CSCE625: Introduction to Artificial Intelligence
Programming Assignment 1:
Simplifying Mathematical Expressions via Search

Report

Wei Qiao 522006310

In this programming assignment, I try to realize the automating process of simplifying symbolic mathematics using python. The main idea includes two steps: the first step is to use A* search with defined actions, goal and cost function to get an initial solution that only the "variable" is on the left side of the equation; the second step is to further simplify the equation by merging numbers and merging the same item.

In the following part, I will depict the realized algorithm in detail by explaining the steps one by one.

1) The first step: A* search

In the first step, I try to simply the input equation into the one that only the target variable is on the left side. It means we should move the other items to the right side of the equation.

I give an example here:

Assume an input equation is "x+z+2=1". This step aims to simply it into "x=1-2-z". That's the initial step of this realized algorithm.

Now I am going to explain how I did it. The algorithm I used here is A* search algorithm. By defining the actions we could do at a given state and the goal we want and the evaluation function, we could exploit A* algorithm to fulfill this process.

A* search is the most widely known form of best-first search. It evaluates nodes by combining $g(n)$, the cost to reach the current state , and $h(n)$, the estimated cost to get from the current state to the goal:

$$f(n) = g(n) + h(n)$$

Hence, we know $f(n)$ is the estimated cost of the cheapest solution though state n. $g(n)$ is the real cost we already take and $h(n)$ is the heuristic function to guide the path choice.

Given a state, we could have several actions to move to the next state. From these actions, we choose the one with the minimum $f(n)$. By using A* search algorithm, we finally get the goal state we want.

After simply explaining A* algorithm, I am going to introduce the actions, the goal and the estimated cost I set.

In this step, the actions are defined based on the current state as:

Table 1. Action table

Action No.	Definition
'BINARYOP'	
1	Exchange the left child node and the right child node.
2	If the node is "-" and it's a 'BINARYOP', move its right child node to the right side.
3	If the node is "+", move its right child node to the right side.
4	If the node is "+", move its left child node to the right side.
5	If the node is "/", move its right child node to the right side.
6	If the node is "*", move its right child node to the right side.
7	If the node is "*", move its left child node to the right side.
'UNARYOP'	
8	If the node is "-" and it's a 'UNARYOP', move its right child node to the right side.
'UNARYFUNCTION'	
9	If the node is "^", change it and its child, then move them to the right side.
10	If the node is "ln", change it and its child, then move them to the right side.
11	If the node is "sqrt", change it and its child, then move them to the right side.

For the cost function, we use the step number to the current state as the real cost $f(n)$ and the depth of the target variable as heuristic function $h(n)$.

Why I set the depth of the target variable as heuristic function is because the depth of the target variable is the lower bound on the cost to reach the goal we want. Meanwhile, it could really give us some inspiration to do search.

The goal I set is that finally only the target variable is on the left side of the root node "=".

By defining these things, we could exploit A* search to simplify the equation and get the initial simplified equation we want.

In fact, all the simplifying process could be down in this step. But it's difficult to define all the actions we want. Defining a complete action space is too time-consuming. Hence, I decided to just use A* search to get an initial result which we can further work on to get the final result.

2) The second step: further simplifying

In this step, I use recursion and back-up algorithm to do further simplification. The idea is try to decide which simplifying action is taken for the current state.

There are several criterions used in this algorithm. Those are:

Criterion.1.

Given a node, if its left child tree or right child tree has no variable, then calculate the corresponding value and use this value to generate a new node without children to replace the original entire child tree. It one of its child trees has a variable, then do recursion on this child node and repeat the algorithm.

Its graphical explanation is shown as Fig.1.

Criterion.2.

This kind of criterions is operated on two nodes, *i.e.*, a node and one of its children nodes. By collecting the information of the two nodes, we could judge whether there are two values (numbers) attached to these two nodes. If there exist two numbers, we could further merge them into one and then we should modify the tree after we merge the numbers.

Its graphical explanation is shown as Fig.2.

WHAT should be noticed here, I just list out 3 kinds of the possible situations. It has 10 possible situations. You can see the detail in the attached python file.

After using this criterion to update the state, we could obtain a new state. Then, we do the recursion on this new state.

By combining these two criterions into one algorithm, we could finally get the recursion algorithm we want. You can see the python file for the detail of this algorithm. It's very intuitive. Meanwhile, if we found some zero division error in the calculation, we will output x as undefined.

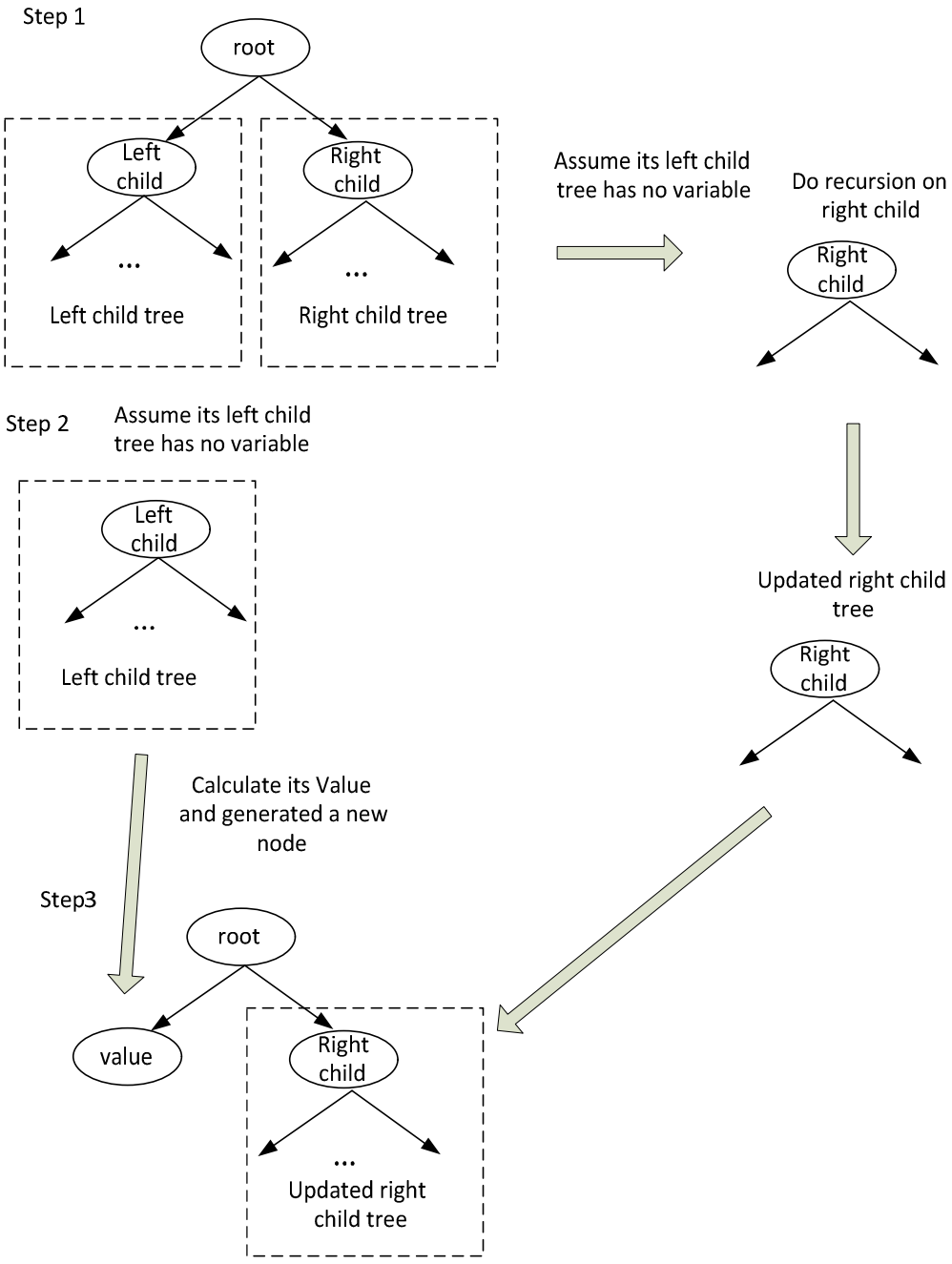

Fig.1. The simplifying action based on one node

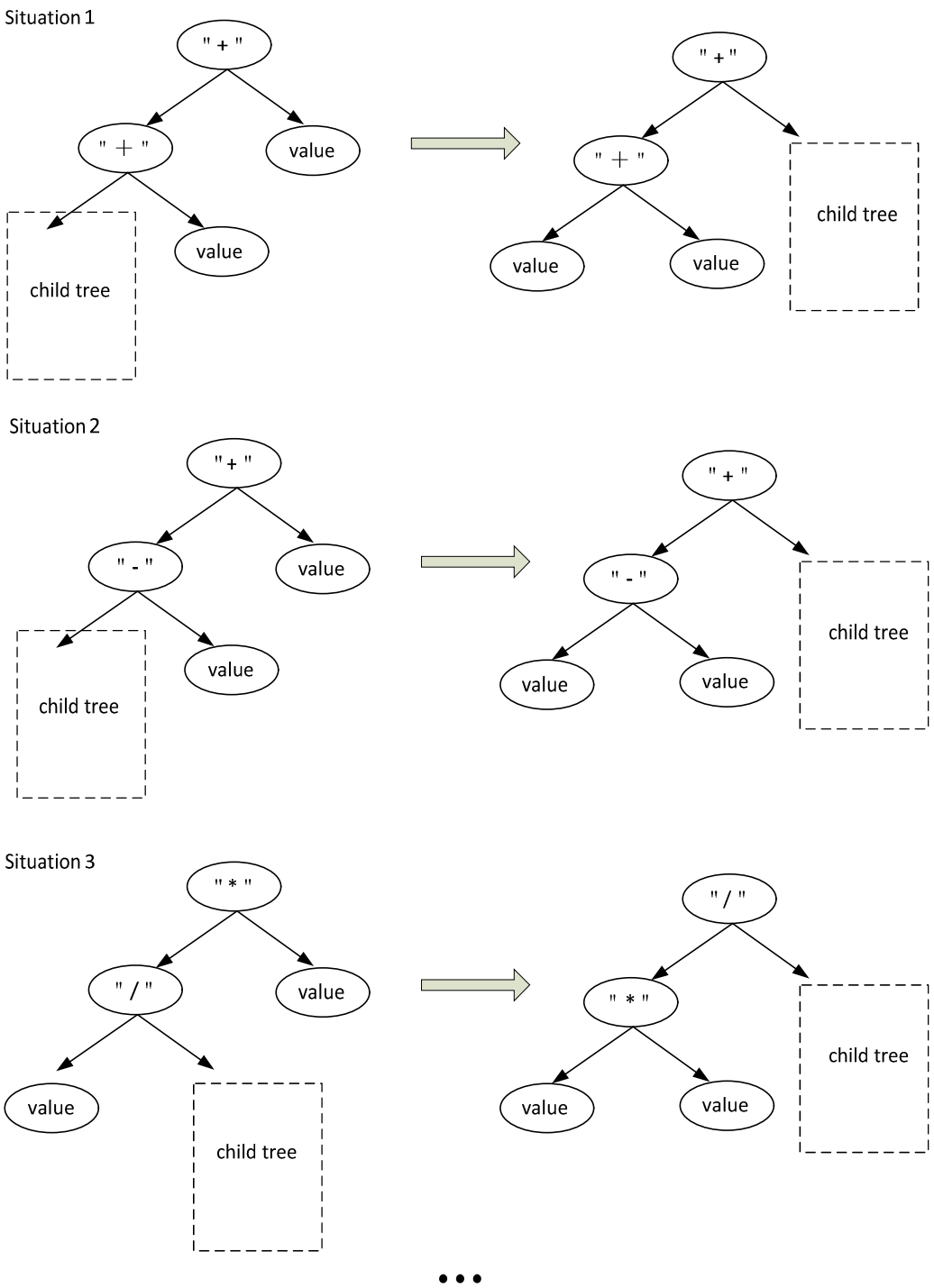

Fig.2. The simplifying action based on two nodes

Results

Now, we give several examples to illustrate the process of the realized symbolic mathematics simplifying algorithm. The examples are shown as follows:

Ex.1 :

x = (2 + 10) * (2^2)

```
qiao-ubuntu@qiaoubuntu-ThinkPad-T430:~/AI$ python equationsimplier.py
eq > x=(2+10)*(2^2)
var > x
This is parsed at: (= x (* (+ 2 10) (^ 2 2)))
The result of the 1st step is:x = (2 + 10) * 2^2
The final result is:x=48
```

Ex.2 :

x = 6 * 2 / (-1 + 4 * 0 + 1)

```
qiao-ubuntu@qiaoubuntu-ThinkPad-T430:~/AI$ python equationsimplier.py
eq > x=6*2/(-1+4*0+1)
var > x
This is parsed at: (= x (/ (* 6 2) (+ (+ (- 1) (* 4 0)) 1)))
The result of the 1st step is:x = 6 * 2 / (-1 + 4 * 0 + 1)
The final result is:x = undefined
```

Ex.3 :

(2 * sqrt(x) * 3) - y = pi

```
eq > (2*sqrt(x)*3)-y=pi
var > x
This is parsed at: (= (- (* (* 2 (sqrt x)) 3) y) pi)
The result of the 1st step is:x = ((pi + y) / 3 / 2)^2
The final result is:x=((pi+y)/6)^2
```

Ex.4 :

(2 * x * 3 * y * 4 * z * 5 * 6) = 800

```
qiao-ubuntu@qiaoubuntu-ThinkPad-T430:~/AI$ python equationsimplier.py
eq > (2*x*3*y*4*z*5*6)=800
var > x
This is parsed at: (= (* (* (* (* (* (* 2 x) 3) y) 4) z) 5) 6) 800)
The result of the 1st step is:x = 800 / 6 / 5 / z / 4 / y / 3 / 2
The final result is:x=((6/z)/y)/6
```

Ex.5 :

sqrt(x)+y-1=2

```
eq > sqrt(x)+y-1=2
var > x
This is parsed at: (= (- (+ (sqrt x) y) 1) 2)
The result of the 1st step is:x = (2 + 1 - y)^2
1
The final result is:x=((3-y)^2)
```

Ex.6 :

sqrt(x)+1 =2

```
eq > sqrt(x)+1=2
var > x
This is parsed at: (= (+ (sqrt x) 1) 2)
The result of the 1st step is:x = (2 - 1)^2
The final result is:x=1
```

Ex.7 :

sqrt(x)+6*y-2*2=1

```
qiao-ubuntu@qiaoubuntu-ThinkPad-T430:~/AI$ python equationsimplier.py
eq > sqrt(x)+6*y-2*2=1
var > x
This is parsed at: (= (- (+ (sqrt x) (* 6 y)) (* 2 2)) 1)
The result of the 1st step is:x = (1 + 2 * 2 - 6 * y)^2
The final result is:x=((5-(6*y))^2)
```

Ex.8 :

sqrt(x)+6*y=pi

```
eq > sqrt(x)+6*y=pi
var > x
This is parsed at: (= (+ (sqrt x) (* 6 y)) pi)
The result of the 1st step is:x = (pi - 6 * y)^2
The final result is:x=(pi-(6*y))^2
```

Ex.9 :

x-1=sin(1)+z

```
eq > x-1=sin(1)+z
var > x
This is parsed at: (= (- x 1) (+ (sin 1) z))
The result of the 1st step is:x = sin(1) + z + 1
The final result is:x=1.84147098481+z
```

References:

[1] PLY: PLY (Python Lex-Yacc) an implementation of lex and yacc parsing tools for Python. http://www.dabeaz.com/ply/

[2] AIAMA-Python: Python implementation of algorithms from Russell and Norvig's "Artificial Intelligence: A Modern Approach" https://code.google.com/p/aima-python/

[3] Artificial Intelligence: A Modern Approach 3rd Edition by Stuart Russell and Peter Norvig, 2009.

An Aggie does not lie, cheat or steal or tolerate those who do.