Санкт-Петербургский государственный университет
Математико-механический факультет
Кафедра Статистического Моделирования

«Научно-исследовательская работа» (семестр 7)

Разработка программных средств и решение задач принятия решений с помощью методов тропической математики.

Выполнил:
Ткаченко Егор Андреевич
группа 19.Б04-мм

Научный руководитель:
д. ф.-м. н., профессор
Кривулин Николай Кимович

Санкт-Петербург
2023

Оглавление

Введение

Многокритериальные задачи оценки альтернатив на основе парных сравнений составляют важный класс задач принятия решений, которые встречаются во многих областях научной и практической деятельности. Пусть имеется набор альтернатив (способов, вариантов) принятия некоторого решения. Известны количественные результаты парных сравнений, при которых любые две альтернативы сравниваются между собой в соответствии с несколькими критериями. Результаты сравнений могут быть получены, например, путем опроса респондентов (экспертов, покупателей, избирателей) или с помощью других процедур сравнения. Требуется на основе относительных результатов парных сравнений определить абсолютный рейтинг (приоритет, степень предпочтения, вес) каждой альтернативы для принятия решения. Такие задачи встречаются при принятии управленческих решений в менеджменте, изучении предпочтений потребителей в маркетинге, анализе социологических опросов в социологии, прогнозе результатов выборов в политологии и в других областях [1].

Для решения задач оценки альтернатив на основе парных сравнений существует два вида методов — эвристические алгоритмы и строго обоснованные математические решения (аналитические методы).

Одним из аналитических решений является метод аппроксимации матрицы парных сравнений в log-чебышевской метрике. Данный метод хорошо записывается в терминах max-алгебры [2].

Имеется проблема разработки эффективных программных средств для решения задач с помощью max-алгебры, в частности, задачи принятия решений. Настоящая работа направлена на решение указанной проблемы и имеет целью разработку указанных программных средств.

Глава 1

Задачи принятия решений

1.1. Однокритериальная задача принятия решений на основе парных сравнений

Дано n альтернатив $\mathcal{A}_1, \ldots, \mathcal{A}_n$ принятия решения, которые сравниваются попарно. Результаты сравнений записываются в виде матрицы парных сравнений $\boldsymbol{A} = (a_{ij})$ порядка n, где элемент $a_{ij} > 0$ показывает во сколько раз альтернатива $\mathcal{A}_i$ превосходит альтернативу $\mathcal{A}_j$. Требуется на основе относительных результатов парных сравнений определить вектор $\boldsymbol{x}$ абсолютных рейтингов альтернатив [1].

1.2. Многокритериальная задача принятия решений на основе парных сравнений

Рассмотрим задачу оценки рейтингов альтернатив, в которой n альтернатив $\mathcal{A}_1, \ldots, \mathcal{A}_n$ сравниваются попарно по m критериям. Пусть $\boldsymbol{A}_k$ обозначает матрицу порядка n результатов парных сравнений альтернатив в соответствии с критерием $k = 1, \ldots, m$. Критерии также сравниваются попарно, а результаты их сравнений образуют матрицу $\boldsymbol{C} = (c_{kl})$, где c_{kl} показывает во сколько раз критерий k важнее для принятия решения, чем l. Необходимо на основе матриц парных сравнений $\boldsymbol{C}$ и $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_m$ найти абсолютный индивидуальный рейтинг каждой альтернативы [1].

Глава 2

Элементы тропической математики

Используются определения max-алгебры и ее элементов из работ [1, 2].

Определение. *Мах-умножить алгебра — множество* $\mathbb{R}_+ = \{x \in \mathbb{R} \,|\, x \geq 0\}$ *с операциями сложения и умножения.*

Сложение обозначается символом $\oplus$ и для всех $x, y \in \mathbb{R}_+$ определено как максимум: $x \oplus y = \max\{x, y\}$. Эта операция обладает свойством идемпотентности в силу того, что $x \oplus x = \max\{x, x\} = x$. Обратного по сложению (противоположного) элемента не существует, а потому операция вычитания в max-алгебре не определена.

Умножение определено и обозначается как обычно. Нейтральные элементы по сложению и умножению совпадают с арифметическими нулем и единицей. Понятия обратного элемента по умножению и степени, в том числе рациональной, числа имеют обычный смысл.

Векторные и матричные операции выполняются по стандартным правилам с заменой арифметического сложения на операцию $\oplus$. В частности, умножение вектора или матрицы на скаляр ничем не отличается от соответствующих операций в обычной арифметике. Нулевой вектор, который обозначается символом $\mathbf{0}$, нулевая матрица, а также положительный вектор имеют стандартный вид.

Для ненулевого вектора-столбца $\boldsymbol{x} = (x_j)$ определен мультипликативно сопряженный вектор-строка $\boldsymbol{x}^- = (x_j^-)$, где $x_j^- = x_j^{-1}$, если $x_j \neq 0$, и $x_j^- = 0$ в противном случае. Для вектора из единиц, который обозначается как $\mathbf{1}$, выполняется $\mathbf{1}^- = \mathbf{1}^{\mathrm{T}}$.

Мультипликативно сопряженное транспонирование преобразует ненулевую матрицу $\boldsymbol{A} = (a_{ij})$ в матрицу $\boldsymbol{A}^- = (a_{ij}^-)$, где $a_{ij}^- = a_{ji}^{-1}$, если $a_{ji} \neq 0$, иначе $a_{ij}^- = 0$.

Линейной комбинацией векторов $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ с коэффициентами $x_1, \ldots, x_n \in \mathbb{R}_+$ называется выражение $x_1\boldsymbol{a}_1 \oplus \cdots \oplus x_n\boldsymbol{a}_n$. Вектор $\boldsymbol{b}$ линейно зависит от векторов $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$, если существуют числа $x_1, \ldots, x_n \in \mathbb{R}_+$ такие, что выполняется равенство $\boldsymbol{b} = x_1\boldsymbol{a}_1 \oplus \cdots \oplus x_n\boldsymbol{a}_n$. Коллинеарность двух векторов имеет обычный смысл: векторы $\boldsymbol{a}$ и $\boldsymbol{b}$ являются коллинеарными, если $\boldsymbol{b} = x\boldsymbol{a}$ для некоторого $x \in \mathbb{R}_+$.

Множество всех линейных комбинаций $x_1\boldsymbol{a}_1 \oplus \cdots \oplus x_n\boldsymbol{a}_n$ векторов $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ образует тропическое линейное пространство. Любой вектор $\boldsymbol{y}$ пространства выражается с помощью (тропического) произведения матрицы $\boldsymbol{A} = (\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n)$, составленной из этих векторов как столбцов, и некоторого вектора $\boldsymbol{x} = (x_1, \ldots, x_n)^{\mathrm{T}}$ в виде $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$.

Рассмотрим квадратные матрицы с элементами из max-алгебры. Единичная матрица обозначается символом $\boldsymbol{I}$ и имеет обычный вид. Целая неотрицательная степень квадратной матрицы $\boldsymbol{A}$ обозначает (тропические) произведения матрицы на себя и определена для всех натуральных p так, что $\boldsymbol{A}^0 = \boldsymbol{I}$, $\boldsymbol{A}^p = \boldsymbol{A}^{p-1}\boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^{p-1}$.

След матрицы $\boldsymbol{A} = (a_{ij})$ порядка n вычисляется по формуле

$$\operatorname{tr} \boldsymbol{A} = a_{11} \oplus \cdots \oplus a_{nn}.$$

Спектральным радиусом матрицы $\boldsymbol{A}$ называется число, которое вычисляется по формуле

$$\lambda = \operatorname{tr} \boldsymbol{A} \oplus \cdots \oplus \operatorname{tr}^{1/n}(\boldsymbol{A}^n) = \bigoplus_{i=1}^{n} \operatorname{tr}^{1/i}(\boldsymbol{A}^i).$$

При условии, что $\lambda \leq 1$, определен оператор Клини (звезда Клини), который сопоставляет матрице $\boldsymbol{A}$ матрицу

$$\boldsymbol{A}^* = \boldsymbol{I} \oplus \boldsymbol{A} \oplus \cdots \oplus \boldsymbol{A}^{n-1} = \bigoplus_{i=0}^{n-1} \boldsymbol{A}^i.$$

Глава 3

Решение многокритериальной задачи парных сравнений

Далее приведен алгоритм решения использующий аппроксимацию матриц сравнений в log-чебышевской метрике, подробнее описанный в работах [2, 3, 4].

1. Для матрицы $\boldsymbol{C}$ находится спектральный радиус λ, составляется матрица $\lambda^{-1}\boldsymbol{C}$, а затем в параметрической форме определяется вектор весов критериев

$$\boldsymbol{w} = (\lambda^{-1}\boldsymbol{C})^{*}\boldsymbol{v}, \qquad \boldsymbol{v} > \boldsymbol{0}, \qquad \lambda = \bigoplus_{i=1}^{m} \mathrm{tr}^{1/i}(\boldsymbol{C}^{i}).$$

2. Если вектор $\boldsymbol{w}$ не единственный (с точностью до положительного множителя), то определяются наилучший и наихудший дифференцирующие векторы весов.

   2.1. Наилучший дифференцирующий вектор весов находится в параметрическом виде с использованием вектора параметров $\boldsymbol{v}_1$ по формуле:

$$\boldsymbol{w}_1 = \boldsymbol{P}(\boldsymbol{I} \oplus \boldsymbol{P}_{lk}^{-}\boldsymbol{P})\boldsymbol{v}_1, \qquad \boldsymbol{v}_1 > \boldsymbol{0},$$

   где матрица $\boldsymbol{P} = (\boldsymbol{p}_j)$ получена из $(\lambda^{-1}\boldsymbol{C})^{*}$ вычеркиванием линейно зависимых столбцов, матрица $\boldsymbol{P}_{lk}$ получена из $\boldsymbol{P} = (p_{ij})$ обнулением всех элементов, кроме p_{lk}, а индексы k и l определяются, исходя из условий:

$$k = \arg\max_{j} \mathbf{1}^{\mathrm{T}}\boldsymbol{p}_j\boldsymbol{p}_j^{-}\mathbf{1}, \qquad l = \arg\max_{i} p_{ik}^{-1}.$$

   2.2. Наихудший дифференцирующий вектор весов находится в параметрическом виде с использованием вектора параметров $\boldsymbol{v}_2$ по формулам:

$$\boldsymbol{w}_2 = (\Delta^{-1}\mathbf{1}\mathbf{1}^{\mathrm{T}} \oplus \lambda^{-1}\boldsymbol{C})^{*}\boldsymbol{v}_2, \qquad \boldsymbol{v}_2 > \boldsymbol{0}, \qquad \Delta = \mathbf{1}^{\mathrm{T}}(\lambda^{-1}\boldsymbol{C})^{*}\mathbf{1}.$$

3. С помощью векторов $\boldsymbol{w}_1 = (w_i^{(1)})$ и $\boldsymbol{w}_2 = (w_i^{(2)})$ строятся взвешенные суммы (или одна сумма, когда векторы совпадают) матриц парных сравнений альтернатив:

$$\boldsymbol{B} = \bigoplus_{i=1}^{m} w_i^{(1)}\boldsymbol{A}_i, \qquad \boldsymbol{D} = \bigoplus_{i=1}^{m} w_i^{(2)}\boldsymbol{A}_i.$$

4. Повторяя действия пунктов 1 и 2.1 (2.2) на основе взвешенной суммы $\boldsymbol{B}$ ($\boldsymbol{D}$) вычисляется вектор рейтингов альтернатив, соответствующий наилучшему (наихудшему) дифференцирующему вектору весов критериев.

Глава 4

Разработка программных средств

4.1. Разработка структуры для хранения чисел

В ходе решения есть шаг, на котором вычисляется линейно независимый набор векторов. При проверке линейной зависимости векторов недопустимо использование типов с плавающей точкой. Поэтому структура для хранения чисел должна быть основана на целочисленных типах, а операции сравнения должны быть точными.

В задаче принятия решений даются матрицы парных сравнений из натуральных и обратных натуральным чисел. Для аналитического решения задачи принятия решения структура должна поддерживать операцию умножения, извлечения корня n-ой степени и отношение линейного порядка. Рациональных чисел $\frac{a}{b}$ не достаточно из-за операции извлечения корня. Необходимо добавить к структуре числа корень целой степени: $\left(\frac{a}{b}\right)^{1/n}$.

Такое представление чисел в программе сужает max-алгебру с множества $\mathbb{R}_+$ на множество $\{x \in \mathbb{R}_+ \,|\, \exists a \in \mathbb{N} \cup 0, b \in \mathbb{N}, n \in \mathbb{N} : x = \left(\frac{a}{b}\right)^{1/n}\}$. Указанное множество замкнуто относительно операций умножения, извлечения корня целой степени, нахождения обратного элемента и линейно упорядочено.

С такой структурой операции и отношения определяются следующим образом:

- Умножение:
$$\left(\frac{a_1}{b_1}\right)^{1/n_1} \times \left(\frac{a_2}{b_2}\right)^{1/n_2} = \left(\frac{a_1^{n_2} a_2^{n_1}}{b_1^{n_2} b_2^{n_1}}\right)^{1/n_1 n_2}.$$
- Сравнение:
$$\left(\frac{a_1}{b_1}\right)^{1/n_1} < \left(\frac{a_2}{b_2}\right)^{1/n_2} \Leftrightarrow \left(\frac{a_1^{n_2}}{b_1^{n_2}}\right)^{1/n_1 n_2} < \left(\frac{a_2^{n_1}}{b_2^{n_1}}\right)^{1/n_1 n_2} \Leftrightarrow \frac{a_1^{n_2}}{b_1^{n_2}} < \frac{a_2^{n_1}}{b_2^{n_1}} \Leftrightarrow a_1^{n_2} b_2^{n_1} < a_2^{n_1} b_1^{n_2}.$$
- Обратный элемент относительно умножения:
$$\left(\left(\frac{a}{b}\right)^{1/n}\right)^{-1} = \left(\frac{b}{a}\right)^{1/n}, \qquad a \neq 0.$$

Однако, если использовать такие формулы, числа будут увеличиваться очень быстро. Причем, часто n_1 и n_2 оказываются равными. Это мотивирует использовать НОД в формулах:

$$n_1 = n_1^* \cdot \gcd(n_1, n_2), \qquad n_2 = n_2^* \cdot \gcd(n_1, n_2).$$

- Умножение:
$$\left(\frac{a_1}{b_1}\right)^{1/n_1} \times \left(\frac{a_2}{b_2}\right)^{1/n_2} = \left(\frac{a_1^{n_2^*} a_2^{n_1^*}}{b_1^{n_2^*} b_2^{n_1^*}}\right)^{1/n_1^* \cdot \gcd(n_1,n_2) \cdot n_2^*}.$$

  После умножения числитель и знаменатель сокращаются на их НОД.

- Сравнение:
$$\left(\frac{a_1}{b_1}\right)^{1/n_1} < \left(\frac{a_2}{b_2}\right)^{1/n_2} \Leftrightarrow a_1^{n_2^*} b_2^{n_1^*} < a_2^{n_1^*} b_1^{n_2^*}.$$

  Реализация в листинге 1.

4.2. Матрицы

Были реализованы элементы тропической математики такие, как нахождение следа, тропического определителя, транспонированный матрицы, спектрального радиуса, матрицы Клини, проверка линейной зависимости вектора от набора векторов, выбор ЛНЗ набора векторов из данных, нахождение лучших и худших дифференцирующих векторов в листинге 2.

4.3. Вывод решения

К каждому классу был добавлен метод вывода в latex в листинге 3.

Глава 5

Пример решения практической задачи

$\boldsymbol{C}$ — матрица парных сравнений критериев, $\boldsymbol{A}_i$ — матрицы парных сравнений альтернатив в соответствии с критерием i:

$$\boldsymbol{C} = \begin{pmatrix} 1 & 1/5 & 1/5 & 1 & 1/3 \\ 5 & 1 & 1/5 & 1/5 & 1 \\ 5 & 5 & 1 & 1/5 & 1 \\ 1 & 5 & 5 & 1 & 5 \\ 3 & 1 & 1 & 1/5 & 1 \end{pmatrix},$$

$$\boldsymbol{A}_1 = \begin{pmatrix} 1 & 3 & 7 & 9 \\ 1/3 & 1 & 6 & 7 \\ 1/7 & 1/6 & 1 & 3 \\ 1/9 & 1/7 & 1/3 & 1 \end{pmatrix}, \qquad \boldsymbol{A}_2 = \begin{pmatrix} 1 & 1/5 & 1/6 & 1/4 \\ 5 & 1 & 2 & 4 \\ 6 & 1/2 & 1 & 6 \\ 4 & 1/4 & 1/6 & 1 \end{pmatrix},$$

$$\boldsymbol{A}_3 = \begin{pmatrix} 1 & 7 & 7 & 1/2 \\ 1/7 & 1 & 1 & 1/7 \\ 1/7 & 1 & 1 & 1/7 \\ 2 & 7 & 7 & 1 \end{pmatrix}, \qquad \boldsymbol{A}_4 = \begin{pmatrix} 1 & 4 & 1/4 & 1/3 \\ 1/4 & 1 & 1/2 & 3 \\ 4 & 2 & 1 & 3 \\ 3 & 1/3 & 1/3 & 1 \end{pmatrix},$$

$$\boldsymbol{A}_5 = \begin{pmatrix} 1 & 1 & 7 & 4 \\ 1 & 1 & 6 & 3 \\ 1/7 & 1/6 & 1 & 1/4 \\ 1/4 & 1/3 & 4 & 1 \end{pmatrix}.$$

Нужные степени матрицы $\boldsymbol{C}$:

$$\boldsymbol{C}^2 = \begin{pmatrix} 1 & 5 & 5 & 1 & 5 \\ 5 & 1 & 1 & 5 & 5/3 \\ 25 & 5 & 1 & 5 & 5 \\ 25 & 25 & 5 & 1 & 5 \\ 5 & 5 & 1 & 3 & 1 \end{pmatrix}, \qquad \boldsymbol{C}^3 = \begin{pmatrix} 25 & 25 & 5 & 1 & 5 \\ 5 & 25 & 25 & 5 & 25 \\ 25 & 25 & 25 & 25 & 25 \\ 125 & 25 & 5 & 25 & 25 \\ 25 & 15 & 15 & 5 & 15 \end{pmatrix},$$

$$\boldsymbol{C}^4 = \begin{pmatrix} 125 & 25 & 5 & 25 & 25 \\ 125 & 125 & 25 & 5 & 25 \\ 125 & 125 & 125 & 25 & 125 \\ 125 & 125 & 125 & 125 & 125 \\ 75 & 75 & 25 & 25 & 25 \end{pmatrix}, \qquad \boldsymbol{C}^5 = \begin{pmatrix} 125 & 125 & 125 & 125 & 125 \\ 625 & 125 & 25 & 125 & 125 \\ 625 & 625 & 125 & 125 & 125 \\ 625 & 625 & 625 & 125 & 625 \\ 375 & 125 & 125 & 75 & 125 \end{pmatrix}.$$

Спектральный радиус матрицы $\boldsymbol{C}$:

$$\lambda_{\boldsymbol{C}} = \mathrm{tr}\boldsymbol{C} \oplus \cdots \oplus \mathrm{tr}^{1/5}(\boldsymbol{C}^5) = (125)^{1/4} \approx 3.3437.$$

Матрица $\lambda^{-1}\boldsymbol{C}$ и ее степени:

$$(\lambda^{-1}\boldsymbol{C})^1 = \begin{pmatrix} (1/125)^{1/4} & (1/78125)^{1/4} & (1/78125)^{1/4} & (1/125)^{1/4} & (1/10125)^{1/4} \\ (5)^{1/4} & (1/125)^{1/4} & (1/78125)^{1/4} & (1/78125)^{1/4} & (1/125)^{1/4} \\ (5)^{1/4} & (5)^{1/4} & (1/125)^{1/4} & (1/78125)^{1/4} & (1/125)^{1/4} \\ (1/125)^{1/4} & (5)^{1/4} & (5)^{1/4} & (1/125)^{1/4} & (5)^{1/4} \\ (81/125)^{1/4} & (1/125)^{1/4} & (1/125)^{1/4} & (1/78125)^{1/4} & (1/125)^{1/4} \end{pmatrix},$$

$$(\lambda^{-1}\boldsymbol{C})^2 = \begin{pmatrix} (1/15625)^{1/4} & (1/25)^{1/4} & (1/25)^{1/4} & (1/15625)^{1/4} & (1/25)^{1/4} \\ (1/25)^{1/4} & (1/15625)^{1/4} & (1/15625)^{1/4} & (1/25)^{1/4} & (1/2025)^{1/4} \\ (25)^{1/4} & (1/25)^{1/4} & (1/15625)^{1/4} & (1/25)^{1/4} & (1/25)^{1/4} \\ (25)^{1/4} & (25)^{1/4} & (1/25)^{1/4} & (1/15625)^{1/4} & (1/25)^{1/4} \\ (1/25)^{1/4} & (1/25)^{1/4} & (1/15625)^{1/4} & (81/15625)^{1/4} & (1/15625)^{1/4} \end{pmatrix},$$

$$(\lambda^{-1}\boldsymbol{C})^3 = \begin{pmatrix} (1/5)^{1/4} & (1/5)^{1/4} & (1/3125)^{1/4} & (1/1953125)^{1/4} & (1/3125)^{1/4} \\ (1/3125)^{1/4} & (1/5)^{1/4} & (1/5)^{1/4} & (1/3125)^{1/4} & (1/5)^{1/4} \\ (1/5)^{1/4} & (1/5)^{1/4} & (1/5)^{1/4} & (1/5)^{1/4} & (1/5)^{1/4} \\ (125)^{1/4} & (1/5)^{1/4} & (1/3125)^{1/4} & (1/5)^{1/4} & (1/5)^{1/4} \\ (1/5)^{1/4} & (81/3125)^{1/4} & (81/3125)^{1/4} & (1/3125)^{1/4} & (81/3125)^{1/4} \end{pmatrix},$$

$$(\lambda^{-1}\boldsymbol{C})^4 = \begin{pmatrix} (1)^{1/4} & (1/625)^{1/4} & (1/390625)^{1/4} & (1/625)^{1/4} & (1/625)^{1/4} \\ (1)^{1/4} & (1)^{1/4} & (1/625)^{1/4} & (1/390625)^{1/4} & (1/625)^{1/4} \\ (1)^{1/4} & (1)^{1/4} & (1)^{1/4} & (1/625)^{1/4} & (1)^{1/4} \\ (1)^{1/4} & (1)^{1/4} & (1)^{1/4} & (1)^{1/4} & (1)^{1/4} \\ (81/625)^{1/4} & (81/625)^{1/4} & (1/625)^{1/4} & (1/625)^{1/4} & (1/625)^{1/4} \end{pmatrix}.$$

Матрица Клини:

$$(\lambda^{-1}\boldsymbol{C})^* = \boldsymbol{I} \oplus (\lambda^{-1}\boldsymbol{C})^1 \oplus (\lambda^{-1}\boldsymbol{C})^2 \oplus (\lambda^{-1}\boldsymbol{C})^3 \oplus (\lambda^{-1}\boldsymbol{C})^4 =$$

$$= \begin{pmatrix} 1 & (1/5)^{1/4} & (1/25)^{1/4} & (1/125)^{1/4} & (1/25)^{1/4} \\ (5)^{1/4} & 1 & (1/5)^{1/4} & (1/25)^{1/4} & (1/5)^{1/4} \\ (25)^{1/4} & (5)^{1/4} & 1 & (1/5)^{1/4} & (1)^{1/4} \\ (125)^{1/4} & (25)^{1/4} & (5)^{1/4} & 1 & (5)^{1/4} \\ (81/125)^{1/4} & (81/625)^{1/4} & (81/3125)^{1/4} & (81/15625)^{1/4} & 1 \end{pmatrix}.$$

Линейно независимые столбцы:

$$\boldsymbol{P} = \begin{pmatrix} 1 & (1/25)^{1/4} \\ (5)^{1/4} & (1/5)^{1/4} \\ (25)^{1/4} & (1)^{1/4} \\ (125)^{1/4} & (5)^{1/4} \\ (81/125)^{1/4} & 1 \end{pmatrix}.$$

Лучший и худший дифференцирующие векторы весов критериев.

$$\boldsymbol{w}_1 = \begin{pmatrix} (1/125)^{1/4} \\ (1/25)^{1/4} \\ (1/5)^{1/4} \\ (1)^{1/4} \\ (81/15625)^{1/4} \end{pmatrix}, \qquad \boldsymbol{w}_2 = \begin{pmatrix} (1/125)^{1/4} & (1/125)^{1/4} \\ (1/25)^{1/4} & (1/25)^{1/4} \\ (1/5)^{1/4} & (1/5)^{1/4} \\ (1)^{1/4} & (1)^{1/4} \\ (1/125)^{1/4} & (1/5)^{1/4} \end{pmatrix}.$$

Взвешенные суммы матриц парных сравнений векторов совпали:

$$\boldsymbol{B} = \boldsymbol{D} = \begin{pmatrix} (1)^{1/4} & (2401/5)^{1/4} & (2401/5)^{1/4} & (6561/125)^{1/4} \\ (25)^{1/4} & (1)^{1/4} & (1296/125)^{1/4} & (81)^{1/4} \\ (256)^{1/4} & (16)^{1/4} & (1)^{1/4} & (81)^{1/4} \\ (81)^{1/4} & (2401/5)^{1/4} & (2401/5)^{1/4} & (1)^{1/4} \end{pmatrix}.$$

Спектральный радиус матрицы $\boldsymbol{B}$:

$$\lambda_{\boldsymbol{B}} = \mathrm{tr}\boldsymbol{B} \oplus \cdots \oplus \mathrm{tr}^{1/4}(\boldsymbol{B}^4) = (614656/5)^{1/8} \approx 4.32721.$$

Матрица $\lambda^{-1}\boldsymbol{B}$:

$$\lambda^{-1}\boldsymbol{B} = \begin{pmatrix} (5/614656)^{1/8} & (2401/1280)^{1/8} & (2401/1280)^{1/8} & (43046721/1920800000)^{1/8} \\ (3125/614656)^{1/8} & (5/614656)^{1/8} & (6561/7503125)^{1/8} & (32805/614656)^{1/8} \\ (1280/2401)^{1/8} & (5/2401)^{1/8} & (5/614656)^{1/8} & (32805/614656)^{1/8} \\ (32805/614656)^{1/8} & (2401/1280)^{1/8} & (2401/1280)^{1/8} & (5/614656)^{1/8} \end{pmatrix}.$$

Матрица Клини:

$$(\lambda^{-1}\boldsymbol{B})^* = \boldsymbol{I} \oplus (\lambda^{-1}\boldsymbol{B})^1 \oplus (\lambda^{-1}\boldsymbol{B})^2 \oplus (\lambda^{-1}\boldsymbol{B})^3 =$$

$$= \begin{pmatrix} 1 & (2401/1280)^{1/8} & (2401/1280)^{1/8} & (6561/65536)^{1/8} \\ (32805/614656)^{1/8} & 1 & (6561/65536)^{1/8} & (32805/614656)^{1/8} \\ (1280/2401)^{1/8} & (1)^{1/8} & 1 & (32805/614656)^{1/8} \\ (1)^{1/8} & (2401/1280)^{1/8} & (2401/1280)^{1/8} & 1 \end{pmatrix}.$$

Линейно независимые столбцы:

$$\boldsymbol{P} = \begin{pmatrix} 1 & (2401/1280)^{1/8} & (6561/65536)^{1/8} \\ (32805/614656)^{1/8} & 1 & (32805/614656)^{1/8} \\ (1280/2401)^{1/8} & (1)^{1/8} & (32805/614656)^{1/8} \\ (1)^{1/8} & (2401/1280)^{1/8} & 1 \end{pmatrix}.$$

Лучший дифференцирующий вектор:

$$\boldsymbol{w}_{best} = \begin{pmatrix} 1 & (6561/65536)^{1/8} \\ (32805/614656)^{1/8} & (32805/614656)^{1/8} \\ (1280/2401)^{1/8} & (32805/614656)^{1/8} \\ (1)^{1/8} & (1)^{1/8} \end{pmatrix} \boldsymbol{v}_1, \qquad \boldsymbol{v}_1 > \boldsymbol{0}.$$

Худший дифференцирующий вектор.

$$\boldsymbol{w}_{worst} = \begin{pmatrix} 1 \\ (1280/2401)^{1/8} \\ (1280/2401)^{1/8} \\ (1)^{1/8} \end{pmatrix} \boldsymbol{v}_2, \qquad \boldsymbol{v}_2 > \boldsymbol{0}.$$

Ответ:

$$\boldsymbol{w}_{best} \approx \begin{pmatrix} 1.000000 & 0.750000 \\ 0.693288 & 0.693288 \\ 0.924384 & 0.693288 \\ 1.000000 & 1.000000 \end{pmatrix} \boldsymbol{v}_1, \qquad \boldsymbol{v}_1 > \boldsymbol{0}, \qquad \boldsymbol{w}_{worst} \approx \begin{pmatrix} 1.000000 \\ 0.924384 \\ 0.924384 \\ 1.000000 \end{pmatrix} \boldsymbol{v}_2, \qquad \boldsymbol{v}_2 > \boldsymbol{0}.$$

Заключение

С такой неинтуитивной алгеброй приятно иметь калькулятор.

В ходе решения задачи принятия решений числа могут стать очень большими, что может быть проблемой при больших размерностях входных матриц. Уже разработана более оптимизированная для max-умножить алгебры структура и ведется ее реализация.

Разработанная структура может пригодиться и в других областях. Например, отсутствие ошибок округления важно дли криптографии.

Список литературы

1. О решении многокритериальных задач принятия решений на основе парных сравнений / Кривулин Н. К., Абильдаев Т., Горшечникова В. Д., Капаца Д., Магдич Е. А. и Мандрикова А. А. // Компьютерные инструменты в образовании. — 2020. — № 2. — С. 27–58. — Режим доступа: http://cte.eltech.ru/ojs/index.php/kio/article/view/1655.
2. Кривулин Н. К., Агеев В. А. Методы тропической оптимизации в многокритериальных задачах оценки альтернатив на основе парных сравнений // Вестн. С.-Петерб. ун-та. Прикл. матем. — 2019. — Т. 15, № 4. — С. 472–488.
3. Krivulin N., Sergeev S. Tropical implementation of the Analytical Hierarchy Process decision method // Fuzzy Sets and Systems. — 2019. — Vol. 377. — P. 31–51. — 1802.01989.
4. Krivulin N., Prinkov A., Gladkikh I. Using pairwise comparisons to determine consumer preferences in hotel selection // Mathematics. — 2022. — Vol. 10, no. 5. — P. 1–25.

Листинг 1. fraction.h

```
#pragma once

#include <string>
#include <assert.h>
#include <iostream>
#include <utility> //pair
#include <numeric> //gcd
#include <cmath>   //pow
#include <cln/integer.h>

class MaxMultiFraction
{
public:
    friend std::string to_string(const MaxMultiFraction &fraction);
    friend std::string to_latex(const MaxMultiFraction &fraction);

    MaxMultiFraction(cln::cl_I numerator, cln::cl_I denominator, uint root = 1)
        : numerator_(numerator),
          denominator_(denominator),
          root_(root)
    {
    }

    MaxMultiFraction(double num = 0)
        : numerator_(1),
          denominator_(1),
          root_(1)
    {
        if (fmod(num, 1) == 0)
        {
            numerator_ = int64_t(num);
        }
        else
        {
            if (fmod(1.0 / num, 1) != 0)
            {
                cout << "fraction constructor error\n"
                     << num << endl;
            }
```

```
            assert(fmod(1.0 / num, 1) == 0);
            denominator_ = int64_t(1 / num);
        }
    }

    MaxMultiFraction(string str)
        : numerator_(1),
            denominator_(1),
            root_(1)
    {
        auto pos = str.find("/");
        if (pos == str.npos)
        {
            numerator_ = str.c_str();
        }
        else
        {
            numerator_ = str.substr(0, pos).c_str();
            denominator_ = str.substr(pos + 1).c_str();
        }
    }

    bool operator<(const MaxMultiFraction &other) const
    {
        auto tmp = Transform(*this, other);
        return tmp.first < tmp.second;
    }
    bool operator<=(const MaxMultiFraction &other) const
    {
        auto tmp = Transform(*this, other);
        return tmp.first <= tmp.second;
    }
    bool operator>(const MaxMultiFraction &other) const
    {
        auto tmp = Transform(*this, other);
        return tmp.first > tmp.second;
    }
    bool operator>=(const MaxMultiFraction &other) const
    {
        auto tmp = Transform(*this, other);
```

```
    return tmp.first >= tmp.second;
}
bool operator==(const MaxMultiFraction &other) const
{
    auto tmp = Transform(*this, other);
    return tmp.first == tmp.second;
}
bool operator!=(const MaxMultiFraction &other) const
{
    auto tmp = Transform(*this, other);
    return tmp.first != tmp.second;
}

MaxMultiFraction operator+=(const MaxMultiFraction &other)
{
    if (*this < other)
    {
        *this = other;
    }
    return *this;
}
MaxMultiFraction operator+(const MaxMultiFraction &other) const
{
    return MaxMultiFraction(*this) += other;
}

MaxMultiFraction operator*=(const MaxMultiFraction &other)
{
    uint root_gcd = std::gcd(root_, other.root_);
    numerator_ = FastPow(numerator_, other.root_ / root_gcd) *
                 FastPow(other.numerator_, root_ / root_gcd);
    denominator_ = FastPow(denominator_, other.root_ / root_gcd) *
                 FastPow(other.denominator_, root_ / root_gcd);
    root_ *= other.root_ / root_gcd;

    Simplify();

    return *this;
}
MaxMultiFraction operator*(const MaxMultiFraction &other) const
```

```
{
    return MaxMultiFraction(*this) *= other;
}

MaxMultiFraction operator/=(const MaxMultiFraction &other)
{
    uint root_gcd = std::gcd(root_, other.root_);
    numerator_ = FastPow(numerator_, other.root_ / root_gcd) *
                 FastPow(other.denominator_, root_ / root_gcd);
    denominator_ = FastPow(denominator_, other.root_ / root_gcd) *
                   FastPow(other.numerator_, root_ / root_gcd);
    root_ *= other.root_ / root_gcd;

    Simplify();

    return *this;
}
MaxMultiFraction operator/(const MaxMultiFraction &other) const
{
    return MaxMultiFraction(*this) /= other;
}

MaxMultiFraction Root(uint root) const
{
    return MaxMultiFraction(numerator_,
                            denominator_,
                            root_ * root)
        .Simplify();
}
MaxMultiFraction Pow(uint pow) const
{
    uint root_pow_gcd = std::gcd(root_, pow);
    pow /= root_pow_gcd;
    return MaxMultiFraction(FastPow(numerator_, pow),
                            FastPow(denominator_, pow),
                            root_ / root_pow_gcd);
}

operator double() const
{
```

```
        return std::pow(cln::double_approx(numerator_)
            / cln::double_approx(denominator_), 1.0 / root_);
    }

private:
    cln::cl_I numerator_;
    cln::cl_I denominator_;
    uint root_;

    MaxMultiFraction Simplify()
    {
        cln::cl_I num_den_gcd = cln::gcd(numerator_, denominator_);
        numerator_ = cln::exquo(numerator_, num_den_gcd);
        denominator_ = cln::exquo(denominator_, num_den_gcd);

        return *this;
    }

    cln::cl_I FastPow(cln::cl_I base, cln::cl_I exp) const
    {
        cln::cl_I result = 1;
        while (exp > 0)
        {
            if (cl_I_to_int(exp) & 1)
            {
                result *= base;
            }
            base *= base;
            exp >>= 1;
        }
        return result;
    }

    std::pair<cln::cl_I, cln::cl_I> Transform(const MaxMultiFraction &first,
                                              const MaxMultiFraction &second) const
    {
        uint gcd_root = std::gcd(first.root_, second.root_);
        return {FastPow(first.numerator_, second.root_ / gcd_root) *
                    FastPow(second.denominator_, first.root_ / gcd_root),
                FastPow(second.numerator_, first.root_ / gcd_root) *
```

```
                    FastPow(first.denominator_, second.root_ / gcd_root)};
    }
};
```

Листинг 2. matrix.h

```
#pragma once

#include <vector>
#include <assert.h>
#include <iostream>
#include <string>

using namespace std;

template <typename T>
class Matrix;
template <typename T>
Matrix<T> Identity(uint size);
template <typename T>
Matrix<T> Ones(uint rows, uint cols);

template <typename T>
class Matrix
{
public:
    Matrix(uint rows, uint cols, T value = 0)
        : matrix_(rows, std::vector<T>(cols, value)) {}
    Matrix(const std::vector<std::vector<T>> &matrix)
        : matrix_(matrix) {}
    Matrix(std::initializer_list<std::vector<T>> matrix)
        : matrix_(matrix)
    {
        // std::cout << "initializer_list T" << std::endl;

        for (uint i = 0; i < rows(); i++)
        {
            assert(cols() == matrix_[i].size());
        }
    }
    template <typename U>
```

```
Matrix(const Matrix<U> &other) : Matrix(other.rows(), other.cols())
{
    // std::cout << "Matrix U" << std::endl;

    for (uint i = 0; i < rows(); i++)
    {
        for (uint j = 0; j < cols(); j++)
        {
            matrix_[i][j] = other[i][j];
        }
    }
}

std::vector<T> &operator[](uint i)
{
    return matrix_[i];
}
const std::vector<T> &operator[](uint i) const
{
    return matrix_[i];
}

Matrix operator+=(const Matrix &other)
{
    assert(rows() == other.rows());
    assert(cols() == other.cols());
    for (uint i = 0; i < rows(); i++)
    {
        for (uint j = 0; j < cols(); j++)
        {
            matrix_[i][j] += other[i][j];
        }
    }
    return *this;
}
Matrix operator+(const Matrix &other) const
{
    return Matrix(*this) += other;
}
```

```
Matrix operator-() const
{
    Matrix result(rows(), cols());
    for (uint i = 0; i < rows(); i++)
    {
        for (uint j = 0; j < cols(); j++)
        {
            result[i][j] = -matrix_[i][j];
        }
    }
    return result;
}
Matrix operator-=(const Matrix &other)
{
    assert(rows() == other.rows());
    assert(cols() == other.cols());
    for (uint i = 0; i < rows(); i++)
    {
        for (uint j = 0; j < cols(); j++)
        {
            matrix_[i][j] -= other[i][j];
        }
    }
    return *this;
}
Matrix operator-(const Matrix &other) const
{
    return Matrix(*this) -= other;
}

Matrix operator*=(const Matrix &other)
{
    assert(cols() == other.rows());
    Matrix result(rows(), other.cols());
    for (uint i = 0; i < result.rows(); i++)
    {
        for (uint j = 0; j < result.cols(); j++)
        {
            for (uint k = 0; k < cols(); k++)
            {
```

```
                result[i][j] += matrix_[i][k] * other[k][j];
            }
        }
    }
    matrix_ = std::move(result.matrix_);
    return *this;
}
Matrix operator*(const Matrix &other) const
{
    return Matrix(*this) *= other;
}

Matrix operator*=(const T &scalar)
{
    for (uint i = 0; i < rows(); i++)
    {
        for (uint j = 0; j < cols(); j++)
        {
            matrix_[i][j] *= scalar;
        }
    }
    return *this;
}
Matrix operator*(const T &scalar) const
{
    return Matrix(*this) *= scalar;
}

Matrix operator/=(const T &scalar)
{
    for (uint i = 0; i < rows(); i++)
    {
        for (uint j = 0; j < cols(); j++)
        {
            matrix_[i][j] /= scalar;
        }
    }
    return *this;
}
Matrix operator/(const T &scalar) const
```

```
{
    return Matrix(*this) /= scalar;
}

Matrix Transpose() const
{
    Matrix result(cols(), rows());
    for (uint i = 0; i < rows(); i++)
    {
        for (uint j = 0; j < cols(); j++)
        {
            result[j][i] = (matrix_[i][j] == T{0} ? T{0}
                                                  : T{1} / matrix_[i][j]);
        }
    }
    return result;
}

T Trace() const
{
    assert(rows() == cols());
    T result;
    for (uint i = 0; i < cols(); i++)
    {
        result += matrix_[i][i];
    }
    return result;
}

T Determinant() const
{
    T result{Trace()};
    Matrix tmp(*this);
    for (uint i = 1; i < cols(); i++)
    {
        tmp *= *this;
        result += tmp.Trace();
    }
    return result;
}
```

```
T SpectralRadius() const
{
    T result{Trace()};
    Matrix tmp(*this);
    for (uint i = 2; i <= cols(); i++)
    {
        tmp *= *this;
        result += tmp.Trace().Root(i);
    }
    return result;
}

Matrix Kleene() const
{
    Matrix tmp{*this};
    Matrix result{Identity<T>(cols())};
    for (uint i = 1; i < cols(); i++)
    {
        result += tmp;
        tmp *= *this;
    }
    return result;
}

bool isLinearlyDependent(const Matrix &b)
{
    assert(b.cols() == 1);
    Matrix &A = *this;
    Matrix result((A * (b.Transpose() * A).Transpose()).Transpose() * b);

    return result[0][0] == T{1};
}

Matrix Span()
{
    Matrix result(getCol(0));
    for (uint j = 1; j < cols(); j++)
    {
        auto tmp = getCol(j);
```

```
            if (!result.isLinearlyDependent(tmp))
            {
                result.cbind(tmp);
            }
        }
        return result;
    }

    Matrix BestVector()
    {
        T lambda = SpectralRadius();
        Matrix P((*this / lambda).Kleene().Span());

        // uint k = 0;
        vector<uint> k;
        T max_value = -1;
        for (uint j = 0; j < P.cols(); j++)
        {
            Matrix col_j(P.getCol(j));
            T tmp = (col_j * col_j.Transpose()).sum();
            if (tmp > max_value)
            {
                // k = j;
                k.clear();
                max_value = tmp;
            }
            if (tmp == max_value)
            {
                k.push_back(j);
            }
        }
        vector<uint> l(k.size(), 0);
        for (uint it = 0; it < k.size(); it++)
        {
            for (uint i = 0; i < P.rows(); i++)
            {
                if (P[i][k[it]] < P[l[it]][k[it]])
                {
                    l[it] = i;
                }
```

```
        }
    }

    Matrix result(P * (Identity<T>(P.cols()) +
                       P.filter(l[0], k[0]).Transpose() * P));
    for (uint i = 1; i < k.size(); i++)
    {
        result.cbind(P * (Identity<T>(P.cols()) +
                          P.filter(l[1], k[1]).Transpose() * P));
    }

    return result.Span().normCol();
}

Matrix WorstVector()
{
    T lambda(SpectralRadius());
    Matrix kleene((*this / lambda).Kleene());
    T Delta = kleene.sum();

    return (Matrix(rows(), cols(), T{1} / Delta)
            + *this / lambda).Kleene().Span().normCol();
}

Matrix cbind(const Matrix &other)
{
    assert(rows() == other.rows());
    for (uint i = 0; i < rows(); i++)
    {
        matrix_[i].insert(matrix_[i].end(), other[i].begin(),
                                            other[i].end());
    }
    return *this;
}

Matrix rbind(const Matrix &other)
{
    assert(cols() == other.cols());
    matrix_.insert(matrix_.end(), other.matrix_.begin(),
                                  other.matrix_.end());
```

```
    return *this;
}

Matrix getCol(uint j)
{
    assert(j < cols());
    Matrix result(rows(), 1);
    for (uint i = 0; i < rows(); i++)
    {
        result[i][0] = matrix_[i][j];
    }
    return result;
}

Matrix filter(uint i, uint j) const
{
    assert(i < rows());
    assert(j < cols());
    Matrix result(rows(), cols());
    result[i][j] = (*this)[i][j];
    return result;
}

Matrix norm()
{
    return *this / sum();
}

Matrix normCol()
{
    Matrix result(*this);
    for (uint col = 0; col < result.cols(); col++)
    {
        T max_in_col;
        for (uint row = 0; row < result.rows(); row++)
        {
            max_in_col += result[row][col];
        }
        for (uint row = 0; row < result.rows(); row++)
        {
```

```
                result[row][col] /= max_in_col;
            }
        }
        return result;
    }

    T sum()
    {
        return (Ones<T>(1, rows()) * (*this) * Ones<T>(cols(), 1))[0][0];
    }

    uint rows() const
    {
        return matrix_.size();
    }
    uint cols() const
    {
        return matrix_.front().size();
    }

private:
    std::vector<std::vector<T>> matrix_;
};

template <typename T>
Matrix<T> operator*(const T &scalar, const Matrix<T> &matrix)
{
    return Matrix(matrix) *= scalar;
}

template <typename T>
Matrix<T> Identity(uint size)
{
    Matrix<T> result(size, size, T{0});
    for (uint i = 0; i < size; i++)
    {
        result[i][i] = T{1};
    }
    return result;
}
```

```
template <typename T>
Matrix<T> Ones(uint rows, uint cols)
{
    Matrix<T> result(rows, cols, T{1});
    return result;
}
```

Листинг 3. to_latex.h

```
#pragma once

#include <iostream>
#include <string>
#include <sstream>
#include "matrix.h"
#include "fraction.h"
#include <cln/integer_io.h>
#include <eigen3/Eigen/Dense>

using std::to_string;

std::string equation(std::string str)
{
    // return "\\begin{equation*}\n" +
    //        str +
    //        "\\end{equation*}";

    return "$$" + str + "$$";
}

template <typename T>
std::ostream &operator<<(std::ostream &out, const Matrix<T> &matrix)
{
    std::cout << to_string(matrix) << std::endl;
    return out;
}

std::ostream &operator<<(std::ostream &out, const MaxMultiFraction &fraction)
{
    std::cout << to_string(fraction);
```

```
    return out;
}

template <typename T>
std::string to_string(const Matrix<T> &matrix)
{
    std::string result;
    for (uint i = 0; i < matrix.rows(); i++)
    {
        if (i != 0)
        {
            result += "\n";
        }
        for (uint j = 0; j < matrix.cols(); j++)
        {
            if (j != 0)
            {
                result += "␣";
            }
            result += to_string(matrix[i][j]);
        }
    }
    return result;
}

std::string to_string(const cln::cl_I &integer)
{
    std::stringstream ss;
    cln::print_integer(ss, 10, integer);

    return ss.str();
}

std::string to_string(const MaxMultiFraction &fraction)
{
    std::string result = to_string(fraction.numerator_);
    if (fraction.denominator_ != 1)
    {
        result += "/";
        result += to_string(fraction.denominator_);
```

```
    }

    if (fraction.root_ != 1)
    {
        result = "(" + result + ")^(1/" + to_string(fraction.root_) + ")";
    }
    return result;
}

template <typename T>
std::string to_latex(const T &obj, std::string name = "")
{
    return name + (name != "" ? " = " : "") + to_string(obj);
}

template <typename T>
std::string to_latex(const Matrix<T> &matrix)
{
    std::string result;
    result += "\\begin{pmatrix}\n";
    for (uint i = 0; i < matrix.rows(); i++)
    {
        if (i != 0)
        {
            result += "\\\\\n";
        }
        for (uint j = 0; j < matrix.cols(); j++)
        {
            if (j != 0)
            {
                result += " & ";
            }
            result += to_latex(matrix[i][j]);
        }
    }
    result += "\n\\end{pmatrix}\n";

    return result;
}
```

```
// template <>
std::string to_latex(const MaxMultiFraction &fraction)
{
    std::string result = to_string(fraction.numerator_);
    if (fraction.denominator_ != 1)
    {
        result = result + "/" + to_string(fraction.denominator_);
    }

    if (fraction.root_ != 1)
    {
        result = "(" + result + ")^{1/" + to_string(fraction.root_) + "}";
    }
    return result;
}

// template <typename T>
// template <>
std::string to_latex(const Eigen::MatrixXd &matrix)
{
    std::string result;
    result += "\\begin{pmatrix}\n";
    for (uint i = 0; i < matrix.rows(); i++)
    {
        if (i != 0)
        {
            result += "\\\\\n";
        }
        for (uint j = 0; j < matrix.cols(); j++)
        {
            if (j != 0)
            {
                result += " & ";
            }
            result += to_latex(matrix(i, j));
        }
    }
    result += "\n\\end{pmatrix}\n";

    return result;
```

```
}

// template <typename scalar>
std::string to_latex(const Eigen::VectorXd &vector)
{
    std::string result;
    result += "\\begin{pmatrix}\n";
    for (uint i = 0; i < vector.rows(); i++)
    {
        if (i != 0)
        {
            result += "\\\\\n";
        }
        result += to_latex(vector(i));
    }
    result += "\n\\end{pmatrix}\n";

    return result;
}
```