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 и математическая логика»

Тема: Минимизация логических функций методом Квайна

Выполнил работу

студент группы ИВТ-22-2б

Мельников Г. В.

Проверила

Доцент кафедры ИТАС

Рустамханова Г. И.

Пермь, 2023

**Постановка задачи**

1. Программа доложена иметь возможность задавать вектор функции через клавиатуру

2. Программа должна выполнять следующие функции:

a. Создание и вывод на экран таблицы истинности

b. Получать совершенную дизъюнктивную нормальную форму (СДНФ)

c. Выполнять неполное склеивание

d. Создавать и выводить на экран импликантную матрицу

e. Проводить минимальное покрытие, и выводить минимальную форму.

**Методы решения задачи**

1. Main() – служит для получения функции с консоли и запуска метода CreateMDNF(functionVector)
2. CreateMDNF(string functionVector) – является «оболочкой» для всех используемых функций, также служит для вывода данных в консоль
3. GetTrueString(string functionVector) – получение значений переменных, при которых функция истина
4. ExecuteBonding(List<Conjunction> relevant) – выполняет склеивание
5. FindCommon(Conjunction conjuctionA, Conjunction conjuctionB) – проверяет, можно ли склеить коньюкторы
6. FillImplicateMatrix(

Dictionary<Conjunction, HashSet<Conjunction>> columnsTable,

Dictionary<Conjunction, HashSet<Conjunction>> rowsTable,

List<Conjunction> initialConjunctions,

HashSet<Conjunction> incapableAbsorptionConjunctions) – заполняет импликантную

матрицу

1. ExtractCore(

HashSet<Conjunction> core,

Dictionary<Conjunction, HashSet<Conjunction>> columnsTable,

Dictionary<Conjunction, HashSet<Conjunction>> rowsTable,

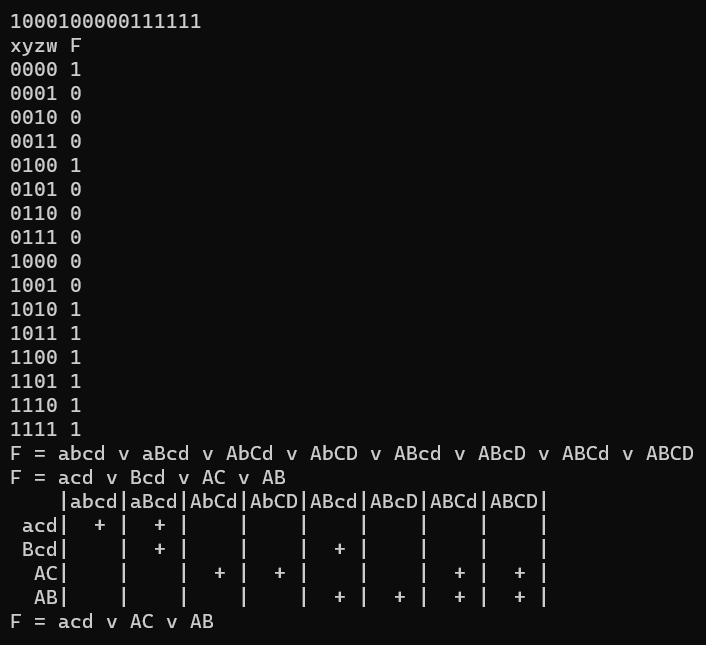
List<Conjunction> initialConjunctions) – убирает ядро из импликантной матрицы

1. HashSet<Conjunction> FindMinNotCoreConjunction(

Dictionary<Conjunction, HashSet<Conjunction>> rowsTable,

HashSet<Conjunction> columns) – рекурсивно ищет минимальное покрытие

**Результаты**

****

**Код программы**

**Program.cs**

namespace \_3

{

internal class Program

{

private static void Main(string[] args)

{

var functionVector = Console.ReadLine()!;

CreateMDNF(functionVector);

}

private static HashSet<Conjunction> CreateMDNF(string functionVector)

{

var initialConjunctions = GetTrueString(functionVector);

Console.WriteLine("xyzw F");

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

{

Console.WriteLine($"{Convert.ToString(i, 2).PadLeft(4, '0')} {functionVector[i]}");

}

Console.WriteLine($"F = {string.Join(" v ", initialConjunctions)}");

var incapableAbsorptionConjunctions = ExecuteBonding(initialConjunctions);

Console.WriteLine($"F = {string.Join(" v ", incapableAbsorptionConjunctions)}");

var columnsTable = initialConjunctions.ToDictionary(t => t, t => new HashSet<Conjunction>());

var rowsTable = incapableAbsorptionConjunctions.ToDictionary(t => t, t => new HashSet<Conjunction>());

FillImplicateMatrix(columnsTable, rowsTable, initialConjunctions, incapableAbsorptionConjunctions);

Console.WriteLine($" |{string.Join("|", initialConjunctions.Select(t => t.ToString().PadLeft(4)))}|");

foreach (var i in incapableAbsorptionConjunctions)

{

Console.WriteLine($"{i,4}|{string.Join("|", initialConjunctions.Select(t => columnsTable[t].Contains(i) ? " + " : " "))}|");

}

var core = new HashSet<Conjunction>();

ExtractCore(core, columnsTable, rowsTable, initialConjunctions);

var result = FindMinNotCoreConjunction(rowsTable, columnsTable.Keys.ToHashSet());

result.UnionWith(core);

Console.WriteLine($"F = {string.Join(" v ", result)}");

return result;

}

private static void ExtractCore(

HashSet<Conjunction> core,

Dictionary<Conjunction, HashSet<Conjunction>> columnsTable,

Dictionary<Conjunction, HashSet<Conjunction>> rowsTable,

List<Conjunction> initialConjunctions)

{

foreach (var conjunction in initialConjunctions)

{

if (columnsTable.ContainsKey(conjunction) && columnsTable[conjunction].Count == 1)

{

var coreConjunction = columnsTable[conjunction].First();

core.Add(coreConjunction);

var coreRow = rowsTable[coreConjunction];

rowsTable.Remove(coreConjunction);

foreach (var row in rowsTable)

{

row.Value.ExceptWith(coreRow);

}

foreach (var column in coreRow)

{

columnsTable.Remove(column);

}

}

}

}

private static void FillImplicateMatrix(

Dictionary<Conjunction, HashSet<Conjunction>> columnsTable,

Dictionary<Conjunction, HashSet<Conjunction>> rowsTable,

List<Conjunction> initialConjunctions,

HashSet<Conjunction> incapableAbsorptionConjunctions)

{

foreach (var conjunctionA in initialConjunctions)

{

foreach (var conjunctionB in incapableAbsorptionConjunctions)

{

if (conjunctionB.All(t => conjunctionA.Contains(t)))

{

columnsTable[conjunctionA].Add(conjunctionB);

rowsTable[conjunctionB].Add(conjunctionA);

}

}

}

}

private static List<Conjunction> GetTrueString(string functionVector)

{

var result = new List<Conjunction>();

for (var i = 0; i < functionVector.Length; i++)

{

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

{

result.Add(new(Convert.ToString(i, 2).PadLeft(4, '0')));

}

}

return result;

}

private static HashSet<Conjunction> ExecuteBonding(List<Conjunction> relevant)

{

HashSet<Conjunction> outsiders = new();

bool[] used = new bool[relevant.Count];

HashSet<Conjunction> newRelevant = new();

while (relevant.Count > 0 && relevant[0].Count > 0)

{

for (var i = 0; i < relevant.Count - 1; i++)

{

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

{

var common = FindCommon(relevant[i], relevant[j]);

if (common != null)

{

used[i] = used[j] = true;

newRelevant.Add(common);

}

}

}

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

{

if (used[i] == false)

{

outsiders.Add(relevant[i]);

}

}

relevant = newRelevant.ToList();

newRelevant.Clear();

used = new bool[relevant.Count];

}

outsiders.UnionWith(relevant);

return outsiders;

}

private static Conjunction? FindCommon(Conjunction conjuctionA, Conjunction conjuctionB)

{

Conjunction common = new(conjuctionA.Count - 1);

int indexCommon = 0;

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

{

if (char.ToLower(conjuctionA[i]) != char.ToLower(conjuctionB[i]))

{

return null;

}

if (conjuctionA[i] == conjuctionB[i])

{

common[indexCommon++] = conjuctionA[i];

}

}

return indexCommon == common.Count ? common : null;

}

private static HashSet<Conjunction> FindMinNotCoreConjunction(

Dictionary<Conjunction, HashSet<Conjunction>> rowsTable,

HashSet<Conjunction> columns)

{

var result = new HashSet<Conjunction>(columns);

FindMinNotCoreConjunction(result, new(), rowsTable, rowsTable.Keys.ToList(), columns.Count, 0);

return result;

}

private static void FindMinNotCoreConjunction(

HashSet<Conjunction> best,

HashSet<Conjunction> common,

Dictionary<Conjunction, HashSet<Conjunction>> rowsTable,

List<Conjunction> rows,

int columnsCount,

int rowIndex)

{

if (rowIndex == rowsTable.Count)

{

var isCovering = common.Select(t => rowsTable[t]).Distinct().Count() == columnsCount;

if (isCovering && (common.Count == best.Count && common.Sum(t => t.Count) < best.Sum(t => t.Count)

|| common.Count < best.Count))

{

best.Clear();

foreach (var conjunction in common)

{

best.Add(conjunction);

}

}

return;

}

FindMinNotCoreConjunction(best, common, rowsTable, rows, columnsCount, rowIndex + 1);

common.Add(rows[rowIndex]);

FindMinNotCoreConjunction(best, common, rowsTable, rows, columnsCount, rowIndex + 1);

common.Remove(rows[rowIndex]);

}

}

}

**Conjuction.cs**

using System.Collections;

namespace \_3

{

public class Conjunction : IEnumerable<char>//Особый тип данных для более удобной работы со словарями

{

private char[] items; // Или HashSet ?

public int Count => items.Length;

public char this[int index]

{

get => items[index];

set { items[index] = value; }

}

public Conjunction(int count) => items = new char[count];

public Conjunction(IEnumerable<char> items)

{

this.items = items.ToArray();

}

public Conjunction(string items)

{

this.items = new char[items.Length];

for (var i = 0; i < items.Length; i++)

{

var alpha = (char)('a' + i);

this[i] = items[i] == '1' ? char.ToUpper(alpha) : alpha;

}

}

public IEnumerator<char> GetEnumerator()

{

return ((IEnumerable<char>)items).GetEnumerator();

}

IEnumerator IEnumerable.GetEnumerator()

{

return items.GetEnumerator();

}

public override bool Equals(object? obj)

{

return obj is Conjunction conjunction && conjunction.SequenceEqual(this);

}

public override int GetHashCode()

{

return this.Aggregate(0, (t, a) => HashCode.Combine(t, a));

}

public override string ToString()

{

return string.Join("", items);

}

}

}