|  |
| --- |
| 23631-2 |
| Булевы функции |
| Отчет по лабораторной работе |

|  |
| --- |
| Решетняк Иван Витальевич  2019 |

# Постановка задачи:

Требуется реализовать программный пакет для работы с различными представлениями булевых функций. Необходимые представления: в виде таблицы истинности, СКНФ, СДНФ, РДНФ, полинома Жегалкина и карты Карно. Необходима возможность конверсии между формами, вывод форм в поток вывода. Также реализовать проверку свойств функции: самодвойственности, сохранения нуля, сохранения единицы, монотонности, линейности, симметричности.

# Решение задачи:

## boolean\_function

*boolean\_function –* вспомогательный класс, содержащий в себе все классы представлений и вспомогательные функции (вроде получений функций «истина»/«ложь», конверсии целого числа в логический вектор и тд.). Также *boolean\_function* хранит внутри себя одну инстанцию signature.

class boolean\_function {

private:

signature Signature;

public:

static std::vector<bool> intToBoolVec( int NoofArgs, int InArgs );

static int boolVecToInt( const std::vector<bool> &Vec );

static int binToGray( int Bin );

static void binToGray( std::vector<bool> &Binary );

class signature;

class truth\_table;

class ccnf;

class cdnf;

class rdnf;

class karnaugh\_map;

class zhegalkin\_poly;

std::vector<bool> one( void ) const;

std::vector<bool> zero( void ) const;

std::vector<bool> intToBoolVec( int InArgs );

std::vector<bool>::reference operator[]( const std::vector<bool> &Args );

boolean\_function( const signature &Signature );

bool isSelfDual( void );

bool isOneSaving( void );

bool isZeroSaving( void );

bool isMonotonous( void );

bool isLinear( void );

bool isSymmetric( void );

};

## Методы проверки свойств

Содержатся в классе *boolean\_function.*

bool boolean\_function::isSelfDual( void ) {

truth\_table Table(Signature);

signature NewSign = Signature;

for (int i = 0, Size = 1 << Signature.getNumArgs(); i < Size; i++)

NewSign[i] = !Table(intToBoolVec(~i));

return NewSign == Signature;

}

bool boolean\_function::isOneSaving( void ) {

truth\_table Table(Signature);

return Table(one());

}

bool boolean\_function::isZeroSaving( void ) {

truth\_table Table(Signature);

return !Table(zero());

}

bool boolean\_function::isMonotonous( void ) {

truth\_table Table(Signature);

bool Val = 0;

for (int i = 0, Size = 1 << Signature.getNumArgs(); i < Size; i++)

if (!Table(intToBoolVec(i)) && Val == 0)

return false;

return true;

}

bool boolean\_function::isLinear( void ) {

zhegalkin\_poly Poly(Signature);

for (int i = 0, Size = 1 << Signature.getNumArgs(); i < Size; i++)

if (Poly[intToBoolVec(i)] && (i != (1 << (int)log2(i))))

return false;

return true;

}

bool boolean\_function::isSymmetric( void ) {

truth\_table Table(Signature);

std::vector<int> ValuesAtNumberOfOnes(Signature.getNumArgs() + 1, -1);

for (int i = 0, Size = 1 << Signature.getNumArgs(); i < Size; i++) {

int NumberOfOnes = 0;

for (int Bit = 0, Size = Signature.getNumArgs(); Bit < Size; Bit++)

if ((i >> Bit) & 1)

NumberOfOnes++;

bool Val = Table(intToBoolVec(i));

if (ValuesAtNumberOfOnes[NumberOfOnes] == -1)

ValuesAtNumberOfOnes[NumberOfOnes] = (int)Val;

else if (ValuesAtNumberOfOnes[NumberOfOnes] != (int)Val)

return false;

}

return true;

}

## signature

Для простоты конверсии все формы представления способны конструироваться из и конвертироваться в *signature* – столбец истинности. *signature* по сути – вектор булевых значений.

class signature : public std::vector<bool> {

private:

int NoofArgs;

friend std::ostream & operator<<( std::ostream &, const signature & );

public:

signature( int NoofArgs, int InSign );

signature( const std::vector<bool> & );

int getNumArgs( void ) const;

};

## truth\_table

Класс таблицы истинности. Хранит в себе только *signature*, все запросы выполняются процедурно за O(1).

class truth\_table {

private:

signature Signature;

friend std::ostream & operator<<( std::ostream &, const truth\_table & );

public:

truth\_table( const signature &InSign );

bool operator()( const std::vector<bool> &Args ) const;

bool signBit( int Num ) const;

bool argsBit( int Num, int Col ) const;

int getNumArgs( void ) const;

operator signature( void ) const;

};

bool boolean\_function::truth\_table::operator()( const std::vector<bool> &Args ) const {

int ArgsBits = 0, BitNum = 0;

for (const auto &Bit : Args)

ArgsBits |= ((int)Bit << BitNum++);

return Signature[ArgsBits];

}

## ccnf

Класс совершенной конъюнктивной нормальной формы. Хранит в себе *signature* и вектор конъюнкций, каждый член которого – булев массив, указывающий, истинен или ложен аргумент в конъюнкции.

class ccnf {

private:

signature Signature;

friend std::ostream & operator<<( std::ostream &, const ccnf & );

using args = std::vector<bool>;

std::vector<args> Conjunctions;

public:

ccnf( const truth\_table &Table );

bool operator()( const std::vector<bool> &Args ) const;

operator signature( void ) const;

};

boolean\_function::ccnf::ccnf( const truth\_table &Table ) : Signature(Table) {

int NoofArgs = Table.getNumArgs();

for (int Row = 0, SignSize = 1 << NoofArgs; Row < SignSize; Row++)

if (!Table.signBit(Row)) {

Conjunctions.push\_back(args());

args &CBack = Conjunctions.back();

for (int Arg = 0, Size = Table.getNumArgs(); Arg < NoofArgs; Arg++)

CBack.push\_back(!Table.argsBit(Row, Arg));

}

}

bool boolean\_function::ccnf::operator()( const std::vector<bool> &ArgsIn ) const {

bool Res = 1;

int NoofArgs = Signature.getNumArgs();

for (const auto &Args : Conjunctions) {

bool ArgsRes = 0;

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

ArgsRes |= (Args[i] == ArgsIn[i]);

Res &= ArgsRes;

}

return Res;

}

## cdnf

Класс совершенной дизъюнктивной нормальной формы. Хранит в себе *signature* и вектор дизъюнкций, каждый член которого – булев массив, указывающий, истинен или ложен аргумент в дизъюнкции.

class cdnf {

private:

signature Signature;

using args = std::vector<bool>;

std::vector<args> Disjunctions;

friend std::ostream & operator<<( std::ostream &, const cdnf & );

public:

cdnf( const truth\_table &Table );

bool operator()( const std::vector<bool> &Args ) const;

operator signature( void ) const;

};

boolean\_function::cdnf::cdnf( const truth\_table &Table ) : Signature(Table) {

int NoofArgs = Table.getNumArgs();

for (int Row = 0, SignSize = 1 << NoofArgs; Row < SignSize; Row++)

if (Table.signBit(Row)) {

Disjunctions.push\_back(args());

args &CBack = Disjunctions.back();

for (int Arg = 0, Size = Table.getNumArgs(); Arg < NoofArgs; Arg++)

CBack.push\_back(Table.argsBit(Row, Arg));

}

}

bool boolean\_function::cdnf::operator()( const std::vector<bool> &ArgsIn ) const {

bool Res = 0;

int NoofArgs = Signature.getNumArgs();

for (const auto &Args : Disjunctions) {

bool ArgsRes = 1;

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

ArgsRes &= (Args[i] == ArgsIn[i]);

Res |= ArgsRes;

}

return Res;

}

## rdnf

Класс редуцированной дизъюнктивной нормальной формы. Хранит в себе *signature* и вектор дизъюнкций, каждый член которого – массив аргументов, входящих в дизъюнкцию, указывающий, истинен или ложен аргумент в дизъюнкции и его номер.

Редуцирование ДНФ происходит сравнением каждой дизъюнкции с каждой, в результате, если они различаются ровно в одном аргументе, они склеиваются в одну. Процесс повторяется пока производится хотя бы одно склеивание.

class rdnf {

private:

class operand {

public:

bool Value;

int Num;

};

signature Signature;

using args = std::vector<operand>;

std::vector<args> Disjunctions;

friend std::ostream & operator<<( std::ostream &, const rdnf & );

public:

rdnf( const truth\_table &Table );

bool operator()( const std::vector<bool> &Args ) const;

operator signature( void ) const;

};

boolean\_function::rdnf::rdnf( const truth\_table &Table ) : Signature(Table) {

std::vector<args> DisjTmp;

int NoofDisj = 1 << Table.getNumArgs();

DisjTmp.resize(NoofDisj);

int NoofArgs = Table.getNumArgs();

for (int Row = 0, SignSize = 1 << NoofArgs; Row < SignSize; Row++) {

int GrayRow = binToGray(Row);

if (Table.signBit(GrayRow)) {

for (int Arg = 0, Size = Table.getNumArgs(); Arg < NoofArgs; Arg++)

DisjTmp[Row].push\_back({Table.argsBit(GrayRow, Arg), Arg});

}

}

bool Changed = true;

while (Changed) {

Changed = false;

NoofDisj = DisjTmp.size();

for (int i = 0; i < NoofDisj - 1; i++)

if (DisjTmp[i].size() > 0) {

for (int j = i + 1; j < NoofDisj; j++) {

if (DisjTmp[j].size() == DisjTmp[i].size()) {

int Pos = 0, Diff = 0, Size = DisjTmp[i].size();

for (int k = 0; k < Size; k++)

if (DisjTmp[i][k].Value != DisjTmp[j][k].Value) {

Pos = k;

Diff++;

}

if (Diff == 1) {

DisjTmp[i].erase(DisjTmp[i].begin() + (int)Pos);

DisjTmp.erase(DisjTmp.begin() + j);

Changed = true;

break;

} else if (Diff == Size) {

DisjTmp.erase(DisjTmp.begin() + j);

Changed = true;

break;

}

}

}

if (Changed)

break;

}

}

for (const auto &it : DisjTmp)

if (it.size() > 0)

Disjunctions.push\_back(it);

}

bool boolean\_function::rdnf::operator()( const std::vector<bool> &ArgsIn ) const {

bool Res = 0;

for (const auto &Args : Disjunctions) {

bool ArgsRes = 1;

for (int i = 0, Size = Args.size(); i < Size; i++)

ArgsRes &= (Args[i].Value == ArgsIn[Args[i].Num]);

Res |= ArgsRes;

}

return Res;

}

## karnaugh\_map

Класс карты Карно. Хранит в себе *thruth\_table (*то есть *signature).* Все запросы выполняются процедурно за O(1). Отличается от таблицы истинности только оператором вывода.

class karnaugh\_map {

private:

truth\_table Table;

friend std::ostream & operator<<( std::ostream &, const karnaugh\_map & );

public:

operator signature( void ) const;

bool operator()( const std::vector<bool> &Args ) const;

karnaugh\_map( const truth\_table &Table );

};

std::ostream & operator<<( std::ostream &Stream, const boolean\_function::karnaugh\_map &Map ) {

int

CellSizeY = Map.Table.getNumArgs() >> 1,

CellSizeX = Map.Table.getNumArgs() - (Map.Table.getNumArgs() >> 1),

SizeX = 1 << CellSizeX,

SizeY = 1 << CellSizeY;

Stream << std::setw(CellSizeX + 1) << "";

for (int Col = 0; Col < SizeX; Col++) {

Stream << " ";

for (int Bit = 0; Bit < CellSizeX; Bit++)

Stream << ((boolean\_function::binToGray(Col) >> Bit) & 1);

}

Stream << std::endl;

for (int Row = 0; Row < SizeY; Row++) {

Stream << std::setw(1 + CellSizeX - CellSizeY) << " ";

for (int Bit = 0; Bit < CellSizeY; Bit++)

Stream << ((boolean\_function::binToGray(Row) >> Bit) & 1);

for (int Col = 0; Col < SizeX; Col++) {

auto

RowBits = boolean\_function::intToBoolVec(CellSizeY, boolean\_function::binToGray(Row)),

ColBits = boolean\_function::intToBoolVec(CellSizeX, boolean\_function::binToGray(Col));

RowBits.insert(RowBits.end(), ColBits.begin(), ColBits.end());

Stream << std::setw(CellSizeX + 1) << (int)Map(RowBits);

}

Stream << std::endl;

}

return Stream;

}

## zhegalkin\_poly

Класс полинома Жегалкина. Хранит в себе *signature* и вектор исключающих или, каждый член которого является булевым массивом, указывающий, входит ли аргумент в данное исключающее или. Строится полином методом треугольника.

class zhegalkin\_poly {

private:

signature Signature;

using args = std::vector<bool>;

std::vector<args> Xors;

friend std::ostream & operator<<( std::ostream &, const zhegalkin\_poly & );

public:

operator signature( void ) const;

bool operator()( const std::vector<bool> &Args ) const;

bool operator[]( const std::vector<bool> &Args ) const; // Check polynomial coefficient

zhegalkin\_poly( const truth\_table &Table );

};

boolean\_function::zhegalkin\_poly::zhegalkin\_poly( const truth\_table &Table ) : Signature(Table) {

int MaxSize = 1 << Table.getNumArgs();

std::vector<bool> Column = (signature)Table;

if (Column[0])

Xors.push\_back(boolean\_function::intToBoolVec(Table.getNumArgs(), 0));

for (int Size = MaxSize, ColNum = 1; Size > 1; Size--, ColNum++) {

for (int i = 0; i < Size - 1; i++)

Column[i] = Column[i] ^ Column[i + 1];

if (Column[0])

Xors.push\_back(boolean\_function::intToBoolVec(Table.getNumArgs(), ColNum));

}

}

bool boolean\_function::zhegalkin\_poly::operator()( const std::vector<bool> &Args ) const {

bool Res = false;

for (int XorNum = 0, Size = Xors.size(); XorNum < Size; XorNum++) {

bool AndRes = 1;

for (int ArgNum = 0, Size = Args.size(); ArgNum < Size; ArgNum++)

if (Xors[XorNum][ArgNum])

AndRes &= Args[ArgNum];

if (XorNum == 0)

Res = AndRes;

else

Res = Res ^ AndRes;

}

return Res;

}

# Обоснование решения

## Таблица истинности/карта Карно

Данные формы не нуждаются в хранении каких-либо дополнительных данных и выполняют обращения за O(1) – преобразуя входной набор аргументов в целое число и возвращая требуемый бит сигнатуры.

## Совершенные нормальные формы

Построения совершенных нормальных форм в программе происходит естественным путем: находятся 0/1 биты внутри сигнатуры и набор аргументов, соответствующий найденной строке, помещается в новую конъюнкцию/дизъюнкцию. Все это происходит за O(2^N \* N), где N – количество аргументов.

Обращение происходит за O(2^N \* N) – для каждой дизъюнкции/конъюнкции набор аргументов сравнивается со входным.

## Редуцированная дизъюнктивная нормальная форма

Построение РДНФ также происходит естественным путем – сначала строится СДНФ (O(2^N \* N)). Затем каждая дизъюнкция сравнивается с каждой (O(2^N \* 2^N \* N)) покомпонентно, в случае различия в одном аргументе происходит склейка, всего не больше 2^N склеек.

Конечная сложность – O(2^3N \* N), обращение происходит так же как и для СДНФ.

## Полином Жегалкина

Построение выполняется методом треугольника – выписывается столбец истинности, затем на каждом шаге выполняется исключающее или каждого его элемента со следующим, уменьшая размер на 1 (O(2^N \* 2^N)), этот процесс составляет треугольник. Затем каждая 1 верхней строки треугольника создает исключающее или, набор аргументов в котором соответствует бинарному коду номера столбца. Сложность – O(2^2N).

Сложность обращения – O(2^N \* N).

# Вычислительный эксперимент

Для функции, заданной сигнатурой , где n – количество аргументов {8, 9, 10}, были вычислены все формы представления и проверены свойства функции.

# Выводы

Таблица истинности – оптимальный способ представления и работы с булевой функцией в программе, с постоянным временем построения и обращения.