南京邮电大学

实 验 报 告

（2024 / 2025 学年 第 一 学期）

课程名称	离散数学
实验名称	利用真值表求主析取范式和主合取范式
实验时间	2024 年 9 月 26 日
指导单位	计算机学院计算机科学与技术系
指导教师	柯昌博

学生姓名	于明宏	班级学号	B23041011
学院(系)	计算机学院	专　　业	计算机科学与技术系

实 验 报 告

实验名称	利用真值表求主析取范式和主合取范式			指导教师	柯昌博
实验类型	验证	实验学时	4	实验时间	2024.9.26

一、 实验目的和要求

为了进一步理解利用真值表求主析取范式和主合取范式的理论和方法，具体的实验要求如下：

输入：可以是具体的命题公式，也可以是任意的命题公式，如果是给定的具体命题公式，此命题公式中应该包含“否定¬，析取∨，合取∧，条件→，双条件↔”这 5 种联接词中的 3 种，其中这 3 种联接词中必须包括“条件→” 或“双条件↔”；命题变元最好不少于 3 个。

输出：命题公式对应的真值表，及对应的主析取范式和主合取范式。

二、实验环境(实验设备)

硬件：微型计算机

软件：Windows 操作系统、Microsoft Visual C++ 2022

三、实验原理及内容

1、这个程序实现了一个用于解析和计算命题逻辑公式的工具，用户输入逻辑公式后，程序首先对公式进行符号替换和简化，将其从中缀表达式转换为后缀表达式。随后，通过构造真值表，对公式进行布尔逻辑运算，并生成公式的主析取范式（PDNF）和主合取范式（PCNF）。程序采用栈来辅助表达式的处理，使用映射和集合存储命题变量及其值，最终输出真值表、PDNF 和 PCNF 的结果。

2、C++源代码：

```
#include <cstdio>
#include <cstring>
#include <windows.h>
#include <iostream>
#include <set>
#include <map>
using namespace std;

// Global variables for formula representation
string Original_Formula;  // Original formula entered by the user
string Simp_Formula;      // Simplified formula (after replacing symbols like -> with >)
string Suffix_Formula;    // Postfix (suffix) notation of the formula
string PCNF;              // The Principal Conjunctive Normal Form of the formula
string PDNF;              // The Principal Disjunctive Normal Form of the formula
char ch[15] = "()!&|->"; // Allowed characters in the formula
map<char, int> variableMap; // Map of variables and their values (0 or 1)
set<char> variables;      // Set of distinct variables found in the formula
int res;                  // Result of the evaluation of the formula
int var_cnt;              // Count of distinct variables in the formula

// Stack class for expression handling
class Stack {
public:
    Stack(int mSize) {
        maxtop = mSize - 1;
        st = new char[mSize];  // Initialize the stack array
        top = -1;  // Start with an empty stack
    }
```

```
    ~Stack() {
        delete[] st;  // Destructor to free the allocated memory
    }
    bool Push(char x) {  // Push operation
        if (top == maxtop) return false;  // If stack is full, return false
        st[++top] = x;  // Push element onto the stack
        return true;
    }
    bool Pop() {  // Pop operation
        if (top == -1) return false;  // If stack is empty, return false
        top--;  // Remove the top element
        return true;
    }
    char Top() {  // Get the top element
        return st[top];
    }
private:
    int top;  // Index of the top element
    char* st; // Stack array
    int maxtop; // Maximum index for the stack
};

Stack Sim_stack(200); // Create a stack with a maximum size of 200
int a, b;  // Temporary variables used for calculations

// Logical NOT operation
void Not() {
    a = Sim_stack.Top();
    Sim_stack.Pop();
    res = (a == 1 ? 0 : 1);  // If a is 1, result is 0; otherwise, result is 1
    Sim_stack.Push(res);  // Push the result back onto the stack
}

// Logical AND operation
void And() {
```

```
        res = a * b;  // AND operation is equivalent to multiplication
        Sim_stack.Push(res);  // Push the result onto the stack
    }

    // Logical OR operation
    void Or() {
        res = (a + b == 0) ? 0 : 1;  // OR operation: if both are 0, result is 0;
otherwise, 1
        Sim_stack.Push(res);  // Push the result onto the stack
    }

    // Logical implication (IF) operation
    void If() {
        res = (b == 1 && a == 0) ? 0 : 1;  // IF b is 1 and a is 0, result is 0;
otherwise, 1
        Sim_stack.Push(res);  // Push the result onto the stack
    }

    // Logical biconditional (IFF) operation
    void Iif() {
        res = (a == b ? 1 : 0);  // If a and b are equal, result is 1; otherwise, 0
        Sim_stack.Push(res);  // Push the result onto the stack
    }

    // Helper function to handle NOT operation separately
    void Not_Not(char c) {
        if (c != '!') {  // If the character is not '!', pop two values from the stack
            a = Sim_stack.Top();
            Sim_stack.Pop();
            b = Sim_stack.Top();
            Sim_stack.Pop();
        }
    }

    // Switch to handle different logical operators
```

```
void Switch_Operator(char c) {
    switch (c) {
    case '~': Iif(); break;  // Handle biconditional
    case '>': If(); break;   // Handle implication
    case '|': Or(); break;   // Handle OR
    case '&': And(); break;  // Handle AND
    case '!': Not(); break;  // Handle NOT
    }
}

// Function to compare operator precedence
bool Jude_canin(char out) {
    char in = Sim_stack.Top();
    int i, o;
    switch (in) {  // Assign precedence values for operators on the stack
    case '#': i = 0; break;
    case '(': i = 1; break;
    case '~': i = 3; break;
    case '>': i = 5; break;
    case '|': i = 7; break;
    case '&': i = 9; break;
    case '!': i = 11; break;
    case ')': i = 12; break;
    }
    switch (out) {  // Assign precedence values for incoming operators
    case '(': o = 12; break;
    case '~': o = 2; break;
    case '>': o = 4; break;
    case '|': o = 6; break;
    case '&': o = 8; break;
    case '!': o = 10; break;
    case ')': o = 1; break;
    }
    return i < o;  // Return true if the operator on the stack has lower precedence
}
```

```
    // Convert infix formula to postfix (suffix) notation
    void Change_to_sufexp() {
        string Tmp = "";
        Sim_stack.Push('#');  // Push a special character to mark the bottom of the stack
        for (int i = 0; i < Simp_Formula.length(); i++) {
            if (isalpha(Simp_Formula[i])) {  // If the character is a variable, add it to the result
                Tmp += Simp_Formula[i];
                continue;
            }
            if (Jude_canin(Simp_Formula[i]))  // If precedence is lower, push the operator onto the stack
                Sim_stack.Push(Simp_Formula[i]);
            else if (Simp_Formula[i] == ')') {  // If it's a closing parenthesis, pop until '(' is found
                while (Sim_stack.Top() != '(') {
                    Tmp += Sim_stack.Top();
                    Sim_stack.Pop();
                }
                Sim_stack.Pop();  // Remove '(' from the stack
            }
            else {  // Pop operators with higher precedence from the stack
                do {
                    Tmp += Sim_stack.Top();
                    Sim_stack.Pop();
                } while (!Jude_canin(Simp_Formula[i]));
                Sim_stack.Push(Simp_Formula[i]);  // Push the current operator onto the stack
            }
        }
        while (Sim_stack.Top() != '#') {  // Pop all remaining operators
            Tmp += Sim_stack.Top();
            Sim_stack.Pop();
```

```
        }
        Sim_stack.Pop();  // Clear the stack
        Suffix_Formula = Tmp;  // Store the resulting postfix formula
    }

    // Evaluate the postfix formula
    void Calculate() {
        for (int i = 0; i < Suffix_Formula.length(); i++) {
            if (isalpha(Suffix_Formula[i])) {  // If it's a variable, push its value
onto the stack
                Sim_stack.Push(variableMap[Suffix_Formula[i]]);
            }
            else {  // Otherwise, perform the corresponding logical operation
                Not_Not(Suffix_Formula[i]);
                Switch_Operator(Suffix_Formula[i]);
            }
        }
    }

    // Replace symbols with their corresponding logical symbols for display
    string ReplaceSymbols(const string& formula) {
        string result;
        for (char ch : formula) {
            if (ch == '>') result += "→";
            else if (ch == '~') result += "←→";
            else if (ch == '&') result += "∧";
            else if (ch == '|') result += "∨";
            else if (ch == '!') result += "┐";
            else result += ch;
        }
        return result;
    }

    // Function to print the truth table and calculate PDNF and PCNF
    void Print() {
```

```
        cout << "Truth Table:" << endl;
        for (char var : variables) {
            cout << var << "\t";
        }

        string formattedFormula = ReplaceSymbols(Simp_Formula);
        cout << formattedFormula << endl;

        int combinations = 1 << var_cnt;  // Number of possible truth combinations
        for (int i = 0; i < combinations; i++) {
            int index = 0;
            for (char var : variables) {
                variableMap[var] = (i >> index) & 1;   // Assign truth values to
variables
                cout << variableMap[var] << "\t";
                index++;
            }

            Calculate();  // Evaluate the formula for the current truth assignment
            if (res == 1) {  // If the result is true, add to PDNF
                if (!PDNF.empty()) PDNF += " ∨ ";
                PDNF += "(";
                for (char var : variables) {
                    if (variableMap[var] == 1) {
                        PDNF += var;
                    }
                    else {
                        PDNF += "¬" + string(1, var);
                    }
                    PDNF += " ∧ ";
                }
                PDNF.erase(PDNF.size() - 3);  // Remove the trailing " ∧ "
                PDNF += ")";
            }
            else {  // If the result is false, add to PCNF
```

```
                if (!PCNF.empty()) PCNF += " ∧ ";
                PCNF += "(";
                for (char var : variables) {
                    if (variableMap[var] == 1) {
                        PCNF += "¬" + string(1, var);
                    }
                    else {
                        PCNF += var;
                    }
                    PCNF += " ∨ ";
                }
                PCNF.erase(PCNF.size() - 3);  // Remove the trailing " ∨ "
                PCNF += ")";
            }
            cout << res << endl;  // Print the result of the formula for the current
truth assignment
        }

        if (!PDNF.empty() && PDNF.back() == '∨') PDNF.erase(PDNF.size() - 3);  //
Clean up PDNF
        if (!PCNF.empty() && PCNF.back() == '∧') PCNF.erase(PCNF.size() - 3);  //
Clean up PCNF
    }

    // Generate PDNF and remove extra symbols if necessary
    void generatePDNF() {
        if (!PDNF.empty() && PDNF.back() == '∨') PDNF.erase(PDNF.size() - 3);
    }

    // Generate PCNF and remove extra symbols if necessary
    void generatePCNF() {
        if (!PCNF.empty() && PCNF.back() == '∧') PCNF.erase(PCNF.size() - 3);
    }

    // Function to display menu instructions
```

```
    void Menu() {
        HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);  // Get console handle
        cout << "Symbol Input Rules:" << endl;
        cout << " ¬ : !       " << endl;
        cout << " ∧: &        " << endl;
        cout << " ∨: |        " << endl;
        cout << " →: ->       " << endl;
        cout << "←→: <->    " << endl;
        cout << "Note: Parentheses and variables A-Z, a-z are all supported." << endl;
    }

    // Function to input and validate the formula
    void InputANDjude_formula() {
        char str1[100] = { 0 }, str2[100];  // Buffer to hold user input and processed
formula
        int cnt = 0;
        cout << "-----------------------------------------------------" << endl;
        cout << "Please input the propositional formula (type 'exit' to exit):" <<
endl;
        gets_s(str1);  // Get input from the user
        for (int i = 0; i < strlen(str1);) {  // Replace symbols like -> with internal
representations
            if (str1[i] == '-') { str2[cnt++] = '>'; i += 2; }
            else if (str1[i] == '<') { str2[cnt++] = '~'; i += 3; }
            else { str2[cnt++] = str1[i]; i += 1; }
        }
        str2[cnt++] = '\0';
        Original_Formula = str1;  // Store the original formula
        Simp_Formula = str2;      // Store the simplified formula
    }

    // Function to enhance robustness by checking for invalid characters
    bool Enh_Robustness() {
        bool Rob = true;
        if (Original_Formula == "exit") return true;  // Exit condition
```

```
        for (int i = 0; i < Original_Formula.length(); i++) {  // Check each character
in the formula
            if        (strchr(ch,        Original_Formula[i])        ==        nullptr
&& !isalpha(Original_Formula[i])) {
                cout << "Input error, please check illegal characters." << endl;
                Original_Formula = "";  // Clear the formula on error
                return false;
            }
        }
        return Rob;
    }

    // Function to count the number of distinct variables in the formula
    void count_varcnt() {
        variables.clear();  // Clear any previously counted variables
        for (int i = 0; i < Simp_Formula.length(); i++) {
            if (isalpha(Simp_Formula[i])) {  // Add variables (alphabet characters) to
the set
                variables.insert(Simp_Formula[i]);
            }
        }
        var_cnt = static_cast<int>(variables.size());  // Count the number of distinct
variables
    }

    int main() {
        Menu();  // Display menu instructions
        while (1) {
            do {
                InputANDjude_formula();  // Get user input
            } while (!Enh_Robustness());  // Validate input
            if (Original_Formula == "exit") {  // If user types 'exit', end the program
                cout  <<  "--------------------------------------------------"  <<
endl;
                cout << "Thank you for using!" << endl;
```

```
            system("pause");
            return 0;
        }

        count_varcnt();  // Count the number of variables in the formula
        Change_to_sufexp();  // Convert the formula to postfix notation
        Print();  // Print the truth table and calculate PDNF and PCNF
        generatePDNF();  // Generate PDNF
        generatePCNF();  // Generate PCNF

        cout << "PDNF: " << endl << PDNF << endl;  // Output PDNF
        cout << "PCNF: " << endl << PCNF << endl;  // Output PCNF

        PDNF.clear();  // Clear PDNF for the next iteration
        PCNF.clear();  // Clear PCNF for the next iteration
    }
    return 0;
}
```

3、运行结果：

```
Symbol Input Rules:
 ┐: !
 ∧: &
 ∨: |
 →: ->
←→: <->
Note: Parentheses and variables A-Z, a-z are all supported.
-----------------------------------------------------
Please input the propositional formula (type 'exit' to exit):
P&!Q
Truth Table:
P       Q       P∧┐Q
0       0       0
1       0       1
0       1       0
1       1       0
```

PDNF:

(P ∧ ¬Q)

PCNF:

(P ∨ Q) ∧ (P ∨ ¬Q) ∧ (¬P ∨ ¬Q)

Please input the propositional formula (type 'exit' to exit):

(P->!Q)<->R|S

Truth Table:

P	Q	R	S	(P→¬Q)←→R∨S
0	0	0	0	0
1	0	0	0	0
0	1	0	0	0
1	1	0	0	1
0	0	1	0	1
1	0	1	0	1
0	1	1	0	1
1	1	1	0	0
0	0	0	1	1
1	0	0	1	1
0	1	0	1	1
1	1	0	1	0
0	0	1	1	1
1	0	1	1	1
0	1	1	1	1
1	1	1	1	0

PDNF:

(P ∧ Q ∧ ¬R ∧ ¬S) ∨ (¬P ∧ ¬Q ∧ R ∧ ¬S) ∨ (P ∧ ¬Q ∧ R ∧ ¬S) ∨ (¬P ∧ Q ∧ R ∧ ¬S) ∨ (¬P ∧ ¬Q ∧ ¬R ∧ S) ∨ (P ∧ ¬Q ∧ ¬R ∧ S) ∨ (¬P ∧ Q ∧ ¬R ∧ S) ∨ (¬P ∧ ¬Q ∧ R ∧ S) ∨ (P ∧ ¬Q ∧ R ∧ S) ∨ (¬P ∧ Q ∧ R ∧ S)

PCNF:

(P ∨ Q ∨ R ∨ S) ∧ (¬P ∨ Q ∨ R ∨ S) ∧ (P ∨ ¬Q ∨ R ∨ S) ∧ (¬P ∨ ¬Q ∨ ¬R ∨ S) ∧ (¬P ∨ ¬Q ∨ R ∨ ¬S) ∧ (¬P ∨ ¬Q ∨ ¬R ∨ ¬S)

Please input the propositional formula (type 'exit' to exit):

!P&Q<->R

Truth Table:

P	Q	R	¬P∧Q←→R
0	0	0	1
1	0	0	1
0	1	0	0
1	1	0	1
0	0	1	0
1	0	1	0
0	1	1	1
1	1	1	0

PDNF:

(¬P ∧ ¬Q ∧ ¬R) ∨ (P ∧ ¬Q ∧ ¬R) ∨ (P ∧ Q ∧ ¬R) ∨ (¬P ∧ Q ∧ R)

PCNF:

(P ∨ ¬Q ∨ R) ∧ (P ∨ Q ∨ ¬R) ∧ (¬P ∨ Q ∨ ¬R) ∧ (¬P ∨ ¬Q ∨ ¬R)

Please input the propositional formula (type 'exit' to exit):

exit

Thank you for using!

四、实验小结（包括问题和解决方法、心得体会、意见与建议等）

说明：这部分内容主要包括：在编程、调试或测试过程中遇到的问题及解决方法、本次实验的心得体会、进一步改进的设想等。

（一）实验中遇到的主要问题及解决方法

1.问题：在实验过程中，复杂的逻辑表达式（尤其是包含条件 ″→″ 和双条件 ″↔″ 的公式）在解析过程中容易出现错误，特别是在转换为后缀表达式时容易出错。

解决方法：通过引入优先级判断以及优化栈的操作顺序，确保在处理中能够准确识别每个逻辑运算符的优先级，避免表达式解析错误。

2.问题：在生成真值表的过程中，由于对条件和双条件运算符的逻辑操作不熟悉，导致生成的真值表不正确。

解决方法：仔细查阅相关布尔逻辑的资料，并通过调试程序，校验每个逻辑运算符的计算规则。最终通过增加单元测试，确保每个运算的结果都是正确的。

3.问题：生成的 PDNF 和 PCNF 初期存在格式不规范问题，输出结果中存在多余的空格和符号，使得最终结果不美观。

解决方法：在最后的输出阶段，对生成的表达式进行格式化处理，去除不必要的空格和多余符号，使得 PDNF 和 PCNF 的表示更简洁清晰。

（二）实验心得

通过本次实验，我对布尔逻辑运算及真值表的生成有了更深入的理解，特别是对复杂逻辑表达式的解析和转换过程有了更清晰的认识。实验中的编程部分提升了我的代码调试能力，让我意识到逻辑问题在编写代码时的重要性。此外，通过调试程序，增强了我在处理复杂数据结构（如栈）的操作能力。本次实验还让我认识到在处理逻辑运算时，需要严格遵循运算优先级和符号规则，以确保结果的正确性。

（三）意见与建议（没有可省略）

可以提供更多的时间上机操作，以确保更多程序设计思路得以实现，提升掌握程度和编程能力。

五、支撑毕业要求指标点

支撑毕业要求的指标点为：

☐ 1-4 掌握计算机科学与技术领域的专业知识，能将专业知识用于分析和解决计算机领域复杂工程问题。

☑ 2-1 能够应用数学、自然科学和工程科学的基本知识，识别和分析计算机领域复杂工程问题的特征。

六、指导教师评语 (含学生能力达成度的评价)					
成　绩		批阅人		日　期	

如果不太想写太多字，“指导教师评语”也可以设计为如下的各选择项用打勾形式（仅仅作为一个简单示例，请各课程负责人根据课程和实验情况以及支撑的指标点来自行设定选择项，同一门课程的不同实验评分细则项允许存在不同）：

	评分项	优秀	良好	中等	合格	不合格
评	遵守实验室规章制度					
	学习态度					
	算法思想准备情况					
	程序设计能力					
分	解决问题能力					
	课题功能实现情况					
	算法设计合理性					
	算法效能评价					
	回答问题准确度					
细	报告书写认真程度					
	内容详实程度					
	文字表达熟练程度					
	其它评价意见					
则	本次实验能力达成评价（总成绩）					