38 seconds |
| 4C | <1 second | <1 second | 2 seconds | <1 second | 1 second |
| 4D\* | <1 second | <1 second | <1 second | <1 second | <1 second |
| 4E\* | 1 hour 59 minutes | 2 seconds | 2 seconds | 1 second | 1 second |
| 4F | 1 second | <1 second | <1 second | <1 second | <1 second |
| 5A | 5 seconds | <1 second | 2 hours 43 minutes | <1 second | 1 hour 4 minutes |
| 6A | too long to test empirically by deadline | 2 minutes 56 seconds | 2 hours 37 minutes | 52 seconds | 1 hour 9 minutes |
| 6B | 2 minutes 32 seconds | <1 second | too long to test empirically by deadline | <1 second | too long to test empirically by deadline |
| 6C\* | too long to test empirically by deadline | 16 seconds | 16 seconds | 5 seconds | 5 seconds |
| 7A\* | too long to test empirically by deadline | 1 hour 55 minutes | 1 hour 55 minutes | 44 minutes | 44 minutes |
| 8A\* | too long to test empirically by deadline | 1hour 3 minutes | 1 hour 3 minutes | 25 minutes | 25 minutes |
| 8B | too long to test empirically by deadline | 46 minutes and 42 seconds | too long to test empirically by deadline | 29 minutes | too long to test empirically by deadline |
| 8C | 1 minute 13 seconds | <1 second | <1 second | <1 second | <1 second |

\* indicates a board with no solutions

\*\* these files were timed with the *time* bash utility