My algorithm:

* Recursive function letter\_by\_letter(board)
* Passes the current board
* With each call, it finds the blank spots on the board and sorts them by how constrained the words that the letter is inside of is
* For each open spot it loops through a string of the most common letters in the given list of words (I found that using an alphabet sorted based on frequency in the English language as a whole was flawed because it put too much emphasis on words like ‘the’ that only appear in the dictionary once)
* I place this letter in the current spot and then extrapolate until the board stops changing or is broken.
* In my ‘extrapolation’ function, named check\_feasability of course, I see if there are any instances where only 1 word fits. If that’s the case, I fill in the letters for that word. If there are any cases where no words in the dict will satisfy the conditions for a word, it returns false right there.
* After extrapolating, I check if the board is done, and then call letter\_by\_letter on the new board
* if the result returned by this call is a board, then I just return it. Sometimes however, it returns an integer.
* The letter\_by\_letter function returns an integer when it has gone through all letters for an open board space and found no possible moves. The integer it returns is the index of the board space it failed on.
* Should an integer be returned, the algorithm keep returning that integer until the current spot that's being tried is in the same row on column of the integer spot being returned. Once I’ve reached a spot where the problematic spot is in the same word as the current spot, I break out and try the next letter with the hope that this won’t cause conflicts later on.

1)

Our algorithms to solve the puzzle are quite similar when examined in larger chunks and general ideas because they both use a letter by letter approach, but the specific fashion in which things are done are different most of the time.

One broad difference between our algorithms is that yours stores a lot more information about the board and each space (available letters for a spot, vs, hs, etc.) than mine, which instead gets finds/creates the information the moment it needs to and typically doesn’t store it for later access. This results in my program being more repetitive and thus inefficient than it needs to be in a few instances. For example, rather than just holding an updating the list of possible letters for a space, I will put any letter in a spot and then check through the entire board to see if it causes any issues, and if it doesn't cause issue then I know if that one letter is allowed. The storage of easily accessible useful information results in different approaches to many subsections of the algorithm.

An example of this is how we find the next spot to fill. Because you have a dictionary of the valid possible letters for each spot, you just find and choose the spot with the least number of letter options. My algorithm on the other hand churns through the whole board to find the open spot with the maximum number of already filled letters in its horizontal/vertical word slots. (i.e. Filling a letter in HE.LO is chosen over filling a letter in .E….). It gives me a sorted list of the ‘most constrained’ spots for that board, so I only have to do this once per unique board, but it takes a fair amount of time.

Next, I took a letter from my sorted letter list and put it in the board. When I used the letter frequency table from blackboard I found that it was prioritizing letters in an inefficient manner because instead of putting the most frequent letter in the dictionary first, it put the most frequent letters in the English language as a whole (i.e. the most common letters in a whole book) first. As a result, letters like T, found in common words like ‘the’ were valued highly, even though ‘the’ is only found once in the dictionary. In order to accurately prioritize the letters, I sorted the alphabet by frequency in the given dictionary at the beginning of each run.

Once I had this new board with the letter added, I extrapolated from the given board to fill as many other spots I could with confidence. For example, if after placing a letter I found a word slot that with the new letter constraints only had one word that would fit, I would fill in the letters to complete that word. I continued to extrapolate on the board until it stopped changing or it was no longer valid (there was a word slot with no words that fit it).

Similar to your algorithm, I needed to find a better way to backtrack than just going back one step to prevent wasting time changing letters that don’t clear the jam. My way around this was basically the same as yours: when I found a spot that couldn’t be filled with any letter without resulting in an unsolvable board, I would return the spot index. If when I recursively called the method after extrapolating the board I received a spot index instead of a board, I kept returning that spot index until the spot I was currently filling was in the same word slot as the returned spot index. Once I hit this point, I would move on to the next letter and continue the method as normal. (this was the part that I changed and didn’t realize I had changed it until I was looking at my code while writing this, and then when I unchanged it I got 100%)

When I profiled my program, I found that it spent a lot of time on those repetitive information finding methods that I mentioned in the 2nd paragraph. In order to cut down on the amount of time spent doing the same thing over and over again, I made caches for many of them. For example, I had this one method that took a partially filled word slot (HE..O) and turned it into a regular expression that would find a word for the slot ((?=\bHE..O\b).{5}\b). This was a simple O(1) method, but when called tens of thousands of times, it began to cost valuable seconds, so rather than doing the string operations over and over again on the same inputs, I made a dictionary of inputs to outputs that the method updated as it solved for new inputs.

Obviously regexing the entire dictionary was a bad idea, so I basically bucketed words by their length. I just made a dict of word lengths to a string of all words of that length at the beginning of each run using the given dictionary, and I searched through the appropriate word length dictionary to find if a word existed or not. This worked pretty well, but there were definitely better ways to do it (like yours for instance) if I had spent more time optimizing this portion of the code.

One big thing that I learned from this assignment is that detailed planning (more than just method names) is needed for assignments like these. Although I did have a decent call tree and an outline of what each method should take as inputs and return, I leaped from naming the methods to writing the methods with the actual manner in which the methods would function still foggy in my head. This resulted in a situation where after a few hours of coding, I had a program that made my CPU regret ever being made. The first time a ‘completed’ crossword appeared on my screen it took about a minute and was a grid of 16 E’s. Because I spent no time planning how the methods would work, they were slow and wrong. These were all eventually scrapped after I sat down with a pencil and paper and mapped out specifically how the main methods would function almost line to line.

Another more specific thing that I learned after finishing the assignment is that I should, as gnasher729 on stack exchange puts it, ‘avoid global variables like the plague’. When I was looking at your code I saw that almost all of your methods, your complicated ones in particular, have tons of parameters (relative to mine at least). For example, my recursive puzzle solving method has one parameter: board, and yours has 6. I had opted to use a lot of global variables for my stuff which didn’t cause any problems, but I was curious if it was a personal thing or if globals vs parameters mattered that much. Logically, it does. I’ve been able to use globals and get away with it so much because all of our assignments, even Othello, have been something along the lines of get a bunch of input and spit out or display a result. For this, global variables allow easy transfer of data between methods while avoiding extra lines and opportunities to pass the wrong variable or have the values for two things swapped. As you move beyond assignment coding however, using globals makes the code harder to maintain, read, and its harder to track down errors when they occur. With this in mind I will use way less global variables in the future because its bad practice and I may as well get out of this habit early on.

Throughout this assignment I learned how much I hated getting rid of any old code, and how problematic this is. As I mentioned earlier, I began writing methods without planning in enough detail because for previous assignment it had worked out well, but this came back to bite me. The effects of this probably could have been mitigated if I had fully accepted my mistake and started fresh, but every time I tried I kept telling myself that my old methods would work fine, and I should either use those as a launching pad for my next iteration or I should continue to attempt to modify them. 2 times I make a whole new python file and then after about 5 minutes I copied like 75% of my old code into the new file and went from there, essentially nullifying the progress made by the idea of starting fresh. Some of this I blame on how I found time to write code (outside of class I typically worked in 30 minute sections before my 5 pm practices, so I always told myself I don't have enough time in this session to truly start fresh, so I might as well continue editing my old code), but my reluctance to let go cost me a lot of time and when I finally convinced myself to let go of my junky old spaghetti code, it made life so much easier.