Министерство науки и высшего образования Российской Федерации Федеральное государственное автономное учреждение высшего образования «Пермский национальный исследовательский политехнический университет»

Электротехнический факультет   
Кафедра «Информационные технологии и автоматизированные системы»

**О Т Ч Ё Т  
по лабораторной работе №3**

**“Минимизация логических функций”**

Дисциплина: «Дискретная математика и Математическая логика»

Выполнил   
студент группы ИВТ-23-2б  
Злыгостев Д.Н.

Проверил   
доцент кафедры ИТАС  
Рустамханова Г. И.

Пермь 2024

**Оглавление**

[**1.Введение** 3](#_Toc183775436)

[**1.Цель работы** 3](#_Toc183775437)

[**2.Задачи** 3](#_Toc183775438)

[**3.Теоретические сведения** 4](#_Toc183775439)

[**4.Описание программы** 4](#_Toc183775440)

[**5.Структура программы** 5](#_Toc183775441)

[**6. Код** 6](#_Toc183775442)

[**7.Git-Hub** 16](#_Toc183775443)

[**8.Пример работы программы** 16](#_Toc183775444)

[**Заключение** 17](#_Toc183775445)

**1.Введение**

Минимизация логических функций является ключевой задачей в цифровой логике, проектировании схем и теории автоматов. Оптимизация булевых функций позволяет сократить количество логических элементов в цифровых схемах, снизить энергопотребление и увеличить их производительность. Одним из методов минимизации является метод Квайна, который представляет собой детерминированный и алгоритмически обоснованный подход к нахождению минимального представления булевых функций.

В данном проекте была разработана программа на языке C# для минимизации булевых функций методом Квайна, с возможностью генерации и отображения табличных истинности, ДНФ, СДНФ и СКНФ. Также реализован вывод пошаговых действий алгоритма для наглядности и удобства обучения.

**1.Цель работы**

Целью данного проекта является создание программы, которая:

1. Генерирует таблицу истинности булевой функции на основе ввода пользователя.
2. Строит исходную ДНФ и СКНФ.
3. Минимизирует функцию методом Квайна.
4. Демонстрирует этапы работы метода для анализа и обучения.
5. Выводит минимизированные представления функции в виде ДНФ и КНФ.

**2.Задачи**

Для достижения поставленной цели были сформулированы следующие задачи:

1. **Анализ требований к программе.**
   * Изучить метод Квайна-Мак-Класки.
   * Определить требования к интерфейсу программы.
   * Обеспечить поддержку булевых функций до 4 переменных.
2. **Разработка структуры программы.**
   * Организовать ввод данных.
   * Реализовать методы построения таблицы истинности, ДНФ и СКНФ.
   * Разработать алгоритмы минимизации с учетом требований.
3. **Реализация программы.**
   * Программировать алгоритм на языке C#.
   * Реализовать интерфейс командной строки для взаимодействия с пользователем.
   * Отладить программу для проверки корректности результатов.
4. **Тестирование и документация.**
   * Проверить программу на различных входных данных.
   * Описать структуру программы и принципы работы алгоритмов.

**3.Теоретические сведения**

**Метод Квайна-Мак-Класки**  
Метод Квайна-Мак-Класки используется для минимизации логических функций на основе анализа табличных истинности. Основные этапы метода:

**Группировка минтермов.** Минтермы группируются по количеству единиц в их двоичном представлении.

**Склеивание.** Минтермы, отличающиеся только в одном разряде, объединяются с заменой различающегося разряда на символ “-” (донт-кэр).

**Построение простых импликант.** Находятся минтермы, которые невозможно объединить дальше.

**Выделение существенных импликант.** Определяются импликанты, покрывающие уникальные минтермы.

**Построение минимизированной функции.** Используются все существенные и дополнительные импликанты для покрытия оставшихся минтермов.

**4.Описание программы**

**Метод Квайна**  
Метод Квайна используется для минимизации логических функций на основе анализа табличных истинности. Основные этапы метода:

1. **Группировка минтермов.** Минтермы группируются по количеству единиц в их двоичном представлении.
2. **Склеивание.** Минтермы, отличающиеся только в одном разряде, объединяются с заменой различающегося разряда на символ “-” (донт-кэр).
3. **Построение простых импликант.** Находятся минтермы, которые невозможно объединить дальше.
4. **Выделение существенных импликант.** Определяются импликанты, покрывающие уникальные минтермы.
5. **Построение минимизированной функции.** Используются все существенные и дополнительные импликанты для покрытия оставшихся минтермов.

**Булевые функции**  
Булевые функции могут быть представлены в двух формах:

* **Дизъюнктивная нормальная форма (ДНФ):** Представляет собой дизъюнкцию (логическое “ИЛИ”) конъюнкций (логическое “И”) переменных.
* **Конъюнктивная нормальная форма (КНФ):** Представляет собой конъюнкцию (логическое “И”) дизъюнкций (логическое “ИЛИ”) переменных.

Минимизация функции позволяет сократить число операций и переменных, что важно для цифрового проектирования

Программа разработана на языке C# с использованием среды .NET. Основные функции программы:

1. **Ввод данных.** Пользователь вводит строку длиной 16 символов (0 или 1), задающую таблицу истинности булевой функции.
2. **Построение таблицы истинности.** Программа генерирует таблицу для всех комбинаций значений четырех переменных.
3. **Построение ДНФ и СКНФ.** Формируются исходные формы, основанные на таблице истинности.
4. **Минимизация методом Квайна-Мак-Класки.**
   * Группировка минтермов.
   * Пошаговое склеивание с отображением промежуточных результатов.
   * Поиск существенных импликант.
5. **Вывод минимизированных форм.**
   * Сокращенная ДНФ (СДНФ).
   * Сокращенная КНФ (СКНФ).

Программа обеспечивает наглядность за счет отображения этапов склеивания, упрощения логических выражений и формирования результатов в понятной форме.

**5.Структура программы**

Программа состоит из нескольких основных методов:

1. Main()
   * Организует ввод и вывод данных.
   * Вызывает вспомогательные методы для построения таблицы истинности, минимизации и вывода результатов.
2. ToBinary(int number, int length)
   * Преобразует число в двоичный формат заданной длины.
3. BuildDNF(List<int> minterms)
   * Генерирует исходную ДНФ на основе минтермов.
4. FindPrimeImplicants(List<int> minterms)
   * Реализует алгоритм поиска простых импликант методом склеивания.
5. FindEssentialImplicants(List<int> minterms, List<string> primeImplicants)
   * Находит существенные импликанты для минимизации.
6. BuildSDNF(List<string> implicants)
   * Формирует сокращенную ДНФ.
7. BuildKNF(List<int> maxterms)
   * Генерирует исходную КНФ в скобках с использованием оператора V.
8. BuildSKNF(List<string> knfTerms)
   * Формирует сокращенную КНФ с минимальным количеством операций.

Каждый метод выполняет строго определенную функцию, что облегчает чтение и сопровождение кода.

**6. Код**

using System;

using System.Collections.Generic;

using System.Linq;

class Program

{

static void Main()

{

Console.Write("Введите 16 символов 0 и 1 для таблицы истинности: ");

string input = Console.ReadLine();

if (input.Length != 16 || input.Any(c => c != '0' && c != '1'))

{

Console.WriteLine("Некорректный ввод.");

return;

}

// Сбор минтермов из входной строки

List<int> minterms = new List<int>();

List<int> maxterms = new List<int>(); // Для СКНФ

Console.WriteLine("\nТаблица истинности:");

for (int i = 0; i < 16; i++)

{

string binary = Convert.ToString(i, 2).PadLeft(4, '0');

int value = input[i] - '0';

Console.WriteLine($"{binary} : {value}");

if (value == 1) minterms.Add(i); // Для ДНФ

else maxterms.Add(i); // Для СКНФ

}

// Построение исходной ДНФ

Console.WriteLine("\nИсходная ДНФ:");

List<string> dnfTerms = BuildDNF(minterms);

int counter = 1;

// Печать ДНФ с номерами для каждой связки

foreach (var term in dnfTerms)

{

Console.WriteLine($"{counter}. {term}");

counter++;

}

// Минимизация методом Квайна

List<string> primeImplicants = FindPrimeImplicants(minterms);

// Нахождение существенных импликант

List<string> essentialImplicants = FindEssentialImplicants(minterms, primeImplicants);

// Построение минимизированной функции (СДНФ)

Console.WriteLine("\nМинимизированная СДНФ:");

Console.WriteLine(BuildSDNF(essentialImplicants));

// Построение минимизированной функции (СКНФ)

Console.WriteLine("\nСКНФ:");

List<string> knfTerms = BuildKNF(maxterms);

Console.WriteLine(BuildSKNF(knfTerms));

}

// Преобразование числа в двоичное представление с заданной длиной

static string ToBinary(int number, int length)

{

return Convert.ToString(number, 2).PadLeft(length, '0');

}

// Построение исходной ДНФ

static List<string> BuildDNF(List<int> minterms)

{

List<string> terms = new List<string>();

foreach (int term in minterms)

{

string binary = ToBinary(term, 4);

string termStr = "";

for (int i = 0; i < binary.Length; i++)

{

if (binary[i] == '1')

termStr += $"X{i + 1}";

else

termStr += $"!X{i + 1}";

}

terms.Add(termStr);

}

return terms;

}

// Подсчет единиц в строке

static int CountOnes(string str)

{

return str.Count(c => c == '1');

}

// Преобразование двоичной строки в логическое выражение

static string ToLogicalExpression(string binary)

{

string expression = "";

for (int i = 0; i < binary.Length; i++)

{

if (binary[i] == '1')

expression += $"X{i + 1}";

else if (binary[i] == '0')

expression += $"!X{i + 1}";

}

return expression;

}

// Поиск простых импликант через попарные склейки

static List<string> FindPrimeImplicants(List<int> minterms)

{

List<List<string>> groups = new List<List<string>>(5);

for (int i = 0; i < 5; i++) groups.Add(new List<string>());

foreach (int m in minterms)

{

string binary = ToBinary(m, 4);

groups[CountOnes(binary)].Add(binary);

}

HashSet<string> primeImplicants = new HashSet<string>();

bool updated = true;

int mergeCount = 1; // Счетчик для склеиваний

while (updated)

{

updated = false;

List<List<string>> newGroups = new List<List<string>>(5);

for (int i = 0; i < 5; i++) newGroups.Add(new List<string>());

HashSet<string> used = new HashSet<string>();

// Выводим шаги объединения

Console.WriteLine("\nШаг склеивания:");

for (int i = 0; i < groups.Count - 1; i++)

{

foreach (var a in groups[i])

{

foreach (var b in groups[i + 1])

{

int diff = 0;

int diffIndex = -1;

for (int k = 0; k < a.Length; k++)

{

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

{

diff++;

diffIndex = k;

}

}

// Если различие ровно по одному биту, склеиваем

if (diff == 1)

{

updated = true;

char[] newTerm = a.ToCharArray();

newTerm[diffIndex] = '-';

string newTermStr = new string(newTerm);

if (!newGroups[CountOnes(newTermStr)].Contains(newTermStr))

newGroups[CountOnes(newTermStr)].Add(newTermStr);

used.Add(a);

used.Add(b);

// Выводим информацию о склеивании с логическим выражением и нумерацией

Console.WriteLine($"{mergeCount}. Склеивание: {ToLogicalExpression(a)} и {ToLogicalExpression(b)} → {ToLogicalExpression(newTermStr)}");

mergeCount++; // Увеличиваем счетчик склеиваний

}

}

}

}

// Добавляем оставшиеся импликанты, которые не были склеены

foreach (var group in groups)

{

foreach (var term in group)

{

if (!used.Contains(term)) primeImplicants.Add(term);

}

}

groups = newGroups;

}

return primeImplicants.ToList();

}

// Поиск существенных импликант

static List<string> FindEssentialImplicants(List<int> minterms, List<string> primeImplicants)

{

Dictionary<string, List<int>> coverage = new Dictionary<string, List<int>>();

foreach (string implicant in primeImplicants)

{

List<int> coveredMinterms = new List<int>();

foreach (int m in minterms)

{

string binary = ToBinary(m, 4);

bool match = true;

for (int i = 0; i < implicant.Length; i++)

{

if (implicant[i] != '-' && implicant[i] != binary[i])

{

match = false;

break;

}

}

if (match) coveredMinterms.Add(m);

}

coverage[implicant] = coveredMinterms;

}

HashSet<int> covered = new HashSet<int>();

List<string> essentialImplicants = new List<string>();

foreach (var item in coverage)

{

bool isEssential = false;

foreach (int term in item.Value)

{

if (!covered.Contains(term))

{

isEssential = true;

covered.Add(term);

}

}

if (isEssential) essentialImplicants.Add(item.Key);

}

return essentialImplicants;

}

// Построение минимизированной СДНФ

static string BuildSDNF(List<string> implicants)

{

List<string> terms = new List<string>();

foreach (string implicant in implicants)

{

string term = "";

for (int i = 0; i < implicant.Length; i++)

{

if (implicant[i] == '1') term += $"X{i + 1}";

else if (implicant[i] == '0') term += $"!X{i + 1}";

}

terms.Add(term);

}

return string.Join(" V ", terms);

}

// Построение минимизированной СКНФ

static List<string> BuildKNF(List<int> maxterms)

{

List<string> terms = new List<string>();

foreach (int term in maxterms)

{

string binary = ToBinary(term, 4);

List<string> termStr = new List<string>();

// Собираем логическое выражение для каждого терма

for (int i = 0; i < binary.Length; i++)

{

if (binary[i] == '1')

termStr.Add($"!X{i + 1}"); // Для 1 ставим отрицание

else

termStr.Add($"X{i + 1}"); // Для 0 оставляем переменную

}

// Добавляем терм с дизъюнкцией (V) и заключаем в скобки

terms.Add($"({string.Join(" V ", termStr)})");

}

return terms;

}

// Построение СКНФ

static string BuildSKNF(List<string> knfTerms)

{

return string.Join(" & ", knfTerms);

}

}

**7.Git-Hub**

Репозиторий для лабы

[ZligostevDenis/Diskretka\_Math](https://github.com/ZligostevDenis/Diskretka_Math)

**8.Пример работы программы**

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

Рисунок.1 – Вывод

**Заключение**

Программа успешно реализует минимизацию логических функций методом Квайна. Она подходит как для учебных целей, так и для практического использования в проектировании цифровых схем. Ключевые особенности программы включают:

* Подробный пошаговый вывод минимизации.
* Интуитивно понятный ввод данных.
* Генерация СДНФ и СКНФ в удобочитаемом формате.

Дальнейшее развитие программы может включать поддержку большего количества переменных и визуализацию алгоритма в графическом интерфейсе.