命题逻辑真值表、主范式

2153726 罗宇翔

(经济与管理学院, 信息管理与信息系统)

目录

1 实验目的

掌握命题逻辑中的联接词、真值表、主范式等，进一步能用它们来解决实际问题。

2 实验内容

求任意一个命题公式的真值表（B），并根据真值表求主范式（C）。

3 实验环境

3.1 Visual Studio Code

Version: 1.89.1

Browser:

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)

AppleWebKit/537.36 (KHTML, like Gecko) Code/1.89.1

Chrome/120.0.6099.291 Electron/28.2.8 Safari/537.36

3.2 g++

Apple clang version 14.0.0 (clang-1400.0.29.202)

Target: x86_64-apple-darwin21.6.0

Thread model: posix

4 实验原理和方法

4.1 真值表

表征逻辑事件输入和输出之间全部可能状态的表格。列出命题公式真假值的表。通常以 1 表示真，0 表示假。命题公式的取值由组成命题公式的命题变元的取值和命题联结词决定，命题联结词的真值表给出了真假值的算法。真值表是在逻辑中使用的一类数学表，用来确定一个表达式是否为真或有效。

4.2 主范式

主析取范式：在含有 n 个命题变元的简单合取式中, 若每个命题变元与其否定不同时存在, 而两者之一出现一次且仅出现一次, 则称该简单合取式为极小项。由若干个不同的极小项组成的析取式称为主析取范式; 与 A 等价的主析取范式称为 A 的主析取范式。任意含 n 个命题变元的非永假命题公式 A 都存在与其等价的主析取范式, 并且是惟一的。

主合取范式：在含有 n 个命题变元的简单析取式中，若每个命题变元与其否定不同时存在，而两者之一出现一次且仅出现一次，称该简单析取式为极大项。由若干个不同的极大项组成的合取式称为主合取范式；与 A 等价的主合取范式称为 A 的主合取范式。任意含 n 个命题变元的非永真命题公式 A 都存在与其等价的主合取范式，并且是惟一的。

5 实验代码

```
#include <math.h>

#include <algorithm>
#include <iostream>
#include <queue>
#include <stack>
#include <vector>
using namespace std;

int is_operator(char chr) {
    switch (chr) {
    case '&':
    case '|':
    case '>':
    case '~':
        return 2;
    case '!':
```

```
        return 1;
    default:
        return 0;
    }
}

class Question {
private:
    string postfix;
    vector< char > letters;

private:
    bool is_operator(char chr) {
        switch (chr) {
        case '!':
        case '&':
        case '|':
        case '>':
        case '~':
        case '(':
        case ')':
            return true;

        default:
            return false;
        }
    }

    int calculate_precedence(char chr) {
        switch (chr) {
        case '!':
            return 5;
```

```
        case '&':
            return 4;
        case '|':
            return 3;
        case '>':
            return 2;
        case '~':
            return 1;

        default:
            return 0;
        }
    }

    string infix2postfix(string infix) {
        stack< char > operatorStack;
        stack< char > numberStack;
        string postfix = "";
        while (!operatorStack.empty()) {
            operatorStack.pop();
        }
        while (!numberStack.empty()) {
            numberStack.pop();
        }
        for (int i = 0; i < infix.size(); i++) {
            if (!is_operator(infix[i]))  // handle Propositions
            {
                postfix.push_back(infix[i]);
            } else  // handle Operators
            {
                if (infix[i] == ')')  // handle the right
    bracket
```

```
            {
                while (operatorStack.top() != '('  //
until meet the left bracket
                {
                    postfix.push_back(operatorStack.top());
                    operatorStack.pop();
                };
                operatorStack.pop();
            } else if (infix[i] == '('  // handle the left
 bracket
            {
                operatorStack.push(infix[i]);
            } else {
                int precedenceOutStack =
calculate_precedence(infix[i]);
                while (!operatorStack.empty() &&
precedenceOutStack <= calculate_precedence(operatorStack.top
())) {
                    postfix.push_back(operatorStack.top());
                    operatorStack.pop();
                }
                operatorStack.push(infix[i]);
            }
        }
    }
    while (!operatorStack.empty()) {
        postfix.push_back(operatorStack.top());
        operatorStack.pop();
    }
    return postfix;
 }
```

```
    vector< char > extract_letters(string expression) {
        vector< char > letters;
        for (int i = 0; i < expression.size(); i++) {
            if (!is_operator(expression[i])) {
                if (find(letters.begin(), letters.end(),
  expression[i]) == letters.end()) {
                    letters.push_back(expression[i]);
                }
            }
        }
        return letters;
    }

public:
    Question(string infix) {
        postfix = infix2postfix(infix);
        letters = extract_letters(infix);
    }
    string get_postfix() {
        return postfix;
    }
    vector< char > get_letters() {
        return letters;
    }
};

class BinaryCode {
private:
    vector< string > codes;

private:
```

```
    void dfs(vector< string > &result, string current, int n,
   int endlen) {
        if (n == endlen) {
            result.push_back(current);
            return;
        }
        dfs(result, current + "0", n + 1, endlen);
        dfs(result, current + "1", n + 1, endlen);
    }

public:
    BinaryCode(Question question) {
        int endlen = question.get_letters().size();
        vector< string > result;
        dfs(result, "", 0, endlen);
        codes = result;
    }
    vector< string > get() {
        return codes;
    }
};

class Validator {
private:
    bool is_alphabet(char chr) {
        return ('a' <= chr && chr <= 'z') || ('A' <= chr && chr
    <= 'Z');
    }
    void error(string &expression, int &pos) {
        printf("syntax error near `%c`[%d]\n", expression[pos],
    pos + 1);
    }
```

```
 bool proposition(string &expression, int &pos) {
     if (expression.size() == 1) {
         return true;
     }
     if (pos == 0) {
         return !is_alphabet(expression[pos + 1]);
     } else if (pos == expression.size() - 1) {
         return !is_alphabet(expression[pos - 1]);
     } else {
         return !is_alphabet(expression[pos - 1]) && !
is_alphabet(expression[pos + 1]);
     }
 }
 bool left_bracket(string &expression, int &pos) {
     if ((pos != 0 && is_alphabet(expression[pos - 1]))) {
// left brackets shouldn't after Proposition
         return false;
     }
     if (expression.size() == 1) {  // left brackets shouldn
't exist solely
         return false;
     }
     return true;
 }
 bool right_bracket(string &expression, int &pos) {
     if ((pos != expression.size() - 1 && is_alphabet(
expression[pos + 1]))) {  // right brackets shouldn't before
 Proposition
         return false;
     }
     if (pos == 0) {  // right brackets shouldn't be first
         return false;
```

```
        }
        if (pos != 0 && is_operator(expression[pos - 1])) {  //
 right brackets shouldn't after Operator
            return false;
        }
        return true;
    }
    bool logic_not(string &expression, int &pos) {
        if (pos == expression.size() - 1) {  // `!` shouldn't
be last
            return false;
        }
        if (pos != 0 && is_alphabet(expression[pos - 1])) {  //
 `!` shouldn't be after Proposition
            return false;
        }
        return true;
    }
    bool double_operator(string &expression, int &pos) {
        if (pos == 0 || pos == expression.size() - 1) {  //
double Operator shouldn't be at first or last
            return false;
        }
        if (!is_alphabet(expression[pos - 1]) && expression[pos
 - 1] != ')') {  // double Operator must after Proposition(
except `)`)
            return false;
        }
        if (expression[pos + 1] != '!' && expression[pos + 1]
!= '(' && !is_alphabet(expression[pos + 1])) {  // double
Operator must before Proposition(except `!` or `(`)
            return false;
```

```
        }
        return true;
    }
    bool is_empty(string &expression) {
        if (expression.size() == 0) {
            return true;
        } else {
            for (int i = 0; i < expression.size(); i++) {
                if (is_alphabet(expression[i])) {
                    return false;
                }
            }
            return true;
        }
    }

public:
    Validator() {}
    bool check(string &expression) {
        if (is_empty(expression)) {
            cout << "empty expression\n";
            return false;
        }
        int bracketNum = 0;
        for (int i = 0; i < expression.size(); i++) {
            if (expression[i] == '(') {  // check left brackets
                if (left_bracket(expression, i)) {
                    bracketNum++;
                } else {
                    error(expression, i);
                    return false;
                }
```

```
            }
            if (expression[i] == ')') {  // check right
brackets
                if (right_bracket(expression, i)) {
                    bracketNum--;
                } else {
                    error(expression, i);
                    return false;
                }
            }
        }
        if (bracketNum != 0) {  // check brackets
            if (bracketNum < 0)
                cout << "miss left bracket\n";
            else
                cout << "miss right bracket\n";
            return false;
        }
        for (int i = 0; i < expression.size(); i++) {  // check
 proposition
            if (is_alphabet(expression[i])) {
                if (!proposition(expression, i)) {
                    error(expression, i);
                    return false;
                }
            }
            if (is_operator(expression[i]) == 1) {  // check
single operator
                if (!logic_not(expression, i)) {
                    error(expression, i);
                    return false;
                }
```

```
            } else if (is_operator(expression[i]) == 2) {  //
    check double operator
                if (!double_operator(expression, i)) {
                    error(expression, i);
                    return false;
                }
            } else {  // check undefined
                if (!is_alphabet(expression[i]) && expression[i
    ] != '(' && expression[i] != ')') {
                    printf("`%c`[%d] is undefined\n",
    expression[i], i + 1);
                    return false;
                }
            }
        }
        return true;
    }
};

class Solver {
private:
    int conjunction_(int p, int q) {
        return p & q;
    }
    int disjunction_(int p, int q) {
        return p | q;
    }
    int condition_(int p, int q) {
        return !p | q;
    }
    int bicondition_(int p, int q) {
        return (p & q) | (!p & !q);
```

```
    }
    int negation_(int p) {
        return !p;
    }

public:
    Solver() {}
    int solve(Question &question, string &trueValues) {
        string postfix = question.get_postfix();
        vector< char > letters = question.get_letters();
        stack< int > propositions;
        while (!propositions.empty()) {
            propositions.pop();
        }
        for (int i = 0; i < postfix.size(); i++) {
            if (is_operator(postfix[i]) == 0) {  // meet
   proposition
                int idx = find(letters.begin(), letters.end(),
   postfix[i]) - letters.begin();
                propositions.push(trueValues[idx] - '0');
            } else if (is_operator(postfix[i]) == 2) {  // meet
    double operator
                int secondProp = propositions.top();    // pop
   out two propositions
                propositions.pop();
                int firstProp = propositions.top();
                propositions.pop();
                switch (postfix[i]) {  // calculate the result
   and push in
                case '&':
                    propositions.push(conjunction_(firstProp,
   secondProp));
```

```
                    break;
                case '|':
                    propositions.push(disjunction_(firstProp,
   secondProp));
                    break;
                case '>':
                    propositions.push(condition_(firstProp,
   secondProp));
                    break;
                case '~':
                    propositions.push(bicondition_(firstProp,
   secondProp));
                    break;
                }
            } else {  // meet single operator
                int prop = propositions.top();
                propositions.pop();
                propositions.push(negation_(prop));
            }
        }
        return propositions.top();
    }
};

void one_turn(string infix) {  // integrate the calculation and
    log
    Question question(infix);
    cout << "=======================\n";
    cout << "true values\n";
    cout << "-----------------------\n";
    for (int i = 0; i < question.get_letters().size(); i++) {
        cout << question.get_letters()[i] << "\t";
```

```
}
cout << infix << endl;
cout << endl;
vector< string > codes = BinaryCode(question).get();
vector< int > DNF;
vector< int > CNF;
Solver solver;
for (int i = 0; i < codes.size(); i++) {
    string trueValue = codes[i];
    for (int j = 0; j < trueValue.size(); j++) {
        cout << trueValue[j] << "\t";
    }
    int result = solver.solve(question, trueValue);
    cout << result << endl;
    if (result == 0) {
        DNF.push_back(i);
    } else {
        CNF.push_back(i);
    }
}
cout << "=======================\n";
cout << "DNF:\n\t";
if (DNF.size() != 0) {
    printf("M(%d)", DNF[0]);
    for (int i = 1; i < DNF.size(); i++) {
        printf("/\\M(%d)", DNF[i]);
    }
} else {
    cout << 0;
}
cout << endl;
cout << "=======================\n";
```

```
    cout << "CNF:\n\t";
    if (CNF.size() != 0) {
        printf("m(%d)", CNF[0]);
        for (int i = 1; i < CNF.size(); i++) {
            printf("\\/m(%d)", CNF[i]);
        }
    } else {
        cout << 0;
    }
    cout << endl;
    cout << "=======================\n";
}

string strip(string &input) {
    string output = "";
    for (int i = 0; i < input.size(); i++) {
        if (input[i] != ' ') {
            output += input[i];
        }
    }
    return output;
}

int main() {
    while (true) {
        string infix;
        cout << "input expression, Qq to quit\n";
        cout << "-----------------------\n";
        cout << "`!`:negation\t`&`:conjunction\n`|`:disjunction
  \t`>`:condition\n`~`:bicondition\n";
        cout << "-----------------------\n";
        getline(cin, infix);
```

```
        infix = strip(infix);
        if (infix == "Qq") {
            cout << "thanks for using\n";
            break;
        }
        Validator validator;
        if (validator.check(infix)) {
            one_turn(infix);
        }
    }
    return 0;
}
```

6 实验数据及结果分析

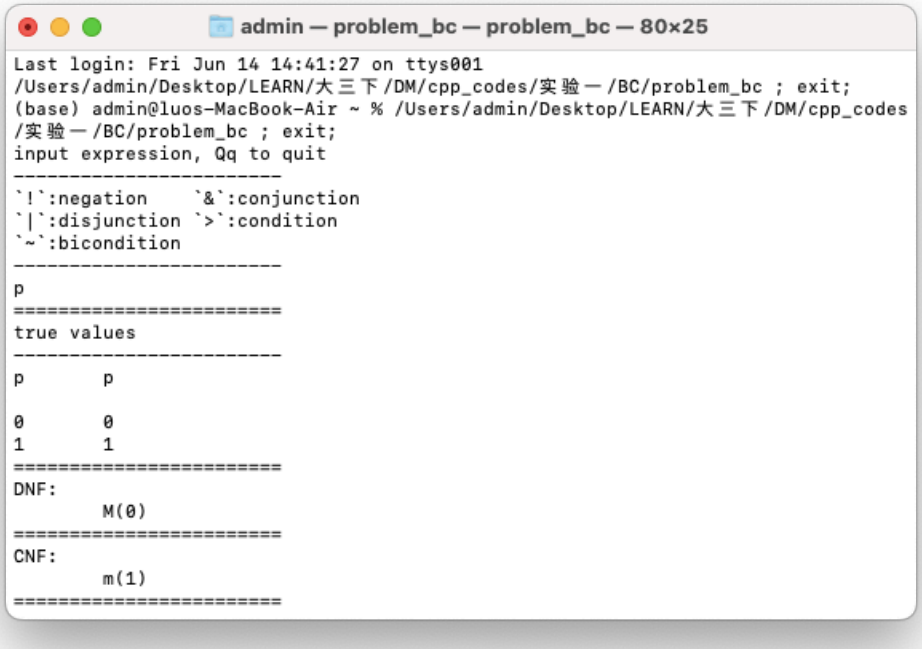

Figure 1: 正确输入 1

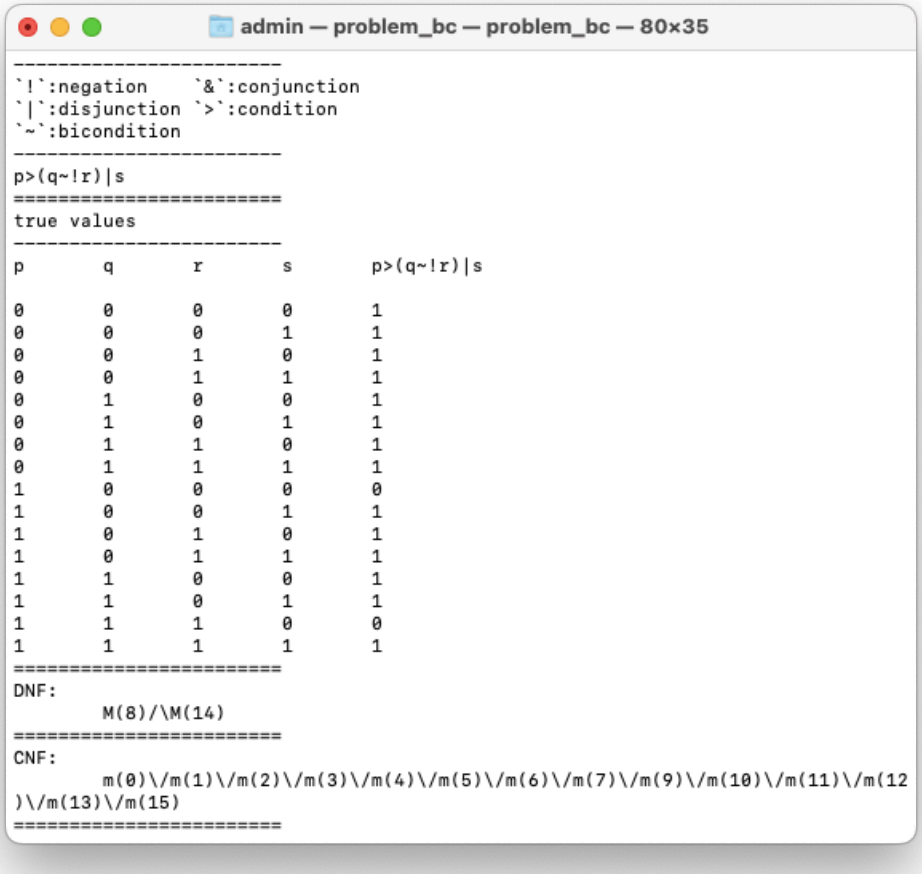

Figure 2: 正确输入 2

正确输入时，程序给出真值表和主范式。

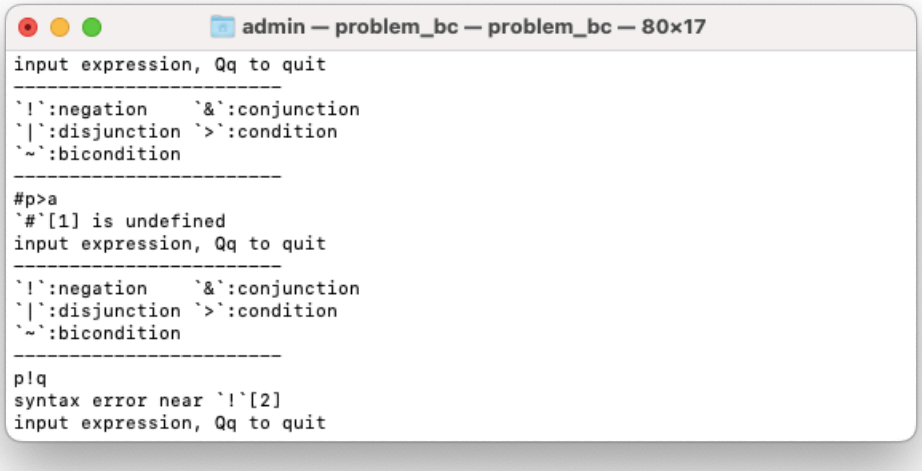

Figure 3: 错误输入 1

未定义字符和运算法则错误。

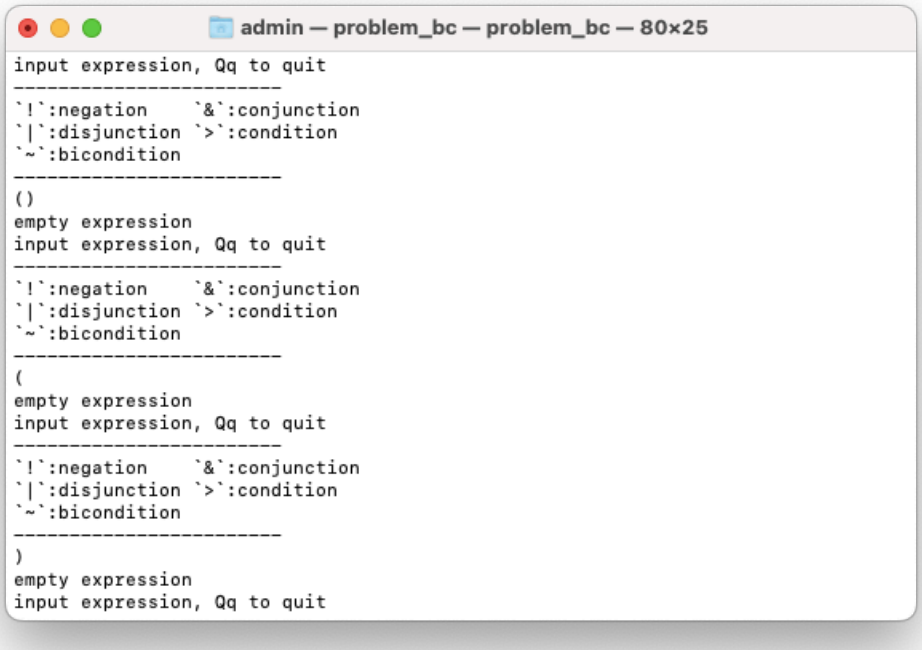

Figure 4: 错误输入 2

空字符错误。

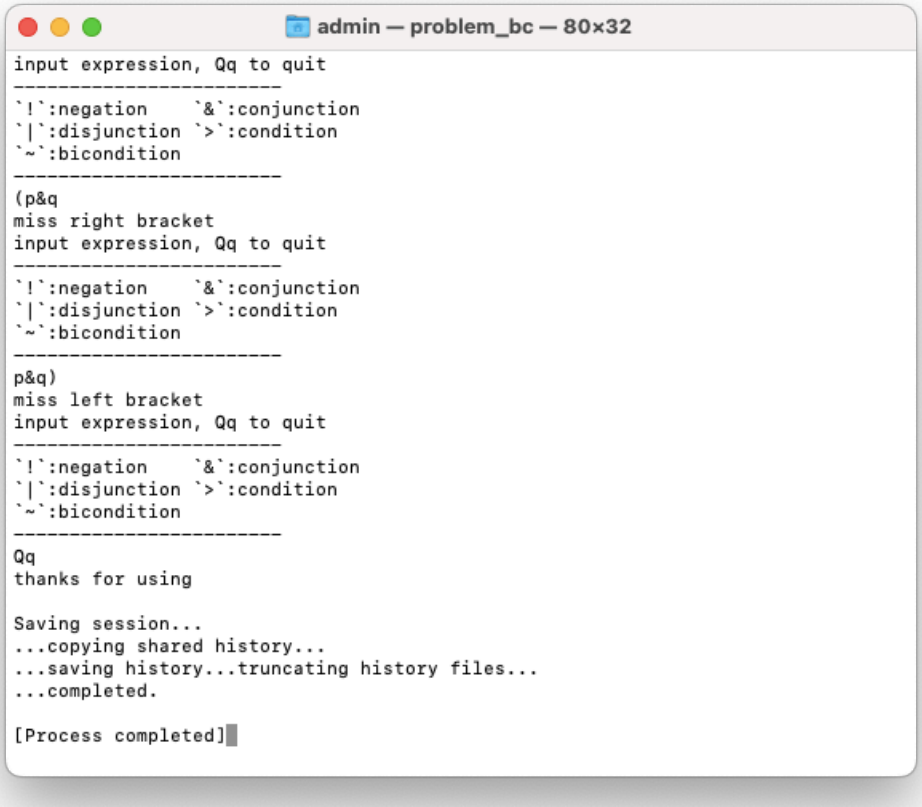

Figure 5: 错误输入 2

括号不匹配错误。