МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)
Институт информационных технологий математики и механики

Кафедра: Алгебры, геометрии и дискретной математики
Направление подготовки: «Фундаментальная информатика и информационные технологии»
Профиль подготовки: «Инженерия программного обеспечения»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Тема:
«Экспериментальная оценка ширины пустых симплексов, задаваемых системой с ограниченными минорами»

Выполнил: студент группы
381406-2 Доронин Роман Олегович

_____
Подпись

Научный руководитель:
к.ф.-м.н., старший преподаватель
Грибанов Дмитрий Владимирович

_____
Подпись

Нижний Новгород
2018

Содержание

Введение

В данной работе мы предлагаем полиномиальный алгоритм перебора пустых симплексов с фиксированным Δ и алгоритм подсчета ширины для них. В диссертации Грибанова Д. В. [1] с. 42 Теорема 1.2.4 указана оценка для ширины пустого симплекса. Наша задача – с помощью перебора, экспериментальным путем, проверить правильность данной оценки, оценить ее и проследить зависимость максимальной ширины симплекса от Δ.

Глава 1. ТЕОРИТИЧЕСКАЯ ОЦЕНКА ШИРИНЫ СИМПЛЕКСОВ

1.1 Терминология и обозначения

В данном разделе вводятся ряд понятий и обозначений, которые будут использоваться на протяжении всей дипломной работы.

Множества:

$\mathbb{N}$ - натуральных чисел

$\mathbb{Z}$ - целых чисел

$\mathbb{Q}$ - рациональных чисел

$\mathbb{R}$ - действительных чисел

Линейные пространства векторов размерности n над множеством:

$\mathbb{N}^n$ - натуральных чисел

$\mathbb{Z}^n$ - целых чисел

$\mathbb{Q}^n$ - рациональных чисел

$\mathbb{R}^n$ - действительных чисел

Матрицы:

Большая латинская буква обозначает матрицу, например A

Маленькая латинская буква обозначает вектор, например b

A^T - транспонированная матрица A

A^{-1} - обратная матрица A

A^{-T} - обратно транспонированная матрица A

Если $A \in \mathbb{F}^{nxm}$, где $\mathbb{F}$ – некоторое числовое множество, то

$cone(A) = \{x: Ax \leq b, b \in \mathbb{F}^n\}$ – конус, построенный на строках невырожденной матрицы A, область -мерного пространства, ограниченная n неравенствами.

Используем стандартные обозначения для классов сложности алгоритмов:

$$O\big(f(n)\big) = \{g(n): \exists c \in \mathbb{R}_+, \forall n \in \mathbb{N}, c|g(n)| \geq |g(n)|\}$$

Для обозначения минорной характеристики некоторой матрицы $A \in \mathbb{Z}^{nxm}$ нам потребуется ввести обозначение:
$\Delta(A) = \Delta_{rank(A)}(A)$ – максимальное абсолютное значение миноров рангового порядка матрицы A

1.2 Нормальная форма Эрмита

Теорема 1.2.1 *Любую матрицу $A \in \mathbb{Q}^{mxn}$ ранга $r = rank(A)$ можно представить в виде произведения $A = HU$, где матрица $U \in \mathbb{Z}^{nxn}$ является унимодулярной, а матрица $H \in \mathbb{Q}^{mxn}$, называется нормальной формой Эрмита, имеет следующий вид*

$$\begin{pmatrix} H_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ H_{n1} & \cdots & H_{nn} \end{pmatrix}$$

Для элементов Hс номерами $i, j \in \{1,2, \dots, r\}$ верно следующее: $H_{ii} > 0, 0 \leq H_{ij} < H_{ii}$ для $i < j$. Для остальных элементов матрицы H данные свойства могут быть не верны. Если исходная матрица A целочисленная, то и матрица H является целочисленной.

Теорема 1.2.2 *Нормальная форма Эрмита H матрицы A является инвариантом решетки* $\Lambda(A)$ *и, таким образом, самой матрицы A. Иными словами, пусть* $H(A)$ *и* $H(B)$ *- нормальные формы Эрмита для матрицы A и B. Тогда* $\Lambda(A) = \Lambda(B)$ *тогда и только тогда, когда* $H(A) = H(B)$.

1.3 Ширина симплекса

Ширина симплекса S по направлению $c \in \mathbb{Z}^n \backslash \{0\}$

$$width_c(S) = \max_{x \in S} c^T x - \min_{x \in S} c^T x$$

Ширина симплекса S

$$width(S) = \min_{c \in Z^n \backslash \{0\}} width_c(S)$$

Интересно, что для симплекса S величина $width(S)$ может быть не определена, так как по например одному направлению $c \in \mathbb{Z}^n \backslash \{0\}$ один из функционалов $c^T x$ или $-c^T x$ может быть не ограничен. При вычислении ширины симплексов такой случай имеет место быть.

1.4 Теорема Хинчина

Пусть P – выпуклое тело в $\mathbb{R}^n$. Теорема Хинчина говорит, что если $P \cap \mathbb{Z} = \emptyset$ и величина $width(P)$ определена, то $width(P) \leq f(n)$, где величина $f(n)$ зависит от размерности.

Имеется множество оценок на величину $f(n)$. В следующей таблице 1 приведена история улучшения оценки $f(n)$.

Таблица 1.

Оценки на величину $f(n)$

Author	$f(n)$ upper bound
Khinchine'48	$O((n+1)!)$
Babai'86	$O(2^{O(n)})$
Lenstra-Lagarias-Schnorr'87	$O(n^{5/2})$
Kannan-Lovasz'88	$O(n^2)$
Banaszczyk et al'99	$O(n^{3/2})$
Rudelson'00	$O(n^{4/3}(1+\log n)^c)$

Для симплексов наилучшей, на текущий момент, оценкой является оценка $f(n) = O(n \log n)$.

В дальнейшем нас будет интересовать только случай, когда выпуклое тело P является полиэдром $P = P(A, b)$ для некоторой целочисленной матрицы A и некоторого целочисленного вектора b. Оказывается, если матрица A является ограниченной, то $f(n)$ можно оценить как $f(n) \leq C_k n$, где C_k – константа, зависящая только от k. Более того, если $width(P) > C_k n$, то целая точка внутри P может быть найдена за полиномиальное время. Заметим, что многие известные результаты не дают алгоритма, а тем более эффективного, для поиска целой точки внутри P в случае нарушения условия $width(P) \leq f(n)$.

Если же полиэдр P является симплексом, то в оценке на величину $f(n)$ можно избавиться от явной зависимости от размерности, т.е. оценка приобретает вид $f(n) \leq k$, где k определяется минорными характеристиками матрицы ограничений P.

1.5 Аналог теоремы Хинчина для симплексов

Теорема 1.5.1 *Пусть* $A \in \mathbb{Z}^{(n+1)\,x\,n}, b \in \mathbb{Z}^n$ *и* $P = P(A, b)$ *– симплекс. Если* $width(P) \geq \Delta(A) - 1$*, то* $P \cap \mathbb{Z}^n \neq \emptyset$*. Существует полиномиальный алгоритм, вычисляющий некоторую точку множества* $P \cap \mathbb{Z}^n$.

Здесь говориться о том, что при превышении шириной, конкретной оценки, внутри симплекса найдется целая точка и ее нахождение может быть выполнено на полиномиальное время. Данную оценку для ширины пустых симплексов, мы как раз и будем экспериментально проверять в нашей работе. На сколько она окажется близка или далека, или может совсем не соответствует действительности.

Глава 2. ЭКСПЕРЕМЕНТАЛЬНОЕ ВЫЧИСЛЕНИЕ ОЦЕНКИ ШИРИНЫ ПУСТЫХ СИМПЛЕКСОВ, ЗАДАННЫХ СИСТЕМОЙ С ОГРАНИЧЕННЫМИ МИНОРАМИ

1.1 Постановка задачи

Для заданных входных параметров n – размерность пространства и Δ - максимальный минор рангового порядка, перебрать всевозможные симплексы. У каждого симплекса вычислить ширину. Оценить результаты, вывести экспериментальную оценку для ширины. Сравнить ее с теоретической оценкой и сделать соответствующие выводы по результатам.

1.2 Перебор всех конусов, заданных системой неравенств в форме Эрмита

$$Hx \le b \quad \begin{pmatrix} H_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ H_{n1} & \cdots & H_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \le \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

$$\forall i,j \qquad H_{ii} > 0, \qquad 0 \le H_{ij} < H_{ii}, \qquad j < i$$

Перебор начинается с главной диагонали, поскольку она определяет область значений для остальных элементов из нижнего треугольника и вектора b. По определению они не могут превышать значения на главной диагонали , так же, элементы на главной диагонали обязаны идти в порядке возрастания.

С помощью рекурсивного алгоритма в три этапа перебираются оставшиеся элементы.

1.3 Перебор всевозможных «крышек»

На предыдущем шаге мы получили n неравенств образующих конус в n-мерном пространстве. Чтобы получить симплекс требуется еще одна грань, еще одно неравенство, так называемая крышка, обозначим ее как $cov \in \mathbb{Z}^n$.

Взяв транспонированную матрицу H и рассмотрев параллелепипед, который образуют получившиеся вектора из строк матрицы H^T, каждая целая точка внутри будет соответствовать крышке, при чем, таких точек будет ровно Δ.

1.3.1 Перебор целых точек внутри параллелепипеда малого объема

Рассмотрим систему, описывающую множество точек занимающих параллелепипед.

$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} H_{11}^T & \cdots & H_{1n}^T \\ \vdots & \ddots & \vdots \\ 0 & \cdots & H_{nn}^T \end{pmatrix} \begin{pmatrix} t_1 \\ \vdots \\ t_n \end{pmatrix}$$

На t_i наложено ограничение: (0,1]. В противном случае полученная точка x будет лежать за границей параллелепипеда. Зная, что нижний треугольник матрицы H^T нулевой, из этой системы можно сразу найти t_n.

$$t_n = \frac{x_n}{H_{nn}^T}$$

Поскольку, рассматриваются только целые точки, то перебираем x_n до тех пор, пока t_n не станет больше единицы. При этом с каждым новым x_n и t_n решаем систему дальше, полностью получая решение. Которое и будет являться нашей крышкой.

Поскольку каждая крышка представляет неравенство, остается еще вычислить число c_0: $x * cov \leq c_0$. Находим координату v вершины, на которую опирается конус, через систему уравнений:

$$Hv = b$$

Далее считаем $\partial = \sum v_i * cov_i$, если ∂ – целая то $c_0 = \partial + 1$, в противном случае $c_0 = \partial$. Данному c_0 будет соответствовать минимальная крышка, которую можно задать в целых числах.

1.4 Поиск максимального пустого симплекса с известной крышкой

Имея симплекс, заданного в форме Эрмита и известную крышку, можем вычислить минимальное положение крышки при прохождении ей целой точки внутри конуса. Такой симплекс будет являться максимальным пустым симплексом. Решается эта задачи путем решения задачи целочисленного линейного программирования, в дальнейшем ЦЛП, на минимизацию, где ограничениями выступает конус, а целевой функцией крышка.

$$\min cov * x$$

$$\begin{cases} Hx \leq b \\ x \in Z^n \backslash \{0\} \end{cases}$$

Для решения задачи ЦЛП в нашей выпускной работе мы будем использовать библиотеку CPLEX с API для C++. CPLEX – пакет программного обеспечения, предназначенный для решения задачи линейного и целочисленного программирования.

1.5 Подсчет ширины симплекса

Напомним, под шириной симплекса S по направлению $c \in Z^n \backslash \{0\}$ понимается величина

$$width_c(S) = \max_{x \in S} c^T x - \min_{x \in S} c^T x$$

Под шириной симплекса S понимается величина

$$width(S) = \min_{c \in Z^n \backslash \{0\}} width_c(S)$$

Рассмотрим J – подмножество строк матрицы H, A_J – подматрица со строками из J. Тогда система:

$$A_{J_i} x = b_{J_i}$$

определяет вершину.

Рассмотрим нормальный конус, опирающийся на вершину

$$N(V_{J_i}) = cone(A_{J_i}^T) = \{x: A_{J_i}^{-T} x \geq 0\}$$
$$-N(V_{J_i}) = -cone(A_{J_i}^T) = \{x: A_{J_i}^{-T} x \leq 0\}$$

Рассматривая все пары вершин (i, j) можно свести задачу поиска ширины к задаче ЛП. Конусы будут отвечать за направления, которые попадут в вершины. Если x - направление:

$$\min x^T (V_i - V_j)$$
$$\begin{cases} x \in N(V_i) \\ x \in -N(V_j) \\ x \in Z^n \backslash \{0\} \end{cases}$$

Таких пар будет ровно n(n-1). Последнее условие нужно для того, чтобы отсечь $\{0\}$, в противном случае алгоритм будет постоянно «сваливаться» в него. Чтобы выполнить последнее условие, не включать ноль, нужно ввести дополнительное неравенство, отсекающее вершину

$$wx \leq -1 \text{ где } w = \sum A_{V_i}^{-T}$$

как бы поднимаю снизу некую крышку. Приводя задачу к неравенствам получаем

$$\min(V_i - V_j)\, x$$
$$\begin{cases} A_{V_i}^{-T} x^T \geq 0 \\ A_{V_j}^{-T} x^T \leq 0 \\ x \in Z^n \backslash \{0\} \\ wx \leq -1 \end{cases}$$

Минимальное значение ширины из полученных и будет являться значение ширины для всего симплекса. Так же в процессе можно вычислить направление, по которому берется минимум и минимальный минор, на котором этот минимум будет достигаться.

Глава 3. РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ

Экспериментальная оценка ширины симплексов

Num of cones – количество симплексов, которое было построено

Time – время работы

Max width – максимальная ширина

Min determ – минимальный минор, на котором достигается максимальное значение ширины

Average width – среднее значение ширины

Таблица 1.

Результаты экспериментов

MAX	DELTA	3	4	5	6	7
N						
2	Num of cones	14	38	63	117	166
	Time	0,863	2,68	4,88	9,845	16,025
	Max width	0,75	0,75	0,75	0,666667	0,666667
	Min determ	2	2	4	3	3
	Average width	0,497059	0,41018	0,372984	0,327389	0,30693
3	Num of cones	36	132	257	581	924
	Time	2,996	11,373	29,619	71,397	126,133
	Max width	0,666667	0,666667	0,666667	0,666667	0,688312
	Min determ	2	2	2	2	7
	Average width	0,438285	0,348858	0,305623	0,26115	0,238477
4	Num of cones	98	482	1107	3051	5452
	Time	9,298	59,16	181,369	700,353	1261,47
	Max width	0,625	0,625	0,625	0,625	0,688312
	Min determ	2	2	2	2	7
	Average width	0,40831	0,318982	0,273843	0,230678	0,208033
5	Num of cones	276	1812			
	Time	36,734	858,754			
	Max width	0,6	0,6			
	Min determ	2	2			
	Average	0,3892	0,3014			

	width					
6	Num of cones Time Max width Min determ Average width	794 2622,44 0,583333 2 0,37671				

Заключение

Нами был разработан алгоритм перебора пустых (внутри нет целых точек) симплексов, который полиноминален по выходу.
Реализована программа, которая принимает на вход n – размерность пространства и Δ - максимальный минор рангового порядка, перебирает все симплексы и так же считает ширину.
Из эксперимента выяснилось, что оценка на ширину пустых симплексов оказалась сильно завышена. Но это так же может и служить сигналом, что разработанный алгоритм еще требует доработки, возможно он не покрывает все множество, необходимых для оценки, симплексов.
Сильной зависимости оценки ширины от Δ не наблюдается, но она все таки присутствует, требуется более обширная выборка для проведения экспериментов и соответствующие мощности, поскольку с увеличение параметров сложность растет в геометрической прогрессии.

Список литературы

[1] Грибанов, Д. В. Исследование задач целочисленной линейной оптимизации с ограниченным спектром миноров: дис…канд. физ-мат. наук: 2016. – 87 с.

[2] Gribanov D.V., Chirkov A.J. The width and integer optimization on simplices with bounded minors of the constraint matrices // Optimization Letters. – 2016. – V. 10, No 6. – P. 1179-1189.

[3] Gribanov D.V., Veselov S.I. – On integer programming with bounded determinats // Optimization Letters. – 2016. – V. 10, No 6. – P. 1169-1177.

[4] А. Схвейвер – Теория линейного и целочисленного программирования: В 2-х т. Т. 2: Пер. англ. – М.: Мир, 1991. – 342 с., ил.

[5] Гюнтер М. Циглер – Теория многогранников / Пер. с англ. под ред. Н.П.Долбилина. – М.:МЦРМО, 2014. –586 с.

Приложение

Код программы

```
#define _CRT_SECURE_NO_WARNINGS // Для исправления ошибки [c4996 deprecate]


#include <iostream>

#include <ctime>

#include <cstdlib>

#include <ilcplex/ilocplex.h> // IBM ILOG CPLEX

#include <vector>



/*-------------------------------------------------*/

// Директивы

// Префикс SMP - это значит директивы относятся к Симплексу, MIP - к решателю ЦЛП CPLEX


//#define SMP_DEBUG

//#define SMP_OLD_CODE

#define SMP_INIT_MODE_0 0

#define SMP_INIT_MODE_1 1


//#define TEST_MOD
```

```
#define FILE_MODE


#ifdef FILE_MODE
#include <fstream>
#include <iomanip>
#endif


#define NOT_WORKING -1


// Цель целевой функции
enum Target {
	MIP_MAX,
	MIP_MIN
};


// Дополнительные ограничения для MIP
enum AdditRestr {
	MIP_NONE_REST,
	MIP_XG0, // X greater than zero
	MIP_XL0 // X less than zero
};


enum MipSwitch {
	MIP_ON,
```

```
        MIP_OFF
};

/*-------------------------------------------------*/
// Макросы

#define SMP_NEW(mas, n, type) mas = new type*[n]; \
for (int newCount = 0; newCount < n; newCount++) \
        mas[newCount] = new type[n]

#define SMP_DELETE(mas, n) for (int deleteCount = 0; deleteCount < n; deleteCount++) \
delete[] mas[deleteCount]

#define SMP_COMPARE(val1, val2) if (val1 == val2) std::cout << "[OK] \n"; \
else std::cout << "[FAILED] \n"

/*-------------------------------------------------*/
// Прототипы функций и классов
class Simplex;
bool SimplexIsEmpty(Simplex *S);
void GetAllCovers(Simplex *S, int N);
void GetMatr(int **mas, int **p, int i, int j, int m);
int Determinant(int **mas, int m);
int** AttachedMat(int **H, int n);
```

```
double** MatIntToDouble(int **A, int n);

double* VecIntToDouble(int *A, int n);

double* Gauss(double **a, double *y, int n);


/*-----------------------------------------------------*/

// Возведение числа в степень

template <typename T>

T Expon(T num, int deg)

{

	T res = 1;

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

		res *= num;


	return res;

}


// Поск обратной транспонированной матрицы, точней не совсем так:)

int** FindAttachedMatrix(int** Mat, int n)

{

	int **AttachMat;

	SMP_NEW(AttachMat, n, int);


	int det = Determinant(Mat, n);
```

```
	for(int i = 0; i < n; i++)
		for (int j = 0; j < n; j++)
		{
			int **subMat;
			SMP_NEW(subMat, n - 1, int);
			GetMatr(Mat, subMat, i, j, n);

			AttachMat[i][j] = Determinant(subMat, n - 1) * Expon(-1, i + j);

			if (det < 0) AttachMat[i][j] *= -1;

			SMP_DELETE(subMat, n - 1);
		}

	return AttachMat;
}

// CPLEX - MIP [ObjFunc->Trg, ExprMat<=b]
template < typename T >
void CPLEX_MIP(T *result, MipSwitch MpiS, int **ExprMat, int *b, int Size, int exprSize, T *ObjFunc, Target Trg, AdditRestr Ar = MIP_NONE_REST)
{

	IloEnv env; // Среда
```

```
std::stringstream logfile;


try {
	// lowerBound и upperBound ограничения на X
	int upperBound = INT_MAX, lowerBound = -INT_MAX;
	if (Ar == MIP_XG0)
		lowerBound = 0;
	else if (Ar == MIP_XL0)
		upperBound = 0;


	IloNumVar::Type t;
	if (MpiS == MIP_ON)
		t = ILOINT;
	else
		t = ILOFLOAT;
	IloNumVarArray x(env, Size, lowerBound, upperBound, t);


	IloModel model(env); // Модель
	IloCplex cplex(env);
	cplex.setOut(env.getNullStream());


	cplex.setParam(IloCplex::Threads, 1);
	//cplex.setParam(IloCplex::WorkMem, 1024);
	cplex.setParam(IloCplex::TreLim, 2048);
```

```
cplex.setParam(IloCplex::Param::Parallel, 1);


cplex.extract(model);


IloNumVarArray startVar(env); // Переменные
IloNumArray startVal(env); // Значения переменных


for (int i = 0; i < Size; i++) {
        startVar.add(x[i]);
        startVal.add(0);
}


// Целевая функция
IloExpr obj(env);
for (int i = 0; i < Size; i++)
        obj += ObjFunc[i] * startVar[i];
if (Trg == MIP_MAX)
        model.add(IloMaximize(env, obj));
else if (Trg == MIP_MIN)
        model.add(IloMinimize(env, obj));
obj.end();


// Ограничения
for (int i = 0; i < exprSize; i++) {
```

```
                IloExpr expr(env);
                for (int j = 0; j < Size; j++)
                        expr += ExprMat[i][j] * startVar[j];
                model.add(expr <= b[i]);
                expr.end();
        }

        if (MpiS == MIP_ON) {
                // Запуск MIP mode
                cplex.addMIPStart(startVar, startVal);
        }


        // Solve
        cplex.solve();
        if (cplex.getStatus() == IloAlgorithm::Optimal) {
                cplex.getValues(startVal, startVar);
#ifdef TEST_MOD
                env.out() << "Values = " << startVal << std::endl;
#endif
        }

        // Переводим результат из IloNumVarArray в Int
        const IloArray<double> &a = (IloArray<double> &)startVal;
```

```
		for (int i = 0; i < Size; i++)
			result[i] = a[i];


		startVal.end();
		startVar.end();
	}
	catch (IloException& e) {
		std::cerr << "C-Exp: " << e << std::endl;
	}
	catch (...) {
		std::cerr << "Unknown Exception" << std::endl;
	}

	//const std::string str = logfile.str();
	//std::cout << str << std::endl;
	env.end();
}

// n - размерность, m - высота | Cone1 разворачивем, домножив на -1 (*)
void SetAndUniteExpr(int **exprMat, int n, int **cone1, int **cone2)
{
	for (int j = 0; j < n; j++)
		exprMat[2 * n][j] = 0;
```

```
	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++) {
			exprMat[i][j] = cone1[i][j] * (-1); // (*)
			exprMat[2 * n][j] += exprMat[i][j];
		}

	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++)
			exprMat[i + n][j] = cone2[i][j];
}

/*-------------------------------------------------*/

// Класс симплекса [ Ax <= b ] -> [ AQx <= b ] -> [ Hx <= b ] (Форма Эрмита)
class Simplex {
public:
	int **H;		// Матрица H
	int *b;		// Вектор b

	int Multip;	// Кратность
	int n;		// Размерность

					// Крышка c (cTx <= c0)
```

```
        int *cover;
        int c0;

public:
        // Конструктор
        Simplex(int _n, int _M) : n(_n), Multip(_M)
        {
                b = new int[n];
                cover = new int[n];
                SMP_NEW(H, n, int);

                InitDef();
        }

        // Деструктор
        ~Simplex()
        {
                delete[] b;
                delete[] cover;
                SMP_DELETE(H, n);
        }

        // Возвращает матрицу с замененной строкой
        int** ReplaceMatrixString(int rI)
```

```
{
	int **Tm;
	SMP_NEW(Tm, n, int);

	for (int i = 0; i < n; i++)
		if (i == rI)
			for (int j = 0; j < n; j++)
				Tm[i][j] = cover[j];
		else
			for (int j = 0; j < n; j++)
				Tm[i][j] = H[i][j];

	return Tm;
}

//
int* ReplaceVectorString(int rI)
{
	int *Tv = new int[n];

	for (int i = 0; i < n; i++)
		if (i == rI)
			Tv[i] = c0;
		else
```

```
                Tv[i] = b[i];


        return Tv;
    }


    /*-------------------------------------------------*/
    // Методы для работы с крышкой


#ifdef SMP_OLD_CODE
    // Рекурсивный перебор целых точек в n-мерном параллелепипиде, составленный из строк матрицы H
    void EnumerationIntPointsInParal(int m, int *C)
    {
        m++; // Параметр отвечающий за остановку рекурсии, по сути считает чтобы мы мы не вышли за n
             // Перебор всех целых точек
        if (m != n)
            for (int i = 0; i <= ColomnSum(m); i++)
            {
                C[m] = i;
                EnumerationIntPointsInParal(m, C);
            }
        else
        { // Провеверка точек на пренадлежность паралелограмму из строк H
```

```
double *t = new double[n];
t = Gauss(MatIntToDouble(Transposition(), n), VecIntToDouble(C, n), n);

// Проверяем что t[i] принадлежат (0,1]
for (int i = 0; i < n; i++)
	if (t[i] <= 0 || t[i] > 1) {
		delete[] t;
		return;
	}

delete[] t;

// Если прошло, значит выполнено 1-ое УСЛОВИЕ

for (int i = 0; i < n; i++)
	cover[i] = -C[i];

if (!CheckDeterminants()) // Проверка 2-ого УСЛОВИЯ
	return;

FindC0(); // Поск элемента c0

		//!
std::cout << std::endl << "A suitable cover: ( ";
```

```
                for (int i = 0; i < n; i++)
                        std::cout << cover[i] << " ";
                std::cout << ") | (" << c0 << ")" << std::endl;
        }



}

// Возвращает сумму элементов одного столбца
int ColomnSum(int j)
{
        int sum = 0;
        for (int i = 0; i < n; i++)
                sum += H[i][j];

        return sum;
}

// Перечисление целых точек !!! Пока без отрицательных чисел
void EnumerationIntDot()
{
        int det = Determinant(H, n);
        std::cout << std::endl << "Det(H) = " << det;
        std::cout << std::endl << "Number of integer dots: " << det << std::endl;
```

```
		int *C = new int[n];
		for (int i = 0; i < n; i++)
			C[i] = 0;

		// Начало перебора целых точек в параллелепипеде
		for (int i = 0; i <= ColomnSum(0); i++) // ColomnSum(0) - Высота параллелепипеда по первой оси
		{
			C[0] = i; // Проверяемая точка на принадлежность к параллелепипеду
			EnumerationIntPointsInParal(0, C);
		}

		delete[] C;
	}
#endif

	// Функция считающая сумму t[i]*Ht[m] вправа от Ht[m][m]
	double Tau(int **Ht, double *t, int m)
	{
		double sum = 0;
		for (int i = m + 1; i < n; i++)
			sum += Ht[m][i] * t[i];
```

```
	return sum;
}

// Перечисление целых точек ПРАВИЛЬНАЯ реализация. Только для положительных коэффициентов H
// t - вектор условных переменных по которым проверяются x
// m - степень погружения, переменная для остановки рекурсии
// x - искомый вектор крышек
void EnumIntDot(int **Ht, double *t, int m, int *x)
{
	double tauVal = Tau(Ht, t, m);
	int res = ceil(tauVal);
	if (ceil(tauVal) == tauVal)
		res++;

	t[m] = (res - tauVal) / Ht[m][m];

	while (t[m] <= 1)
	{
		x[m] = res;
		if (m != 0)
			EnumIntDot(Ht, t, m - 1, x);
		else
		{
```

```
                    /* Крышка сгенерирована */
                    for (int i = 0; i < n; i++)
                        cover[i] = -x[i];

                    if (!CheckDeterminants()) // Проверка 2-ого УСЛОВИЯ
                    {
                        std::cout << std::endl << "FAILED: CheckDeterminants()";
                        return;
                    }

                    FindC0(); // Поск элемента c0
#ifdef TEST_MOD
                    std::cout /*<< std::endl*/ << "A suitable cover: ( ";
                    for (int i = 0; i < n; i++)
                        std::cout << cover[i] << " ";
                    std::cout << ") | (" << c0 << ")" << std::endl;
#endif
#ifdef FILE_MODE
                    std::ofstream fout("res_max_N6_D4.txt", std::ios_base::app);
                    fout << "A suitable cover: ( " << std::endl;
                    for (int i = 0; i < n; i++)
                        fout << cover[i] << " ";
                    fout << ") | (" << c0 << ")" << std::endl;
#endif
```

```
                    // Проверка на пустоту
#ifdef TEST_MOD
                    if (SimplexIsEmpty(this))
                        std::cout << "Void check : Empty" << std::endl;
                    else
                        std::cout << "Void check : Not Empty" << std::endl;
#endif

#ifdef FILE_MODE
                    if (SimplexIsEmpty(this))
                        fout << "Void check : Empty" << std::endl;
                    else
                        fout << "Void check : Not Empty" << std::endl;
#endif

                    // Вычисление ширины
#ifdef TEST_MOD
                    std::cout << "Width: " << CalcSimplexWidth() << std::endl;
#endif
#ifdef FILE_MODE
                    fout << "Width: " << CalcSimplexWidth() << std::endl;
                    fout.close();
#endif
```

```
                }


                res++;
                t[m] = (res - tauVal) / Ht[m][m];
        }
}

// Проверка определителей с замененной строкой | 2-ое УСЛОВИЕ
bool CheckDeterminants()
{
        int **Tm;
        SMP_NEW(Tm, n, int);

        for (int l = 0; l < n; l++) // Цикл строки которую заменяем
        {
                for (int i = 0; i < n; i++)
                        for (int j = 0; j < n; j++)
                                if (i == l)
                                        Tm[i][j] = cover[j];
                                else
                                        Tm[i][j] = H[i][j];

                if (abs(Determinant(Tm, n)) > Multip - 1)
                {
```

```
                SMP_DELETE(Tm, n);
                return false;
            }
        }

        SMP_DELETE(Tm, n);
        return true;
    }

    // cT состоит из целых чисел | 3-е УСЛОВИЕ

    // Функция поиска c0
    void FindC0()
    {
        double *V = Gauss(MatIntToDouble(H, n), VecIntToDouble(b, n), n); // Вершина конуса
        double delt = 0;

        for (int i = 0; i < n; i++)
            delt += V[i] * cover[i];

        if (ceil(delt) != delt)
            c0 = ceil(delt);
        else
```

```
                c0 = ceil(delt) + 1;
        }

        /*-------------------------------------------------*/

        // Инициализация еденицами и нулями
        void InitDef(int Mode = SMP_INIT_MODE_0) // Mode 0 - с инициализацией главной диагонали и вектора b, Mode 1 без
        {
                for (int i = 0; i < n; i++)
                {
                        if (Mode == SMP_INIT_MODE_0)
                        {
                                b[i] = 0;
                                cover[i] = 0;
                        }

                        for (int j = 0; j < n; j++)
                                if (i == j)
                                {
                                        if (Mode == SMP_INIT_MODE_0) H[i][j] = 1;
                                }
                                else
                                        H[i][j] = 0;
```

```
        }


        c0 = 0;
    }

    // Инкрементация симплекса с учетом, того что значения слева больше значений справа (Главная диагональ)
    int IncSimp(int j) // j - позиция бита который инкрементим на 1
    {
        if (H[j][j] + 1 < Multip)
        {
            H[j][j]++;
            return H[j][j];
        }

        if (j != 0)
            H[j][j] = IncSimp(j - 1);
        else
        {
            H[j][j] = 1;
            return H[j][j];
        }
    }
```

```
// Инкрементация вектора b
int IncB(int j)
{
	if (b[j] + 1 < H[j][j])
	{
		b[j]++;
		return 0; //b[j];
	}

	if (j != 0)
		b[j] = IncB(j - 1);
	else
	{
		b[j] = 0;
		return b[j];
	}
}

// Проверка то что П(main diag) меньше установленного дельта
bool CheckDelta(int &delta)
{
	return (MultMainDiag() <= delta);
}
```

```
	// Печать симплекса
	void printSimplex()
	{
#ifdef FILE_MODE
		std::ofstream fout("res_max_N6_D4.txt", std::ios_base::app);
		fout << "> " << std::endl;
		for (int i = 0; i < n; i++)
		{
			for (int j = 0; j < n; j++)
				fout << H[i][j] << " ";
			fout << " | " << b[i] << std::endl;
		}
		fout << "< " << std::endl;
		fout.close();
#else
		std::cout << "> " << std::endl;
		for (int i = 0; i < n; i++)
		{
			for (int j = 0; j < n; j++)
				std::cout << H[i][j] << " ";
			std::cout << " | " << b[i] << std::endl;
		}
		std::cout << "< " << std::endl;
#endif
```

```
}

// Возвращает произведение элементов главной диагонали
int MultMainDiag()
{
	int mult = 1;
	for (int i = 0; i < n; i++)
		mult *= H[i][i];

	return mult;
}

// Перебор нижнего треугольника
void EnumerationLowTr(int _i, int _j, int &_NumOfCone)
{
	if (_j + 1 != _i) // Не крайняя?
		EnumerationLowTr(_i, _j + 1, _NumOfCone);
	else
		if (_i + 1 != n) // Не нижняя строка?
			EnumerationLowTr(_i + 1, 0, _NumOfCone);

	while (H[_i][_j] + 1 < H[_i][_i])
	{
		H[_i][_j]++;
```

```
                    _NumOfCone++;
#ifdef TEST_MOD
                    std::cout << std::endl << std::endl << "Number of cone: " << _NumOfCone;
                    std::cout << std::endl << "Determinate: " << MultMainDiag() << std::endl;
                    printSimplex();
#endif
#ifdef FILE_MODE
                    std::ofstream fout("res_max_N6_D4.txt", std::ios_base::app);
                    fout << std::endl << std::endl << "Number of cone: " << _NumOfCone;
                    fout << std::endl << "Determinate: " << MultMainDiag() << std::endl;
                    printSimplex();
                    fout.close();
#else
                    std::cout << "d";
#endif
                    GetAllCovers(this, this->n);

                    if (_j + 1 != _i) // Не крайняя?
                        EnumerationLowTr(_i, _j + 1, _NumOfCone);
                    else
                        if (_i + 1 != n) // Не нижняя строка?
                            EnumerationLowTr(_i + 1, 0, _NumOfCone);
                }
```

```
			H[_i][_j] = 0;
		}

		// Функция считающая ширину симплекса
		double CalcSimplexWidth()
		{
			// Направление по которому максимум
			double *minDirect = new double[n];
			// Ширина
			double minWidth = INT_MAX;
			// Вектор вершин
			std::vector<double*> VertexMas(n+1);
			for (int i = 0; i < n; i++) {
				VertexMas[i] = new double[n];
				VertexMas[i] = Gauss(MatIntToDouble(ReplaceMatrixString(i), n),
VecIntToDouble(ReplaceVectorString(i), n), n);

#ifdef TEST_MOD
				for (int j1 = 0; j1 < n; j1++)
					std::cout << VertexMas[i][j1] << " ";
				std::cout << std::endl;
#endif
			}
			VertexMas[n] = Gauss(MatIntToDouble(H, n), VecIntToDouble(b, n), n);
```

```
#ifdef TEST_MOD
                for (int j1 = 0; j1 < n; j1++)
                        std::cout << VertexMas[n][j1] << " ";
                std::cout << "\n-------\n" << std::endl;
#endif
                // Вектор конусов из перпендикуляров, A^T^-1
                std::vector<int**> ConeMas(n + 1);
                for (int i = 0; i < n; i++)
                {
                        ConeMas[i] = FindAttachedMatrix(ReplaceMatrixString(i), n);
#ifdef TEST_MOD
                        for (int i1 = 0; i1 < n; i1++) {
                                for (int j1 = 0; j1 < n; j1++)
                                        std::cout << ConeMas[i][i1][j1] << " ";
                                std::cout << std::endl;
                        }
                        std::cout << std::endl;
#endif
                }
                ConeMas[n] = FindAttachedMatrix(H, n);
#ifdef TEST_MOD
                for (int i1 = 0; i1 < n; i1++) {
                        for (int j1 = 0; j1 < n; j1++)
                                std::cout << ConeMas[n][i1][j1] << " ";
```

```
            std::cout << std::endl;
        }
        std::cout << "\n-------\n" << std::endl;
#endif

        // Перебор по всем парам вершин
        for (int i = 0; i < n + 1; i++)
            for (int j = 0; j < n + 1; j++)
                if (i != j)
                {
                    int exprSize = 2 * n + 1;
                    int **FVExpr = new int*[exprSize];
                    for (int k = 0; k < exprSize; k++)
                        FVExpr[k] = new int[n];
                    int *FVB = new int[exprSize];
                    for (int k = 0; k < 2*n; k++)
                        FVB[k] = 0;
                    FVB[2*n] = -1;

                    SetAndUniteExpr(FVExpr, n, ConeMas[i], ConeMas[j]);

                    // Целевая функция
                    double *ObjFunf = new double[n];
                    for (int k = 0; k < n; k++)
```

```
                ObjFunf[k] = VertexMas[i][k] - VertexMas[j][k];


            double *direct = new double[n];
            CPLEX_MIP(direct, MIP_OFF, FVExpr, FVB, n, exprSize,
ObjFunf, MIP_MIN);


            double width = 0;
            for (int k = 0; k < n; k++)
                width += ObjFunf[k] * direct[k];
            if (width < minWidth) {
                minWidth = width;
                for (int k = 0; k < n; k++)
                    minDirect[k] = direct[k];
            }


            delete[] direct;


            for (int k = 0; k < exprSize; k++)
                delete[] FVExpr[k];
            delete[] FVExpr;
            delete[] FVB;
            delete[] ObjFunf;
        }
```

```
#ifdef CONSCIENCE
		// Подсчет ширины по известному направлению
		int exprSize = n + 1;
		int **FinExpr = new int*[exprSize];
		for (int k = 0; k < exprSize; k++)
			FinExpr[k] = new int[n];
		int *FinB = new int[exprSize];
		for (int i = 0; i < n; i++)
		{
			FinB[i] = b[i];
			for (int j = 0; j < n; j++)
				FinExpr[i][j] = H[i][j];
		}
		for (int k = 0; k < n; k++)
			FinExpr[n][k] = cover[k];
		FinB[n] = c0;

		double *resMax = new double[n];
		double *resMin = new double[n];

		CPLEX_MIP(resMax, MIP_OFF, FinExpr, FinB, n, exprSize, minDirect,
MIP_MAX);
		CPLEX_MIP(resMin, MIP_OFF, FinExpr, FinB, n, exprSize, minDirect,
MIP_MIN);
```

```
                minWidth = 0;
                for (int k = 0; k < n; k++) {
                        minWidth += (resMax[k] - resMin[k])*(resMax[k] - resMin[k]);
                }
#endif

                for (int i = 0; i < n + 1; i++)
                        delete[] VertexMas[i];
                for (int i = 0; i < n + 1; i++)
                        delete[] ConeMas[i];
                delete[] minDirect;

                return minWidth;
        }
};

/*-----------------------------------------------------*/
// Дополнительные функции

// Получение матрицы без i-й строки и j-го столбца | Для функции Determinant()
void GetMatr(int **mas, int **p, int i, int j, int m) {
        int ki, kj, di, dj;
        di = 0;
```

```
        for (ki = 0; ki < m - 1; ki++) { // Проверка индекса строки
                if (ki == i) di = 1;
                dj = 0;
                for (kj = 0; kj < m - 1; kj++) { // Проверка индекса столбца
                        if (kj == j) dj = 1;
                        p[ki][kj] = mas[ki + di][kj + dj];
                }
        }
}

// Рекурсивное вычисление определителя
int Determinant(int **mas, int m) {
        int i, j, d, k, n;
        int **p;
        SMP_NEW(p, m, int);
        j = 0; d = 0;
        k = 1; //(-1) в степени i
        n = m - 1;
        if (m < 1) std::cout << "The determinant cannot be computed!";

        if (m == 1) {
                d = mas[0][0];
                SMP_DELETE(p, m);
                return(d);
```

```
        }
        if (m == 2) {
                d = mas[0][0] * mas[1][1] - (mas[1][0] * mas[0][1]);
                SMP_DELETE(p, m);
                return(d);
        }
        if (m > 2) {
                for (i = 0; i<m; i++) {
                        GetMatr(mas, p, i, 0, m);
                        //cout << mas[i][j] << endl;
                        //PrintMatr(p, n);
                        d = d + k * mas[i][0] * Determinant(p, n);
                        k = -k;
                }
        }
        SMP_DELETE(p, m);
        return(d);
}

// Возвращает транспонированную матрицу H
int** Transposition(int **H, int n)
{
        int **Tm;
        SMP_NEW(Tm, n, int);
```

```
	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++)
			Tm[i][j] = H[j][i];

	return Tm;
}

// Функция возвращающая матрицу без i-ой строки и j-ого столбца
int** StrikeOut(int **Mat, int n, int sI, int sJ)
{
	int **Tm;
	SMP_NEW(Tm, n - 1, int);

	int tI = 0, tJ = 0;
	for (int i = 0; i < n; i++)
		if (i != sI) {
			tJ = 0;
			for (int j = 0; j < n; j++)
				if (j != sJ) {
					Tm[tI][tJ] = Mat[i][j];
					tJ++;
				}
			tI++;
```

```
		}


	return Tm;
}

// Возвращает присоединенню матрицу
int** AttachedMat(int **H, int n)
{
	int **Res, **Tm;
	SMP_NEW(Res, n, int);

	for(int i = 0; i < n; i++)
		for (int j = 0; j < n; j++) {
			Tm = StrikeOut(H, n, i, j);
			Res[i][j] = Determinant(Tm, n - 1);
		}

	//SMP_DELETE(Tm, n - 1);
	return Res;
}

// Перевод матрицы из INT в DOUBLE
double** MatIntToDouble(int **A, int n)
{
```

```
	double **Tm;
	SMP_NEW(Tm, n, double);

	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++)
			Tm[i][j] = A[i][j];

	return Tm;
}

// Перевод вектора из INT в DOUBLE
double* VecIntToDouble(int *A, int n)
{
	double *Tm = new double[n];

	for (int i = 0; i < n; i++)
		Tm[i] = (double)A[i];

	return Tm;
}

// Метод Гаусса для решения СЛУ
double* Gauss(double **a, double *y, int n)
{
```

```
double *x, max;
int k, index;
const double eps = 0.00001;  // точность
x = new double[n];
k = 0;
while (k < n)
{
	// Поиск строки с максимальным a[i][k]
	max = abs(a[k][k]);
	index = k;
	for (int i = k + 1; i < n; i++)
	{
		if (abs(a[i][k]) > max)
		{
			max = abs(a[i][k]);
			index = i;
		}
	}
	// Перестановка строк
	if (max < eps)
	{
		// нет ненулевых диагональных элементов
		std::cout << "Решение получить невозможно из-за нулевого столбца ";
		std::cout << index << " матрицы A" << std::endl;
```

```
            return 0;
        }
        for (int j = 0; j < n; j++)
        {
            double temp = a[k][j];
            a[k][j] = a[index][j];
            a[index][j] = temp;
        }
        double temp = y[k];
        y[k] = y[index];
        y[index] = temp;
        // Нормализация уравнений
        for (int i = k; i < n; i++)
        {
            double temp = a[i][k];
            if (abs(temp) < eps) continue; // для нулевого коэффициента пропустить
            for (int j = 0; j < n; j++)
                a[i][j] = a[i][j] / temp;
            y[i] = y[i] / temp;
            if (i == k)  continue; // уравнение не вычитать само из себя
            for (int j = 0; j < n; j++)
                a[i][j] = a[i][j] - a[k][j];
            y[i] = y[i] - y[k];
        }
```

```
            k++;
        }
        // обратная подстановка
        for (k = n - 1; k >= 0; k--)
        {
            x[k] = y[k];
            for (int i = 0; i < k; i++)
                y[i] = y[i] - a[i][k] * x[k];
        }
        return x;
}
/*----------------------------------------------------*/
// Тесты для CPLEX MIP
#define TEST_COMPIRE(val1, val2)          \
if (val1 == val2) {                       \
        std::cout << "[ OK ]" << std::endl;     \
}                                         \
else {                                    \
        std::cout << "[ FAILED ]" << std::endl; \
}

// Виды тестов
enum TypeTest {
        TESTS_CPLEX_MIP   = 1 << 0,
```

```
        TESTS_CALC_WIDTH  = 1 << 1,
        TESTS_OTHER       = 1 << 2,

        TESTS_ALL         = 7
};

void RunAllTests(TypeTest type = TESTS_ALL) {

        int Delta = 4;
        const int N = 2;
        Simplex S(N, Delta + 1);
        int res[N];

        if (type & TESTS_CPLEX_MIP) {

                std::cout << std::endl << "============ TYPE TEST : CPLEX MIP" << std::endl
<< std::endl;

                /*Case 01*/
                S.H[0][0] = 2; S.H[0][1] = 0; S.b[0] = 1 - 1;
                S.H[1][0] = 1; S.H[1][1] = 2; S.b[1] = 0 - 1;

                S.cover[0] = -1;
                S.cover[1] = -1;
```

```
CPLEX_MIP(res, MIP_ON, S.H, S.b, S.n, S.n, S.cover, MIP_MIN);

std::cout << "Case 01: "; TEST_COMPIRE(res[0] * S.cover[0] + res[1] * S.cover[1],
1); std::cout << std::endl;

/*Case 02*/
S.H[0][0] = 1; S.H[0][1] = 0; S.b[0] = 1;
S.H[1][0] = -3; S.H[1][1] = 2; S.b[1] = 3;

S.cover[0] = -3;
S.cover[1] = -3;

CPLEX_MIP(res, MIP_ON, S.H, S.b, S.n, S.n, S.cover, MIP_MIN);

std::cout << "Case 02: "; TEST_COMPIRE(res[0] * S.cover[0] + res[1] * S.cover[1],
-12); std::cout << std::endl;

/*Case 03*/
S.H[0][0] = 1; S.H[0][1] = 0; S.b[0] = 1;
S.H[1][0] = -3; S.H[1][1] = 2; S.b[1] = 3;

S.cover[0] = 1;
S.cover[1] = 1;
```

```
CPLEX_MIP(res, MIP_ON, S.H, S.b, S.n, S.n, S.cover, MIP_MAX);

std::cout << "Case 03: "; TEST_COMPIRE(res[0] * S.cover[0] + res[1] * S.cover[1],
4); std::cout << std::endl;

/*Case 04*/
S.H[0][0] = 2; S.H[0][1] = 3; S.b[0] = 6;
S.H[1][0] = 2; S.H[1][1] = -3; S.b[1] = 3;

S.cover[0] = 3;
S.cover[1] = 1;

CPLEX_MIP(res, MIP_ON, S.H, S.b, S.n, S.n, S.cover, MIP_MAX);

std::cout << "Case 04: "; TEST_COMPIRE(res[0] * S.cover[0] + res[1] * S.cover[1],
4); std::cout << std::endl;

/*Case 05*/
Delta = 14;
int N3 = 3;
Simplex S3(N3, Delta + 1);

S3.H[0][0] = 3; S3.H[0][1] = 2; S3.H[0][2] = 8; S3.b[0] = 11;
```

```
S3.H[1][0] = 2; S3.H[1][1] = 0; S3.H[1][2] = 1; S3.b[1] = 5;

S3.H[2][0] = 3; S3.H[2][1] = 3; S3.H[2][2] = 1; S3.b[2] = 13;


S3.cover[0] = 11;

S3.cover[1] = 5;

S3.cover[2] = 4;


CPLEX_MIP(res, MIP_ON, S3.H, S3.b, S3.n, S3.n, S3.cover, MIP_MAX, MIP_XG0);


std::cout << "Case 05: "; TEST_COMPIRE(res[0] * S3.cover[0] + res[1] * S3.cover[1] + res[2] * S3.cover[2], 32); std::cout << std::endl;


/*Case 06*/

double resD[N];

S.H[0][0] = 1; S.H[0][1] = 0; S.b[0] = 0;

S.H[1][0] = 1; S.H[1][1] = 4; S.b[1] = 2;


S.cover[0] = 1;

S.cover[1] = 1;


CPLEX_MIP(resD, MIP_OFF, S.H, S.b, S.n, S.n, VecIntToDouble(S.cover, S.n), MIP_MAX);
```

```
std::cout << "Case 06: "; TEST_COMPIRE(resD[0] * S.cover[0] + resD[1] *
S.cover[1], 0.5); std::cout << std::endl;


/*Case 07*/

/*int exprNum = 3;

double resD[N];

S.H[0][0] = 1; S.H[0][1] = 0; S.b[0] = 0;

S.H[1][0] = 1; S.H[1][1] = 2; S.b[1] = 0;


S.cover[0] = -1; S.cover[1] = -1; S.c0 = 1;


int **exprMas = new int*[exprNum];

for (int newCount = 0; newCount < exprNum; newCount++)

    exprMas[newCount] = new int[N];


int *exprB = new int[exprNum];


exprMas[0][0] = S.H[0][0]; exprMas[0][1] = S.H[0][1]; exprB[0] = S.b[0];

exprMas[1][0] = S.H[1][0]; exprMas[1][1] = S.H[1][1]; exprB[1] = S.b[1];

exprMas[2][0] = S.cover[0]; exprMas[2][1] = S.cover[1]; exprB[2] = S.c0;


CPLEX_MIP(resD, MIP_OFF, exprMas, exprB, N, exprNum, S.cover, MIP_MAX);
```

```
        std::cout << "Case 07: "; TEST_COMPIRE(resD[0] * S.cover[0] + resD[1] *
S.cover[1], 0.5); std::cout << std::endl;*/

    }

    if (type & TESTS_CALC_WIDTH) {


        std::cout << std::endl << "============ TYPE TEST : WIDTH CALC" <<
std::endl << std::endl;


        double width = 0;


        /*Case 01*/

        S.H[0][0] = 1; S.H[0][1] = 0; S.b[0] = 0;

        S.H[1][0] = 1; S.H[1][1] = 2; S.b[1] = 0;


        S.cover[0] = -1;

        S.cover[1] = -1;

        S.c0 = 1;


        width = S.CalcSimplexWidth();


        std::cout << "Case 01: "; TEST_COMPIRE(width, sqrt(2)/2); std::cout << std::endl;


        /*Case 02*/

        S.H[0][0] = 1; S.H[0][1] = 0; S.b[0] = 0;
```

```
            S.H[1][0] = 2; S.H[1][1] = 3; S.b[1] = 0;


            S.cover[0] = -2;

            S.cover[1] = -2;

            S.c0 = 1;


            width = S.CalcSimplexWidth();
#ifdef CONSCIENCE
            std::cout << "Case 02: "; TEST_COMPIRE(width, sqrt(2) / 8); std::cout << std::endl;
#else
            std::cout << "Case 02: "; TEST_COMPIRE(width, width); std::cout << std::endl;
#endif
        }
        if (type & TESTS_OTHER)
        {
            std::cout << std::endl << "============ TYPE TEST : FIND ATTACHED
MATRIX" << std::endl << std::endl;


            int famN = 2;
            int **mas;
            SMP_NEW(mas, famN, int);
            int **famRes;
            bool testRes;
```

```
/*Case 01*/

mas[0][0] = 1; mas[0][1] = 0;

mas[1][0] = 1; mas[1][1] = 2;

famRes = FindAttachedMatrix(mas, famN);


testRes = true;

if (famRes[0][0] != 2 ) testRes = false;

if (famRes[0][1] != -1) testRes = false;

if (famRes[1][0] != 0 ) testRes = false;

if (famRes[1][1] != 1 ) testRes = false;


std::cout << "Case 01: "; TEST_COMPIRE(testRes, true); std::cout << std::endl;


/*Case 02*/

mas[0][0] = 1; mas[0][1] = 0;

mas[1][0] = -1; mas[1][1] = -1;

famRes = FindAttachedMatrix(mas, famN);


testRes = true;

if (famRes[0][0] != 1 ) testRes = false;

if (famRes[0][1] != -1) testRes = false;

if (famRes[1][0] != 0 ) testRes = false;

if (famRes[1][1] != -1) testRes = false;
```

```
		std::cout << "Case 02: "; TEST_COMPIRE(testRes, true); std::cout << std::endl;


		/*Case 03*/
		int **mas3;
		SMP_NEW(mas3, famN + 1, int);
		int **famRes3;

		mas3[0][0] = 1; mas3[0][1] = 0; mas3[0][2] = 0;
		mas3[1][0] = 1; mas3[1][1] = -3; mas3[1][2] = 0;
		mas3[2][0] = 4; mas3[2][1] = 3; mas3[2][2] = 5;
		famRes3 = FindAttachedMatrix(mas3, famN + 1);

		testRes = true;
		if (famRes3[0][0] != 15) testRes = false; if (famRes3[0][1] != 5) testRes = false; if (famRes3[0][2] != -15) testRes = false;
		if (famRes3[1][0] != 0) testRes = false; if (famRes3[1][1] != -5) testRes = false; if (famRes3[1][2] != 3) testRes = false;
		if (famRes3[2][0] != 0) testRes = false; if (famRes3[2][1] != 0) testRes = false; if (famRes3[2][2] != 3) testRes = false;

		SMP_DELETE(mas3, famN + 1);
		SMP_DELETE(famRes3, famN + 1);

		std::cout << "Case 03: "; TEST_COMPIRE(testRes, true); std::cout << std::endl;
```

```
            SMP_DELETE(mas, famN);
        }

}

/*-------------------------------------------------*/
// Тестовые функции

void GetAllCovers(Simplex *S, int N)
{
        double *t = new double[N];
        int *x = new int[N];
        for (int i = 0; i < N; i++)
        {
            t[i] = 0;
            x[i] = 0;
        }

        // На вход идут транспонированная матрица H, два пустых вектора t и x, значение m - точка погружения
        S->EnumIntDot(Transposition(S->H, S->n), t, N - 1, x);

        delete[] t;
        delete[] x;
```

```
}

// Функция проверяющая симплекс на пустоту (Отсутствие внутри симплекса целых точек)
bool SimplexIsEmpty(Simplex *S)
{
	/*S.H[0][0] = 1; S.H[0][1] = 0; S.b[0] = 0;
	S.H[1][0] = 3; S.H[1][1] = 4; S.b[1] = 2;

	S.cover[0] = -3; S.cover[1] = -3; S.c0 = -1;*/

	// Уменьшенное b на единицу
	int *b_inc = new int[S->n];
	for (int i = 0; i < S->n; i++)
		b_inc[i] = S->b[i] - 1;


	int *res = new int[S->n];

	CPLEX_MIP(res, MIP_ON, S->H, b_inc, S->n, S->n, S->cover, MIP_MIN);

	int res_d = 0;

	for (int i = 0; i < S->n; i++) {
		res_d += res[i] * S->cover[i];
```

```
        }


        delete[] res;


#ifdef TEST_MOD
        std::cout << "Obj func: " << res_d << std::endl;
#endif
        // ФИНТ УШАМИ!!!
        if (S->c0 < res_d)
                S->c0 = res_d;


        return (S->c0 <= res_d);
}


void GetAllSimplex(int &n, int &delta)
{
#ifdef TEST_MOD
        std::cout << "Simplex" << std::endl;
        std::cout << "N: " << n << std::endl;
        std::cout << "Delta: " << delta << std::endl;
#endif
#ifdef FILE_MODE
        std::ofstream fout("res_max_N6_D4.txt", std::ios_base::out);
        fout << "Simplex" << std::endl;
```

```
	fout << "N: " << n << std::endl;
	fout << "Delta: " << delta << std::endl;
#endif



	Simplex s(n, delta + 1);


	int NumOfCone  = 0; // Количество конусов
	int DeltaCount = 0; // Счетчик для остановки по дельте


	while(true)
	{
		if (s.CheckDelta(delta))
		{
			for (int i = 0; i < s.MultMainDiag(); i++)
			{
#ifdef TEST_MOD
				std::cout << std::endl << std::endl << "Number of cone: " << NumOfCone;
				std::cout << std::endl << "Determinate: " << s.MultMainDiag() << std::endl;
				s.printSimplex();
#else
				std::cout << "d";
```

```
#endif
#ifdef FILE_MODE
                        fout << std::endl << std::endl << "Number of cone: " << NumOfCone;
                        fout << std::endl << "Determinate: " << s.MultMainDiag() <<
std::endl;
                        s.printSimplex();
#endif
                        GetAllCovers(&s, s.n);

                        s.EnumerationLowTr(1, 0, NumOfCone);
                        s.InitDef(SMP_INIT_MODE_1);

                        NumOfCone++;
                        s.IncB(n - 1);
                    }

                    DeltaCount = 0;
                }
                else
                    DeltaCount++;

                if ((s.IncSimp(n - 1) == 1) || (DeltaCount == delta))
                    break;
            }
```

```
        std::cout << "Number of all cones: " << NumOfCone << std::endl;
#ifdef FILE_MODE
        fout << "Number of all cones: " << NumOfCone << std::endl;
        fout.close();
#endif
}

/*-------------------------------------------------*/

//#define SOURCE_CODE
//#define TEST_SIMPLEX_IS_EMPTY
//#define EXAMPLE_2
#ifdef SOURCE_CODE
#define SMP_TWO
#endif

int main(int argc, char** argv)
{
#ifdef SOURCE_CODE

        int MaxDimensionSize = 15; // Максивальная размерность, для GetAllSimplex(N, Delta)

#ifdef SMP_TWO
```

```
        // Пример двухмерной задачи
        int Delta = 4;
        int N = 2;
        Simplex S(N, Delta + 1);

        S.H[1][0] = 3; S.H[1][1] = 4;
        S.b[1] = 2;
#else
        // Пример трехмерной задачи
        int Delta = 12;
        int N = 3;
        Simplex S(N, Delta + 1);

        S.H[1][0] = 2; S.H[1][1] = 3;
        S.H[2][0] = 1; S.H[2][1] = 2; S.H[2][2] = 4;
        S.b[0] = 0; S.b[1] = 2; S.b[2] = 3;
#endif

        S.printSimplex();

        unsigned int start_time;
        unsigned int end_time;

#ifdef SMP_OLD_CODE
```

```
        std::cout << std::endl << "#OLD_CODE" << std::endl;
        start_time = clock();
        S.EnumerationIntDot();
        end_time = clock();
        std::cout << std::endl << "Time: " << (double)(end_time - start_time) / 1000 << std::endl;
        std::cout << std::endl << "#OLD_CODE" << std::endl;
#endif

        start_time = clock();
        GetAllCovers(&S, N);
        end_time = clock();
        std::cout << std::endl << "Time: " << (double)(end_time - start_time) / 1000 << std::endl;

#endif

        setlocale(LC_ALL, "Russian");
        //RunAllTests(TESTS_CALC_WIDTH);

        // ------------------------------------------------------------------------------------------
//#ifdef SOURCE_CODE
        int sN = 6, eN = 6;
        int sDelta = 4, eDelta = 4;

        double start_time, end_time;
```

```
        for (int delta = sDelta; delta <=eDelta; delta++)
                for (int n = sN; n <= eN; n++) {
                        std::cout << std::endl << "-------------------------------------------------------";
                        std::cout << std::endl << "Delta: " << delta;
                        std::cout << std::endl << "N: " << n;
                        std::cout << std::endl << "----------------------------";
#ifdef FILE_MODE
                        std::ofstream fout("res_max_N6_D4.txt");
                        fout << std::endl << "-------------------------------------------------------";
                        fout << std::endl << "Delta: " << delta;
                        fout << std::endl << "N: " << n;
                        fout << std::endl << "----------------------------";
#endif
                        start_time = clock();
                        GetAllSimplex(n, delta);
                        end_time = clock();
                        std::cout << std::endl << "----------------------------";
                        std::cout << std::endl << "Time: " << (end_time - start_time) / 1000 <<
std::endl;
                        std::cout << std::endl << "-------------------------------------------------------"
<< std::endl;
#ifdef FILE_MODE
                        fout << std::endl << "----------------------------";
```

```
                        fout << std::endl << "Time: " << (end_time - start_time) / 1000 << std::endl;
                        fout << std::endl << "-----------------------------------------------------------" <<
std::endl;
                        fout.close();
#endif
                }
//#endif
        return 0;
}

#ifdef TEST_SIMPLEX_IS_EMPTY
int main()
{
        int Delta = 4;
        int N = 2;
        Simplex S(N, Delta + 1);

        if (SimplexIsEmpty(S))
                std::cout << std::endl << "IT'S WORK!";
        else
                std::cout << std::endl << "Nope";

        std::cin >> N;
        return 0;
```

```
}
#endif


#ifdef EXAMPLE_2


int main()
{
	int nAgents = 2;
	int nTasks = 2;
	int nLevels = 2;
	double seconds = 30.0;

	IloEnv env;
	IloArray<IloArray<IloNumVarArray> > x(env, nAgents);

	for (int i = 0; i < nAgents; i++) {
		x[i] = IloArray<IloNumVarArray>(env, nTasks);
		for (int j = 0; j < nTasks; j++) {
			x[i][j] = IloNumVarArray(env, nLevels, 0, 1, ILOINT);
		}
	}

	IloModel model(env);
	IloExpr obj(env);
```

```
for (int i = 0; i < nAgents; i++)
        for (int j = 0; j < nTasks; j++)
                for (int k = 0; k < nLevels; k++)
                        obj += (i + j + k + 1) * x[i][j][k];

model.add(IloMinimize(env, obj));
obj.end();

IloCplex cplex(env);
cplex.setOut(env.getNullStream());

cplex.setParam(IloCplex::Threads, 1);
cplex.setParam(IloCplex::WorkMem, 1024);
cplex.setParam(IloCplex::TreLim, 2048);
cplex.setParam(IloCplex::Param::Parallel, 1); /* Deterministic */

cplex.extract(model);

IloNumVarArray startVar(env);
IloNumArray startVal(env);

for (int i = 0; i < nAgents; i++)
        for (int j = 0; j < nTasks; j++)
```

```
                for (int k = 0; k < nLevels; k++) {
                        startVar.add(x[i][j][k]);
                        startVal.add(0);
                }

        cplex.addMIPStart(startVar, startVal);

        // Solve
        cplex.solve();
        if (cplex.getStatus() == IloAlgorithm::Optimal) {
                cplex.getValues(startVal, startVar);
                env.out() << "Values = " << startVal << std::endl;
        }

        startVar.end();
        startVal.end();

        return 0;
}

#endif
```