
24 点游戏的算法与实现

许杨春

(杭州 311100)

摘要:24 点是以扑克为道具的数学游戏，给定四张牌，要求给出一个等于 24 的四则运算表达式的。其算法实现要求答案要完备且不重复。本文提出了一个思路，并 Python 3 做了实现。

关键词: 24 点 算法 Python

The algorithm & code of Poker 24 game

Xu YangChun

(Hangzhou, 311100)

【Abstract】 Poker 24 is a math game to figure out a algebraic expression equal to 24 win . The difficulty of the algorithm is how to reduce the equivalence and duplicate. This article give out a the implementation in Python 3.

【Key words】 Poker 24 algorithm Python

24 点是以扑克为道具的数学游戏，给定四张牌，通常不是用 J, Q, K, A, 也就是只使用 2 到 10 的数字，给出一个等于 24 的四则运算表达式。

24 点也是程序员喜欢尝试的程序，网上例子或文章很多，基本特点是：用穷举法来试错，效率不高；大案重复的问题没有很好解决或以只输出一个表达式的方式忽略；数据结构比较教条，运算树就用教科书的标准方法来表示。

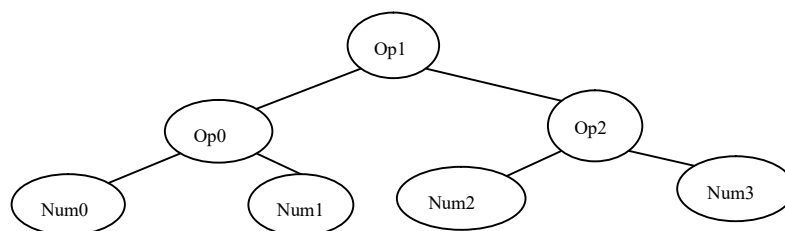
本文尝试先从运算树的角度来解决完备和等价（重复）的问题，推导出可能的运算树类型，避免了穷举和重复的问题。数据结构上，运算树用逆波兰式表示，这样可以直接存在一个列表/数组中，不需要左右子树指针来表示，这样不但数据结构简洁，而且程序实现也方便。

1 运算树

按[1],所有表达式都可以用一个运算树来表示。4 个操作数就意味这树只有 4 片叶子。这种树有多少类型呢?按相关定律：任何一棵二叉树 T ，如果其终端结点数为 n_0 ，度为 2 的结点数为 n_2 ，则 $n_0 = n_2 + 1$ 。由于四则运算都是二元运算，需要的运算符个数为 n_2, n_0 代表的叶子数目为 4, 则 $n_2 = n_0 - 1 = 3$ 。

1.1 可能的运算树类型

这么一个 4 片叶子，3 个根节点的运算树有几种类型呢。如果是平衡树，按定义左右子树的深度差不超过 1, 由于只有 4 片叶子，所以只能如下一种类型, 左右子树深度都为 2，它也是一满二叉树，标记为 type 1。

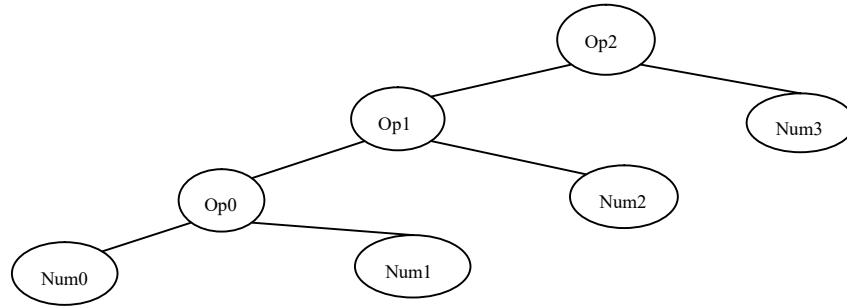


Type 1: 平衡运算树

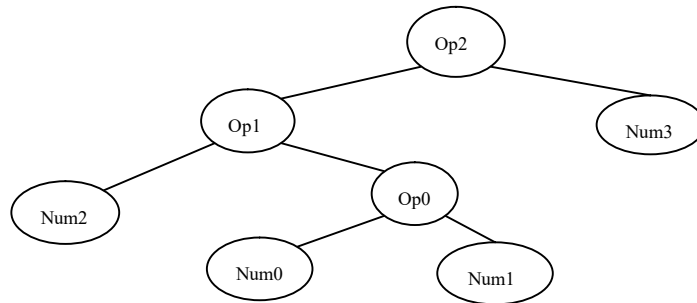
非平衡树，由于用二元运算符的，左右子树不会为空，先考虑右子树是一片叶子的情况, 左子树还可以分为两种情况，其左右子树分别为为 (2, 1) 的 type 2，(1, 2) type 3。

许杨春 (1972 年出生) 男 系统分析师 从事通讯软件开发工作

E-mail: xuyc@sina.com

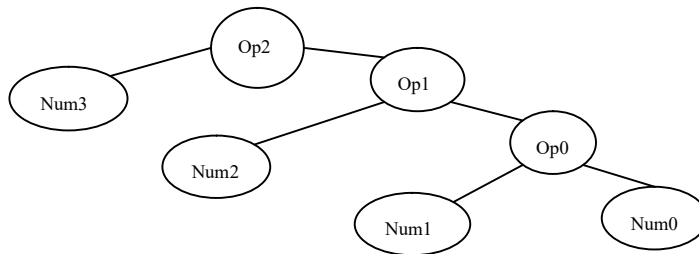


Type 2: 非平衡运算树

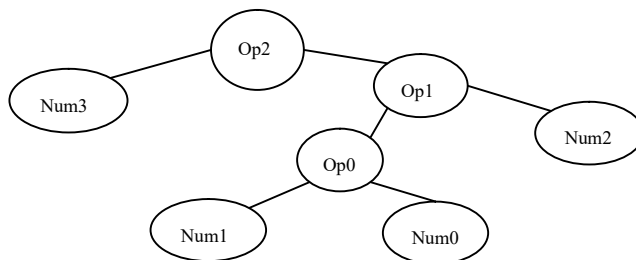


Type 3 非平衡运算树

类似的得到左子树是一片叶子的 type 4 ,type 5 运算树。



Type 4 非平衡运算树



Type 5 非平衡运算树

如上，5 种运算树就表示出所有有效的表达式类型。当用户输入 4 个数字后，只要对数字进行排列，运算符

(op0,op1,op2)在四则运算符中任选，然后填入对应的运算树节点，这样形成的表达式的候选集就是完备的。

1.2 运算树的等价问题

等价问题可以在运算树加入候选集时就进行检查。由于程序是按按 type1,type 2, type 3,type 4,type 5 的顺序执行的，所以 type 1 可以只考虑内部的树是否有等价问题，type 2 则考虑考虑是否与 type 1 的某棵树等价，type 3 则考虑是否与 type 1,type 2 的某棵树等价，依次类推。

- type 1: 如果 (子) 树的运算符是 '+' 或 '*', 按照交换率, 可以只考虑升序 (包括相等) 的运算树, 因为类似 3+4 与 4+3 实质上完全等价的。由于这一步工作适合在表达式 (第二节) 中实现, 所以这一步这里先跳过。
- type 2: 如果 op1,op2 都为 '+' 或都为 '*', 则运算树对应的表达式为 $V+num2+num3$ 或 $V*num2*num3$, 他们等价于 $V+(num2+num3)$, $V*(num2*num3)$, 也就是 type 1 的形式。这里 V 表示的是此前运算的中间值。
- type 3: 类似, 如果 op0,op1 都为 '+' 或都为 '*', 则其表达式等价于就是 type 2 的形式。
- type 4: 如果 op2 是 '+', '*', 由于满足交换律, 左右子树可以互换, 而互换后就等价于 type 2 中某棵树。
- type 5: 类似, 如果 op0,op1 都为 '+' 或都为 '*', 则其表达式等价于就是 type 4 的形式。

代码中这部分工作主要在函数 form_polish_expr () 内实现。

1.3. 树的数据结构

常规而应, 树需要用一个根节点加上左右子树指针来表示, 但本文中树的只是用来设计和证明算法的完备性的。由于最终目的是根据输入构造表达式, 然后求值, 如果符合条件输出表达式的。而所有的表达式都可以用逆波兰式, 即树的后序遍历得到。对上图的各种树作后序遍历后有:

Type 1 对应 num0,num1,op0,num2,num3,op2,op1

Type 2 对应 num0,num2,op0,num2,op1,num3,op2

Type 3 对应 num2,num0,num1,op0,op1,num3,op1

Type 4 对应 num3,num2,num1,num0,op0,op1,op2

Type 5 对应 num3,num1,num0,op0,num2,op1,op2

如果 C 语言中, 以上的树结构可以用数组来表示, 在 Python 中就用 list 表示即可,

([n[0], n[1], o[0], n[2], n[3], o[1], o[2]])

2. 表达式的求值和筛选

如果按编译原理的教材, 表达式求值用堆栈实现, Python 中 list 就可以作为堆栈使用, 同时它还可以看成队列, 使用十分方便。

筛选首先是排除其值不等于 24 的, 其次要删除重复的。所谓表达式重复是从人的理解角度来看:

1.3+4 与 4+3 从一般人的认识来说, 按交换律是等价的。解决方法是: 在表达式求值过程中增加升序 (包括相等) 的条件, 剔除不符合条件的 4+3。

2. 4- (4-2), 从计算机内部来说这两个 4 是不同的, 但从人的角度来说是完全一致, 也许要作为重复剔除。这在 python 中可以通过 Set 实现, 如下代码就删除全局变量 EXPR_LIST (用来存储所有符合要求的表达式的字符串形式, 如 '4*(5-2+3)') 中的重复元素。

```
EXPR_LIST = list(set(EXPR_LIST))
```

该方法的一个副作用是解决了 1 中操作数相等的情况, 如 4+4, 并不能用升序或降序的排除特定的一个, 只能通过这一步删除重复的一个。

3. 表达式输出

许杨春 (1972 年出生) 男 系统分析师 从事通讯软件开发工作

E-mail: xuyc@sina.com

也就是将逆波兰式(如上所述, 存在一个 list 中) 转换为对应的字符串形式表达式, 如 4,5,2,-,3,+,*
=>4*(5-2+3)。笔者是用树的归并算法来实现。先在 list 中往后扫描, 扫描到操作符 ‘-’ 后, 再往回找它对应的左右子树, 分别是 5 和 2, 然后合并得到一个子表达式 ‘5-2’, 然后将该表达式作为一个子树放回 list, 这时它的值变为: 4, ‘5-2’,3,+,*, 下一步归并‘5-2’,3,+, 得到 ‘5-2+3’, 就这样循环归并左右子树, 直到最后 list 中只剩下最终表达式: 4*(5-2+3)。同时需要考虑是否需要加括号的问题, 实现代码如下:

Python 的 list 不同于 C 等语言的数组, 它不限制元素类型, 则对编程带来了便利。

4 代码实现

```
import itertools
POLISH = []
PRIO = {'+': 1, '-': 1, '*': 2, '/': 2, '#': 0}
def form_polish_expr(o, l_nums):
    for n in l_nums:
        POLISH.append([n[0], n[1], o[0], n[2], n[3], o[1], o[2]])
        if [o[1], o[2]] not in ['+', '+'], ['*', '*']:
            POLISH.append([n[0], n[1], o[0], n[2], o[1], n[3], o[2]])
        if [o[0], o[1]] not in ['+', '+'], ['*', '*']:
            POLISH.append([n[0], n[1], n[2], o[0], o[1], n[3], o[2]])
        if o[2] not in ['+', '*']:
            POLISH.append([n[0], n[1], n[2], n[3], o[0], o[1], o[2]])
        if [o[0], o[1]] not in ['+', '+'], ['*', '*']:
            POLISH.append([n[0], n[1], n[2], o[0], n[3], o[1], o[2]])
    return

def cal_polish(ll):
    ss = []
    for c in ll:
        if c in ['+', '-', '*', '/']:
            n1 = ss.pop(0)
            n0 = ss.pop(0)
            if n0 > n1 and c in ['+', '*']:
                raise Exception("skip due to commutation")
            elif c == '+':
                val = n0 + n1
            elif c == '-':
                val = n0 - n1
            elif c == '*':
                val = n0*n1
            elif c == '/':
                val = n0/n1
            else:
                raise Exception("unexpected op")
```

许杨春 (1972 年出生) 男 系统分析师 从事通讯软件开发工作

E-mail: xuyc@sina.com

```
        ss.insert(0, val)
    else:
        ss.insert(0, c)
    return val
```

```
def polish_to_expr(pol):
    pol.append('#')
    l_idx_op=[]
    for i, c in enumerate(pol):
        if c in ['+', '-', '*', '/', '#']:
            l_idx_op.append(i)
    for i,idx in enumerate(l_idx_op):
        if pol[idx]=='#':
            break
        if l_idx_op[i+1]-idx > 2: # tree type 1 only
            idx_next = l_idx_op[i+2]
        else:
            idx_next = l_idx_op[i+1]
        prio,prio_next= PRIO[pol[idx]],PRIO[pol[idx_next]]
        right = 0
        for i in range(0,idx):
            if pol[i] not in ['+', '-', '*', '/', '$']:
                left,right=right,i
        if prio < prio_next:
            braced = True
        elif prio > prio_next:
            braced = False
        else:
            if [pol[idx],pol[idx_next]] in [['+', '+'], ['-', '+'], ['*', '*']]:
                braced = False
            elif [pol[idx],pol[idx_next]] in [['-', '-'], ['+', '-'], ['*', '/']]:
                braced = (idx_next - idx ==1)
            else:
                braced = True
        if braced:
            expr = ['(', str(pol[left]), pol[idx],str(pol[right]), ')']
        else:
            expr = [str(pol[left]), pol[idx], str(pol[right])]
        expr = ".join(expr)
        pol[idx], pol[left], pol[right] = expr, '$', '$'
    return expr
```

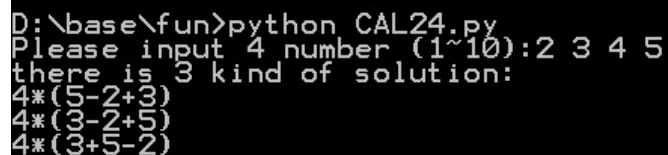
```
l = input("Please input 4 number (1~10):").split()
```

```

l_nums = list(set(itertools.permutations([int(x) for x in l])))
for op1 in ['+', '-', '*', '/']:
    for op2 in ['+', '-', '*', '/']:
        for op3 in ['+', '-', '*', '/']:
            form_polish_expr([op1, op2, op3], l_nums)
l_expr = []
for l in POLISH:
    try:
        if 24 == cal_polish(l):
            expr=polish_to_expr(l)
            l_expr.append(expr)
    except:
        continue
l = len(l_expr)
if l > 0:
    l_expr = list(set(l_expr))
    print("there is %d kind of solution:" % (l))
    for ll in l_expr:
        print(ll)
else:
    print("no solution")

```

执行示例：



```

D:\base\fun>python CAL24.py
Please input 4 number (1~10):2 3 4 5
there is 3 kind of solution:
4*(5-2+3)
4*(3-2+5)
4*(3+5-2)

```

4 小结

笔者以前有用 C 和 C++ 实现过 24 点游戏，代码都在 300 行以上。这次 Python 用不到 100 行的代码就达到目的。Python 的优点不仅仅在于自带 list, set 等数据类型，涉及数字计算也非常智能，如表达式 $(1/3) * 6$ ，c 语言就需要考虑浮点数 $1/3$ 乘以 6 后是否会等于 2，Python 中 这些都不需要考虑。

参考文献：

[1] 严蔚敏 吴伟民 数据结构 清华大学出版社 1992.