Yu's Coding Garden

Every geek had a dream...

Thursday, August 22, 2013

leetcode Question 127: Word Ladder

Word Ladder
Given two words (*start* and *end*), and a dictionary, find the length of shortest transformation sequence from *start* to *end*, such that:

- Only one letter can be changed at a time
- Each intermediate word must exist in the dictionary

For example,

Given:
start = `"hit"`
end = `"cog"`
dict = `["hot","dot","dog","lot","log"]`

As one shortest transformation is `"hit" -> "hot" -> "dot" -> "dog" -> "cog"`,
return its length `5`.

Note:

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.

Analysis:

Use this problem to review the classic method : **Double breadth first search.**

Firstly, we targeting this specific problem. The idea is pretty straight forward:
From the start string, find the all the possible "next" string in the dictionary, and for each "next" string, find the "next next" strings, until meets the end string.

If there is a tree structure came up in your mind while you have seen this question for several mins, you almost get there! Yes! At least this problem can be solved using the classical searching algorithm, Depth First Searching (DFS) and Breadth First Searching (BFS).

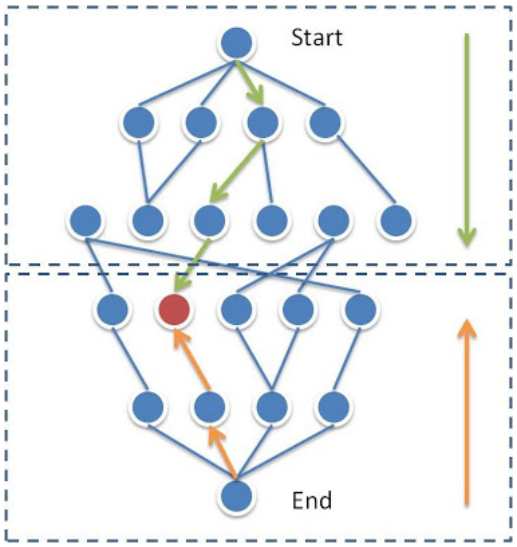
Which is better? The answer is "it depends". But in this problem, from my personal perspective, the BFS is more reasonable since the problem requires a specific shortest path. Since BFS searches in a level-by-level way, it is more suitable to find the shortest path. And for this problem, both the start state (start string) and end state (end string) are known, it thus reminds me to try the double breadth first search!

What else should be considered?
At first, I'm doing it like this: because I plan to use BFS, it is natural to find the valid string in the dictionary every time for the current top element in the queue, e.g. for i=1:dict_size { if valid(topstr, dict[i]), push();}. Does it work? Yes, there is no problem theoretically. However, let's think a little deeper for this specific problem: if we do the above loop, the complexity for this "add possible elements" is O(m*n), m is the length of the string, n is the size of the dictionary. Can we do better? Let's see the question again, where said that "only one letter can be changed", what does this mean? Of course it means we can check two strings from 1st to the last. Here is the trick when we saw the interface of the code------<unordered_set> . A nice property of unordered_set is it can find an element in O(1) time. The key here is: the search for all valid strings in the dictionary can be considered as we **generate all the possible strings and see if they are in the dictionary.** For each char in the string, we can change it to the 26 letters, one position at a time, check if the new string exists in the dictionary. What is the complexity of this step? O(m*26).

Actually I provide both code of the above idea, the 1st one can pass most of the test cases, but 3 failed 3 large cases. The 2nd one pass all the test cases. But the structure of the 1st one is more practical and general, I think it will help you more on the concept of BFS, as well as its extensions to other problems.

Double breadth first search.

When the start state and the aim state are all available for a search problem like this one, it is a good very very very good chance to use the double breadth first search! How it works, if you got the idea of the BFS, it is not hard to understand---------Search from both direction! Every time, we search one level from the start and one level from the end (there are also other schemes), the stop condition is found one node which has been marked by the other direction. A figure is show below which illustrates the idea and you can take a look at the code to get a better understanding of how it works.



Code (Cannot pass 3 of the large cases):

```
class Solution {
public:

    bool valid(string s1,string s2){
        bool flag=false;
        for (int i=0;i<s1.size();i++){
            if (s1[i]!=s2[i]){
                if (flag==true){return false;}
                else{flag=true;}
            }
        }
        return true;
    }

    int ladderLength(string start, string end, unordered_set<string> &dict) {
        // Start typing your C/C++ solution below
        // DO NOT write int main() function
        if (valid(start,end)){return 2;}

        if (dict.size()>2500){return 100;}



        struct node{
            string str;
            int lev;
            node(string s,int l):str(s),lev(l){}
        };

        queue<node>q;
        queue<node>rq;
        map<string,bool>mark;
        map<string,bool>rmark;
        for (auto it=dict.begin();it!=dict.end();it++){
            mark[*it]=false;
            rmark[*it]=false;
        }

        int level=1;
        int rlevel=1;
        q.push(node(start,1));
        rq.push(node(end,1));

        while (!q.empty() && !rq.empty()){


        if (q.size()<rq.size()){
            while (!q.empty() && q.front().lev==level){
                for (auto it=dict.begin();it!=dict.end();it++){
                    if (!mark[*it] && valid(q.front().str,*it)){
                        mark[*it]=true;
                        if (rmark[*it]){return q.front().lev+rq.back().lev;}
                        q.push(node(*it,level+1));
                    }
                }
                q.pop();
            }
```

questions *have been tested and have* ***passed both small and large*** *tests of the online judge system from leetcode.*

Pages

Home

Labels

answer (1) array (15) backtracking (1) BFS (12) binary search (7) binary tree (2) bit manipulations (6) BST (2) bubble sort (1) bucket sort (1) c++ (1) collision (1) common prefix (1) compare (1) design (1) DFS (14) dict (2) divide and conquer (1) DP (8) graph (2) hash (2) hash function (1) hash table (6) heap (2) heap sort (3) Index (2) inorder (2) insert sort. (2) iteration (3) iterative (2) iteratively (2) iterator (1) KMP (1) leetcode (199) linked list (7) list (1) loop (5) map (3) mask (1) math (4) merge sort (2) minimum substring (1) non-recursive (1) O(n) (1) online judge (1) oop (1) permutations (1) pointers (7) postorder (1) preorder (2) python (1) questions (1) queue (2) quick sort (1) radix sort (1) Re-view (3) recursive (7) selection sort (1) solutions (1) sort (1) sorting algorithm (2) stack (5) starter (4) string (9) stringstream (2) topology (1) trapping rain water (1) traverse (1) Tree (24) Tree Traversal (8) Trie (1) twosum (1) XOR (1)

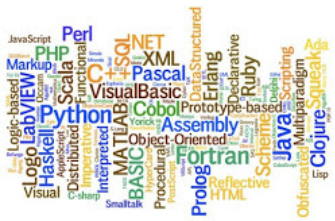
Blog Archive

- ► 2017 (1)
- ► 2015 (43)
- ► 2014 (15)
- ▼ 2013 (126)
  - ► Dec 2013 (14)
  - ► Oct 2013 (3)
  - ► Sep 2013 (2)
  - ▼ Aug 2013 (6)
    - leetcode Question 133: Palindrome Partitioning II
    - leetcode Question 131: Surrounded Regions
    - leetcode Question 132: Palindrome Partitioning
    - [Re-View]Hash Table (Basics)
    - leetcode Question 127: Word Ladder
    - [Re-view] Recursive: Array Permutations
  - ► Jul 2013 (1)
  - ► May 2013 (18)
  - ► Apr 2013 (32)
  - ► Mar 2013 (6)
  - ► Feb 2013 (9)
  - ► Jan 2013 (35)
- ► 2012 (29)

```
            level++;
        }else{

            while (!rq.empty() && rq.front().lev==rlevel){
                for (auto it=dict.begin();it!=dict.end();it++){
                    if (!rmark[*it] && valid(*it,rq.front().str)){
                        rmark[*it]=true;
                        if (mark[*it]){return rq.front().lev+q.back().lev;}
                        rq.push(node(*it,rlevel+1));
                    }
                }
                rq.pop();
            }

            rlevel++;
        }
        }

        return 0;
    }
};
```

Code (Pass all the test cases):

```
class Solution {
public:

    vector<string> findDict(string str, unordered_set<string> &dict){
        vector<string> res;
        int sz = str.size();
        string s = str;
        for (int i=0;i<sz;i++){
            s = str;
            for (char j = 'a'; j<='z'; j++){
                s[i]=j;
                if (dict.find(s)!=dict.end()){
                    res.push_back(s);
                }
            }

        }
        return res;
    }
 bool valid(string s1,string s2){
        bool flag=false;
        for (int i=0;i<s1.size();i++){
            if (s1[i]!=s2[i]){
                if (flag==true){return false;}
                else{flag=true;}
            }
        }
        return true;
    }
    int ladderLength(string start, string end, unordered_set<string> &dict) {
        // Start typing your C/C++ solution below
        // DO NOT write int main() function
        if (valid(start,end)){return 2;}


       struct node{
            string str;
            int lev;
            node(string s,int l):str(s),lev(l){}
        };

        queue<node>q;
        queue<node>rq;
        map<string,bool>mark;
        map<string,bool>rmark;
        for (auto it=dict.begin();it!=dict.end();it++){
            mark[*it]=false;
            rmark[*it]=false;
        }

        int level=1;
        int rlevel=1;
        q.push(node(start,1));
        rq.push(node(end,1));

        while (!q.empty() && !rq.empty()){


        if (q.size()<rq.size()){
            while (!q.empty() && q.front().lev==level){

                vector<string> l = findDict(q.front().str,dict);
                for (auto it=l.begin();it!=l.end();it++){
                    if (!mark[*it]){
                        mark[*it]=true;
                        if (rmark[*it]){return q.front().lev+rq.back().lev;}
                        q.push(node(*it,level+1));
                    }
                }
                q.pop();
            }
            level++;
        }else{

            while (!rq.empty() && rq.front().lev==rlevel){

                vector<string> lr = findDict(rq.front().str,dict);
```



```
                for (auto it=lr.begin();it!=lr.end();it++){
                    if (!rmark[*it]){
                        rmark[*it]=true;
                        if (mark[*it]){return rq.front().lev+q.back().lev;}
                        rq.push(node(*it,rlevel+1));
                    }
                }
                rq.pop();
            }

            rlevel++;
        }
        }

        return 0;
    }
};
```

Posted by Yu Zhu at 2:02 PM

Labels: leetcode

12 comments:

Ming Chen November 5, 2013 at 1:37 AM

Your second code (which passes all tests) may put the "start" and "end" into the queue for the second time, if they both appear in dict. Although it does not affect the correctness, it is unnecessary. One way to correct this is to avoid add itself into the vector in findDict().

Reply

Replies

udit gupta January 24, 2014 at 5:27 AM

Yes,Before starting bfs we should set mark[start]= true and rmark[end]=true

Reply

X Wang December 14, 2013 at 5:01 PM

Nice solution and good ads.

Reply

X Wang December 14, 2013 at 5:03 PM

programcreek.com

Reply

Replies

Yu Zhu December 14, 2013 at 5:37 PM

Thanks for you comment.
I've seen the website, programcreek.com, which is very good and I bookmarked it. Thanks!

Reply

Nison Li January 2, 2014 at 4:26 PM

Double BFD, Great work! It will reduce quite a lot of work if data set is huge! Thanks for sharing!

Reply

Deephan January 12, 2014 at 5:15 PM

The actual name of the algorithm is bi-directional search. http://en.wikipedia.org/wiki/Bidirectional_search

Good job.

Reply

udit gupta January 24, 2014 at 5:25 AM

valid(string s1,string s2) method is not needed as it will return 2 for case when start = "abc" end = "abc" and dictionary has only "def".
Before starting bfs we should set mark[start]= true and rmark[end]=true.

Reply

mlnotes September 8, 2014 at 8:50 PM

Very clean and nice solution, thanks !

Reply

Rayna September 13, 2014 at 9:27 PM

very smart!

Reply

boyapati ravi kumar December 5, 2014 at 2:26 PM

what is the complexity in first and second case!! in order of no of words!!

Reply

Badshah Pathan October 8, 2015 at 4:09 AM

this is not working...
for
hit
cog

dictionary=hot dot dog

Reply

Enter your comment...

Comment as: Google Accou

Publish Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)