Word Search II

Given a 2D board and a list of words from the dictionary, find all words in the board.

Each word must be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

For example,
Given **words** = `["oath","pea","eat","rain"]` and **board** =

```
[
  ['o','a','a','n'],
  ['e','t','a','e'],
  ['i','h','k','r'],
  ['i','f','l','v']
]
```

Return `["eat","oath"]`.
Note:
You may assume that all inputs are consist of lowercase letters `a-z`.

click to show hint.

You would need to optimize your backtracking to pass the larger test. Could you stop backtracking earlier?

If the current candidate does not exist in all words' prefix, you could stop backtracking immediately. What kind of data structure could answer such query efficiently? Does a hash table work? Why or why not? How about a Trie? If you would like to learn how to implement a basic trie, please work on this problem: Implement Trie (Prefix Tree) first.

Solution 1

Compared with Word Search, I make my DFS with a tire but a word. The Trie is formed by all the words in given *words*. Then during the DFS, for each current formed word, I check if it is in the Trie.

```
public class Solution {
    Set<String> res = new HashSet<String>();

    public List<String> findWords(char[][] board, String[] words) {
        Trie trie = new Trie();
        for (String word : words) {
            trie.insert(word);
        }

        int m = board.length;
        int n = board[0].length;
        boolean[][] visited = new boolean[m][n];
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                dfs(board, visited, "", i, j, trie);
            }
        }

        return new ArrayList<String>(res);
    }

    public void dfs(char[][] board, boolean[][] visited, String str, int x, int y
, Trie trie) {
        if (x < 0 || x >= board.length || y < 0 || y >= board[0].length) return;
        if (visited[x][y]) return;

        str += board[x][y];
        if (!trie.startsWith(str)) return;

        if (trie.search(str)) {
            res.add(str);
        }

        visited[x][y] = true;
        dfs(board, visited, str, x - 1, y, trie);
        dfs(board, visited, str, x + 1, y, trie);
        dfs(board, visited, str, x, y - 1, trie);
        dfs(board, visited, str, x, y + 1, trie);
        visited[x][y] = false;
    }
}
```

written by Lnic original link here

Solution 2

The idea is to use a Trie to build a prefix tree for words to simplify the search and do DFS to search all the possible strings. For Trie, 26 pointers to point the sub-strings and a bool leaf to indicate whether the current node is a leaf (i.e. a string in words) and also idx is used to save the index of words for the current node. For DFS, just check if the current position is visited before (board[i][j]=='X'), if so, return, check if there is a string with such prefix (nullptr == root->children[words[idx][pos]-'a']), if not, return; otherwise, check if the current searched string is a leaf of the trie (a string in words), if so, save it to res and set leaf of the trie node to false to indicate such string is already found. At last, move to its neighbors to continue the search. Remember to recover the char [i][j] at the end.

```
    class Solution {
        class Trie{
        public:
            Trie *children[26]; // pointers to its substrings starting with 'a' t
o 'z'
            bool leaf; // if the node is a leaf, or if there is a word stopping a
t here
            int idx; // if it is a leaf, the string index of the array words
            Trie()
            {
                this->leaf = false;
                this->idx = 0;
                fill_n(this->children, 26, nullptr);
            }
        };

    public:
        void insertWords(Trie *root, vector<string>& words, int idx)
        {
            int pos = 0, len = words[idx].size();
            while(pos<len)
            {
                if(nullptr == root->children[words[idx][pos]-'a']) root->children
[words[idx][pos]-'a'] = new Trie();
                root = root->children[words[idx][pos++]-'a'];
            }
            root->leaf = true;
            root->idx = idx;
        }

        Trie *buildTrie(vector<string>& words)
        {
            Trie *root = new Trie();
            int i;
            for(i=0; i<words.size();i++) insertWords(root, words, i);
            return root;
        }

        void checkWords(vector<vector<char>>& board, int i, int j, int row, int co
l, Trie *root, vector<string> &res, vector<string>& words)
        {
            char temp;
```

```
            char temp;
            if(board[i][j]=='X') return; // visited before;
            if(nullptr == root->children[board[i][j]-'a']) return ; // no string
with such prefix
            else
            {
                temp = board[i][j];
                if(root->children[temp-'a']->leaf)  // if it is a leaf
                {
                    res.push_back(words[root->children[temp-'a']->idx]);
                    root->children[temp-'a']->leaf = false; // set to false to in
dicate that we found it already
                }
                board[i][j]='X'; //mark the current position as visited
// check all the possible neighbors
                if(i>0) checkWords(board, i-1, j, row, col, root->children[temp-'
a'], res, words);
                if((i+1)<row) checkWords(board, i+1, j, row, col,  root->children
[temp-'a'], res, words);
                if(j>0) checkWords(board, i, j-1,  row, col, root->children[temp-
'a'], res, words);
                if((j+1)<col)  checkWords(board, i, j+1,  row, col, root->childre
n[temp-'a'], res, words);
                board[i][j] = temp; // recover the current position
            }
        }

        vector<string> findWords(vector<vector<char>>& board, vector<string>& wor
ds) {
           vector<string> res;
           int row = board.size();
           if(0==row) return res;
           int col = board[0].size();
           if(0==col) return res;
           int wordCount = words.size();
           if(0==wordCount) return res;

           Trie *root = buildTrie(words);

           int i,j;
           for(i =0 ; i<row; i++)
           {
               for(j=0; j<col && wordCount > res.size(); j++)
               {
                   checkWords(board, i, j, row, col, root, res, words);
               }
           }
           return res;
        }
 };
```

Based on the comments received. I created another version with Trie node counter (thanks, zhiqing_xiao and gxyeecspku). However, for the current test set, it doesn't help too much. Anyway, my version with Trie node counter.

```
class Solution {
```

```
private:
class Trie
{
public:
    Trie * children[26];
    bool isLeaf;
    int  wordIdx;
    int prefixCount;

    Trie()
    {
        isLeaf = false;
        wordIdx = 0;
        prefixCount = 0;
        fill_n(children, 26, nullptr);
    }

    ~Trie()
    {
        for(auto i=0; i<26; ++i) delete children[i];
    }
};
    void insertWord(Trie *root,  const vector<string>& words, int idx)
    {
        int i, childID, len = words[idx].size();
        for(i=0, root->prefixCount++ ; i<len; ++i)
        {
            childID = words[idx][i]-'a';
            if(!root->children[childID]) root->children[childID] = new Trie();
            root = root->children[childID];
            ++root->prefixCount;
        }
        root->isLeaf = true;
        root->wordIdx = idx;
    }

    Trie *buildTrie(const vector<string> &words)
    {
        Trie *root = new Trie();
        for(int i=0; i < words.size(); ++i) insertWord(root, words, i);
        return root;
    }

    int dfs_Trie(vector<string> &res, Trie *root, vector<vector<char>>& board, ve
ctor<string>& words, int row, int col)
    {
        int detected = 0;

        if(root->isLeaf)
        {
            ++detected;
            root->isLeaf = false;
            res.push_back(words[root->wordIdx]);
        }

        if( row<0 || row>=board.size() || col<0 || col>=board[0].size() || board[
row][col]=='*' || !root->children[ board[row][col]-'a'] ) return root->children[ board[r
```

```
row][col]== '*' || !root->children[ board[row][col]- 'a' ] || root->children[ board[r
ow][col]-'a']->prefixCount <= 0 ) return detected;
        int curC = board[row][col] - 'a';
        board[row][col] = '*';
        detected += dfs_Trie(res, root->children[curC], board, words, row-1, col)
+
               dfs_Trie(res, root->children[curC], board, words, row+1, col) +

               dfs_Trie(res, root->children[curC], board, words, row, col - 1) +

               dfs_Trie(res, root->children[curC], board, words, row, col + 1) ;
        root->prefixCount -=detected;
        board[row][col] = curC+'a';
        return detected;
    }

public:
    vector<string> findWords(vector<vector<char>>& board, vector<string>& words)
{
        int M, N, wordNum = words.size();
        vector<string> res;
        if( !(M = board.size()) || !(N = board[0].size()) || !wordNum) return res
;
        Trie *root = buildTrie(words);
        for(auto i=0; i<M && root->prefixCount; ++i)
            for(auto j=0; j<N; ++j)
                dfs_Trie(res, root, board, words, i, j);
        delete root;
        return res;
    }
};
```

written by dong.wang.1694 original link here

Solution 3

```
public class Solution {
    public class TrieNode{
        public boolean isWord = false;
        public TrieNode[] child = new TrieNode[26];
        public TrieNode(){

        }
    }

    TrieNode root = new TrieNode();
    boolean[][] flag;
    public List<String> findWords(char[][] board, String[] words) {
        Set<String> result = new HashSet<>();
        flag = new boolean[board.length][board[0].length];

        addToTrie(words);

        for(int i = 0; i < board.length; i++){
            for(int j = 0; j < board[0].length; j++){
                if(root.child[board[i][j] - 'a'] != null){
                    search(board, i, j, root, "", result);
                }
            }
        }

        return new LinkedList<>(result);
    }

    private void addToTrie(String[] words){
        for(String word: words){
            TrieNode node = root;
            for(int i = 0; i < word.length(); i++){
                char ch = word.charAt(i);
                if(node.child[ch - 'a'] == null){
                    node.child[ch - 'a'] = new TrieNode();
                }
                node = node.child[ch - 'a'];
            }
            node.isWord = true;
        }
    }

    private void search(char[][] board, int i, int j, TrieNode node, String word,
Set<String> result){
        if(i >= board.length || i < 0 || j >= board[i].length || j < 0 || flag[i]
[j]){
            return;
        }

        if(node.child[board[i][j] - 'a'] == null){
            return;
        }

        flag[i][j] = true;
```

```
        node = node.child[board[i][j] - 'a'];
        if(node.isWord){
            result.add(word + board[i][j]);
        }

        search(board, i-1, j, node, word + board[i][j], result);
        search(board, i+1, j, node, word + board[i][j], result);
        search(board, i, j-1, node, word + board[i][j], result);
        search(board, i, j+1, node, word + board[i][j], result);

        flag[i][j] = false;
    }
}
```

written by harish-v original link here