用吸收马克洛夫链解 Doomsday Fuel 问题

Riguz Lee · 2022-06-22 · 刨根问底, 算法

Contents

问题描述 **1**
Doomsday Fuel 1
问题分析 2
马尔科夫链 **3**
代码实现 **5**
分式计算 5
最大公约数、最小公倍数计算 6
分式四则运算及约分 6
矩阵运算 7
矩阵减法 7
矩阵乘法 8
余子式及代数余子式 8
行列式的值 8
矩阵的逆 9
马尔科夫链计算 9
结果约分 11
Test Case 3 11

这是在做 Google foobar 挑战的第三关时遇到的问题，相对于前两关而言，这个问题的难度明显增大了不少，需要一定的数学知识才能完成。

问题描述

Doomsday Fuel

Making fuel for the LAMBCHOP's reactor core is a tricky process because of the exotic matter involved. It starts as raw ore, then during processing, begins randomly changing between forms, eventually reaching a stable form. There may be multiple stable forms that a sample could ultimately reach, not all of which are useful as fuel.

Commander Lambda has tasked you to help the scientists increase fuel creation efficiency by predicting the end state of a given ore sample. You have carefully studied the different structures that the ore can take and which transitions it undergoes. It appears that, while random, the probability of each structure transforming is fixed. That is, each time the ore is in 1 state, it has the same probabilities of entering the next state (which might be the same state). You have recorded the observed transitions in a matrix. The others in the lab have hypothesized more exotic forms that the ore can become, but you haven't seen all of them.

Write a function solution(m) that takes an array of array of nonnegative ints representing how many times that state has gone to the next state and return an array of ints for each terminal state giving the exact probabilities of each terminal state, represented as the numerator for each state, then the denominator for all of them at the end and in simplest form. The matrix is at most 10 by 10. It is guaranteed that no matter which state the ore is in, there is a path from that state to a terminal state. That is, the processing will always eventually end in a stable state. The ore starts in state 0.

The denominator will fit within a signed 32-bit integer during the calculation, as long as the fraction is simplified regularly.

For example, consider the matrix m:

```
[
  [0,1,0,0,0,1],  # s0, the initial state, goes to s1 and s5 with equal
probability
  [4,0,0,3,2,0],  # s1 can become s0, s3, or s4, but with different
probabilities
  [0,0,0,0,0,0],  # s2 is terminal, and unreachable (never observed in
practice)
  [0,0,0,0,0,0],  # s3 is terminal
  [0,0,0,0,0,0],  # s4 is terminal
  [0,0,0,0,0,0],  # s5 is terminal
]
```

So, we can consider different paths to terminal states, such as:

- s0 -> s1 -> s3
- s0 -> s1 -> s0 -> s1 -> s0 -> s1 -> s4
- s0 -> s1 -> s0 -> s5

Tracing the probabilities of each, we find that

- s2 has probability 0
- s3 has probability 3/14
- s4 has probability 1/7
- s5 has probability 9/14

So, putting that together, and making a common denominator, gives an answer in the form of [s2.numerator, s3.numerator, s4.numerator, s5.numerator, denominator] which is [0, 3, 2, 9, 14].

问题分析

这个问题前面的描述几乎可以忽略，可以简化一下：

- 有一系列的状态，如 s0, s1, ..., s5
- 一个状态以固定的概率迁移到另一个状态，采用矩阵的形式描述了不同状态间迁移的概率。注意矩阵里面给的是次数而不是概率，比如第一列数据 [0,1,0,0,0,1]，那么 s0 迁移到 s1 和 s5 的概率就是 1/2, 1/2
- 终止状态不能迁移到其他状态

那么，题目最后要求得出的概率是怎么算出来的？s2 很简单，没有可达的路径，肯定为 0。但是 s3 为啥是 3/14？

从数据可以看出，要到达 s3，除了从 s1 直接到 s3 外，s0 -> s1 本身是存在一个环的，可以重复很多次：

- s0 -> s1 -> s3，概率为$\frac{1}{2} \times \frac{3}{9} = \frac{1}{6}$
- s0 -> s1 -> s0 -> s1 -> s3, 概率为$\frac{1}{2} \times \frac{4}{9} \times \frac{1}{2} \times \frac{3}{9} = \frac{2}{9} \times \frac{1}{6}$
- s0 -> s1 -> s0 -> s1 -> s0 -> s1-> s3, 概率为$\frac{1}{2} \times \frac{4}{9} \times \frac{1}{2} \times \frac{4}{9} \times \frac{1}{2} \times \frac{3}{9} = \left(\frac{2}{9}\right)^2 \times \frac{1}{6}$
- ...

因此，综合所有情况，s3 对应的概率为:

$$\begin{aligned}\sum_{i=0}^{n}\frac{1}{6}\times\left(\frac{2}{9}\right)^{n} &= \frac{1}{6}\times\sum_{i=0}^{n}\left(\frac{2}{9}\right)^{n}\\ &= \frac{1}{6}\times\left(\frac{1}{1-\frac{2}{9}}\right)\\ &= \frac{1}{6}\times\frac{9}{7}\\ &= \frac{3}{14}\end{aligned} \tag{1}$$

这里用到了等比数列的无限项之和公式，即:

$$\begin{aligned}&S_n = a + ar + ar^2 + ... + ar^{n-1} ..., r \in (-1, 1)\\ &S_\infty = \frac{a}{1-r}\end{aligned} \tag{2}$$

这样倒是能够算出来概率，但是如何用程序计算呢? 这可难倒我了，正常的思维是用有向图来处理，根据图计算每一条可达路径上，并将其概率相加得到最终的概率。但是这个是有环的图, 这种图怎么样计算路径, 并计算概率呢? 思来想去也没有找到一个方法可以直接计算。倒是可以将有环图去掉环，变成无环图，然后进行一些特殊标记之类的，但处理之后是否正确，又如何能够适应各种情况（一环套一环），是很难去证明的。

因此我意识到，通过以上的数学知识是难以解决这个问题的。

马尔科夫链

马尔科夫链描述的是一个状态迁移到另一个状态时的概率是确定的，仅跟之前的状态有关。而解决这个问题可以用到马尔科夫链(Markov Chain)的一个特殊版本，吸收马尔科夫链(Absorbing Markov Chain)。吸收马尔科夫链是指所有状态最终能到达一个“吸收态”，一旦到达这个状态之后，就不能离开（是不是像个黑洞?）

吸收马尔科夫链需要满足如下条件:

- 存在至少一个吸收态
- 从任意一个状态都可以经过有限步到达一个吸收态

这样再回过头来看题目，是不是完全满足吸收马尔科夫链的条件!

吸收马尔科夫链的转移矩阵 P 可以表示为:

$$P = \begin{pmatrix} Q & R \\ 0 & I_r \end{pmatrix} \tag{3}$$

设马尔科夫链中有 t 个瞬时态，r 个吸收态，则:

- Q 为 txt 的瞬时态转移概率
- R 为 txr 瞬时态到吸收态的概率
- I 为 rxr 单位矩阵

吸收态马尔科夫链有一个很重要的性质，就是吸收态的概率可以通过下面的公式直接计算出来：

$$
\begin{aligned}
N &= (I_t - Q)^{-1} \\
B &= NR
\end{aligned}
\tag{4}
$$

以上面的题目为例，

```
[0,1,0,0,0,1],  # s0, the initial state, goes to s1 and s5 with equal probability
[4,0,0,3,2,0],  # s1 can become s0, s3, or s4, but with different probabilities
[0,0,0,0,0,0],  # s2 is terminal, and unreachable (never observed in practice)
[0,0,0,0,0,0],  # s3 is terminal
[0,0,0,0,0,0],  # s4 is terminal
[0,0,0,0,0,0],  # s5 is terminal
```

s0, s1 为瞬时态， s2, s3, s4, s5 为吸收态。有：

$$
\begin{aligned}
Q &= \begin{pmatrix} 0 & \frac{1}{2} \\ \frac{4}{9} & 0 \end{pmatrix} \\
R &= \begin{pmatrix} 0 & 0 & 0 & \frac{1}{2} \\ 0 & \frac{3}{9} & \frac{2}{9} & 0 \end{pmatrix} \\
I &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}
\end{aligned}
\tag{5}
$$

则可计算出：

$$
\begin{aligned}
N &= (I_t - Q)^{-1} \\
&= \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{pmatrix} 0 & \frac{1}{2} \\ \frac{4}{9} & 0 \end{pmatrix} \right)^{-1} \\
&= \begin{pmatrix} 1 & -\frac{1}{2} \\ -\frac{4}{9} & 1 \end{pmatrix}^{-1} \\
&= \frac{1}{1 \times 1 - \left((-\frac{1}{2}) \times (-\frac{4}{9}) \right)} \begin{pmatrix} 1 & \frac{1}{2} \\ \frac{4}{9} & 1 \end{pmatrix} \\
&= \frac{9}{7} \begin{pmatrix} 1 & \frac{1}{2} \\ \frac{4}{9} & 1 \end{pmatrix} \\
&= \begin{pmatrix} \frac{9}{7} & \frac{9}{14} \\ \frac{4}{7} & \frac{9}{7} \end{pmatrix}
\end{aligned}
\tag{6}
$$

从而，得出吸收态的概率为：

$$
\begin{aligned}
B &= NR \\
&= \begin{pmatrix} \frac{9}{7} & \frac{9}{14} \\ \frac{4}{7} & \frac{9}{7} \end{pmatrix} \times \begin{pmatrix} 0 & 0 & 0 & \frac{1}{2} \\ 0 & \frac{3}{9} & \frac{2}{9} & 0 \end{pmatrix} \\
&= \begin{pmatrix} 0 & \frac{3}{14} & \frac{1}{7} & \frac{9}{14} \\ 0 & \frac{3}{7} & \frac{2}{7} & \frac{2}{7} \end{pmatrix}
\end{aligned} \tag{7}
$$

如上，第一列和第二列分别代表从 s0 和 s1 两个瞬时态开始到达吸收态的概率。因此，第一列就是题目所要求解的结果了。再看另一个例子，

```
Input:
Solution.solution({
  {0, 2, 1, 0, 0},
  {0, 0, 0, 3, 4},
  {0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0}})
Output:
    [7, 6, 8, 21]
```

同样，先计算：

$$
\begin{aligned}
Q &= \begin{pmatrix} 0 & \frac{2}{3} \\ 0 & 0 \end{pmatrix} \\
R &= \begin{pmatrix} \frac{1}{3} & 0 & 0 \\ 0 & \frac{3}{7} & \frac{4}{7} \end{pmatrix} \\
I_t &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\
N &= \begin{pmatrix} 1 & \frac{2}{3} \\ 0 & 1 \end{pmatrix} \\
B &= \begin{pmatrix} \frac{1}{3} & \frac{2}{7} & \frac{8}{21} \\ 0 & \frac{3}{7} & \frac{4}{7} \end{pmatrix}
\end{aligned} \tag{8}
$$

代码实现

原理已经掌握，但是要用代码实现仍然十分麻烦，主要是涉及到很多矩阵计算，以及一些其他的数学运算。因此，在实际开始计算马尔科夫链之前，需要写一些辅助类来。

分式计算

首先是需要将题目中所有的运算都转化为分式运算，最终的结果也是通过分式表示的，而不是浮点数。另外，即便允许使用浮点数，在计算过程中直接用浮点数计算也会导致累积误差。

```
public class Fraction {
    private final int numerator;
    private final int denominator;
```

```
    public Fraction(int numerator, int denominator) {
        if (denominator == 0) {
            throw new IllegalArgumentException("Denominator should not
be zero");
        }
        this.numerator = numerator;
        this.denominator = denominator;
    }
}
```

分式直接表示为分子/分母的形式即可。

最大公约数、最小公倍数计算

分式的计算本身非常简单，但是需要对分式进行约分，就需要来计算最大公约数(GCD)、最小公倍数（LCM）了。使用欧几里得法计算 GCD（又称为辗转相除法）：

$$gcd(a,b) = gcd(b, a\%b) \tag{9}$$

```
public static int gcd(int m, int n) {
    return m % n == 0 ? n : gcd(n, m % n);
}

public static int lcm(int m, int n) {
    return m * n / gcd(m, n);
}
```

分式四则运算及约分

在以上的基础上就可以实现分式的四则运算及约分操作了。

```
public Fraction reduce() {
    int gcd = Math.gcd(numerator, denominator);
    int n = numerator / gcd, d = denominator / gcd;
    boolean swap = d < 0;
    return new Fraction(swap ? -n : n, swap ? -d : d);
}

public Fraction negative() {
    return new Fraction(-numerator, denominator);
}

public Fraction reciprocal() {
    return new Fraction(denominator, numerator);
}

public Fraction convert(int common) {
    int factor = common / denominator;
    return new Fraction(numerator * factor, common);
}

public Fraction add(Fraction another) {
    int common = Math.lcm(denominator, another.denominator);
    Fraction a = convert(common), b = another.convert(common);
```

```
    return new Fraction(a.numerator + b.numerator,
            common)
            .reduce();
}

public Fraction subtract(Fraction another) {
    return add(another.negative());
}

public Fraction multiply(Fraction another) {
    return new Fraction(numerator * another.numerator,
            denominator * another.denominator)
            .reduce();
}

public Fraction divide(Fraction another) {
    return multiply(another.reciprocal());
}
```

矩阵运算

因全部使用分式来进行计算，所以矩阵可以定义为:

```
public class Matrix {
    private final Fraction[][] matrix;
    private final int rows;
    private final int columns;

    public Matrix(int rows, int columns) {
        this.rows = rows;
        this.columns = columns;
        this.matrix = new Fraction[rows][columns];
    }
}
```

矩阵减法

我们需要用到矩阵的减法，减法十分简单，直接对于每一个元素减去对应的位置的元素即可:

```
public Matrix subtract(Matrix another) {
    if (this.rows != another.rows || this.columns != another.columns) {
        throw new IllegalArgumentException("Unable to add two matrix
with different size");
    }
    Fraction[][] result = new Fraction[rows][columns];
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            result[i][j] = matrix[i][j].subtract(another.matrix[i][j]);
        }
    }
    return new Matrix(result, rows, columns);
}
```

矩阵乘法

矩阵乘法就麻烦一点，是以左侧矩阵的行，乘以右侧列的积再加到一起，成为一个元素。

```
public Matrix multiply(Matrix another) {
    if (this.columns != another.rows) {
        throw new IllegalArgumentException("Unable to add two matrix
with different size");
    }
    Matrix r = new Matrix(rows, another.columns);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < another.columns; j++) {
            Fraction sum = new Fraction(0, 1);
            for (int m = 0; m < columns; m++) {
                sum = sum.add(matrix[i][m].multiply(another.matrix[m]
[j]));
            }
            r.matrix[i][j] = sum;
        }
    }
    return r;
}
```

余子式及代数余子式

要计算矩阵的逆及矩阵对应行列式的值，需要计算其余子式。余子式就是去除矩阵中 m,n 对应的行和列之后得到的一个子矩阵：

```
public Matrix complementMinor(int row, int column) {
    Matrix m = new Matrix(rows - 1, columns - 1);
    int rowOffset = 0;
    for (int i = 0; i < rows; i++) {
        if (i == row) {
            continue;
        }
        int columnOffset = 0;
        for (int j = 0; j < columns; j++) {
            if (j == column) {
                continue;
            }
            m.getMatrix()[rowOffset][columnOffset] = matrix[i][j];
            columnOffset++;
        }
        rowOffset++;
    }
    return m;
}
```

而代数余子式，则需要乘以$(-1)^{m+n}$即可。

行列式的值

矩阵对应的行列式的值，等于其任意一列（或者行）的元素乘以对应的代数余子式的值。因此行列式的值可以递归计算得到：

```
public Fraction determinantValue() {
    if (rows != columns || rows < 1) {
        throw new IllegalArgumentException("Not supported");
    }
    if (rows == 1) {
        return matrix[0][0].reduce();
    } else if (rows == 2) {
        return matrix[0][0].multiply(matrix[1][1]).subtract(matrix[0]
[1].multiply(matrix[1][0]));
    } else {
        Fraction sum = new Fraction(0, 1);
        for (int i = 0; i < columns; i++) {
            Fraction sign = Fraction.of(Math.pow(-1, i));
            sum = sum.add(matrix[0]
[i].multiply(sign.multiply(complementMinor(0, i).determinantValue())));
        }
        return sum;
    }
}
```

矩阵的逆

矩阵的逆稍微麻烦，可以通过伴随矩阵的方式来计算：

```
public Matrix adjugateMatrix() {
    Matrix m = new Matrix(rows, columns);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            Fraction sign = Fraction.of(Math.pow(-1, (i + j)));
            m.matrix[j][i] = sign.multiply(complementMinor(i,
j).determinantValue());
        }
    }
    return m;
}

public Matrix inverse() {
    Fraction f = determinantValue();
    Matrix adjugateMatrix = adjugateMatrix();
    Matrix m = new Matrix(rows, columns);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            m.matrix[i][j] = adjugateMatrix.matrix[i][j].divide(f);
        }
    }
    return m;
}
```

马尔科夫链计算

在有以上的辅助类的基础上，要计算马尔科夫链的吸收态概率就直接套公式就可以了。但是这一个题目还有一些地方需要处理才能得到最终结果。 ### 计算基础矩阵 虽然题目中的例子上，吸收态是在后面的，但是其实还有一些隐含的测试用

例是乱序的。因此，如果要计算出基础矩阵,首先需要对原始的概率矩阵进行排序，让瞬时态在前，吸收态在后。

```
public static int[] solution(int[][] nums) {
    int totalStates = nums.length, absorbingStates = 0;

    Set<Integer> absorbingStateIds = new HashSet<>();
    for (int i = 0; i < totalStates; i++) {
        if (sum(nums[i]) == 0) {
            absorbingStateIds.add(i);
        }
    }
    absorbingStates = absorbingStateIds.size();
    int[][] transformed = new int[totalStates][totalStates];

    int [] indexMapping = new int[totalStates];
    int offset = 0;
    for (int i = 0; i < totalStates; i++) {
        if(!absorbingStateIds.contains(i))
            indexMapping[offset++] = i;
    }
    for(int id: absorbingStateIds){
        indexMapping[offset++] = id;
    }

    for(int i = 0; i < totalStates; i++) {
        for(int j = 0; j < totalStates; j++) {
            transformed[i][j] = nums[indexMapping[i]][indexMapping[j]];
        }
    }
    nums = transformed;
    // ...
}
```

应该有更简洁的办法来做这个事情，但因为这个并不是十分重要的步骤，因此偷懒用最搓的办法实现。然后，要求出基础矩阵就十分容易：

```
Matrix matrixQ = new Matrix(transientStates, transientStates),
        matrixR = new Matrix(transientStates, absorbingStates),
        matrixI = new Matrix(transientStates, transientStates);

for (int i = 0; i < transientStates; i++) {
    int common = sum(nums[i]);
    for (int j = 0; j < transientStates; j++) {
        matrixQ.getMatrix()[i][j] = new Fraction(nums[i][j], common);
        matrixI.getMatrix()[i][j] = i == j ? new Fraction(1, 1) :
                new Fraction(0, 1);
    }
}

for (int i = 0; i < transientStates; i++) {
    int common = sum(nums[i]);
```

```
    for (int j = 0; j < absorbingStates; j++) {
        matrixR.getMatrix()[i][j] = new Fraction(nums[i][j +
transientStates], common);
    }
}
Matrix matrixN = matrixI.subtract(matrixQ).inverse();
Matrix matrixB = matrixN.multiply(matrixR);
```

结果约分

对计算结果，因为是分式的形式表示，要处理成同一个分母的形式。因此，还需要计算出多项结果的最小公倍数，并将分式转换为相同的分母：

```
int[] result = new int[matrixB.getColumns() + 1];
int lcm = 0;
for (int i = 0; i < matrixB.getColumns(); i++) {
    if (matrixB.getMatrix()[0][i].getNumerator() == 0) {
        continue;
    }
    if (lcm == 0) {
        lcm = matrixB.getMatrix()[0][i].getDenominator();
    } else {
        lcm = Math.lcm(lcm, matrixB.getMatrix()[0][i].getDenominator());
    }
}
for (int i = 0; i < matrixB.getColumns(); i++) {
    result[i] = matrixB.getMatrix()[0][i].getNumerator() * lcm /
            matrixB.getMatrix()[0][i].getDenominator();
}
result[result.length - 1] = lcm;
```

Test Case 3

在以上实现的基础上，就可以通过除了 Test Case 3 之外的所有测试了。但是，有一个 Test Case 3 却通不过，也无法得知具体是什么测试用例：

```
foobar:~/doomsday-fuel solee.linux$ verify Solution.java
Verifying solution...
Test 1 passed!
Test 2 passed!
Test 3 failed  [Hidden]
Test 4 passed! [Hidden]
Test 5 passed! [Hidden]
Test 6 passed! [Hidden]
Test 7 passed! [Hidden]
Test 8 passed! [Hidden]
Test 9 passed! [Hidden]
Test 10 passed! [Hidden]
```

原来，还有极端的场景没有考虑到，就是如果 s0 就是终止态的情况。这种情况作为一个特殊前提处理即可，十分简单，就到此为止吧！