离散数学 期末大作业报告

陈国赐 工学院 2100011047

2023 年 1 月 14 日

§ 第一题: 谓词公式计算求值

确定合适的整数范围，编写程序计算语句的真值。

程序运行说明

程序使用 `Python` 编写。程序在 `Python 3.8.10` 能够正常运行，在较高版本的 `Python3` 解释器上应当也能正常运作。

在命令行中执行 `python t1.py` 即能运行程序。

程序简要说明

我们可以知道语句 $\forall n \in S\ (P(n))$ 等价于 Python 语句 `all(P(n) for n in S)`，以及 $\exists n \in S\ (P(n))$ 等价于 Python 语句 `any(P(n) for n in S)`。而多层量词复合则等价于 all 和 any 语句的对应复合，如 $\exists n \in S \forall m \in T\ (P(n,m))$ 等价于 Python 语句 `any(all(P(n, m) for m in T) for n in S)`。我们再用 `range()` 生成的迭代对象来表示变量的取值集合，即可将语句转化成在对应的在有限集内验证真值的 Python 语句。

分析整数范围 & 语句真值结果

我们的程序主要起验证性作用，也就是说，我们选取的数据范围要么能包含到某个特例，要么能够一定程度上揭示这个语句成立（或不成立）的

普遍性规律。

我们直接将程序执行结果中每条语句的真值附在语句的整数范围之后。

a. $\forall n \exists m\ (n^2 < m)$

通过理论分析，我们知道这个语句成立，只需要取 $m = n^2 + 1$ 即满足条件。于是我们取 $n \in [-100, 100], m \in [0, 10009]$，即可知道 $n \in [-100, 100]$ 时语句成立，并可认为语句在定义域上普遍为真。

结果: `Statement A: True`

b. $\exists n \forall m\ (n < m^2)$

通过理论分析，我们知道这个语句成立，只需要取 $n = -1$ 即满足条件。于是我们取 $n \in [-1, 0], m \in [0, 99999]$，即可知道 $n = -1, m \in [0, 99999]$ 时内层语句成立，并可认为语句在定义域上普遍为真。

结果: `Statement B: True`

c. $\forall n \exists m\ (n + m = 0)$

通过理论分析，我们知道这个语句成立，只需要取 $m = -n$ 即满足条件。于是我们取 $n \in [-1000, 1000], m \in [-1000, 1000]$，即可知道 $n \in [-1000, 1000]$ 时语句成立，并可认为语句在定义域上普遍为真。

结果: `Statement C: True`

d. $\exists n \forall m\ (nm = m)$

通过理论分析，我们知道这个语句成立，只需要取 $n = 1$ 即满足条件。于是我们取 $n \in [0, 2], m \in [0, 99999]$，即可知道 $n = 1, m \in [0, 99999]$ 时语句成立，并可认为语句在定义域上普遍为真。

结果: `Statement D: True`

e. $\exists n \exists m\ (n^2 + m^2 = 5)$

通过理论分析，我们知道这个语句成立，只需要取 $n = 2, m = 1$ 即满足条件。于是我们取 $n \in [0, 4], m \in [0, 4]$，发现存在变量取值 $n = 2, m = 1$ 使命题成立，于是语句为真。

结果: `Statement E: True`

f. $\exists n \exists m\ (n^2 + m^2 = 6)$

通过理论分析，我们知道这个命题在 $\max(|n|, |m|) > 4$ 时显然不成立。且 n, m 的符号不影响命题真值。于是我们的取值范围可以限定为 $n \in [0, 4], m \in [0, 4]$，即可知道给定集合内不存在变量取值使命题成立，故语句为假。

结果: `Statement F: False`

g. $\exists n \exists m\ (n + m = 4 \wedge n - m = 1)$

对上式估计，我们知道这个命题在 $\max(|n|, |m|) > 10$ 时显然不成立。于是我们取 $n \in [-10, 10], m \in [-10, 10]$，即可知道给定集合内不存在变量取值使命题成立，故语句为假。

结果: `Statement G: False`

h. $\exists n \exists m\ (n + m = 4 \wedge n - m = 2)$

变量取值范围估计同 g.。于是我们取 $n \in [-10, 10], m \in [-10, 10]$，发现存在变量取值 $n = 3, m = 1$ 使命题成立，于是语句为真。

结果: `Statement H: True`

i. $\forall n \forall m \exists p\ (p = \frac{n+m}{2})$

首先我们可以做估计: $\min(n, m) \leq \frac{n+m}{2} = p \leq n + m$，通过理论分析，我们知道这个语句不成立，只需要取 $n = 1, m = 0$ 即满足条件。于是我们取 $n \in [0, 4], m \in [0, 4], p \in [0, 9]$，发现存在变量取值 $n = 1, m = 0$ 使内层命题在 $p \in [0, 9]$ 不成立，可认为内层命题始终为假，于是语句为假。

结果: `Statement I: False`

§ 第二题: 演员关系图的计算

建立演员关系无向图 $G = (V, E)$，节点 v 为演员，如果两个演员 $v1$ 和 $v2$ 出演过同一部电影，则建立一条边 $e = v1, v2$。

建立图结构并实现功能:

- 输出图 G 的统计信息：节点数，边数，每个节点的度数分布直方图；
- 输入两位演员的名字，打印输出最短路径，没有路径的显示“无法到达”；
- 输出图 G 连通分支个数，每个连通分支包含多少节点，每个连通分支的直径；
- 每个演员的 Bacon Number，即到”Kevin Bacon (I)” 这位演员的最短路径长度，并画出 Bacon Number=0 9 的演员数量分布直方图；

程序运行说明

程序使用 `Python` 编写，并调用了 `matplotlib` 库，可能需要在运行前通过 `pip3` 安装。经测试，程序在 `Ubuntu 20.04.1, Python 3.8.10` 及 `Windows 10, Python 3.9.6` 环境下均能够正常运行。

程序文件结构如表格所示：

文件名	功能
t2.py	主程序
t2part.cpp	求连通分支及其直径的子程序
t2part	t2part.cpp 在 Linux 下编译的可执行文件
t2part.exe	t2part.cpp 在 Windows 下编译的可执行文件

在命令行中执行 `python t2.py` 即能运行程序。程序花费一定时间读取并建立图结构，然后显示文字菜单，可以根据程序显示的指示进行操作。其中可能需要注意的是，**选项 (2) 仅适用于小数据集**，在 `full` 和 `no-tv-v` 数据集上会产生段错误。

建议将数据集文件放在 t2 目录下。**运行程序前需要将 `t2.py` 开头的 `FILENAME` 变量修改为数据集文件名**。

运行结果

imdb.top250.txt　原图的节点数: 12466

电影个数: 250

原图的边数: 588737

节点度数直方图如下图:

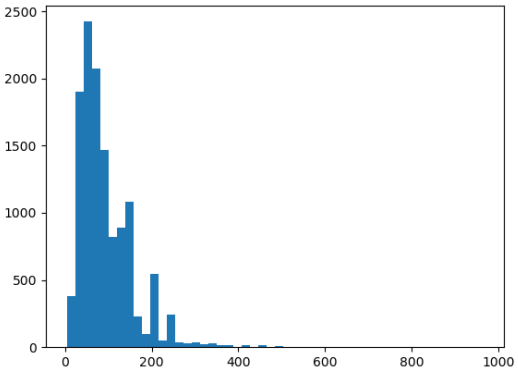


图 1: Deg for top250

测试从 Ernie Adams (I) 到 William Adams (II) 的最短路，输出为:

距离: 3

路径为:

Ernie Adams (I) -> Irving Bacon -> Blue Washington -> William Adams (II)

Bacon Number 在 0 ~ 9 的分布如下图:

每个演员的 Bacon Number 以及连通分支信息可在 `t2output/top250/`中查看。

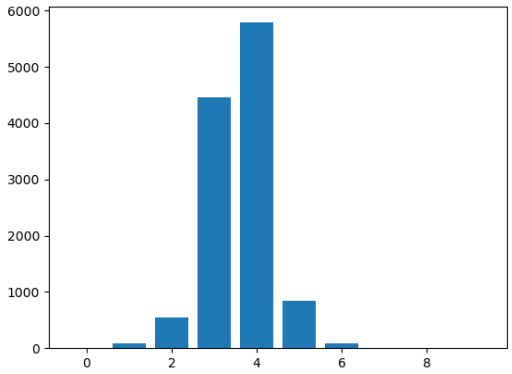


图 2: Bacon number for top250

imdb.no-tv-v.txt 原图的节点数: 2386567

电影个数: 655856

原图的边数: 98533085

节点度数直方图如下图:

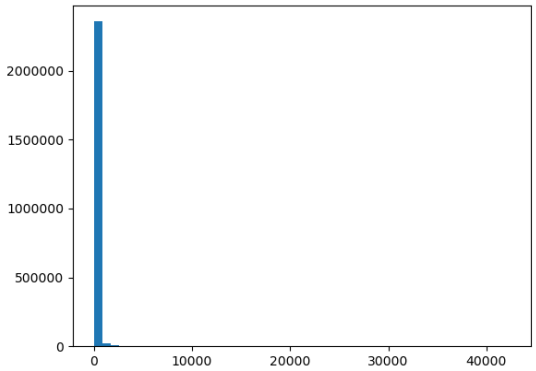


图 3: Deg for no-tv-v

Bacon Number 在 0 ~ 9 的分布如下图:

其他信息可在 `t2output/no-tv-v/`中查看, 其中 `conn2.out` 的直径是两次 bfs 的近似解。

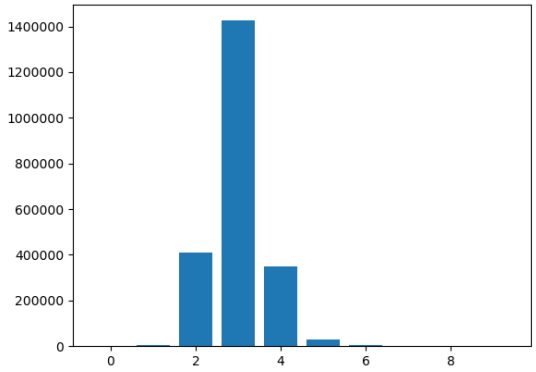


图 4: Bacon number for no-tv-v

程序分析

1. 图结构的转换 为了方便下面的讨论，我们记 n 为演员个数，m 为电影个数，p_i 为第 i 个电影的参演演员个数，q_{ij} 为第 i 个电影的第 j 个参演演员的编号。

最开始，我直接按照题目给的方式建边，发现在跑 full 数据集的时候电脑会死机。其中 `Python` 语言的特性带来的内存开销虽确实大，但我们发现图中的边数达到了 10^8 的较大的数量级，对内存和时间都是一个挑战。理论分析可知 $e = \sum p_i^2$。若能够优化图结构，使得边数下降一个数量级，能大大优化我们程序的性能。

我们发现电影 i 使得原图中 p_i 个点两两相连，这也就是边数可能较大的原因。我们尝试对电影 i 在图中建立虚点，编号为 $-i$，并且将 $-i$ 和每个参演演员 q_{ij} 相连。我们可以推知这样建立的图中一条“演员-电影-演员”的路径和原图中的一条边一一对应，最短路径的性质也不变 (只是距离需要乘 2)，而新的图中边数变为 $\sum p_i$，使内存得到了大程度的优化。

至于图的边数和节点度数，我们可以对每个电影 i 维护一个参演演员的集合 $\{q_{ij}\}$。每个电影对原图边数的贡献为 p_i^2，对演员 q_{ij} 的度数贡献为 $p_i - 1$。上述过程我们并没有真的建立原图的结构，在后续的过程中我们也不需要维护原图的结构。

综上所述，我们只需维护转化后的图结构就能完成我们的任务。

2. 最短路径 为了方便下面的讨论，我们记 $N = n + m$ 为新图节点个数，$M = \sum p_i$ 为新图边个数。

由于 $N + M$ 在 10^7 的数量级，我们可以接受取其中一个演员作为源点，对整个图做单源最短路。又因为图中边权全为 1, 我们按照 bfs 的顺序访问节点，距离是单调不减的。所以我们可以直接用 bfs 求最短路，从节点 u 访问到未被访问过的节点 v 时，源点 s 到 v 的（其中一条）最短路路径一定是 s->u->v。于是距离 dis[v]=dis[u]+1，并记录节点 v 的前驱节点 pre[v]=u。打印路径时，可以从另一个演员 t 处通过 pre[] 数组倒着走回源点 s 来还原路径。

Bacon Number 实质上也是以 Kevin Bacon(I) 为源点的单源最短路，思路类似，故不再赘述。

3. 求直径 ver1: 失败的尝试 由于一开始以为图的直径的求法和树的直径求法相同，都是从任意点 bfs 求最远点 s，再从 s 用 bfs 求最远点 t 得到一条直径 s-t，于是用这个假算法 “通过” 了 full 数据集而且跑得飞快 (2min 以内)。后来意识到树的性质足够良好，两个节点间有且只有一条路径，能保证第一次 bfs 的最远点一定是直径端点，而图不能保证，所以这个算法不能按原方法证明（后来测出来确实是假算法）。

但是两次 bfs 方法求得的最长路径是直径的一个下界，而且和直径长度较为接近，所以我们可以把这个算法作为对直径的一个较合理的估计，特别是对暴力算不出来的大数据集，我们可以用这个算法算出近似解。

4. 求直径 ver2: 暴力 + 奇技淫巧 对一般无向图求直径貌似没有较为优秀的算法，我们对小数据采用 $O(N(N + M))$ 的暴力，就是对每个连通分支，以每个点为源做一次 bfs 求最长路径，最后统计最长路径的最大值作为连通分支直径。

这样的算法理论上对 $N = 10^4, M = 3 \times 10^4$ 的数据集理论上应该跑得飞快，但是 Python 的常数太大了，需要 10min 才能跑出结果，只能忍痛割爱，用 C++ 写这一部分。当然我们不需要将整个题的代码重写（而且 C++ 的字符串处理过于麻烦），只需要将图结构导入文件，然后用 `subprocess` 调用对应的 C++ 程序即可。重写后导出图结构的时间 + C++ 程序 IO 时间 + 计算时间在 10 秒以内，令人感叹。

5. 连通分支个数和大小 3 和 4 中 bfs 的过程中都遍历了每个连通分支，顺便统计一下即可。

§ 第三题：正则表达式的有限状态机

字符集取 a, b，编写程序，实现如下功能：

- 输入一个正则表达式（长度 <20），如：`ab*(a|b)`，输出对应的 DFA，并绘制图示；
- 得到 DFA 后，输入一个仅包含 ab 的字符串，输出是否接受此串，并输出状态转移序列；

程序运行说明

程序使用 Python 编写，并调用了 matplotlib, tkinter,graphviz 库，其中 matplotlib 库可能需要在运行前通过 pip3 安装。graphviz 库在 Ubuntu 环境下可以通过 sudo apt install graphviz && pip3 install graphviz 安装，在 Windows 环境下的安装可参考网页:https://zhuanlan.zhihu.com/p/268532582，并**在确认环境变量完全更新之后**运行程序。

经测试,程序在 Ubuntu 20.04.1，Python 3.8.10 及 Windows 10，Python 3.9.6 环境下均能够正常运行。

程序文件结构如表格所示：

文件名	功能
automata.py	自动机相关的类和函数
graph.py	调用 graphviz 库，画出自动机的图像
main_cli.py	命令行界面的主程序
main_gui.py	图形界面的主程序 (需要 tkinter 库)

在命令行中执行 python main_cli.py 即能运行命令行界面的程序，执行 python main_gui.py 即能运行图形界面的程序。

运行结果

用题目样例 ab*(a|b) 作为正则表达式，abbb 作为测试字串，输出如下：

```
输入只含a,b,*,|和括号的合法正则表达式:ab*(a|b)
-----------
对应的NFA:
初始状态 0
终止状态 11

节点 0
eps -> []
a -> [1]

节点 1
eps -> [4]

节点 2
eps -> []
b -> [3]

节点 3
eps -> [2, 5]

节点 4
eps -> [2, 5]

节点 5
eps -> [10]

节点 6
eps -> []
a -> [7]
```

```
节点 7
eps -> [11]

节点 8
eps -> []
b -> [9]

节点 9
eps -> [11]

节点 10
eps -> [6, 8]

节点 11
eps -> []
是终止节点
图像导出至 nfa.png
-----------
对应的DFA:
初始状态 0
终止状态 [2, 3]

节点 0
a -> 1

节点 1
a -> 2
b -> 3

节点 2
是终止节点
节点 3
a -> 2
```

```
b -> 3
是终止节点
图像导出至 dfa.png

输入匹配字符串:abbb
转移过程表:
0    == a ==>    1
1    == b ==>    3
3    == b ==>    3
3    == b ==>    3

匹配成功
```

建立的 NFA 和 DFA 图像如下:

GUI 界面如下:

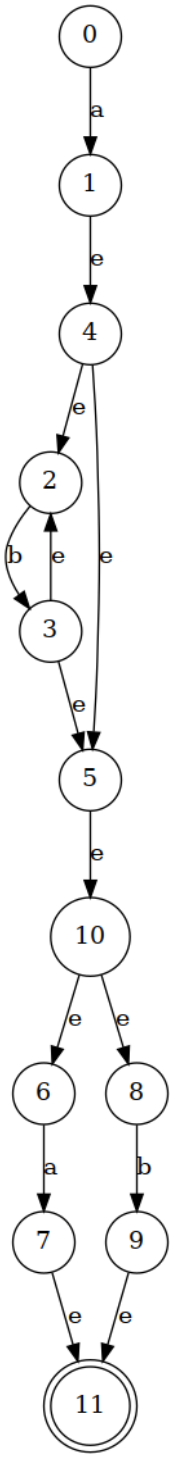


图 5: NFA

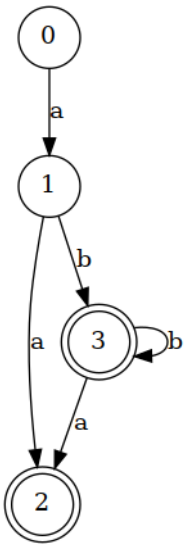


图 6: DFA

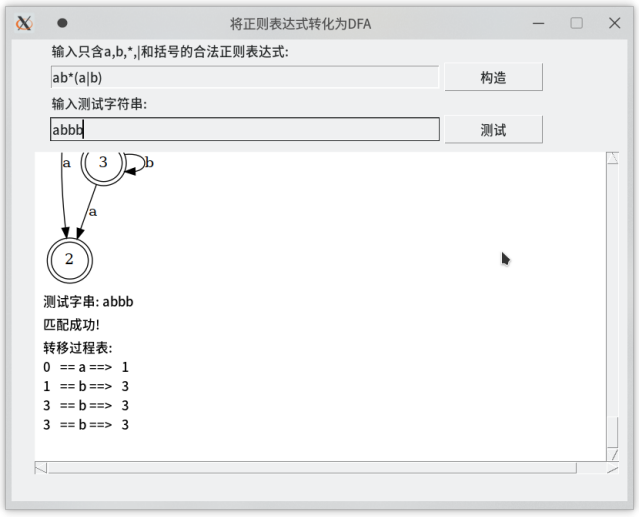


图 7: main_gui.py

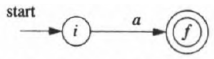
程序分析

这里我们仅对将正则表达式转化为 DFA 的算法进行讨论。这个算法我参考了机械工业出版社的《编译原理——原理、技术与工具》书中 3.7 节“从正则表达式到自动机”的内容，采用了先将正则表达式转化为 NFA，再将 NFA 转化为 DFA 的方式。

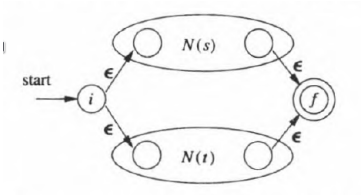
1. 正则表达式到 NFA：递归构造 我们先添加一个额外的符号”.”，这表示连接两个表达式的符号。在 RE 中适当位置插入”.”，此时我们的 RE 有了两个两目运算符和一个单目运算符。

接下来我们考虑递归构造的规则。

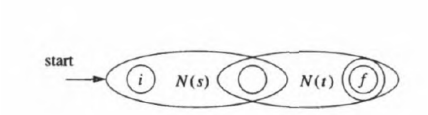
1. 对单个字符 c，我们可以构造如下 nfa。



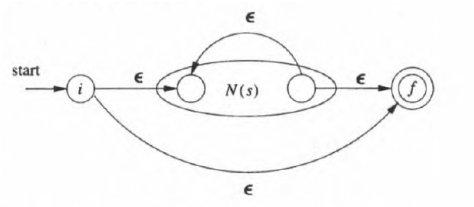
2. 若 r=s|t，我们可以构造如下 nfa。



3. 若 r=s.t，我们可以构造如下 nfa。



4. 若 r=s*，我们可以构造如下 nfa。



实际操作时，我们用类 NFANode 表示 NFA 的一个节点对象。先将 RE 化成对应的后缀表达式，再像数算中的 “表达式求值” 题一般，用两个栈分别维护节点对象和运算符，由此建立起对应的 NFA。

2. NFA 到 DFA：子集构造法 从 NFA 转移到 DFA，我们的目的是消除掉所有的 ϵ 转移，以及使某个状态对同一个输入符号不会有多个转移。而子集构造法的基本思想是让 DFA 的每个节点对应 NFA 的某个节点集合。DFA 在读入 $s_1 s_2 \dots s_n$ 之后到达的状态对应了 NFA 读入 $s_1 s_2 \dots s_n$ 之后到达的状态集合。某个 DFA 节点是终止状态当且仅当对应 NFA 节点集合中存在处于终止状态的节点。

我们引入概念:ϵ 闭包 `EClose(T)`，表示从状态集合 T 中的某个元素 s，只通过 ϵ 转移到的所有状态的集合。以及 `move(T,c)`，表示从状态集合 T 中的某个元素 s，通过一次字符 c 转移到的所有状态的集合。

则子集构造法的伪代码可以表示如下：

Algorithm 1 子集构造法

$DStates \leftarrow \{EClose(s_0)\}$
$visited \leftarrow \{\}$
while $\exists T \in DStates \wedge T \notin visited$ **do**
  $visited \leftarrow visited \cup \{T\}$
  for $c \in language$ **do**
    $U \leftarrow EClose(move(T, a))$
    if $U \notin DStates$ **then**
      $DStates \leftarrow DStates \cup \{U\}$
    end if
    $Dtrans[T, c] \leftarrow U$
  end for
end while

而 ϵ 闭包可以通过 bfs 或 dfs 求得，本题代码采用的是 bfs。

3. 字串匹配 构造出 DFA 后，字串匹配就不再有难度。我们只需从初始状态开始，从左到右取字串的字符并转移到下一个状态。若某一步无法转移则匹配失败。最终若是停在终止状态则匹配成功，否则匹配失败。