Optiver - UK

2020 Dec18

Implement a error-checking algorithm for binary tree, it reads a string composed of sequence of tree-edges :

```
(A,B) (A,C) (B,D) (B,E) (C,F) (C,G)
```

- it is a sequence of bracketed label-pairs
- each label-pair is a tree-edge in `(parent,child)` format
- each label-pair is delimited by single space
- labels inside bracket are separated by a comma
- labels are all upper-single char

This statement is important. Whether a bracketed edge is a `(parent,child)` relation or just a node-tuple makes a huge difference in the algorithm. The former is a directed edge while the latter is a non-directed edge.

The error-checking algorithm should check to see if this is a valid binary tree (NOT necessarily a binary search tree) :

- if there is error, return the error
- if there are more than one errors, return the one with highest priority : `E1> E2> E3> E4> E5`
- if there is no error, print the tree in this format :

```
(parent(1st-child-subtree)(2nd-child-subtree))
```

- `1st` child means the first encounted child-node in the string
  `2nd` child means the second encounted child-node in the string
  `3rd` child onwards are regarded as error

Here is the list of errors :

`E1`	incorrect input string format
`E2`	there exists duplicated edges
`E3`	a node having more than 2 children
`E4`	a tree not covering all nodes
`E5`	not a normal tree, either ...
-	cycle formed, or
-	a node having multi-parents

My O(N) solution - This solution cannot pass Optiver requirement. Oooo Why?

```
std::uint32_t c2i(char c) { return c-'A'; }
char i2c(std::uint32_t i) { return 'A'+i; }

class tree_checker
{
public:
    tree_checker()
    {
        clear();
    }

    void clear()
    {
        root = empty;
        for(auto& x:tree)
        {
            x.parent = empty;
            x.lhs = empty;
            x.rhs = empty;
        }
        for(auto& x:errors)
        {
            x = false;
        }
    }

    bool load(const std::string& str)
    {
        std::uint32_t n=0;
        while(n<str.size())
        {
            char x; char y;
            if (!extract_one_edge(str, n, x, y))
            {
                errors[0] = true;
                return false; // stop processing and return E1
            }

            std::uint32_t xi = c2i(x);
            std::uint32_t yi = c2i(y);

            // In the following, whenever we encounter error, can we just stop processing and return that error?
            // No, because there may be another error with higher priority lying at the latter part of the input string.

            if      (tree[xi].lhs == empty)        tree[xi].lhs = y;
            else if (tree[xi].lhs == y)            errors[1] = true; // duplicated
            else if (tree[xi].rhs == empty)        tree[xi].rhs = y;
            else if (tree[xi].rhs == y)            errors[1] = true; // duplicated
            else                                   errors[2] = true; // more than 2 children

            if      (tree[yi].parent == empty)     tree[yi].parent = x;
            else if (tree[yi].parent == x)         errors[1] = true; // duplicated (this checking is redundant)
            else                                   errors[4] = true; // more than 1 parent, i.e. not a tree
        }

        // Final root checking
        for(std::uint32_t m=0; m!=26; ++m)
        {
            if (tree[m].parent == empty && (tree[m].lhs!=empty || tree[m].rhs!=empty))
            {
                if (root == empty) root = i2c(m);
                else errors[3] = true; // multiple roots
            }
        }
        if (root == empty) errors[4] = true; // If there is no root, implies that the tree forms a cycle.
        return true;
    }

    std::string get_output() const
    {
        if (errors[0]) return "E1";
        if (errors[1]) return "E2";
        if (errors[2]) return "E3";
        if (errors[3]) return "E4";
        if (errors[4]) return "E5";

        std::stringstream ss;
        ss << "(";
        print_to_ss(root, ss);
        ss << ")";
        return ss.str();
    }
```

```
private:
    bool extract_one_edge(const std::string& str, std::uint32_t& n, char& x, char& y)
    {
        // read a space char
        if (n>0)
        {
            if (str[n]!=' ') return false;
            ++n;
        }
        if (str[n]!='(') return false;
        ++n;
        if (str[n]<'A' || str[n]>'Z') return false;
        x = str[n];
        ++n;
        if (str[n]!=',') return false;
        ++n;
        if (str[n]<'A' || str[n]>'Z') return false;
        y = str[n];
        ++n;
        if (str[n]!=')') return false;
        ++n;
        return true;
    }

    // Recursive function for depth first search
    void print_to_ss(char c, std::stringstream& ss) const
    {
        std::uint32_t i = c2i(c);

        ss << c;
        if (tree[i].lhs != empty)
        {
            ss << "(";
            print_to_ss(tree[i].lhs, ss);
            ss << ")";
        }
        if (tree[i].rhs != empty)
        {
            ss << "(";
            print_to_ss(tree[i].rhs, ss);
            ss << ")";
        }
    }

private:
    static const char empty = '*';
    struct info
    {
        char parent;
        char lhs;
        char rhs;
    };

    char root;
    std::array<info,26> tree;
    std::array<bool,5> errors;
};


int main()
{
    std::string str;
    std::getline(std::cin, str);

    tree_checker chk;
    chk.load(str);
    chk.debug();
    std::cout << chk.get_output();
    return 0;
}
```

What's the problem with my solution?
After indepth evaluation, this problem is tricky, it is not about tree algorithm, instead it is about test-driven development. We need to design all possible test cases. During test design we can get a better understanding of what a tree is and what features the algorithm should offer. Make sure all assumptions and definitions are well understood, if the definitions are misunderstood, the solution must be wrong.

As far as I can recall from the question (and as far as I can understand) :

- given an edge `(A,B)`, then `A` must be parent and `B` must be child, so it is directed edge (not non-directed edge)
- errors are defined as following discussion (may not be the original definition, I try my best to recall them ...)

Root is different from parent :

- root is the parent of all nodes
- there should be one root for a tree
- there should be one parent for a node

There are no `lhs_child` nor `rhs_child` in the question, we simply :

- consider the first encountered child in the input string as `1st_child` and
- consider the second encountered child in the input string as `2nd_child`

When there are multiple errors, return the one with the highest priority :

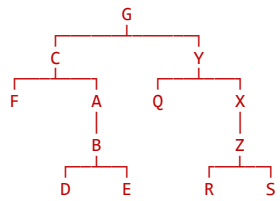
- therefore in my test cases, I always put expected error at the end of input string

Definition of `E4`

There are two cases regarding to `E4`, it means the set of nodes either :

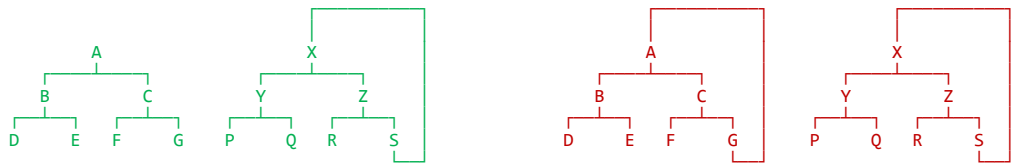
- forming multiple non-overlapping trees (green tree below), or
- forming multiple overlapping trees (red tree below)
- the red tree violates both `E4` (a tree not covering all nodes, i.e. multi-roots) and `E5` (a node having multi-parents)

For a general tree definition, IF edge `(A,B)` is just a node-tuple rather than a `(parent,child)` relationship, then the above red tree is also a valid tree, as there is only one path between any two nodes, it can be reorganised as :




```
note that :
parent connects to 1st_child
parent connects to 2nd_child
both children do not connect
```

However in this question, the edges are directed `(parent,child)` pair, hence normal tree-definition (no multi-path between two nodes) does not apply here. Furthermore, `E4` does not always imply multi-roots, for example the following returns `E4` with single root only or even no root (when `E4,E5a` happen together) :



One more remark : if we remove the tree rooted with node `A` above, the answer will become `E5a`. Woo ... subtle diff.

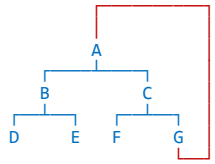
Definition of E5

There are 4 cases regarding to E5, I name them as E5a/E5b/E5c/E5d respectively :

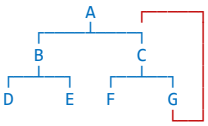
	major differences
• when a node having the root as its child	*form cycle, no multi-parents*
• when a node having non-root parent or non-root grandparent as its child	*form cycle, with multi-parents*
• when a node having nodes from another branch as its child	*no cycle, with multi-parents*
• when a node having nodes from another tree as its child	*no cycle, with multi-parents and E4*

In this question, a cycle means if we perform tree-traversal from any node in the cycle, by tracing the directed edges to its children, we will end up in the starting node. Therefore, multi-parents (like E5c) does not imply cycle.

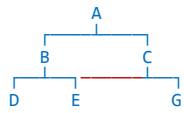
case E5a (form cycle, no multi-parents)



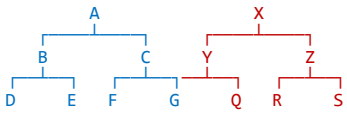
case E5b (form cycle, with multi-parents)



case E5c (no cycle, with multi-parents)



case E5d (no cycle, with multi-parents and E4)



In short, my final implementation is based on the assumption that :

- E4 is defined as **multi-roots** *OR* **multi-blobs** in region growing (grow in both parent and children links)
- E5 is defined as **multi-parents** *OR* **no-root**

Test Driven Development

This is TDD, we design test cases first. The expected result is invariant to order of input pairs. Please add shuffling.

```
// Suite 0 : Normal case (with single child or both children)
if (n==0)   { str = "(A,B) (A,C) (B,D) (C,F)";                  expected = "(A(B(D))(C(F)))";       }
if (n==1)   { str = "(A,B) (A,C) (B,D) (C,F) (C,G)";            expected = "(A(B(D))(C(F)(G)))";    }
if (n==2)   { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (C,G)";      expected = "(A(B(D)(E))(C(F)(G)))"; }

// Suite 1 : single error
if (n==3)   { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (C G)";      expected = "E1"; } // E1
if (n==4)   { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (B,E)";      expected = "E2"; } // E2
if (n==5)   { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (B,X)";      expected = "E3"; } // E3
if (n==6)   { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (X,Y)";      expected = "E4"; } // E4
if (n==7)   { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (D,A)";      expected = "E5"; } // E5a
if (n==8)   { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (D,C)";      expected = "E5"; } // E5b
if (n==9)   { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (C,D)";      expected = "E5"; } // E5c

// Suite 2 : double errors
if (n==10)  { str = "[A,B) (A,C) (B,D) (B,E) (C,F) (B,E)";      expected = "E1"; } // E1,E2
if (n==11)  { str = "(A B) (A,C) (B,D) (B,E) (C,F) (B,X)";      expected = "E1"; } // E1,E3
if (n==12)  { str = "(A,B] (A,C) (B,D) (B,E) (C,F) (X,Y)";      expected = "E1"; } // E1,E4
if (n==13)  { str = "(A,B),(A,C) (B,D) (B,E) (C,F) (D,A)";      expected = "E1"; } // E1,E5a
if (n==14)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (D,C) ";     expected = "E1"; } // E1,E5b
if (n==15)  { str =" (A,B) (A,C) (B,D) (B,E) (C,F) (C,D)";      expected = "E1"; } // E1,E5c
                                                                // no E5d unless there is E4
// when duplicated edge is not related to E3,E4,E5
if (n==16)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (B,X) (A,C)"; expected = "E2"; } // E2,E3
if (n==17)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (X,Y) (A,C)"; expected = "E2"; } // E2,E4
if (n==18)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (D,A) (A,C)"; expected = "E2"; } // E2,E5a
if (n==19)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (D,C) (A,C)"; expected = "E2"; } // E2,E5b
if (n==20)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (C,D) (A,C)"; expected = "E2"; } // E2,E5c
```

```
// when duplcated edge is ALSO the edge that causes E3,E4,E5
if (n==21)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (B,X) (B,X)";          expected = "E2"; } // E2,E3
if (n==22)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (X,Y) (X,Y)";          expected = "E2"; } // E2,E4
if (n==23)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (D,A) (D,A)";          expected = "E2"; } // E2,E5a
if (n==24)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (D,C) (D,C)";          expected = "E2"; } // E2,E5b
if (n==25)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (C,D) (C,D)";          expected = "E2"; } // E2,E5c

if (n==26)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (X,Y) (B,I)";          expected = "E3"; } // E3,E4
if (n==27)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (D,A) (B,I)";          expected = "E3"; } // E3,E5a
if (n==28)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (D,C) (B,I)";          expected = "E3"; } // E3,E5b
if (n==29)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (C,D) (B,I)";          expected = "E3"; } // E3,E5c

if (n==30)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (D,A) (X,Y)";          expected = "E4"; } // E4,E5a [FAIL-1]
if (n==31)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (D,C) (X,Y)";          expected = "E4"; } // E4,E5b
if (n==32)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (C,D) (X,Y)";          expected = "E4"; } // E4,E5c
if (n==33)  { str = "(A,B) (A,C) (B,D) (B,E) (C,F) (X,Y) (Y,F)";          expected = "E4"; } // E4,E5d

// Suite 3 : triple errors (not involve E1 as I know how its implemented)
if (n==34)  { str = "(A,B) (A,C) (B,D) (D,E) (D,F) (X,Y) (D,G) (D,G)";    expected = "E2"; } // E2,E3,E4
if (n==35)  { str = "(A,B) (A,C) (B,D) (D,E) (D,F) (D,A) (D,A)";          expected = "E2"; } // E2,E3,E5a
if (n==36)  { str = "(A,B) (A,C) (B,D) (D,E) (D,F) (D,C) (D,C)";          expected = "E2"; } // E2,E3,E5b [FAIL-2]
if (n==37)  { str = "(A,B) (A,C) (B,D) (D,E) (X,Y) (D,A) (D,A)";          expected = "E2"; } // E2,E4,E5a
if (n==38)  { str = "(A,B) (A,C) (B,D) (D,E) (X,Y) (D,C) (D,C)";          expected = "E2"; } // E2,E4,E5b
if (n==39)  { str = "(A,B) (A,C) (B,D) (D,E) (X,Y) (Y,C) (Y,C)";          expected = "E2"; } // E2,E4,E5d
if (n==40)  { str = "(A,B) (A,C) (B,D) (D,E) (D,F) (X,Y) (D,A)";          expected = "E3"; } // E3,E4,E5a
if (n==41)  { str = "(A,B) (A,C) (B,D) (D,E) (D,F) (X,Y) (D,C)";          expected = "E3"; } // E3,E4,E5b
if (n==42)  { str = "(A,B) (A,C) (B,D) (D,E) (D,F) (X,Y) (D,Y)";          expected = "E3"; } // E3,E4,E5d

// Suite 4 : quad errors
if (n==43)  { str = "(A,B) (A,C) (B,D) (D,E) (D,F) (X,Y) (D,Y) (D,Y)";    expected = "E2"; } // E2,E3,E4,E5d [FAIL-2]

// Suite 5 : minimal node-set with E2,E3,E4,E5
if (n==44)  { str = "(A,B)";                                              expected = "(A(B))"; }
if (n==45)  { str = "(a,a)";                                              expected = "E1"; } // E1
if (n==46)  { str = "(A,B) (A,B)";                                        expected = "E2"; } // E2
if (n==47)  { str = "(A,B) (A,C) (A,D)";                                  expected = "E3"; } // E3
if (n==48)  { str = "(A,B) (C,D)";                                        expected = "E4"; } // E4
if (n==49)  { str = "(A,A)";                                              expected = "E5"; } // E5a
if (n==50)  { str = "(A,B) (B,A)";                                        expected = "E5"; } // E5b
if (n==51)  { str = "(A,B) (B,C) (A,C)";                                  expected = "E5"; } // E5c
if (n==52)  { str = "(A,B) (C,B)";                                        expected = "E4"; } // E4,E5d
if (n==53)  { str = "()";                                                 expected = "E1"; } // E1
if (n==54)  { str = "";                                                   expected = "E5"; } // E5, not sure ...
```

My solution fails to handle 2 cases :

- when there are multiple trees (say *N*) and *N-1* of them have no roots (due to cycle), like `case30`
- when `E2/E3/E5` happen simultaneously, like `case36` and `case43`

My final solution

We can have a six-pass algo, one pass for each error and final pass for returning output string. In fact we can do better.

My final solution is a 3-pass algo :

- first pass for checking `E1,E2` using a histogram or `std::unordered_map` instead of `lhs_child` and `rhs_child`
- second pass for checking `E3,E4,E5` plus region growing algorithm
- third pass for generating output string

However, how can we perform region growing for cycle like `E5a`? Should we promote region growing to ***Union Find***?

- no ... we just need to do region growing with both `parent` link and `children` link
- we may pass our test if we blindly start region growing from node `A`, thus we need to shuffle the input strings

Union Find algo

Someone in the web suggested to replace region growing with ***Union Find*** algorithm (also known as ***Disjoint Set Union***) which takes *O(N/alpha(N))*, where *1/alpha(N)* is inverse Ackermann function.

However, I think region growing is more suitable for this case. Please read `algorithm.doc` for the difference between region growing and ***Union Find*** algorithm. The former finds connected region given whole set of edges in batch, while the latter checks if two points lie in the same region when given set of edges incrementally, it can stop immediately when positive.

Here is an updated solution, it solves all test cases. Some common members are skipped.

```
class tree_checker
{
public:
    // All complicated logics are here ...
    bool load(const std::string& str)
    {
        std::uint32_t n=0;
        while(n<str.size())
        {
            // *** E1 check *** //
            char x,y;
            if (!extract_one_edge(str, n, x, y))
            {
                errors[0] = true;
                return false;
            }

            // *** E2 check (fill tree-struct) *** //
            auto iter0 = tree.find(x);
            if (iter0 == tree.end())
            {
                tree[x] = info{empty,y,empty,{},{y}};
            }
            else
            {
                auto i = iter0->second.children.find(y);
                if  (i == iter0->second.children.end())         iter0->second.children.insert(y);
                else                                            errors[1] = true;
                if      (iter0->second.lhs_child == empty)      iter0->second.lhs_child = y;
                else if (iter0->second.rhs_child == empty)      iter0->second.rhs_child = y;
            }

            // *** E2 check (fill tree-struct) *** //
            auto iter1 = tree.find(y);
            if (iter1 == tree.end())
            {
                tree[y] = info{empty,empty,empty,{x},{}};
            }
            else
            {
                auto i = iter1->second.parents.find(x);
                if  (i == iter1->second.parents.end())          iter1->second.parents.insert(x);
                else                                            errors[1] = true; // redundant
            }
        }

        for(const auto& x:tree)
        {
            // *** E3 check *** //
            if (x.second.children.size()>2) errors[2] = true;

            // *** E4 1st-check (multiple roots) *** //
            if (x.second.parents.empty())
            {
                if (root == empty) root = x.first;
                else errors[3] = true;
            }

            // *** E5 1st-check (multiple parents) *** //
            if (x.second.parents.size()>1) errors[4] = true;
        }

        // *** E4 2nd-check (multiple blobs) *** //
        if (!error[3] && tree.size()>0) errors[3] = multiple_blobs(tree.begin()->first);

        // *** E5 2nd-check (no root) *** //
        if (root == empty) errors[4] = true;

        return true;
    }
```

```
    // Iterative implementation
    bool multiple_blobs(char c)
    {
        std::queue<char> q;
        q.push(c);

        while(!q.empty())
        {
            auto iter = tree.find(q.front());
            q.pop();

            if (iter != tree.end() && iter->second.label == empty)
            {
                iter->second.label = labelled;
                for(auto& x:iter->second.parents)  q.push(x);
                for(auto& x:iter->second.children) q.push(x);
            }
        }

        for(const auto& x:tree)
        {
            if (x.second.label != labelled) return true;
        }
        return false;
    }

private:
    static const char empty = '*';
    static const char labelled = 'x';
    struct info
    {
        char label;
        char lhs_child; // in fact, children is good enough, lhs_child is added for printing final output as FIFO served
        char rhs_child; // in fact, children is good enough, rhs_child is added for printing final output as FIFO served
        std::unordered_set<char> parents;
        std::unordered_set<char> children;
    };

private:
    char root;
    std::unordered_map<char,info> tree;
    std::array<bool,5> errors;
};
```