确定性推理：归结推理

# 1实验目的

1.了解谓词公式化为子句集的方法；

2.了解鲁滨逊归结原理；

3.了解归结反演；

4.了解应用归结原理求解问题。

# 2实验要求

本次试验后，要求学生能：

1. 熟练将谓词公式化为子句集；
2. 理解归结原理；
3. 能将实际问题转化为谓词逻辑，并化为子句集，最后编写程序利用归结原理实现推理证明或问题求解。

# 3实验原理

## 3.1谓词公式化为子句集的方法

### 3.1.1子句集定义

（1）不能再分解的命题称为**原子谓词公式**。

（2）原子谓词公式及其否定，称为**文字**，P为正文字，﹁P为负文字，称P与﹁P为一对互补文字。

（3）任何文字的析取式称为**子句**。任何文字本身也是子句。

（4）由子句构成的集合称为**子句集**。

（5）不包含任何文字的子句称为空子句，表示为**NIL**。

由于空子句不含任何文字，它不能被任何解释所满足，所以，空子句是永假的。

在谓词逻辑中，任何一个谓词公式都可以用过等价关系和推理规则化为相应的子句集，从而根据子句集更加容易第判断谓词公式的可满足性。

### 3.1.2谓词公式化为子句集的一般步骤

（1）消去蕴含式（→）和等价式（↔）。

可以利用如下的谓词公式等价关系：

P→Q ⇔﹁ P∨Q  
P↔Q ⇔ (P∧Q)∨(﹁P∧﹁Q)

（2）将否定符号移到紧靠谓词的位置上。

可利用如下等价关系：

双重否定律：﹁(﹁P) ⇔ P

德摩尔根律：﹁(P∧Q) ⇔﹁P∨﹁Q  
﹁(P∨Q) ⇔﹁P∧﹁Q

量词转换律：﹁ (∀x)P(x) ⇔ (∃x) ﹁P(x)  
﹁ (∃x)P(x) ⇔ (∀x)￢P(x)

（3）变量标准化

重新命名每个变元，使每个量词采用不同的变元，从而使不同量词的约束元有不同的名字。这是因为在任一量词的辖域内，受到该量词约束的变元为一哑元，它可以在该辖域内被另一个没有出现过的任一变元统一替代，而不改变谓词公式的值。

（4）消去存在量词

存在量词不出现在全称量词的辖域内时，只要用一个新的个体常量替代收该存在量词约束的变元，就可以消去存在量词。

存在量词在一个或多个全称量词辖域内时，要用Skolem函数替换受该存在量词约束的变元，从而消去存在量词。

（5）化为前束形

所谓前束形，就是把所有的全称量词都移动到公式的前面，使每个量词的辖域都包括公式后的整个部分。

（6）化为Skolem标准形

Skolem标准形的一般形式为：

(∀x1) (∀x2)……(∀xn)M

M是子句的合取式

一般利用析取和合取的分配律把谓词公式化为Skolen标准形。

P∨(Q∧R) ⇔ (P∨Q)∧(P∨R)

P∧(Q∨R) ⇔ (P∧Q)∨(P∧R)

（7）略去全程量词

由于公式中的变量都是全称量词，因此，可以省略全称量词。

（8）消去合取词

消去合取词，把母式用子句集表示。

（9）子句变量标准化

即，使每个子句中的变量符号不同。

### 3.1.3谓词公式可满足性定理

谓词公式不可满足的充要条件是其子句集不可满足。

## 3.2归结原理

前一小节将谓词公式的不可满足性转化为子句集中子句的不可满足性。

为了能用机器自动地证明子句集的不可满足性，鲁滨孙提出了归结原理。其基本思想为：检查子句集S是否包含空子句，若包含，则S不可满足；若不包含，则再子句集中选择合适的子句进行归结，一旦归结到空子句，则说明子句集S是不可满足的。

（1）命题逻辑的归结原理

定义1：设C1与C2是子句集中的任意两个子句，如果 C1中的文字L1与 C2中的文字L2互补，那么从C1和 C2中分别消去L1和L2，并将二个子句中余下的部分析取，构成一个新子句C12。

定理1：归结式C12是其亲本子句C1与C2的逻辑结论。即如果 C1与C2为真，则C12为真。

推论1-1：设C1与C2是子句集S中的两个子句，C12是它们的归结式，若用C12代替C1与C2后得到新子句集S1，则由S1不可满足性可推出原子句集S的不可满足性。

推论1-2：C1与C2是子句集S中的两个子句，C12是它们的归结式，若C12 加入原子句集S，得到新子句集S1，则S与S1在不可满足的意义上是等价的。

（2）谓词逻辑的归结原理

在谓词逻辑中，由于子句含有变元，所以不能像命题逻辑那样可以直接消去互补文字，需要先用最一般合一对变元进行代换，然后才能进行归结。

与命题逻辑意义，对于谓词逻辑，归结式是其亲本子句的逻辑结论。

对于一阶谓词逻辑，即若子句集是不可满足的，则必存在一个从该子句集到空子句的归结演绎；若从子句集存在一个到空子句的演绎，则该子句集是不可满足的。

需要注意，如果没有归结出空子句，则既不能说S不可满足，也不能说S是可满足的。

## 3.3归结反演

欲证明Q是P1、P2……Pn的逻辑结论，只需证明（（P1∧P2∧……∧Pn）∧﹁Q）是不可满足的。

可得到归结反演的一般步骤：

（1）将已知条件表示为谓词公式F；

（2）将带证明的结论表示为谓词公式Q，将Q否定得到﹁Q；

（3）将{F，﹁Q }化为子句集S；

（4）应用归结原理对子句S中的子句进行归结，并把每次归结得到的归结式都并入S中，直到出现空子句停止，即可证明Q为真。

## 3.4应用归结原理求解问题

用归结原理求解问题的一般步骤为：

（1）将已知条件表示为谓词公式，将其化为子句集记为S；

（2）将待求解的问题也用谓词公式F表示，将它的否定与答案谓词ANSWER构成析取式，如﹁F（x）∨ANSWER（x）；

（3）把（2）中构造的析取式也加入S，形成新的子句集S’；

（4）对S’用归结原理进行归结；

（5）若最终能得到归结式ANSWER（A），则答案就是ANSWER的项A。

# 4实验步骤

## 4.1实验介绍

本实验需要同学们有一定的Python语言编程基础，需要同学们编写程序，实现自动地对子句集进行归结推理，从而解决问题，题目如下：

已知：A、B、C三人中有老实人（**从来不说假话**），也有大骗子（**从来不说真话**）。有一天，小明好奇地同时问他们：你们谁才是老实人？A说：“B和C都是大骗子”；B却说：“A和C都是大骗子”；C说：“A和B中至少有一个大骗子”。

求：谁是老实人（**从来不说假话**），谁是大骗子（**从来不说真话**）？

## 4.2构建子句集

用谓词T（x）表示x说的都是真话。

如果A是老实人，则有：

T(A) → ﹁T(B) ∧ ﹁T(C)

如果A是大骗子，则有：

﹁T(A) → T(B) ∨ T(C)

对B、C说的话进行相同的处理，可得：

T(B) → ﹁T(A) ∧ ﹁T(C)

﹁T(B) → T(A) ∨ T(C)

T(C) → ﹁T(A) ∨ ﹁T(B)

﹁T(C) → T(A) ∧ T(B)

将上面的句子手动化为子句集，可得S0。

（1）﹁T(A) ∨ ﹁T(B)

（2）﹁T(A) ∨ ﹁T(C)

（3）T(A) ∨ T(B) ∨ T(C)

（4）﹁T(B) ∨ ﹁T(C)

（5）﹁T(C) ∨ ﹁T(A) ∨ ﹁T(B)

（6）T(A) ∨ T(C)

（7）T(B) ∨ T(C)

若要求谁是老实人，则把﹁T(x) ∨ Answer（x）作为第（8）个子句并入S0，形成S1，若最后能归结出只含Answer（x）的子句，则归结后Answer的项即为答案。

若要证明A不是老实人，则有﹁T(A)，把它的否定﹁(﹁T(A)) ，即T(A)，作为第（8）子句加入S0，形成S2，若能归结出空子句，说明A不是老实人。

## 4.3编程实现

为了方便编程，用“|”表示析取“∨”，用“&”表示合取“∧”，用“~”表示非“﹁”，用“-”表示空集。用x、y、z、x0、x1、x2表示变元，用A、B、C、D表示常量。

"""

"~"：非

"|"：析取

"&"：合取

"-"：空集

"""

# 变元集合

X = ['x', 'y', 'z', 'x0', 'x1', 'x2', 'x3']

# 常量集合

a = ['a', 'b', 'c', 'd', 'A', 'B', 'C', 'D']

S = [] # 保存子句集中的谓词

Var = [] # 保存S中谓词对应的项

# 由已知条件得到的初始子句集

Set = ["~T(A)|~T(B)", "~T(A)|~T(C)", "~T(B)|~T(C)", "T(A)|T(B)|T(C)", "~T(C)|~T(A)|~T(B)",

"T(C)|T(A)", "T(B)|T(C)"]

### 4.3.1归结反演实验

（1）为证明A不是老实人，将﹁T(A)的否定﹁(﹁T(A)) ，即T(A)，作为第8个子句加入子句集S。

Set = ["~T(A)|~T(B)", "~T(A)|~T(C)", "~T(B)|~T(C)", "T(A)|T(B)|T(C)", "~T(C)|~T(A)|~T(B)",

"T(C)|T(A)", "T(B)|T(C)", "T(A)"]

（2）读取子句集中的信息

将子句集中的每个子句的文字用“|”分隔，保存成文字的集合和文字项的集合方便后续操作。例如，子句集["P(A)|~Q(B)", "~R(A)|Q(C)"]转化成S=[['P', '~Q'], ['~R', 'Q']]与Var=[['A', 'B'], ['A', 'C']]。

def read\_set():

global S, Var

for clause in Set:

clause = clause.split("|")

Var\_temp = []

P\_temp = []

for word in clause:

# 找出谓词

if word[0] == "~":

P\_temp.append(word[0:2])

else:

P\_temp.append(word[0])

# 找出谓词的项

for i in range(len(word)):

for j in range(len(word) - 1, i, -1):

if word[i] == "(" and word[j] == ")":

word\_slice = word[i + 1:j]

for character in word\_slice:

if character in X or a:

Var\_temp.append(character)

Var.append(Var\_temp)

S.append(P\_temp)

（3）实现归结算法。

定义opposite（L）函数，作用是返回L的逆文字。例如，对于L1 = ﹁T(A)，返回T(A)。

def opposite(word):

if word[0] == "~":

return word.replace("~", "")

else:

return "~" + word

定义函数decline\_self(I)，目的是去掉C1、C2的归结式C12中重复的文字；同时，若C12中存在互补文字，由于C12内部各文字之间为析取关系，而互补文字析取之后永真，所以将C12从子句集里删除。

因为子句永真而删除，则返回1；其他情况正常返回0.

def decline\_self(I):

global S, Var

for count1 in range(len(S[I])):

if count1 >= len(S[I]):

continue

for count2 in range(count1 + 1, len(S[I])):

if count1 >= len(S[I]):

continue

if count2 >= len(S[I]):

continue

# 删去重复的文字

if S[I][count1] == S[I][count2] and Var[I][count2] == Var[I][count2]:

del S[I][count2]

del Var[I][count2]

# 如果存在子式~T(A)|T(A)，说明这个子句永真，直接从子句集删掉

if opposite(S[I][count1]) == S[I][count2] and Var[I][count1] == Var[I][count2]:

del S[I]

del Var[I]

return 1

return 0

定义函数decline(I, J, K, L)对C1、C2进行归结消解，I、J为C1中文字L1在S中的序号，K、L为C2中文字L2在S中的序号。例如，~R在S=[['P', '~Q'], ['~R', 'Q']]中的序号为1，0（~R是S中第1号子句的第0号文字）。

归结消解之前要先进行最一般合一代换：如果L1、L2的项相同，直接进行消解；如果L1的项是变元x，L2的项是常量a，则对C1中的所有x进行最一般合一代换令x=a；如果如果L2的项是变元x，L1的项是常量a，则对C2中的所有x进行最一般合一代换令x=a。

如果L1、L2的项是不同的常量时，函数直接返回1，表示L1、L2不是互补文字不可消解。

归结消解过程就是将互补文字从C1、C2中删除，然后将剩下的部分析取成新的子句加入S。 如果检测到归结出了空子句，说明带证命题为真，返回0让推理停止，

def decline(I, J, K, L):

global S, Var

L1 = S[I][J]

L2 = S[K][L]

# 变元用最一般合一代换

L1\_var = Var[I][J]

L2\_var = Var[K][L]

# 如果L1、L2的项相同，可以直接消解；如果不同，需按一下步骤进行合一代换

if L1\_var != L2\_var:

# 项都为常数则不能消解

if L1\_var in a and L2\_var in a:

return 1

# 一个项为变元，另一个为常数，则先最一般合一代换再消解

elif L1\_var in a:

for n in range(len(Var[K])):

if Var[K][n] == L2\_var:

Var[K][n] = L1\_var

elif L2\_var in a:

for m in range(len(Var[I])):

if Var[I][m] == L1\_var:

Var[I][m] = L2\_var

print("用%d" % I + "与%d" % K + "进行归结")

print("用" + "v".join(S[I]) + "与" + "v".join(S[K]) + "进行归结")

# 消解

del S[I][J]

del Var[I][J]

del S[K][L]

del Var[K][L]

S[I] = S[I] + S[K]

Var[I] = Var[I] + Var[K]

# 用反演推理时，出现空集就停

for ss in S:

# 出现空集

if not ss:

print(S)

print(Var)

print("NIL")

print("出现空集，说明" + opposite((Set[-1]) + "为真"))

return 0

del\_self\_flag = decline\_self(I)

del S[K - del\_self\_flag] # 如果前面删掉了一个永真式，k的实际位置往前挪了一格

del Var[K - del\_self\_flag]

print(S)

print(Var)

（4）实现自动地对子句集归结推理

不断循环遍历子句集S中各子句的各个文字。找出含有一对互补文字L1、L2的子句C1、C2，i、j为L1在S中的序号，k、l为L2在S中的序号，调用decline(i, j, k, l)对C1、C2进行归结消解。直到归结出空集，说明问题得证。

由于在循环中进行了删除操作，会导致S的结构发生变化（如S的长度、S中子句的长度），所以i、j、k、l很可能越界，所以i、j、k、l每次更新后都要判断是都越界，还要判断是否已经达到了终止条件（问题得证时flag = decline(I,J,K,L) = 0）。

if \_\_name\_\_ == '\_\_main\_\_':

read\_set()

print("S")

print(S)

print("Var")

print(Var)

flag = 1

while True:

if flag == 0:

break

for i in range(len(S)):

if flag == 0:

break

if i >= len(S):

continue

for j in range(len(S[i])):

if flag == 0:

break

if i >= len(S):

continue

if j >= len(S[i]):

continue

for k in range(i + 1, len(S)):

if flag == 0:

break

if k >= len(S):

continue

for l in range(len(S[k])):

if k >= len(S):

continue

if l >= len(S[k]):

continue

# 找出含有互补文字L1、L2的C1、C2，以及L1、L2在S中的序号

if opposite(S[i][j]) == S[k][l]:

flag = decline(i, j, k, l)

if flag == 1:

continue

elif flag == 0:

break

### 4.3.2应用归结原理求解问题

（1）为求解谁是老实人，设老实人为x，用A(x)代表Answer谓词，将~T(x)与A(x)析取成~T(x)|A(x)作为第8个子句加入子句集S。

对子句集S进行归结推理，如果能得到归结式A（person），则person即为所求的解。

Set = ["~T(A)|~T(B)", "~T(A)|~T(C)", "~T(B)|~T(C)", "T(A)|T(B)|T(C)", "~T(C)|~T(A)|~T(B)",

"T(C)|T(A)", "T(B)|T(C)", "~T(x)|A(x)"]

（2）只需对上面归结反演的代码中，Set子句集、decline（I,J,K,L）函数、主函数部分稍加调整，即可实现应用归结原理自动求解问题。

归结消解部分，消解停止的条件改为得到归结式A（person），其中person为某一常量。为了方便，将含A(x)的子句固定在子句集的最后一个。

代码调整为：

def decline(I, J, K, L):

global S, Var

L1 = S[I][J]

L2 = S[K][L]

# 变元用最一般合一代换

L1\_var = Var[I][J]

L2\_var = Var[K][L]

if L1\_var != L2\_var:

# 都在集合a则不能消解

if L1\_var in a and L2\_var in a:

return 1

elif L1\_var in a:

for n in range(len(Var[K])):

if Var[K][n] == L2\_var:

Var[K][n] = L1\_var

elif L2\_var in a:

for m in range(len(Var[I])):

if Var[I][m] == L1\_var:

Var[I][m] = L2\_var

print("用%d" % I + "与%d" % K + "进行归结")

print("用" + "v".join(S[I]) + "与" + "v".join(S[K]) + "进行归结")

# 消解

del S[I][J]

del Var[I][J]

del S[K][L]

del Var[K][L]

# 为了保证最后一个子句的最后一个文字是Answer

if I == -1:

S[K] = S[K] + S[I]

Var[K] = Var[K] + Var[I]

else:

S[I] = S[I] + S[K]

Var[I] = Var[I] + Var[K]

# 求解时，调整为出现Answer（A）时停止

if len(S[-1]) == 1 and Var[-1][-1] in a:

print(S)

print(Var)

print("答案是" + Var[-1][-1])

return 0

del\_self\_flag = decline\_self(I)

if K < I:

del\_self\_flag = 0

del S[K - del\_self\_flag]

del Var[K - del\_self\_flag]

print(S)

print(Var)

主函数部分，调整为：

if \_\_name\_\_ == '\_\_main\_\_':

# -------这部分与之前一致------

read\_set()

print("S")

print(S)

print("Var")

print(Var)

flag = 1

while True:

if flag == 0:

break

for i in range(len(S)):

# print("i%d" % i)

if flag == 0:

break

if i >= len(S):

continue

for j in range(len(S[i])):

if flag == 0:

break

if i >= len(S):

continue

if j >= len(S[i]):

continue

# print("j%d" % j)

for k in range(i + 1, len(S)):

if flag == 0:

break

if k >= len(S):

continue

# print("k%d" % k)

for l in range(len(S[k])):

if k >= len(S):

continue

if l >= len(S[k]):

continue

# print("l%d" % l)

if opposite(S[i][j]) == S[k][l]:

flag = decline(i, j, k, l)

if flag == 1:

continue

elif flag == 0:

break

# 为了更快归结出A（person）让子句尝试与含有A(x)的子句归结

if opposite(S[-1][0]) == S[k][l]:

flag=decline(-1, 0, k, l) #上面已将含有A(x)的子句固定在子句集最后

if flag == 1:

continue

elif flag == 0:

break

# 5实验结果

## 5.1求解实验

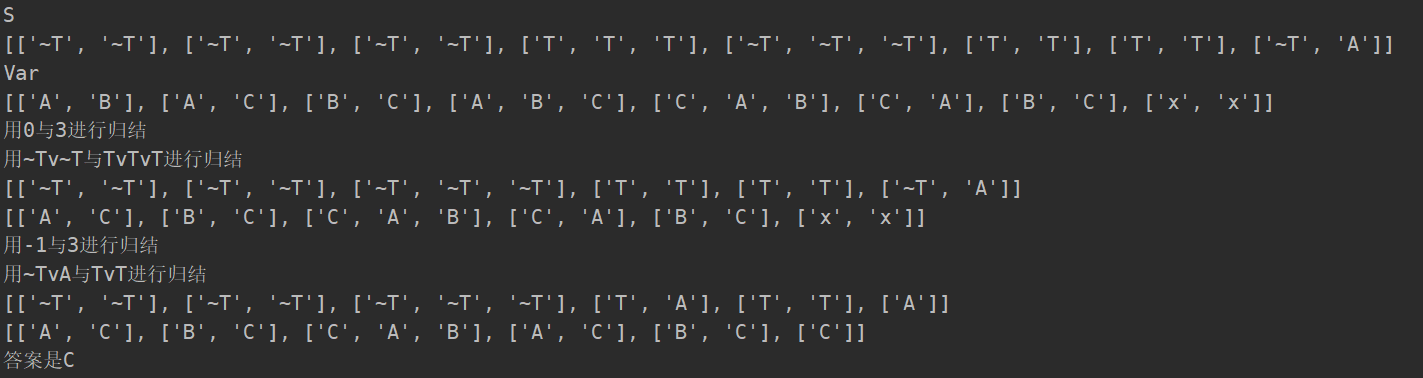
将﹁T(x) ∨ Answer（x）作为子句并入S进行归结推理，最后Answer的项为C，说明C才是老实人。

图1 求解实验结果

## 5.2证明实验

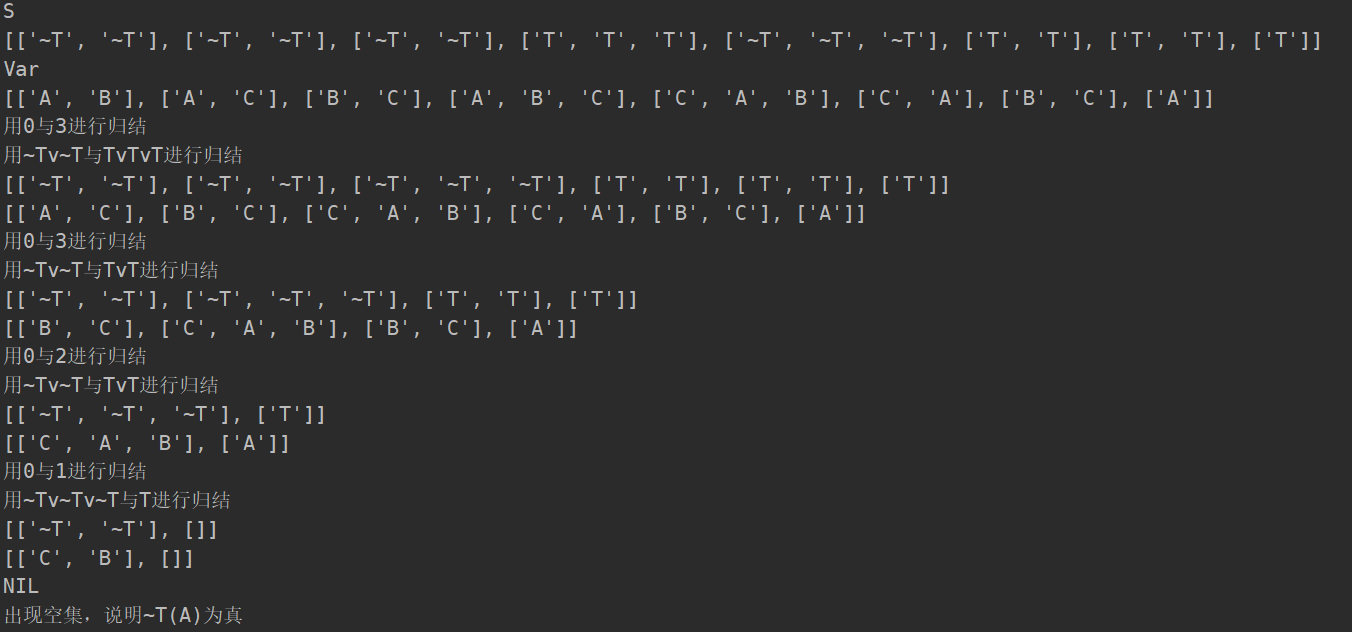
证明A不是老实人，则有﹁T(A)，把它的否定﹁(﹁T(A)) ，即T(A)作为子句加入S。最后归结出空子句，说明T(A)为假，即A不是老实人。

图2 证明实验结果