Fundamentals of Data Structures

Projects2 : Autograd for Algebraic Expressions

Date : 2024/03/04

2023–2024 Spring & Summer Semester

Table of Contents

Chapter 1 : Introduction

Problem

Description

Nowadays the application of automatic differentiation technology has greatly facilitated people's implementation and training of deep learning algorithms.Our task is to create an automatic differentiation program for algebraic expressions.

Input

Input an infix expression composed of brackets,operators including power(^), multiplication(*), division(/), addition(+) and subtraction(-), variables(strings of lowercase English letters) and literal constant.

Output

For each variable in the expression,output an arithmetic expression that represents the derivative of the input expression with respect to the variable.

Arrange the output in the lexicographical order of the variables.

Sample Input

```
a+b^c*d
```

Sample Output

```
a: 1
b: c*1/b*b^c*d
c: 1*ln(b)*b^c*d
d: b^c*1
```

Algorithm Analysis

The whole Algorithm can be devided into three main parts:

- Input the infix expression and create an expression tree.
- Store all the variable names in the expression and sort them in lexicographical order.
- Output the derivative of the input expression with respect to each variable.

Chapter 2 : Algorithm Specification

Step 1 : Create the expression tree

We create a **Struct** to store the information of the nodes in the expression tree.Every nodes in the expression tree has three components.One is a string indicating the value of the node (may be an operator,a variable name or a constant number) and the other two are two pointers respectively indicating leftchild and rightchild of the node.

We need the data structure **Stack** to assist building the expression tree.Nodestack stores the subtrees created from the expression while opstack stores all the operators.Pseudo-code is shown below:

```
for(every ch in the expression){
    if(ch=='(')
        push '(' into the opstack;
    if(ch==')'){//build a subtree from the expression in the brackets
        pop operator from opstack;
        while(operator!='('){
            pop a,b from nodestack;
            create subtree (a,operator,b)
            push the subtree into nodestack;
            pop operator from opstack;
        }
        pop '(' from opstack;
    }
    if(ch is a letter or a number)
        add ch to a variable name;//it must be a component of variable name/a number
    if(ch is an operator){
        pop a,b from nodestack;
        create subtree (a,operator,b)
        push the subtree into nodestack;
    }
}
while(opstack is not empty){
    pop operator from opstack;
    pop a,b from nodestack;
    create subtree (a,operator,b)
    push the subtree into nodestack;
}
pop root from nodestack;//root is the root of expression tree;
```

Step 2 : Store all variable names and sort

We use a **String Array** to store all variable names.Pseudo-code is shown below:

```
for(every ch in the expression){
    if(ch is a letter or a number)
        add ch to a variable name;
    else if (variable name is not in the array)
        add the variable name to the String Array;
}
sort all variable names;
```

Step 3 : Output

We define a function differentiate(root,var) to build an expression tree of the answer. Pseudo-code is shown below:

```
if(root==NULL)
    return NULL;
if(root->value is a constant number)//F(x)=c,F'(x)=0
    return NULL;
if(root->value==var)//F(x)=x,F'(x)=1
    return node("1");
if(root->value is an operator){
    left=differentiate(root->left,var);//get the derivative of left expression
    right=differentiate(root->right,var);//get the derivative of right expression
    if(root->value=="+")//F(x)=f(x)+g(x),F'(x)=f'(x)+g'(x)
        return subtree (left,"+",right);
    if(root->value=="-")//F(x)=f(x)-g(x),F'(x)=f'(x)-g'(x)
        return subtree (left,"-",right);
    if(root->value=="*"){//F(x)=f(x)*g(x),F'(x)=f'(x)*g(x)+f(x)*g'(x)
        build subtree s1 (left,"*",root->right);
        build subtree s2 (root->left,"*",right);
        return subtree s3 (s1,"+",s2);
    }
    if(root->value=="/"){//F(x)=f(x)/g(x),F'(x)=[f'(x)*g(x)-f(x)*g'(x)]/g(x)^2
        build subtree s1 (left,"*",root->right);
        build subtree s2 (root->left,"*",right);
        build subtree s3 (s1,"-",s2);
        build subtree s4 (root->right,"^","2");
        return subtree s5 (s3,"/",s4);
    }
    if(root->value=="^"){//F(x)=f(x)^[g(x)],F'(x)=
[g'(x)*lnf(x)+g(x)*f'(x)/f(x)]*f(x)^g(x)
        build subtree s1 (right,"*",ln(root->left));
        build subtree s2 (root->right,"*",left);
        build subtree s3 (s2,"/",root->left);
        build subtree s4 (s1,"+",s3);
        build subtree s5 (root->left,"^",root->right);
        return subtree s6 (s4,"*",s5);
    }
}
```

Chapter 3 : Testing Results

Test case 1

Test case 1 wants to test expression including "+,-,*,/"

Input

```
a+b*c-d/e
```

Output

```
a: 1
b: 1*c
c: b*1
d: -1*e/e^2
e: -(-d*1)/e^2
```

Test case 2

Test case 2 wants to test expression including "+,-,*,/,(,)".

Input

```
(a+b)*c-(d-e)/f
```

Output

```
a: 1*c
b: 1*c
c: (a+b)*1
d: -1*f/f^2
e: -(-1)*f/f^2
f: -(-(d-e)*1)/f^2
```

Test case 3

Test case 3 wants to test expression including variables whose length is longer.

Input

```
(aabbcc+ddffea)*ddfa-bb/ss*cc*(ddffs+ee)
```

Output

```
aabbcc: 1*ddfa
bb: -1*ss/ss^2*cc*(ddffs+ee)
cc: -bb/ss*1*(ddffs+ee)
ddfa: (aabbcc+ddffea)*1
ddffea: 1*ddfa
ddffs: -bb/ss*cc*1
ee: -bb/ss*cc*1
ss: -(-bb*1)/ss^2*cc*(ddffs+ee)
```

Test case 4

Test case 4 wants to test expression including forms of "^".

Input

```
aa^2+2^aa+2^2+aa^aa
```

Output

```
aa: 2*1/aa*aa^2+1*ln(2)*2^aa+(1*ln(aa)+aa*1/aa)*aa^aa
```

Test case 5

Test case 5 mix all the situations in Test case 1,2,3,4.

Input

```
aa*10*bb+2^ab/ab+abb^abb/aab-(ba+bba)*baa^20
```

Output

```
aa: 1*10*bb
aab: (-abb^abb*1)/aab^2
ab: (1*ln(2)*2^ab*ab-2^ab*1)/ab^2
abb: (1*ln(abb)+abb*1/abb)*abb^abb*aab/aab^2
ba: -1*baa^20
baa: -(ba+bba)*20*1/baa*baa^20
bb: aa*10*1
bba: -1*baa^20
```

Chapter 4 : Analysis and Comments

Time Complexity

For step 1 : Creating the expression tree, we traverse all operators, variables and numbers in the expression, so the time complexity of step 1 is $O(n)$ (Assume the length of the expression is n).

For step 2 : Store all variable names and sort, we traverse the expression again and sort all variable names. The time complexity of Quick_sort Algorithm is $O(mlogm)$, so the time complexity of step 2 is $O(n) + O(mlogm) = O(n + mlogm)$ (Assume there are m variable names in the expression)

For step 3 : Output, firstly we differentiate all m variables and use the function m times.The function differentiate traverse all operators, variables and numbers in the expression tree and extend constant number of nodes, so the time complexity of step 3 is $O(mp)$ (Assume there are p nodes in the expression tree)

To sum up, the total time complexity of the program is $O(n + mlogm + mp)$(n indicates the length of the expression, m indicates the number of the variables, p indicates the number of nodes in the expression tree)

Space Complexity

The whole program construct $c_1 * p$ nodes(c_1 is a constant) , c_2 strings(c_2 is a constant) to store the expression and a string array(but the total length is the length of the expression). So the total space complexity is $O(p + n)$ (n indicates the length of the expression, p indicates the number of nodes in the expression tree)

Comments

The function of this program is still limited, for it can't support mathmatic functions such as $\sin x, \cos x, \tan x, \ln x, \log(x, y), \exp(x)$. . . . Also it can't simplify both the input expression or the output expression. It still needs to be improved.

Appendix : Source Code

```
#include<iostream>
#include<string>
#include<algorithm>
using namespace std;

struct TreeNode{
    string value;
    TreeNode *left;
    TreeNode *right;
};

struct Stack{//define a stack to build the expression tree from inorder expression
    TreeNode *data[100];
    int top;

    void push(TreeNode *node){//pushing the node into the stack
        data[++top]=node;
    }

    TreeNode *pop(){//pop and acquire the top element in the stack
        return data[top--];
    }

    TreeNode *visit(){//acquire the top element in the stack
        return data[top];
    }

    bool empty(){//judge whether the stack is empty
        return top==-1;
    }
};

int precedence(char op){//get the precedence of the operator
    switch(op){
        case '^':return 3;break;
        case '*':return 2;break;
        case '/':return 2;break;
        case '+':return 1;break;
        case '-':return 1;break;
        default:return -1;//define the precedence of a number or a variable is the
lowest
    }
}

bool cmp(string s1,string s2){//the assisting function for quick_sort algorithm
    return s1<s2;
}

bool isnumber(string s){//determine whether s is a number
    for(int i=0;i<(int)s.length();i++)
        if((s[i]<'0')||(s[i]>'9'))
```

```
            return false;
    return true;
}

bool isoperator(string s){//determine whether s is an operator
    return (s[0]=='+')||(s[0]=='-')||(s[0]=='*')||(s[0]=='/')||(s[0]=='^');
}

TreeNode* createnode(string s){//function for creating a new treenode
    TreeNode *temp=new TreeNode;
    temp->value=s;
    temp->left=NULL;temp->right=NULL;
    return temp;
}

TreeNode* buildtree(string expression){//function for building a tree
    Stack nodestack,opstack;//nodestack stores variables,numbers;opstack stores operators
    nodestack.top=opstack.top=-1;//set the two tops
    string var;//collect the variable name/number
    for(int i=0;i<(int)expression.length();i++){
        if(expression[i]=='(')//if we encounter left bracket,push it into opstack for encountering right bracket
            opstack.push(createnode(string(1,expression[i])));
        else if(isalnum(expression[i]))//if expression[i] is a letter or number,collect it into var
            var+=expression[i];
        else if(expression[i]==')'){//if we encounter right bracket,then create expression tree in the brackets
            if(!var.empty())//push the new variable into the stack
            {
                nodestack.push(createnode(var));
                var.clear();//clear the string and start over
            }
            while((!opstack.empty())&&(opstack.visit()->value!="(")){//create until we encounter left bracket
                TreeNode *op=opstack.pop();//get the operator
                TreeNode *rightnode=nodestack.pop();//get the numbers
                TreeNode *leftnode=nodestack.pop();
                op->left=leftnode;op->right=rightnode;//create the tree
                nodestack.push(op);//push the new tree back into the stack
            }
            opstack.pop();//pop the left bracket
        }
        else{//if expression[i] is an operator,then get two numbers from nodestack and create the tree
            if(!var.empty()){
                nodestack.push(createnode(var));
                var.clear();
            }
            while((!opstack.empty())&&(precedence(opstack.visit()->value[0])>=precedence(expression[i]))){//if the precedence of previous operator is greater,handle the operator first
                TreeNode *op=opstack.pop();
                TreeNode *rightnode=nodestack.pop();
```

```
                TreeNode *leftnode=nodestack.pop();
                op->left=leftnode;op->right=rightnode;
                nodestack.push(op);
            }
            opstack.push(createnode(string(1,expression[i])));//push the new operator into opstack
        }
    }
    /*there is still a variable in var,a operator in opstack,a variable in nodestack,create the tree*/
    if(!var.empty()){
        nodestack.push(createnode(var));
        var.clear();
    }
    while(!opstack.empty()){
        TreeNode *op=opstack.pop();
        TreeNode *rightnode=nodestack.pop();
        TreeNode *leftnode=nodestack.pop();
        op->left=leftnode;op->right=rightnode;
        nodestack.push(op);
    }
    return nodestack.pop();//return the headnode
}

TreeNode* addnodes(TreeNode *left,TreeNode *right){//merge leftnode,rightnode and operator "+"
    if(left==NULL)//if left is NULL then there is no need to print "+"(in case "+a")
        return right;
    if(right==NULL)//if right is NULL then there is no need to print "+"(in case "a+")
        return left;
    TreeNode *temp=createnode("+");
    temp->left=left;temp->right=right;
    return temp;
}

TreeNode* subnodes(TreeNode *left,TreeNode *right){//merge leftnode,rightnode and operator "-"
    if(right==NULL)//if right is NULL then there is no need to print "-"(in case "a-")
        return left;
    TreeNode *temp=createnode("-");
    temp->left=left;temp->right=right;
    return temp;
}

TreeNode* mulnodes(TreeNode *left,TreeNode *right){//merge leftnode,rightnode and operator "*"
    if((left==NULL)||(right==NULL))//if right or left is NULL then the whole result is 0
        return NULL;
    TreeNode *temp=createnode("*");
    temp->left=left;temp->right=right;
    return temp;
```

```
}

TreeNode* divnodes(TreeNode *left,TreeNode *right){//merge leftnode,rightnode and operator "/"
    if((left==NULL)||(right==NULL))//if right or left is NULL then the whole result is 0
        return NULL;
    TreeNode *temp=createnode("/");
    temp->left=left;temp->right=right;
    return temp;
}

TreeNode* powernodes(TreeNode *left,TreeNode *right){//merge leftnode,rightnode and operator "^"
    TreeNode *temp=createnode("^");
    temp->left=left;temp->right=right;
    return temp;
}

TreeNode* differentiate(TreeNode *root,string var){
    if(root==NULL)
        return NULL;
    if(isnumber(root->value))//the derivative of constant number is 0
        return NULL;
    else if(root->value==var)//the derivative of the var is 1
        return createnode("1");
    else{//root->value is an operator
        TreeNode *leftchild=differentiate(root->left,var);//get the derivative of leftchild
        TreeNode *rightchild=differentiate(root->right,var);//get the derivative of rightchild

        if(root->value=="+")//F(x)=f(x)+g(x),F'(x)=f'(x)+g'(x)
            return addnodes(leftchild,rightchild);
        if(root->value=="-")//F(x)=f(x)-g(x),F'(x)=f'(x)-g'(x)
            return subnodes(leftchild,rightchild);
        if(root->value=="*")//F(x)=f(x)*g(x),F'(x)=f'(x)*g(x)+f(x)*g'(x)
            return addnodes(mulnodes(leftchild,root->right),mulnodes(root->left,rightchild));
        if(root->value=="/"){//F(x)=f(x)/g(x),F'(x)=[f'(x)*g(x)-f(x)*g'(x)]/g(x)^2
            TreeNode *term1=mulnodes(leftchild,root->right);
            TreeNode *term2=mulnodes(root->left,rightchild);
            TreeNode *term3=powernodes(root->right,createnode("2"));
            return divnodes(subnodes(term1,term2),term3);
        }
        if(root->value=="^"){//F(x)=f(x)^[g(x)],F'(x)=[g'(x)*lnf(x)+g(x)*f'(x)/f(x)]*f(x)^g(x)
            TreeNode *term1=mulnodes(rightchild,createnode("ln("+root->left->value+")"));
            TreeNode *term2=divnodes(mulnodes(root->right,leftchild),root->left);
            TreeNode *term3=powernodes(root->left,root->right);
            return mulnodes(addnodes(term1,term2),term3);
        }
    }
}
```

```
191
192 void printtree(TreeNode *root){//print the expression tree back to inorder
    expression
193     if(root==NULL)
194         return;
195     if(root->left!=NULL){
196         if(isoperator(root->value)&&isoperator(root->left->value)){
197             bool flag=precedence(root->value[0])>precedence(root->left->value[0]);
198             if(flag)//if root's operator is higher than root's leftchild's
    operator,then expression in leftchild need brackets
199                 cout<<"(";
200             printtree(root->left);
201             if(flag)
202                 cout<<")";
203         }
204         else
205             printtree(root->left);
206     }
207     cout<<root->value;
208     if(root->right!=NULL){
209         if(isoperator(root->value)&&isoperator(root->right->value)){
210             bool flag=precedence(root->value[0])>precedence(root->right->value[0]);
211             if(flag)//if root's operator is higher than root's rightchild's
    operator,then expression in rightchild need brackets
212                 cout<<"(";
213             printtree(root->right);
214             if(flag)
215                 cout<<")";
216         }
217         else
218             printtree(root->right);
219     }
220 }
221
222 int main(){
223     string expression;
224     cin>>expression;
225
226     /*build the expression tree*/
227     TreeNode* root=buildtree(expression);
228
229     /*count all the variable names*/
230     string variables[100];
231     string var;
232     int total=0;
233     for(int i=0;i<(int)expression.size();i++){
234         if(isalnum(expression[i]))
235             var+=expression[i];
236         else{
237             if(!var.empty()){
238                 bool flag=true;//flag=true means the new variable name is not in
    the old set of variable names
239                 for(int i=1;i<=total;i++)
240                     if(var==variables[i]){
241                         flag=false;
```

```
                        break;
                    }
                if(flag)//add the new variable name
                    variables[++total]=var;
                var.clear();
            }
        }
    }
    if(!var.empty()){
        bool flag=true;
        for(int i=1;i<=total;i++)
            if(var==variables[i]){
                flag=false;
                break;
            }
        if(flag)
            variables[++total]=var;
        var.clear();
    }
    /*sort all the variable name in lexicographical order*/
    sort(variables+1,variables+total+1,cmp);

    for(int i=1;i<=total;i++)
        if(!isnumber(variables[i])){
            cout<<variables[i]<<": ";
            TreeNode *ans=differentiate(root,variables[i]);//differentiate all
variables
            printtree(ans);
            cout<<endl;
        }
    return 0;
}
```

Declaration

I hereby declare that all the work done in this project titled “Autograd for Algebraic Expressions” is of my independent effort.