

## ГЕНЕРИРОВАНИЕ БУЛЕАНА ДЛЯ СИНТЕЗА КВАНТОВОГО ПРОЦЕССОРА

Предлагаются кубитные (квантовые) структуры данных и вычислительных процессов для существенного повышения быстродействия при решении задач дискретной оптимизации. Анализируются работы по генерированию булеана, как существенного компонента при решении комбинаторных задач. Описываются аппаратно-ориентированные модели параллельного (за один цикл) вычисления булеана (множества всех подмножеств) на универсуме из  $n$  примитивов для решения задач покрытия, минимизации булевых функций, сжатия данных, синтеза и анализа цифровых систем за счет реализации процессорной структуры в форме диаграммы Хассе.

### 1. Актуальность

Задача генерирования булеана как множества всех подмножеств часто встречается на практике [1-20]. В связи с этим представляют интерес порождающие алгоритмы, к которым предъявляются следующие требования: 1. Полнота – порождение алгоритмом всего множества элементов. 2. Непротиворечивость – порождение алгоритмом только тех элементов, которые относятся к множеству. 3. Неповторимость – порождение алгоритмом элементов только один раз. Последнее свойство вызвано возможностью ухудшения временных характеристик и усложнением проверки условий остановки алгоритма. Задача порождения множества всех подмножеств может быть реализована с использованием кодов принадлежности [3] и алгоритмов генерации  $k$ -элементных подмножеств (сочетаний) [10, 12, 14-16]. 4. Практическая направленность. Квантовые вычисления в последние годы становятся интересными для анализа кибернетического пространства, создания новых Интернет-технологий, благодаря их некоторой альтернативности существующим моделям вычислительных процессов. Кроме того, рыночная привлекательность квантовых или кубитных моделей основывается на высоком параллелизме решения практически всех задач дискретной оптимизации, факторизации, минимизации булевых функций, эффективном сжатии, компактном представлении и телепортации данных, отказоустойчивом проектировании за счет существенного повышения аппаратных затрат. Но такая плата в настоящее время вполне допустима, поскольку существуют проблемы заполнения площадей кремникового кристалла, который содержит до 1 миллиарда вентиляей.

### 2. Линейный код и код Грея

*Код принадлежности.* Пусть дано множество  $X = \{x_1, x_2, \dots, x_i, \dots, x_n\}$ . Код принадлежности для подмножества  $\sigma \subseteq X$  определяется как набор  $K = \langle k_1, k_2, \dots, k_i, \dots, k_n \rangle$ , где

каждый компонент  $k_i$  есть  $k_i = \begin{cases} 1, & x_i \in \sigma; \\ 0, & x_i \notin \sigma. \end{cases}$  Очевидно, что количество всех подмножеств

равно  $2^n$ .

*Линейный код.* Алгоритм порождает подмножества следующим образом: на 0-м ярусе – код пустого множества; на 1-м ярусе – коды всех подмножеств, состоящих из одного элемента; на 2-м ярусе – коды всех двухэлементных подмножеств; далее – аналогично, на  $k$ -м ярусе – коды всех  $k$ -элементных подмножеств; на  $n$ -м ярусе – код  $n$ -элементного подмножества, т.е. самого множества как несобственного подмножества самого себя. Ярусы связаны дугами с номерами, которые соответствуют порождающим операциям  $N = 2^0, 2^1, 2^2, \dots, 2^n$ , выполняемым по следующим правилам.

1. Для  $N$  имеется  $K_{lev+1} = K_{lev} \vee N_2$ , где  $K_{lev}$  – порождающий код принадлежности к ярусу с номером  $lev$  (level – уровень),  $K_{lev+1}$  – порожденный код,  $N$  – натуральное число,  $N_2$  – двоичный код числа  $N$ .

2. Если  $K_{lev}$  содержит 1, совпадающую с соответствующим разрядом  $N_2$ , порождение прекращается.

Алгоритм с такими правилами последовательно продвигается по ярусам, сначала порождая подмножества из одного элемента, затем – из двух, трех и далее аналогично (рис. 1).

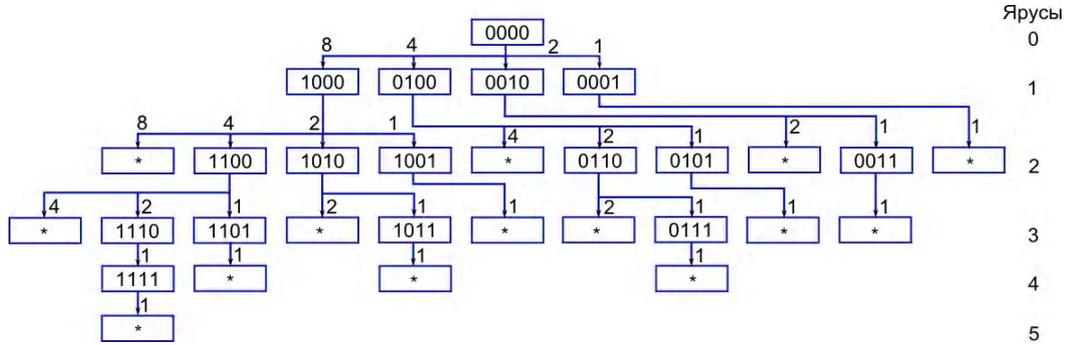


Рис. 1. Алгоритм порождения булеана

*Коды Грея.* Как и для линейного кода, каждый ярус содержит коды принадлежности булеану. Количество элементов каждого кода определяется номером яруса. В частности, ярус 3 содержит  $2^3=8$  кодов принадлежности всех подмножеств, построенных из трех элементов (рис. 2).

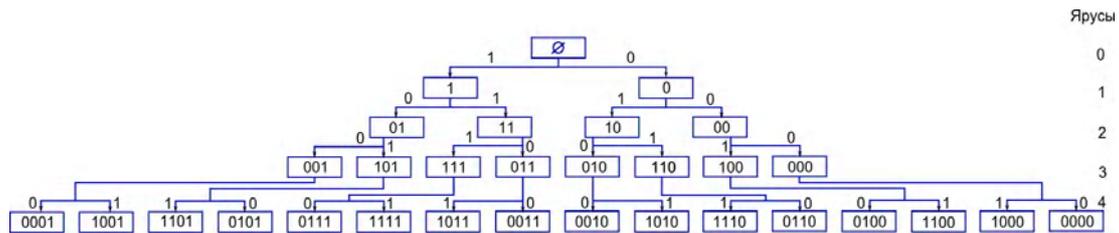


Рис. 2. Генерирования кода Грея

Код принадлежности на ярусе  $lev+1$  получается приписыванием слева к коду принадлежности  $lev$  числа  $\lambda$ , которое попеременно принимает значения 0 или 1. Полученная на каждом ярусе последовательность кодов принадлежности называется *n-элементным кодом Грея*, где  $n$  – номер яруса. Код Грея обладает двумя следующими свойствами:

1. Два соседних набора в коде Грея отличаются на 1 в единственной позиции, наборы находятся на расстоянии по Хэммингу, равном единице. *Расстояние по Хэммингу* в общем случае определяется количеством позиций, по которым отличаются двоичные наборы.

2. Отдельные бинарные наборы кода являются вершинами  $n$ -мерного двоичного куба, а последовательность двоичных наборов есть гамильтонов цикл, по которому можно обойти вершины, включая каждую только один раз (рис. 3).

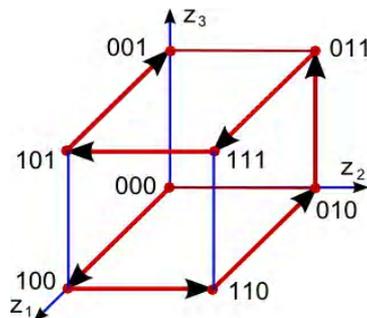


Рис. 3. Генерирование соседнего кода

### 3. Генерация k-элементных подмножеств (сочетаний)

В комбинаторике такие подмножества называют сочетаниями из  $n$  элементов по  $k$  и обозначают  $C_n^k$ . Их количество вычисляется по формуле:

$$C_n^k = \frac{n!}{k!(n-k)!}, 0 \leq k \leq n. \quad (1)$$

Однако при программировании гораздо удобнее использовать следующие рекуррентные соотношения:

$$\begin{aligned} C_n^k &= C_{n-1}^{k-1} + C_{n-1}^k, 0 < k < n; \\ C_n^k &= \frac{n}{k} C_{n-1}^{k-1}, 0 < k \leq n. \end{aligned} \quad (2)$$

Поскольку в формуле (1) числитель и знаменатель растут очень быстро, не всегда возможно точно вычислить значение  $C_n^k$ , даже когда последнее не превосходит максимально представимое целое число. При фиксированном значении  $n$  число сочетаний достигает максимального значения при  $k = n/2$ . Для четного  $n$  максимум  $k = n/2$  (индекс  $k$  соответствует номеру среднего члена разложения бинома Ньютона), а для нечетного – максимум достигается на двух соседних значениях  $k = [n/2]$  и  $k = [n/2] + 1$ . Поэтому полезной оказывается следующая оценка для четных  $n$  (очевидно, что при нечетных  $n$  отличия будут минимальными), основанная на формуле Стирлинга [4]:

$$C_n^{n/2} = \frac{2^n}{\sqrt{\pi n/2}} (1 + O(1/n)). \quad (3)$$

Если допустить, что за время, отведенное для решения задачи, можно перебрать около  $10^6$  вариантов, то из формулы (3) следует, что генерацию всех сочетаний из  $n$  элементов для любого фиксированного  $k$  можно проводить для  $n \leq 24$ . Далее предлагается рассмотреть алгоритмические вопросы, связанные с порождением сочетаний. Для множества  $M = \{1, 2, \dots, n\}$  порождение всех сочетаний (подмножеств) описывается несколькими алгоритмами [1, 2, 6, 8, 11, 13, 18].

#### 3.1. Алгоритм «Сочетания»

1) Дано: множество  $M$ . Найти: последовательность подмножеств множества  $M$ . 2) Начало: пустое множество  $\emptyset$ . 3) Пусть порождено множество  $S$ . Тогда следующее множество  $S'$  получается включением в  $S$  наименьшего не принадлежащего ему элемента и удалением из него всех элементов, меньших включенного. Если таких элементов нет, выполняется завершение алгоритма. Справедливость алгоритма доказывается индукцией по  $n$ : сначала алгоритм порождает все подмножества множества  $M' = \{1, 2, \dots, n-1\}$ , затем повторяет эту последовательность, добавляя к каждому подмножеству элемент  $n$ . Последним членом в последовательности будет само множество  $M$ , и алгоритм автоматически остановится.

#### 3.2. Алгоритм «Сочетания добавлением/исключением одного элемента»

Наряду с рассмотренным выше существует алгоритм, который порождает все подмножества множества  $M$  так, что каждое следующее отличается от предыдущего не более, чем на один элемент. Данный алгоритм определяется индуктивно.

1) Если  $n = 1$ , то результатом алгоритма есть последовательность  $\emptyset, \{1\}$ .

2) Пусть для  $n-1$  результатом алгоритма является последовательность  $L_{n-1}$ . Пусть  $\overline{L_{n-1}}$  – записанная в обратном порядке последовательность  $L_{n-1}$ . Тогда для  $n$  результатом алгоритма будет последовательность  $L_n = L_{n-1}, \overline{L_{n-1}} \vee \{n\}$ , где  $\overline{L_{n-1}} \vee \{n\}$  означает, что к каждому члену последовательности  $\overline{L_{n-1}}$  добавлен элемент  $n$ .

**Пример 1.**  $L_2 = \{\emptyset, \{1\}, \{1,2\}, \{2\}\}$ ;  $L_3 = \{\emptyset, \{1\}, \{1,2\}, \{2\}, \{2,3\}, \{1,2,3\}, \{1,3\}, \{3\}\}$ . Справедливость алгоритма подтверждается индукцией по  $n$ , если учесть, что  $L_{n-1}$  заканчивается множеством  $\{n-1\}$ . Первая половина последовательности  $L_n$  совпадает с  $L_{n-1}$  и по индукции удовлетворяет необходимым требованиям. Следующий элемент после  $L_{n-1}$  есть  $\{n-1, n\}$ , и затем проходит последовательность  $L_{n-1}$  в обратном порядке с добавлением элемента  $\{n\}$ , и требования также выполняются. Поскольку первый элемент в  $L_{n-1}$  есть  $\emptyset$ , то последний элемент в  $L_n$  есть  $\{n\}$ . Если длина  $L_{n-1}$  есть  $2^{n-1}$ , то длина  $L_n$  есть  $2 \cdot 2^{n-1}$ .

### 3.3. Алгоритм порождения сочетаний в лексикографическом порядке

Как правило, генерация всех  $k$ -элементных подмножеств выполняется в лексикографическом порядке, тем более что это не приводит к усложнению алгоритма или увеличению его вычислительной трудоемкости. Порядок подмножеств называется лексикографическим, если для любых двух подмножеств раньше должно быть сгенерировано то из них, из индексов элементов которого можно составить меньшее  $k$ -значное число в  $n$ -ричной системе счисления (в десятичной, для  $n < 10$ ). Например, при  $n=6$  и  $k=3$  сочетание из третьего, первого и пятого элементов должно быть сгенерировано раньше, чем из второго, третьего и четвертого, поскольку  $135 < 234$ .

Рассмотрим рекурсивный алгоритм решения данной задачи. Идея ее сведения к задаче меньшей размерности заключается в следующем. Первым элементом подмножества может быть любой из них, начиная с первого и заканчивая  $(n-k+1)$ -м элементом. После того, как индекс первого элемента подмножества зафиксирован, остается выбрать  $k-1$  элемент из множества с индексами большими, чем у первого. Далее – аналогично. Когда выбран последний элемент, то достигнут конечный уровень рекурсии, и выбранное подмножество можно использовать по назначению.

Ниже приведен алгоритм, который порождает все  $r$ -сочетания  $n$ -элементного множества  $M = \{1, 2, \dots, n\}$  в лексикографическом порядке. Каждое  $r$ -сочетание представляется числами  $a_1 < a_2 < \dots < a_r$ . 1) Начало алгоритма:  $a_1 = 1, a_2 = 2, \dots, a_r = r$ . 2) Пусть порождено  $r$ -сочетание  $a_1, a_2, \dots, a_r$ . Тогда определяется наибольшее  $a_i$  такое, что  $a_i + 1$  не входит в данное сочетание. Если таких  $a_i$  нет (это происходит только при  $a_1 = n - r + 1, a_r = n$ ), то стоп. 3) Если такое  $a_i$  есть, то образуем новое сочетание  $a'_1, a'_2, \dots, a'_r$ , полагая

$$a'_1 = a_1, \dots, a'_{i-1} = a_{i-1}, \dots, a'_i = a_i + 1, a'_{i+1} = a_i + 2, \dots, a'_r = a_i + r - 1.$$

**Пример 2.** Алгоритм для  $n = 4, r = 2$ :

Шаги	Сочетания	$a_i$
1	1,2	2
2	1,3	3
3	1,4	1
4	2,3	3
5	2,4	2
6	3,4	Стоп

В предлагаемом ниже листинге 1 массив  $a$  содержит значения элементов исходного множества и может быть заполнен произвольным образом. В массиве  $p$  формируется очередное сочетание из  $k$  элементов.

Листинг 1:

```
const nmax = 24;
type list = array[1..nmax] of integer;
var k, i, j, n, q: integer;
a, p: list;
```

```

procedure print(k: integer);
var i: integer;
begin
for j := 1 to k do
write(p[j]:4);
writeln
end; {print}
procedure cnk(n, k: integer);
procedure gen(m, L: integer);
var i: integer;
begin
if m = 0 then print(k)
else
for i := L to n - m + 1 do
begin
p[k - m + 1] := a[i];
gen(m - 1, i + 1)
end
end; {gen}
begin {cnk}
gen(k, 1)
end; {cnk}
begin {main}
readln(n,k);
for i := 1 to n do
a[i] := i;
{заполнить массив можно и по-другому}
cnk(n,k)
end

```

Генерация сочетаний выполняется в рекурсивной подпрограмме gen со следующими параметрами: m – количество элементов, которые осталось выбрать из множества; L – элемент исходного множества, начиная с которого следует выбирать эти m элементов. Вложенная структура описания процедур cnk и gen позволяет не передавать при рекурсивных вызовах значения n и k, а из основной программы обращаться к процедуре cnk с параметрами, соответствующими постановке задачи.

#### 4. Генерация всех подмножеств данного множества (булеана)

##### 4.1. Перебор по возрастанию/убыванию числа элементов

При отыскании минимального/максимального подмножества, состоящего как можно из меньшего/большого числа элементов, эффективнее всего организовать перебор так, чтобы сначала проверялись все подмножества из одного элемента, затем – из двух, трех и далее элементов (для максимального подмножества — в обратном порядке). В этом случае первое подмножество, удовлетворяющее условию задачи, будет искомым, и дальнейший перебор следует прекратить. Для реализации такого перебора можно воспользоваться описанной выше процедурой cnk и ввести в нее еще один параметр – логическую переменную flag, которая будет идентифицировать соответствие текущего сочетания элементов условию задачи. При получении очередного сочетания необходимо обратиться к процедуре его проверки check, которая определяет значение флага. Тогда начало процедуры gen следует переписать следующим образом (листинг 2).

Листинг 2:

```

procedure gen(m, L: integer);
var i: integer;
begin
if flag then exit;
if m = 0 then

```

```

begin check(p,k,flag);
end
else ...

```

Далее процедура совпадает с предыдущей версией (см. листинг 1). В основной программе единственное обращение к данной процедуре следует заменить следующим фрагментом (листинг 3).

```

Листинг 3:
k:=0;
flag:=false;
repeat k:=k+1;
cnk(n,1,flag) until flag or (k=n);
if flag then print(k)
else writeln ('no solution');

```

Видно, что в основной программе запрос значения переменной k теперь не производится.

#### 4.2. Перебор по двоичной системе (бинарному или двоичному коду)

Рассматривается альтернативный подход к перебору всех подмножеств заданного множества. Любое подмножество можно охарактеризовать, указав относительно каждого элемента исходного множества его принадлежность данному подмножеству и поставив в соответствие каждому элементу множества 0 или 1. Каждому подмножеству соответствует n-значное число в двоичной системе счисления. Поскольку числа могут начинаться с произвольного количества нулей, которые не являются значащими цифрами, в соответствии ставятся n или менее значные числа. Отсюда следует, что полный перебор всех подмножеств исходного множества соответствует перебору всех чисел в двоичной системе счисления от  $\underbrace{0\dots0}_n 1$  до  $\underbrace{1\dots1}_n$ .

Теперь можно подсчитать количество различных подмножеств данного множества. Оно равно  $2^n - 1$  или  $2^n$  с учетом пустого множества.

Таким образом, сопоставляя два способа перебора всех подмножеств данного множества, можно получить следующую формулу:

$$C_n^0 + C_n^1 + \dots + C_n^n = 2^n, \quad (4)$$

которая известна как сумма биномиальных коэффициентов [7].

Итак, в рамках сделанной выше оценки на количество допустимых вариантов перебора можно рассмотреть все подмножества исходного множества только при  $n \leq 20$ .

Применение программ, соответствующих второму способу перебора, целесообразно, когда: 1. Необходимо перебрать все подмножества данного множества, например, требуется найти все решения, удовлетворяющие заданному условию. 2. По условию задачи не имеет значения количество элементов, которое должно входить в искомое подмножество. На примере такой задачи далее приводится программа генерации всех подмножеств исходного множества в лексикографическом порядке [5].

**Пример 3.** Даны целочисленный массив  $a[1, \dots, N]$ ,  $N \leq 20$  и число M. Найти подмножество элементов массива  $a[i_1], a[i_2], \dots, a[i_k]$  такое, что  $1 \leq i_1 < i_2 < \dots < i_k \leq N$  и  $a[i_1] + a[i_2] + \dots + a[i_k] = M$ .

В качестве решения приводится процедура генерации всех подмножеств, которые можно составить из элементов массива, и функция проверки конкретного подмножества на соответствие условию задачи (листинг 4).

```

Листинг 4:
function check(j: longint): boolean;
var k: integer; s: longint;
begin
s := 0;
for k := 1 to n do

```

```

{данное условие означает, что в k-й справа позиции числа j во 2-й системе стоит 1}
if ((j shr (k — 1)) and 1) = 1
then s := s + a[k];
if s = m then
begin
for k := 1 to n do
if ((j shr (k — 1)) and 1) = 1
then write(a[k]:4);
writeln
end
end;
procedure subsets(n: integer);
var q,j: longint;
begin
{помещаем в q число 2^n}
q := 1 shl n;
{цикл по всем подмножествам}
for j := 1 to q — 1 do
if check(j) then exit
end

```

Следует заметить, что если все элементы в массиве положительные, то решение приведенной выше задачи можно сделать более эффективным путем изменения порядка рассмотрения подмножеств, а именно: если сумма элементов какого-либо подмножества уже больше  $M$ , то рассматривать подмножества, включающие его в себя, уже не имеет смысла. Пересчет сумм может быть оптимизирован, если каждое следующее сгенерированное подмножество будет отличаться от предыдущего не более чем на один элемент [4]. Приведенная программа достаточно проста, но обладает одним недостатком: с ее помощью нельзя перебрать все подмножества множеств из более, чем 30 элементов, что обусловлено максимальным числом битов (32), отводимых на представление целых чисел. На самом деле перебор всех подмножеств множеств большей размерности вряд ли возможен за время, отведенное для решения той или иной задачи.

**Пример 4.** Для трехэлементного множества  $M = \{1, 2, 3\}$  все подмножества следующие:  $\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}$ . Первым в списке идет пустое множество, которое является подмножеством любого множества, последним – исходное множество как несобственное подмножество самого себя. Рассматриваются числа в диапазоне от 0 до  $2^n - 1$  в двоичной системе счисления с разрядностью  $n$ , которые можно увеличить при необходимости незначащими нулями: 0 (000); 1 (001); 2 (010); 3 (011); 4 (100); 5 (101); 6 (110); 7 (111).

Каждый из этих двоичных кодов можно использовать как маску подмножества, а именно: единица в  $i$ -м бите характеризует наличие  $i$ -го элемента множества в подмножестве, ноль – его отсутствие. Например:  $101 = \{1, 3\}$ . Таким образом, для генерации подмножеств необходимо организовать цикл от 0 до  $2^n - 1$ , при этом на каждой итерации представить значение счетчика в виде бинарного кода и на его основе составить подмножество. Реализация на языке Си для четырехэлементного множества  $M = \{3, 6, 7, 9\}$  представлена ниже.

```

Листинг 5:
#include <stdio.h>
#include <stdlib.h>
//—для возведения в степень
#include <math.h>
int main()
{
int M[4] = {3, 6, 7, 9}; //—множество
int w = 4; //—кол-во элементов множества
int i, j, n;

```

```

n = pow(2, w);
for ( i = 0; i < n; i++ ) //—перебор битовых масок
{
printf(“{“);
for ( j = 0; j < w; j++ ) //—перебор битов в маске
if ( i & (1 << j) ) //—если j-й бит установлен
printf(“%d “, M[j]); //—то выводим j-й элемент множества
printf(“}\n”);
}
return 0;
}

```

Результат представлен следующим набором подмножеств: {}, {3}, {6}, {3, 6}, {7}, {3, 7}, {6, 7}, {3, 6, 7}, {9}, {3, 9}, {6, 9}, {3, 6, 9}, {7, 9}, {3, 7, 9}, {6, 7, 9}, {3, 6, 7, 9}. Порядок в данном случае не имеет значения. Программная реализация вывода подмножеств достаточно проста. В процедуре `if ( i & (1 << j) )` оператор `<<` есть логический сдвиг влево. Чтобы узнать значение j-го бита числа i (биты отсчитываются слева направо, нумерация начинается с нуля), следует сначала сдвинуть единицу на j. Допустим,  $i=18$  (10010),  $j=2$ . После логического сдвига влево единицы (00001) на 2 получается число 00100. Затем число i логически умножается на полученную после сдвига маску. Маска представляет совокупность нулей с установленным в единицу j-м битом благодаря сдвигу. При умножении, если j-й бит в числе i равен единице, получается некоторое число, равное маске; если j-й бит равен нулю, то результат умножения будет нулем: 10010, 00100. Побитовое умножение двух чисел – логическая операция `and` – дает число 00000. Это говорит о том, что 2-й бит числа 18 (согласно порядку нумерации битов) выставлен в ноль. Данный алгоритм позволяет довольно просто сгенерировать случайное подмножество множества. Для этого необходимо взять случайное число в диапазоне от 0 до  $2^n - 1$ , использовать его как битовую маску и сгенерировать на ее основе подмножество.

### 4.3. Процедура генерирования булеана

Задано множество из четырех компонентов: A, B, C, D. Необходимо выбрать подмножество, состоящее из нескольких примитивов, обладающее некоторым свойством. Способ решения задачи заключается в генерировании всех возможных подмножеств данного множества и для каждого из сгенерированных подмножеств выполняется проверка, удовлетворяет ли оно заданному свойству. Альтернативный вариант решения задачи – вычислить все подмножества данного множества, обладающие заданным свойством. Например, для множества из четырех символов A, B, C, D булеан включает следующие множества: пустое множество; одноэлементные множества: {A}, {B}, {C}, {D}; двухэлементные множества: {A, B}, {A, C}, {A, D}, {B, C}, {B, D}, {C, D}; трехэлементные множества: {A, B, C}, {A, B, D}, {A, C, D}, {B, C, D}; четырехэлементное множество: {A, B, C, D}.

В случае, когда порядок генерации подмножеств не имеет значения, алгоритм генерации множества всех подмножеств выглядит следующим образом [20]. Иницируется вектор B, состоящий из четырех разрядов, каждый из которых может принимать значение 0 или 1. Единица указывает на включение соответствующего по номеру компонента исходного множества в искомое. Значение 0 указывает на то, что элемент отсутствует. Пусть имеется последовательность двоичных чисел от 0 до 15 и соответствующие им подмножества:

```

{0000} – пустое множество;
{0001} – A, {0010} – B, {0100} – C, {1000} – D;
{0011} – AB, {0101} – AC, {0110} – BC, {1001} – AD, {1010} – BD, {1100} – CD;
{0111} – ABC, {1011} – ABD, {1101} – ACD, {1110} – BCD;
{1111} – ABCD.

```

Таким образом, имеется 16 различных подмножеств универсума из 4 элементов. В общем случае множество всех подмножеств множества из N элементов содержит  $2^N$  элементов. Алгоритм, обеспечивающий генерацию множества всех подмножеств из N элементов, может быть неформально описан следующим образом: формирование вектора,

состоящего из N нулей и соответствующего пустому множеству. Для получения следующего подмножества из текущего обрабатывается текущий вектор следующим образом: справа от первого элемента массива к последнему ищется первое число, равное 0. Если такое число не найдено, то текущее подмножество является решением – множеством, состоящим из всех элементов. Если же элемент, равный 0, найден, то он заменяется на 1, а все числа справа от него, если такие имеются, заменяются на нули. Данный алгоритм может быть записан следующим образом:

```

Ввод (N)
Обнуление массива В из N+1 элемента
Вывод (пустое множество)
Пока В[N+1]=0
  i:=1
  Пока В[i]=1 делать В[i]=0, i=i+1
  В[i]=1
Вывод (множество, определяемое массивом В)
Соответствующая программа имеет следующий вид (листинг 6).

```

Листинг 6:

```

Программа генерации множества всех подмножеств
const
alphabet : string[26] = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
var
b : array [1..100] of byte;
N,i : byte;
begin
readln(N);
for i:=1 to N+1 do b[i]:=0;
writeln ('Пустое множество');
while (b[N+1]=0) do
begin
i:=1;
while B[i]=1 do
begin B[i]:=0; inc(i); end;
B[i]:=1;
for i:=1 to n do
if b[i]=1 then write(alphabet[i]);
writeln;
end;
end

```

## 5. Квантовый процессор оптимального покрытия

Цель создания кубит-процессора – существенное уменьшение времени при решении задач оптимизации путем параллельного вычисления векторных логических операций над множеством всех подмножеств от примитивных компонентов за счет увеличения памяти для хранения промежуточных данных.

Задачи: 1) Определение структур данных для взятия булеана при решении задачи покрытия столбцов матрицы  $M = \begin{bmatrix} M_{ij} \end{bmatrix}, i = \overline{1, m}; j = \overline{1, n}$  единичными значениями строк. В частности, при  $m = n = 8$ , необходимо выполнить параллельно-логическую операцию над 256 вариантами всех возможных сочетаний векторов (строк матрицы), составляющих булеан. 2) Система команд процессора должна включать следующие операции (and, or, xor) над векторами (словами), размерности  $m$ . 3) Разработка архитектуры кубит-процессора для параллельного вычисления  $2^n - 1$  вариантов сочетаний, направленных на оптимальное решение NP-полной задачи покрытия. 4) Реализация прототипа кубит-процессора на базе программируемой логики PLD и верификация (валидация) аппаратного решения на примерах минимизации булевых функций. 5) Приведение других практических задач дискретной оптимизации к форме задачи покрытия для последующего решения на кубит-процессоре.

В качестве примера предлагается решить задачи поиска оптимального единичного покрытия всех столбцов минимальным числом строк матрицы M, представленной ниже:

M	1	2	3	4	5	6	7	8
a	1	.	.	.	.	1	.	.
b	.	.	1	.	.	.	1	.
c	1	.	.	.	1	.	1	.
d	.	1	.	1	.	.	.	1
e	.	1	.	.	1	.	.	.
f	1	.	1	.	.	1	1	.
g	.	1	.	1	.	.	.	1
h	.	.	1	.	1	.	.	.

Для этого необходимо сделать перебор всех 255 сочетаний: из восьми по одной строке, по двум, трем, четырем, пяти, шести, семи и восьми. Минимальное количество примитивов (строк), формирующее покрытие, есть оптимальное решение. Таких решений может быть несколько. Диаграмма Хассе есть компромиссное предложение относительно времени и памяти, или такая стратегия решения задачи покрытия, когда ранее полученный результат впоследствии используется для создания более сложной суперпозиции. Поэтому для каждой таблицы покрытия, содержащей n примитивов (строк), необходимо генерировать собственную мультипроцессорную структуру в форме диаграммы Хассе, которая далее должна быть использована для почти параллельного решения NP-полной задачи. Например, для четырех строк таблицы покрытия диаграмма Хассе – структура мультипроцессора – будет иметь вид, представленный на рис. 4.

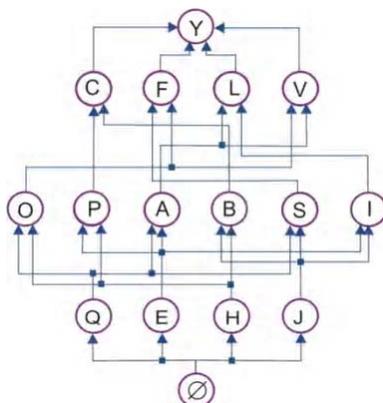


Рис. 4. Квантовая структура вычислительных процессов

Оптимальные решения задачи покрытия для матрицы M, которая генерирует 255 вариантов возможных сочетаний, представлены строками в форме ДНФ:  $C = fgh \vee efg \vee cdf$ .

Автомат управления вычислительным процессом для квантовой структуры путем восходящего анализа вершин графа основан на последовательном выполнении следующих шагов:

1. Занесение информации о примитивах в регистры (матрицы)  $L_i^1 = P_i$  первого уровня с последующим анализом качества покрытия каждым примитивом в двоичном формате (1 – есть покрытие, 0 – нет покрытия). Если один из примитивов организует покрытие  $\bigwedge_{j=1}^m L_{ij}^1 = 1$ , процесс анализа структуры Хассе заканчивается. Иначе – выполнение перехода ( $r = r + 1$ ) на следующий уровень графа:

$$L_i^1 = P_i \rightarrow \bigwedge_{j=1}^m L_{ij}^1 = \begin{cases} 0 \rightarrow n = n + 1; \\ 1 \rightarrow \text{end.} \end{cases}$$

2. Инициирование команды обработки очередного (второго) уровня. Последовательное выполнение векторных (матричных) операций ( $\text{or } L_i^r = L_{ij}^{r-1} \bigwedge_{j=1}^m L_{ij}^{r-1}$ , and  $\bigwedge_{j=1}^m L_{ij}^r = 1$ ) в целях анализа покрывающей способности сочетаний примитивных элементов r-уровня. Здесь

$t = \overline{1, m}, i = \overline{1, m}, r = \overline{1, n}$ ,  $n$  – число ярусов или количество строк в таблице покрытия,  $m$  – число столбцов в ней. Если существует сочетание на рассматриваемом уровне, создающее полное покрытие, которое формирует оценку, равную 1, то обработка всех последующих уровней процессора не выполняется. В противном случае выполняется переход для анализа следующего уровня процессорной структуры:

$$L_i^r = L_{ij}^{r-1} \bigwedge_{j=1}^m L_{ij}^{r-1} \rightarrow \bigwedge_{j=1}^m L_{ij}^r = \begin{cases} 0 \rightarrow r = r + 1; \\ 1 \rightarrow \text{end}. \end{cases}$$

Для поиска оптимального покрытия всегда достаточно двух элементов нижнего уровня, что означает – все операционные вершины имеют два регистровых (матричных входа), что существенно уменьшает стоимость аппаратных затрат. Количество временных тактов для обработки структуры процессора в худшем случае равно  $n$ . Можно создать алгоритм поиска оптимального покрытия путем нисходящего анализа вершин графа. В этом случае при нахождении полного покрытия в одном из ярусов необходим еще один спуск по структуре, чтобы убедиться в отсутствии в нижнем соседнем ярусе полного покрытия. При положительном ответе на данный вопрос полученное решение является оптимальным. Иначе – необходимо выполнять спуск до такого уровня, когда, более нижний, соседний ярус не будет содержать полного покрытия.

Вершины процессорной структуры могут иметь более одной бинарной (унарной) регистровой логической операции. Тогда необходимо создавать простейший дешифратор команд для активизации, например, операций: and, or, xor, not.

Таким образом, достоинства кубитного Хассе процессора (Qubit Hasse Processor – QHP) заключаются в возможности использовать не более, чем двухвходовые схемы векторных логических операций (and, or, xor), а значит, в существенном уменьшении стоимости по Квайну реализации процессорных элементов (вершин) и памяти за счет применения последовательных вычислений и незначительного увеличения времени обработки всех вершин графа Хассе. Для каждой вершины используется критерий качества покрытия – наличие всех единиц в координатах вектора-результата. Если критерий качества выполняется, то все остальные вычисления можно не производить, поскольку диаграмма Хассе есть строго иерархическая структура по числу сочетаний в каждом ярусе. Это означает, что самое лучшее решение находится на более низком уровне иерархии. Варианты одного уровня равнозначны по реализации (стоимости), поэтому полученное

первое качественное покрытие ( $Q = \sum_{i=1}^n q_i = n$ ) есть лучшее решение, предполагающее остановку всех последующих вычислений по стратегии диаграммы Хассе. С учетом последовательно-параллельной стратегии анализа вершин графа, время (цикл) обработки всех примитивов QH-процессора определяется числом уровней иерархии, функционально зависящим от количества битов в кубитной переменной:  $T = \log_2 2^n \times t = t \times n$ . При этом длина  $m$  строки таблицы покрытия не влияет на оценку быстродействия. Анализ вершины включает две команды: логическую (and, or, xor) и операцию вычисления критерия качества покрытия в форме скаляра путем применения функции and ко всем разрядам вектора-результата:

$$m_{ir,j} = M_{i,j} \vee M_{r,j}, (j = \overline{1, n}; \{i \neq r\} = \overline{1, m});$$

$$m_{ir}^s = \bigwedge m_{ir,j} = \bigwedge (M_{i,j} \vee M_{r,j}).$$

Аппаратные затраты на реализацию QH-процессора зависят от суммарного числа вершин графа Хассе и от количества битов (разрядов) в строке таблицы покрытия:

$$H = 2^n \times k \times m,$$

где  $k$  – коэффициент аппаратной реализации (сложности) одного разряда бинарной векторной логической операции и последующей команды вычисления критерия качества покрытия.

Таким образом, высокое быстродействие решения задачи покрытия достигается существенным повышением аппаратных затрат (в  $2^n \times k \times m / k \times m \times n = 2^n / n$  раз по сравнению

нию с последовательной обработкой графовых вершин), которое обеспечивает компромиссный вариант между полностью параллельной структурой вычислительных процессов (здесь затраты аппаратуры определяются числом примитивов в каждой вершине  $H = k \times m \times n \times 2^n$ , а увеличение аппаратуры составляет  $2^n$  раз) и последовательными вычислениями однопроцессорного компьютера (здесь быстродействие обработки графа Хассе равно  $T = t \times 2^n$ , а аппаратные затраты равны  $H^* = k \times m \times n$ ). Уменьшение аппаратуры по сравнению с параллельным вариантом обработки графа составляет  $Q^H = k \times m \times n \times 2^n / k \times m \times 2^n = n$ . Как следствие существенной аппаратной избыточности, уменьшение времени анализа вершин графа по сравнению с последовательной обработкой структуры имеет следующую оценку:

$$Q^T = \frac{t \times 2^n}{t \times n} = \frac{2^n}{n}.$$

## 6. Практическая реализация кубит-процессора оптимального покрытия

Модель квантового устройства разработана на языке Verilog. Элементарная ячейка процессора состоит из двух регистровых вентилях. Регистровый элемент выполняет логическую операцию над двумя векторами, формируя вектор результата. Регистровый вентиль and выполняет свертку всех битов вектора по операции and, формируя однобитовый элемент, идентифицирующий единичным значением оптимальное решение задачи покрытия. Представлено формирование значений для вершин диаграммы Хассе шести уровней, где каждый элемент в схеме имеет на выходе примитив анализа качества покрытия в виде функции and. Реализация вычислительного устройства выполнена на основе кристалла FPGA фирмы Xilinx xc3s1600e-4-fg484, где основные параметры имеют следующий вид:

Листинг 7:

Map-report

Logic Utilization:

Number of Slice Flip Flops: 2,286 out of 29,504 7%

Number of 4 input LUTs: 2,715 out of 29,504 9%

Logic Distribution:

Number of occupied Slices: 1,514 out of 14,752 10%

Number of Slices containing only related logic: 1,514 out of 1,514 100%

Number of Slices containing unrelated logic: 0 out of 1,514 0%

\*See NOTES below for an explanation of the effects of unrelated logic.

Total Number of 4 input LUTs: 2,715 out of 29,504 9%

Number of bonded IOBs: 321 out of 376 85%

Number of BUFGMUXs: 1 out of 24 4%

Timing parameters of project:

Tclk\_to\_clk = 4.672 ns

Tclk\_to\_pad\_max = 11.552 ns

Period = max{ Tclk\_to\_clk, Tclk\_to\_pad\_max };

Period = 11.552 ns

Fclk = 86,5 МГц

### Заключение

Представлен обзор и классификация моделей, структур данных и методов генерирования булеана для решения практических задач дискретной оптимизации.

*Научная новизна* – предложена модель данных и структура аппаратной реализации квантового компьютера, которая характеризуется использованием диаграммы Хассе, что дает возможность существенно ( $\times 100$ ) повысить быстродействие решения комбинаторных задач дискретной оптимизации.

*Практическая значимость* – существенное повышение быстродействия при решении задач покрытия и других задач дискретной оптимизации за счет увеличения аппаратных затрат для параллельного выполнения векторных логических операций на Хассе-структуре квантового вычислительного устройства.

**Список литературы:** 1. *Burris, Stanley N., Sankappanavar H.P.* A Course in Universal Algebra. — Springer-Verlag, 1981. 332 p. 2. *Peter Jipsen, Henry Rose.* Varieties of Lattices – Lecture Notes in Mathematics 1533, Springer Verlag, 1992. 162 p. 3. *Носов В.А.* Комбинаторика и теория графов. М.: Изд-во МГИЭМ, 1999. 112 с. 4. *Стив Тейксейра, Ксавье Пачеко.* Borland Delphi 6. Руководство разработчика (Steve Teixeira, Xavier Pacheco. Delphi 6 Developer's Guide First Edition). М.: Издательский дом "Вильямс", 2002. 1120 с. 5. *Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн.* Алгоритмы: построение и анализ. М.: Издательский дом "Вильямс", 2012. 1296 с. 6. *Комбинаторный анализ.* Задачи и упражнения / Под ред. Рыбникова К.А. М.: Наука, 1982. 184 с. 7. *Гаврилов Г.П., Сапоженко А.А.* Сборник задач по дискретной математике. М.: Наука, 1977. 368 с. 8. *Липский В.* Комбинаторика для программистов: Пер. с польск. М.: Мир, 1988. 213 с. 9. *Романовский И.В.* Дискретный анализ. СПб: Невский Диалект; БХВ Петербург, 2003. 320 с. 10. *Ахо А., Хопкрофт Дж., Ульман Дж.* Структуры данных и алгоритмы. М.: Издательский дом «Вильямс», 2000. 384 с. 11. *Вирт Н.* Алгоритмы и структуры данных / Пер. с англ. М.: Мир, 1989. 360 с. 12. *Окулов С.М., Ашихмина Т.В., Бушмелева Н.А.* и др. Задачи по программированию / Под ред. С.М. Окулова. М.: БИНОМ. Лаборатория знаний, 2006. 820 с. 13. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999. 960 с. 14. *Кнут Д.* Искусство программирования. Сортировка и поиск. Т. 3. М.: «Вильямс», 2007. 824 с. 15. *Кнут Д.* Искусство программирования. Генерация всех кортежей и перестановок. Т. 4, вып. 2. М.: «Вильямс», 2008. 160 с. 16. *Кнут Д.* Искусство программирования, том 4, выпуск 3. Генерация всех сочетаний и разбиений. М.: «Вильямс», 2007. 208 с. 17. *Грэхем Р., Кнут Д., Паташник О.* Конкретная математика. Основание информатики / Пер. с англ. М.: Мир, 1998. 703 с. 18. *Липский В.* Комбинаторика для программистов. М.: Мир, 1988. 77 с. 19. *Рейнгольд Э., Нивергельт Ю., Део Н.* Комбинаторные алгоритмы: теория и практика. М.: Мир, 1980. 476 с. 20. *Долинский М.С.* Алгоритмизация и программирование на Turbo Pascal: от простых до олимпиадных задач: Учебное пособие. Изд-во: Питер, 2006. 366 с. 21. *Beth T.* Quantum computing: an introduction // Proceedings of the IEEE International Symposium on Circuits and Systems, ISCAS. 2000. Geneva. Vol. 1. P. 735 – 736. 22. *Jonker P., Jie Han.* On quantum and classical computing with arrays of superconducting persistent current qubits // Proceedings of Fifth IEEE International Workshop on Computer Architectures for Machine Perception. 2000. P. 69 – 78. 23. *Keyes R.W.* Challenges for quantum computing with solid-state devices // Computer. Jan. 2005. Vol. 38, Issue 1. P. 65 – 69. 24. *Инфраструктура мозгоподобных вычислительных процессов / М.Ф. Бондаренко, О.А. Гузь, В.И. Хаханов, Ю.П. Шабанов-Кушнаренко.* Харьков: Новое Слово. 2010. 160 с. 25. *Mark Gregory Whitney.* Practical Fault Tolerance for Quantum Circuits. PhD Dissertation in Computer Science. Berkeley: University of California. 2009. 206 p. 26. *Хаханов В. И., Литвинова Е. И., Чумаченко С. В., Гузь О. А.* Логический ассоциативный вычислитель // Электронное моделирование. 2011. № 1. С. 73-90. 27. *Hahanov V., Wajeb Gharibi, Litvinova E., Chumachenko S.* Information analysis infrastructure for diagnosis // Information. An international interdisciplinary journal. 2011. Japan. Vol. 14, No 7. P. 2419-2433. 28. *Хаханов В.И.* Проектирование и тестирование цифровых систем на кристаллах / В.И. Хаханов, Е.И. Литвинова, О.А. Гузь. Харьков: ХНУРЭ, 2009. 484 с.

Поступила в редколлегию 14.12.2011

**Чумаченко Светлана Викторовна**, д-р техн. наук, профессор кафедры АПВТ ХНУРЭ. Научные интересы: математическое моделирование, теория рядов, методы дискретной оптимизации. Увлечения: путешествия, любительское фото. Адрес: Украина, 61166, Харьков, пр. Ленина, 14, тел. 70-21-326. E-mail: ri@kture.kharkov.ua.

**Хаханов Владимир Иванович**, декан факультета КИУ ХНУРЭ, д-р техн. наук, профессор кафедры АПВТ ХНУРЭ. Научные интересы: техническая диагностика цифровых систем, сетей и программных продуктов. Увлечения: баскетбол, футбол, горные лыжи. Адрес: Украина, 61166, Харьков, пр. Ленина, 14, тел. 70-21-326. E-mail: hahanov@kture.kharkov.ua.

**Мурад Али Аббас**, аспирант кафедры АПВТ ХНУРЭ. Научные интересы: компьютерные системы и сети. Адрес: Украина, 61166, Харьков, пр. Ленина, 14, тел. 70-21-326.

**Горобец Олег Александрович**, магистрант Центра последипломного образования ХНУРЭ. Научные интересы: аналитика программного обеспечения, облачные вычисления и социальные сети. Адрес: Украина, 61166, Харьков, пр. Ленина, 14, e-mail: gorobetsu@gmail.com.