Министерство образования и науки РФ

Пермский национальный исследовательский политехнический университет

Электротехнический факультет

Кафедра информационных технологий и автоматизированных систем

Дискретная математика

Лабораторная работа № 3

Тема: «СКНФ и СДНФ»

Выполнил: студентка группы ИВТ-23-1б

Хорошилова К. П.

Проверил: ст. преподаватель

Рустамханова Г. И.

г. Пермь – 2024**Цель работы**

Изучить метод минимизации булевых функций с использованием таблицы истинности, СДНФ, СКНФ, импликантной матрицы и метода коротких покрытий.

**Задачи работы**

1. Составить таблицу истинности для булевой функции на основе заданного вектора истинности.
2. Построить СДНФ и СКНФ.
3. Создать импликантную матрицу, связывающую начальные термины функции с их покрытием.
4. Найти и вывести все возможные короткие покрытия для минимизации функции.
5. Оформить отчет.

# Код программы

using System;

using System.Collections.Generic;

using System.Linq;

public class Minimization

{

private List<List<int>> beforePCNF = new List<List<int>>(); // СКНФ

private List<List<int>> beforePDNF = new List<List<int>>(); // СДНФ

public List<List<int>> resultGluing = new List<List<int>>(); // Результат склеивания

private string vectBool;

private int countVar;

private List<List<int>> implicants = new List<List<int>>(); // Импликанты

public void ReadVector()

{

Console.WriteLine("Введите вектор истинности:");

vectBool = Console.ReadLine();

countVar = (int)Math.Log2(vectBool.Length);

}

public void CreateTruthTable()

{

int numRows = 1 << countVar;

if (vectBool.Length != numRows)

{

throw new ArgumentException("Вектор истинности должен соответствовать числу переменных.");

}

Console.WriteLine("Таблица истинности:");

Console.WriteLine(string.Join("\t", Enumerable.Range(1, countVar).Select(i => $"x{i}")) + "\tF");

for (int i = 0; i < numRows; i++)

{

var tmp = new List<int>();

for (int j = countVar - 1; j >= 0; j--)

{

tmp.Add((i >> j) & 1);

}

Console.WriteLine(string.Join("\t", tmp) + $"\t{vectBool[i]}");

if (vectBool[i] == '0')

beforePCNF.Add(tmp);

else

beforePDNF.Add(tmp);

}

}

public void ShowNF()

{

// Вывод СКНФ и СДНФ

Console.WriteLine("СКНФ:");

foreach (var row in beforePCNF)

{

Console.WriteLine(string.Join("", row));

}

Console.WriteLine("СДНФ:");

foreach (var row in beforePDNF)

{

Console.WriteLine(string.Join("", row));

}

}

private bool CanCombine(List<int> a, List<int> b, out List<int> result)

{

// Проверка возможности склеивания двух терминов

result = new List<int>(a);

int diffCount = 0;

for (int i = 0; i < a.Count; i++)

{

if (a[i] != b[i])

{

if (a[i] == -1 || b[i] == -1)

return false;

diffCount++;

result[i] = -1;

}

}

return diffCount == 1;

}

private List<List<int>> CombineTerms(List<List<int>> terms)

{

// Склеивание терминов

var newTerms = new List<List<int>>();

var usedFlag = new bool[terms.Count];

for (int i = 0; i < terms.Count; i++)

{

for (int j = i + 1; j < terms.Count; j++)

{

if (CanCombine(terms[i], terms[j], out var combined))

{

newTerms.Add(combined);

usedFlag[i] = true;

usedFlag[j] = true;

}

}

}

for (int i = 0; i < terms.Count; i++)

{

if (!usedFlag[i])

{

newTerms.Add(terms[i]);

}

}

return newTerms;

}

public void Gluing()

{

implicants = new List<List<int>>(beforePDNF);

while (true)

{

var newImplicants = CombineTerms(implicants);

if (newImplicants.SequenceEqual(implicants))

break;

implicants = newImplicants;

}

resultGluing = implicants;

}

public void PrintAllCombinedTerms()

{

// Вывод всех этапов склеивания

Console.WriteLine("Склеивание:");

foreach (var term in resultGluing)

{

Console.WriteLine(TermToString(term));

}

}

public List<List<int>> BuildImplicantMatrix()

{

// Создание матрицы импликант

var matrix = new List<List<int>>(resultGluing.Count);

for (int i = 0; i < resultGluing.Count; i++)

{

var row = new List<int>(beforePDNF.Count);

for (int j = 0; j < beforePDNF.Count; j++)

{

bool covers = true;

for (int k = 0; k < beforePDNF[j].Count; k++)

{

if (resultGluing[i][k] != -1 && resultGluing[i][k] != beforePDNF[j][k])

{

covers = false;

break;

}

}

row.Add(covers ? 1 : 0);

}

matrix.Add(row);

}

return matrix;

}

public List<int> MinimizeCover(List<List<int>> matrix)

{

// Минимизация покрытия

var result = new List<int>();

var covered = new bool[matrix[0].Count];

while (covered.Any(c => !c))

{

int bestRow = -1, maxCover = -1;

for (int i = 0; i < matrix.Count; i++)

{

int coverCount = 0;

for (int j = 0; j < matrix[i].Count; j++)

{

if (matrix[i][j] == 1 && !covered[j])

coverCount++;

}

if (coverCount > maxCover)

{

maxCover = coverCount;

bestRow = i;

}

}

if (bestRow != -1)

{

result.Add(bestRow);

for (int j = 0; j < matrix[bestRow].Count; j++)

{

if (matrix[bestRow][j] == 1)

covered[j] = true;

}

}

}

return result;

}

public void PrintImplicants()

{

Console.WriteLine("Итоговые импликанты:");

foreach (var term in resultGluing)

{

Console.WriteLine(TermToString(term));

}

}

public string TermToString(List<int> term)

{

return string.Join("", term.Select(t => t == -1 ? "-" : t.ToString()));

}

public void PrintMatrix(List<List<int>> matrix)

{

Console.WriteLine("Матрица импликант:");

foreach (var row in matrix)

{

Console.WriteLine(string.Join(" ", row));

}

}

}

class Program

{

static void Main()

{

var minimization = new Minimization();

minimization.ReadVector();

minimization.CreateTruthTable();

Console.WriteLine();

minimization.ShowNF();

minimization.Gluing();

minimization.PrintAllCombinedTerms();

var matrix = minimization.BuildImplicantMatrix();

minimization.PrintMatrix(matrix);

var minimizedCover = minimization.MinimizeCover(matrix);

Console.WriteLine("Минимизированная функция:");

foreach (var idx in minimizedCover)

{

Console.WriteLine(minimization.TermToString(minimization.resultGluing[idx]));

}

// Вывод итоговых импликантов

minimization.PrintImplicants();

}

}

**Анализ кода**

Этот код реализует минимизацию булевых функций методом склеивания.

Функция ReadVector запрашивает у пользователя ввод вектора истинности — строку из нулей и единиц, которая представляет значения функции F для всех возможных комбинаций переменных. Затем она рассчитывает количество переменных, используя логарифмическую зависимость log2 (длина вектора).

Функция CreateTruthTable строит таблицу истинности, сопоставляя каждую строку вектора истинности с комбинацией значений переменных. Для этого создаются все возможные комбинации значений переменных, представляющих двоичные числа от 0 до 2^n−1. Результат функции F из вектора истинности выводится вместе с этими комбинациями. Кроме того, строки таблицы, соответствующие значению F=0, сохраняются в список beforePCNF для построения СКНФ (совершенной конъюнктивной нормальной формы), а строки, где F=1, сохраняются в список beforePDNF для построения СДНФ (совершенной дизъюнктивной нормальной формы).

Функция ShowNF отвечает за вывод СКНФ и СДНФ. Она просто печатает сохраненные в списках beforePCNF и beforePDNF строки в соответствующем формате.

Функция CanCombine проверяет возможность склеивания двух строк таблицы истинности. Она проходит по всем элементам двух строк и проверяет, отличаются ли они ровно в одном месте. Если строки отличаются в одном месте, создается новая строка, где различающийся элемент заменяется на −1 (может быть 0 или 1).

Функция CombineTerms выполняет процесс склеивания терминов. Она принимает текущие строки (термины), проверяет возможность их склеивания с помощью функции CanCombine и создает новые термины. Строки, которые не были использованы при склеивании, добавляются в результат как есть.

Функция Gluing выполняет основной цикл склеивания. Она берет начальные термины из списка beforePDNF и многократно применяет CombineTerms, пока не перестанут появляться новые термины. Итоговые термины сохраняются в список resultGluing.

Функция PrintAllCombinedTerms выводит все получившиеся термины после завершения процесса склеивания. Каждый термин отображается в формате −1.

Функция BuildImplicantMatrix строит матрицу импликант. Строки матрицы соответствуют итоговым терминам после склеивания, а столбцы — строкам из beforePDNF. Если термин покрывает строку, в соответствующем месте матрицы записывается 1, иначе 0. Термин считается покрывающим строку, если все фиксированные значения термина совпадают с соответствующими значениями строки.

Функция MinimizeCover минимизирует покрытие, выбирая наименьшее множество строк (импликантов), которые покрывают все строки из beforePDNF. Алгоритм выбирает строку, покрывающую максимальное число ещё не покрытых столбцов, добавляет её в результат и помечает покрытые столбцы. Этот процесс повторяется, пока не будут покрыты все столбцы.

Функция PrintImplicants выводит итоговые термины, полученные после минимизации, в удобном формате. Каждый термин отображается с использованием символа −1.

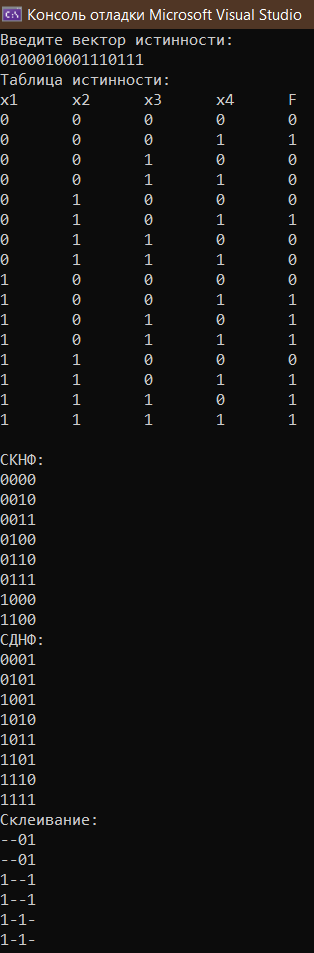
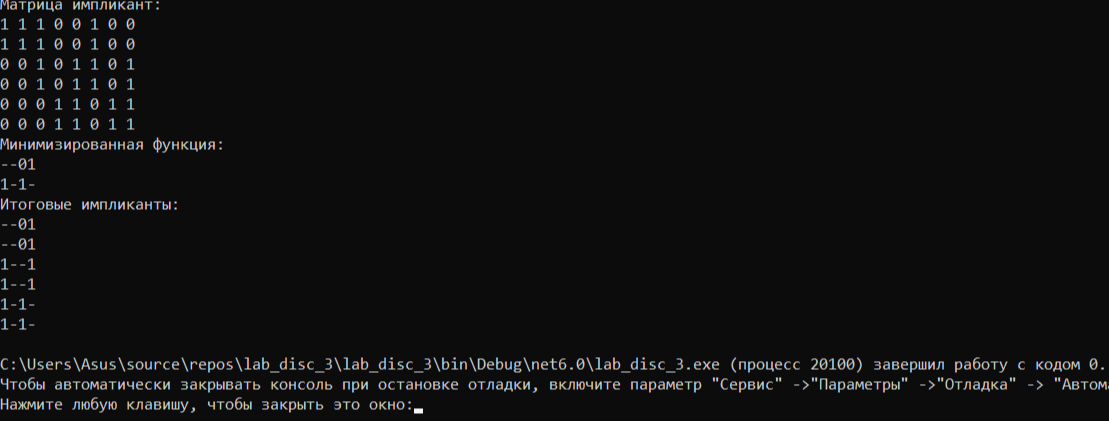
Функция TermToString преобразует термин (список чисел) в строку, где 0 и 1 сохраняются как есть, а −1 заменяется на −.

Функция PrintMatrix выводит матрицу импликант, созданную с помощью BuildImplicantMatrix, в виде таблицы из нулей и единиц.

В целом, программа запрашивает у пользователя вектор истинности, строит таблицу истинности, находит начальные термины для СДНФ, выполняет процесс склеивания для упрощения функции, строит матрицу импликант, минимизирует покрытие и выводит итоговую минимизированную булеву функцию.

Ссылка на Github: <https://github.com/ksinph/disc_math>

**Тесты**

# Заключение

# В ходе лабораторной работы я изучила метод минимизации булевых функций с использованием таблицы истинности, СДНФ, СКНФ, импликантной матрицы и метода коротких покрытий.