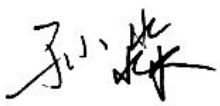
合肥工业大学
计算机与信息学院
实验报告

课　　程：人工智能原理

姓　　名： 

学　　号：2018211958

专业班级：计算机科学与技术 18-2 班

日　　期：10 月 25 日

目录

实验一 猴子摘香蕉问题的 Python 编程实现

一、 实验目的

(1)熟悉谓词逻辑表示法

(2)掌握人工智能谓词逻辑中的经典例子——猴子摘香蕉的编程实现

二、 实验内容

房子里有一只猴子(即机器人)， 位于 a 处。在 c 处上方的天花板上有一串香蕉，猴子想吃，但摘不到。房间的 b 处还有一个箱子, 如果猴子站到箱子上，就可以摸着天花板。如图 1 所示，对于.上述问题，可以通过谓词逻辑表示法来描述知识。要求通过 python 语言编程实现猴子摘香蕉问题的求解过程。

三、 实验原理

这个实验是利用一阶谓词逻辑求解猴子摘香蕉问题，我们需要：

定义描述状态的谓词：

AT(x, y)：x 在 y 处

ONBOX：猴子在箱子上

GB：猴子得到香蕉

个体域：

x：{monkey, box, banana}

Y：{a, b, c}

问题的初始状态

AT(monkey, a)

AT(box, b)

¬ONBOX, ¬GB

问题的目标状态

AT(monkey, c), AT(box, c)

ONBOX, GB

定义描述操作的谓词：

Goto(u, v)：猴子从 u 处走到 v 处

PushBox(v, w)：猴子推着箱子从 v 处移到 w 处

ClimbBox( )：猴子爬上箱子

Grasp( )：猴子摘取香蕉

各操作的条件和动作

Goto(u, v)

条件：¬ONBOX ， AT(monkey, u)，

动作：删除：AT(monkey, u)

添加：AT(monkey, v)

PushBox(v, w)

条件：¬ONBOX ， AT(monkey, v)， AT(box, v)

动作：删除：AT(monkey, v)， AT(box, v)

添加：AT(monkey, w)， AT(box,w)

ClimbBox( )

条件：¬ONBOX, AT(monkey, w)， AT(box,w)

动作：删除：¬ONBOX

添加：ONBOX

Grasp( )

条件：ONBOX， AT(box, c)

动作：删除：¬GB

添加：GB

我们不难得到实验代码应该为

Monkey_go_box(monkey, box)

Monkey_move_box(box, banana)

Monkey_on_box()

Monkey_get_banana()

PS：逻辑学基础

（1）命题和真值

一个陈述句称为一个断言。凡有真假意义的断言称为命题。命题的意义通常称为真值，它只有真、假两种情况。

（2）论域

也称为个体域，是由讨论的对象的全体构成的非空集合

（3）谓词

实现的是从个体域中的个体到 T 或 F 的映射。分为谓词名和个体两个部分

谓词名：表示个体的性质、状态或个体之间的关系，用大写英文字母表示

个体：命题中的主语，用小写英文字母表示。可以是常量、变元和函数

（4）函数

实现的是从一个个体到另一个个体的映射，函数没有真值。

在谓词逻辑中，函数本身不能单独使用，它必须嵌入到谓词之中。

举例：王洪的父亲是教师

TEACHER(father(Wang Hong))，其中，TEACHER 是谓词，而 father 是函数

（5）连接词和量词

连接词：¬，∨，∧，→，↔

量词：∀，∃

四、 实验代码

```
'''
猴子摘香蕉问题的 Python 编程实现
'''
#全局变量 i
i=0
def Monkey_go_box(x,y):
    global i
    i=i+1
    print('step:',i,'monkey 从',x,'走到'+y)

def Monkey_move_box(x,y):
    global i
    i = i + 1
    print('step:', i, 'monkey 把箱子从', x, '运到' + y)
```

```
def Monkey_on_box():
    global i
    i = i + 1
    print('step:', i, 'monkey 爬上箱子')

def Monkey_get_banana():
    global i
    i = i + 1
    print('step:', i, 'monkey 摘到香蕉')


import sys

#读取输入的运行参数，
codeIn=sys.stdin.read()
codeInList=codeIn.split()
#运行参数分别表示 monkey、banana、box 的位置，
monkey=codeInList[0]
banana=codeInList[1]
box=codeInList[2]
print('操作步骤如下：')
#请用最少步骤完成猴子摘香蕉任务
###########开始#############
Monkey_go_box(monkey, box)
Monkey_move_box(box, banana)
Monkey_on_box()
Monkey_get_banana()
###########结束#############
```

五、 实验结果

平台上的运行结果如下：

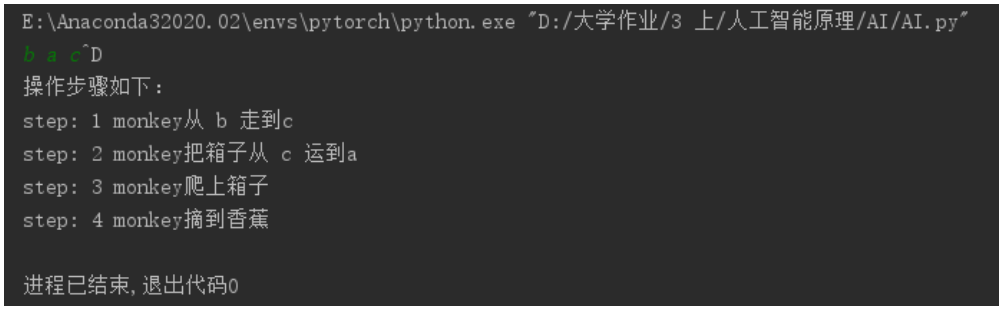
实际输出

```
操作步骤如下:
step: 1 monkey从 b 走到c
step: 2 monkey把箱子从 c 运到a
step: 3 monkey爬上箱子
step: 4 monkey摘到香蕉
```

Python 运行结果如下：

六、 心得体会

通过这次实验，我学会了如何用谓词来表示生活中的某些具体事务，并通过编程给出具体的操作步骤，感觉获益匪浅，在实验中也遇到了一些问题，比



如一开始用谓词描述的不准确，还有逻辑上也不知怎么表达，后来通过查找资料和对老师上课的内容进行复习，从而解决了这些问题，成功实现了预期的目标。

在上述过程中，我们应该注意，当猴子执行某-一个操作之前，需要检查当前状态是否可使所要求的条件得到满足，即证明当前状态是否蕴涵操作所要求的状态的过程。在行动过程中，检查条件的满足性后才进行变量的代换。代入新条件后的新状态如果是日标状态，则问题解决;否则看是否满足下面的操作，如果不满足或即使满足却又回到了原来的状态，那么代入无效。

此外，刚开始我转到 python 进行运行的时候老是不能出来结果，仔细看了实验要求发现老师给了详细的教程（含有输入的需要按 CTRL+D 运行），然后就顺风顺水了。

后来还研究了一下为什么要 CTRL+D 而不是像我平常写的 Python 程序直接回车就行了呢，原来是因为由于输入方式 codeIn=sys.stdin.read()和 codeInList=codeIn.split()的特殊性，所以如果是用 pycharm 编译器运行此代码，则输入 a b c（中间有空格）之后，需要按下 Ctrl+D 即可得到运行结果。

还有就是注意 Python 对缩进的严格要求要遵守，不然运行会报错的。

实验三 搜索算法求解 8 数码问题

一、 实验目的

(1)熟悉人工智能系统中的问题求解过程;

(2)熟悉状态空间中的盲目搜索策略;

(3)掌握盲目搜索算法，重点是宽度优先搜索和深度优先搜索算法。

二、 实验内容

用 Python 语言编程,采用宽度优先搜索求解 8 数码问题

采用宽度优先算法，运行程序，要求输入初始状态

假设给定如下初始状态 S0

2 8 3

1 6 4

7 0 5

和目标状态 Sg

2 1 6

4 0 8

7 5 3

验证程序的输出结果，写出心得体会。

(2)对代码进行修改(选作),实现深度优先搜索求解该问题

提示:每次选扩展节点时，从数组的最后一个生成的节点开始找，找一个没有被扩展的节点。这样也需要对节点添加一个是否被扩展过的标志。

三、 实验原理

盲目搜索按预定的控制策略进行搜索，搜索过程中获得的中间信息不用来改变搜索策略。搜索总是按预定的路线进行，不考虑问题本身的特性，这种搜索有盲目性，效率不高，不利于求解复杂问题。

宽度优先搜索(BFS-Breadth First Search)

由近及远逐层访问图中顶点（典型的层次遍历）。

1.节点深度：

起始节点 S（根节点，图中选定起始搜索顶点）深度为 0；其他节点等于父节点深度加 1。

2.基本思想

从初始节点 S 开始，依据到 S 的深度，逐层扩展节点并考察其是否目标节点。

在第 n 层节点没有完全扩展之前，不对第 n+1 层节点进行扩展。

即：OPEN 表排序策略为新产生的节点放到 OPEN 表的末端。

BFS 遍历搜索算法

从初始状态节点 S 出发宽度优先搜索遍历图的算法 bfs(S):

1) 访问 S

2) 依次访问 S 的各邻接点

3) 设最近一层访问序列为 v_{i1}，v_{i2}，…，v_{ik}，则依次访问 v_{i1}，v_{i2}，…，v_{ik} 的未被访问过的邻接点。

4) 重复（3），直到找不到未被访问的邻接点为止。

深度优先搜索(DFS-Depth First Search)

1.基本思想

从初始节点 S 开始，优先扩展最新产生的节点（最深的节点）。

即：OPEN 表排序策略为新产生的节点放到 OPEN 表的前端，优先扩展。

2.DFS 遍历搜索算法

从初始状态顶点 S 出发深度优先遍历图的方法 dfs (S)：

1) 访问 S——visit（S）；

2) 依次从 S 的未被访问过的邻接点出发进行深度遍历.

但是解决八数码问题实际上是状态空间的盲目搜索，此处先列出状态空间宽度优先搜索（实验的），然后列出状态空间深度优先搜索：

状态空间广度优先搜索

(1)把初始节点 S0 放入 Open 表中；

(2)如果 Open 表为空，则问题无解，失败退出；

(3)把 Open 表的第一个节点取出放入 Closed 表，并记该节点为 n；

(4)考察节点 n 是否为目标节点。若是，则得到问题的解，成功退出；

(5)若节点 n 不可扩展，则转第(2)步；

(6)扩展节点 n，将其子节点放入 Open 表的尾部，并为每一个子节点设置指向父节点的指针，然后转第(2)步。

状态空间深度优先搜索：

(1)把初始节点 S 放入 OPEN 表

(2)如果 OPEN 表为空,则问题无解,退出

(3)把 OPEN 表的第一个节点(记为节点 n)取出,放入 CLOSED 表

(4)考查节点 n 是否为目标节点.若是,则求得了问题的解,退出

(5)若节点 n 不可扩展,则转第(2)步

(6)扩展节点 n,将其子节点放入 OPEN 表的首部,并为每一个子节点都配置指向父节点的指针,转第(2)步

四、 实验代码

```
import numpy as np

class State:
    def __init__(self, state, directionFlag=None, parent=None):
```

```
            self.state = state
            # state is a ndarray with a shape(3,3) to storage the state
            self.direction = ['up', 'down', 'right', 'left']
            if directionFlag:
                self.direction.remove(directionFlag)
            # record the possible directions to generate the sub-states
            self.parent = parent


        def showInfo(self):
            for i in range(3):
                for j in range(3):
                    print(self.state[i, j], end='    ')
                print("\n")
            print('->')
            return

        def getEmptyPos(self):
            postion = np.where(self.state == self.symbol)
            return postion

        def generateSubStates(self):#产生子节点
            if not self.direction:
                return []
            subStates = []
            boarder = len(self.state) - 1
            # the maximum of the x,y
            row, col = self.getEmptyPos()
            if 'left' in self.direction and col > 0:#向左移动
                s = self.state.copy()
                #标志位 symbol=0 向左移动，产生新的状态节点，加入到
subStates 中
                temp = s.copy()
                s[row, col] = s[row, col-1]
                s[row, col-1] = temp[row, col]
                news = State(s, directionFlag='right', parent=self)
                subStates.append(news)
            if 'up' in self.direction and row > 0:
                s = self.state.copy()
                #  标志位 symbol=0 向上移动，产生新的状态节点，加入到
subStates 中
                temp = s.copy()
                s[row, col] = s[row-1, col]
                s[row-1, col] = temp[row, col]
```

```
                news = State(s, directionFlag='down', parent=self)
                subStates.append(news)
            if 'down' in self.direction and row < boarder:              #it can move to down
place
                s = self.state.copy()
                #  标志位 symbol=0 向下移动，产生新的状态节点，加入到
subStates 中
                temp = s.copy()
                s[row, col] = s[row+1, col]
                s[row+1, col] = temp[row, col]
                news = State(s, directionFlag='up', parent=self)
                subStates.append(news)
            if self.direction.count('right') and col < boarder:       #it can move to right
place
                s = self.state.copy()
                #  标志位 symbol=0 向右移动，产生新的状态节点，加入到
subStates 中
                temp = s.copy()
                s[row, col] = s[row, col+1]
                s[row, col+1] = temp[row, col]
                news = State(s, directionFlag='left', parent=self)
                subStates.append(news)
            #end1
            return subStates

        def BFS(self):
            # generate a empty openTable
            openTable = []   #存放状态的地方
            # append the origin state to the openTable
            openTable.append(self)#将初始状态加入
            steps = 1#步骤
            while len(openTable) > 0:
                n = openTable.pop(0)#pop()  函数用于移除列表中的一个元素（默
认最后一个元素），并且返回该元素的值。
                subStates = n.generateSubStates()

                #  查看子状态中有没有最终状态，如果有则输出之前的父状态到
path 中，输出 step+1
                ############开始 1##############
                for i in subStates:
                    if(State.answer==i.state).all():
                        path=[]
                        while i.parent!=None:
```

```
                                    if not
(np.array([[2,8,3],[1,6,4],[7,0,5]])==i.parent.state).all():
                                            path.insert(0,i.parent)
                                    i=i.parent
                                return path,steps+1
                        ###########结束 1#############

                        #  将子状态添加到 openTable 中
                        ###########开始 2#############
                        for i in subStates:
                                openTable.append(i)
                        steps+=1
                        ###########结束 2#############
                else:
                        return None, None
State.symbol = 0
State.answer = np.array([[1, 2, 3], [8, 0, 4], [7, 6, 5]])
s1 = State(np.array([[2, 8, 3], [1, 6 , 4], [7, 0, 5]]))
path, steps = s1.BFS()
if path:        # if find the solution
        for node in path:
                        # print the path from the origin to final state
                        node.showInfo()
        print(State.answer)
        print("Total steps is %d" % steps)
```

五、 实验结果

平台上的运行结果如下：

实际输出

```
2 8 3

1 0 4

7 6 5

->
2 0 3

1 8 4

7 6 5

->
0 2 3

1 8 4

7 6 5

->
1 2 3

0 8 4

7 6 5

->
[[1 2 3]
[8 0 4]
[7 6 5]]
Total steps is 27
```

Python 运行结果如下：

```
E:\Anaconda32020.02\envs\pytorch\python.exe "D:/大学作业/3 上/人工智能原理/AI/AI.py"
2  8  3

1  0  4

7  6  5

->
2  0  3

1  8  4

7  6  5

->
0  2  3

1  8  4

7  6  5

->
1  2  3

0  8  4

7  6  5

->
[[1 2 3]
 [8 0 4]
 [7 6 5]]
Total steps is 27

进程已结束,退出代码0
```

六、 心得体会

在本次实验中最大的问题就是对于两个算法的理解和实现，比如是迭代还是递归，还是回溯，都是我们值得去思考和理解的。其实，把数据结构里面的原理理解清楚了，这两个算法其实不难的。

总结一下就是：

深度优先遍历：对每一个可能的分支路径深入到不能再深入为止，而且每个结点只能访问一次。要特别注意的是，二叉树的深度优先遍历比较特殊，可以细分为先序遍历、中序遍历、后序遍历（我们前面使用的是先序遍历）。它们三个的区别如下：

先序遍历：对任一子树，先访问根，然后遍历其左子树，最后遍历其右子树。

中序遍历：对任一子树，先遍历其左子树，然后访问根，最后遍历其右子树。

后序遍历：对任一子树，先遍历其左子树，然后遍历其右子树，最后访问根。

广度优先遍历：从上往下对每一层依次访问，在每一层中，从左往右（也可以从右往左）访问结点，访问完一层就进入下一层，直到没有结点可以访问为止。

为了了解更深，我还翻出来以前的数据结构书，找到里面对深度和广度优先算法的分析比较：

深度优先搜素算法：不全部保留结点，占用空间少；有回溯操作(即有入栈、出栈操作)，运行速度慢。

广度优先搜索算法：保留全部结点，占用空间大； 无回溯操作(即无入栈、出栈操作)，运行速度快。

通常深度优先搜索法不全部保留结点，扩展完的结点从数据库中弹出删去，这样，一般在数据库中存储的结点数就是深度值，因此它占用空间较少。

所以，当搜索树的结点较多，用其它方法易产生内存溢出时，深度优先搜索不失为一种有效的求解方法。

广度优先搜索算法，一般需存储产生的所有结点，占用的存储空间要比深度优先搜索大得多，因此，程序设计中，必须考虑溢出和节省内存空间的问题。

但广度优先搜索法一般无回溯操作，即入栈和出栈的操作，所以运行速度比深度优先搜索要快些

其次就是 anaconda 里面自带的 numpy 版本实在太低了，import numpy as np 会运行报错，得更新到最新的 numpy，研究出这个报错的原因花了我半天的时间，实验室的师兄后来推荐我可以用 anaconda，但是最好不要用里面的环境，因为下载 anaconda 时附带下载的库都是很低版本的。

本次实验在很大程度上让我进一步了解了这两种算法的区别和原理，对专业课数据结构，计算机网络，离散数学，操作系统这些涉及到深度和宽度优先算法的学科都有着触类旁通的作用。

此外，根据查阅资料和实际测试我还发现：对于一些简单的八数码问题，宽度优先算法可以比较快得找到目标，但是对于一些复杂的步数较多的问题，宽度优先搜索的的效率很低，比如当初始状态为：[2,4 8,6，0,3,1,7,5],目标状态为：[1,2,3,8,0,4,7,6,5]时的八数码问题，当最大深度设置为 10 时，宽度优先搜索无法找到目标状态。解决办法是设置最大深度为 15 时，这样就可以找到目标状态。

另外的小收获就是我在解决问题的过程中提高了 Python 编程能力，对用编程解决实际问题有了更加深刻的认识。

此次实验我采用了宽度优先算法，成功的完成了本次实验，我对宽度优先算法有了更为深刻的理解，我相信我再经过几次宽度优先算法的练习，我就可以用它来解决一些生活中的实际问题。

做完这个实验，我们可以总结出八数码问题的广度和深度优先搜索解决方案：

八数码问题的宽度优先搜索步骤如下：

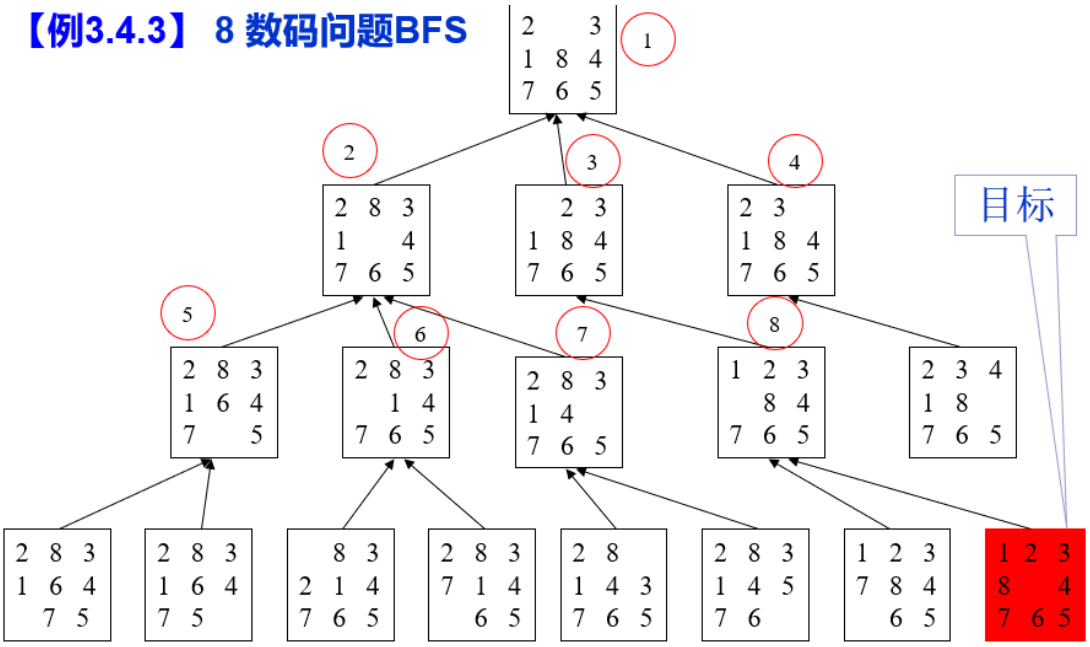
1、判断初始节点是否为目标节点，若初始节点是目标节点则搜索过程结束；若不是则转到第 2 步；

2、由初始节点向第 1 层扩展，得到 3 个节点：2、3、4；得到一个节点即判断该节点是否为目标节点，若是则搜索过程结束；若 2、3、4 节点均不是目标节点则转到第 3 步；

3、从第 1 层的第 2 个节点向第 2 层扩展，得到节点 8；从第 1 层的第 1 个节点向第 2 层扩展，得到 3 个节点：5、6、7；从第 1 层的第 3 个节点向第 2 层扩展得到节点 9；得到一个节点即判断该节点是否为目标节点，若是则搜索过程结束；若 6、7、8、9 节点均不是目标节点则转到第 4 步；

4、按照上述方法对下一层的节点进行扩展，搜索目标节点；直至搜索到目标节点为止。

5、搜索过程见下图所示。



八数码问题的深度优先搜索步骤如下：

1、设置深度界限，假设为 5；

2、判断初始节点是否为目标节点，若初始节点是目标节点则搜索过程结束；若不是则转到第 2 步；

3、由初始节点向第 1 层扩展，得到节点 2，判断节点 2 是否为目标节点；若是则搜索过程结束；若不是，则将节点 2 向第 2 层扩展，得到节点 3；

4、判断节点 3 是否为目标节点，若是则搜索过程结束；若不是则将节点 3 向第 3 层扩展，得到节点 4；

5、判断节点 4 是否为目标节点，若是则搜索过程结束；若不是则将节点 4 向第 4 层扩展，得到节点 5；

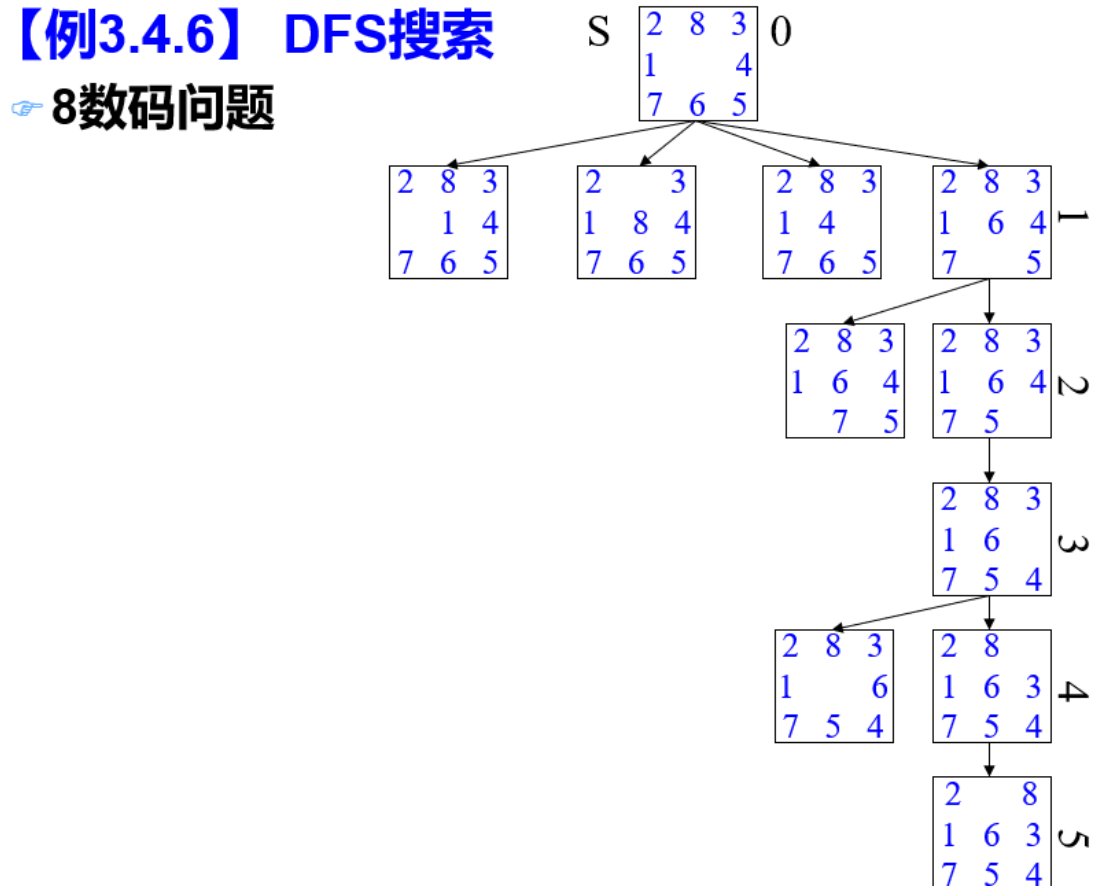
6、判断节点 5 是否为目标节点，若是则搜索过程结束；若不是则结束此轮搜索，返回到第 2 层，将节点 3 向第 3 层扩展得到节点 6；

7、判断节点 6 是否为目标节点，若是则搜索过程结束；若不是则将节点 6 向第 4 层扩展，得到节点 7；

8、判断节点 7 是否为目标节点，若是则结束搜索过程；若不是则将节点 6 向第 4 层扩展得到节点 8；

9、依次类推，知道得到目标节点为止。

10、搜索过程见下图所示。



实验四 字句集消解实验

一、 实验目的

(1)熟悉子句集化简的九个步骤;

(2)理解消解规则，能把任意谓词公式转换成子句

集。

二、 实验内容

在谓词逻辑中，任何一个谓词公式都可以通过应用等价关系及推理规则化成相应的子句集。

用 python 实现以下 6 个步骤

其化简步骤如下: .

(1)消去连接词

“→”和↔

反复使用如下等价公式:

P→Q↔P∨Q

P↔Q↔ (P∧Q) ∨(¬P∧¬Q)

即可消去谓词公式中的连接词“→"和"↔"

(2)减少否定符号的辖域

反复使用双重否定率

¬(¬ P) ↔P

摩根定律

- -(P∧Q) ↔-P∨-Q

- -(P∨Q) ↔P∧-Q

将每个否定符号"-”移到仅靠谓词的位置，使得每

个否定符号最多只作用于一个谓词上。

(3)对变元标准化

在一个量词的辖域内，把谓词公式中受该量词约束的

元全部用另外一个没有出现过的任意变元代替，使
不同量词约束的变元有不同的名字。

(4)化为前束范式

化为前束范式的方法:把所有量词都移到公式的左边,
并且在移动时不能改变其相对顺序。

(5)消去存在量词

(6)化为 Skolem 标准形

对上述前束范式的母式应用以下等价关系

$P \vee (Q \wedge R) \leftrightarrow (P \vee Q) \wedge (P \vee R)$

(7)消去全称量词

(8)消去合取词

在母式中消去所有合取词，把母式用子句集的形式表
示出来。其中，子句集中的每一个元素都是一个子
句。

(9)更换变量名称

对子句集中的某些变量重新命名,使任意两个子句中
不出现相同的变量名。

三、 实验原理

1. 文字

原子谓词公式及其否定统称为文字。

例如：P(x)、Q(x)、 ¬ P(x)、 ¬ Q(x)等都是文字。

2.子句 (Clause)

单个文字或任何文字的析取公式称为子句。

例如，

P(x)

$P(x) \vee Q(x)$

$P(x, f(x)) \vee Q(x, g(x))$ 都是子句。

3. 空子句

不含任何文字的子句称为空子句。

由于空子句不含有任何文字，也就不能被任何解释所满足，因此空子句是永假的，不可满足的。

4. 子句集

由子句构成的集合称为子句集。

在谓词逻辑中，任何一个谓词公式都可以通过应用等价关系及推理规则化成相应的子句集。

5. 谓词公式的子句集表示

任何一个谓词公式都可以化为合取范式，合取的每一个部分就是一个子句，所有子句组成这个谓词公式的子句集合，我们将这个子句集合叫做谓词公式的子句集表示。

下面我们结合实例介绍将谓词公式化为子句集表示的一般步骤。设有公式：

$(\forall x)((\forall y)P(x,y)\rightarrow \neg\ (\forall y)(Q(x,y)\rightarrow R(x,y)))$

(1)消去蕴涵符号

使用等价公式　$P\rightarrow Q \Leftrightarrow \neg\ P\vee Q$　消去谓词公式中所有的蕴含连接词。

例如公式：

$(\forall x)((\forall y)P(x,y)\rightarrow \neg\ (\forall y)(Q(x,y)\rightarrow R(x,y)))$

消去蕴含后等价变化为：

$(\forall x)(\neg(\forall y)P(x,y)\vee \neg\ (\forall y)(\neg Q(x,y)\vee R(x,y)))$

(2) 缩小否定符号的作用范围 （辖域）

反复使用以下推理规则，将每个否定符号“$\neg$”移到紧靠谓词的位置，即使得每个否定符号只作用于一个谓词上。

双重否定律

$\neg(\neg P) \Leftrightarrow P$

德摩根定律

$\neg(P\wedge Q) \Leftrightarrow \neg P\vee \neg Q$

$\neg(P\vee Q) \Leftrightarrow \neg P\wedge \neg Q$

量词转换律

¬ (∀x)P(x) ⇔ (∃x) ¬P(x)

¬ (∃x)P(x) ⇔ (∀x)¬P(x)

例如，

(∀x)(¬(∀y)P(x,y)∨¬ (∀y)(¬Q(x,y)∨R(x,y)))

上式经等价变换后为：

(∀x)((∃y)¬P(x，y)∨(∃y)(Q(x，y) ∧¬R(x，y)))

(3) 变量命名标准化

在一个量词的辖域内，把谓词公式中受该量词约束的变元全部用另外一个没有出现过的任意变元代替，使不同量词约束的变元有不同的名字。

例如，

(∀x)((∃y)¬P(x，y)∨(∃y)(Q(x，y) ∧¬R(x，y)))

上式经变换后为

(∀x){(∃y)¬P(x，y)∨(∃z)[Q(x，z) ∧¬R(x，z)]}

(4) 消去存在量词(Skolem 化)

引入 Skolem 函数，消去存在量词，分为两种情况。

① 形如 (∀x) [(∃y)P(x,y)]，存在量词在全称量词的辖域内

a) 存在量词 y 可能依赖于 x，令这种依赖关系为 y=f(x)，由 f(x)把每个 x 值映射到存在的那个 y，f(x)叫做 Skolem 函数。

引入 Skolem 函数 f(x)后，上面的公式变为：

(∀x) [P(x, f(x)]

② 形如 (∃y)(∀x)P(x,y)，存在量词前没有全称量词约束

a) 这种情况比较容易理解，即在变量 y 的论域内，存在一些特定的取值，使得公式为真，那么我们就可以用 y 论域上使公式为真的一个特定取值--常量，来替代变量 y，从而消去存在量词。所以这种情况的 Skolem 函数为一常量，即用变量 y 论域内的，使得公式为真的特定取值的一个常量符号替代原来的变量 y，这个常量符号不能与公式中的其它符号同名，也就是没有在公式中出现过。

例上面的公式，令 y=A，Skolem 化后，变为：(∀x) [P(x, A)]

(5) 化为前束范式

将公式中的所有约束量词移到公式最前面，形成前束公式，形式如下：

前束式=（约束量词前缀）（母式）

在移动时不能改变其相对顺序。

由于第(3)步已对变元进行了标准化，每个量词都有自己的变元，这就消除了任何由变元引起冲突的可能，即：移动不改变量词辖域。

例：前例已经是前束范式，

$(\forall x)\{\neg P(x, f(x))\vee[Q(x,g(x))\wedge\neg R(x,g(x))]\}$

(6) 消去全称量词

由于母式中的全部变元均受全称量词的约束，并且全称量词的次序已无关紧要，因此可以省掉全称量词。

剩下的母式，默认所有变元受全称量词约束。

例如，上式消去全称量词后为

$\{\neg P(x, f(x))\vee[Q(x,g(x))\wedge\neg R(x,g(x))]\}$

(7) 化为合取范式

反复运用谓词公式的结合律和分配律，将公式化为合取范式。

分配率

$P\wedge(Q\vee R) <=> (P\wedge Q) \vee (P\wedge R)$

$P\vee(Q\wedge R) <=> (P\vee Q) \wedge (P\vee R)$

结合率

$(P\vee Q)\vee R <=> P\vee(Q\vee R)$

$(P\wedge Q)\wedge R <=> P\wedge(Q\wedge R)$

(8) 将公式用子句集合表示

取出合取范式中的每个子句；

对子句变量换名，使任意两个子句中不出现相同的变量名，便于推理中的置换、合一。

例如，对前面的公式，

$[\neg P(x, f(x))\vee Q(x,g(x))] \wedge [\neg P(x, f(x))\vee \neg R(x,g(x))]$

可把第二个子句中的变元名 x 更换为 y，得到如下子句集：

$\{ \neg P(x,f(x))\vee Q(x,g(x)), \neg P(y,f(y))\vee \neg R(y,g(y))\}$

四、 实验代码

```
'''
字句集消解实验
'''
'''
->:>
析取：%
合取：^
全称：@
存在：#
'''
# 1.消去>蕴涵项  a>b 变为~a%b
def del_inlclue(orign):

    ind = 0
    flag=0
    orignStack=[]
    right_bracket = 0
    while (ind < len(orign)):
        orignStack.append(orign[ind])
        if ((ind+1<len(orign)) and (orign[ind+1]=='>')):
            flag=1
            if orign[ind].isalpha():#是字母
                orignStack.pop()
                orignStack.append('~')
                orignStack.append(orign[ind])
                orignStack.append('%')
                ind=ind+1
            if orign[ind]==')':
                right_bracket=right_bracket+1
                tempStack = []
                while(right_bracket!=-1):
                    tempStack.append(orignStack[-1])
                    if orignStack[-1]=='(':
                        right_bracket=right_bracket-1
                    orignStack.pop()
                right_bracket = right_bracket + 1
                tempStack.pop()#去掉'('
                orignStack.append('(~')
                tempStack.reverse()
                for i in tempStack:
                    orignStack.append(i)
```

```
                    orignStack.append('%')
                    ind=ind+1
            ind=ind+1
    if flag==1:
            a=''
            for i in orignStack:
                a=a+i
            return a
    else:
            return orign


#2.处理否定连接词
def dec_neg_rand(orign):

    #处理~(@x)p(x)  变为(#x)~p(x)#######################################
    ind = 0
    flag = 0
    orignStack = []
    left_bracket = 0
    while (ind < len(orign)):
            orignStack.append(orign[ind])
            if orign[ind]=='~':
                if orign[ind+1]=='(':
                    if orign[ind+2]=='@' or orign[ind+2]=='#':
                        flag=1
                        ind=ind+1
                        orignStack.pop()#去掉前面的~
                        orignStack.append(orign[ind])
                        if orign[ind+1]=='@':
                            orignStack.append('#')
                        else:
                            orignStack.append('@')
                        orignStack.append(orign[ind+2])#'x'
                        orignStack.append(orign[ind+3])#')'
                        orignStack.append('~')
                        ind=ind+3
            ind=ind+1
    if flag==1:
            a=''
            for i in orignStack:
                a=a+i
            orign2=a
    else:
```

```
        orign2=orign
    #print('orign2',orign2)


    # 处理~(p%q) 变为(~p^~q)#####################################
    ind = 0
    flag = 0
    flag2 = 0   # 判断是否进入 while left_bracket>=1:循环 ，若进入，出来后 ind 再减 1
    orignStack = []
    left_bracket = 0
    while (ind < len(orign2)):
        orignStack.append(orign2[ind])
        if orign2[ind] == '~':
            if orign2[ind + 1] == '(':
                orignStack.pop()

                ind=ind+2#此时为 p
                left_bracket=left_bracket+1
                orignStack.append('(~')
                while left_bracket>=1:
                    flag2=1
                    orignStack.append(orign2[ind])
                    if orign2[ind]=='(':
                        left_bracket=left_bracket+1
                    if orign2[ind]==')':
                        left_bracket=left_bracket-1
                    if left_bracket==1 and orign2[ind+1]=='%' and orign2[ind+2]!='@' and orign2[ind+2]!='#':
                        flag=1
                        orignStack.append('^~')
                        ind=ind+1
                    if left_bracket == 1 and orign2[ind + 1] == '^' and orign2[ind + 2] != '@' and orign2[ind + 2] != '#':
                        flag = 1
                        orignStack.append('%~')
                        ind = ind + 1
                    ind=ind+1
        if flag2==1:
            ind=ind-1
            flag2=0
        ind=ind+1
    if flag==1:
        a=''
```

```
        for i in orignStack:
            a=a+i
        orign3=a
else:
        orign3=orign2
#print('orign3',orign3)


#  处理~~p  变为 p#####################################
ind = 0
flag = 0
bflag = 0
orignStack = []
while (ind < len(orign3)):
        orignStack.append(orign3[ind])
        if orign3[ind] == '~':
            if orign3[ind + 1] == '~':
                flag = 1
                orignStack.pop()
                ind = ind + 1
        ind = ind + 1

if flag == 1:
        a = ''
        for i in orignStack:
            a = a + i
        orign4 = a
else:
        orign4 = orign3
# print('orign4', orign4)
#  处理~(~p)  变为 p#####################################
ind = 0
flag = 0
bflag=0
orignStack = []
while (ind < len(orign4)):
        orignStack.append(orign4[ind])
        if orign4[ind] == '~':
            if orign4[ind + 1] == '(':
                left_bracket = 1
                if orign4[ind+2]=='~':
                    flag=1
                    orignStack.pop()
                    ind=ind+2
```

```
                    while left_bracket>=1:
                        orignStack.append(orign4[ind+1])
                        if orign4[ind+1]=='(':
                            left_bracket=left_bracket+1
                        if orign4[ind+1]==')':
                            left_bracket=left_bracket-1
                        if orign4[ind+1]=='%' or orign4[ind+1]=='^':
                            bflag=1
                        ind=ind+1

                    orignStack.pop()

        ind = ind + 1

    if flag==1 and bflag==0:
        a=''
        for i in orignStack:
            a=a+i
        orign5=a
    else:
        orign5=orign4
    #print('orign5',orign5)
    return orign5


#3.命题变量标准化，使后面 y=w
def standard_var(orign):#对变量标准化,简化,不考虑多层嵌套

    flag = 1
    desOri=[]
    des=['w','k','j']
    j=0
    orignStack = []
    left_bracket=0
    ind = 0
    while flag!=0:
        flag=0
        while (ind < len(orign)):
            orignStack.append(orign[ind])
            if orign[ind] == '@' or orign[ind]=='#':
                x=orign[ind+1]#保存 x
                if orign[ind+1] in desOri:
                    orignStack.append(des[j])
                    desOri.append(des[j])
```

```
            j=j+1
            orignStack.append(')')
            ind=ind+3
            if ind<len(orign):
                if orign[ind].isalpha():#(@x)p(x,y)这种情况
                    orignStack.append(orign[ind])#p
                    ind = ind + 1
                    if orign[ind]=='(':
                        left_bracket = left_bracket + 1
                        orignStack.append(orign[ind])
                        ind=ind+1
                        while left_bracket>0:
                            if orign[ind]== ')':
                                left_bracket = left_bracket - 1
                            if orign[ind]== '(':
                                left_bracket=left_bracket+1
                            if orign[ind]== x:
                                flag=1
                                orignStack.append(des[j-1])
                            else:
                                orignStack.append(orign[ind])
                            ind=ind+1
                        ind=ind-1

            if ind<len(orign):
                if orign[ind] == '(' :
                    left_bracket = left_bracket + 1
                    orignStack.append(orign[ind])
                    ind = ind + 1
                    while left_bracket > 0:
                        if orign[ind] == ')':
                            left_bracket = left_bracket - 1
                        if orign[ind] == '(':
                            left_bracket = left_bracket + 1
                        if orign[ind] == x:
                            flag = 1
                            orignStack.append(des[j - 1])
                        else:
                            orignStack.append(orign[ind])
                        ind = ind + 1
                    ind=ind-1

        else:
            desOri.append(orign[ind+1])
```

```
                ind=ind+1

        a=''
        for i in orignStack:
            a=a+i
        orign2=a
        return orign2

#4.消去存在量词（skolem 化）
def del_exists(orign):
        ind = 0
        flag = 1
        orignStack = []
        x=''
        y=''
        # 第 1 种情况：前面有全称量词 (@x)((#y)p(x,y))变为（@x）p(x,f(x))
        while flag!=0: #为了嵌套的情况出现
            flag=0
            while (ind < len(orign)):
                orignStack.append(orign[ind])

                if orign[ind] == '(' and orign[ind+1] == '@' and orign[ind+4]=='(' :
                    x=orign[ind+2]
                    orignStack.append(orign[ind+1:ind+5])
                    ind=ind+5#指向
                    while orign[ind]!='#':
                        orignStack.append(orign[ind])
                        ind=ind+1
                    orignStack.pop()
                    y=orign[ind+1]#为 y
                    ind=ind+2#指向 p
                    flag=1

                ind = ind + 1

            if flag==1:
                orignStack2=[]
                for i in orignStack:
                    if i==y:
                        orignStack2.append('g(')
                        orignStack2.append(x)
                        orignStack2.append(')')
                    else:
                        orignStack2.append(i)
```

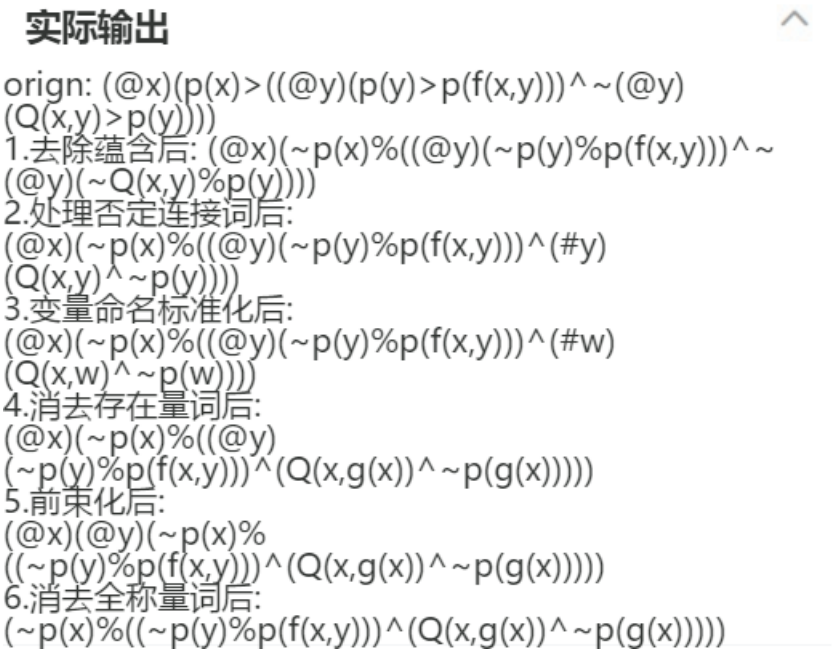
```
    a = ''
    for i in orignStack2:
        a = a + i
    orign2 = a
    ind = 0
    flag = 1
    orignStack = []
    # 第 2 种情况：前面没有全称量词 (#y)p(x,y)变为 p(x,A)
    while flag != 0:  # 为了嵌套的情况出现
        flag = 0
        while (ind < len(orign2)):
            orignStack.append(orign2[ind])
            if orign2[ind] == '#' :
                y=orign2[ind+1]
                orignStack.pop()
                orignStack.pop()
                ind=ind+2#指向')'
                flag=1
            ind = ind + 1
        if flag==1:
            orignStack2 = []
            for i in orignStack:
                if i == y:
                    orignStack2.append('A')
                else:
                    orignStack2.append(i)

    a = ''
    for i in orignStack2:
        a = a + i
    orign2 = a
    return orign2


#5.前束化
def convert_to_front(orign):#化为前束形
    ind = 0
    orignStack = []
    tempStack=[]#存放全称量词
    while (ind < len(orign)):
        orignStack.append(orign[ind])
        if orign[ind]=='(' and orign[ind+1]=='@':
            orignStack.pop()
            tempStack.append(orign[ind:ind+4])
```

```
                ind=ind+3

            ind = ind + 1

        orignStack=tempStack+orignStack
        a=''
        for i in orignStack:
            a = a + i
        orign2 = a
        return orign2

#6.消去全称量词
def del_all(orign):
        ind = 0
        orignStack = []
        ###########开始 1#############
        while (ind < len(orign)):
            if orign[ind] == '(' and orign[ind + 1] == '@':
                ind = ind + 4
            else:
                orignStack.append(orign[ind])
                ind = ind + 1
        a = ''
        for i in orignStack:
            a = a + i
        return a
        ###########结束 1#############
import sys
codeIn = sys.stdin.read()
orign=codeIn
#orign = '(@x)(p(x)>((@y)(p(y)>p(f(x,y)))^~(@y)(Q(x,y)>p(y))))'
print('orign:',orign)
a=del_inlclue(orign)
print('1.去除蕴含后:',a)
a=dec_neg_rand(a)
print('2.处理否定连接词后:')
print(a)
a=standard_var(a)
print('3.变量命名标准化后:')
print(a)
a=del_exists(a)
print('4.消去存在量词后:')
print(a)
a=convert_to_front(a)
```

```
print('5.前束化后:')
print(a)
a=del_all(a)
print('6.消去全称量词后:')
print(a)
```

五、 实验结果

平台上的运行结果如下：



Python 上的运行结果如下：

```
E:\Anaconda32020.02\envs\pytorch\python.exe "D:/大学作业/3 上/人工智能原理/AI/AI.py"
(@x)(p(x)> ((@y)(p(y)> p(f(x,y)))^~(@y)(Q(x,y)> p(y))))^D
Traceback (most recent call last):
orign: 测试输入:(@x)(p(x)> (@y)(p(y)> p(f(x,))^~(@y)
(Q(x,y)> p()).
  File "D:/大学作业/3 上/人工智能原理/AI/AI.py", line 394, in <module>
orign: (@x(p(x)> ((@y)(p(y)> p(x,))^~(@y)
    a = standard_var(a)
(Q(x,yj> p(y))

  File "D:/大学作业/3 上/人工智能原理/AI/AI.py", line 215, in standard_var

    orignStack.append(des[j])
1.去除蕴含后: 测试输入:(@x)(~p(x)% (@y)(~p(y)% p(f(x,))^~(@y)
IndexError: list index out of range
(~Q(x,y)% p()).
orign: (@x(~p(x)% ((@y)(~p(y)% p(x,))^~(@y)
(Q(x,y~j% p(y))


2.处理否定连接词后:
测试输入:(@x)(~p(x)% (@y)(~p(y)% p(f(x,))^(#y)~
(~Q(x,y)% p()).
orign: (@x(~p(x)% ((@y)(~p(y)% p(x,))^(#y)~
(Q(x,y~j% p(y))



进程已结束,退出代码1
```

六、 心得体会

通过本次实验，我进一步熟悉了子句集化简的九步法具体的解决问题的过程，熟悉和掌握化为子句集的九步法的原理、实质，熟悉了谓词公式化为子句集的九个步骤，学会了使用化为子句集的九步法解决问题。

上课我们知道，任何一个谓词公式都可以化为合取范式，合取的每一个部分就是一个子句，所有子句组成这个谓词公式的子句集合，我们将这个子句集合叫做谓词公式的子句集表示。

实验过程中巩固了所学的知识，进一步理解了消解规则。通过实验不仅提高了自己的编程和思维能力，更提高了我们用代码来解决生活中的实际问题的能力，使我们收获良多。

由于这个实验我之前在学习 C++时做过，所以是很熟悉的，找到以前的代码，处理的思路基本是一致的。

实验七 粒子群优化算法实验

一、 实验目的

(1)理解粒子群算法的基本思想

(2)能够实现粒子群算法并进行优化

二、 实验内容

粒子群算法(Particle Swarm Optimization,PSO)是 20 世纪 90 年代兴起的一门学科,因其概念简明、实现方便、收敛速度快而为人所知。粒子群算法的基本思想是模拟鸟群随机搜寻食物的捕食行为，乌群通过自身经验和种群之间的交流调整自己的搜寻路径,从而找到食物最多的地点。其中每只鸟的位置/路径则自变量组合,每次到达的地点的食物密度即函数值。每次搜寻都会根据自身经验(自身历史搜寻的最优地点)和种群交流(种群历史搜寻的最优地点)调整自身搜寻方向和速度，这个称为跟踪极值，从而找到最优解。粒子群算法中所涉及到的参数有:种群数量:粒子群算法的最大特点就是速度快,因此初始种群取 50-1000 都是可以的，虽然初始种群越大收敛性会更好,不过太大了也会影响速度;
要求用 Python 语言实现对粒子群算法的优化

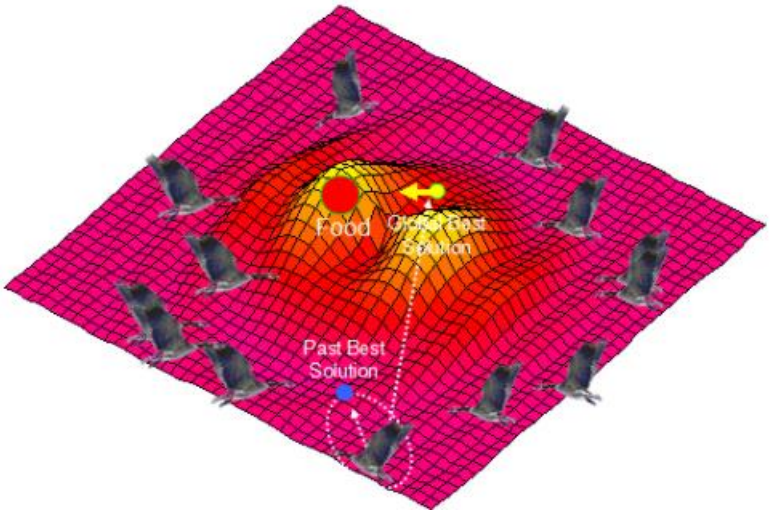
三、 实验原理

粒子群优化算法(PSO)是一种进化计算技术(evolutionary computation)，1995 年由 Eberhart 博士和 kennedy 博士提出，源于对鸟群捕食的行为研究 。PSO 同遗传算法类似，是一种基于迭代的优化算法。系统初始化为一组随机解，通过迭代搜寻最优值。但是它没有遗传算法用的交叉(crossover)以及变异(mutation)，而是粒子在解空间追随最优的粒子进行搜索。同遗传算法比较，PSO 的优势在于简单容易实现并且没有许多参数需要调整。目前已广泛应用于函数优化，神经网络训练，模糊系统控制以及其他遗传算法的应用领域。

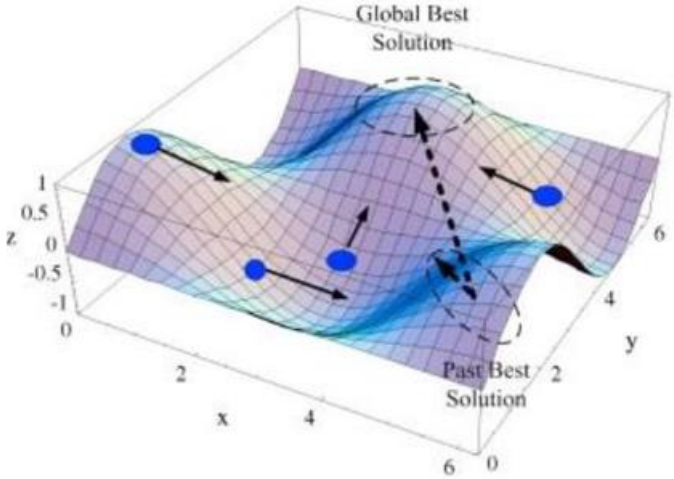
粒子群优化算法是一种基于种群寻优的启发式搜索算法。在 1995 年由 Kennedy 和 Eberhart 首先提出来的。

它的主要启发来源于对鸟群群体运动行为的研究。我们经常可以观察到鸟群表现出来的同步性，虽然每只的运动行为都是互相独立的，但是在整个鸟群的飞行过程中却表现出了高度一致性的复杂行为，并且可以自适应的调整飞行的状态和轨迹。

鸟群具有这样的复杂飞行行为的原因,可能是因为每只鸟在飞行过程中都遵循了一定的行为规则，并能够掌握邻域内其它鸟的飞行信息。



粒子群优化算法借鉴了这样的思想，每个粒子代表待求解问题搜索解空间中的一个潜在解，它相当于一只鸟，“飞行信息”包括粒子当前的位置和速度两个状态量。



每个粒子都可以获得其邻域内其它个体的信息，对所经过的位置进行评价，并根据这些信息和位置速度更新规则，改变自身的两个状态量，在“飞行"过程中传递信息和互相学习，去更好地适应环境。

随着这一过程的不断进行，粒子群最终能够找到问题的近似最优解。

四、 实验代码

```
# -*- coding: utf-8 -*-
"""
粒子群优化算法
粒子群算法求解函数最大值（最小值）
f(x)= x + 10*sin5x + 7*cos4x    根据具体情况  改函数
"""
import numpy as np


# 粒子（鸟）
class Particle:
    def __init__(self):
        self.p = 0    # 粒子当前位置
        self.v = 0    # 粒子当前速度
        self.pbest = 0    # 粒子历史最好位置


class PSO:
    def __init__(self, N=20, iter_N=200):
        self.w = 0.2    # 惯性因子
        self.c1 = 1    # 自我认知学习因子
        self.c2 = 2    # 社会认知学习因子
        self.gbest = 0    # 种群当前最好位置
        self.N = N    # 种群中粒子数量
        self.POP = []    # 种群
        self.iter_N = iter_N    # 迭代次数

    # 适应度值计算函数
    def fitness(self, x):
        return x + 10 * np.sin(5 * x) + 7 * np.cos(4 * x)

    # 找到全局最优解
    def g_best(self, pop):
        for bird in pop:
```

```
                if bird.fitness > self.fitness(self.gbest):
                    self.gbest = bird.p

    # 初始化种群
    def initPopulation(self, pop, N):
        for i in range(N):
            bird = Particle()
            bird.p = np.random.uniform(-10, 10)#从一个均匀分布[low,high)中随
机采样，注意定义域是左闭右开，即包含 low，不包含 high.
            bird.fitness = self.fitness(bird.p)
            bird.pbest = bird.fitness
            pop.append(bird)
        # 找到种群中的最优位置
        self.g_best(pop)

    # 更新速度和位置，和 p_best
    def update(self, pop):
        for bird in pop:
            v = self.w * bird.v + self.c1 * np.random.random() * (
                    bird.pbest - bird.p) + self.c2 * np.random.random() *
(self.gbest - bird.p)

            p = bird.p + v

            if -10 < p < 10:
                bird.p = p
                bird.v = v
                # 更新适应度
                bird.fitness = self.fitness(bird.p)

                # 是否需要更新本粒子历史最好位置
                if bird.fitness > self.fitness(bird.pbest):
                    bird.pbest = bird.p

    def implement(self):
        # 初始化种群
        self.initPopulation(self.POP, self.N)
        ############开始 1##############
        # 迭代
        # 更新速度和位置,p_best,更新种群中最好位置
        i=0
        while(i<self.iter_N):
            self.update(self.POP)
            self.g_best(self.POP)
```

```
                i+=1
            ############结束 1##############
pso = PSO(N=20, iter_N=200)
pso.implement()

'''
#可以查看求解过程
for ind in pso.POP:
    print("x = ", ind.p, "f(x) = ", ind.fitness)
print("最优解  x = ", pso.gbest, "相应最大值  f(x) = ", pso.fitness(pso.gbest))
'''

if (pso.gbest>=7.850) and (pso.gbest<=7.860) :
    if (pso.fitness(pso.gbest)>=24.850) and (pso.fitness(pso.gbest)<=24.860) :
        print('success')
```

五、 实验结果

平台上的运行结果：

实际输出

success

Python 上的运行结果：

```
E:\Anaconda32020.02\envs\pytorch\python.exe "D:/大学作业/3 上/人工智能原理/AI/AI.py"
success

进程已结束,退出代码0
```

六、 心得体会

我感觉速度这个向量其实就是跟梯度下降算法中的微分是一样的道理，就是控制鸟往接近全局最优解的地方去飞；然后 PSO 比梯度下降好的地方是，大家一起来确定怎么飞，梯度下降只是一只鸟往哪儿飞，所以梯度下降很容易飞到局部最优解，PSO 可以飞到大家都认为的全局最优解；初始化的速度向量，初始化的量应该不会影响 PSO 算法的最终结果，但是会影响计算的过程。

通过这次实验，我总结一下 PSO 算法的一些优点：

（1）它是一类不确定算法。不确定性体现了自然界生物的生物机制，并且在求解某些特定问题方面优于确定性算法。

（2）是一类概率型的全局优化算法。非确定算法的优点在于算法能有更多机会求解全局最优解。

（3）不依赖于优化问题本身的严格数学性质。

（4）是一种基于多个智能体的仿生优化算法。粒子群算法中的各个智能体之间通过相互协作来更好的适应环境，表现出与环境交互的能力.

（5）具有本质并行性。包括内在并行性和内含并行性。

（6）具有突出性。粒子群算法总目标的完成是在多个智能体个体行为的运动过程中突现出来的。

（7）具有自组织和进化性以及记忆功能，所有粒子都保存优解的相关知识。

（8）都具有稳健性。稳健性是指在不同条件和环境下算法的实用性和有效性，但是现在粒子群算法的数学理论基础还不够牢固，算法的收敛性还需要讨论。

由于这个实验在 Python 和小程序上没有可视化的过程，所以我又在 MATLAB 上进行了测试，以此来得到一个动态可视的运行全过程，更加有助于我们的理解：

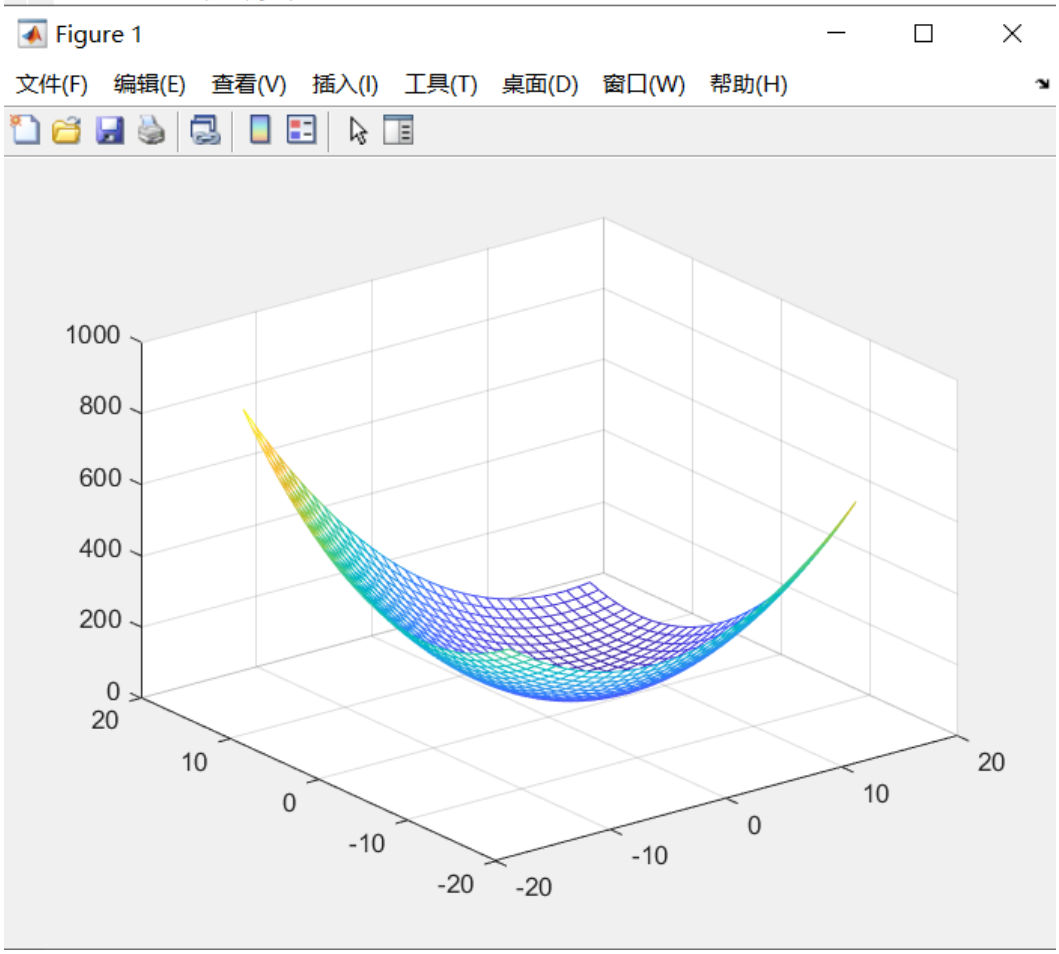
这里我使用的是 MATLAB R2020a

首先我构建目标函数。这里使用只有两个参数的函数，这样便于画出三维图型。

```
1    function y=All_01(x)
2
3 —      y=x(1)^2+x(2)^2-x(1)*x(2)-10*x(1)-4*x(2)+60;
```

然后我们从直观上看看这个函数：

```
x1=-15:1:15;
x2=-15:1:15;
[x1,x2]=meshgrid(x1,x2);
y=x1.^2+x2.^2-x1.*x2-10.*x1-4.*x2+60;
mesh(x1,x2,y);
```

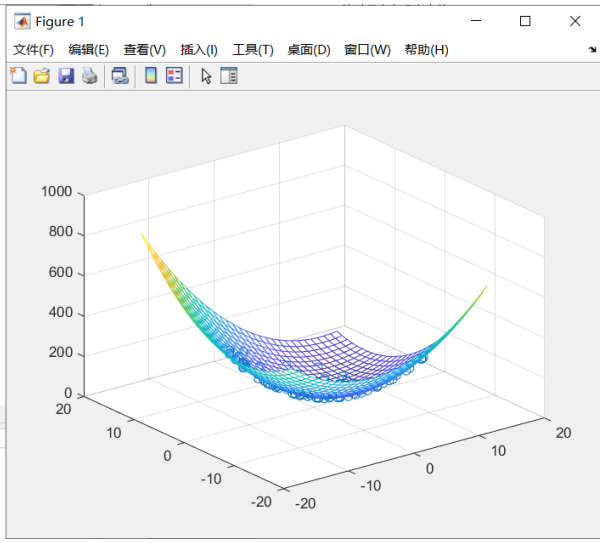


然后我们先看看预设分布粒子和速度的取值范围后的粒子群分布如何

```
w=0.9;%权重一般设为0.9
K=50; %迭代次数

%%分布粒子的取值范围及速度的取值范围
x=-10+20*rand(n,d);  %x在[-10,10]中取值
v=-5+10*rand(n,d); %v在[-5,5]中取值

%%计算适应度
fit=zeros(n,1);
for j=1:n
    fit(j)=A11_01(x(j,:));
end
pbest=x;
ind=find(min(fit)==fit);
gbest=x(ind,:);
h=scatter3(x(:,1),x(:,2),fit,'o');  %图2 粒子的初始分布图

% %%更新速度与位置
```

命令行窗口

fx >>



最后，我加上更新速度与位置并且重新计算适应度的代码
这里我们为了更直观的看到粒子群的运动情况，设置0.1s更新一次，按时间顺序截图了其中三张，如下：(运行全过程的视频我附在了文件里面)