СПбПУ Петра Великого

Отчёт по дисциплине «Дискретная математика»

Лабораторная №2

Илья Козлов
23631/2

Contents

Постановка задачи

Реализовать, на C++, класс булевых функций (далее – б.ф.), представимых таблицей истинности в виде вектора, СКНФ, СДНФ, сокращённой ДНФ, КФ, ДФ, полиномом Жегалкина и картой Карно.

Класс должен содержать:

- данные: вектор таблицы истинности, матрицу/вектор сокращённой ДНФ, матрицы/вектора КФ и ДФ, вектор коэффициентов полинома Жегалкина и карту Карно.
- методы конверсий таблицы истинности в: полином Жегалкина, карту Карно, СДНФ, СКНФ и обратно + метод перевода из СДНФ в сокращенную ДНФ (метод Квайна);
- методы вычисления значения в точке, записи в таблицу истинности
- методы потокового ввода/вывода из/в ‘cin’ | ‘cout’ | ‘*.txt’
- методы проверки свойств: самодвойственности, сохранения нуля, единицы, монотонности, линейности, симметричности.

При реализации класса необходимо использовать побитовые булевы операции `&`, `|`, `^` и сдвиги `<<`, `>>`.

Ограничения

Накладывается следующее ограничение на количество переменных: $n < 32$. Это связано с тем, что, во-первых, мы работаем с типом uint и нам удобнее работать с числами, в которых число битов меньше 32, а во-вторых, с огромным количеством памяти и времени, которое потребуется для хранения и инициализации при $n \geq 32$. Уже при $n = 32, memory = \frac{2^{32}}{8*2^{20}} = 512$мб, что недопустимо при работе с оперативной памятью.

Решение задачи

Прототип класса

```
class bool_func;
```

Перечисление типов

```
using uint = unsigned int;          // 32 bits
using luint = long unsigned int;    // 64 bits
using uvec = std::vector<uint>;     // package of 32bits numbers
using umatr = std::vector<uvec>;    // matrix of packages of 32bits numbers for tKarn
using byte = unsigned char;         // 8 bits

enum rep_type
{
  tTVT, tPDNF, tPCNF, tZheP, tKarn, tRDNF, tCF, tDF
}; // end of 'rep_type' enum
```

Пояснение: tTVT – таблица истинности, tPDNF – совершенная дизъюнктивная форма, tPCNF – совершенная конъюктивная форма, tZheP – полином Жегалкина, tKarn – карта Карно, tRDNF – сокращенная дизъюнктивная форма, tCF – конъюктивная форма, tDF – дизъюнктивная форма.

Хранимая информация

```
private:
byte n; // number of variables
uint mPDNF, mPCNF, mRDNF, mCF, mDF;
uvec TVT, PDNF, PCNF, ZheP, RDNF, CF, DF;
umatr Karn;
```

Пояснение: “m*” – дополнителная информация о том или ином представлении, то есть, mPDNF, например, показывает количество конъюнктов. Uvec TVT, … – хранимые вектора представлений, umatr Karn – матрица карты Карно.

Конструкторы

```
public:
bool_func() = default;
bool_func(byte n, const uvec & source, rep_type tp = tTVT, luint m = 0);
bool_func(byte n, const umatr & KarnMatr);
```

Пояснение: первый конструктор - коструктор «по умолчанию», используется для создания пустого класса б.ф.; второй – используется для инициализации класса б.ф. от любого из типов представления, кроме карты Карно, так как там требуется матрица, поэтому для инициализации от типа tKarn используется отдельный – третий – конструктор.

Методы конверсий

```
private:
void fromTVT(rep_type tp);
void toTVT(rep_type tp = tTVT);
void Quine();  // PDNF -> RDNF
```

Пояснение: для всех типов, кроме tRDNF реализованы две функции для конверсий – fromTVT(tp), переводящая из таблицы истинности в заданный тип tp и обратная ей – toTVT(tp), создающая таблицу истинности, исользуя текущий тип tp, таким образом конверсии из всех типов во все происходит через таблицу истинности. РДНФ создается отдельно, с помощью метода Квайна из СДНФ.

Методы вычисления/установки значения

```
public:
bool value(uint x, rep_type tp = tTVT); // Getting value

private:
void setValue(uint x, bool val);
```

Пояснение: value – публичный метод для вычисления значения б.ф. в точке x с помощью типа представления tp, а setValue – приватный метод установки значения в точке x в вектор истинности, используемый только внутри класса при конверсиях.

Методы ввода/вывода

```
private:
friend std::istream & operator>>(std::istream & in, bool_func & bf);
friend std::ostream & operator<<(std::ostream & out, bool_func & bf);

public:
void read(std::istream & in, rep_type tp = tTVT, luint m = 0);
void write(std::ostream & out, rep_type tp = tTVT);
```

Пояснение: операторы используются только для вывода/ввода из/в таблицу истинности, для всех остальных же типов используются read и write.

Методы проверки свойств

```
public:
bool saveZero();
bool saveOne();
bool isSelfDual();
bool isLinear();
bool isMonotone();
```

Вспомогательные методы

```
private:
void convers(rep_type typeFrom, std::vector<rep_type> typesTo, bool needQuine);
void pack(uvec inv, rep_type tp, uvec ent = uvec());   // RDNF, PDNF, PCNF
uint getSet(uint i, uvec bf, luint m);  // RDNF, PDNF, PCNF, CF, DF
void print(std::ostream & out, int w, uvec bf);
bool consistPair(uvec & entV, uvec & invV, uint ent, uint inv);
static uint Grey(uint ind);      // From index to Grey code
static uint Grey2Int(uint g);    // From Grey code to index
```

Пояснение: convers() – функция, облегчающая визуально конструктор класса б.ф., где производятся все необходимые переводы; pack() – метод для упаковки СДНФ, СКНФ и РДНФ в 32битные вектора; getSet() – метод для получения i-того конъюнкта/дизъюнкта для СДНФ/СКНФ или i-той маски инверсии/вхождения для РДНФ, КФ или ДФ; print() – функция, облегчающая вывод представлений б.ф. consistPair() - вспомогательный метод для метода Квайна, позволяющий отследить присутствие пары масок вхождение–инверсия в промежуточных массивах пар таких масок; Grey() – метод получения кода Грея из индекса; Grey2Int() – обратный метод – из кода Грея получается индекс.

Реализация класса

Конструкторы

```
bool_func() = default;
```

Пояснение: все переменные класса заполняются значениями по умолчанию, создаются вектора нулевого размера.

```
bool_func::bool_func(byte n, const uvec & source, rep_type tp, luint m)
{
  this->n = n;

  TVT = uvec((((1 << n) - 1) >> 5) + 1);
  PDNF.clear();
  PCNF.clear();
  RDNF.clear();
  ZheP = uvec((((1 << n) - 1) >> 5) + 1);
  CF.clear();
  DF.clear();
  Karn = umatr(1 << (n >> 1));
  for (auto i = Karn.begin(); i != Karn.end(); i++)
    *i = uvec(((1 << ((n >> 1) + (n & 1))) >> 5) + 1);

  switch (tp)
  {
  case tTVT:
    TVT = source;
    convers(tp, std::vector<rep_type>({tPDNF, tPCNF, tZheP, tKarn}), true);
    break;
  case tPDNF:
    PDNF = source;
    mPDNF = m;
    convers(tp, std::vector<rep_type>({tPCNF, tZheP, tKarn}), false);
    Quine(); // PDNF -> RDNF
    break;
  case tPCNF:
    PCNF = source;
    mPCNF = m;
    convers(tp, std::vector<rep_type>({tPDNF, tZheP, tKarn}), true);
    break;
  case tZheP:
    ZheP = source;
    convers(tp, std::vector<rep_type>({tPDNF, tPCNF, tKarn}), true);
    break;
  case tRDNF:
    RDNF = source;
    mRDNF = m;
    convers(tp, std::vector<rep_type>({tPDNF, tPCNF, tZheP, tKarn}), false);
    break;
  case tDF:
    DF = source;
    mDF = m;
    convers(tp, std::vector<rep_type>({tPDNF, tPCNF, tZheP, tKarn}), true);
    break;
  case tCF:
    CF = source;
    mCF = m;
    convers(tp, std::vector<rep_type>({tPDNF, tPCNF, tZheP, tKarn}), true);
    break;
  }
} // End of constructor
```

Пояснение: в первых строчках происходит заблаговременное выделение памяти / очистка векторов (для последующего заполнения с помощью прямого доступа / с помощью операции push_back). Далее происходит копирование пришедшей информации в то или иное представление и вызывается вспомогательная функция convers, которая производит все переводы.

Пояснение формул: (((1 << n) - 1) >> 5) + 1 соответствует формуле $\frac{2^n-1}{32}+1$, где 2^n – количество кортежей, то есть количество необходимых битов для таблицы истинности, так как наша таблица истинности – вектор uint, надо разделить количество битов без одного (так как при $n \geq 5$ количество кратно 32 и получится неверный размер массива) на 32, чтобы узнать количество uint, в которое поместится таблица истинности целиком, после чего необходимо прибавить единицу, чтобы получить итоговый размер необходимого массива. То же самое и с полиномом Жегалкина.

На карту Карно сначала выделяется память на строки – их 2^{ndiv2}, а затем выделяется память в каждой строке на столбцы – их $2^{ndiv2+nmod2}$, чему соответствуют вышеуказанные формулы.

```
bool_func::bool_func(byte n, const umatr & KarnMatr)
{
  this->n = n;
  TVT = uvec((((1 << n) - 1) >> 5) + 1);
  PDNF.clear();
  PCNF.clear();
  RDNF.clear();
  ZheP = uvec((((1 << n) - 1) >> 5) + 1);
  Karn = KarnMatr;
  convers(tKarn, std::vector<rep_type>({tPDNF, tPCNF, tZheP}), true);
} // End of constructor
```

Пояснение: отдельно выделенный конструктор для карты Карно. Происходят те же самые выделения памяти и присвоение присланной информации, что и выше.

Методы конверсий

```
void bool_func::fromTVT(rep_type tp)
{
  uint bits;
  switch (tp)
  {
  case tPDNF:
    for (int i = 0; i < 1 << n; i++)
      if (value(i))
        PDNF.push_back(i);
    mPDNF = PDNF.size();
    pack(PDNF, tp);
    break;
  case tPCNF:
    for (int i = 0; i < 1 << n; i++)
      if (!value(i))
```

```
        PCNF.push_back(i);
    mPCNF = PCNF.size();
    pack(PCNF, tp);
    break;
  case tKarn:
    uint x, i, j, s, bMask, wInd;
    bits = 1u << ((n >> 1) + (n & 1));
    bits = bits > 0x20 ? 0x20 : bits;
    s = Karn[0].size();
    for (x = 0; i = Grey2Int(x >> n - (n >> 1)), j = Grey2Int(x & (1 << (n - (n >> 1))) -
1), wInd = s - 1 - (j >> 5), bMask = 1 << bits - 1 - (j & 0x1f), x < 1u << n; x++)
      Karn[i][wInd] ^= ((Karn[i][wInd] & bMask) == bMask) != value(x) ? bMask : 0;
    break;
  case tZheP:
    uint D = 1u << n, wZind, bZind, bZmask, wTind, bTind, bTmask;
    bits = n > 5 ? 0x1f : (1u << n) - 1;
    for (uint d = 0; wZind = d >> 5, bZind = d & 0x1f, bZmask = 1u << bits - bZind, d <
D; d++)
      for (uint x = 0; wTind = TVT.size() - 1 - (x >> 5), bTind = x & 0x1f, bTmask = 1u
<< bTind, x <= d; x++)
        ZheP[wZind] ^= (TVT[wTind] & bTmask) == bTmask && x == (x & d) ? bZmask : 0;
    break;
  }
} // End of 'bool_func::fromTVT' function
```

Пояснение:

PDNF: так как нам уже известна таблица инстинности, нам нужно просто получить кортежи, в которых функция равна единице, то есть проходим по всему пространству, и если функция дает true в текущей точке, добавляем эту точку в вектор СДНФ, тогда вспомогательная mPDNF (количество конъюнктов) равна размеру вектора. Далее происходит упаковка вектора для более компактного хранения.

PCNF: то же самое, с точностью до того факта, что в случае СКНФ учитываются нули функции, а не единицы.

Karn: bits – количество битов в слове в строчке ($2^{ndiv2+nmod2}$ при $n \leq 5$ и 32 при больших), s – количество выделенных uint для хранения одной строчки таблицы Карно. За один цикл происходит проход по всему булеву пространству, для каждой точки высчитываются координаты в карте Карно с помощью вспомогательной функции Grey2Int: i – индекс кода Грея первой половины текущего кортежа, что будет номером строчки в матрице Karn, j – индекс второй половины, но для получения индекса слова в строчке необходимо посчитать wInd, который получается по вышеуказанной формуле, так как мы храним информацию слева направо (иначе было бы просто j >> 5). Далее в полученном слове с помощью маски и операции XOR устанавливается значение функции в данной точке x. Маска получается сдвигом единицы на разность количества битов в слове без единицы и остатка от деления индекса j на 32. Побитовое И слова с маской даст маску только в случае присутствия в слове единицы на том же месте, что и в маске, что означает, что значение функции в этой точке тоже единица (true), то же

самое и при нуле, то есть при несовпадении значений операций value() и сравнения маски с результатом побитового И, бит в текущем слове карты Карно необходимо сменить на обратный, что достигается путем применения побитого исключающего или с маской, и с нулем в обратном случае, когда бит не подлежит изменению.

ZheP: bits – опять же количество битов в одном слове вектора. В первом цикле происходит проход по всему булеву пространству для d, считается индекс слова в векторе коэффициентов полинома Жегалкина wZind, маска bZmask (опять же bits – bZind так как хранение слева направо), для соответствующего бита в выбранном слове. Во втором цикле пробегаем уже не по всему пространству, а только до d, так как нам нужны $x \preccurlyeq d$, так же считаются индексы слова уже в таблице истинности и маска для соответствующего бита в слове. Далее происходит суммирование по модулю два для соответствующего бита в слове вектора коэффициентов с помощью посчитанной маски.

$$Zh_{[f]} =\!\!\!+_{2} {}_{d\in\{0,1\}^n} C_d \& x^d => C_d =\!\!\!+_{2} f\big((bool^n)(\delta)\big), \{0,1\}^n \ni \delta \preccurlyeq d$$

Первое равенство в условии триплексной операции – проверка на истинность функции в текущей точке x, а второе – $x \preccurlyeq d$, при выполнении этих двух условий, текущий бит, соответствующий коэффициенту при x, увеличивается (по модулю два) на единицу с помощью маски, в обратном случае соответственно бит не меняется. Таким образом устанавливаются все коэффициенты для всего булева пространства.

```
void bool_func::toTVT(rep_type tp)
{
  switch (tp)
  {
  case tPDNF:
    for (int i = 0; i < 1 << n; i++)
      setValue(i, false);
    for (uint x, i = 0; i < mPDNF; i++)
      x = getSet(i, PDNF, mPDNF), setValue(x, true);
    break;
  case tPCNF:
    for (int i = 0; i < 1 << n; i++)
      setValue(i, true);
    for (uint x, i = 0; i < mPCNF; i++)
      x = getSet(i, PCNF, mPCNF), setValue(x, false);
    break;
  case tZheP:
  case tRDNF:
  case tDF:
  case tCF:
  case tKarn:
    for (int i = 0; i < 1 << n; i++)
      setValue(i, value(i, tp));
    break;
  }
} // End of 'bool_func::toTVT' function
```

Пояснение: для СДНФ и СКНФ сначала происходит заполнение всей таблицы значением по умолчанию (0 при СДНФ и 1 при СКНФ), далее для всех x, принадлежащих СДНФ (получаются с помощью вспомогательной функции getSet, «достающей» кортеж из упакованного вектора) устанавливается значение функции в true, а для СКНФ, наоборот, false. Для всех остальных представлений пробегаемся по всему булеву пространству, находим значение в точке с помощью того или иного представления методом value и устанавливаем это значение в таблицу истинности с помощью метода setValue.

```
void bool_func::Quine()
{
  uvec ent, inv, entF, invF, entUsed, invUsed;
  uint max = (1u << n) - 1, v;

  for (uint i = 0; i < 1u << n; i++)
    if (value(i))
      inv.push_back(i);

  for (uint i = 0; i < inv.size(); i++)
    ent.push_back(max);

  for (uint j = 0; j < inv.size(); j++)
  {
    uint size = ent.size();
    for (uint k = j + 1; k < size; k++)
      if ((v = ent[j] == ent[k] ? inv[j] ^ inv[k] : 0) && !(v & (v - 1)))
        ent.push_back((v ^ max) & ent[j]), inv.push_back(inv[j]),
entUsed.push_back(ent[k]), entUsed.push_back(ent[j]), invUsed.push_back(inv[k]),
invUsed.push_back(inv[j]);

    if (size == ent.size() && !consistPair(entUsed, invUsed, ent[j], inv[j]) &&
!consistPair(entF, invF, ent[j], inv[j]))
      entF.push_back(ent[j]), invF.push_back(inv[j]);
  }
  mRDNF = invF.size();
  pack(invF, tRDNF, entF);
} // End of 'bool_func::Quine' function
```

Пояснение: выше представлен метод Квайна перевода из СДНФ в РДНФ. Для реализации метода используются 6 дополнительных векторов: ent – вектор вхождений переменных в соответствующем кортеже, inv – инверсий, entF – финальный вектор вхождений, invF – инверсий, entUsed – вектор использованных масок вхождения и invUsed – использованных масок инверсий. max – максимально возможное число в 32битном слове при текущем количестве переменных n. Для начала я забиваю массив инверсий точками булева пространства, в которых функция принимает значение true (то есть просто кортежи СДНФ, которые сами по себе и являются масками инверсий), а массив вхождений – максимальным числом, то есть при трех переменных, массив забьется числом 111_2, которое значит, что все три бита входят в кортеж. Таким образом я имею пару маска инверсии – маска вхождения для каждого кортежа из СДНФ. Далее я иду по этой паре векторов

(можно идти по любому, так как равны размеры) и произвожу «склейку» сверху вниз. Разрабатывая алгоритм, я пришел к выводу, что склейка производится только при равенстве масок вхождения, что и проверяется при склейке во втором цикле. Если маски равны, то с помощью побитого исключающего или масок инверсий мы можем получить единицы в тех местах, где маски отличаются, то есть, если мы получим после XOR число, равное степени двойки, то есть число с единственной единицей в двоичной записи, то наши кортежи отличаются всего на один бит, а значит, их можно склеить. Проверка на степень двойки осуществляется по формуле $v \;\&\&\; !\left(v \;\&\; (v-1)\right)$. Итак, мы получили число, которое указывает в каком месте отличаются наши кортежи, в таком случае обратное число будет показывать новую маску вхождения переменных в полученный склейкой кортеж. Таким образом я добавляю в массив всех масок вхождений число, обратное полученному в результате XOR, не забывая при этом и учесть предыдущее значение маски вхождений для этого кортежа, ибо некоторые переменные могли и не учитываться при операции исключающего или, и первую маску инверсий (можно добавить и inv[k], так как маски одинаковые в битах, соответствующих единицам в маске вхождения). Так же я здесь же добавляю обе маски вхождений и инверсий в векторы использованных масок, то есть произвожу операцию, подобную «вычеркиванию». До второго цикла я фиксирую размер текущего массива масок вхождений/инверсий. После цикла производится проверка на изменения этого размера, то есть если мы за весь цикл не изменили размер вектора масок, то, получается джитый кортеж мы ни с кем не склеили, тогда, казалось бы, его можно добавить в финальный вектор необходимых инверсий и вхождений, но следует не забывать, что вычеркнутые маски мы не добавляем, поэтому производится проверка на присутствие текущей пары масок вхождения–инверсии в векторах использованных масок и финальных масок, дабы не добавлять лишнего. То есть, если не произошло ни одной склейки (размер текущего рабочего массива не изменился) и пара не содержится в векторах, она добавляется в финальные массивы масок. Таким образом, при достижении конца вектора масок инверсий будут построены финальные вектора масок вхождений и инверсий для редуцированной ДНФ, причем invF.size() даст количество конъюнктов. Далее происходит упаковка пар масок в один вектор uint.

Методы вычисления/установки значения

```
bool bool_func::value(uint x, rep_type tp)
{
  uint sum = 0, bits, xorMask = (1u << n) - 1;

  switch (tp)
  {
  case tTVT:
  case tPDNF:
```

```
  case tPCNF:
    return TVT[TVT.size() - 1 - (x >> 5)] >> (x & 0x1f) & 1;
  case tZheP:
    bits = n > 5 ? 0x1f : (1 << n) - 1;
    uint set, wZind, bZind, bZmask;
    for (set = 0; wZind = set >> 5, bZind = set & 0x1f, bZmask = 1u << bits - bZind, set
< 1u << n; set++)
      sum ^= (ZheP[wZind] & bZmask) == bZmask && (set & x) == set;
    return sum;
  case tRDNF:
    for (uint i = 0; i < mRDNF; i++)
      if (((getSet(mRDNF + i, RDNF, mRDNF * 2) ^ xorMask ^ x) & getSet(i, RDNF, mRDNF *
2)) == getSet(i, RDNF, mRDNF * 2))
        return true;
    return false;
  case tDF:
    for (uint i = 0; i < mDF; i++)
      if (((getSet(mDF + i, DF, mDF * 2) ^ xorMask ^ x) & getSet(i, DF, mDF * 2)) ==
getSet(i, DF, mDF * 2))
        return true;
    return false;
  case tCF:
    for (uint i = 0; i < mCF; i++)
      if ((getSet(mCF + i, CF, mCF * 2) ^ x) & getSet(i, CF, mCF * 2) == 0)
        return false;
    return true;
  case tKarn:
    uint s = Karn[0].size();
    bits = 1 << ((n >> 1) + (n & 1));
    uint i = Grey2Int(x >> n - (n >> 1)), j = Grey2Int(x & (1 << n - (n >> 1)) - 1), wInd
= s - 1 - (j >> 5), bMask = bits - 1 - (j & 0x1f);
    return Karn[i][wInd] >> bMask & 1;
  }

  return false;
} // End of 'bool_func::value' function
```

Пояснение:

TVT, PDNF, PCNF: здесь берется значение из таблицы истинности, сначала считается индекс слова в таблице, затем слово сдвигается на необходимое количество битов таким образом, чтобы нужный нам бит стоял с правого края, таким образом побитовое И с единицей даст значение в точке.

ZheP: значение получается в одном цикле, пробегаясь по всем точкам булева пространства, высчитывая индексы слова в векторе коэффициентов и маски соответствующего точке бита в слове, путем суммирования по модулю два коэффициентов, соответствующих текущей точке set, для которой верно: коэффициент при этой точке ненулевой и $set \preccurlyeq x$.

RDNF: для получения значения в точке необходимо подействовать маской вхождения на x: сначала получаем маску методом getSet, в полученной маске 0 стоят на местах, где переменную следует обратить, а обращение происходит с помощью битового XOR с единицей, для чего и сделано сначала обращение маски, чтобы на места тех нулей встали единицы и уже после производится побитовое исключающее ИЛИ с x, далее с помощью

побитового И с маской вхождения проверяется, стоят ли единицы на одинкаковых местах, если да, то в наш конъюнкт даст единицу, а так как РДНФ – это дизъюнктивная форма, то ноль уже никак не может получиться, поэтому уже на данном этапе можно вернуть true как значение функции в точке. Иначе, если ни одного такого конъюнкта не встретилось, возвращаем false.

DF: такая же ситуация

CF: симметричная ситуация, здесь на местах инверсий стоят уже единицы, поэтому маска вхождения сразу действует на x. Если на всех местах, где маска вхождения имеет единицу, точка имеет нули, то дизъюнкт даст ноль и так как КФ – конъюнктивная форма, единица уже никак не сможет получиться, а значит уже можно вернуть false как значение функции в точке. Иначе, если не втретится ни одного такого дизъюнкта, то есть все они будут истинны, то и их конъюнкция даст true.

Karn: значение получается примерно так же, как из таблицы истинности: находятся индексы строки и слова в строке с помощью методов Grey2Int и соответствующих сдвигов, далее слово смещается вправо так, чтобы нужный бит оказался справа и сравнивается с единицей.

```
void bool_func::setValue(uint x, bool val)
{
  uint wTind = TVT.size() - 1 - (x >> 5), bTind = x & 0x1f, bTmask = 1 << bTind;
  TVT[wTind] ^= (TVT[wTind] & bTmask) == bTmask != val ? bTmask : 0;
} // End of 'bool_func::setValue' function
```

Пояснение: как было показано ранее, считаются индексы слова в векторе истинности и маска для нужного бита и с помощью операции XOR устанавливается соответствующий бит в нужном слове.

Методы ввода/вывода

```
std::istream & operator>>(std::istream & in, bool_func & bf);
```

Пояснение: в виду отсутствия особой математики, не буду приводить код оператора, следует заметить только, что количество переменных вычисляется как `n = (byte)log2(str.size() << 2);` То есть считается длина текстовой строки и умножается на 4 (так как одна литера шестнадцатеричного кода представляет 4 двоичного), при этом получается 2^n значений таблицы истинности, а значит логарифм этого числа и будет количеством переменных.

```
std::ostream & operator<<(std::ostream & out, bool_func & bf);
```

Пояснение: по тем же причинам стоит указать лишь, что w – это ширина одного шестнадцатеричного слова при данном количестве переменных.

```
void bool_func::read(std::istream & in, rep_type tp, luint m)
```

Пояснение: содержит довольно примитивные расчеты количества переменных, далее происходит чтение строки по словам из 8 символов (те же 32 бита) и вызывается уже разобранный конструктор от того или иного представления.

```
void bool_func::write(std::ostream & out, rep_type tp)
```

Пояснение: так же довольно примитивные расчеты необходимой ширины слова и вызов вспомогательной функции print, позволяющей печатать каждое слово из вектора с выбранной шириной (то есть слово дополнится нулями слева при нехватке).

Методы проверки свойств

`bool saveZero(), bool saveOne()` – очевидно

```
bool bool_func::isSelfDual()
{
  uint max = (1u << n) - 1;
  for (uint x = 0; x < 1u << n; x++)
    if (value(x) != ~value(x ^ max))
      return false;
  return true;
} // End of 'bool_func::isSelfDual' function
```

Пояснение: цикл по всему булеву пространству, проверяющий на равенство значения функции в точке и обратного значения от инвертированной точки, если они хоть в какой то точке не совпали – функция не самодвойственная, а значит, возвращаем нуль, если же такого кортежа не встретилось, то функция самодвойственна.

```
bool bool_func::isLinear()
{
  uint bits = n > 5 ? 0x1f : (1 << n) - 1;
  uint x, wZind, bZind, bZmask;
  for (x = 0; wZind = x >> 5, bZind = x & 0x1f, bZmask = 1u << bits - bZind, x < 1u << n;
x++)
    if ((ZheP[wZind] & bZmask) == bZmask && !(x & (x - 1)))     // deg(ZheP) > 1
      return false;
  return true;
} // End of 'bool_func::isLinear' function
```

Пояснение: для проверки на линейность достаточно проверить степень полинома Жегалкина – если она больше единицы, то функция не линейна. Опять же пробегаем по всем точкам пространства, строим индекс и маску и проверяем, если коэффициент при кортеже равен единице (первое равенство) и точка не является нулем или степенью двойки, то бишь ненулевой коэффициент стоит при кортеже, в котором больше одной единицы, что значит, что полином Жегалкина как минимум второй степени, а значит функция уже не может быть линейной, поэтому можно не доходить до конца цикла и уже здесь вернуть false. Если же при всех таких кортежах нулевые

коэффициенты, то степень полином не больше единицы, а значит функция действительно линейная.

```
bool bool_func::isMonotone()
{
  for (uint x = 0; x < 1u << n; x++)
    for (uint y = x; y < 1u << n; y++)
      if ((x | y) == y && value(x) > value(y))  // ![(x <= y) => (f(x) <= f(y))] == (x <=
y) & f(x) > f(y)
        return false;
  return true;
} // End of 'bool_func::isMonotone' function
```

Пояснение: $\nearrow(f) \equiv \forall x, y \in bool^n \left((x \preccurlyeq y) => \left(f(x) \leq f(y)\right)\right)$, где $(x \preccurlyeq y) \equiv ((x \mid y) == y) \in bool$, далее путем отрицания импликации была получена вышеописанная формула, из истинности которой вытекает немонотонность функции. Первый цикл идет по всему пространству, второй же начиная с x.

Вспомогательные методы

Из важных, с точки зрения математики, можно отметить:

```
void bool_func::pack(uvec inv, rep_type tp, uvec ent)
{
  int size = inv.size();
  uint bits = n * size, wInd;
  bits = bits > 0x20 ? 0x20 : bits;
  int bInd;

  switch (tp)
  {
  case tPDNF:
    PDNF = uvec((size * n >> 5) + 1);

    for (uint i = 0; wInd = i * n >> 5, bInd = bits - (i * n & 0x1f) - n, i < size; i++)
      if (bInd >= 0)
        PDNF[wInd] ^= inv[i] << bInd;
      else
        PDNF[wInd] ^= inv[i] >> (-bInd), PDNF[wInd + 1] ^= (inv[i] & ((1 << (-bInd)) -
1)) << bits + bInd;
    break;
  case tPCNF:
    PCNF = uvec((size * n >> 5) + 1);

    for (uint i = 0; wInd = i * n >> 5, bInd = bits - (i * n & 0x1f) - n, i < size; i++)
      if (bInd >= 0)
        PCNF[wInd] ^= inv[i] << bInd;
      else
        PCNF[wInd] ^= inv[i] >> (-bInd), PCNF[wInd + 1] ^= (inv[i] & ((1 << (-bInd)) -
1)) << bits + bInd;
    break;
  case tRDNF:
    RDNF = uvec((size * 2 * n >> 5) + 1);
    bits = bits * 2 > 0x20 ? 0x20 : bits * 2;
    uint ind;
    uvec *ptr;
```

```
    for (uint i = 0; ind = i >= size ? i - size : i, ptr = i >= size ? &inv : &ent, wInd
= i * n >> 5, bInd = bits - (i * n & 0x1f) - n, i < size * 2; i++)
      if (bInd >= 0)
        RDNF[wInd] ^= (*ptr)[ind] << bInd;
      else
        RDNF[wInd] ^= (*ptr)[ind] >> (-bInd), RDNF[wInd + 1] ^= ((*ptr)[ind] & ((1 << (-
bInd)) - 1)) << bits + bInd;
    break;
  }
} // End of 'bool_func::pack' function
```

Пояснение: метод занимается упаковкой из вектора[ов] инверсий [и вхождений] в единственный вектор представления для более компактного хранения.

bits – опять же количество битов в слове

PDNF: сначала происходит выделение памяти под вектор размера size (количество конъюнктов) * n / 32 (так как храним по 32 бита). Далее проходим по всем конъюнктам, подсчитывая индекс слова в массиве и маску bInd для макси инверсии. Построения достаточно очевидны, однако, следует заметить, что в bInd дополнительно отнимается количество переменных n, так как мы настраиваем в слове не бит, как раньше, а целую маску размера n битов. Может произойти ситуация, когда маска уже полностью не помещается в оставшееся место в слове, например, при 10 переменных, на четвертой маске (i == 3), маска bInd будет равна 32 - (3 * 10 % 32) – 10 = -8, что значит, что первые два бита (10 + (-8) = 2) все еще помещяются в текущее слово, а оставшиеся 8 надо поместить на первое место в следующее слово. То есть при положительной bInd происходит просто настройка n битов слова PDNF[wInd], иначе первые помещающиеся биты маски inv[i] попадают в текущее слово, оставшиеся же помещются в начало следующего слова (PDNF[wInd + 1]). Первые биты получаются просто смещением вправо на модуль bInd, а последние, то есть идущие во второе слово, считаются с помощью побитового И маски с числом, в котором |bInd| единиц. Маска для второго слова получается очень легко – достаточно прибавить к количеству битов в слове наш отрицательный bInd, то есть по примеру выше получится: 32 – 8 = 24, а значит оставшиеся 8 битов маски инверсий сместятся на 24 влево и настроят те самые 8 битов в начале второго слова.

PCNF: симметричная ситуация, конъюнкты заменяются на дизъюнкты

RDNF: ситуация такая же с точностью до того момента, что в RDNF я храню в одном векторе сначала маски вхождений, потом маски инверсий, этим объясняется умножение на 2 количества конъюнктов в выделении памяти и количество итераций в цикле. Я решил, в целях экономии, сделать не два цикла, где сначала вектор будет заполняться масками взождений и потом

масками инверсий, а только один, для этого мне понадобился указатель uvec *ptr на вектор, с которым я работаю в конкретный момент, то есть пока i меньше size, ptr указывает на вектор вхождения, а после – на вектор инверсий, поэтому индекс ind маски в конкретном векторе тоже считается по другому начиная с i == size. Теперь, зная, какое слово мы будем менять и какой маской (вхождения или инверсии), выполняем уже вышеописанный алгоритм упаковки.

```
uint bool_func::getSet(uint i, uvec bf, luint m)
{
  uint bits = n * m, wInd = i * n >> 5, max = (1 << n) - 1;
  bits = bits > 0x20 ? 0x20 : bits;
  int bInd = bits - (i * n & 0x1f) - n;
  return bInd >= 0 ? bf[wInd] >> bInd & max : bf[wInd] << (-bInd) & max | bf[wInd + 1] >>
bits + bInd;
} // End of 'bool_func::getSet' function
```

Пояснение: как уже сказано выше, метод «достает» итый кортеж из плотно упакованных представлений PDNF, PCNF, RDNF, CF и DF. Преобразования довольно очевидные, так как являются обратными по отношению к упаковке. То есть если мы получили положительный индекс bInd, то все хорошо и мы просто сдвигаем слово на bInd вправо и сравнивая с максимальным числом с n единицами, получаем необходимый кортеж. При отрицательном же результате берем первые (n + bInd) битов, для чего сдвигаем слово вправо на модуль bInd и сравниваем с тем же max и с помощью побитовой операции ИЛИ «склеиваем» с оставшимися |bInd| битами из начала второго слова.

```
uint bool_func::Grey(uint ind)
{
  return ind ^ ind >> 1;
} // End of 'bool_func::Grey' function
```

Пояснение: для получения кода Грея из иднекса достаточно применить операцию XOR к индексу и тому же индексу, поделенному на два, то есть сдвинутому на один бит вправо.

```
uint bool_func::Grey2Int(uint g)
{
  uint ind, m, b; // Most Significant Bit xOR-Accumulator

  for (m = 1U << 31; ~(g & m) & m; m >>= 1); // Most Significant Bit of G
  for (ind = b = 0; m; b ^= g & m, ind ^= b, b >>= 1, m >>= 1);

  return ind;
} // End of 'bool_func::Grey2Int' function
```

Пояснение: метод, обратный вышеуказанному – получение индекса из кода Грея. В первом for мы находим наибольший значимый бит кода g, то есть идем от $m = 2^{31}$, с единицей у левого края до 1 у правого края. Условие цикла будет выполняться до тех пор, пока на соответствующем m месте в g тоже не появится единица. Как только она повится, цикл прервется и мы

будем иметь представление в виде m, указывающее на старший бит кода g. Далее индекс получается просто суммированием по модулю два с числами, составляющими сдвинутый вправо код g. Как только все такие числа просуммируются, цикл закончится и мы получим индекс ind кода g.

Эксперимент

Ниже представляю результаты работы программы для варианта N = 6.

Таблица истинности задается по формуле: $(N + 192)^{2^{n-3}} => 198^{2^{n-3}}$

Были проведены исследования для трёх n: 8, 9 и 10.

Вывод представлений

8 переменных

TVT = 198^32

TVT:

00119F904F16FD5A59A64EDE557E4671C425545E616A183708B0028100000000

PDNF:

2027293435373B40414244454B4C51535556585D5E61626364666A6C6E7072757A7E7F8084
8586898A8E919293949596989A9C9EA1A2A3A4A6A7A9AAABAEB1B2B5B7B8BBBCBE
C1C3C4C6C8CACBCCCDCECFD1D2D4D8D9DADBDEE4E7E8E9EAEBECEFF0F400

PCNF:

000102030405060708090A0B0C0D0E0F10111213141516171819 1A1B1C1D1E1F2122232425
26282A2B2C2D2E2F303132333638393A3C3D3E3F43464748494A4D4E4F50525457595A5B
5C5F60656768696B6D6F717374767778797B7C7D81828387888B8C8D8F9097999B9D9FA0
A5A8ACADAFB0B3B4B6B9BABDBFC0C2C5C7C9D0D3D5D6D7DCDDDFE0E1E2E3E5E
6EDEEF1F2F3F5F6F7F8F9FAFBFCFDFEFF000000

RDNF:

FF7FFEBF7FFDDFDF7FFDF7DFF77FDF7FFDFE7FFD7FFEFBDFFDBFFEDFEFEFBFEFF
DF7F7F7FB6F7DFAEB6FF3E7F5EBEEEDDDBDAF9FE7D7B7DB9FCFDBF3B7F5F9BDDB
DDF5FAF6F3BDBEF5CFF9EDDDDBFAEEDEDBFCFCFC57AB202934353B4041424B5155
5556585E5E61626A70707E808991919292A1A2A7ABC1C3D1E7F027354041416262646A84
84848484848686868A8A8E919292949898989CA1A2A2A2A9AAC4C4C8C8C8C8CACACA
CBCCD8E8448A0000

ZheP:

00000000FEABF5B1938DD1361662963BF693882D723665122D837864DCEAD99F

Karn:

0000
0000
0704
8402
7998
9238
6A51
D384
69FD
511F

8100
05AF
7D1E
5695
7B99
8B1A

9 переменных

TVT = 198^64

TVT:

00000136949F14A547025B86343005710BDF37F73C0D2762D881C926037336DE36078B31A
F308C76A1D24107E8A36FD99138BC9981664501000000000000000000

PDNF:

20120944E28948AA562C17CC0633219CD46B361B4DE733A1D4F07C3FA0106844321D108
94522D1A8E482452A974DA753C9F5028544A8572C568B65BAE17ABF60B0988C56332996
CF6A355B0D96D36DBADF70391CAE874BADDEF078BC9F2FA7E3F6030281C120D0784C
2A190D8844629158B462332191496512B974BE61379DCF27D3FA0D168D48A4D2A9B50A
954EB55BAE576C161B1592CB66B3DA2D36AB65B6E171B95CEE976BBDE2F37BC06130
B86C462B2995CD6733BA1D169F51A9D5EB35DB1DC6E775BEE070B8BC7E5733A1D1E9
74FA9D7ED773BFE1F1793CBE80000000

PCNF:

000040403020140C0704024140B060341C0F0804424130A0542C170C064341B0E0743C1F10
0844423120944C27140A4542B160B45C2F180C464331A0D46C371C0E4743B1E0F47C3F20
9088644229188E49259309A4F2814CA8572C968B65C2E978C26232998D069371C0E2723B1
DCF27A3D9F4FC8141215188F4924D28964C265349C51A914AA653AA554AB562B55EB05
92CD6AB95D2ED78BE6030D8EC864B319ACE68345A4D36B35DB8DE70B89C6E673BA9
D8ED773CDE8F57B3DDF0FB7F3FE01058442E1D0F8944E2F1A8DC723B1E8FC824724934
9E532A964B6633299CD26B369C4E6753DA050A8744A3D2E994EA7D46A955AB55EB159
AF57EC76BB75BEEB7ABE5F6FD7FC0E0B0787C4E2F198DC763F2191C964F2D97CC6673
79ECFE8347A4D2E9B55ADD7EC365B3DA6D76DB7DDEF37BBFE0F0F89C6E472797CDE
773FABD6EC767B7DDEF783C7E6F3FA7D5EBF67B7DDEFF87C7E5F3FA7D7EDF7FC7E7
F5FBFE7F7FDFF0

RDNF:

FFFDFBFEFFDFEFF6FFEFDFFBF7FEFDFFDEFFFF7F7DEFFFF7EDFF7FF7FEFEFFEFBF
BFF7FBF7FBFBFFEFDF7FFDFEFEBFDFFBFFEFFBF7FB7FF7DFFEFFF7F7DFFEFEFDFE
FFEFEFDFBFF7DFFFAFFF7DFEFFFBDFF7FFAFFBFDFF7FF7FDFEFF7F7DFFEFBFDFFB
FDFDDFADF3FDEDF7F3F3BDDDE7FBDBF9F7D77BEF5FBFC7EB7F7CBFD7FD76EF7EB
EEFBBBDCFEDDDEEF5FBDFDDDE6FFB9DFEBEDFF3F3DB7F79F6F9FFD7B6FB6FDDC
FFF5F6CFDFB773F9FCFFD7EEB7D7FBDDDB7F76FBF6EF5FAEFD7FADEFF97EF77F3E
BFAF76BFEDB7F3FBBDBE7FE74FE5E6F4E9C804024128A25229154AA582C17CBE63339
A8DA6F399D0F0783E2010683432190E89452351C8E48245229E5028144AE58AD16CC1613
51B4DF703A1D2E9783C1E2F27E40605028241A192B9BCDE773CA3536B95CC061735A8D
4EB371BAE17178FCCE874FAE4825128C06031990C864321ACD88040201088444221108A4
522D2A954AA5D369D4EA7540A1542A184C46231594CA65B2DB0D96CB92030180C0A09

0482426151088442292190C8A512B974BA8B48A452AA150A954AA552A9D82C364B259A
E171B96131594CA68751C0E17428360B058000000

ZheP:
0000000000000000FF309FAE1FF9648DE36D4DD57A088C137311B7B1FFF2E1EE7A16D91
F19994472D6E9C88BC4ADBAF96F4168C3F27B0677640652D2112BD350

Karn:
00000000
00000000
A5EC85C4
8019845A
5B2CF6C0
8327530B
D01186B2
AD5F2E63
8B0913C0
5497D802
53010000
968815AF
4A4B330D
DFCBE0BF
52352D21
7DCAC0D3

10 переменных

TVT = 198^128

TVT:
00000000000178CC478E5301BAC8D5943C62C9AAFF4733909C0593E42287B780D54A5E4
1CB8A6CFDE1B1BB4FA722DC8034F20A4F94CE5A39344CA9A1253F229170E72028690B
5D2A10CB50556180028E461AF4B53224957605F477239CC0108898689A08B12E22B575702
D9811658A0100000000000000000000000000000000

PDNF:
2008922C8F24092254962609C28CA429CA82A8AB2B4B42D4B62E0BA2F0BD2F8C0308C4
314C7324CD344D234CD5360DC374DF38CE93ACEC3BCF33D4F63ECFC3FD0341D0C459
174691B4711F48121495284A52A4B12D4B9324D1354D9374E13A5054251145519485294C5
3D52555595715D581625916559D6A5B16D5B96F5C5735D1795E97E6058260D87625976619
D679A0689A4699AC6B9B06C5B36D9B76F1C170DC5721CA72DCC739D0745D3761DB77
5DE78DE57B5F07C5F27D5F67DDFC7F5FE8020481E09836108461284E148561886A1D8822
589E288AE2D8BE328CE368EA3C8F64090E449164992E4C93A5194A5395A5796A5C97E60
9866298E669A66B9C6749D6769DE7A9F27DA1E8AA2E8CA3A8FA4695A6299A6A9DA7E
A0A86A2A8EA6AA2A9AAEACAB6AFAC2B4AD6B7AE2BDAFABFB02C2B0EC4B16C6B
1ECAB2ECDB3AD1B4ED7B62D9B6EDEB7EE0B9AE9BAAEBBB2EEBC6F3BDAF8BEAF
CBFAFFC1F08C270AC330DC3F10C4712C5F19C7722C9726C9F28CA72CCBF30CCB3ACE
F3CCFF44D1F48D274CD3750D4752D5B58D675AD6F5CD775ED7F61D8F65D9F68DAF6E
DBF71DD776DEB7BDF37DE0B84E1F88E2B8CE3B8FE4F96E5F99E6F9CE779FE83A8EA7
ACEBBB1ECBB3EDFB8EE7BAEFBC2F0FC6F1FCBF33CDF3BD00000

PCNF:

00001008030100501807020090280B0300D0380F040110481305015058170601906 81B0701D0781F080210882309025098270A0290A82B0B02D0B82F0C0310C8330D0350D8370E0390E83B0F03D0F83F10041108431104511847120491284B1304D1384F1405114853150551585716059168 5B1705D1785F18061188631906519867 1A0691A86B1B06D1B86F1C0711C8731D0751D8771E0791E87B1F07D1F87F2048220C842148621C882288C2348E24493250972649A26C9D2789F280A1288A5298A92B0AE2BCB02C4B22CCB72E4BB2FCC130CC6320CA32CCC338CF340D4358D7364DA36CDE380E1388E4394E639CE83A8ED3B8F03C4F23D0F73E0F93E8FD3F900405024110541908425 0A42D0D4390F4411144913451154611947 51E489234912649D2B4BD304C5334E53B4F13D4F93F5014351D4952D4D5395054553551565 5D585695B5795F58563599685A56B5C1725D5765DD785ED7C5F57F60184615866218A62D8C6358E63D906459264D94655966659A66D9C67DA168DA569DA86A5AA6ADAD6BDB26D1B56E1B96E9BB6F5BE6FDC0709C4719C7725CD73DD2751D5759D7765DA771DF781E1789E4799E77A1E97A9EB7B1EE7BDF37D1F87E5FA7EDFF8060280E0581A0882A0B8320E83E1685E1986E1C87A1F8862288E2489A298AA2C8BA308C6348D6378E2398EE3E8FE4190A4691E4892A4D93E50952559625996E5D97A64996679A26A9B26D9BA6F9C2729CE789E67B9FA7FA0281A0A83A1285A1A88A268DA4292A4E94A5A97A6E9CA7AA4A96A7AAAAEAC6B2ACEB6AE6BAAEEBCB06C8B26CCB3ED0B4AD4B56D6B6ADCB76E1B8AE3B92E5B9EE8BB6EFBC2F2BD2F5BDEF9BEEFDC0301C0B03C1305C1B0BC3B13C5315C5B18C6B1BC731EC7F20C8723C932ACAF2DCBB31CCF34CD736CDF38CE73DCFB40D0742D0F45D1B4AD2F4ED3F53D5355D5F60D8B64D9B69DAB6CDB770DCB73DD377DE379DFB7FE0381E0F85E1B89E2F8DE4391E4B94E5798E6B9EE87A2E8FA4E97A6E9FAAEAFADEBFB0ED3B5EDBBBEF3BDEFFC0F07C4F17C8F27CAF3FD1F4BD3F53D5F5BD7F63D9F6BDBF73DDF7BDFF83E1F8BE3F93E5F9BE7FA3E9FABEBFB3EDFBBEFFC3F1FCBF3FD3F5FDBF7FE3F9FEBFBFF3FDFFBFF0000

RDNF:

FBFBFFF7BFBFFFDFDFFBEFFDFFEFEFBFFFE7FFEFBFFFBF7FFBFFBFEFF6FFBFEFF7FFF7FFBDFFEFBFDFFEFFEFBFFEFEFFF7EFF7DFFFDFDFDFFFDFEFEFFF7F7DFDFFDFFF7FFBDF7FFBF7FFBFFBDFFFBFEFF7FEFFFBEFFDDFFF7FDFFF7FBFEFFDFDFFEFF7FBEFFFDDFEFFFEF7FDFFEFFEFBFEFFFBBFEFFBFFFDFEFEFFFBEFFBFFBDFFEFFDFBFFDFEFFFBDFFFBEFFDFFEFF7FB7EDDFE7DDFF6FBB77EBF6FF9FEDDF7FB5EF6FDEFFD9FEFD5F7AFFF9AFDFD7EDFE7F7E77DEFD7F5DF77FE77F7EB7BF7BEBF9F77DBFF7B5FFDBF3EDDBFFE9FEFE7ED7EFBBEBFBEEF77D5FDF7FF37E6FF7BF5FDEF5DF7D7F7DDEDFB7BD7F7EDFEFDBF57F7777DFEBFF1FEEF5DFFE7F57F7BE7F5FBEFBEDF9DF7FE5FDF7BE7FAFBBE7EDF9FEFBAFFCFBBFFCFEAFEFD6FDEF77BCFEBFF7BBEF5FB7B7EBFDDF7DE7F5F9FF77CFFF5BF6F79FF7BCFF7DDBEF7AFFBEAFF7BD7FD7DDF6FDFAF7F775FBBF7DD7EDFFEAFBBDF7BF7ADFDDEDFB7FFCFD6FDBBEF7DBF7DBEFFCFDAF7BFAF7AFFFAEFBBBBEFDBE7EFDB7EFEAFFCFBBFF3EDFD7EFB9FBEFDDBF79FFF3F6F9EF7FD7FABFAFBBBFF9EF7EDFD7BEFE7F6FCFEBF9EF7FF3D7BDFDEF6FCFFD7DDFCFDEF9FEBFB7F9F5F77E7F3FF7B7DE7F3FEFBDDEF7B7DEFBDEDF7EDDFBEDFBF6FD7F3F7BDEFE7EBFCFE7FE7DDEF7F3F6FF9FB96FBEBBAD6FABFABF82008022489224902409224896 28CA4290AA2F8C2314C9324D1348DC374DD38CF53D9174691A46D20481254C934509425114553D52555595756058164 5C5715C5715F981609826099D6759E681A0699AC70DC8729CE741D0775DE7C1F17C9F57D9FC7F2328DA4C93A5F98A7A9F28AA8AA6AB2B0ADEBEB370DC8B22C9B2CCBF3BD436BE0B82E138CE5B99E73AEEC7B2EE3BA22C8B2549526098260982609C28CA32A0A82ACAD2D0B42D0B52D8B82F0C0300C4314C5314D1344D337CE33A4EB3B1034310C4310C4591748121495254A1284A1284A1

2A4A92C4B12C4B1324D1364E941509445114451548521485314C5655C591655956559D6A5
B16E60597681A46B1B06C5B36D9C170DC3721CC741E5802008020781E098260D83610842
1084210846118461184A1284A1284A128561886A1D8762088225896278A22B8AE2D8B6328
EA3C8F2409024390E43912449264B932519465194A5294A5395A5695E5A972609866198663
99A699C6749D2759F287A1E8AA2A8AA2A8BA2E8CA328EA3A8FA3E91A4695A6298A66
9DA769FA82A0A82A0A86A8AA2A8AA2A9AA6ADAD6BEB02C2B0AC3B0EC3B12C6B1
ACAB2ACEB46D1B4EDEB9AEABB2F8C1F07C1F07C2308C230CC4310C4311C6725C9F2
8CCB3AD3350D4352D6B5CD8761D8F87E238CE3393E4F99EA107B0B08C2309D23580000
0000

ZheP:

0000000000000000000000000000000FFBA3580E07D238B382D809136B190C9101F1231BF
2086F97FB24BF5177153EF9986506AE9B2ABD3702C467CC2AEBDC2F1A173EA0AB0A9
E3150DE20E65D3EEAC0F35178960D8A4BCF9FFA371F0017BBDE02DF88080934E316582
A41409A35B0B5566C5989EB1EADAA30799149D82C82C

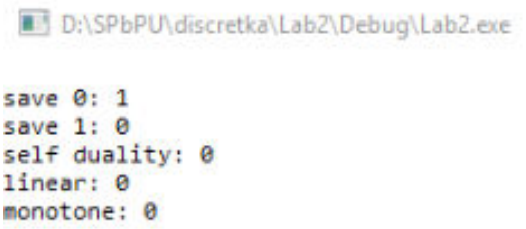
Karn:

00000000
00000000
00000000
00000000
9742874E
20A62554
254D9BD0
80268159
74028A20
99900137
22400B7B
629DAA07
5BA94348
97F85886
C2DB90F8
24803530
04EB422B
1EA33509
66353352
05C3FF1B
3CD40001
00000000
8093D82E
15B96734
04BCD642
F8E78EE1
889E9D16
BF5CEC26
A396153E
F80613F2

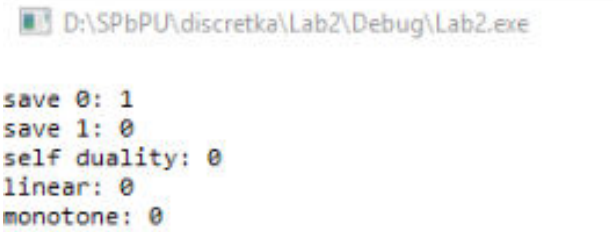
8665131C
854292CF

Проверка свойств

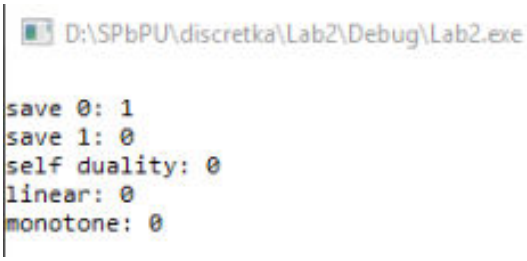
8 переменных



9 переменных



10 переменных



Оценка времени и памяти

Память в байтах

n	TVT	ZheP	RDNF	Karn	PDNF	PCNF
8	48	48	192	276	116	176
9	80	80	444	280	240	372
10	144	144	1000	536	488	828

Время инициализации (ms)

n	TVT→ZheP	TVT→PCNF	TVT→Karn	TVT→PDNF	Quine	∑
8	21	1	1	1	42	69
9	84	2	1	1	162	245
10	322	3	2	2	615	990

Вывод

Был реализован класс булевых функций bool_func, имеющий следующие представления: таблица истинности, СДНФ, СКНФ, РДНФ, полином Жегалкина и карта Карно, а так же КФ и ДФ, в случае если булева функция инзначально задается через них. В классе реализован «полный граф» переводов, то есть коневерторы из любого представления в любое, что не является на деле полным графом, так как большинство переводов происходят через перевод в/из таблицы истинности, кроме метода Квайна, который реализует переход от СДНФ к РДНФ.

Все представления в классе хранятся не по указателю, а с помощью STL контейнера std::vector<unsigned int>, что есть динамический массив. Такое хранение позволяет не «заморачиваться» с конструктором по умолчанию и деструктором, уменьшает лишнюю работу с памятью и удобно обозначает свой размер методом size(), то есть это минус одна лишняя переменная. В остальном же, чтение и запись происходят так же, как с указателями.

Засчет наличия enum представлений, класс получился достаточно компактным и плотным.

Из вычислительного эксперимента видна корректность построения представлений. Следует заметить, что класс не оптимально использовать для функций с большим количеством переменных: видно, что уже при 10 переменных класс инициализируется порядка секунды, что уже достаточно заметно невооруженным взглядом.

Меньше всего памяти, очевидно занимают таблица истинности и полином Жегалкина, так как для этих представлений необходимо всего 2^n битов, а больше всего RDNF, что логично, ибо для нее используется упакованный вектор, хранящий как маски вхождений, так и маски инверсий кортежей.

По времени очевидно, что метод Квайна работает дольше всех, однако, следует заметить, что он работает примерно в 2 раза дольше конвертора из TVT в полином Жегалкина, сложность которого составляет порядка $O(2^{2n})$, из чего можно сделать предположение, что метод Квайна имеет сложность порядка $O(2^{2n+1})$. Остальные представления инициализируются гораздно быстрее, по очевидным причинам, порядка 1ms.

Источники

- Ф.А.Н. «Дискретная математика для программистов», 3-е издание
- Стахов С.В. «BoolFuncLabTaskExpl_v4.docx»
- http://en.cppreference.com/w/