Санкт-Петербургский государственный политехнический университет
Факультет технической кибернетики
Кафедра распределённых вычислений и компьютерных сетей

А.Б.Беляев
И.В. Шошмина

Использование бинарных решающих диаграмм для решения логических задач. Библиотека BuDDy

ПРЕДСТАВЛЕНИЕ БУЛЕВЫХ ФУНКЦИЙ В ВИДЕ БИНАРНЫХ РЕШАЮЩИХ ДИАГРАММ

Булевы функции могут быть представлены многими различными способами:

1) таблица истинности:

pqr	***f***
000	1
001	1
010	1
011	0
100	0
101	0
110	1
111	0

- бинарное решающее дерево (рис. 1):

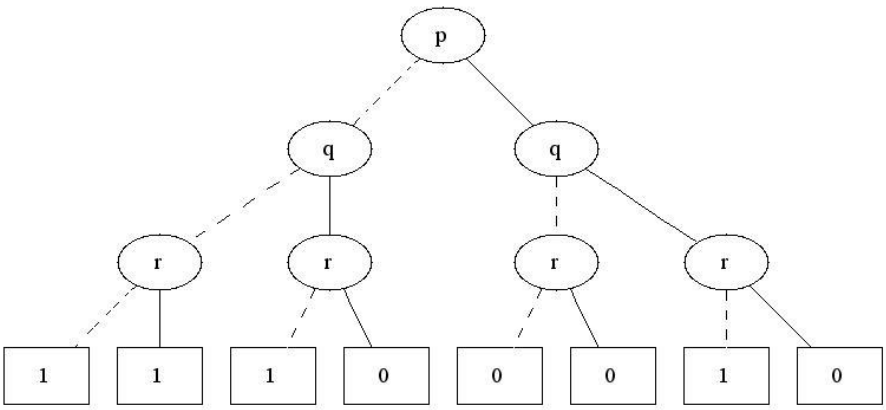


Рис. 1

2) пропозициональная формула: $f(p,q,r) = \bar{p} \vee q \oplus rq(p \vee r)$;
   - дизъюнктивная нормальная форма: $f(p,q,r) = \bar{p}\bar{q} \vee q\bar{r}$;
   - конъюнктивная нормальная форма: $f(p,q,r) = (\bar{p} \vee q)(\bar{q} \vee \bar{r})$;
   - совершенная дизъюнктивная нормальная форма: $f(p,q,r) = \bar{p}\bar{q}r \vee \bar{p}\bar{q}\bar{r} \vee qp\bar{r} \vee q\bar{p}\bar{r}$;
   - полином Жегалкина: $f(p,q,r) = 1 \oplus p \oplus pq \oplus qr$.

Здесь p, q и r – двоичные переменные.

Все приведённые представления громоздки. Вследствие экспоненциального роста они не позволяют решать основные проблемы, связанные с булевыми функциями:

- найти значение БФ на конкретной интерпретации;
- найти все интерпретации, на которых функция истинна;

- проверить выполнимость, общезначимость, невыполнимость БФ;
- проверить на совпадение две БФ.

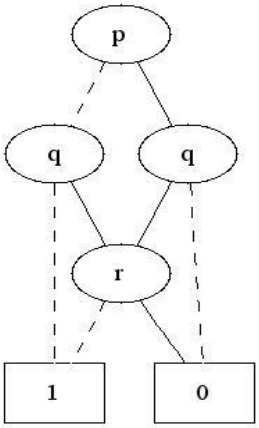


Рис. 2

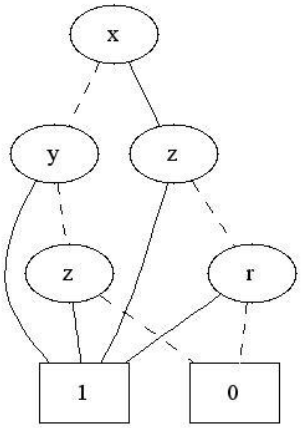


Рис. 3

В 1986 г. Randel Bryant предложил новую очень эффективную форму представления БФ, которая называется бинарная решающая диаграмма – Binary Decision Diagrams (BDD) [1].

BDD – ациклический орграф, в котором отсутствуют повторения в структуре, с одной корневой вершиной, двумя листьями, помеченными 0 и 1, и промежуточными вершинами. Корневые и промежуточные вершины помечены переменными, из них выходят ровно два ребра (рис. 2). BDD – это семантическое дерево без избыточностей. Как правило, представление в форме BDD используемых на практике булевых функций растет полиномиально или даже линейно.

Упорядоченная (Ordered) BDD – это BDD, в которой переменные встречаются в фиксированном порядке. Не всякая BDD упорядочена (рис. 3). Бинарная решающая диаграмма булевой функции может быть построена из ее семантического дерева применением операций редукции, которые состоят в следующем.

1. Если несколько вершин орграфа помечено одним именем, и они имеют изоморфные подструктуры, то граф можно редуцировать: оставить только одну из них.

2. Если обе выходные дуги промежуточной вершины v орграфа ведут в одну вершину, то эта вершина v может быть выброшена.

OBDD, полученная как результат многократного выполнения любом порядке этих двух операций, является каноническим представлением, она носит название редуцированной (Reduced). Именно ROBDD обладают всеми полезными свойствами. Далее будем под BDD подразумевать ROBDD.

Следует отметить, что сложность BDD зависит от порядка переменных. На рис. 4 и 5 изображены представления функции $f = a_1 b_1 \vee a_2 b_2 \vee a_3 b_3$ при двух разных упорядочениях переменных. Первая BDD – при порядке

$a_1 < a_2 < a_3 < b_1 < b_2 < b_3$,

вторая BDD – при порядке

$a_1 < b_1 < a_2 < b_2 < a_3 < b_3$.

Задача нахождения оптимального порядка переменных при построении BDD является $\mathcal{NP}$-полной, однако существуют эффективные эвристики для нахождения субоптимального порядка.

По BDD может быть восстановлена ДНФ функции. Она равна дизъюнкции всех путей из корневой вершины в терминальный узел 1. ДНФ булевой функции $f(p,q,r) = \bar{p}\bar{q} \vee \bar{p}q\bar{r} \vee pq\bar{r}$, BDD которой представлена на рис.1:.

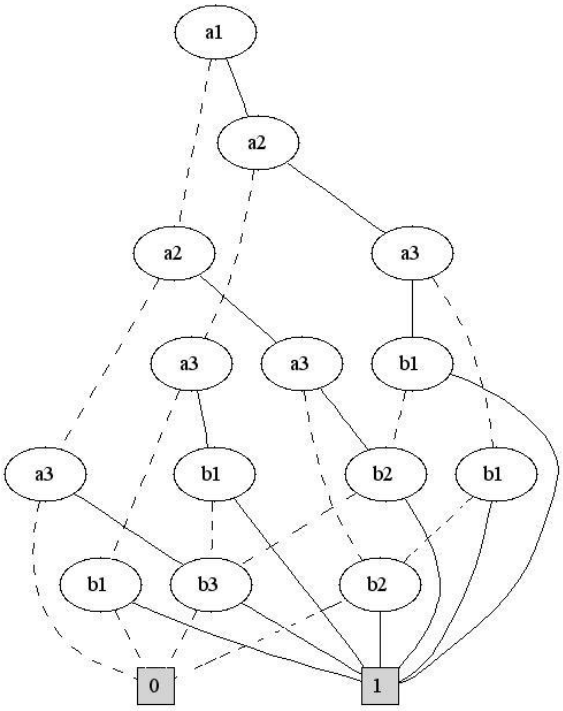


Рис. 4

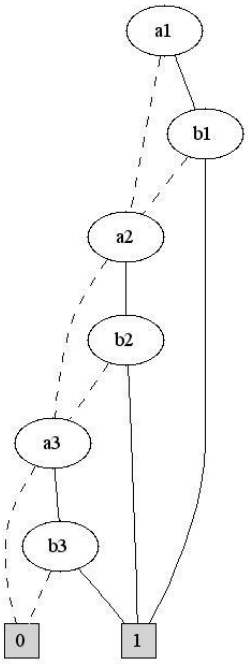
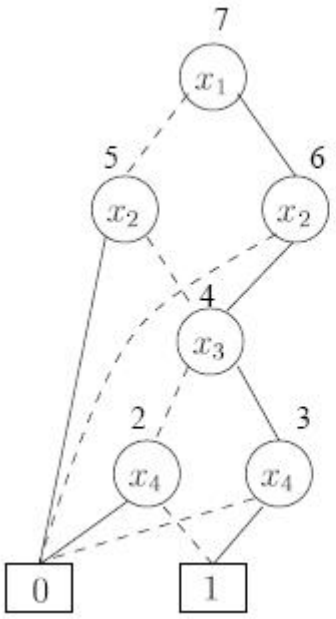


Рис. 5

Скажем несколько слов о реализации BDD. Диаграмма представляется таблицей (рис. 6), в которой каждой вершине графа отводится одна строка. В строке указываются номер вершины, номер переменной, и те две вершины, куда направлены два выхода, 0 и 1, из данной вершины. Сложность представления БФ в BDD пропорциональна числу вершин.

Достаточно просто реализуются булевы операции над булевыми функциями, уже представленными в BDD. На рис. 7 показано, как выполняется некоторая произвольная бинарная логическая операция $\otimes$ над вершинами одного и разных уровней ($x_i < x_k$).



u	var	low	$high$
0			
1			
2	4	1	0
3	4	0	1
4	3	2	3
5	2	4	0
6	2	0	4
7	1	5	6

с

Рис. 6

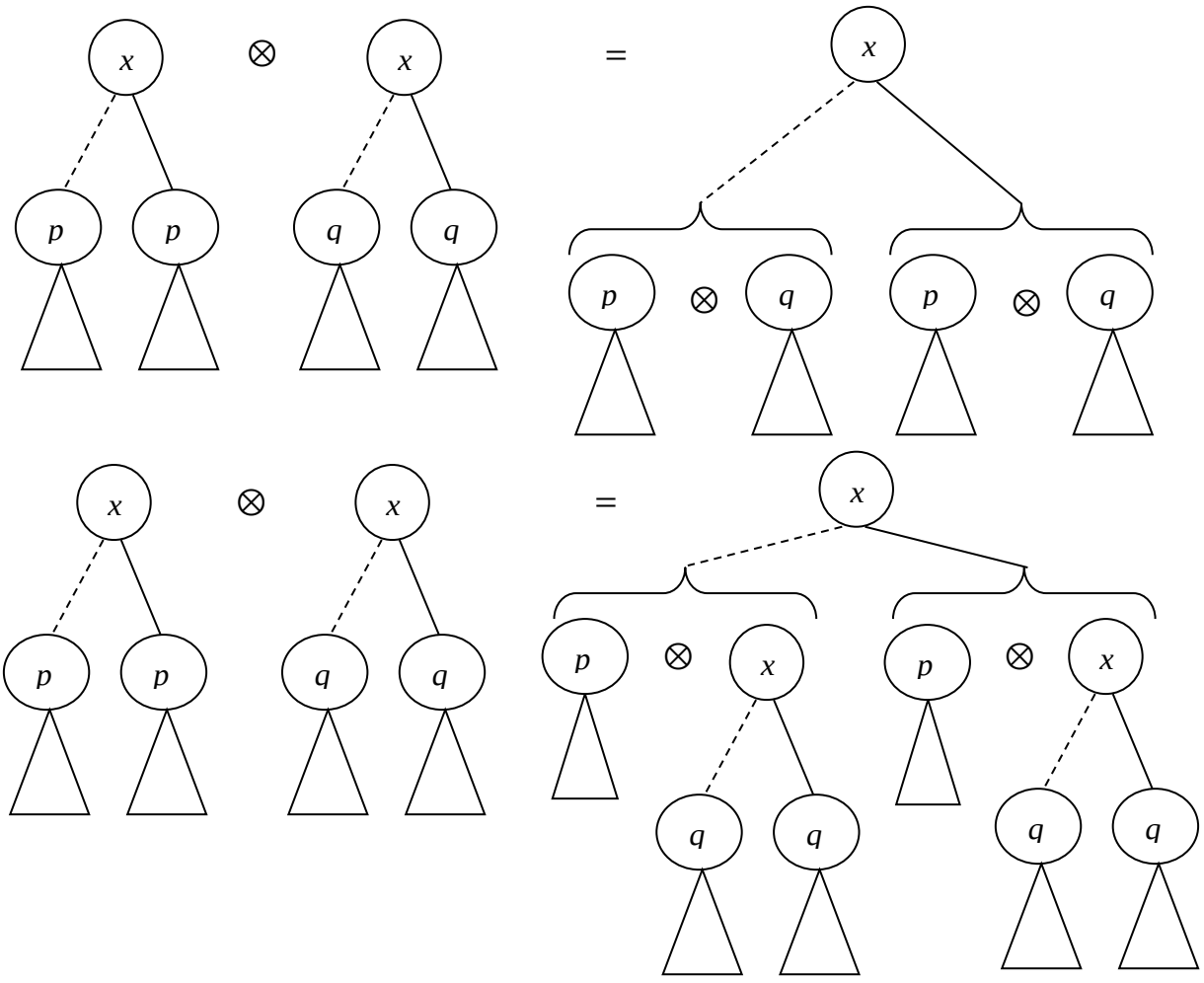


Рис. 7

РЕАЛИЗАЦИЯ BDD в БИБЛИОТЕКЕ BuDDy

Для представления BDD и выполнения операций над ними существует много различных библиотек. Мы рассмотрим свободно распространяемую библиотеку BuDDy для языка C++.

В настоящий момент версия BuDDy 2.4 доступна по адресу:

http://sourceforge.net/projects/buddy/

В пакет BuDDy входят исходный код библиотеки, документация и примеры решённых задач. Для использования библиотеки необходимо подключить скомпилированный файл *bdd.lib* к проекту. Объявления типов данных и функций для работы с ними находятся в заголовочном файле *bdd.h*.

Основным типом данных в библиотеке BuDDy является класс *bdd*. Для него перегружены следующие операторы:

- & (&=) – конъюнкция двух BDD (и соответственно с присваиванием);
- | (|=) – дизъюнкция;
- ^ (^=) – сложение по модулю 2;
- ! – отрицание;
- >> (>>=) – правосторонняя импликация;
- << (<<=) – левосторонняя импликация;
- – (– =) – следует понимать как разность характеризуемых множеств: a – b = a & !b, где a и b булевы функции;
- > – булева функция «больше»;
- < – булева функция «меньше»;
- == (!=) – функции сравнения двух BDD. Возвращают значения “истина” или “ложь” в виде целых 0 или 1;
- оператор потокового вывода <<.

  Последний может использоваться совместно с манипуляторами:
  - *bddset* – вывод интерпретаций, на которых булева функция, представленная в форме BDD, истинна (этот режим по-умолчанию);
  - *bddtable* – вывод значимых строк таблицы (см. рис. 6);
  - *bdddot* – вывод исходного кода программы *dot* пакета *graphviz*;
  - *bddall* – вывод всех строк таблицы.

Определены константы *bddtrue* и *bddfalse*.

Начинать работу с библиотекой следует с вызова функции

```
int bdd_init(int nodenum, int cachesize);
```

Она выделяет память для *nodenum* строк таблицы. При необходимости библиотека автоматически увеличит объём занимаемой памяти, на что, естественно, будет затрачено время.

В целях повышения быстродействия существует кэш для хранения результатов последних операций. Его статический размер устанавливается равным *cachesize*.

В документации к библиотеке рекомендуется использовать следующие значения описанных параметров:

- малые тестовые примеры: *nodenum* = 1000, *cachesize* = 100;
- задачи малой размерности: *nodenum* = 10000, *cachesize* =1000;
- задачи средней размерности: *nodenum* = 100000, *cachesize* = 10000;
- задачи высокой размерности: *nodenum* = 1000000, динамический кэш.

Для включения динамического кэша необходимо вызвать функцию

```
int bdd_setcacheratio(int cacheratio);
```

Здесь *cacheratio* – число строк таблицы, приходящееся на одно вхождение кэша.

Далее пользователь должен установить число булевых переменных, используемых для решения задачи:

```
int bdd_setvarnum(int N);
```

Следует отметить, что булева переменная представляет собой BDD простейшей структуры (рис. 8), поскольку эти две функции: f(x) = x и f(x) = !x.

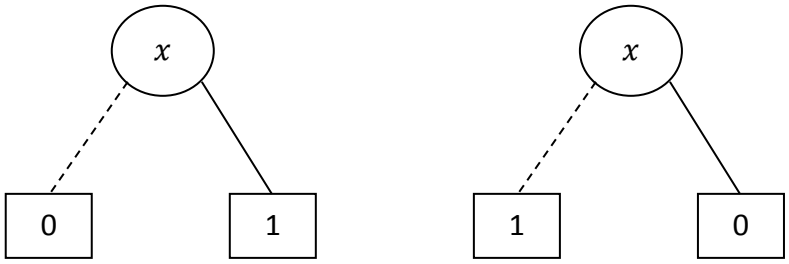


Рис. 8

Для обращения к *i*-й переменной ($i \in 0..N\text{-}1$) и её отрицанию необходимо воспользоваться следующими функциями соответственно:

```
bdd bdd_ithvar(int i);
bdd bdd_nithvar(int i);
```

Завершать работу с библиотекой следует вызовом функции

```
void bdd_done(void);
```

Рассмотрим выполнение различных операций над BDD на примере простой программы.

```
#pragma comment(lib, "bdd.lib") // подключение библиотеки
#include "bdd.h"                // включение заголовочного файла
```

```
#include <fstream>

using namespace std;

ofstream out;

void fun(char* varset, int size); // см. далее

void main(void)
{
	// Инициализация: начальное число вершин и статический кэш для операций
	bdd_init(1000, 100);

	bdd_setvarnum(6); // 6 булевых переменных
```

Возьмем для примера упомянутую ранее функцию $z = a_1b_1 \vee a_2b_2 \vee a_3b_3$. Установим для неё следующий порядок переменных: $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$.

```
	bdd a1 = bdd_ithvar(0);
	bdd a2 = bdd_ithvar(1);
	bdd a3 = bdd_ithvar(2);
	bdd b1 = bdd_ithvar(3);
	bdd b2 = bdd_ithvar(4);
	bdd b3 = bdd_ithvar(5);

	bdd z = a1&b1 | a2&b2 | a3&b3; // функция z(a1,a2,a3,b1,b2,b3)
```

Узнаем число вершин при таком порядке переменных при помощи функции *bdd_nodecount():*

```
	// число вершин BDD
	cout<<"\na1 < a2 < a3 < b1 < b2 < b3: "<<bdd_nodecount(z)<<"
nodes\n"<<endl;
```

На экран будет выведен следующий результат:

a1 < a2 < a3 < b1 < b2 < b3: 14 nodes

Построим графически BDD функции *z*, используя программу *dot*.

```
	out.open("bdd1.txt");
	out<<bdddot<<z<<endl; //вывод исходника dot
	out.close();
```

Результат обработки файла “bdd1.txt” на рис. 9 (слева).

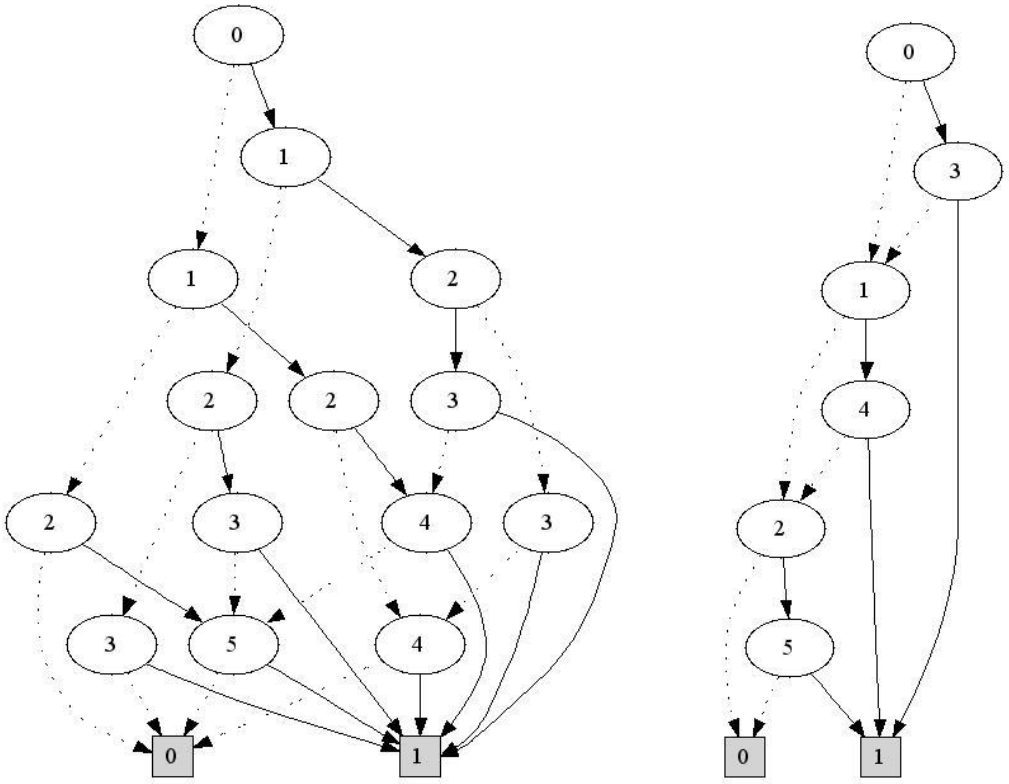


Рис. 9

Для построения BDD этой функции при порядке переменных $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$ выполним следующие операции:

```
	//меняем порядок переменных
	bdd_swapvar(1, 3);
	bdd_swapvar(2, 1);
	bdd_swapvar(2, 4);

	cout<<"\na1 < b1 < a2 < b2 < a3 < b3: "<<bdd_nodecount(z)<<"
nodes"<<endl;
	out.open("bdd2.txt");
	out<<bdddot<<z<<endl;
	out.close();
```

На экран будет выведено

a1 < b1 < a2 < b2 < a3 < b3: 6 nodes

Граф, закодированный в файле “bdd2.txt”, представлен на рис. 9 (справа). Напомним, что оба графа BDD представляют одну и ту же булеву функцию.

Выведем теперь все интерпретации, на которых функция z истинна, и количество этих интерпретаций (функция *bdd_satcount()*).

```
	//вывод множеств, на которых z = 1
	out.open("set.txt");
	out<<bdd_satcount(z)<<" sats:\n"<<endl;
	out<<bddset<<z<<endl;
```

Результат в файле “set.txt”:

```
37 sats:

<0:0, 1:0, 2:1, 5:1>
<0:0, 1:1, 4:0, 2:1, 5:1>
<0:0, 1:1, 4:1>
<0:1, 3:0, 1:0, 2:1, 5:1>
<0:1, 3:0, 1:1, 4:0, 2:1, 5:1>
<0:1, 3:0, 1:1, 4:1>
<0:1, 3:1>
```

Переменные в наборах выводятся как ‘*номер: значение’*. Если переменная с номером от 0 до *N* отсутствует, значит, она может быть равна как 0, так и 1 в совокупности с остальными переменными набора.

Часто возникает необходимость каким-либо образом обработать эти наборы. Для этого служит функция

void bdd_allsat(bdd r, bddallsathandler handler);

Здесь *handler* – указатель на функцию с прототипом

void fun(char varset, int size);*

Функция *bdd_allsat* для каждого такого набора выполняет функцию *fun*. *varset* – это массив элементов текущего набора. Причём, если *i*-я переменная божет быть как 0, так и 1, *varset*[*i*] == –1; *size* – длина *varset*.

Пусть функция *fun* реализована следующим образом:

```
void fun(char* varset, int size)
{
	for (int i = 0; i < size; i++) out<<(varset[i] < 0 ? 'X' : (char)('0' + varset[i]));
	out<<endl;
}
```

Выполним действия:

```
	out.open("allsat.txt");
	out<<bdd_satcount(z)<<" sats:\n"<<endl;
	bdd_allsat(z, fun);
	out.close();
```

Результат получим в файле “allsat.txt”:

```
	37 sats:

	001XX1
	011X01
	01XX1X
	1010X1
	111001
	11X01X
	1XX1XX
```

Чтобы получить BDD булевой функции, истинной только на одном из 37 множеств, можно воспользоваться функцией
bdd bdd_satone(bdd);

При реализации неявных символьных алгоритмов, где булевы функции выступают в роли характеристических функций конечных множеств и бинарных отношений над ними, часто возникает необходимость исключить некоторые переменные из функции (при переходе от бинарного отношения к множеству). Формально это соответствует операции «∃-квантификации»:
∃x f(x,y) = f(0,y) ∨ f(1, y)
Исключим из функции $z = a_1b_1 \vee a_2b_2 \vee a_3b_3$ переменные b_1, b_2 и b_3.

```
//exist-квантификация по b1, b2 и b3: z = a1 | a2 | a3
z = bdd_exist(z, b1&b2&b3);

out.open("bdd3.txt");
out<<bdddot<<z<<endl;
out.close();
```

Результат на рис. 10 (слева).

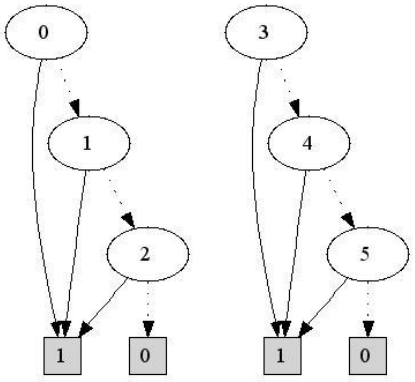


Рис. 10

$z = a_1 \vee a_2 \vee a_3$. Заменим (переименуем) переменные a_1, a_2 и a_3 на b_1, b_2 и b_3. Для этого воспользуемся следующим механизмом:

```
int A[] = {0, 1, 2}; // Будем менять переменные с номерами 0, 1, 2
int B[] = {3, 4, 5}; // на перемнные с номерами 3, 4, 5
bddPair* pair = bdd_newpair(); // инициализируем множество пар
bdd_setpairs(pair, A, B, 3);   // устанавливаем 3 пары: A[i] <-> B[i]
z = bdd_replace(z, pair); // заменяем переменные: z := b1 | b2 | b3
bdd_freepair(pair); // уничтожаем пары

out.open("bdd4.txt");
out<<bdddot<<z<<endl;
out.close();
```

Результат представлен на рис. 10 (справа).

Перед завершением работы выведем таблицу, представляющую итоговую функцию $z = b_1 \vee b_2 \vee b_3$:

```
        out.open("Table.txt");
        out<<bddtable<<z<<endl;
        out.close();

        bdd_done(); // завершаем работу с библиотекой
}
```

Файл “Table.txt”:

```
        ROOT: 23
        [   12]   5 :   0   1
        [   21]   4 :  12   1
        [   23]   3 :  21   1
```

Нами были рассмотрены основные функции библиотеки BuDDy, необходимость в использовании которых возникает при решении большинства задач. Остальные функции библиотеки описаны в документации.

ПРОГРАММИРОВАНИЕ В ОГРАНИЧЕНИЯХ

Приведенная задача является частным случаем широко известного класса задач программирования в ограничениях (Constraint Satisfaction Problem – CSP), в котором объекты должны удовлетворять некоторому набору ограничений. CSP отличаются высокой сложностью, в общем случае относится к классу NP-полных задач. При решении задач программирования в ограничениях применяют метод комбинаторного поиска на основе техник – распространение ограничения, локальный поиск и возврат.

Задача программирования в ограничениях – это тройка $\langle X, D, C \rangle$, где X – набор переменных, D – пространство доменов переменных, C – набор ограничений. Каждое ограничение – это пара $\langle t, R \rangle$, где t – кортеж переменных, R – набор кортежей переменных. Оценкой называется функция, действующая из пространства переменных в пространство доменов $v : X \to D$. Такие оценки удовлетворяют ограничению $\langle (x_1, \ldots, x_n), R \rangle$, если $(v(x_1), \ldots, v(x_n)) \in R$. Решение – это такая оценка, которая удовлетворяет всем ограничениям.

Приведенный выше подход, очевидно, позволяет решать любые задачи программирования в ограничениях (без применения техники возврата) при условии, что множества возможных значений параметров конечны.

ЗАДАЧА ЭЙНШТЕЙНА

Воспользуемся библиотекой BuDDy для решения головоломки, известной как «Задача Эйнштейна». Она формулируется следующим образом.

5 разных человек в 5 разных домах разного цвета, курят 5 разных марок сигарет, выращивают 5 разных видов животных, пьют 5 разных видов напитков. Известно следующее:

1. *Норвежец живет в первом доме.*
2. *Англичанин живет в красном доме.*
3. *Зеленый дом находится слева от белого.*
4. *Датчанин пьет чай.*
5. *Тот, кто курит Rothmans, живет рядом с тем, кто выращивает кошек.*
6. *Тот, кто живет в желтом доме, курит Dunhill.*
7. *Немец курит Marlboro.*
8. *Тот, кто живет в центре, пьет молоко.*
9. *Сосед того, кто курит Rothmans, пьет воду.*
10. *Тот, кто курит Pall Mall, выращивает птиц.*
11. *Швед выращивает собак.*
12. *Норвежец живет рядом с синим домом.*
13. *Тот, кто выращивает лошадей, живет в синем доме.*
14. *Тот, кто курит Philip Morris, пьет пиво.*

15. В зеленом доме пьют кофе.
Вопрос: Кто выращивает рыбок?

Очевидно, что для ответа на этот вопрос необходимо с каждым человеком сопоставить его национальность, марку сигарет, животное, напиток, цвет дома.

Формализуем задачу.

Пусть имеется система, состоящая из N объектов, обладающих M свойствами. Будем считать, что у двух различных объектов значения k-го свойства не совпадают. Следовательно, каждое свойство может принимать N различных значений.

В данном случае N = 5, M = 5. Объектами являются номера домов. Каждое из пяти свойств может принимать 5 различных значений из соответствующих множеств:

Свойство 1 – Цвет дома ∈ {красный, белый, зеленый, желтый, синий};
Свойство 2 – Национальность ∈ {англичанин, швед, датчанин, норвежец, немец};
Свойство 3 – Напиток ∈ {чай, молоко, пиво, вода, кофе};
Свойство 4 – Домашнее животное ∈ {кошка, лошадь, собака, птица, рыбка};
Свойство 5 – Сигареты ∈ {*Rothmans*, *Dunhill*, *Marlboro*, *Pall Mall*, *Philip Morris*}.

Поскольку множества значений свойств конечны, их можно закодировать. Пусть j – номер элемента множества значений свойства 1 ($j \in 0..N-1$), тогда закодируем это множество при помощи двоичных переменных. В данном случае каждое множество кодируется тремя переменными (например, *z0, z1, z2*):

Элементы множества свойства 1 (цвет)	Номер элемента множества свойства 1	Код элемента множества
красный	0	*!z0 ∧ !z1 ∧ !z2*
белый	1	*z0 ∧ !z1 ∧ !z2*
зеленый	2	*!z0 ∧ z1 ∧ !z2*
желтый	3	*z0 ∧ z1 ∧ !z2*
синий	4	*!z0 ∧ !z1 ∧ z2*

Т.е. зная номер элемента в множестве j, можно получить его код p при помощи младшего бита после оператора правого сдвига, например, для *j=4*:

(j >> 0) & 1 = 0	*p = !z0*
(j >> 1) & 1 = 0	*p = !z0 ∧ !z1*
(j >> 2) & 1 = 1	*p = !z0 ∧ !z1 ∧ z2*

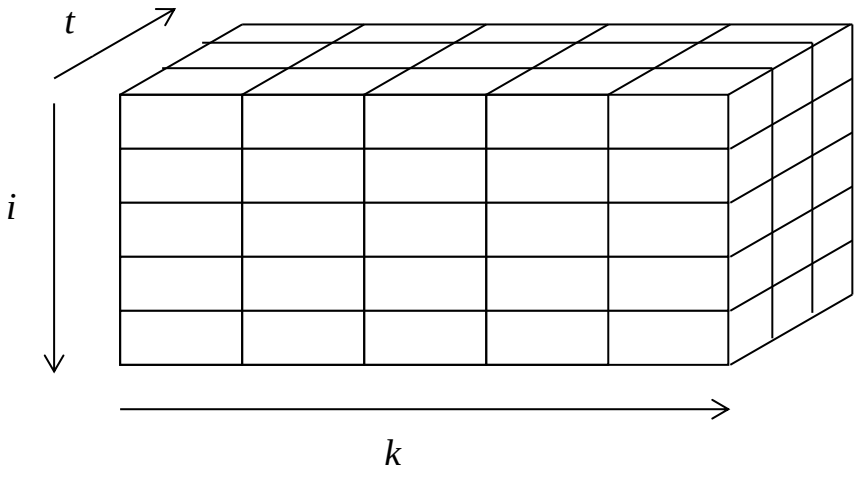
В результате код *j=4* будет *p = !z0 ∧ !z1 ∧ z2*.

Представим систему в виде таблицы:

Объект 1					Объект 2					…	Объект 5				
Свойство 1	Свойство 2	Свойство 3	Свойство 4	Свойство 5	Свойство 1	Свойство 2	Свойство 3	Свойство 4	Свойство 5	…	Свойство 1	Свойство 2	Свойство 3	Свойство 4	Свойство 5
000	000	000	000	000	000	000	000	000	000	…	000	000	000	000	000
000	000	000	000	000	000	000	000	000	000	…	000	000	000	000	001
• • •															
111	111	111	111	111	111	111	111	111	111	…	111	111	111	111	111

Данная таблица представляет собой все возможные состояния системы.

Пользуясь имеющимися сведениями, можно построить булеву функцию F от 75 переменных ($z_0 \dots z_{74}$), которая будет истинна на тех интерпретациях (двоичных наборах) из возможных 2^{75}, которые удовлетворяют ограничениям. Если задача имеет единственное решение, то функция F истинна на одном единственном наборе двоичных значений ее аргументов, если задача имеет множество решений, то таких интерпретаций, на которых F истинна, будет несколько. В случае противоречивых условий задача может не иметь решений совсем. В этом случае функция F невыполнима *(F = false)*.



Приведенный выше массив переменных *z* является трехмерным и каждый элемент массива *z[i,k,t]* находится в соответствующей ячейке, где индекс *i* соответствует номеру объекта и изменяется в диапазоне от *0* до *N-1*, индекс *k* соответствует номеру свойства и изменяется в диапазоне от *0* до *M-1*, а индекс *t* соответствует биту свойства и изменяется в диапазоне от *0* до *⌈logN⌉*.

Переведем эквивалентным образом трехмерный массив z в одномерный. Обозначим $\lceil logN \rceil = T$. Элементу трехмерного массива $z[i,j,t]$ будет соответствовать элемент одномерного массива $x[T*M*i+T*k+t]$.

Обозначим j – номер элемента множества значений некоторого свойства ($j \in 0..N-1$). Введем $p(k,i,j)$ функцию, подобно тому, как делали выше для свойства 1, получим код j элемента множества как представление в двоичной системе счисления:

$$p(k,i,j) = (\text{Свойство } k \text{ объекта } i \text{ имеет значение } j) = \bigwedge_{t=0}^{T} y_{TMi+Tk+t} \text{ , где}$$

$$y_{TMi+Tk+t} = \begin{cases} x_{TMi+Tk+t}, \text{если } t-\text{й бит } j \text{ равен 1} \\ \overline{x_{TMi+Tk+t}}, \text{если } t-\text{й бит } j \text{ равен 0} \end{cases}$$

Рассмотрим возможные типы ограничений:

1. Ограничение, устанавливающее свойству *k1* конкретного объекта *i1* значение *j1* («*Норвежец живет в первом доме*»):
   F := F & p(k1, i1, j1).

2. Ограничение, устанавливающее соответствие между двумя свойствами какого-либо объекта («*Англичанин живет в красном доме*»):
   F := F & (p(k1, i, j1) ⇔ p(k2, i, j2)) для всех *i* от 0 до N-1.

3. Ограничение расположения объектов – расположение «слева» («справа»). Это ограничения типа объект, у которого свойство *k1* имеет значение *j1*, расположен слева от объекта, у которого свойство *k2* имеет значение *j2*(*«Зеленый дом находится слева от белого»*):
   F := F & (p(k1, i, j1) ⇔ p(k2, i + 1, j2)) для всех *i* от 0 до N – 2.
   Очевидно, нужно добавить ограничение: объект, у которого свойство *k2* имеет значение *j2*, не будет первым, а объект, у которого свойство *k1* имеет значение *j1*, не будет последним.
   F := F & !p(k2, 0, j2) & !p(k1, N-1, j1).

4. Ограничение расположения объектов – расположение «рядом» – дизъюнкция ограничений «слева» и «справа».
   F := F & [(p(k1, i, j1) ⇔ p(k2, i + 1, j2)) ∨ (p(k2, i, j2) ⇔ p(k1, i + 1, j1))] для всех *i* от *0* до *N – 2*.

По умолчанию существуют еще следующие ограничения:

5. У двух различных объектов значения любого параметра (свойства) не совпадают.
   F := F & (p(k, i1, j) ⟹ !p(k, i2, j)) при *i1≠ i2*.

6. Параметры принимают значения только из заданных множеств (значение свойств должно быть меньше N).

Изначально F = *true*. Введя в любом порядке имеющиеся ограничения, получим искомую функцию F.

Пример частичного решения задания курсовой работы

Условие:

- выбрать систему, состоящую из N объектов, которые обладают M свойствами, принимающими N различных значений;
- задать n_1 ограничений типа 1, n_2 ограничений типа 2, n_3 ограничений типа 3 и n_4 ограничений типа 4 (типы ограничений возьмите из задачи Эйнштейна, $n_i > 0$);
- найти все возможные решения;
- придумать физическую интерпретацию задачи;
- если задача не имеет решений или же имеет не одно решение, следует убрать, добавить и/или изменить некоторые ограничения так, чтобы задача имела только одно единственное решение.

Для выполнения задания необходимо написать программу на языке C++ с использованием библиотеки BuDDy.

Решение задачи

$N = 3, M = 3, n_1 = 1, n_2 = 1, n_3 = 1, n_4 = 1$.

Для кодирования трёх разных значений параметров требуются две булевы переменные. Следовательно, для описания всего пространства решений необходимо 18 переменных.

Программа для предложенной системы:

```
#pragma comment(lib, "bdd.lib")
#include "bdd.h"
#include <fstream>

using namespace std;

#define N_VAR 18  // число булевых переменных
#define N 3             // число объектов
#define M 3             // число свойств
#define LOG_N 2

ofstream out;

void fun(char* varset, int size); // функция, используемая для вывода решений

void main(void)
{
      // инициализация
      bdd_init(10000, 1000);
      bdd_setvarnum(N_VAR);

      // ->--- вводим функцию p(k, i, j) следующим образом ( кодируем pk[i][j]
):
/     bdd p1[N][N];
      bdd p2[N][N];
      bdd p3[N][N];

      unsigned I = 0;
      for (unsigned i = 0; i < N; i++)
```

```
	{
		for (unsigned j = 0; j < N; j++)
		{
			p1[i][j] = bddtrue;
			for (unsigned k = 0; k < LOG_N; k++) p1[i][j] &= ((j >> k) &
1) ? bdd_ithvar(I + k) : bdd_nithvar(I + k) ;
			p2[i][j] = bddtrue;
			for (unsigned k = 0; k < LOG_N; k++) p2[i][j] &= ((j >> k) &
1) ? bdd_ithvar(I + LOG_N + k) : bdd_nithvar(I + LOG_N + k) ;
			p3[i][j] = bddtrue;
			for (unsigned k = 0; k < LOG_N; k++) p3[i][j] &= ((j >> k) &
1) ? bdd_ithvar(I + LOG_N*2 + k) : bdd_nithvar(I + LOG_N*2 + k) ;
		}
		I += LOG_N*M;
	}

	// здесь должны быть ограничения

	// вывод результатов
	out.open("out.txt");
	unsigned satcount = (unsigned)bdd_satcount(task);
	out<<satcount<<" solutions:\n"<<endl;
	if (satcount) bdd_allsat(task, fun);
	out.close();

	bdd_done(); // завершение работы библиотеки
}

// Ниже приведена реализация функций, управляющих выводом результатов.
// Рекомендуется самостоятельно с ними ознакомиться.
// В собственных заданиях следует использовать эти функции
// или придумать собственные.

char var[N_VAR];

void print(void)
{
	for (unsigned i = 0; i < N; i++)
	{
		out<<i<<": ";
		for (unsigned j = 0; j < N; j++)
		{
			unsigned J = i*N*LOG_N + j*LOG_N;
			unsigned num = 0;
			for (unsigned k = 0; k < LOG_N; k++) num += (unsigned)(var[J
+ k] << k);
			out<<num<<' ';
		}
		out<<endl;
	}
	out<<endl;
}
void build(char* varset, unsigned n, unsigned I)
{
	if (I == n - 1)
	{
		if (varset[I] >= 0)
		{
			var[I] = varset[I];
			print();
			return;
		}
		var[I] = 0;
		print();
```

```
            var[I] = 1;
            print();
            return;
      }
      if (varset[I] >= 0)
      {
            var[I] = varset[I];
            build(varset, n, I + 1);
            return;
      }
      var[I] = 0;
      build(varset, n, I + 1);
      var[I] = 1;
      build(varset, n, I + 1);
}

void fun(char* varset, int size)
{
      build(varset, size, 0);
}
```

Получаем следующий ответ:

5 solutions:

0: 0 0 2 // у 0-го объекта: свойство 1 имеет значение 0,
1: 1 2 0 // свойство 2 - значение 0, свойство 3 – значение 2; у 1-го
2: 2 1 1 // объекта: свойство 1 имеет значение 1, и т.д.

0: 0 0 1
1: 1 2 0
2: 2 1 2

0: 0 0 1
1: 1 1 2
2: 2 2 0

0: 0 2 0
1: 2 0 2
2: 1 1 1

0: 0 1 1
1: 2 0 2
2: 1 2 0

Интерпретация решения

Дедка, Жучка и внучка тянут репку. У всех разный цвет глаз (серые, зелёные, голубые). Известно, что герой либо капризный, либо имеет хвост, либо пенсионер.

У нас имеется три героя (номер в очереди к репке) и три свойства.

Предлагается закодировать домены свойств следующим образом:

a) Имя героя – дедка=0, Жучка=1, внучка=2;
b) Цвет глаз – сероглазый=0, зеленоглазый=1, голубоглазый=2;
c) Характер героя – капризный=0, хвостатый=1, пенсионер=2.

Ограничения из программы, представленной выше (циклы и ограничения на крайние значения пропущены.):

1. `task &= p1[0][0]` – свойство 1 у героя с 0 номером в очереди равно 0, или *дедка держится за репку*.
2. `task &= !( p2[i][2] ^ p3[i][0] )` – если у героя свойство 2 со значением 2, то у него свойство 3 имеет значение 0 и обратно, или *голубоглазый – капризный*.
3. `task &= !( p2[i][0] ^ p1[i + 1][1] )` – если у героя свойство 2 имеет значение 0, то он стоит до героя, у которого свойство 1 имеет значение 1, и если у героя свойство 1 имеет значение 1, то он стоит после героя, у которого свойство 2 имеет значение 0, или *Жучка держится за сероглазого*.

4. `temp1 &= !( p3[i][2] ^ p2[i + 1][2] ); temp2 &= !( p2[i][2] ^ p3[i + 1][2] ); task &= temp1 | temp2` – герой с значением 2 в свойстве 3 стоит до или после героя с значением 2 в свойстве 2, или *голубоглазый рядом с пенсионером*.

Интерпретируем первое решение

0: 0 0 1
1: 1 1 2
2: 2 2 0

Дедка с серыми глазами и хвостом держится за репку. Зеленоглазый пенсионер Жучка держится за дедку. За Жучку держится капризная голубоглазая внучка.

Как видно, ни одно из утверждений этого решения (и других найденных решений) не противоречит условиям задачи.

Для исключения остальных решений попробуем добавить еще одно условие:

1. *У внучки голубые глаза*.

```
for (unsigned i = 0; i < N; i++) task &= !( p1[i][2] ^ p2[i][2] );
```

Результат:

1 solutions:

0: 0 0 1
1: 1 1 2
2: 2 2 0

А сейчас предположим, что в условии задачи задано, что Жучка тоже имеет хвост. Это ограничение представится циклом:

```
for (unsigned i = 0; i < N; i++) task &= !( p1[i][1] ^ p3[i][1] );
```

Результат:

0 solutions:

Индивидуальное задание на курсовую работу:

- Пусть имеется N=9 объектов. Расположены объекты следующим образом:

- “Соседские” отношения между объектами определяются по индивидуальному варианту относительно центрального объекта.

	*	
*	0	*
	*	

Рис.1. Пример соседских отношений

Например, на рис. 1 соседями центрального объекта (помечен 0) будут объекты, расположенные выше, ниже, слева и справа (помечены *). Соответственно, понятие “объекты, расположенные рядом” – это соседи, а понятие “объекты, расположенные слева” – это соседи слева-сверху и слева-справа.

Для всех остальных объектов на рис.1 отношения строятся сдвигом. Так для объекта в правом углу для этих соседских отношений соседями будут:

	*	0
		*

- Необходимо выбрать $M=4$ свойств, принимающих N различных значений.
    - Задать n_1 ограничений типа 1, n_2 ограничений типа 2 (типы ограничений возьмите из задачи Эйнштейна, $n_i > 0$) по своему варианту.
    - Для ограничений типа 3 и типа 4 используйте ‘’соседские’’ отношения, заданные Вашим вариантом. Придумать, как ограничения, подобные типу 3, 4, выражаются в соседских отношениях Вашего варианта, задать n_3 ограничений подобных отношениям типа 3 и n_4 ограничений подобных отношениям типа 4 в соответствии со своим заданием.
- Необходимо самостоятельно описать дополнительный тип ограничения n_7: “сумма свойств объектов-соседей не должна быть больше K”, где K – некоторое число от 0 до $N*M$. K выбирается студентом.
- Найдите все возможные решения;
- Придумайте физическую интерпретацию задачи;
- Если задача имеет не одно решение, следует добавить и/или изменить некоторые ограничения так, чтобы задача имела только одно единственное решение.

- Если задача не имеет решений, следует удалить и/или изменить некоторые ограничения так, чтобы задача имела только одно единственное решение. Ограничение типа n_7 удалять нельзя.

Требования к реализации (если вы сдаете на компьютере преподавателя):

- Для выполнения задания необходимо написать программу на языке C++ с использованием библиотеки BuDDy.
- Компилировать под MS Visual Studio 2012.
- Реализация должна быть масштабируемой, т. е. изменение параметров N, M, K не должны приводить к изменению основной части программы.

Варианты для курсовой работы

Посчитайте свой номер варианта: из google-документа группы возьмите номер А со своей фамилией. (Остаток от деления А на 18) + 1 – ваш номер варианта. По этому номеру выбираете количество ограничений n1, n2, n3, n4 и соседские отношения.

1) n1 = 3, n2 = 2, n3 = 5, n4 = 5
2) n1 = 2, n2 = 3, n3 = 4, n4 = 5
3) n1 = 2, n2 = 5, n3 = 4, n4 = 4
4) n1 = 2, n2 = 4, n3 = 5, n4 = 4
5) n1 = 3, n2 = 4, n3 = 5, n4 = 3
6) n1 = 4, n2 = 5, n3 = 4, n4 = 2
7) n1 = 3, n2 = 6, n3 = 4, n4 = 2
8) n1 = 5, n2 = 6, n3 = 3, n4 = 1
9) n1 = 2, n2 = 6, n3 = 5, n4 = 3
10) n1 = 4, n2 = 2, n3 = 5, n4 = 5
11) n1 = 3, n2 = 3, n3 = 4, n4 = 5
12) n1 = 3, n2 = 5, n3 = 4, n4 = 4
13) n1 = 3, n2 = 4, n3 = 5, n4 = 4
14) n1 = 4, n2 = 4, n3 = 5, n4 = 3
15) n1 = 5, n2 = 5, n3 = 4, n4 = 2
16) n1 = 4, n2 = 6, n3 = 4, n4 = 2
17) n1 = 6, n2 = 6, n3 = 3, n4 = 1
18) n1 = 2, n2 = 6, n3 = 4, n4 = 3

Соседские отношения

1)		*	
	*	0	*
		*	
2)		*	
		0	*
3)			
	*	0	
		*	
4)	*		*
		0	
5)	*		
		0	
			*
6)			*
		0	
	*		

7)		*	
	*	0	
8)			
		0	*
		*	
9)			*
		0	
			*
10)	*		
		0	
	*		
11)			
		0	
	*		*
12)	*		
		0	
	*		
13)		*	
		0	
		*	
14)			
	*	0	*
15)			*
		0	*
16)			
	*	0	
	*		
17)	*	*	
		0	
18)			
		0	
		*	*