山西大学计算机与信息技术学院

**实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 姓 名 | 李京 | 学 号 | 202202501110 | 专业班级 | 计算机科学与技术2202 | |
| 课程名称 | 人工智能 | | | | 实验日期 | 2024.10.14 |
| 成 绩 |  | 指导教师 | 李琳 | | 批改日期 |  |
| 实验名称 | | 实验三 智能搜索技术 | | | | |
| 一、实验目的：  通过本实验，掌握不确定性推理技术、理解消解原理的应用，并设计实现一个基于产生式系统的植物识别系统。  二、实验内容：   1. **不确定性推理实验**：学习并实现不确定性推理的数学方法，特别是基于概率的似然推理，使用贝叶斯网络和贝叶斯公式来计算给定事件和条件概率下的目标概率，并编写程序实现这一推理过程。 2. **消解原理实验**：理解消解规则在谓词演算公式中的应用，通过消解过程从子句集中推导出新的子句，掌握子句消解的原理和规则，并编写程序实现这一推理过程。 3. **产生式系统实验**：设计并实现一个植物识别系统的产生式系统，包括数据库的创建和规则库的设计，通过正向和逆向推理来识别植物，并编写程序实现这一过程，最终输出植物识别结果。   三、实验平台：  MacBook Air M3  Microsoft Visual Studio Code  Mamba environment + python 3.12.6  四、实验步骤：  贝叶斯  1-Bayes.py 脚本实现了一个贝叶斯网络的概率推理。以下是主要步骤的简要说明：  1. 导入模块  - sys: 处理命令行参数。  - itertools: 生成隐藏变量的所有可能组合。  - tabulate: 格式化并输出结果表格。  2. read\_input(file\_path) 函数  目的  读取并解析输入文件，提取变量、邻接矩阵、父节点关系、条件概率表（CPTs）和查询。  步骤  1. 读取文件内容: 打开并读取指定路径的输入文件，过滤掉空行。  2. 解析变量数量和名称:  - 第一行指定变量数量 N。  - 第二行列出变量的名称。  3. 构建邻接矩阵:  - 接下来的 N 行表示邻接矩阵，定义变量之间的依赖关系。  4. 确定每个变量的父节点:  - 根据邻接矩阵，构建一个列表，记录每个变量的父节点。  5. 解析条件概率表（CPTs）:  - 对于每个变量，根据其父节点的数量，读取相应数量的概率值。  6. 提取查询:  - 剩余的行作为要处理的查询。  3. compute\_probability(variables, parents, CPTs, query\_var, evidence) 函数  目的  计算在给定证据下，查询变量取值为真或假的概率。  步骤  1. 变量索引映射: 创建变量名称到索引的映射。  2. 识别证据变量和隐藏变量:  - 排除查询变量和已知证据变量，识别出需要枚举的隐藏变量。  3. 初始化概率存储: 为查询变量的每个可能取值（真/假）初始化概率。  4. 枚举所有隐藏变量的可能赋值:  - 使用 itertools.product 生成隐藏变量的所有可能组合。  5. 计算联合概率:  - 对于每种赋值，计算联合概率，基于CPTs和当前赋值的变量取值。  6. 累加概率: 将每种赋值的概率累加到对应的查询变量取值的总概率中。  7. 归一化: 将累加后的概率归一化，以确保其总和为1。  4. parse\_query(query\_line) 函数  目的  解析查询字符串，提取查询变量和证据条件。  步骤  1. 格式验证: 确保查询字符串以 P( 开始，并包含 | 分隔符。  2. 提取查询变量和证据:  - lhs 为查询变量。  - rhs 为证据条件，解析成变量-值对。  3. 转换证据值: 将证据中的字符串值（如 "true" 或 "false"）转换为布尔值。  5. main() 函数  目的  脚本的主入口，负责整体流程的协调和执行。  步骤  1. 命令行参数验证: 确保用户提供了正确的输入文件路径。  2. 读取输入文件: 调用 read\_input 函数解析输入文件，获取变量、结构、CPTs 和查询。  3. 处理每个查询:  - 遍历所有查询行，解析查询并计算概率。  - 将结果格式化为表格行。  4. 输出结果: 使用 tabulate 模块将所有查询结果以表格形式打印出来。  6. 执行入口保护  目的  确保脚本作为主程序运行时调用 main() 函数。  代码  if \_\_name\_\_ == "\_\_main\_\_":  main()  解析贝叶斯网络的结构和条件概率表，处理用户的概率查询，并以易于阅读的格式输出结果。  import sys  import itertools  from tabulate import tabulate  def read\_input(*file\_path*):  with open(*file\_path*, 'r') as f:  lines = [line.strip() for line in f if line.strip() != '']  N = int(lines[0])  variables = lines[1].split()  adj\_matrix = []  idx = 2  for \_ in range(N):  adj\_matrix.append(list(map(int, lines[idx].split())))  idx += 1  parents = [[] for \_ in range(N)]  for j in range(N):  for i in range(N):  if adj\_matrix[i][j] == 1:  parents[j].append(i)  CPTs = []  for var\_idx in range(N):  num\_parents = len(parents[var\_idx])  num\_lines = 2 \*\* num\_parents  cpt = []  for \_ in range(num\_lines):  if idx < len(lines):  probs = list(map(float, lines[idx].split()))  if len(probs) >= 1:  cpt.append(probs[0]) # Store P(variable=true | parents)  idx +=1  else:  idx +=1  CPTs.append(cpt)  queries = lines[idx:]  return N, variables, parents, CPTs, queries  def compute\_probability(*variables*, *parents*, *CPTs*, *query\_var*, *evidence*):  var\_indices = {var: idx for idx, var in enumerate(*variables*)}  Q\_idx = var\_indices[*query\_var*]  evidence\_vars = {var\_indices[var]: val for var, val in *evidence*.items()}  hidden\_vars = [i for i in range(len(*variables*)) if i != Q\_idx and i not in evidence\_vars]  # Compute numerator for Q=true and Q=false  probs = {}  for q\_val in [True, False]:  total\_prob = 0.0  for values in itertools.product([True, False], *repeat*=len(hidden\_vars)):  assignment = {}  for var\_idx, val in evidence\_vars.items():  assignment[var\_idx] = val  assignment[Q\_idx] = q\_val  for idx, val in zip(hidden\_vars, values):  assignment[idx] = val  prob = 1.0  for var\_idx in range(len(*variables*)):  var\_parents = *parents*[var\_idx]  parent\_vals = tuple(assignment[p\_idx] for p\_idx in var\_parents)  num\_parents = len(var\_parents)  if num\_parents == 0:  index = 0  else:  index = sum((parent\_vals[i] << (num\_parents - i -1)) for i in range(num\_parents))  p\_true = *CPTs*[var\_idx][index]  var\_val = assignment[var\_idx]  if var\_val:  prob \*= p\_true  else:  prob \*= (1 - p\_true)  total\_prob += prob  probs[q\_val] = total\_prob  total = probs[True] + probs[False]  probs[True] /= total  probs[False] /= total  return probs[True], probs[False]  def parse\_query(*query\_line*):  # Example: P(Burglar | Alarm=true, Earthquake=true)  *query\_line* = *query\_line*.strip()  if *query\_line*.startswith('P(') and '|' in *query\_line*:  content = *query\_line*[2:-1]  lhs, rhs = content.split('|')  query\_var = lhs.strip()  evidence = {}  for item in rhs.strip().split(','):  if '=' in item:  var, val = item.strip().split('=')  evidence[var.strip()] = True if val.strip().lower() == 'true' else False  return query\_var, evidence  else:  return None, None  def main():  if len(sys.argv) != 2:  print("Usage: python bayesian\_network.py <input\_file>")  sys.exit(1)  input\_file = sys.argv[1]  N, variables, parents, CPTs, queries = read\_input(input\_file)  var\_indices = {var: idx for idx, var in enumerate(variables)}  results = []  for query\_line in queries:  if not query\_line.strip():  continue  query\_var, evidence = parse\_query(query\_line)  if query\_var is None:  continue  p\_true, p\_false = compute\_probability(variables, parents, CPTs, query\_var, evidence)  evidence\_str = ', '.join([f'{var}={str(val)}' for var, val in evidence.items()])  results.append([f"P({query\_var} | {evidence\_str})", f"{p\_true:.3f}", f"{p\_false:.3f}"])  headers = ["Query", "P(True)", "P(False)"]  print(tabulate(results, *headers*=headers, *tablefmt*="grid"))  if \_\_name\_\_ == "\_\_main\_\_":  main()  消解定律  1. 导入模块  import sys  import re  from collections import defaultdict  from copy import deepcopy  - sys: 用于处理命令行参数和系统操作。  - re: 正则表达式模块，用于字符串解析和匹配。  - defaultdict: 来自 collections 模块，用于创建带有默认值的字典。  - deepcopy: 用于深拷贝对象，防止修改原始数据。  2. 类定义  2.1 Literal 类  class Literal:  def \_\_init\_\_(self, name, args=[], is\_neg=False):  self.name = name  self.args = args  self.is\_neg = is\_neg  def \_\_repr\_\_(self):  neg = "¬" if self.is\_neg else ""  if self.args:  args\_str = ", ".join(self.args)  return f"{neg}{self.name}({args\_str})"  else:  return f"{neg}{self.name}"  def negate(self):  return Literal(self.name, self.args, not self.is\_neg)  def \_\_eq\_\_(self, other):  return (  self.name == other.name and  self.is\_neg == other.is\_neg and  self.args == other.args  )  def \_\_hash\_\_(self):  return hash((self.name, tuple(self.args), self.is\_neg))  - 属性:  - name: 谓词名称。  - args: 谓词的参数列表。  - is\_neg: 是否为否定文字。  - 方法:  - \_\_repr\_\_: 返回文字的字符串表示，考虑否定和参数。  - negate: 返回当前文字的否定。  - \_\_eq\_\_ 和 \_\_hash\_\_: 使 Literal 对象可以在集合中使用，并支持比较。  2.2 Clause 类  class Clause:  def \_\_init\_\_(self, literals=[]):  self.literals = literals  def \_\_repr\_\_(self):  return "∨".join([str(lit) for lit in self.literals])  - 属性:  - literals: 文字的列表，构成一个子句。  - 方法:  - \_\_repr\_\_: 返回子句的字符串表示，文字之间用逻辑或符号连接。  3. 解析函数  3.1 parse\_literal  def parse\_literal(literal\_str):  literal\_str = literal\_str.strip()  is\_neg = False  if literal\_str.startswith("¬"):  is\_neg = True  literal\_str = literal\_str[1:].strip()  match = re.match(r'(\w+)(?:\((.\*)\))?', literal\_str)  if not match:  raise ValueError(f"Invalid literal format: {literal\_str}")  name = match.group(1)  args\_str = match.group(2)  args = []  if args\_str:  args = split\_args(args\_str)  return Literal(name, args, is\_neg)  - 功能: 将字符串形式的文字解析为 Literal 对象。  - 步骤:  1. 去除字符串首尾的空白。  2. 检查是否有否定符号 ¬，并相应设置 is\_neg。  3. 使用正则表达式匹配谓词名称和参数。  4. 分析并拆分参数，生成 Literal 对象。  3.2 split\_args  def split\_args(args\_str):  args = []  current = ""  depth = 0  for char in args\_str:  if char == ',' and depth == 0:  args.append(current.strip())  current = ""  else:  if char == '(':  depth += 1  elif char == ')':  depth -= 1  current += char  if current:  args.append(current.strip())  return args  - 功能: 将参数字符串按照逗号分割，考虑嵌套括号的情况。  - 逻辑:  - 使用 depth 变量跟踪括号嵌套层数。  - 仅在 depth 为 0 时才按逗号分割，确保正确处理嵌套函数。  3.3 parse\_clause  def parse\_clause(clause\_str):  literals\_str = clause\_str.split("∨")  literals = [parse\_literal(lit) for lit in literals\_str]  return Clause(literals)  - 功能: 将字符串形式的子句解析为 Clause 对象。  - 逻辑:  - 按照逻辑或符号 ∨ 分割多个文字。  - 使用 parse\_literal 解析每个文字，生成 Clause 对象。  4. 标准化  Standardizer 类  class Standardizer:  def \_\_init\_\_(self):  self.counter = defaultdict(int)  def standardize(self, clause):  substitution = {}  new\_literals = []  for lit in clause.literals:  new\_args = []  for arg in lit.args:  if is\_variable(arg):  if arg not in substitution:  substitution[arg] = f"{arg}"  self.counter[arg] += 1  new\_args.append(substitution[arg])  elif is\_function(arg):  new\_args.append(self.standardize\_term(arg, substitution))  else:  new\_args.append(arg)  new\_literals.append(Literal(lit.name, new\_args, lit.is\_neg))  return Clause(new\_literals)  def standardize\_term(self, term, substitution):  if not is\_function(term):  if is\_variable(term):  if term not in substitution:  substitution[term] = f"{term}"  self.counter[term] += 1  return substitution[term]  else:  return term  else:  name = get\_function\_name(term)  args = get\_arguments(term)  new\_args = []  for arg in args:  if is\_variable(arg):  if arg not in substitution:  substitution[arg] = f"{arg}"  self.counter[arg] += 1  new\_args.append(substitution[arg])  elif is\_function(arg):  new\_args.append(self.standardize\_term(arg, substitution))  else:  new\_args.append(arg)  return f"{name}({', '.join(new\_args)})"  - 功能: 对子句中的变量进行标准化，避免变量名冲突。  - 逻辑:  - 使用 substitution 字典记录变量替换。  - 对每个文字和其参数进行遍历，替换变量名。  - 处理嵌套函数中的变量，保证所有变量名唯一。  5. 辅助函数  判断变量和函数  def is\_variable(term):  return term[0].islower()  def is\_function(term):  return '(' in term and term.endswith(')')  def get\_function\_name(term):  return term[:term.find('(')]  def get\_arguments(term):  args\_str = term[term.find('(')+1:-1]  return split\_args(args\_str)  - is\_variable: 判断术语是否为变量（以小写字母开头）。  - is\_function: 判断术语是否为函数（包含括号且以 ) 结尾）。  - get\_function\_name: 获取函数名。  - get\_arguments: 提取函数的参数列表。  6. 统一算法  unify  def unify(x, y, substitution):  if substitution is None:  return None  elif x == y:  return substitution  elif is\_variable(x):  return unify\_var(x, y, substitution)  elif is\_variable(y):  return unify\_var(y, x, substitution)  elif is\_function(x) and is\_function(y):  if get\_function\_name(x) != get\_function\_name(y):  return None  args\_x = get\_arguments(x)  args\_y = get\_arguments(y)  if len(args\_x) != len(args\_y):  return None  for arg1, arg2 in zip(args\_x, args\_y):  substitution = unify(arg1, arg2, substitution)  if substitution is None:  return None  return substitution  else:  return None  - 功能: 实现\*\*统一算法\*\*，试图使两个术语在给定的替换下相等。  - 逻辑:  1. 如果替换为空，返回 None。  2. 如果两个术语相等，返回当前替换。  3. 如果其中一个是变量，调用 unify\_var。  4. 如果都是函数，检查函数名和参数数量是否一致，递归统一其参数。  5. 否则，返回 None，表示无法统一。  unify\_var  def unify\_var(var, x, substitution):  if var in substitution:  return unify(substitution[var], x, substitution)  elif is\_variable(x) and x in substitution:  return unify(var, substitution[x], substitution)  elif occurs\_check(var, x, substitution):  return None  else:  substitution = deepcopy(substitution)  substitution[var] = x  return substitution  - 功能: 处理变量的统一。  - 逻辑:  1. 如果变量已经有替换，递归统一。  2. 如果另一个术语是变量并有替换，递归统一。  3. 执行\*\*发生检查\*\*（occurs\_check），防止自我引用。  4. 否则，将变量替换为另一个术语。  occurs\_check  def occurs\_check(var, x, substitution):  if var == x:  return True  elif is\_function(x):  for arg in get\_arguments(x):  if occurs\_check(var, arg, substitution):  return True  elif x in substitution:  return occurs\_check(var, substitution[x], substitution)  return False  - 功能: 检查变量是否出现在术语中，防止无限递归替换。  - 逻辑:  - 如果变量等于术语，返回 True。  - 如果术语是函数，递归检查所有参数。  - 如果术语是变量且有替换，继续递归检查。  - 否则，返回 False。  substitute\_literal 和 substitute\_term  def substitute\_literal(literal, substitution):  new\_args = []  for arg in literal.args:  new\_arg = substitute\_term(arg, substitution)  new\_args.append(new\_arg)  return Literal(literal.name, new\_args, literal.is\_neg)  def substitute\_term(term, substitution):  while is\_variable(term) and term in substitution:  term = substitution[term]  if is\_function(term):  name = get\_function\_name(term)  args = get\_arguments(term)  new\_args = [substitute\_term(arg, substitution) for arg in args]  return f"{name}({', '.join(new\_args)})"  else:  return term  - 功能: 应用替换到文字和术语上。  - 逻辑:  - 对于文字，替换其所有参数。  - 对于术语，递归应用替换，处理函数嵌套。  substitute\_clause  def substitute\_clause(clause, substitution):  new\_literals = [substitute\_literal(lit, substitution) for lit in clause.literals]  return Clause(new\_literals)  - 功能: 将替换应用到整个子句中的所有文字。  7. 解析和处理子句  resolve\_clauses  def resolve\_clauses(clause\_str1, clause\_str2):  clause1 = parse\_clause(clause\_str1)  clause2 = parse\_clause(clause\_str2)  standardizer = Standardizer()  clause1 = standardizer.standardize(clause1)  clause2 = standardizer.standardize(clause2)  for lit1 in clause1.literals:  for lit2 in clause2.literals:  if lit1.name == lit2.name and lit1.is\_neg != lit2.is\_neg:  substitution = {}  substitution = unify\_literals(lit1, lit2, substitution)  if substitution is not None:  new\_clause1 = substitute\_clause(clause1, substitution)  new\_clause2 = substitute\_clause(clause2, substitution)  resolved\_lit1 = substitute\_literal(lit1, substitution)  resolved\_lit2 = substitute\_literal(lit2, substitution)  new\_literals1 = [lit for lit in new\_clause1.literals if lit != resolved\_lit1]  new\_literals2 = [lit for lit in new\_clause2.literals if lit != resolved\_lit2]  combined\_literals = new\_literals1 + new\_literals2  unique\_literals = list(set(combined\_literals))  return Clause(unique\_literals)  return None  - 功能: 对两个子句进行解析，尝试找到可以消解的文字，并生成新的子句（解析子句）。  - 逻辑:  1. 解析输入的子句字符串，生成 Clause 对象。  2. 使用 Standardizer 对变量进行标准化，避免变量名冲突。  3. 遍历两个子句中的文字，寻找名称相同且一个为否定另一个为正的文字。  4. 对符合条件的文字进行统一，生成替换。  5. 应用替换到两个子句，移除被解析的文字。  6. 合并剩余的文字，去除重复，生成新的子句。  7. 如果找到可解析的文字，返回新的子句；否则，返回 None。  unify\_literals  def unify\_literals(lit1, lit2, substitution):  if lit1.name != lit2.name or lit1.is\_neg == lit2.is\_neg:  return None  if len(lit1.args) != len(lit2.args):  return None  for arg1, arg2 in zip(lit1.args, lit2.args):  substitution = unify(arg1, arg2, substitution)  if substitution is None:  return None  return substitution  - 功能: 统一两个文字，确保名称相同且一个为否定另一个为正，且参数能够统一。  - 逻辑:  - 检查名称和否定状态。  - 检查参数数量。  - 对每对参数进行统一，累积替换。  8. 格式化输出  format\_clause  def format\_clause(clause):  return " ∨ ".join([str(lit) for lit in sorted(clause.literals, key=lambda x: str(x))])  - 功能: 将 Clause 对象格式化为可读的字符串形式，文字按字典序排序并用逻辑或连接。  9. 主程序入口  if \_\_name\_\_ == "\_\_main\_\_":  if len(sys.argv) != 2:  print(f"Usage: python {sys.argv[0]} <input\_file>")  sys.exit(1)  input\_file = sys.argv[1]  try:  with open(input\_file, 'r') as file:  input1 = file.readline().strip()  input2 = file.readline().strip()  except FileNotFoundError:  print(f"Error: The file '{input\_file}' does not exist.")  sys.exit(1)  resolvent = resolve\_clauses(input1, input2)  if resolvent:  print(format\_clause(resolvent))  else:  print("Error: resolution failed")  - 功能: 作为脚本的入口，处理命令行输入，读取子句，执行解析操作，并输出结果。  - 逻辑:  1. 检查命令行参数，确保提供了输入文件路径。  2. 打开并读取输入文件的前两行，分别作为两个子句。  3. 调用 resolve\_clauses 对两个子句进行解析，生成解析子句。  4. 如果解析成功，格式化并打印解析子句；否则，输出错误信息。  1. 解析输入: 将字符串形式的子句转换为内部数据结构（Literal 和 Clause）。  2. 标准化: 确保不同子句中的变量名不冲突。  3. 统一与解析: 查找可以消解的文字，通过统一算法生成替换，并生成新的子句。  4. 输出结果: 将解析结果以可读的形式输出，或提示解析失败。  该脚本适用于基本的逻辑推理任务，可以作为更复杂推理系统的基础模块。  import sys  import re  from collections import defaultdict  from copy import deepcopy  class Literal:  def \_\_init\_\_(*self*, *name*, *args*=[], *is\_neg*=False):  *self*.name = *name*  *self*.args = *args*  *self*.is\_neg = *is\_neg*  def \_\_repr\_\_(*self*):  neg = "¬" if *self*.is\_neg else ""  if *self*.args:  args\_str = ", ".join(*self*.args)  return f"{neg}{*self*.name}({args\_str})"  else:  return f"{neg}{*self*.name}"  def negate(*self*):  return Literal(*self*.name, *self*.args, not *self*.is\_neg)  def \_\_eq\_\_(*self*, *other*):  return (  *self*.name == *other*.name and  *self*.is\_neg == *other*.is\_neg and  *self*.args == *other*.args  )  def \_\_hash\_\_(*self*):  return hash((*self*.name, tuple(*self*.args), *self*.is\_neg))  class Clause:  def \_\_init\_\_(*self*, *literals*=[]):  *self*.literals = *literals*  def \_\_repr\_\_(*self*):  return "∨".join([str(lit) for lit in *self*.literals])  def parse\_literal(*literal\_str*):  *literal\_str* = *literal\_str*.strip()  is\_neg = False  if *literal\_str*.startswith("¬"):  is\_neg = True  *literal\_str* = *literal\_str*[1:].strip()  match = re.match(r'*(\w*+*)(?:*\(*(.*\**)*\)*)*?', *literal\_str*)  if not match:  raise ValueError(f"Invalid literal format: {*literal\_str*}")  name = match.group(1)  args\_str = match.group(2)  args = []  if args\_str:  args = split\_args(args\_str)  return Literal(name, args, is\_neg)  def split\_args(*args\_str*):  args = []  current = ""  depth = 0  for char in *args\_str*:  if char == ',' and depth == 0:  args.append(current.strip())  current = ""  else:  if char == '(':  depth += 1  elif char == ')':  depth -= 1  current += char  if current:  args.append(current.strip())  return args  def parse\_clause(*clause\_str*):  literals\_str = *clause\_str*.split("∨")  literals = [parse\_literal(lit) for lit in literals\_str]  return Clause(literals)  class Standardizer:  def \_\_init\_\_(*self*):  *self*.counter = defaultdict(int)  def standardize(*self*, *clause*):  substitution = {}  new\_literals = []  for lit in *clause*.literals:  new\_args = []  for arg in lit.args:  if is\_variable(arg):  if arg not in substitution:  substitution[arg] = f"{arg}"  *self*.counter[arg] += 1  new\_args.append(substitution[arg])  elif is\_function(arg):  new\_args.append(*self*.standardize\_term(arg, substitution))  else:  new\_args.append(arg)  new\_literals.append(Literal(lit.name, new\_args, lit.is\_neg))  return Clause(new\_literals)  def standardize\_term(*self*, *term*, *substitution*):  if not is\_function(*term*):  if is\_variable(*term*):  if *term* not in *substitution*:  *substitution*[*term*] = f"{*term*}"  *self*.counter[*term*] += 1  return *substitution*[*term*]  else:  return *term*  else:  name = get\_function\_name(*term*)  args = get\_arguments(*term*)  new\_args = []  for arg in args:  if is\_variable(arg):  if arg not in *substitution*:  *substitution*[arg] = f"{arg}"  *self*.counter[arg] += 1  new\_args.append(*substitution*[arg])  elif is\_function(arg):  new\_args.append(*self*.standardize\_term(arg, *substitution*))  else:  new\_args.append(arg)  return f"{name}({', '.join(new\_args)})"  def is\_variable(*term*):  return *term*[0].islower()  def is\_function(*term*):  return '(' in *term* and *term*.endswith(')')  def get\_function\_name(*term*):  return *term*[:*term*.find('(')]  def get\_arguments(*term*):  args\_str = *term*[*term*.find('(')+1:-1]  return split\_args(args\_str)  # 统一算法  def unify(*x*, *y*, *substitution*):  if *substitution* is None:  return None  elif *x* == *y*:  return *substitution*  elif is\_variable(*x*):  return unify\_var(*x*, *y*, *substitution*)  elif is\_variable(*y*):  return unify\_var(*y*, *x*, *substitution*)  elif is\_function(*x*) and is\_function(*y*):  if get\_function\_name(*x*) != get\_function\_name(*y*):  return None  args\_x = get\_arguments(*x*)  args\_y = get\_arguments(*y*)  if len(args\_x) != len(args\_y):  return None  for arg1, arg2 in zip(args\_x, args\_y):  *substitution* = unify(arg1, arg2, *substitution*)  if *substitution* is None:  return None  return *substitution*  else:  return None  def unify\_var(*var*, *x*, *substitution*):  if *var* in *substitution*:  return unify(*substitution*[*var*], *x*, *substitution*)  elif is\_variable(*x*) and *x* in *substitution*:  return unify(*var*, *substitution*[*x*], *substitution*)  elif occurs\_check(*var*, *x*, *substitution*):  return None  else:  *substitution* = deepcopy(*substitution*)  *substitution*[*var*] = *x*  return *substitution*  def occurs\_check(*var*, *x*, *substitution*):  if *var* == *x*:  return True  elif is\_function(*x*):  for arg in get\_arguments(*x*):  if occurs\_check(*var*, arg, *substitution*):  return True  elif *x* in *substitution*:  return occurs\_check(*var*, *substitution*[*x*], *substitution*)  return False  def substitute\_literal(*literal*, *substitution*):  new\_args = []  for arg in *literal*.args:  new\_arg = substitute\_term(arg, *substitution*)  new\_args.append(new\_arg)  return Literal(*literal*.name, new\_args, *literal*.is\_neg)  def substitute\_term(*term*, *substitution*):  while is\_variable(*term*) and *term* in *substitution*:  *term* = *substitution*[*term*]  if is\_function(*term*):  name = get\_function\_name(*term*)  args = get\_arguments(*term*)  new\_args = [substitute\_term(arg, *substitution*) for arg in args]  return f"{name}({', '.join(new\_args)})"  else:  return *term*  def substitute\_clause(*clause*, *substitution*):  new\_literals = [substitute\_literal(lit, *substitution*) for lit in *clause*.literals]  return Clause(new\_literals)  def resolve\_clauses(*clause\_str1*, *clause\_str2*):  clause1 = parse\_clause(*clause\_str1*)  clause2 = parse\_clause(*clause\_str2*)  standardizer = Standardizer()  clause1 = standardizer.standardize(clause1)  clause2 = standardizer.standardize(clause2)  for lit1 in clause1.literals:  for lit2 in clause2.literals:  if lit1.name == lit2.name and lit1.is\_neg != lit2.is\_neg:  substitution = {}  substitution = unify\_literals(lit1, lit2, substitution)  if substitution is not None:  new\_clause1 = substitute\_clause(clause1, substitution)  new\_clause2 = substitute\_clause(clause2, substitution)  resolved\_lit1 = substitute\_literal(lit1, substitution)  resolved\_lit2 = substitute\_literal(lit2, substitution)  new\_literals1 = [lit for lit in new\_clause1.literals if lit != resolved\_lit1]  new\_literals2 = [lit for lit in new\_clause2.literals if lit != resolved\_lit2]  combined\_literals = new\_literals1 + new\_literals2  unique\_literals = list(set(combined\_literals))  return Clause(unique\_literals)  return None  def unify\_literals(*lit1*, *lit2*, *substitution*):  if *lit1*.name != *lit2*.name or *lit1*.is\_neg == *lit2*.is\_neg:  return None  if len(*lit1*.args) != len(*lit2*.args):  return None  for arg1, arg2 in zip(*lit1*.args, *lit2*.args):  *substitution* = unify(arg1, arg2, *substitution*)  if *substitution* is None:  return None  return *substitution*  def format\_clause(*clause*):  return " ∨ ".join([str(lit) for lit in sorted(*clause*.literals, *key*=lambda *x*: str(*x*))])  if \_\_name\_\_ == "\_\_main\_\_":  if len(sys.argv) != 2:  print(f"Usage: python {sys.argv[0]} <input\_file>")  sys.exit(1)  input\_file = sys.argv[1]  try:  with open(input\_file, 'r') as file:  input1 = file.readline().strip()  input2 = file.readline().strip()  except FileNotFoundError:  print(f"Error: The file '{input\_file}' does not exist.")  sys.exit(1)  resolvent = resolve\_clauses(input1, input2)  if resolvent:  print(format\_clause(resolvent))  else:  print("Error: resolution failed")  1. 规则定义 (rules)  rules = [  {"conditions": {"种子有果皮"}, "conclusion": "被子植物"},  {"conditions": {"种子无果皮"}, "conclusion": "裸子植物"},  {"conditions": {"无茎叶", "无根"}, "conclusion": "藻类植物"},  {"conditions": {"被子植物", "有托叶"}, "conclusion": "蔷薇科"},  {"conditions": {"被子植物", "吸引菜粉蝶"}, "conclusion": "十字花科"},  {"conditions": {"被子植物", "十字形花冠"}, "conclusion": "十字花科"},  {"conditions": {"被子植物", "缺水环境"}, "conclusion": "仙人掌科"},  {"conditions": {"被子植物", "蔷薇科", "有刺"}, "conclusion": "玫瑰"},  {"conditions": {"被子植物", "水生", "可食用", "结果实"}, "conclusion": "荷花"},  {"conditions": {"被子植物", "仙人掌科", "喜阳", "有刺"}, "conclusion": "仙人球"},  {"conditions": {"藻类植物", "水生", "药用"}, "conclusion": "水棉"},  {"conditions": {"被子植物", "蔷薇科", "木本", "可食用", "结果实"}, "conclusion": "苹果树"},  {"conditions": {"被子植物", "十字花科", "黄色花", "可食用", "结果实"}, "conclusion": "油菜"},  {"conditions": {"藻类植物", "水生", "可食用", "有白色粉末"}, "conclusion": "海带"},  {"conditions": {"裸子植物", "木本", "叶片针状", "结果实"}, "conclusion": "松树"},  ]  - 描述：rules 列表包含多个规则，每个规则由条件集合 (conditions) 和结论 (conclusion) 组成。这些规则用于推导植物的分类，从基本的植物门类（如被子植物、裸子植物、藻类植物）到更具体的科或种（如蔷薇科、玫瑰）。  2. 特征列表 (all\_features)  all\_features = [  "种子有果皮", "种子无果皮", "无茎叶", "无根", "有托叶", "吸引菜粉蝶",  "十字形花冠", "缺水环境", "有刺", "水生", "可食用", "结果实", "喜阳",  "药用", "木本", "有白色粉末", "叶片针状", "黄色花",  "被子植物", "裸子植物", "藻类植物", "蔷薇科", "十字花科", "仙人掌科"  ]  - 描述：all\_features 列表包含了系统可以识别的所有植物特征。用户将根据这些特征输入相关编号来描述植物。  3. 植物集合 (plants)  plants = {"玫瑰", "荷花", "仙人球", "水棉", "苹果树", "油菜", "海带", "松树"}  - 描述：plants 集合列出了系统能够最终识别的具体植物种类。  4. 正向推理函数 (forward\_reasoning)  def forward\_reasoning(known\_features, rules):  """  正向推理函数，基于已知特征和规则库推导新的特征。  """  inferred = True  while inferred:  inferred = False  for rule in rules:  if rule["conditions"].issubset(known\_features) and rule["conclusion"] not in known\_features:  known\_features.add(rule["conclusion"])  inferred = True  return known\_features  - 功能：根据当前已知的特征集合，遍历所有规则，若规则的条件全部满足且结论尚未被添加，则将结论添加到已知特征中。重复此过程直到无法推导出新的特征。  5. 匹配植物函数 (match\_plants)  def match\_plants(known\_features, plant\_rules):  """  计算每个植物的匹配程度，返回匹配的植物及其匹配度。  """  plant\_scores = {}  for plant\_rule in plant\_rules:  plant = plant\_rule["conclusion"]  conditions = plant\_rule["conditions"]  match\_count = len(conditions.intersection(known\_features))  total\_conditions = len(conditions)  score = match\_count / total\_conditions  plant\_scores[plant] = score  return plant\_scores  - 功能：计算每个可能植物的匹配度。匹配度是已知特征与植物规则条件交集的比例。  6. 逆向推理函数 (backward\_reasoning)  def backward\_reasoning(known\_features, possible\_plants, plant\_rules):  """  逆向推理函数，询问用户可能的特征以提高匹配度。  """  for plant, score in possible\_plants:  print(f"\nIn order to determine whether it is {plant}, please answer the following features:")  for feature in plant\_rules[plant]:  if feature not in known\_features:  answer = input(f"Is the plant feature '{feature}'? (Y/n): ")  if answer.lower() in {"y", ""}:  known\_features.add(feature)  known\_features = forward\_reasoning(known\_features, rules)  return known\_features  - 功能：对于匹配度较高的可能植物，询问用户其特征是否存在，以进一步确认植物类别。每询问一个特征后，重新进行正向推理以更新已知特征。  7. 主函数 (main)  def main():  # Create a mapping from serial numbers to features  feature\_mapping = {str(i + 1): feature for i, feature in enumerate(all\_features)}  total\_features = len(all\_features)  while True:  print("\nThese are the features you can choose from: ")    # \*\*\* Modified Output Section Start \*\*\*  # Create a list of lists, each inner list represents a row with up to 3 features  table\_rows = []  row = []  for i, feature in enumerate(all\_features, start=1):  cell = f"{i}. {feature}"  row.append(cell)  if i % 4 == 0:  table\_rows.append(row)  row = []  if row: # Append any remaining features that don't make a full row  table\_rows.append(row)    # Define headers (empty since we're numbering the features)  headers = []  # Print the table using tabulate  print(tabulate(table\_rows, headers=headers, tablefmt="grid", stralign="left"))  # \*\*\* Modified Output Section End \*\*\*  print("\n")  # 用户输入特征序号  user\_features = set()  user\_input = input("Please enter the plant feature numbers separated by spaces: ").strip()    input\_numbers = user\_input.split()  invalid\_numbers = []  for num in input\_numbers:  if num in feature\_mapping:  user\_features.add(feature\_mapping[num])  else:  invalid\_numbers.append(num)    if invalid\_numbers:  print(f"The following numbers are invalid and will be ignored: {' '.join(invalid\_numbers)}")  # 正向推理  known\_features = forward\_reasoning(user\_features, rules)  # 检查是否得出植物名称  identified\_plants = plants.intersection(known\_features)  if identified\_plants:  print("\nResult:")  for plant in identified\_plants:  print(f"The plant may be: {plant}")  else:  # 计算匹配度  plant\_scores = match\_plants(known\_features, [rule for rule in rules if rule["conclusion"] in plants])  # 排序植物按匹配度降序  sorted\_plants = sorted(plant\_scores.items(), key=lambda x: x[1], reverse=True)    # 过滤出匹配度大于0的植物  possible\_plants = [(plant, score) for plant, score in sorted\_plants if score > 0]  if not possible\_plants:  print("\nSorry, the system cannot identify the plant.")  continue  print("\nFailed to uniquely identify the plant, possible plants and matching degree:")  for plant, score in possible\_plants:  print(f"{plant}，matching degree: {score\*100:.2f}%")    # 逆向推理，询问用户更多特征  # Pass a dictionary mapping plant to their conditions for easier access  plant\_rules\_dict = {plant: [cond for cond in rule["conditions"]] for rule, plant in zip([r for r in rules if r["conclusion"] in plants], [r["conclusion"] for r in rules if r["conclusion"] in plants])}    known\_features = backward\_reasoning(known\_features, possible\_plants, plant\_rules\_dict)  # 再次检查是否得出植物名称  identified\_plants = plants.intersection(known\_features)  if identified\_plants:  print("\nResult:")  for plant in identified\_plants:  print(f"The plant may be: {plant}")  else:  print("\nSorry, the system cannot identify the plant.")  continue\_query = input("\nAre you going to continue the next query? (Y/n): ")  if continue\_query.lower() not in {"y", "yes", ""}:  print("Exiting the system...")  break  else:  print("\033c", end="") # Clear the screen (works on some terminals)  if \_\_name\_\_ == "\_\_main\_\_":  main()  主要步骤  1. 特征映射：将每个特征与一个序号对应，便于用户通过输入序号选择特征。  python feature\_mapping = {str(i + 1): feature for i, feature in enumerate(all\_features)}  2. 特征展示：使用 tabulate 库以表格形式展示所有可选特征，每行最多显示4个特征。  python table\_rows = [] row = [] for i, feature in enumerate(all\_features, start=1): cell = f"{i}. {feature}" row.append(cell) if i % 4 == 0: table\_rows.append(row) row = [] if row: table\_rows.append(row) print(tabulate(table\_rows, headers=[], tablefmt="grid", stralign="left"))  3. 用户输入：用户输入特征的序号，程序将其转换为对应的特征集合。如果输入的序号无效，将提示用户忽略这些输入。  4. 正向推理：基于用户输入的特征，通过 forward\_reasoning 函数推导出更多的特征。  5. 植物识别：  - 直接识别：如果已知特征能直接对应到某个植物，则直接输出。  - 匹配度计算：如果无法直接识别，计算每个可能植物的匹配度，并按匹配度排序。  - 逆向推理：对于匹配度较高的植物，进一步询问用户相关特征，以提高识别准确性。  6. 继续查询：询问用户是否进行下一次查询，若否则退出系统，若是则清屏并重新开始。  from tabulate import tabulate  rules = [  {"conditions": {"种子有果皮"}, "conclusion": "被子植物"},  {"conditions": {"种子无果皮"}, "conclusion": "裸子植物"},  {"conditions": {"无茎叶", "无根"}, "conclusion": "藻类植物"},  {"conditions": {"被子植物", "有托叶"}, "conclusion": "蔷薇科"},  {"conditions": {"被子植物", "吸引菜粉蝶"}, "conclusion": "十字花科"},  {"conditions": {"被子植物", "十字形花冠"}, "conclusion": "十字花科"},  {"conditions": {"被子植物", "缺水环境"}, "conclusion": "仙人掌科"},  {"conditions": {"被子植物", "蔷薇科", "有刺"}, "conclusion": "玫瑰"},  {"conditions": {"被子植物", "水生", "可食用", "结果实"}, "conclusion": "荷花"},  {"conditions": {"被子植物", "仙人掌科", "喜阳", "有刺"}, "conclusion": "仙人球"},  {"conditions": {"藻类植物", "水生", "药用"}, "conclusion": "水棉"},  {"conditions": {"被子植物", "蔷薇科", "木本", "可食用", "结果实"}, "conclusion": "苹果树"},  {"conditions": {"被子植物", "十字花科", "黄色花", "可食用", "结果实"}, "conclusion": "油菜"},  {"conditions": {"藻类植物", "水生", "可食用", "有白色粉末"}, "conclusion": "海带"},  {"conditions": {"裸子植物", "木本", "叶片针状", "结果实"}, "conclusion": "松树"},  ]  all\_features = [  "种子有果皮", "种子无果皮", "无茎叶", "无根", "有托叶", "吸引菜粉蝶",  "十字形花冠", "缺水环境", "有刺", "水生", "可食用", "结果实", "喜阳",  "药用", "木本", "有白色粉末", "叶片针状", "黄色花",  "被子植物", "裸子植物", "藻类植物", "蔷薇科", "十字花科", "仙人掌科"  ]  plants = {"玫瑰", "荷花", "仙人球", "水棉", "苹果树", "油菜", "海带", "松树"}  def forward\_reasoning(*known\_features*, *rules*):  """  正向推理函数，基于已知特征和规则库推导新的特征。  """  inferred = True  while inferred:  inferred = False  for rule in *rules*:  # 如果规则的条件是已知特征的子集，且结论尚未得出  if rule["conditions"].issubset(*known\_features*) and rule["conclusion"] not in *known\_features*:  *known\_features*.add(rule["conclusion"])  inferred = True  return *known\_features*  def match\_plants(*known\_features*, *plant\_rules*):  """  计算每个植物的匹配程度，返回匹配的植物及其匹配度。  """  plant\_scores = {}  for plant\_rule in *plant\_rules*:  plant = plant\_rule["conclusion"]  conditions = plant\_rule["conditions"]  match\_count = len(conditions.intersection(*known\_features*))  total\_conditions = len(conditions)  score = match\_count / total\_conditions  plant\_scores[plant] = score  return plant\_scores  def backward\_reasoning(*known\_features*, *possible\_plants*, *plant\_rules*):  """  逆向推理函数，询问用户可能的特征以提高匹配度。  """  for plant, score in *possible\_plants*:  print(f"\nIn order to determine whether it is {plant}, please answer the following features:")  for feature in *plant\_rules*[plant]:  if feature not in *known\_features*:  answer = input(f"Is the plant feature '{feature}'? (Y/n): ")  if answer.lower() in {"y", ""}:  *known\_features*.add(feature)  *known\_features* = forward\_reasoning(*known\_features*, rules)  return *known\_features*  def main():  # Create a mapping from serial numbers to features  feature\_mapping = {str(i + 1): feature for i, feature in enumerate(all\_features)}  total\_features = len(all\_features)  while True:  print("\nThese are the features you can choose from: ")    # \*\*\* Modified Output Section Start \*\*\*  # Create a list of lists, each inner list represents a row with up to 3 features  table\_rows = []  row = []  for i, feature in enumerate(all\_features, *start*=1):  cell = f"{i}. {feature}"  row.append(cell)  if i % 4 == 0:  table\_rows.append(row)  row = []  if row: # Append any remaining features that don't make a full row  table\_rows.append(row)    # Define headers (empty since we're numbering the features)  headers = []  # Print the table using tabulate  print(tabulate(table\_rows, *headers*=headers, *tablefmt*="grid", *stralign*="left"))  # \*\*\* Modified Output Section End \*\*\*  print("\n")  # 用户输入特征序号  user\_features = set()  user\_input = input("Please enter the plant feature numbers separated by spaces: ").strip()    input\_numbers = user\_input.split()  invalid\_numbers = []  for num in input\_numbers:  if num in feature\_mapping:  user\_features.add(feature\_mapping[num])  else:  invalid\_numbers.append(num)    if invalid\_numbers:  print(f"The following numbers are invalid and will be ignored: {' '.join(invalid\_numbers)}")  # 正向推理  known\_features = forward\_reasoning(user\_features, rules)  # 检查是否得出植物名称  identified\_plants = plants.intersection(known\_features)  if identified\_plants:  print("\nResult:")  for plant in identified\_plants:  print(f"The plant may be: {plant}")  else:  # 计算匹配度  plant\_scores = match\_plants(known\_features, [rule for rule in rules if rule["conclusion"] in plants])  # 排序植物按匹配度降序  sorted\_plants = sorted(plant\_scores.items(), *key*=lambda *x*: *x*[1], *reverse*=True)    # 过滤出匹配度大于0的植物  possible\_plants = [(plant, score) for plant, score in sorted\_plants if score > 0]  if not possible\_plants:  print("\nSorry, the system cannot identify the plant.")  continue  print("\nFailed to uniquely identify the plant, possible plants and matching degree:")  for plant, score in possible\_plants:  print(f"{plant}，matching degree: {score\*100:.2f}%")    # 逆向推理，询问用户更多特征  # Pass a dictionary mapping plant to their conditions for easier access  plant\_rules\_dict = {plant: [cond for cond in rule["conditions"]] for rule, plant in zip([r for r in rules if r["conclusion"] in plants], [r["conclusion"] for r in rules if r["conclusion"] in plants])}    known\_features = backward\_reasoning(known\_features, possible\_plants, plant\_rules\_dict)  # 再次检查是否得出植物名称  identified\_plants = plants.intersection(known\_features)  if identified\_plants:  print("\nResult:")  for plant in identified\_plants:  print(f"The plant may be: {plant}")  else:  print("\nSorry, the system cannot identify the plant.")  continue\_query = input("\nAre you going to continue the next query? (Y/n): ")  if continue\_query.lower() not in {"y", "yes", ""}:  print("Exiting the system...")  break  else:  print("\033c", *end*="") # Clear the screen (works on some terminals)  if \_\_name\_\_ == "\_\_main\_\_":  main()  五、实验结果：      六、实验体会：  通过实验，我学会了如何运用贝叶斯网络处理不确定性信息，并体会到了概率推理在实际问题中的应用价值。实验让我掌握了逻辑推理中的消解规则，提高了我的逻辑分析和编程实现能力。实验加深了我对专家系统构建的理解，锻炼了我在知识表示和推理策略设计方面的实践技能。 | | | | | | |
| 教 师  评 语 |  | | | | | |