作业 4 问题回顾

比特币复杂度、中缀表达式转前缀、HTML 括号匹配

李睢

sui@pku.edu.cn

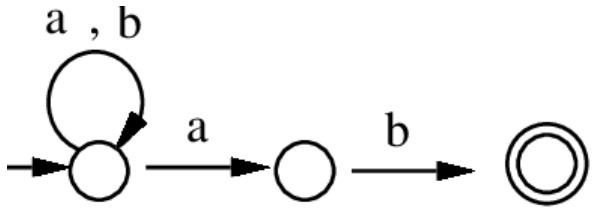
Apr 20, 2021

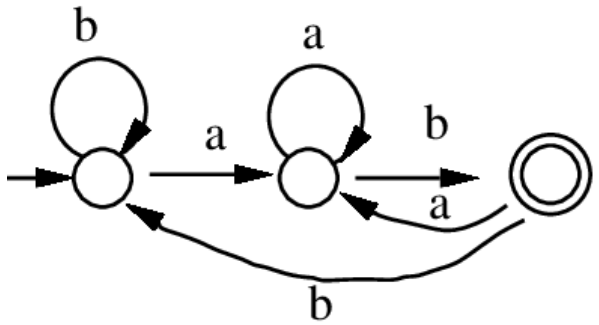
目录

1. 算法复杂度、P 与 NP
   - 理解 NP 问题
   - 作业详解 – 比特币挖矿问题
2. 栈的应用
   - Tips – 栈和队列的实现、输入输出格式、编程习惯
   - 作业详解 – 中缀表达式转前缀
   - 作业详解 – HTML 括号匹配
3. 附录：同学们在作业 1 和作业 4 中的亮点

理解 NP – 从非确定性图灵机的角度

- 背景：很多计算问题都可规约为判定问题，因此主要关注判定问题
- P：多项式时间可判定 **P**olynomial Time
- NP：非确定性多项式时间可判定 **N**ondeterministic **P**olynomial Time
  - 非确定性图灵机（**N**TM）可以以多项式时间判定 的问题
    - 接收同样的输入时可以同时进行多种不同的状态转移，可以同时处于多个状态
    - 当前处于的状态中只要有一个状态是接受状态，整个计算就接受
  - 等价定义：当答案为“是”时，存在一个证书（“短证明”）可以被 确定性 图灵机（DTM）以多项式时间验证 的问题





验证字符串是否以“ab”结尾的自动机

- **N**FA：非确定性有限状态自动机
- DFA：确定性有限状态自动机

NP 问题举例 – 布尔可满足性问题（SAT）

布尔类型变量 $x_1, x_2, ..., x_n$ 的取值均为 True 或 False。给定一个布尔表达式 $F(x_1, ..., x_n)$，是否存在一组 $(x_1, ..., x_n)$，使得 $F(x_1, ..., x_n) = True$?

- 例：$F(x1, x2, x3) = (x_1 \vee \neg x_2) \wedge (\neg x_1 \wedge x2 \vee x3) \wedge \neg x_1$，存在一组取值 $x_1 = False, x2 = False, x3 = True$ 使 F 为真
- 为什么考虑这个判定问题？计算问题“求一组 $(x_1, ..., x_n)$，使得布尔表达式 $F(x_1, ..., x_n)$ 的值为真”可规约为这个判定问题
  - 假设 $x_1 = True$, 是否存在一组 $(x_2, ..., x_n)$，使得布尔表达式 $F(True, x_2, ..., x_n)$ 的值为真?
  - 若存在，则 $x_1 = True$ 否则 $x_1 = False$。重复该过程，就得到了原计算问题的解
- 可以被 NTM 在 $O(n)$ 时间判定：
  - 随机选择 x_1，...，随机选择 x_n。计算 $F(x_1, ..., x_n)$，若结果为 True 则接受，否则拒绝。（NTM 只要有一个可能状态为接受，则整个计算结果为接受。）
- 当答案为“是”时，存在一个“短证明”$(x_1, ..., x_n)$ 可以被 DTM 在 $O(n)$ 时间验证：
  - 计算 $F(x_1, ..., x_n)$，若结果为 True 则接受，否则拒绝。

比特币挖矿的算法复杂度 – 问题描述

- `validate()` 和 `find_nonce()` 的复杂度分别是多少？它们是什么关系？分别属于 P 还是 NP?
- 如果想把“Alice send .1 coins to Kate”改成“Alice send 100 coins to Kate”，计算一个新的 nonce 来保持“块哈希”值不变，算法复杂度是多少？（自行假设 validate 的每秒运行次数，估计所用时间）

比特币挖矿的算法复杂度 – 问题描述

validate()

```
import hashlib
def bthash(unicode):
    return
     ↪ hashlib.sha256(unicode.encode("utf8")).hexdigest()
difficulty_bits = 4
difficulty = 2 ** difficulty_bits
target = 2 ** (256 - difficulty_bits)

# 核心问题，求解 nonce，满足条件: Curr_hash =
 ↪ hash(transactions + nonce + Prev_hash) < target
def validate(transactions, nonce):
    block = transactions + [nonce] + [" 前一个块哈
     ↪ 希:13b1b06...76d3d"]
    hash_result = bthash(str(block))
    return int(hash_result, 16) < target
```

比特币挖矿的算法复杂度 – 问题描述

find_nonce()

```
def find_nonce(transactions):                    # 挖矿过程
    ntries = 256
    for nonce in range(ntries):                  # 随机数
        if validate(transactions, nonce):
            print("A new block mined with nonce
            ↪  {}".format(nonce))
            return nonce
    else:
        print("Failed in ntries:{}".format(ntries))
        return None
```

validate 和 find_nonce 的复杂度

答案

- validate 的复杂度为 $O(n)$，其中 n 为 transactions（中所有字符串）的长度，因为 sha256 算法的复杂度是 O(输入长度)
- find_nonce 的复杂度为 $O(n \times 2^m)$，其中 m=difficulty_bits，因为 find_nonce 平均需要调用 2^m 次 validate 才能挖到矿而终止

常见错误

- 忽略了 sha256 算法的复杂度
  “*validate* 是 $O(1)$ 的”
- 不说明字母 n 指代的是什么
  “*validate* 复杂度为 $O(n)$, *find_nonce* 复杂度为 $O(n)$”
- 没有理解问题 / 没有理解“常数”
  - “因为有上限次数 ntries=256，所以 find_nonce 是 $O(1)$ 的”
  - “因为 difficulty_bits=4，所以 find_nonce 是 $O(1)$ 的”
  - “因为 256bit 是有限的长度，2^{length} 看起来即时再大也是常数！所以 find_nonce 是 $O(1)$ 的”

validate 和 find_nonce 是 P 还是 NP，二者关系

- **判定问题**：是否存在一个 *nonce*，使得 $sha256(transactions + nonce + Prev\_hash) < 2^{(256-difficulty\_bits)}$？
  - find_nonce 是与之对应的**计算问题**
  - nonce 是这个判定问题的**证书**（“短证明”），也是计算问题的**解**
  - validate 是这个判定问题的证书（计算问题的解）的**验证机**

答案

- `validate` 属于 P，`find_nonce` 属于 NP
- `validate` 是 **验证** `find_nonce` 解的正确性的方法

常见错误

- 概念不清

篡改交易记录后计算一个新的 nonce 来保持“块哈希”值不变所需要的时间

- 题目背景：包含“Alice send .1 coins to Kate”的块已经计算完成，满足 $sha256(transactions + nonce + Prev\_hash) < 2^{(256-difficulty\_bits)}$ 的 nonce 值已经确定，该块的哈希值已经广播给所有人。
- 此时恶意攻击者想把上述记录改成“Alice send 100 coins to Kate”，从而把对方的额外 99.9 个币划归自己名下。那么他需要计算出一个新的 nonce，使得新的块哈希与已经被广播的那块的块哈希的 256 bit 完全一致（不是小于 target），并把这块广播给所有人

答案

平均需要 $2^{256} \times t$，其中 t 是平均运行一次 validate 的所需的时间

常见错误

- “代码中的 block 不包括 nonce 的部分一共有 204 个字符，修改后有 205 个字符，因此每次 validate 运行时间的时间增加了约 1/200”
- ~~“不妨设每秒可以运行 2^{128} 次 validate...”~~

FAQ: 实际刷题/作业中需要自己实现栈和队列吗?

- “不要重复发明轮子”
- 用 python 自带的工具实现栈和队列
  - 栈：`list` 底层由数组实现，向末尾添加和删除元素是 $O(1)$ 的

```
stack = []              # []
stack.append('a')       # ['a']
stack.append('b')       # ['a', 'b']
v = stack.pop()         # ['a']
```

  - 队列：`collections.deque` 底层由双向链表实现，向两端添加和删除元素都是 $O(1)$ 的

```
from collections import deque
queue = deque()         # []
queue.append('a')       # ['a']
queue.append('b')       # ['a', 'b']
v = queue.popleft()     # ['b']
```

注意：`list.insert(0, v)`，`list.pop(0)` 和 `list[1:]` 都是 $O(n)$ 的，慎用 list 模拟队列!

Tips – 关于作业文件名、输入输出格式的问题

- 因为助教会写脚本来辅助批改作业，如果文件名和输入输出格式不对的话会导致脚本无法得到正确的结果
- 除了要求的输出外，请不要输出任何额外的内容。提交前请测试自己的代码保证能接收规定的格式的任何输入，并以规定的格式输出。若有任何额外的说明或感想等请写在代码的注释中。

常见错误

- 作业 1 文件名"iknows.txt"：写成"iknow.txt" "iknwos.txt" "iknows.txt.txt"
- `input(" 请输入中缀表达式：")`
- `print(html)`
- 输出大小写错误"Yes", "No"
- 输出格式错误"True", "False"

Tips – 养成良好的编程习惯和代码风格

- 时刻注意自己代码的复杂度，避免不必要的高复杂度操作
- 尽量避免使用全局变量（易冲突、难懂、难以 debug...）
- 一个变量只做一件事，不要更改变量的定义
- 避免复制粘贴重复的代码（要定义成函数）
- 易懂的变量命名，不要重名
- 合适的注释

常见错误

- `html = html[html.index('>'):]`
- `global i`
- `stack.insert(-2, op) ; new_num = ' '.join(stack[-3:])`
- `input=input()`

中缀表达式转前缀 – 问题描述

- 实现一个“中缀转前缀”算法。
- 可能需要从右往左扫描，可能需要多次全量反转顺序。
- 文件名：`in2pre.py`
- 输入格式：从标准输入读取一行中缀表达式，格式如课件所述，字符之间由空格分隔，操作符包括 +-*/和小括号，操作数为单个字母。
- 输出格式：输出一行，为转换后的前缀表达式，字符间由空格分隔，其中操作数顺序与必须与输入中缀表达式的顺序相同。（答案唯一，不许使用交换律）
- 样例输入：(A + B) * (C + D)
- 样例输出：* + A B + C D

中缀表达式转前缀 – 测试样例

1. (((((((((A))))))))))
2. A + B * C - D / E
3. (A + B) * (C - D)
4. (A * B + C * D) / (E * F - G / H)
5. A - B * C * D * E * F * G
6. A + B / (C - D / (E + F / (G - H / I)))
7. (X - X) * (X + X) / (X - X)
8. A + B / (C + D * (E + F * U + V) / W + X) * Y + Z
9. (A * B + C / D) * (E / F - G * H) - (I * J + K / L) * (M / N - O * P)
10. A + B * C + B * C + B * C + B * C + B * C + B * C

中缀表达式转前缀 – 思路

例子

- 中缀：A+B+(C+D)*E
- 后缀：AB+CD+E*+
- 前缀：++AB*+CDE

- 中缀：A+B
- 后缀：AB+
- 前缀：+AB

- 中缀：A+B+C
- 后缀：AB+C+
- 前缀：++ABC

思路

- 初始猜想：后缀表达式的输出左右翻转，就是前缀表达式?
- 问题 1：左右运算数顺序颠倒，“+AB” vs “+BA”
- 修改 1：从右向左读取表达式（括号相应翻转看待）
- 问题 2：同级算符优先级颠倒，“++ABC” vs “+A+BC”
- 修改 2：更改算符弹栈的逻辑，使得遇到同优先级的算符不弹栈

中缀表达式转前缀 – 算法

算法框架

中缀转后缀

- 从左往右扫描输入
  - 遇到操作数：输出
  - 遇到左括号：入栈
  - 遇到右括号：输出栈顶算符并弹栈直到栈顶为左括号，左括号弹栈
  - 遇到运算符：输出栈顶算符并弹栈直到栈顶优先级低于当前算符，当前算符入栈
- 输出栈顶算符并弹栈直到栈空

中缀转前缀

- 从右往左 扫描输入
  - 遇到操作数：输出
  - 遇到右括号：入栈
  - 遇到左括号：输出栈顶算符并弹栈直到栈顶为右括号，右括号 弹栈
  - 遇到运算符：输出栈顶算符并弹栈直到栈顶优先级低于或等于 当前算符，当前算符入栈
- 输出栈顶算符并弹栈直到栈空
- 左右翻转输出

中缀表达式转前缀 – 参考代码 I

```
tokens = input().strip().split(' ')
stack = []
output = []
op_dict = {'+': 0, '-': 0, '*': 1, '/': 1}

for c in reversed(tokens):
    if c == ')':
        stack.append(c)
    elif c == '(':
        op = stack.pop()
        while op != ')':
            output.append(op)
            op = stack.pop()
```

中缀表达式转前缀 – 参考代码 II

```
    elif c in op_dict:
        while stack and stack[-1] in op_dict and
        ↪  op_dict[stack[-1]] > op_dict[c]:
            op = stack.pop()
            output.append(op)
        stack.append(c)
    else:
        output.append(c)

while stack:
    op = stack.pop()
    output.append(op)

print(' '.join(reversed(output)))
```

中缀表达式转前缀 – 错误

常见错误

- 同优先级运算符没有从左到右计算
- list 拼接是 $O(n)$ 操作，不可频繁使用

  `prefix_list = [operator] + prefix_list`
- list.insert(0) 是 $O(n)$ 的操作，不可频繁使用

  `prefix_list.insert(0, operator)`
- 反复找优先级最低的操作符，递归计算其两个操作数。$O(n^2)$
- `if token in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ':`

  复杂度过高，如果一定要这样，请用

  `if token.isalpha():`

HTML 括号匹配 – 问题描述

- 读入一个 HTML 文件，算法检查是否有标记不匹配的情况。
- 文件名：`html.py`
- 输入格式：从标准输入读取 html 格式文本，其中每组成对的标记名称均为字母/数字组合，可能包含多行，可能包含多余的空格和空行
- 输出格式：若输入的 html 括号匹配是合法的，输出 yes，否则输出 no

```
<html>
    <head>
        <title>
            Example
        </title>
    </head>
    <body>
        <h1> Hello,
        ↪  world</h1>
    </body>
</html>
```

HTML 括号匹配 – 测试样例 I

1. `<html></html>`
2. `<<<<html>>>></html>`
3. `<html<<<<></html>`
4. `<html><ahtml>`
5. `<html></html1>`
6. `<html><head><title>Example</title></head><body><h1>`
   ↪ `Hello, world </h1></body></html>`
7. `<html>  <head><title>Example`
   ↪ `</title></head><body><h1> Hello, world </h1></body>`
   ↪ `</html>`
8. `<html><head><title>Example</title></head><body><h2>`
   ↪ `Hello, world </h1></body></html>`
9. `</html><head><title>Example</title></head><body><h1>`
   ↪ `Hello, world </h1></body><html>`

HTML 括号匹配 – 测试样例 II

10 `<html><head><title>Example<title></head><body><h1>`
`↪ Hello, world </h1></body></html>`

11 `<!DOCTYPE html><html lang="en"><head><title>Sample`
`↪ page</title></head><body><h1>Sample`
`↪ page</h1><p>This is a <a`
`↪ href="demo.html">simple</a> sample.</p><!-- this is`
`↪ a comment --></body></html>`

HTML 括号匹配 – 思路

工程问题，算法简单但细节繁琐。具体内容大致可分解为：

- 空格和空行的处理
- 尖括号处理
  - 判断尖括号是否有不合理、不匹配的情况
  - 正确提取尖括号内的标签名、斜杠等标识符
- HTML 的属性、注释、特殊标签的处理（奖励分）
- 开闭标签的匹配

推荐做法

将问题分解为多个抽象层次，每一次集中在一个特定的层次上操作。
例如以下是编译器的常见架构：

- 词法分析：把字符串分成一个个的词
- 语法分析：解析这一系列词的语法结构
- 语义分析：根据语法结构做相应的操作

HTML 括号匹配 – 参考代码 I

```
import sys

# 第一步：词法分析
# 输入：'<html lang="en">Hello, World<\html>'
# 输出：['<', 'html', 'lang', '=', '"en"', '>', 'Hello',
↪   ',', 'World', '<', '\', 'html', '>']
def get_next_token(text, i) -> (str, int):
    # 跳过空格和空行
    while i < len(text) and text[i].isspace():
        i += 1
    if i >= len(text):
        return None, len(text)

    # 引号内部的内容整体作为一个词
    elif text[i] == '\"' or text[i] == '\'':
```

HTML 括号匹配 – 参考代码 II

```
        j = i + 1
        while j < len(text) and (text[j] != text[i] or
        ↪  text[j - 1] == '\\'):
            j += 1
        if j == len(text):
            raise SyntaxError("quotation marks not bounded:
            ↪  {}".format(text[i:j]))
        return text[i:j + 1], j + 1

    # 连续的数字和字母作为一个词
    elif text[i].isalnum():
        j = i + 1
        while j < len(text) and text[j].isalnum():
            j += 1
        return text[i:j], j
```

HTML 括号匹配 – 参考代码 III

```
    # 其余字符全部单独作为一个词
    else:
        return text[i], i + 1

# 将切词结果整理到数组
def tokenize(text):
    tokens = []
    i = 0
    while i < len(text):
        token, i = get_next_token(text, i)
        if token is not None:
            tokens.append(token)
    return tokens
```

HTML 括号匹配 – 参考代码 IV

```
# 第二步: 语法分析 1 - 提取所有 tag 内的内容
# 输入: ['<', 'html', 'lang', '=', '"en"', '>', 'Hello',
↪  ',', 'World', '<', '\', 'html', '>']
# 输出: [['html', 'lang', '=', '"en"'], ['\', 'html']]
def get_next_tag(tokens, i):
    while i < len(tokens) and tokens[i] != '<':
        i += 1
    if i >= len(tokens):
        return None, len(tokens)
    j = i + 1
    while j < len(tokens) and tokens[j] != '>':
        if tokens[j] == '<':
            raise SyntaxError("angle brackets not bounded:
            ↪  {}".format(str(tokens[i:j + 1])))
        j += 1
```

HTML 括号匹配 – 参考代码 V

```
    if j == len(tokens):
        raise SyntaxError("angle brackets not bounded:
        ↪ {}".format(str(tokens[i:j])))
    return tokens[i + 1:j], j + 1

# 将 tags 整理到数组（不必要）
def get_tags(tokens):
    tags = []
    i = 0
    while i < len(tokens):
        tag, i = get_next_tag(tokens, i)
        if tags is not None:
            tags.append(tag)
    return tags
```

HTML 括号匹配 – 参考代码 VI

```
# 第三步: 语法分析 2 - 标签匹配
# 输出: [['html', 'lang', '=', '"en"'], ['\', 'html']]
# 输出: True
def match_tags(tags):
    stack = []
    for tag in tags:
        if tag[0] == '!':
            continue
        elif tag[0] == '/':
            if not stack or stack[-1][0] != tag[1]:
                return False
            stack.pop()
        else:
            stack.append(tag)
    return not stack
```

HTML 括号匹配 – 参考代码 VII

```
# 主程序，依次进行上述三个步骤
try:
    data = sys.stdin.read()
    tokens = tokenize(data)
    tags = get_tags(tokens)
    print('yes' if match_tags(tags) else 'no')
except SyntaxError:
    print('no')
```

附录：同学们在作业 1 和作业 4 中的亮点

- 作业 1 best_group.txt 给出全部 7 种分组方法
- 作业 1 crossroad_model.png 画出好图
- 作业 4 html.py 完成奖励分
- 作业 4 in2pre.py 不同解法

作业 1 best_group.txt – 给出全部 7 种分组方法

环科 17 蔡开奎 & 陈立新；数院 20 林逸云

AD、BA、CB、DC 不冲突，可直接右转通行，不受红绿灯约束。另外 8 条路线分为四组有 7 种分法（注: 在实际过程中还需要考虑各条路车流量等信息来确定最优的方案）:

方案一:

```
Group_A: AB CD
Group_B: AC CA
Group_C: BC DA
Group_D: BD DB
```

方案二:

```
Group_A: AB CD
Group_B: DA DB
Group_C: BC BD
Group_D: AC CA
```

方案三:

```
Group_A: BC BD
Group_B: DA DB
Group_C: CD CA
Group_D: DA DB
```

方案四:

```
Group_A: CD CA
Group_B: AB AC
Group_C: BC DA
Group_D: DB BD
```

方案五:

```
Group_A: CD BD
Group_B: DA CA
Group_C: AC BC
Group_D: AB DB
```

方案六:

```
Group_A: AB DB
Group_B: CD BD
Group_C: AC CA
Group_D: BC DA
```

方案七:

```
Group_A: AB CD
Group_B: BD DB
Group_C: CA DA
Group_D: AC BC
```

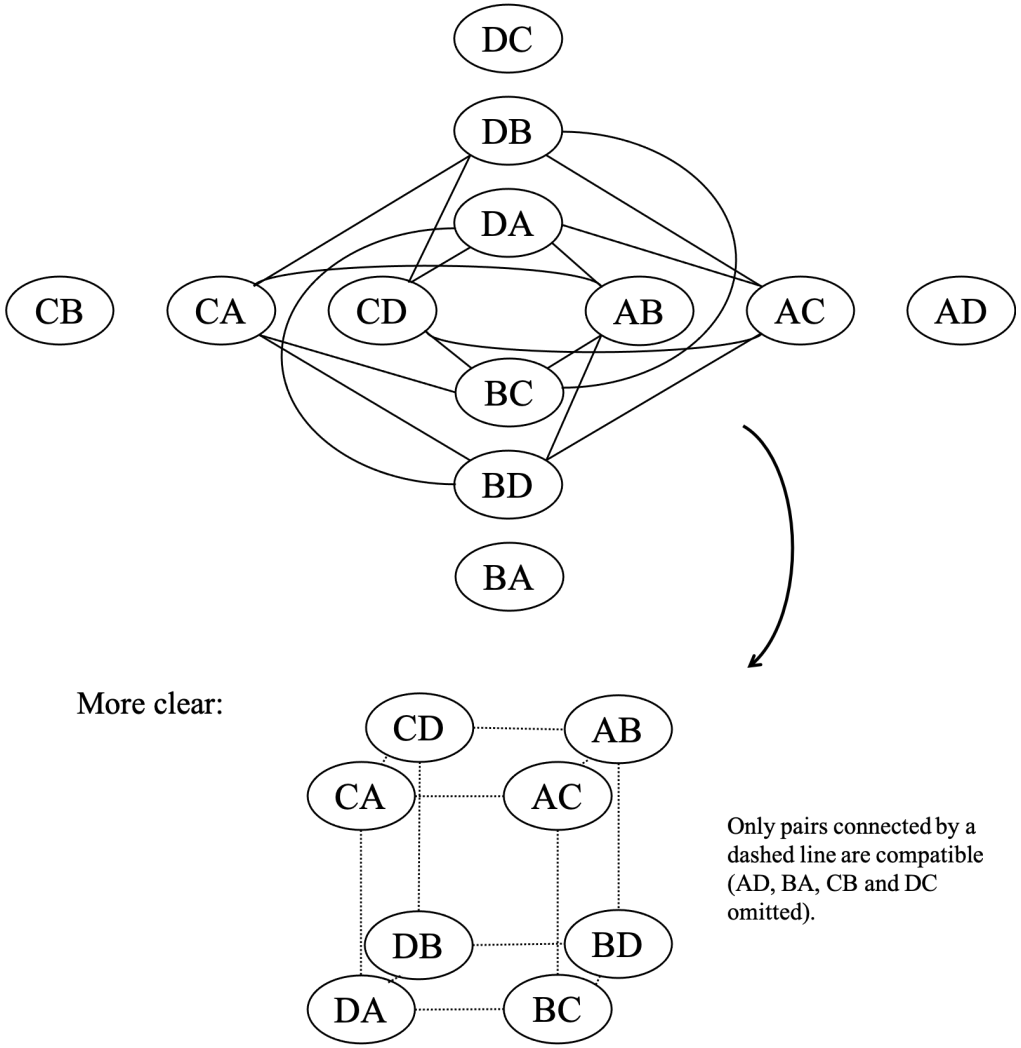
作业 1 best_group.txt – 编程列举出所有可能的染色

工学院 18 谢锦宸

```
The element in the list represents 'AB','AC','AD','BA','BC'
↪  ','BD','CA','CB','CD','DA','DB','DC' respectively.
The numbers represent four groups.
There are 43008 solutions:
[1, 1, 1, 1, 2, 2, 3, 1, 3, 4, 4, 1]
[1, 1, 1, 1, 2, 2, 3, 1, 3, 4, 4, 2]
[1, 1, 1, 1, 2, 2, 3, 1, 3, 4, 4, 3]
[1, 1, 1, 1, 2, 2, 3, 1, 3, 4, 4, 4]
[1, 1, 1, 1, 2, 2, 3, 2, 3, 4, 4, 1]
[1, 1, 1, 1, 2, 2, 3, 2, 3, 4, 4, 2]
[1, 1, 1, 1, 2, 2, 3, 2, 3, 4, 4, 3]
...
```

作业 1 crossroad_model.png – 好图

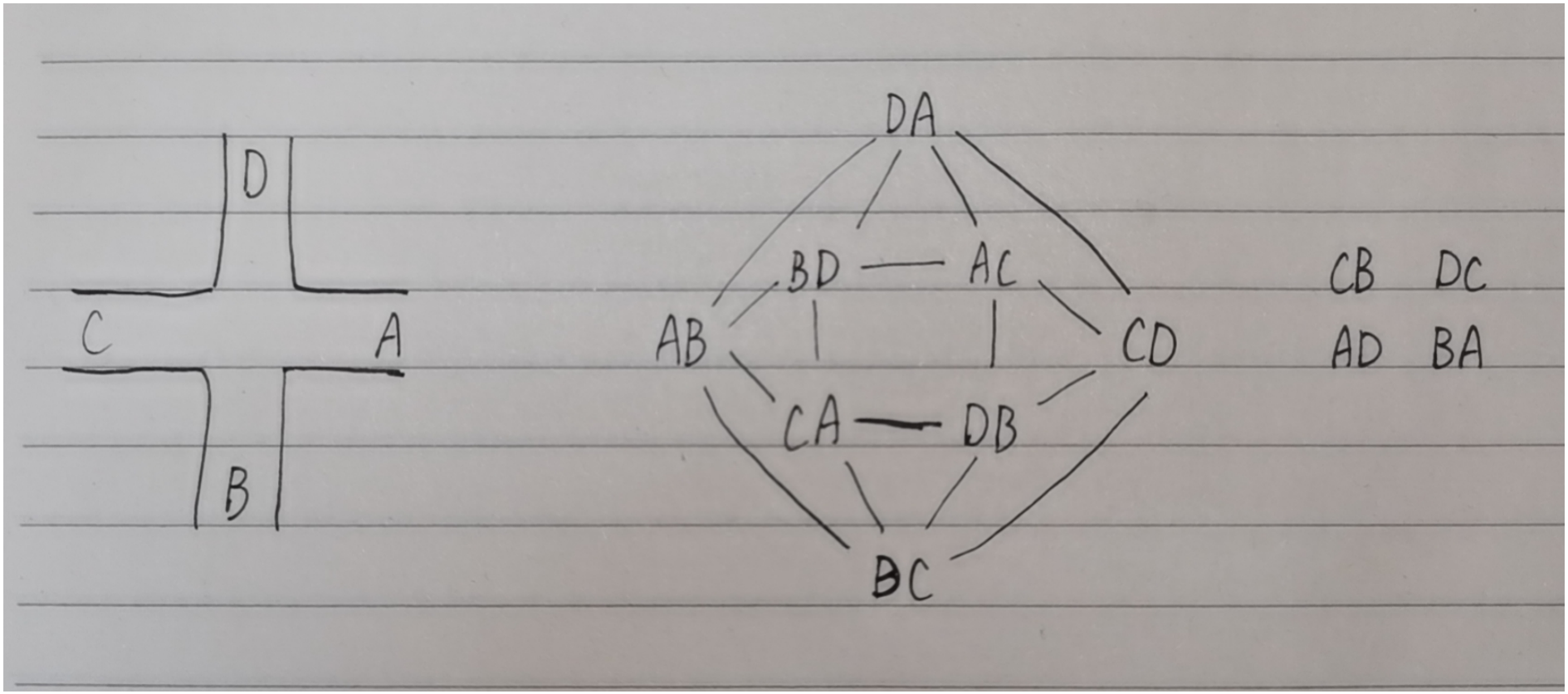
物院 18 沈定宇



注：下图 DB-CD 和 BD-AB 有误，应交叉

作业 1 crossroad_model.png – 好图

数院 19 郭时琨



作业 4 html.py – 完成属性、注释的识别 I

城环 19 赵云迪

```
from pythonds.basic import Stack
import sys
txt = ''
for line in sys.stdin:
    txt+=str(line)
txt.replace(' ','')
txt.replace('\n','')
txt.replace('\r','')

def HTMLformatChecker(html):
    s=Stack()
    slist = []
    j=0
    i=html[j:].find('<')+j
    while (i-j)!=-1:
        j=html[i:].find('>')+i
        h=html[i:j].find(' ')+i
        if (j-i)==-1:
```

作业 4 html.py – 完成属性、注释的识别 II

城环 19 赵云迪

```
            return 'no'
        elif html[i+1]=='!':
            i = html[j:].find('<') + j
            continue
        elif (j-i)!=-1 and html[i+1]=='/':
            if s.isEmpty():
                return 'no'
            else:
                c=html[i+2:j]
                d=s.pop()
                if c!=d:
                    return 'no'
                else:
                    i = html[j:].find('<') + j
        else:
            if h-i==-1:
                a=html[i+1:j]
                s.push(html[i+1:j])
```

作业 4 html.py – 完成属性、注释的识别 III

城环 19 赵云迪

```
            else:
                b=html[i+1:h]
                s.push(html[i+1:h])
            i = html[j:].find('<')+j
    if s.isEmpty():
        return 'yes'
    else:
        return 'no'

print(HTMLformatChecker(txt))
```

由于频繁的取子串操作，算法复杂度变成了 $O(n^2)$，把 `html[i:].find('>')` 改成 `html.find('>', i)` 即可恢复 $O(n)$

作业 4 in2pre.py – 一个不需要左右反转的方法 I

城环 19 覃吴颖；化院 18 刘玉昕；光华 19 李雨航

```
def priority(op) :
    if op == '+' :return 1
    if op =='-':return 1
    if op =='*':return 2
    if op =='/':return 2
    if op =='(':return 0

def expand(alpha:Stack,operator:Stack):
    n2 = alpha.pop()
    n1 = alpha.pop()
    alpha.push(operator.pop()+n1+n2)   # 将运算后的新项放入栈中
    return 0

alpha = Stack()
operator = Stack()
s = input()
for i in s:
    if i in
    ↪  'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234568790'
    ↪  :
```

作业 4 in2pre.py – 一个不需要左右反转的方法 II

城环 19 覃吴颖；化院 18 刘玉昕；光华 19 李雨航

```
            alpha.push(i)

        if i in '+-*/' :
            while operator.size() > 0:
                if  priority(operator.peek())>=priority(i) : # 接下来就降优先
                ↪  级了，运算前面的式子
                    expand(alpha,operator)
                else : break
            operator.push(i)

        if i =='(':operator.push(i)
        if i == ')' :
            while 1 :
                if operator.peek()=='(':
                    operator.pop()
                    break
                else :expand(alpha, operator)
    while operator.isempty() == 0 :
```

作业 4 in2pre.py – 一个不需要左右反转的方法 III

城环 19 覃吴颖；化院 18 刘玉昕；光华 19 李雨航

```
    expand(alpha,operator)
    operator.isempty()
print(alpha.peek())
```

目前的实现做了过于频繁的字符串拼接操作，导致复杂度是 $O(n^2)$，如果用链表存储并把加法留到最后做可以 $O(n)$