**Cracking the Crossword**

**Dictionary Setup**

Crossword.java uses the first command line argument to decide how to store the dictionary information from “dict8.txt.” If it is “DLB”, it makes a De La Brandais representation of the file, 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. Once a solution to the crossword is found, it 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 both implementations of the dictionary in a reasonable amount of real time.

**Board Representation Setup**

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 (e. g., and N x N board), hereafter referred to as N, as an integer value. 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. 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. This duplication of work is justified because we need to verify that row i and column j are both still valid every time we add a character at board[i][j], so we’ll need StringBuilders for both the row and column that we can pass to the searchPrefix method.

**Verifying a StringBuilder**

Before writing the main backtracking algorithm, it was clear to me that I would need to write a method to check if a given StringBuilder hit a dead end. I knew searchPrefix by itself was not going to be enough because of the existence of the wall tiles. I needed to be able to verify a row or column that represented more than one word or prefix at a time. I also needed to check if parts of strings were words and not just prefixes in places other than just the ends of rows and columns. I called this method verifyBuilder.

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. I had the main algorithm call the simple version of searchPrefix if the board contained no walls. I checked that I had 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 did have walls, 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 do not contribute to the validity of 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. 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. I call print the solution if it is a multiple of 10000, then return. If we’re using a DLB implementation of the DictInterface, the previous call will resume where it left off and continue to find additional solutions, if they exist. This is one of the possible base cases.

If the current row is not equal to N, 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. We add c to the row StringBuilder and call verifyBuilder. If c breaks the row, we must backtrack. This is the pruning part of the algorithm where we peel off huge pieces of our exponential search space. We remove c from the StringBuilder and return. Otherwise, we add it to the column StringBuilder also and call verifyBuilder again. If it fails, we trim c from the end of both StringBuilders and backtrack. If it succeeds, we have placed a character that might lead to a solution and so we recurse. Either we move down the row if there is room there, or else move to the first column of the next row (accomplished by row indexing mod N).

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 can proceed forward recursively, we do. We only step back where necessary, and only back one space. 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. When our recursive calls finish and we are out of characters left to try, we trim the StringBuilders and backtrack to keep looking for solutions. The recursive chain only ends when we’ve tried all possibilities and found all solutions, or we terminated the whole program early when using the MyDictionary implementation of the dictionary.

**Asymptotic Runtime Analysis**

It’s easy to see that runtimes can get huge here even for paltry values of N. How bad are they in 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 we know that this isn’t going to 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 for small enough problem sizes.

**MyDictionary**

Alas, 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, which is linearithmic. 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.

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**

Is the DLB trie better? 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. 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 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.

Crossword.java Performance

|  |  |  |  |
| --- | --- | --- | --- |
| TEST FILE | MyDictionary | DLB (first solution) | DLB (all solutions) |
| 3A | <1 second | <1 second | 5 seconds |
| 3B | 2 seconds | <1 second | <1 second |
| 4A | 6 seconds | <1 second | 23 minutes |
| 4B | 6 seconds | <1 second | 1 hour 54 minutes |
| 4C | 1 second | <1 second | 5 seconds |
| 4D\* | <1 second | <1 second | <1 second |
| 4E\* | 2 hours | 4 seconds | 4 seconds |
| 4F | 8 seconds | <1 second | <1 second |
| 5A | 5 seconds | 1 second | 5 hours 11 minutes |
| 6A | too long to test empirically by deadline | 3 minutes 15 seconds | 3 hours 22 minutes |
| 6B | 1 hour 12 minutes | 4 seconds | too long to test empirically by deadline |
| 6C\* | at least several hours | 17 seconds | 17 seconds |
| 7A\* | too long to test empirically by deadline | 2 hours 2 minutes | 2 hours 2 minutes |
| 8A\* | too long to test empirically by deadline | 1 hour 3 minutes | 1 hour 3 minutes |
| 8B | too long to test empirically by deadline | 1 hour 15 minutes | too long to test empirically by deadline |
| 8C | 22 minutes | 1 second | 1 second |

\* indicates a board with no solutions

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

**Backtracking Algorithm Refinement**

Relatively close to the deadline for this assignment, I successfully created an adjusted version of the crossword generation algorithm. I noticed that it was sometimes desirable 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. To run the revised solution, pass the string “fast” as the third command line argument to Crossword.java. See page 10 for results.

In each recursive call, I introduced a boolean to keep track of whether there were any valid vertical solutions for ANY of the possible options in the current tile. 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. To do this, I changed the return value of solveTile from void to integer. This way, I can pass how many extra tiles (and thus activation records) I want to backtrack whenever I encounter a vertical dead end. For an arbitrary board, this simply means backtracking (N-1) more tiles than normal. Normal circumstances simply return zero. Then, I just need to test the return value of each recursive call when it returns to decide whether to immediately return (oldReturnValue - 1) or to continue execution as normal. As a minor side note, I also altered the algorithm to verify columns prior to verifying rows to ensure its correctness.

When tested empirically, this version of the algorithm performed markedly better for certain boards, particularly those with tiles which must be used exactly as specified. This makes sense, since in these cases one failed vertical test meant I’d reached a vertical dead (rather than 26 consecutive failed tests for an unspecified open square), so the heuristic was employed more often. I didn’t encounter any cases where the overhead of the additional procedures outweighed the advantage of the more intelligent backtracking, though a few cases did not significantly improve.

**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 some horrifying runtimes here, two or three times faster probably matters to us a lot. Other than the increased memory usage, the main downside of the multiway trie is that its implementation is too elegant and trivial to properly torment Algorithms students for two weeks’ time.

Revised 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