**Part C:**

**Parameters of considerations:**

d: Total number of characters in the dictionary (dictionary size)

$l_m$: The maximum length of the longest word in the dictionary

k: The size of the alphabet

n: The dimension of the puzzle board

**Worst case assumption:**

For worst case analysis, we will assume that the dictionary is designed to maximize the size of the prefix-tree (which might be different in each case) and the characters used inside the grid is set to against any potential advantages.

**FOR ORIGINAL PROBLEM:**

**Algorithm used:**

The algorithm used to solve this problem uses recursion/ backtracking. The details are included in the code implementation for part A, but rough steps are:

1. Start from a cell[i][j] on the board
2. If already reached the leaves of the prefix tree, then mark the found words (by marking an index in my implementation) and return
3. If the cells are already marked as visited or is the location is not correct, return early
4. If not, then check if the character on the board[i][j] matches the character on the edge of any child of the current prefix tree node
5. If not, return early
6. If yes, then it is a (potentially) valid path, thus:
    a. Mark current cell as visited
    b. Move down the prefix tree following the edge with the matched character
    c. Recursively calls step 2 to 4 on the 8 cells nearby from the current cell
7. After finish step 6, mark the current cell as unvisited while backtracking, to allow different path to visit cell[i][j] if haven't visited it before

Essentially, the algorithm tests all possible path of length $l_m$ starting from each cell[i][j] in the board.

**Worst case implication:**

Since there is no limit/ constraints on the type of input, since there are k possible characters at each position in the string, and the maximum string length is $l_m$, at worst, the number of all possible combinations are:

$k^{lm} + k^{lm-1} + k^{lm-2} + \cdots + k = \frac{(k^{lm+1} - k)}{k-1} \approx k^{lm}$ (for large value of k -> infinity).

This algorithm is not input-sensitive, since it doesn't make use of any specific characteristics of the grid, thus only the grid size matters.

**Space complexity:** $O(n^2 + k*k^{lm})$ for the table to store the puzzle table (and a table with the same size to check if a cell is visited or not) and the prefix tree.

For the prefix tree, since there are $k^{lm}$ different string combinations, and each of them will have an extra k-sized array storing their single terminal null value, the total size of the tree is therefore $k*k^{lm}$. For the intermediate character, the overlapping between the string ensures that the prefix tree is only $k^{lm}$ for all the string combinations.

**Time complexity:**

The algorithm involves 2 steps: building the prefix-tree and searching for matched words, which are best analyzed separately, since they serve different purposes and have different basic operations, so attempting to fuse them into 1 complexity would yield either an incomplete or meaningless result.

**Basic operation:** Allocate memory for the prefix-tree

**For prefix-tree building:**

$O(k * k^{lm})$, as explained above, this is the maximum size of the prefix-tree in the worst possible tree.

**Note:** If the dictionary has an infinite duplicates of the same string, then the scanning operations can become dominant, yielding an O(d) worst case, with d being the total number of characters in the dictionary. Memory allocation is sometimes higher, due to the

fact that for 1 character, it might need k memory allocations for the k-sized pointers, if the character leads to a brand-new node.

**For matched word searching:**

**Basic operation:** Character comparison (between the cell's and the prefix-tree's character)

Let $r = \min(n^2, l_m)$

For each cell[i][j], the upper bound for the number of paths is $7^r$ as there are at most 7 viable directions at any position in the path (ignoring 1 direction coming back to the previous cell, which will be terminated immediately), and we need to pick these directions r times for the path with max length r (equivalent to the min(max height of the prefix-tree, all cells in the grid)).

There are n * n cells on the board, so the total time complexity for searching the matched words is: $\mathbf{O(n^2\ 7^{\min(n*n,\ lm)})}$

**FOR INCORPORATING THE FACT THAT THE LETTER CAN ONLY APPEAR ONCE:**

**Note:**

For convenience, each cell[i][j] on the board will be represented by a node (labelled uniquely by (dimension*i + j)), connecting initially to its surrounding cells/ nodes (which will be referred to as children of cell[i][j]). The edge represents that we can move between the 2 connected node.

**Algorithm used:**

The fact that each word can only contains each letter once can be incorporated by:

1. On the tree: Filter the dictionary and only use the words with unique letters for building the prefix tree.
2. On the board traversal: From each cell, before traversing to the children, compress all the child nodes with the same characters. Details included below.

For filtering, the algorithm is:

1. Use a k-sized array letterCount as a dictionary
2. Scan through the letter of each word
3. If letterCount[letter] = 0, mark letterCount[letter] = 1 (saw first time)
4. If letterCount[letter] = 1, then the letter is repeated, so terminate the algorithm and return false
5. If reach the end of the string without termination, return true

Since the algorithm needs O(k) to allocate the temp array, and O(d) to scan through all the characters, the total complexity of the checking is lower than the required scanning and initialize the single root for the prefix-tree, so doesn't affect the total complexity of tree building.

For compression step referred above, before traversing down the children, 4 main steps can be incorporated:

1. Split the children into groups sharing the same characters. By allocating an array of size k, with each position storing a linked list to store the node with same characters, then iterate through the children, putting them into the correct position (similar to counting sort), this takes $O(k\ n^2)$. Otherwise, if k is large, then using heap-sort for grouping would lead to $O(n^2 \log(n))$.
2. Pick the first node from the group as a primaryNode (referred as primaryCell in my code), and compress these nodes by:

    a. Combine all the children of all nodes in the group into the children of the primaryNode. Using an adjacency matrix, this involves traversing through each row for each child node, and combining it with the primaryNode's row using OR operation. Using an adjacency matrix will take $O(n^4)$ but using an adjacency list can reduce this to $O(n^2)$ (since for each children, it can have originally at max 8 children).
3. From the parent node's viewpoint, only process this single primaryNode instead of the entire group, since they are equivalent. The process, in this case, involves recursively traversing down the primaryNode with the new list of combined children.
4. After finished processing, "decompress" the primaryNode (by reverting its neighbour row back to its original state, taking $O(n^2)$ if storing the original adjacency matrix).

   This decompression is crucial for correctness. From the parent's view, since it can reach all children with the same path, the children with same characters only differ in their children, which is encompassed by the primaryNode. However, from other nodes (eg: the "brother" node), this might not be true, since for those nodes, the path leading from them to those grouped children might not be the same.

Initially, create a null node connecting to all nodes (cells) on the board, to ensure that even the choice to start the path is also compressed and grouped based on the characters by step 2, 3, 4. Fully optimized, all 4 steps take overall $O(\min(k, \log n)\, n^2)$ (depends on the grouping if k is large).

The compression step is feasible, since the valid path/ word doesn't contain repeated characters, allowing the parent node to ignore all the edges/ potential path between the nodes with same characters and treat this group of nodes as a single node.

Note that, with such compression, the overlapping children and grandchildren are fused into one. This ensures that each unique path on the grid is only traversed once, reducing the number of possible paths required for traversing significantly if there are repeated characters (the more the better).

Besides the introduction of the 4 steps above, the same backtracking algorithm with cross-comparison with the prefix-tree similar to part A is used.

**Worst case implications:**

As explained above, unlike without constraints, the algorithm utilizing the constraints benefits from the repeated characters on the grid (which cut down the number of unique

paths on the grid). Thus, the worst input for this is when the grid is filled with unique characters. Note: this implies that k >= $n^2$. For simplicity, we will assume that k = $n^2$ (as both needs to approach infinity anyway and picking k = $n^2$ would also ensure all string in the dictionary fits in the grid, which is reasonable).

After the filtering, the remaining words will have max length $l_m$ <= k, as it can only contain up to all letters in the k alphabet. This will also limit the possible letters arrangement, in the worst case, to all the permutation of k letters of length from 1 to k. Total unique string, thus, is:

$$kPk + kP(k-1) + \cdots + k = k! + \frac{k!}{1!} + \frac{k!}{2!} + \cdots + \frac{k!}{(k-1)!} = k!\,(\sum_{i=0}^{k-1} \frac{1^i}{i!}) \approx k!\,e^1$$, as k -> infinity

**Space complexity:** For the total unique string, note that all of the string with length < k are simply the prefix of the kPk set of strings. However, due to the current implementation suggested in the assignment, we still need to store an additional k-sized array for the single null pointer for each of them. Thus, the tree size ends up as O(k * k!).

During implementation, the board needs to be constructed as a graph. If implemented as an adjacency list, then it takes 8 * $n^2$, which is O($n^2$). During compression, the total number of links change at most by a factor of 2 (since we only copies the links into the primary node once during compression, and since there is no repeated character in a path, no node contains more than one compression). Other tables (eg: visited, found, ...) might also be used to store flags/ info, but none of them has size larger than the grid or the prefix-tree. The total complexity is therefore **O($n^2$ + k * k!)**

**Time complexity:**

**For building prefix-tree:**

The additional filtering, as explained above, contributes nothing to the overall complexity.

**Basic operation:** Allocate memory for the prefix-tree

**For prefix-tree building: O(k * k!),** as explained, this is the maximum size of the prefix-tree in the worst possible tree. The same caution apply as for original, since O(d) might become dominant if the dictionary contains multiple duplicated entries.

**For matched word searching:**

**Basic operation:** Character comparison (between the cell's and the prefix-tree's character, as well as during the grouping of nodes with the same character).

As shown above, there are $O(k * n^2)$ basic operations required for the compression/ decompression and comparison between the current cell character and the prefix-tree.

$O(\min(k, \log n) * n^2)$ is a loose bound, as with the assumption of unique characters, there are no possible compression, thus, except the initial null node, the maximum number of children node is 8, making the additional computation $O(k * 8) = O(k)$. Still, since this operation is input-sensitive, I'd take the cautious, absolute worst performance as the bounds.

For each cell[i][j], the upper bound for the number of paths is $7^k$ as there are at most 7 viable directions at any position in the path (ignoring 1 direction coming back to the previous cell, which will be terminated immediately), and we need to pick these directions k times for the path with max length k (equivalent to the min(max height of the prefix-tree, all cells in the grid), assuming $k = n^2$).

Let $n_{grid}$ = number of unique path on the grid, that is also in the dictionary

$n_{grid} \leq number\ of\ unique\ string\ in\ the\ dictionary \leq k!$

$n_{grid} \leq upper\ bound\ of\ number\ of\ paths\ in\ the\ grid \leq 7^k$

Thus, with the constraint of the algorithm, the final time complexity:

**$O(\min(k, \log n) * n^2 * n_{grid}) <= O(\min(k, \log n) * n^2 * \min(k!, 7^k))$**

**OVERALL COMPARISON:**

**Note:** The advantage is compared between the upper bound big O, which is technically not a good metrics, since having a tighter upper bound doesn't necessarily mean the algorithm is asymptotically better. Still, I tried to make the bound the tightest (still not $\theta$ because the algorithm is input-sensitive, so quite hard to find $\theta$), thus having a tighter upper bound likely indicate a better algorithm.

The 2nd version discussed above is clearly input-sensitive, since we are utilizing repeated characters on the grid to speed up the process. Obviously, if the grid only has unique

characters, all possible paths are unique and the 2nd version still traverses all the paths available, making it equally or less efficient than the 1st version, due to the overhead incurred for scanning through the children to check if able to compress. Still, this is easily overcome by simply running an $O(k + n^2)$ through the grid first to check if there are repeated values (using an k-sized array to count). If yes, then use the 2nd version, otherwise use the first version. The additional check contributes nothing compared to the massive complexity during the actual search. Such hybrid approach would ensure the 2nd version performs equally or more efficiently than the 1st version.

For the cases when the 2nd version is significantly better than the 1st version:

1. When k << max length of the string in the dictionary: This will be handled by the filtering initially, capping the length to only k and only path with unique characters.
2. When the number of unique path in the grid or dictionary < $7^k$ / (min(k, log n)): Since the edited version traverses the number of unique path in the grid or dictionary, this follows naturally. This happens with:
    a. Repeated characters: n duplicates of a character (clustering relatively together) lead to at least n times duplicated unique path containing that character (and a lot more if path with repeated characters are also considered)
    b. Limited unique string/ path in the dictionary itself

Due to it being input-sensitive, best method would be to practically compare the average runtime between the 2. For theoretical analysis, the 2nd version is at least as efficient as the 1st version, with the potential advantages as listed above.