中山大学计算机学院

人工智能

本科生实验报告

（2022 学年春季学期）

课程名称：Artificial Intelligence

教学班级	21 级计科 3 班	专业（方向）	计算机科学与技术
学号	21307185	姓名	张礼贤

一、 实验题目

Python 实现归结

二、 实验内容

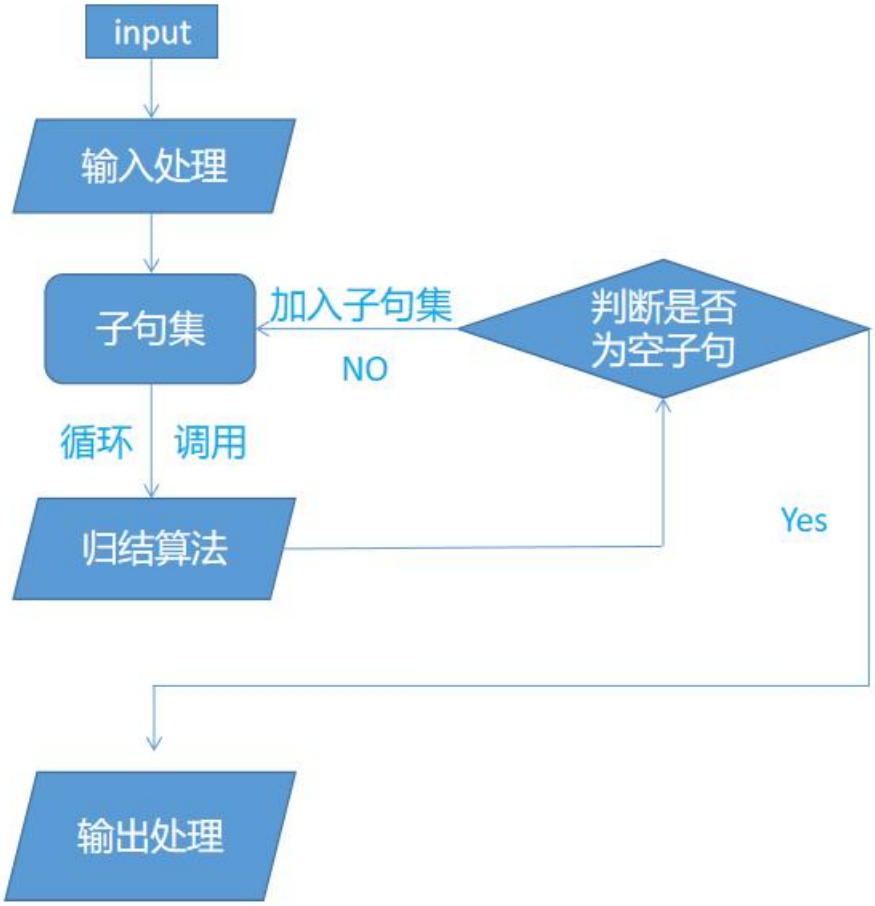
1. 算法原理

1、**输入处理：**先将输入放到文件中，以读文件的方式逐行读入，并对每一行进行处理，分理出每个谓词，并初始化谓词中的各个元素（如正 or 负文字、谓词名、参数列表、行和列等）。后将一行中的所有谓词存入列表中形成一个子句，为了后续算法的实现方便，使用树节点方式存储，即将每个子句存入 TreeNode 中的 val 中，左右儿子都置空。依照此方式将文件每一行都进行处理，并将生成的树节点加入总的列表中，相当于子句集。

2、**匹配算法：**使用内外双循环，先将第一行子句与后面的子句进行匹配。在匹配过程中，按照书上的匹配算法进行，需要将变量用实例进行替换，并用 replace 字典进行记录，方便在更新子句集和输出的时候进行替换查询和替换。之后将新生成的子句构造成树节点，并将两个生成该子句的节点用左儿子和右儿子进行连接，并加入到子句集的末尾。之后依次这样遍历，直到生成一个空子句。此时，已经生成了一棵以[ ]为 val 值的有向无环图（本质上是二叉树，但是可能会存在一个节点有多个父亲的情况，因此是有向无环图），之后从上往下进行遍历即可得到参与归结的子句，这时候要注意去重，不能重复。

3、**输出处理：**因为在输出的时候要将参与归结的字句的行和所在子句的位置进行输出；并且要输出变量的替换实例，以及新生成的子句。这些内容都被记录在谓词结构体和 TreeNode 结构体中，方便输出的进行。由于在子句集合里面有很多子句并未参与到归结，于是在输出的时候要考虑将没有用到的子句进行标记，即 wasted。这时候需要将用到的子句的行数进行前移，方便做输出处理，同时保持进行归结的两个子句的相对位置不变。

2. 伪代码与流程图



3. 关键代码展示（带注释）

```
import numpy
import copy
class Predicate:
    """表示谓词的类"""
    def __init__(self,name='',arr=[],row=0,col='a',flag=True):
        self.name=name      #谓词名
        self.arr=arr        #存储变量或实例
        self.row=row        #谓词所在的行
        self.col=col        #谓词在子句中的位置
        self.flag=flag      #正或负文字
        self.function=[]    #存储谓词中函数的列表

class TreeNode:
    def __init__(self,val,left = 0,right = 0,replace = {},r_map = {},parent = [],parent_col = []):
        self.val=val                #存储子句
        self.left=left              #左儿子
        self.right=right            #右儿子
        self.replace=replace        #变元替换的实例
```

```
        self.r_map=r_map            #函数映射
        self.parent=parent          #由哪两句归结而来
        self.parent_col=parent_col  #如果为 0 则表示一行中只有一个谓词
        self.wasted = False         #表示是否用于归结
```

```
def read_line(s,row):
    """读取一行的字符串并转换为合适的数据结构"""
    remain = 0
    stack_predicate = []    #存储谓词的栈，模拟递归地读入谓词
    stack_function = []     #存储函数的栈，模拟递归地读入函数
    while (' ' in s):s=s.replace(' ',"")
    if(s[0]=='('):s = s[1:-1]   #如果左右两边有左括号和右括号，则去掉
    col = -1        #初始化 col 计数子
    res=[]          #存储子句集合
    p=0             #遍历字符串的指针
    while (p < len(s)):
        if(s[p] == '('):    #如果遇到左括号则将 remain + 1 并 continue
            remain += 1
            p += 1
            continue
        if(s[p] == ')'):    #如果遇到右括号则将 remain - 1 并 continue
            remain -= 1
            p += 1
            continue
        if(remain==0 and s[p]==','):    #当 remain == 0 表示在谓词层面，如果遇到逗号，则表示谓词的分割，将谓词加入子句
            p += 1
            res.append(stack_predicate[-1])
            continue
        if(remain == 1 and s[p] == ','): #如果 remain==1，则表示在变量层面，此时直接继续
            p+=1
            continue
        if(remain == 0):    #谓词名层面
            temp = Predicate()  #初始化谓词
            col += 1
            if(s[p] == '~'):    #如果有'~'则为负文字
                temp.flag = False
                p += 1
            j = p
            while (s[j]!='('):  #遍历到左括号获取谓词名
                j+=1
            temp.name = s[p:j]  #使用切片进行获取
            temp.row = row
            temp.col = chr(ord('a')+col)
```

```
            temp.arr=[]         #初始化谓词变量列表
            p=j-1
            stack_predicate.append(temp)
        if(remain == 1): #谓词变元层面
            j = p
            flag = True
            temp = stack_predicate[-1]  #获取谓词栈中栈顶元素
            while(s[j]!=')'):
                if(s[j] == '('):
                    flag = False
                    break
                j+=1
            if(flag): #表示变元中没有函数
                cur = s[p:j]
                array = cur.split(',')  #如果没有函数存在，则可直接根据','进行字符串切分获取每一个字
符

                for a in array:
                    temp.arr.append(a)
                    temp.function.append({})    #在每一个位置都安放一个字典，为以后的函数映射做准备
                p=j-1
            else:   #表示变元中有函数
                k = p
                while(s[k]!=',' and s[k]!=')'):k+=1
                word = s[p:k]
                if('(' in word):    #表示该项中有括号即有函数
                    k = p
                    while(s[k]!='('):k+=1
                    fun = s[p:k]    #获取函数名
                    temp.arr.append(fun)
                    temp.function.append({})
                    stack_function.append(fun)
                    p = k-1
                else :  #如果没有括号则直接将 word 读入谓词结构体的参数列表中
                    temp.arr.append(word)
                    temp.function.append({})
                    p = k-1
        if(remain >= 2):    #函数层面，利用循环营造非递归调用压栈读取形式
            temp=stack_predicate[-1]
            j = p
            while(s[j]!=')'):j+=1
            word = s[p:j]
            if('(' in word):
                fun_pre = stack_function[-1]    #获取之前的函数名，方便使用映射
                k = p
```

```
                while(s[k]!='('):k+=1
                fun = s[p:k]
                temp.function[-1][fun_pre]=fun
                stack_function.append(fun)
                p = k-1
            else :
                fun = stack_function[-1]    #获取栈顶的函数名
                temp.function[-1][fun]=word
                p = j-1
        p+=1
    res.append(stack_predicate[-1])
    return res
```

```
def reverse_function_map(function):
    """反转映射函数字典的键值对"""  #方便进行归结匹配
    func = {v:k for k,v in function.items()}
    return func
```

```
def create_list():
    """将文件输入转化为列表存储"""
    f=open('test4.txt','r')
    count_row=1     #计算行数
    store=[]
    for line in f:
        line=line.strip()
        s=line
        array = read_line(s,count_row)
        for i in range(len(array)):
            for j in range(len(array[i].arr)):
                x = array[i].arr[j]
                while (x in array[i].function[j]):x = array[i].function[j][x]
                array[i].function[j]=reverse_function_map(array[i].function[j])
                array[i].arr[j] = x
        store.append(array)
        count_row += 1
    return convert_TreeNode_list(store)
```

```
def convert_TreeNode_list(store):
    """将每个字句存储在树的节点中,并将所有树的节点存放到一个列表中并返回"""
    TreeNode_store=[]
    for i in range(len(store)):
        line = store[i]
        root = TreeNode(line)   #使用默认参数列表进行构造
        TreeNode_store.append(root)
```

```
    return TreeNode_store
```

```
def is_same_predicate(node_a,node_b):
    """判断两个谓词是否相同"""
    if(node_a.name != node_b or len(node_a.arr) != len(node_b.arr) or node_a.flag !=
node_b.flag):return False
    for i in range(len(node_a.arr)):
        if(node_a.arr[i]!=node_b.arr[i]) : return False
    return True
```

```
def is_in_line(node,line):
    """判断一个谓词是否在字句集中"""
    for l in line:
        if(is_same_predicate(node,l)):return True
    return False
```

```
def construct(left,right,replace,index,cur_len):
    """创建归结后的列表,并返回节点"""
    if(len(right)): #如果第二个子句中还剩余谓词，就将其插入第一个字句中的相应位置
        for i in range(len(right)):
            if(not is_in_line(right[i],left)):
                left.insert(index,right[i])
                index+=1
    for i in range(len(left)):
        left[i].col = chr(ord('a')+i)
        left[i].row = cur_len           #更新行和列
        for j in range(len(left[i].arr)):
            if(left[i].arr[j] in replace):  #利用 replace 字典替换变量为实例
                left[i].arr[j] = replace[left[i].arr[j]]
    return left
```

```
def bfs(root):
    """通过层序遍历即广度优先搜索生成匹配序列"""
    store = []
    store.append(root)
    res = []
    while(len(store)):
        temp = store[0]     #获取队首元素
        store.pop(0)        #弹出队首元素
        if (temp.left != 0):store.append(temp.left)
        if (temp.right != 0):store.append(temp.right)
        res = res + temp.parent            #这里可能出现一个节点是多个节点的父亲，后面会进行去重操作
    return res
```

```
def match_line(root1,root2,store_len):
    """匹配子句集,按逐行方式匹配"""
    line1 = root1.val
    line2 = root2.val
    for k in range(len(line1)):
        for j in range(len(line2)):
        #依次获取每一行中的每一个谓词，进行匹配
            node1=line1[k]
            node2=line2[j]
            replace={}  #合一算法用到的变量替换实例的映射字典
            r_map = {}
            if(node1.name != node2.name or len(node1.arr) != len(node2.arr) or
node1.flag==node2.flag):continue
            #如果不满足以上条件则直接跳过
            flag = True
            for i in range(len(node1.arr)):
                if (len(node1.arr[i])<len(node2.arr[i]) and len(node1.arr[i]) == 1 and node1.arr[i]
not in node1.function[i]):
                #如果参数名的长度较小且为 1 则为变量
                    replace[node1.arr[i]]=node2.arr[i]
                elif (len(node1.arr[i])>len(node2.arr[i]) and len(node2.arr[i]) == 1 and
node2.arr[i] not in node2.function[i]):
                    replace[node2.arr[i]]=node1.arr[i]
                else :
                #如果变量的长度相等则进行变量之间的匹配和实例之间的匹配
                    if(node1.arr[i]!=node2.arr[i] and len(node1.arr[i])>1):
                        flag = False
                    elif (node1.arr[i]!=node2.arr[i] and len(node1.arr)==1):
                        #如果有函数的情况则判断是否可以匹配
                        if(node1.arr[i] in node1.function[i] and node2.arr[i] not in
node2.function[i]):
                            r = node1.arr[i]
                            if(r in node1.function[i]):replace[node2.arr[i]] = r
                            while(r in node1.function[i]):
                                r_map[r] = node1.function[i][r]
                                r = node1.function[i][r]
                        elif(node2.arr[i] in node2.function[i] and node1.arr[i] not in
node1.function[i]):
                            r = node2.arr[i]
                            if(r in node2.function[i]):replace[node1.arr[i]] = r
                            while (r in node2.function[i]):
                                r_map[r] = node2.function[i][r]
                                r = node2.function[i][r]
```

```
            if(flag):
                line1_temp=copy.deepcopy(line1)
                line2_temp=copy.deepcopy(line2) #进行深拷贝，防止对原列表中的元素进行修改
                index1=line1.index(node1)
                index2=line2.index(node2)
                line1_temp.pop(index1)
                line2_temp.pop(index2)      #弹出已经进行匹配的谓词
                root_val = construct(line1_temp,line2_temp,replace,index1,store_len+1)
                col1=0
                col2=0                          #为了避免一个子句中只有一个谓词而错误加上 abc 等
                if(len(line1)>1):col1=index1+1
                if(len(line2)>1):col2=index2+1
                root =
TreeNode(root_val,root1,root2,replace,r_map,[node1.row,node2.row],[col1,col2])
                #创建根节点
                root.left = root1
                root.right = root2
                if(not in_store(root,store)):
                    store.append(root)      #判断新生成的 root_node 在不在 store 中，如果在的话则去掉，
否则加入到末尾
                return True
    return False
```

```
def remove_trash(store,bfs_list,initial_len):
    """将没有参与匹配的字句进行标记和对子句列表进行重规划行"""
    bfs_list.sort()
    map = {}        #表示归结所用到的子句行在输出时需要减去的行数
    count =0
    for i in range(len(store)):
        if(i+1 <= initial_len):
            map[i+1] = 0
        elif(i+1 in bfs_list):
            map[i+1] = count
        else:
            count += 1
            store[i].wasted = True      #不在遍历序列中，则舍弃，在输出的时候不显示
    return map
```

```
def in_store(node,store):
    """判断一个树节点是否在 store 列表中"""
    flag = False
    for s in store:
        if(len(node.val) != len(s.val) ):continue
        for i in range(len(node.val)):
```

```
            count = 0
            if(len(node.val[i].arr) != len(s.val[i].arr)):break
            for j in range(len(node.val[i].arr)):
                if(node.val[i].arr[j] == s.val[i].arr[j]):
                    count+=1
            if(node.val[i].name == s.val[i].name and node.val[i].flag == s.val[i].flag and count
== len(node.val[i].arr)):
                flag=True
    return flag
```

```
def create_root(store):
    """将 store 中的树节点依次匹配,并添加到 store 的末尾,最终生成一个空列表树节点"""
    length=len(store)
    i=0
    while(i<length-1):
        """注意:这里如果使用 for i in range(len(store))的话,不会实时更新,导致有很多子句没有被遍历到"""
        j=i+1
        while(j<length):
            match_line(store[i],store[j],len(store))
            if(len(store[-1].val)==0):#如果最后生成了一个空列表，则返回
                return store[-1]
            j=j+1
        store[i].visited=True
        i=i+1
        length=len(store)
    return store[-1]
```

```
def get_parent_list(root,map,size):
    """获取双亲节点,便于答案的生成"""
    store = []  #队列，进行层序遍历
    store.append(root)
    array = []
    while(len(store)):
        temp = store[0]
        store.pop(0)
        if(temp.left != 0):store.append(temp.left)
        if(temp.right != 0):store.append(temp.right)
        if(len(temp.parent)==0):continue
        if(temp.parent_col[0]):s =
str(temp.parent[0]-map[temp.parent[0]])+chr(temp.parent_col[0]-1+ord('a'))
        else :s=str(temp.parent[0]-map[temp.parent[0]])
        s += ","
        if(temp.parent_col[1]):s +=
str(temp.parent[1]-map[temp.parent[1]])+chr(temp.parent_col[1]-1+ord('a'))
```

```
        else :s += str(temp.parent[1]-map[temp.parent[1]])
        cur = 0
        if(len(temp.val)==0):cur = size
        else: cur = temp.val[0].row
        node = [cur,s]
        flag = True
        for arr in array:
            if(s==arr[1]):flag=False
        if(flag):array.append(node)  #列表中的第一个元素表示生成的新的归结子句在第几行，第二个元素表示由哪几个谓词归结得来
    array.sort()            #按照生成行的顺序进行排序
    return array
```

```
def flush_store(store,map):
    """刷新 store 列表,将用到的归结子句的行进行更改"""
    for i in range(len(store)):
        line = store[i].val
        for j in range(len(line)):
            if(line[j].row in map):
                line[j].row -= map[line[j].row]
```

```
def print_list(store):
    """打印带结构体的列表"""
    for i in range(len(store)):print_line(store[i].val)
```

```
def print_line(line):
    """打印子句"""
    flag = len(line)>1
    if(flag):print("(",end="")
    for i in range(len(line)):
        print_predicate(line[i])
        if(i<len(line)-1):print(",",end="")
    if(flag):print(")",end="")
    print()
```

```
def print_predicate(node):
    """打印谓词"""
    if(node.flag==False):print("~",end="")
    print(node.name,"(",end="",sep="")
    for i in range(len(node.arr)):
        print_function(node.arr[i], node.function[i])
        if( i < len(node.arr)-1):
            print(",",end="")
    print(")",end="")
```

```
def print_function(fun,func):
    """打印函数"""
    x = fun
    while(x in func):x = func[x]
    map = reverse_function_map(func)
    fun = x
    print(fun,end="")
    left = 0
    while(fun in map):
        print('(',map[fun],end="",sep="")
        fun = map[fun]
        left += 1
    while(left):
        print(')',end="")
        left -= 1
```

```
def print_answer(store,array,map):
    """打印答案"""
    for k in range(len(array)):
        arr = array[k]
        print("R[",arr[1],"]",end="",sep="")
        node = store[arr[0]-1]
        if(node.replace):
            for k,v in node.replace.items():
                print("(",k,'=',sep="",end="")
                if(v in node.r_map):
                    print_function(v, node.r_map)
                else: print(v,end="")
                print(")",end="")
        print(" = ",end="")
        if(len(node.val)==0):print("[]")
        else: print_line(node.val)
```

```
"""主函数"""
store=create_list()
print_list(store)
size=len(store)
root = create_root(store)
map = remove_trash(store, bfs(root), size)
s = get_parent_list(root, map, len(store))
flush_store(store, map)
print_answer(store,s,map)
```

4. 创新点&优化（如果有）

创新点：

1、通过树节点存储子句，经过归结算法后，会形成一张有向无环图，此时再进行广度优先遍历，从根节点出发，便可以得到是从什么子句归结得来的。就避免了通过遍历去寻找该子句所在的行数。这样提高了算法检索的效率，也使得代码更加简明和清晰。

2、对于附加题的函数输入，在这里也做了一些处理，通过栈的方式对左右括号进行计数处理，当左括号超过右括号 x 时（x=0：谓词层面；x=1，参数列表层面；x>=2，函数层面），进行不同的操作。并通过字典来记录函数的映射，为了防止一个谓词中出现多个同名函数且映射值不同的情况，在每一个参数的位置都设置了 dict 来映射，防止出现一对多而被覆盖的情况。在进行匹配的时候，通过将映射列表进行反转，直接对函数中的变元进行匹配，从而不用考虑过多的因素。（但是这个也有缺点，只能进行单层函数的匹配，如果遇到嵌套的情况像附加题的第二题就无法应对了）

优化：

1、通过字典记录参与归结的子句前面有多少没有参与归结的子句，再利用该字典将参与归结的子句的行数进行前移，达到删去多余输出的操作，使得输出更加紧凑，不至于冗杂。

2、将各个模块函数化，虽然比较多且复杂，但是代码的可以执行良好，因此在上面做改动比较方便。每个函数都实现特定的功能。

三、 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

第一个实验:

```
A(tony)
A(mike)
A(john)
L(tony,rain)
L(tony,snow)
(~A(x),S(x),C(x))
(~C(y),~L(y,rain))
(L(z,snow),~S(z))
(~L(tony,u),~L(mike,u))
(L(tony,v),L(mike,v))
(~A(w),~C(w),S(w))
R[2,6a](x=mike) = (S(mike),C(mike))
R[2,11a](w=mike) = (~C(mike),S(mike))
R[5,9a](u=snow) = ~L(mike,snow)
R[8a,14](z=mike) = ~S(mike)
R[12a,15] = C(mike)
R[13a,16] = S(mike)
R[15,17] = []
```

第二个实验：

```
GradStudent(sue)
(~GradStudent(x),Student(x))
(~Student(x),HardWorker(x))
~HardWorker(sue)
R[1,2a](x=sue) = Student(sue)
R[3a,5](x=sue) = HardWorker(sue)
R[4,6] = []
```

第三个实验：

```
On(aa,bb)
On(bb,cc)
Green(aa)
~Green(cc)
(~On(x,y),~Green(x),Green(y))
R[1,5a](x=aa)(y=bb) = (~Green(aa),Green(bb))
R[2,5a](x=bb)(y=cc) = (~Green(bb),Green(cc))
R[3,6a] = Green(bb)
R[4,7b] = ~Green(bb)
R[8,9] = []
```

2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

时间复杂度：

由于在进行输出处理的时候采用了层序遍历获取匹配序列，故该代码段的时间复杂度为 O(2log(n))，即大致对总的谓词个数取对数，由于每次只会参与两个子句进行匹配，所以不会呈指数级增长，拥有相对良好的时间复杂度，体现了利用树结构的优势。

程序的时间复杂度主要集中在匹配算法中，设子句集中共有 m 条子句，每条子句有 n 条谓词，每个谓词共有 p 个参数列表。故如果按逐行匹配的循环算法进行匹配，则匹配过程中的时间复杂度为 O(m^2*n^2*p)

四、 思考题（附加题）

1.

对于第一个有单层函数输入的归结，可以采取 dict 映射的方法进行归结的简化，对每一个参数位置都设置一个 dict 进行映射，在函数输入的时候由 f->x，在进行归结的时候则进行反转，即 x->f 方便进行归结。之后便可以用 dict 进行输出的处理。下图为运行结果。

```
I(bb)
U(aa,bb)
~F(u)
(~I(y),~U(x,y),F(f(z)))
(~I(v),~U(w,v),E(w,f(w)))
R[1,4a](y=bb) = (~U(x,bb),F(f(z)))
R[2,6a](x=aa) = F(f(z))
R[3,7](u=f(z)) = []
```