Министерство образования и науки РФ

Пермский национальный исследовательский политехнический университет

Электротехнический факультет

Кафедра Информационные технологии и автоматизированные системы

Дискретная математика

Лабораторная работа № 3

Тема: «Минимизация логических функций методом Квайна»

Выполнил студент группы

РИС-22-2б:

Прядеин И.А

Проверил: Ст. Преподаватель кафедры ИТАС:

Рустамханова Г.И.

г. Пермь – 2023

Оглавление

[Постановка задачи 3](#Постановка_Задачи)**[.](#Постановка_Задачи)**

[Алгоритм работы](#Алгоритм_работы) **[4.](#Алгоритм_работы)**

[Результат тестирования 7](#Результат_тестирования)**[.](#Результат_тестирования)**

[Исходный код 8](#Исходный_код)

# Постановка задачи

* Дан вектор логической функции длиной 16 символов, состоящий из 4 логических переменных.
* Реализовать нахождение СДНФ исходного вектора.
* Реализовать вывод таблицы покрытия.
* Реализовать нахождение импликант функции.

# Алгоритм работы

Программа разделена на два основных класса: “conjunction”, представляющий собой конъюнкцию и имеющий вектор значений 4-х переменных, и класс “QuineMinimization”, являющийся основным и реализующий основные задачи работы.

За проверку корректности вводимого вектора отвечает метод “IsValid” класса “QuineMinimization”. В случае, если пользователь введёт несоответствующий условию вектор, будет произведён повторный ввод до тех пор, пока исходный вектор не будет подходящим условию задачи.

Далее происходит преобразование вектора из строкового типа в вектор типа “conjunction”, каждый элемент которого обозначает конъюнкцию 4-х переменных.

Вывод исходной матрицы и функции в виде СДНФ реализовано при помощи методов “PrintMatrix” и “PrintConjunctions” класса “QuineMinimization”, также метода “Print” класса “conjunction”, выводящего конкретную конъюнкцию.

Получение импликант реализовано методом “GetImplicants” класса “QuineMinimization”. На “Рис. 1. изображен исходный код данной функции. Изначально объявляются переменные “result”, являющийся конечным вектором импликант, логическая переменная “combined”, векторы “uncombinedCon” и “combinedCon”. Далее в цикле происходит удаление дубликатов функцией “deleteDublicates”, склеивание и вставка данных значений в массив “combinedCon”. После подсчитывается количество несклеенных конъюнкций и вставка их в результирующий массив. Если подсчитываемое количество конъюнкций равно длине массива “uncombinedCon”, логическая переменная “combined” принимает значение “true”, обозначая тем самым окончание склейки. Следующая итерация произойдет в случае, если в массиве “uncombinedCon” остались несклеенные конъюнкции.

Рис. 1 - Функция “GetImplicants”

vector<conjunction> QuineMinimization::GetImplicants() {

vector<conjunction> result;

bool combined = false;

vector<conjunction> uncombinedCon = \_conjunctions;

vector<conjunction> combinedCon;

do {

deleteDublicates(uncombinedCon);

for (int i = 0; i < uncombinedCon.size() - 1; i++)

for (int j = i + 1; j < uncombinedCon.size(); j++)

if (canBeCombined(uncombinedCon[i], uncombinedCon[j]))

combinedCon.push\_back(Combine(uncombinedCon[i], uncombinedCon[j]));

int uncombinedCount = 0;

for (int i = 0; i < uncombinedCon.size(); i++)

if (!uncombinedCon[i].isCombined()) {

result.push\_back(uncombinedCon[i]);

uncombinedCount++;

}

if (uncombinedCount == uncombinedCon.size())

combined = true;

uncombinedCon.clear();

uncombinedCon = combinedCon;

combinedCon.clear();

} while (!combined);

return result;

}

В конце каждой итерации цикла происходит очистка массива “uncombinedCon”, присваивание данному массиву значения конъюнкций вектора “combinedCon” и его очистка. В конце работы функции возвращается вектор “result”.

Вывод таблицы покрытия реализован с помощью метода “PrintCoverageTable” класса “QuineMinimization”, изображенного на “Рис. 2”.

Рис. 2 - Функция “PrintCoverageTable”

void QuineMinimization::PrintCoverageTable(vector<conjunction> implicants) {

const int width = 10;

cout << left << setw(width) << "";

for (int i = 0; i < \_conjunctions.size(); i++) {

cout << left << setw(width) << \_conjunctions[i].getString();

}

cout << '\n';

for (int i = 0; i < implicants.size(); i++) {

cout << left << setw(width) << implicants[i].getString();

for (int j = 0; j < \_conjunctions.size(); j++) {

if (partOfConjunction(implicants[i], \_conjunctions[j]))

cout << left << setw(width) << "+";

else

cout << left << setw(width) << "";

}

cout << "\n\n";

}

}

Вначале метода объявляется переменная “width”, отвечающая за ширину выводимых значений. Далее выписываются все конъюнкты СДНФ. В цикле, проходящем по каждой импликанте, расположен внутренний цикл, отвечающий за проверку вхождений импликанты в исходные конъюнкты СДНФ. Проверка выплонена с помощью метода “partOfConjunction”.

# Результат тестирования

В случае, если пользователь введёт неверный вектор логической функции, появляется предложение ввести новый вектор, так же, как изображено на “Рис. 3”.

IMG_256

Рис. 3 - Неверный ввод вектора

Если введёный вектор удовлетворяет условиям как на “Рис. 4”, программа успешно выводит исходную матрицу, СДНФ и таблицу покрытия

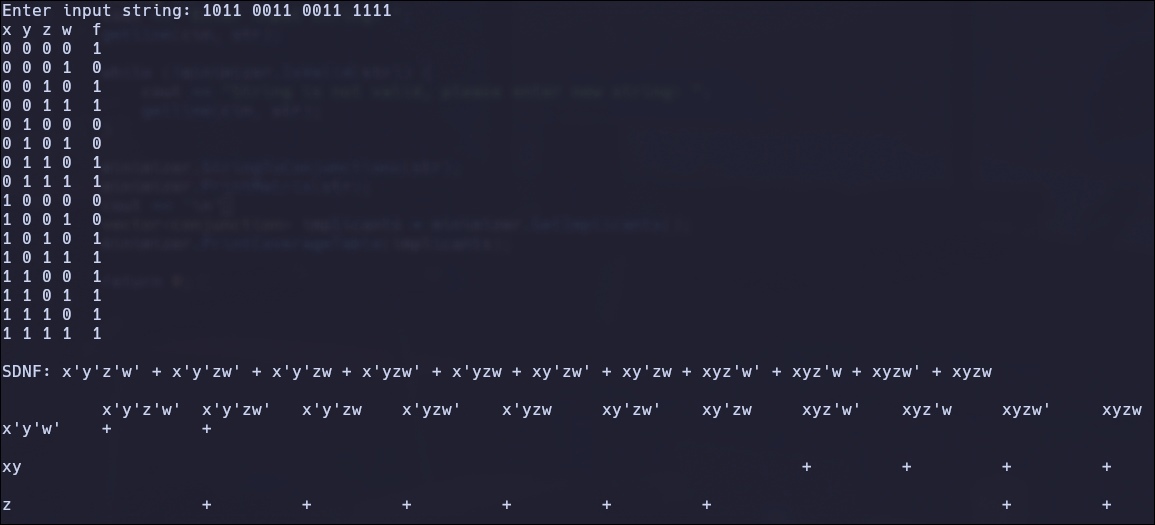


Рис. 4 - Успешное выполнение программы

Конечное минимальное покрытие находится пользователем.

# Исходный код

Файл “main.cpp”:

#include "quine\_minimization.h"  
#include <iostream>  
#include <string>  
using std::cout;  
using std::cin;  
using std::getline;  
using std::string;  
  
int main() {  
QuineMinimization minimizer;  
string str1 = "1111 1100 1111 1100";  
string str2 = "1011 0011 0011 1111";  
string str;  
cout << "Enter input string: ";  
getline(cin, str);  
  
while (!minimizer.IsValid(str)) {  
cout << "String is not valid, please enter new string: ";  
getline(cin, str);  
}  
  
minimizer.StringToConjunctions(str);  
minimizer.PrintMatrix(str);  
cout << '\n';  
vector<conjunction> implicants = minimizer.GetImplicants();  
minimizer.PrintCoverageTable(implicants);  
  
return 0;  
}

Файл “quine\_minimization.cpp”:

#include "quine\_minimization.h"  
  
bool QuineMinimization::IsValid(string& str) {  
str.erase(remove\_if(str.begin(), str.end(), isspace), str.end());  
if (str.size() == 16) {  
for (int i = 0; i < str.size(); i++)  
if (str[i] != '1' && str[i] != '0')  
return false;  
return true;  
}  
return false;  
}  
  
void QuineMinimization::StringToConjunctions(const string& str) {  
int i = 0;  
for (int x = 0; x < 2; x++)  
for (int y = 0; y < 2; y++)  
for (int z = 0; z < 2; z++)  
for (int w = 0; w < 2; w++) {  
if (str[i] == '1') {  
conjunction curr(x, y, z, w);  
\_conjunctions.push\_back(curr);  
}  
i++;  
}  
}  
  
void QuineMinimization::PrintMatrix(const string& str) {  
std::cout << "x y z w f\n";  
int i = 0;  
for (int x = 0; x < 2; x++)  
for (int y = 0; y < 2; y++)  
for (int z = 0; z < 2; z++)  
for (int w = 0; w < 2; w++) {  
if (str[i] == '1')  
std::cout << x << " " << y << " " << z << " " << w << " 1\n";  
else  
std::cout << x << " " << y << " " << z << " " << w << " 0\n";  
i++;  
}  
std::cout << "\nSDNF: ";  
  
PrintConjunctions(\_conjunctions);  
}  
  
void QuineMinimization::PrintConjunctions(vector<conjunction> conjunctions) {  
if (conjunctions.size() == 0)  
return;  
for (int i = 0; i < conjunctions.size(); i++) {  
conjunctions[i].Print();  
if (i != conjunctions.size() - 1)  
std::cout << " + ";  
}  
std::cout << '\n';  
}  
  
void QuineMinimization::PrintCoverageTable(vector<conjunction> implicants) {  
const int width = 10;  
cout << left << setw(width) << "";  
for (int i = 0; i < \_conjunctions.size(); i++) {  
cout << left << setw(width) << \_conjunctions[i].getString();  
}  
cout << '\n';  
  
for (int i = 0; i < implicants.size(); i++) {  
cout << left << setw(width) << implicants[i].getString();  
for (int j = 0; j < \_conjunctions.size(); j++) {  
if (partOfConjunction(implicants[i], \_conjunctions[j]))  
cout << left << setw(width) << "+";  
else  
cout << left << setw(width) << "";  
}  
cout << "\n\n";  
}  
}  
  
bool QuineMinimization::partOfConjunction(conjunction con1, conjunction con2) {  
int matchedCount = 0;  
int implicantBits = 0;  
  
for (int i = 0; i < con1.getLength(); i++) {  
if (con1.getBit(i) == con2.getBit(i))  
matchedCount++;  
if (con1.getBit(i) != -1)  
implicantBits++;  
}  
  
if (matchedCount == implicantBits)  
return true;  
  
return false;  
}  
  
vector<conjunction> QuineMinimization::GetImplicants() {  
vector<conjunction> result;  
  
bool combined = false;  
vector<conjunction> uncombinedCon = \_conjunctions;  
vector<conjunction> combinedCon;  
  
do {  
deleteDublicates(uncombinedCon);  
  
for (int i = 0; i < uncombinedCon.size() - 1; i++)  
for (int j = i + 1; j < uncombinedCon.size(); j++)  
if (canBeCombined(uncombinedCon[i], uncombinedCon[j]))  
combinedCon.push\_back(Combine(uncombinedCon[i], uncombinedCon[j]));  
  
int uncombinedCount = 0;  
  
for (int i = 0; i < uncombinedCon.size(); i++)  
if (!uncombinedCon[i].isCombined()) {  
result.push\_back(uncombinedCon[i]);  
uncombinedCount++;  
}  
  
if (uncombinedCount == uncombinedCon.size())  
combined = true;  
  
uncombinedCon.clear();  
uncombinedCon = combinedCon;  
combinedCon.clear();  
} while (!combined);  
  
return result;  
}  
  
void QuineMinimization::deleteDublicates(vector<conjunction>& conjunctions) {  
vector<conjunction> result;  
bool inVector;  
  
for (int i = 0; i < conjunctions.size(); i++) {  
inVector = false;  
for (int j = 0; j < result.size(); j++)  
if (isEqual(conjunctions[i], result[j])) {  
inVector = true;  
break;  
}  
if (!inVector)  
result.push\_back(conjunctions[i]);  
}  
  
conjunctions = result;  
}  
  
bool QuineMinimization::canBeCombined(conjunction con1, conjunction con2) {  
if (con1.CountBits() != con2.CountBits() || isEqual(con1, con2))  
return false;  
  
int diff = 0;  
for (int i = 0; i < con1.getLength(); i++)  
if (con1.getBit(i) != con2.getBit(i))  
diff++;  
  
if (diff > 1)  
return false;  
  
return true;  
}  
  
bool QuineMinimization::isEqual(conjunction con1, conjunction con2) {  
for (int i = 0; i < con1.getLength(); i++)  
if (con1.getBit(i) != con2.getBit(i))  
return false;  
return true;  
}  
  
conjunction QuineMinimization::Combine(conjunction& con1, conjunction& con2) {  
conjunction result(-1, -1, -1, -1);  
  
for (int i = 0; i < 4; i++)  
if (con1.getBit(i) != con2.getBit(i))  
result.setBit(i, -1);  
else  
result.setBit(i, con1.getBit(i));  
  
con1.setCombined(true);  
con2.setCombined(true);  
return result;  
}

Файл “conjunction.cpp”:

#include "conjunction.h"  
  
conjunction::conjunction(int x, int y, int z, int w) {  
\_bits = { x, y, z, w };  
combined = false;  
}  
  
void conjunction::Print() {  
if (\_bits[0] == 1)  
cout << "x";  
else if (\_bits[0] != -1)  
cout << "x'";  
  
if (\_bits[1] == 1)  
cout << "y";  
else if (\_bits[1] != -1)  
cout << "y'";  
  
if (\_bits[2] == 1)  
cout << "z";  
else if (\_bits[2] != -1)  
cout << "z'";  
  
if (\_bits[3] == 1)  
cout << "w";  
else if (\_bits[3] != -1)  
cout << "w'";  
}  
  
int conjunction::CountOnes() {  
int count = 0;  
for (int bit : \_bits)  
if (bit == 1)  
count++;  
return count;  
}  
  
int conjunction::CountBits() {  
int count = 0;  
for (int bit : \_bits)  
if (bit != -1)  
count++;  
return count;  
}  
  
int conjunction::getLength() {  
return \_bits.size();  
}  
  
int conjunction::getBit(unsigned int index) {  
return \_bits[index];  
}  
  
string conjunction::getString() {  
string str;  
  
if (\_bits[0] == 1)  
str += "x";  
else if (\_bits[0] != -1)  
str += "x'";  
  
if (\_bits[1] == 1)  
str += "y";  
else if (\_bits[1] != -1)  
str += "y'";  
  
if (\_bits[2] == 1)  
str += "z";  
else if (\_bits[2] != -1)  
str += "z'";  
  
if (\_bits[3] == 1)  
str += "w";  
else if (\_bits[3] != -1)  
str += "w'";  
  
return str;  
}  
  
void conjunction::setBit(unsigned int index, int bit) {  
\_bits[index] = bit;  
}  
  
bool conjunction::isCombined() {  
return combined;  
}  
  
void conjunction::setCombined(bool combined) {  
this->combined = combined;  
}