Министерство образования и науки Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования

«**Пермский национальный исследовательский политехнический университет**»

Кафедра «Информационные технологии и автоматизированные системы»

ОТЧЁТ

по лабораторной работе №3

Дисциплина: «Дискретная математика»

Тема: Минимизация логических функций.

Выполнил:

Студент группы ИВТ-20-2б

Карелов Вадим Андреевич

Проверила:

Старший преподаватель кафедры ИТАС

Рустамханова Г.И.

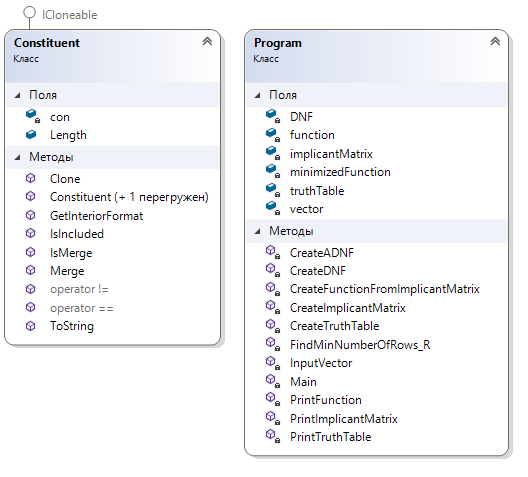
Пермь, 2021

**Постановка задачи**

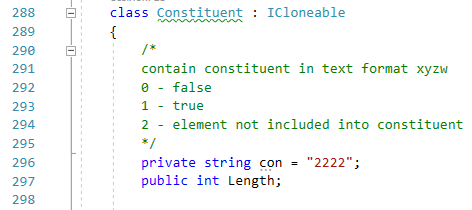
Ввести таблицу истинности логической функции. Реализовать метод Квайна-Макксласки для нахождения минимизированной формы логической функции.

**Анализ задачи**

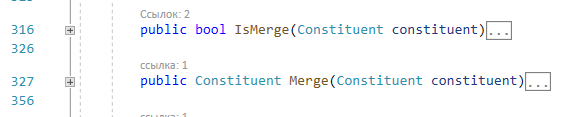
1. В программе используются 2 класса: Program & Constituent.



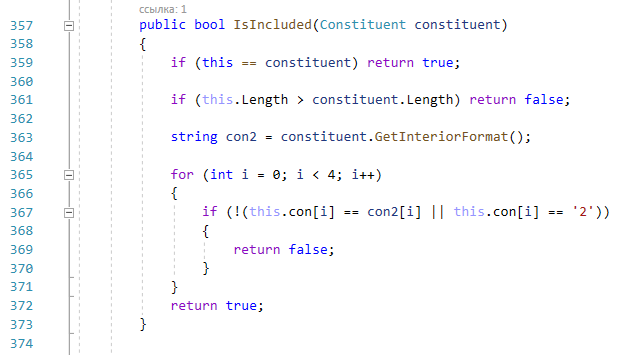
1. Класс Constituent. Хранит конституенту в виде строки. 0 символ соответствует x, 1 символ соответствует y, 2 символ соответствует z, 3 символ соответствует w. Для каждого символа: 0 – ложь, 1 – истина, 2 – элемента в конституенте нет.



* 1. В классе реализованы методы для проверки возможности склейки и осуществления самой склейки.

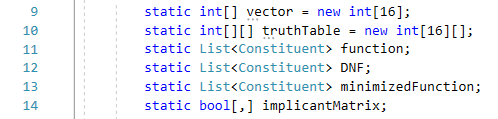


* 1. Метод проверки вхождения конституенты в другую.



* 1. Переопределен метод приведения типа в string, чтобы конституента выводилась в виде xyzw. Предусмотрен случай, когда конституента – это ИСТИНА. В таком случае будет выводиться 1.

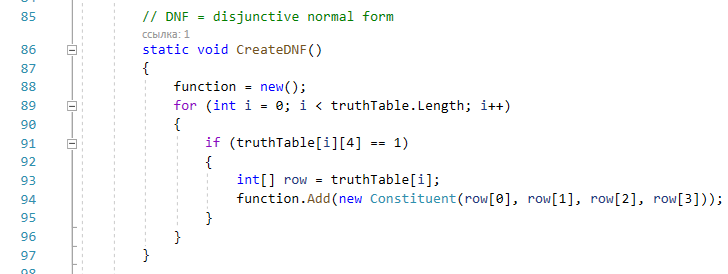
1. Класс Program является основным.
   1. Вектор, таблица истинности, ДНФ, СкДнф, импликантная матрица и минимизированная форма хранятся в виде статических переменных класса.



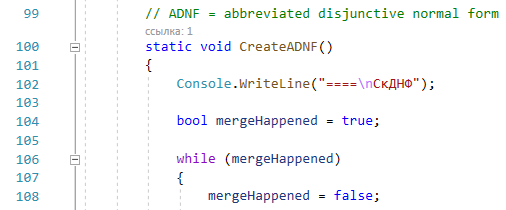
* 1. Из Main() поэтапно вызываются шаги алгоритма.



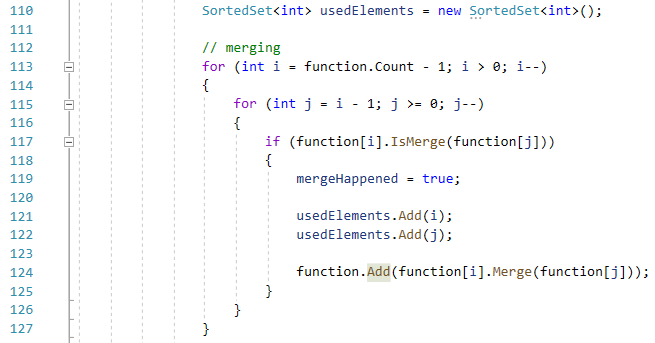
* 1. Первыми вызываются методы ввода вектора, создания таблицы истинности и вывода этой таблицы на экран.
  2. Затем создается ДНФ.



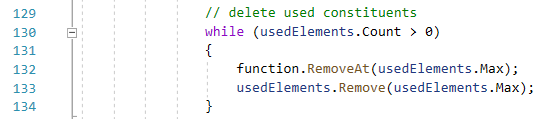
* 1. После создания ДНФ, получившаяся функция сохраняется для будущей обработки.
  2. Склеивание. На каждом этапе проверяется – произошла ли склейка.



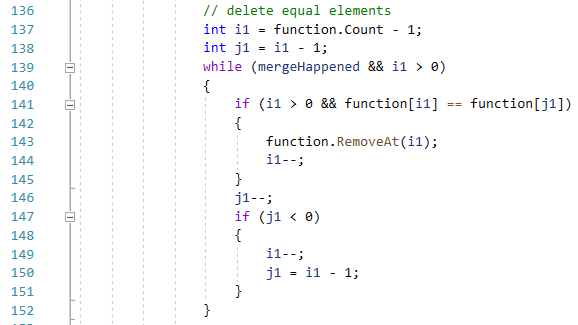
* + 1. Происходит проход по текущей функции с конца. Если есть возможность – происходит склейка и получившийся элемент помещается в конец.



* + 1. Производится проверка – использованные конституенты удаляются.



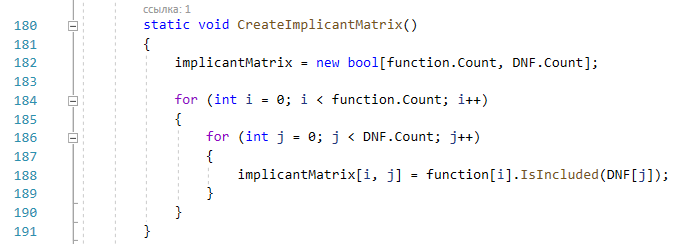
* + 1. Удаляются повторяющиеся элементы.

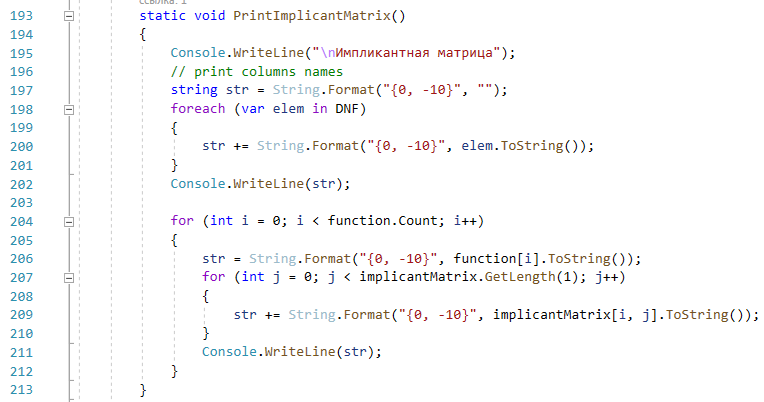


* + 1. Печать текущего состояния функции после прохода.



* 1. Создается и печатается импликантная матрица.





* 1. Рекурсивно находится минимизированная форма.

**Код программы**

using System;

using System.Collections.Generic;

using System.Linq;

namespace MinimizingBooleanFunctions

{

class Program

{

static int[] vector = new int[16];

static int[][] truthTable = new int[16][];

static List<Constituent> function;

static List<Constituent> DNF;

static List<Constituent> minimizedFunction;

static bool[,] implicantMatrix;

static void Main(string[] args)

{

// input

InputVector();

CreateTruthTable();

PrintTruthTable();

// DNF

CreateDNF();

PrintFunction(function);

Constituent[] t = new Constituent[function.Count];

function.CopyTo(t);

DNF = t.ToList();

// abbreviated DNF

CreateADNF();

// implicant matrix

CreateImplicantMatrix();

PrintImplicantMatrix();

CreateFunctionFromImplicantMatrix();

// output

Console.WriteLine();

PrintFunction(minimizedFunction);

}

static void InputVector()

{

Console.WriteLine("Введите вектор");

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

{

do

{

Console.WriteLine($"Введите элемент {i + 1}");

} while (!int.TryParse(Console.ReadLine(), out vector[i]) || !(vector[i] == 0 || vector[i] == 1));

}

}

static void CreateTruthTable()

{

for (int x1 = 0; x1 < 2; x1++)

{

for (int x2 = 0; x2 < 2; x2++)

{

for (int x3 = 0; x3 < 2; x3++)

{

for (int x4 = 0; x4 < 2; x4++)

{

int[] row = { x1, x2, x3, x4, vector[x1 \* 8 + x2 \* 4 + x3 \* 2 + x4] };

truthTable[x1 \* 8 + x2 \* 4 + x3 \* 2 + x4] = row;

}

}

}

}

}

static void PrintTruthTable()

{

Console.WriteLine("Таблица истиности");

Console.WriteLine(" x | y | z | w | F ");

Console.WriteLine("-------------------");

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

{

int[] row = truthTable[i];

Console.WriteLine($" {row[0]} | {row[1]} | {row[2]} | {row[3]} | {row[4]}");

}

}

// DNF = disjunctive normal form

static void CreateDNF()

{

function = new();

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

{

if (truthTable[i][4] == 1)

{

int[] row = truthTable[i];

function.Add(new Constituent(row[0], row[1], row[2], row[3]));

}

}

}

// ADNF = abbreviated disjunctive normal form

static void CreateADNF()

{

Console.WriteLine("====\nСкДНФ");

bool mergeHappened = true;

while (mergeHappened)

{

mergeHappened = false;

SortedSet<int> usedElements = new SortedSet<int>();

// merging

for (int i = function.Count - 1; i > 0; i--)

{

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

{

if (function[i].IsMerge(function[j]))

{

mergeHappened = true;

usedElements.Add(i);

usedElements.Add(j);

function.Add(function[i].Merge(function[j]));

}

}

}

// delete used constituents

while (usedElements.Count > 0)

{

function.RemoveAt(usedElements.Max);

usedElements.Remove(usedElements.Max);

}

// delete equal elements

int i1 = function.Count - 1;

int j1 = i1 - 1;

while (mergeHappened && i1 > 0)

{

if (i1 > 0 && function[i1] == function[j1])

{

function.RemoveAt(i1);

i1--;

}

j1--;

if (j1 < 0)

{

i1--;

j1 = i1 - 1;

}

}

if (mergeHappened)

PrintFunction(function);

}

function.Reverse();

}

static void PrintFunction(List<Constituent> function)

{

string res = "f(x, y, z, w) = ";

if (function.Count > 0)

{

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

{

res += function[i].ToString() + " \\/ ";

}

res = res.Remove(res.Length - 3);

}

else

{

res += "0";

}

Console.WriteLine(res);

}

static void CreateImplicantMatrix()

{

implicantMatrix = new bool[function.Count, DNF.Count];

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

{

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

{

implicantMatrix[i, j] = function[i].IsIncluded(DNF[j]);

}

}

}

static void PrintImplicantMatrix()

{

Console.WriteLine("\nИмпликантная матрица");

// print columns names

string str = String.Format("{0, -10}", "");

foreach (var elem in DNF)

{

str += String.Format("{0, -10}", elem.ToString());

}

Console.WriteLine(str);

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

{

str = String.Format("{0, -10}", function[i].ToString());

for (int j = 0; j < implicantMatrix.GetLength(1); j++)

{

str += String.Format("{0, -10}", implicantMatrix[i, j].ToString());

}

Console.WriteLine(str);

}

}

static void CreateFunctionFromImplicantMatrix()

{

minimizedFunction = new();

SortedSet<int> path = new();

// find path

path = FindMinNumberOfRows\_R(path, 0, 0);

// add functions from the path

while (path.Count > 0)

{

minimizedFunction.Add(function[path.Max]);

path.Remove(path.Max);

}

}

static SortedSet<int> FindMinNumberOfRows\_R(SortedSet<int> path, int i, int j)

{

// i = rows = function

// j = columns = DNF

// проверку столбцов скорее всего можно будет упразднить

// check for end of matrix

if (i < function.Count && j < DNF.Count)

{

// we are at the right way

if (implicantMatrix[i, j])

{

if (path.Count == 0)

path.Add(i);

return FindMinNumberOfRows\_R(path, i, j + 1);

}

// miss

else

{

// find another way

for (int k = 0; k < function.Count; k++)

{

bool IsFirst = true;

// if find another way

if (implicantMatrix[k, j])

{

// copy

var curPath = path;

// add new line

curPath.Add(k);

curPath = FindMinNumberOfRows\_R(curPath, k, j + 1);

// if we did not find min path yet

if (IsFirst)

{

path = curPath;

}

else

{

// if we find better path

if (curPath.Count() < path.Count())

{

path = curPath;

}

}

}

}

return path;

}

}

else

{

return path;

}

}

}

class Constituent : ICloneable

{

/\*

contain constituent in text format xyzw

0 - false

1 - true

2 - element not included into constituent

\*/

private string con = "2222";

public int Length;

public Constituent(string stringFormat)

{

con = stringFormat;

Length = (con[0] != '2' ? con[0] : 0) + (con[1] != '2' ? con[1] : 0) + (con[2] != '2' ? con[2] : 0) + (con[3] != '2' ? con[3] : 0);

}

public Constituent(int x, int y, int z, int w)

{

con = x.ToString() + y.ToString() + z.ToString() + w.ToString();

Length = x != 2 ? 1 : 0 + y != 2 ? 1 : 0 + z != 2 ? 1 : 0 + w != 2 ? 1 : 0;

}

public string GetInteriorFormat()

{

return con;

}

public bool IsMerge(Constituent constituent)

{

// only to equal number of elements

if (this.Length != constituent.Length) return false;

return this.con[0] != constituent.GetInteriorFormat()[0] && this.con[1] == constituent.GetInteriorFormat()[1] && this.con[2] == constituent.GetInteriorFormat()[2] && this.con[3] == constituent.GetInteriorFormat()[3]

|| this.con[0] == constituent.GetInteriorFormat()[0] && this.con[1] != constituent.GetInteriorFormat()[1] && this.con[2] == constituent.GetInteriorFormat()[2] && this.con[3] == constituent.GetInteriorFormat()[3]

|| this.con[0] == constituent.GetInteriorFormat()[0] && this.con[1] == constituent.GetInteriorFormat()[1] && this.con[2] != constituent.GetInteriorFormat()[2] && this.con[3] == constituent.GetInteriorFormat()[3]

|| this.con[0] == constituent.GetInteriorFormat()[0] && this.con[1] == constituent.GetInteriorFormat()[1] && this.con[2] == constituent.GetInteriorFormat()[2] && this.con[3] != constituent.GetInteriorFormat()[3];

}

public Constituent Merge(Constituent constituent)

{

if (!IsMerge(constituent)) throw new Exception("Merging can't be completed");

string con2 = constituent.GetInteriorFormat();

int[] op1 = { int.Parse(con[0].ToString()), int.Parse(con[1].ToString()), int.Parse(con[2].ToString()), int.Parse(con[3].ToString()) };

int[] op2 = { int.Parse(con2[0].ToString()), int.Parse(con2[1].ToString()), int.Parse(con2[2].ToString()), int.Parse(con2[3].ToString()) };

int[] res = new int[4];

for (int i = 0; i < 4; i++)

{

res[i] = op1[i] + op2[i] - 1;

}

// return to normal form

for (int i = 0; i < 4; i++)

{

if (res[i] == 0 || res[i] == 3)

{

res[i] = 2;

}

else if (res[i] == -1)

{

res[i] = 0;

}

}

return new Constituent(res[0], res[1], res[2], res[3]);

}

public bool IsIncluded(Constituent constituent)

{

if (this == constituent) return true;

if (this.Length > constituent.Length) return false;

string con2 = constituent.GetInteriorFormat();

for (int i = 0; i < 4; i++)

{

if (!(this.con[i] == con2[i] || this.con[i] == '2'))

{

return false;

}

}

return true;

}

public object Clone()

{

return this.MemberwiseClone();

}

public override string ToString()

{

if (con == "2222")

return "1";

string res = "";

if (con[0] != '2')

{

if (con[0] == '0')

res += '\'';

res += 'x';

}

if (con[1] != '2')

{

if (con[1] == '0')

res += '\'';

res += 'y';

}

if (con[2] != '2')

{

if (con[2] == '0')

res += '\'';

res += 'z';

}

if (con[3] != '2')

{

if (con[3] == '0')

res += '\'';

res += 'w';

}

return res;

}

public static bool operator ==(Constituent operand1, Constituent operand2)

{

return operand1.GetInteriorFormat() == operand2.GetInteriorFormat();

}

public static bool operator !=(Constituent operand1, Constituent operand2)

{

return operand1.GetInteriorFormat() != operand2.GetInteriorFormat();

}

}

}

**Примеры работы**

