Метод эллипсоидов

Самсонов Владислав
396 группа ФИВТ ПМИ
e-mail: vvladxx@gmail.com

17 мая 2015

1 Введение

В данной работе рассматривается алгоритмическая реализация метода эллипсоидов на языке C++ для решения задач линейного программирования.

Впервые данный метод предложили Н. З. Шор [1], Д. Б. Юдин и А. С. Немировский [2]. Впоследствии, Л. Хачиян [3] показал как применять метод эллипсоидов для решения задач линейного программирования.

Важно отметить, в какой постановке задача решается за полиномиальное время. Рассмотрим

$$\max\{c^T x : Ax \leq b\}$$

т.ч.

$$x \in \mathbb{R}^n, A \in \mathbb{Q}^{m\times n}, b \in \mathbb{Q}^m$$

Тогда метод эллипсоидов позволяет решить задачу за полиномиальное время от $m+n+\rho$, где ρ — максимальная длина бинарного представления рационального числа в A или b.

2 Теоретические сведения

Базовый метод эллипсоидов позволяет решать следующую задачу: дан многогранник P

$$P := \{Ax \leq b\}$$

Найти любую точку $x \in P$ из многогранника P, или вывести, что P вырожден или неограничен. Таким образом, метод позволяет определить, имеет ли решение система неравенств $Ax \leq b$. Можно предпологать, что A имеет полный ранг по столбцам. В противном случае методом Гаусса находится матрица A', т.ч. $AU = [A'|0]$. A' имеет полный ранг по столбцам. Тогда система $A'x \leq b$ разрешима тогда и только тогда, когда $Ax \leq b$ разрешима.

Данная задача эквивалентна следующей задаче линейного программирования [4]

$$\min\{c^T x : Ax \leq b, x \geq 0\} \tag{2.1}$$

В свою очередь, для решения 2.1 достаточно найти любое решение системы неравенств

$$\begin{aligned} Ax &\leq b \\ x \geq 0 &\Leftrightarrow -x \leq 0 \\ A^T y \geq c &\Leftrightarrow -A^T y \leq -c \\ y \geq 0 &\Leftrightarrow -y \leq 0 \\ b^T y - c^T x &\leq 0 \end{aligned} \tag{2.2}$$

Любое допустимое решение 2.2 является оптимальным решением 2.1. Таким образом, применяя метод эллипсоидов к 2.2, получаем решение задачи линейного программирования 2.1. Поскольку входные данные для 2.2 полиномиально зависят от 2.1, полученный алгоритм тоже будет полиномиальным (при условии, что метод эллипсоидов имеет полиномиальное время).

2.1 Определения

Назовём эллипсоидом $E \subset \mathbb{R}^n$ с центром в точке $x_0 \in \mathbb{R}^n$ и направляющей матрицей $E_0 \in \mathbb{R}^{n \times n}$ множество точек

$$\{x \in \mathbb{R}^n : (x - x_0)^T E_0^{-1} (x - x_0) \leq 1\}$$

Матрица E_0 – симметричная, положительно определённая.
В частном случае, если $E_0 = R^2 I_n$, где I_n – единичная матрица, то E – n-мерный шар радиуса R.

2.2 Метод эллипсоидов

Пусть известно следующее:

(1) ε — если P невырожденный, то он содержит шар, а значит $\exists\varepsilon$, т.ч. $vol(P) \geq \varepsilon$.

(2) Эллипсоид E_0, т.ч. $P \subset E_0$, если P ограничен.

Тогда метод эллипсоидов описывается так:

1. $k = 0$

2. Если центр x_k эллипсоида E_k лежит в P (другими словами, выполнено $Ax_k \leq b$), то найдено решение.

3. Если $vol(E_k) < \varepsilon$, то P вырожден или неограничен.

4. Иначе, пусть a_i – первая строка матрицы A, для которой выполнено $a_i x_k > b$. Полагаем

$$E_{k+1} = \frac{n^2}{n^2-1}(E_k - \frac{2}{n+1}CC^T)$$

$$x_{k+1} = x_k - \frac{1}{n+1}C$$

, где $C = \frac{Aa_i^T}{\sqrt{a_i A a_i^T}}$. Возвращаемся на 2-ой шаг алгоритма, полагая $k = k+1$.

Доказательство корректности формул для переходов $E_k \to E_{k+1}$ и $x_k \to x_{k+1}$ можно найти в [5].

Осталось понять, как выбирать ε и E_0.

Лемма 1. *Если $P \neq \emptyset$ и бинарное представление A конечно, то*

$$vol(P) \geq 2^{n^3}(\prod_{i,j}(A_{ij}+1))^{-(n+1)}$$

Доказательство можно найти в [6].

Исходя из леммы, можно выбрать

$$\varepsilon = 2^{n^3}(\prod_{i,j}(A_{ij}+1))^{-(n+1)}$$

Лемма 2. *P лежит в шаре с центром в 0 и радиусом*

$$R = \sqrt{n}2^{-n^2}\prod_{i,j}(A_{ij}+1)\prod_{i}(b_i+1)$$

Доказательство можно найти в [5].

В качестве E_0 выбираем шар $\mathbb{B}(0,R)$.

Лемма 3.

$$\frac{vol(E_{k+1})}{vol(E_k)} < e^{-\frac{1}{2(n+1)}}$$

Лемма позволяет оценить скорость сходимости алгоритма. Действительно, после k итераций $vol(E_k) \le vol(E_0)e^{-\frac{k}{2(n+1)}}$. Если решение существует, то верхняя оценка количества итераций равна $2(n+1)\ln\frac{vol(E_0)}{vol(P)}$.

Доказательство. Воспользуемся тем фактом, что объем эллипсоида пропорционален произведению его осей. Имеем

$$E_{k+1} = \frac{n^2}{n^2-1}(E_k - \frac{2}{n+1}CC^T)$$

Тогда

$$\frac{vol(E_{k+1})}{vol(E_k)} = \frac{(\frac{n}{n+1})(\frac{n^2}{n^2-1})^{\frac{n-1}{2}}}{1} = \frac{n}{n+1}(\frac{n^2}{n^2-1})^{\frac{n-1}{2}}$$

Используем неравенство $1+x < e^x$, где $x > 0$

$$< e^{-\frac{1}{n+1}}e^{\frac{n-1}{2(n^2-1)}} = e^{-\frac{1}{n+1}}e^{\frac{1}{2(n+1)}} = e^{-\frac{1}{2(n+1)}}$$

□

3 Реализация алгоритма

3.1 Описание

Для удобства, весь приведённый здесь код также доступен по ссылке: https://github.com/VladX/Ellipsoid-Method

На вход алгоритму подается матрица A и вектор b. Алгоритм находит произвольное решение системы $Ax \le b$, либо выводит, что решений нет. В алгоритме не производится никаких дополнительных проверок на корректность матрицы A.

Первой строкой считывается число n — размер матрицы A. Матрица считается квадратной. Если это не так, то нужно дополнить матрицу нулями до квадратной.

Далее считывается n^2 чисел — элементы матрицы A.

Последней строкой считывается n чисел — элементы вектора b.

3.2 Сравнение с симплекс-методом, тесты

Для оценки эффективности было сгенерировано несколько случайных 100-мерных выпуклых многогранника. Затем решалась 100-мерная задача оптимизации симплекс методом и эквивалентная ≈200-мерная задача поиска допустимого решения методом эллипсоидов. По результатам тестовых запусков метод эллипсоидов совершал в среднем на 241% больше итераций.

3.3 Код на C++

```
#include <iostream>
#include <math.h>
#include <string.h>
using namespace std;

class Vector {
private:
	double * data;
	const size_t n;
public:
	inline Vector (size_t n) : n(n) {
		data = new double[n]();
	}

	inline Vector (size_t n, const double * d) : n(n) {
		data = new double[n];
		memcpy(data, d, sizeof(double) * n);
	}

	inline Vector (const Vector & v) : n(v.n) {
		data = new double[v.n];
		memcpy(data, v.data, sizeof(double) * v.n);
```

```
        }

        inline ~Vector () {
                delete[] data;
        }

        inline double operator[] (size_t i) const {
                return data[i];
        }

        inline double & operator[] (size_t i) {
                return data[i];
        }

        inline double operator* (const Vector & v) const {
                double res = 0;
                for (size_t i = 0; i < n; ++i)
                        res += data[i] * v[i];
                return res;
        }

        inline void operator-= (const Vector & v) {
                for (size_t i = 0; i < n; ++i)
                        data[i] -= v[i];
        }

        inline void operator/= (double s) {
                for (size_t i = 0; i < n; ++i)
                        data[i] /= s;
        }

        inline void dump () const {
                for (size_t i = 0; i < n; ++i)
                        cout << data[i] << ' ';
                cout << endl;
        }
};

class Matrix {
private:
        double * data;
        const size_t n;
public:
        inline Matrix (size_t n) : n(n) {
                data = new double[n*n]();
```

```
	}

	inline Matrix (const Matrix * m) : n(m->n) {
		data = new double[m->n * m->n];
		memcpy(data, m->data, sizeof(double) * m->n *
		    m->n);
	}

	inline Matrix (const Matrix & m) : n(m.n) {
		data = new double[m.n * m.n];
		memcpy(data, m.data, sizeof(double) * m.n * m
		    .n);
	}

	inline ~Matrix () {
		delete[] data;
	}

	inline size_t size () const {
		return n;
	}

	inline const double * operator[] (size_t i) const {
		return data + i * n;
	}

	inline double * operator[] (size_t i) {
		return data + i * n;
	}

	// Определитель
	inline double det () const {
		double det = 1;
		double ** a = new double *[n];
		for (size_t i = 0; i < n; ++i)
			a[i] = new double[n];
		const double EPS = 1E-9;
		for (size_t i = 0; i < n; ++i)
			for (size_t j = 0; j < n; ++j)
				a[i][j] = data[i * n + j];
		for (size_t i = 0; i < n; ++i) {
			size_t k = i;
			for (size_t j=i+1; j<n; ++j)
				if (fabs(a[j][i]) > fabs(a[k
				    ][i]))
```

```
					k = j;
			if (fabs(a[k][i]) < EPS) {
				det = 0;
				break;
			}
			swap(a[i], a[k]);
			if (i != k)
				det = -det;
			det *= a[i][i];
			for (size_t j = i + 1; j < n; ++j)
				a[i][j] /= a[i][i];
			for (size_t j = 0; j < n; ++j)
				if (j != i && fabs(a[j][i]) > EPS)
					for (size_t k=i+1; k<n; ++k)
						a[j][k] -= a[i][k] * a[j][i];
		}
		for (size_t i = 0; i < n; ++i)
			delete[] a[i];
		delete[] a;
		return det;
	}

	inline void operator*= (double s) {
		for (size_t i = 0; i < n*n; ++i)
			data[i] *= s;
	}

	inline void operator-= (const Matrix & m) {
		for (size_t i = 0; i < n*n; ++i)
			data[i] -= m.data[i];
	}
};

// Единичная матрица
class IdentityMatrix : public Matrix {
public:
	IdentityMatrix (size_t n) : Matrix(n) {
		for (size_t i = 0; i < n; ++i)
			this->operator[](i)[i] = 1;
	}
};
```

```

// Объем эллипсоида с точностью до константы
double volume (const Matrix & B) {
        return sqrt(B.det());
}

// Радиус сферы, сод. многогранник
double ComputeInitialRadius (const Matrix & A, const Vector & b) {
        const size_t n = A.size();
        double R = ceil(sqrt(n)) * ceil(1.0/pow(2.0, n*n));
        for (size_t i = 0; i < n; ++i)
                R *= fabs(b[i]) + 1.0;
        for (size_t i = 0; i < n; ++i)
                for (size_t j = 0; j < n; ++j)
                        R *= fabs(A[i][j]) + 1.0;
        return R;
}

int main () {
        size_t n; // Размер матрицы A (n*n)
        cin >> n;
        IdentityMatrix I(n);
        Matrix A(n);
        Vector b(n);
        for (size_t i = 0; i < n; ++i)
                for (size_t j = 0; j < n; ++j)
                        cin >> A[i][j]; // Считываем матрицу A
        for (size_t i = 0; i < n; ++i)
                cin >> b[i]; // Считываем вектор b
        double R = ComputeInitialRadius(A, b); // Считаем начальный радиус сферы, целиком сод. многогранник
        Vector x(n);
        Matrix B(I); // Начальный эллипсоид
        B *= R*R;
        const double L = 1e-2;
        while (volume(B) > L) {
                size_t i = 0;
                for (i = 0; i < n; ++i) { // Находим первое неравенство, которое не выполняется
                        double s = 0;
                        for (size_t j = 0; j < n; ++j)
                                s += A[i][j] * x[j];
                        if (s > b[i])
                                break;
                }
```

```
        if (i == n) { // Все неравенства выполнены, решение найдено
            cout << "Решение:\n";
            for (i = 0; i < n; ++i)
                cout << x[i] << '␣';
            cout << endl;
            return 0;
        }
        Vector a(n, A[i]);
        Vector Bka(n);
        for (size_t i = 0; i < n; ++i)
            for (size_t j = 0; j < n; ++j)
                Bka[i] += B[i][j] * a[j];
        double aTBka = a * Bka;
        { // Пересчитываем x (центр эллипсоида)
            Vector tmp(Bka);
            tmp /= (n + 1) * sqrt(aTBka);
            x -= tmp;
        }
        { // Пересчитываем B (матрица, задающая эллипсоид)
            Matrix tmp(n);
            for (size_t i = 0; i < n; ++i)
                for (size_t j = 0; j < n; ++j)
                    tmp[i][j] = Bka[i] * Bka[j];
            tmp *= 2.0 / ((n + 1.0) * aTBka);
            B -= tmp;
            B *= ((double) n*n) / (n*n - 1.0);
        }
    }
    cout << "Решений не найдено" << endl;
    return 0;
}
```

Список литературы

[1] Шор Н.З. Использование операций растяжения пространства в задачах минимизации выпуклых функций. Кибернетика, 13(1):94–96, 1970.

[2] Юдин Д.Б. и Немировский А.С. Вычислительная сложность и эффективные методы решения выпуклых оптимизационных задач. Эко-

номика и математические методы, 12:357–369, 1976.

[3] Л. Хачиян. Полиномиальный алгоритм в линейном программировании. Доклады академии наук СССР, 244:1093–1097, 1979.

[4] S. Rebennack. Ellipsoid Method. Encyclopedia of Optimization, Second Edition, C.A. Floudas and P.M. Pardalos (Eds.), Springer, pp. 890–899, 2008.

[5] M. Grötschel, L. Lovász, and A. Schrijver. Geometric Algorithms and Combinatorial Optimization, volume 2 of Algorithms and Combinatorics. Springer, 1988.

[6] M. Grötschel, L. Lovász, and A. Schrijver. The Ellipsoid Method and Its Consequences in Combinatorial Optimization. Combinatorica, 1:169–197, 1981.