行列累乗を用いた動的計画法の高速化

埼玉県立大宮高等学校
黒瀬 諭一朗

第１章　概要

本研究は、計算機科学における動的計画法（Dynamic Programming）の計算時間を、行列演算を用いて劇的に短縮する手法「行列累乗」について、その理論的背景と実用性を検証したものである。

一般に、漸化式を用いた動的計画法では、第n項を求めるために$O(n)$の計算量を要する。しかし、nが10^{18}といった巨大な数値になる場合、この線形時間のアルゴリズムでは現実的な時間内での計算が不可能となる。本研究では、線形漸化式を行列の積として表現し、繰り返し二乗法を応用することで計算量を$O(\log n)$まで圧縮する方法に着目した。

Codeforces の問題を題材に、通常の動的計画法と行列累乗を用いた手法の実装を行い、実行時間を比較した結果、$n=10^{18}$の場合、通常の手法では380年程度かかると推定される計算が、行列累乗を用いることで$1.0\times10^{-5}[s]$未満で完了することを確認した。一方で、$n\leq100$程度の小規模な入力においては、行列演算のオーバーヘッドにより、逆に計算速度が低下するという現象も観測された。

本稿では、これらの理論的導出から実験による実証までを体系的に論じ、大規模な数値計算における線形代数の応用可能性を示す。

第２章　研究の背景と目的

多くの動的計画法（DP）は、直前の状態から次の状態を決定する漸化式に基づいている。漸化式を用いる数値計算において、第n項を求めるのに要する計算量は$O(n)$であり、$n\leq10^{9}$程度であれば、現代のコンピュータを用いて数秒以内に計算が可能である。

しかし、暗号理論や大規模シミュレーションの分野では、$n=10^{18}$といった巨大なステップ数を扱う場合がある。このようにnが巨大になると、$O(n)$のアルゴリズムでは計算終了までに莫大な時間を要することになり、事実上計算不可能である。そこで、行列の積と指数法則を応用したアルゴリズムが、競技プログラミングでは重要視されている。

本研究の目的は、行列累乗を用いた動的計画法の高速化について、その理論的背景を整理するとともに、実用性を評価することである。具体的には、まず線形漸化式が行列積に帰着できる数学的構造を示し、計算量が削減される原理を解説する。次に、具体的な問題に対して通常の動的計画法と行列累乗を用いた手法の双方をC++で実装し、入力の大きさに応じた実行時間を比較することで、その実用性を検証する。

第３章　行列累乗の基礎

第１節　繰り返し二乗法

通常、ある値xのn乗を求めるには、xをn回掛け合わせる必要があり、計算量は$O(n)$となる。そこで、指数法則と2進展開を利用して計算回数を劇的に減らすアルゴリズムが、繰り返し二乗法である。

任意の正の整数nは、2の累乗和として一意に表すことができる（2進展開）。

$$n=2^{k_1}+2^{k_2}+\ldots+2^{k_m}\quad(0\leq k_1<k_2<\ldots<k_m)$$

とすると、x^nは

$$x^n=x^{2^{k_1}+2^{k_2}+\ldots+2^{k_m}}=x^{2^{k_1}}x^{2^{k_2}}\ldots x^{2^{k_m}}$$

と変形できる。

したがって、$x^{2^1},x^{2^2},\ldots$を順次計算し、nの2進数表現においてビットが立っている桁に対応するx^{2^k}を掛け合せることにより、x^nを求めることができる。nを2進数表記したときの桁数は$\lfloor\log_2 n\rfloor+1$であるため、計算量は$O(\log n)$となる。

第２節　行列の積

$m \times n$行列Aと$n \times p$行列Bをそれぞれの成分を用いて$A=(a_{ij}), B=(b_{ij})$と表すとき、積ABは$m \times p$行列となり、その(i, j)成分c_{ij}は

$$c_{ij}=\sum_{k=1}^{n} a_{ik} b_{kj}$$

で定義される。すなわち、行列の積の(i, j)成分は、行列Aの第i行ベクトルと行列Bの第j列ベクトルの内積として計算される。

$N \times N$行列同士の積を計算する場合、成分の数はN^2個あり、各成分の計算にN回の乗算と加算が必要となるため、計算量は$O(N^3)$となる。

定義式におけるΣの構造は、「iからkを経由してjに到達する」という遷移の数え上げと密接に関係しており、これが動的計画法に行列を利用できる数学的根拠となっている。

また、行列の積において結合法則が成り立つ。この性質により、行列Aをn回掛け合わせる際、どのような順序で演算を行っても結果が一意に定まる。したがって、繰り返し二乗法を用いてることで、$N \times N$行列Aをn掛け合わせたA^nを$O(N^3 \log n)$で計算できる。

第４章　行列累乗の応用

第１節　線形漸化式の第n項

線形漸化式の第n項を高速に計算する手法として、行列累乗を応用する。

（１）フィボナッチ数列

線形漸化式の基本的な例として、フィボナッチ数列について考える。フィボナッチ数列の漸化式を

$$F_0=0, F_1=1, F_{n+2}=F_{n+1}+F_n$$

とする。これは、行列を用いて

$$\begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix}=\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$$

と表せる。この2×2行列をAとおくと、数学的帰納法により

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}=A^n\begin{pmatrix} F_1 \\ F_0 \end{pmatrix}=A^n\begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

が成立する。

したがって、繰り返し二乗法を用いてA^nを求めることにより、計算量$O(\log n)$でF_nを計算できる。

（２）一般の線形漸化式

次に、一般の線形漸化式について考える。$k+1$項間線形漸化式を

$$a_{n+k}=\sum_{i=1}^{k} b_{k-i} a_{n+k-i}$$

とする。これは、行列を用いて

$$\begin{pmatrix} a_{n+k} \\ a_{n+k-1} \\ a_{n+k-2} \\ \vdots \\ a_{n+1} \end{pmatrix}=\begin{pmatrix} b_{k-1} & b_{k-2} & \cdots & b_1 & b_0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}\begin{pmatrix} a_{n+k-1} \\ a_{n+k-2} \\ a_{n+k-3} \\ \vdots \\ a_n \end{pmatrix}$$

と表せる。

したがって、この$k \times k$行列をn乗することで、$O(k^3 \log n)$でa_nを計算できる。

漸化式に定数項Cが含まれる。つまり漸化式が

$$a_{n+k}=\sum_{i=1}^{k} b_{k-i} a_{n+k-i}+C$$

である場合は、行列を用いて

$$\begin{pmatrix} a_{n+k} \\ a_{n+k-1} \\ a_{n+k-2} \\ \vdots \\ a_{n+1} \\ 1 \end{pmatrix} = \begin{pmatrix} b_{k-1} & b_{k-2} & \dots & b_1 & b_0 & C \\ 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_{n+k-1} \\ a_{n+k-2} \\ a_{n+k-3} \\ \vdots \\ a_n \\ 1 \end{pmatrix}$$

と表せる。

したがって、この$(k+1)\times(k+1)$行列をn乗することで、定数項を含む場合も同様に$O(k^3 \log n)$でa_nを計算できる。

第２節　グラフの長さkのパスの総数

各辺の重みが1である有向グラフ上の、長さkのパスの総数を高速に計算する手法として、行列累乗を応用する。

頂点数Nの有向グラフGを考える。Gの各辺の重みを1とし、隣接行列をAとする。

ここで、頂点iから頂点jへ至る長さkのパスの総数が、A^kの(i, j)成分$(A^k)_{ij}$と等しくなることを数学的帰納法で示す。

まず、$k=1$の場合、A^1はグラフGの隣接行列を表すため明らかに成立する。

次に、$k=l$で成立すると仮定する。つまり、頂点iから頂点jへ至る長さlのパスの総数が$(A^l)_{ij}$であると仮定する。頂点iから頂点jへ至る長さ$l+1$のパスの総数は、頂点iから頂点pへ長さlで到達し、頂点pから頂点jへ長さ1で到達する場合の数を、すべての頂点pについて足し合わせることで得られるため、

$$\sum_{p=1}^{N} (A^l)_{ip} A_{pj}$$

となる。これは$(A^{l+1})_{ij}$と表せるので、任意の自然数kにおいて、頂点iから頂点jへ至る長さkのパスの総数が、A^kの(i, j)成分$(A^k)_{ij}$と等しくなることが示された。

したがって、グラフ全体における長さkのパスの総数は、行列A^kの全成分の和

$$\sum_{i=1}^{N} \sum_{j=1}^{N} (A^k)_{ij}$$

として、$O(N^3 \log k)$で計算できる。

第３節　行列の累乗和

$N\times N$行列Aの累乗和

$$S_n = \sum_{i=1}^{n} A^i$$

を高速に計算する手法として、行列累乗を応用する。

単純に$A^1, A^2, \dots, A^n$を足し合わせる方法では、全体の計算量は$O(N^3 n)$となり、繰り返し二乗法による高速化の恩恵が得られない。そこで、累乗和の計算を行列累乗の形に帰着させることを考える。

まず、数列Tを、

$$T_n = \sum_{i=0}^{n-1} A^i$$

と定める。このとき、Tは

$$T_{n+1} = A^n + T_n$$

という漸化式で表せる。$N\times N$零行列をO、$N\times N$単位行列をIとすると、

$$\begin{pmatrix} A^{n+1} \\ T_{n+1} \end{pmatrix} = \begin{pmatrix} A & O \\ I & I \end{pmatrix} \begin{pmatrix} A^n \\ T_n \end{pmatrix}$$

という関係式が導ける。この$2N \times 2N$行列をMとおくと、数学的帰納法により

$$\begin{pmatrix} A^n \\ T_n \end{pmatrix} = M^n \begin{pmatrix} A^0 \\ T_0 \end{pmatrix} = M^n \begin{pmatrix} I \\ O \end{pmatrix}$$

が成立する。したがって、繰り返し二乗法を用いてM^nを計算することで、$O\left(N^3 \log n\right)$でT_nを計算できる。

すると、累乗和S_nは、

$$S_n = T_{n+1} - I$$

として求められる。

第５章　行列累乗による DP 高速化の実践

Codeforces Round 113 (Div.2)で出題された問題「Tetrahedron」を、高速化の対象として取り上げる。対象とした問題の公式解説では動的計画法による解法が採用されていたが、これは行列累乗を用いて解くこともできる。そこで本章では、それぞれの解法の実行時間を比較検証することで、行列累乗の実用性について考察する。

第１節　問題の概要

蟻が四面体 ABCD の頂点 D に立っている。蟻は、「立っている頂点から別の頂点へ1歩で移動する」という行動を繰り返す。蟻が頂点 D からn歩で頂点 D に戻る経路の総数を10^9+7で割った余りを求めよ。

nは自然数とする。なお、原題での制約は$1 \le n \le 10^7$であるが、本研究ではアルゴリズムの性能差を検証するため、$1 \le n \le 10^{18}$とする。

第２節　動的計画法による解法

頂点 D からn歩で頂点 D へ至る経路の総数をdpD_nとする。四面体の対称性により、頂点 D からn歩で頂点 A,B,C へ至る経路の総数はそれぞれ等しいので、この値を$dpABC_n$とする。このとき、dpD , $dpABC$ は

$$dpD_0 = 1 \text{ , } dpABC_0 = 0$$
$$dpD_{n+1} = 3\, dpABC_n$$
$$dpABC_{n+1} = dpD_n + 2\, dpABC_n$$

という漸化式で表せる。よって以下の解法が記述できる。計算量は$O\left(n\right)$である。

```
#include <iostream>

const int MOD = 1000000007;

int main() {
  long long n;
  std::cin >> n;
  long long dpD0 = 1;
  long long dpABC0 = 0;
  for (long long i = 0; i < n; i++) {
    long long dpD1 = dpABC0 * 3 % MOD;
    long long dpABC1 = (dpABC0 * 2 + dpD0) % MOD;
    dpD0 = dpD1;
    dpABC0 = dpABC1;
  }
  std::cout << dpD0 << std::endl;
  return 0;
}
```

第３節　行列累乗を用いた解法

上記の漸化式は行列を用いて

$$\begin{pmatrix} dpD_{n+1} \\ dpABC_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 3 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} dpD_n \\ dpABC_n \end{pmatrix}$$

と表せる。この2×2行列をAとおくと、数学的帰納法により

$$\begin{pmatrix} dpD_n \\ dpABC_n \end{pmatrix} = A^n \begin{pmatrix} dpD_0 \\ dpABC_0 \end{pmatrix} = A^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

が成立する。よって以下の解法が記述できる。計算量は$O(\log n)$である。

```
#include <iostream>
#include <array>

using mat = std::array<std::array<long long, 2>, 2>;
const int MOD = 1000000007;

mat mul(mat A, mat B) {
  mat C{};
  for (int i = 0; i < 2; i++) {
    for (int k = 0; k < 2; k++) {
      for (int j = 0; j < 2; j++) {
        C[i][j] = (C[i][j] + A[i][k] * B[k][j]) % MOD;
      }
    }
  }
  return C;
}

mat pow(mat A, long long n) {
  mat B{{{1, 0}, {0, 1}}};
  while (n > 0) {
    if (n & 1) {
      B = mul(B, A);
    }
    A = mul(A, A);
    n >>= 1;
  }
  return B;
}

int main() {
  long long n;
  std::cin >> n;
  mat A{{{0, 3}, {1, 2}}};
  A = pow(A, n);
  std::cout << A[0][0] << std::endl;
  return 0;
}
```

第４節　実行時間の比較検証

それぞれの解法の性能を、プログラムの実行時間を測定することで検証する。

（１）計測環境

CPU：12th Gen Intel(R) Core(TM) i7-1260P (16) @ 4.70 GHz

OS：Arch Linux x86_64

プログラミング言語：C++

コンパイラ：g++ (GCC) 15.2.1

（２）計測方法

上記の２つのプログラムにおいて$n=10^0, 10^1, 10^2, \ldots$としたときの、入力後から出力前までの実行時間を std::chrono::steady_clock を用いて計測する。各nに対して5回計測し、その中央値を測定値とする。

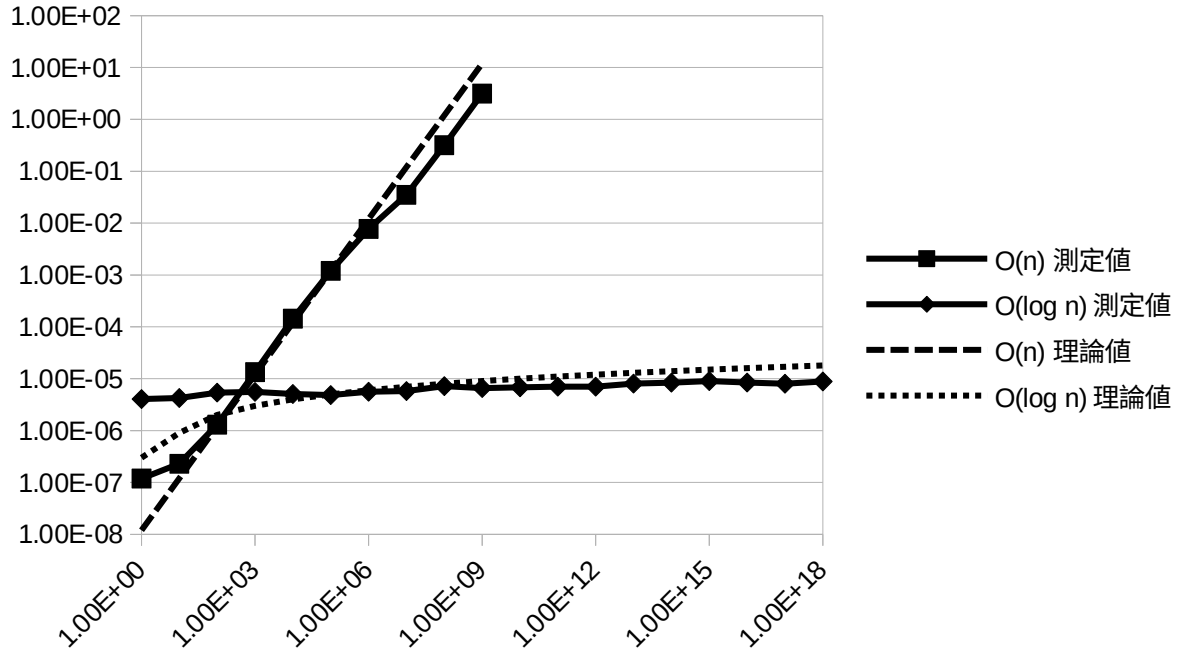
（３）計測結果

横軸にnの値、縦軸に実行時間[s]を両対数グラフでとり、各解法における測定値を描画すると以下のようになった。ただし、$O(n)$解法では$n=10^{10}$以降は現実的な時間内に終了しないため打ち切りとした。

理論値の算出にあたり、$O(n)$、$O(\log n)$の実行時間はそれぞれ、定数k_1, k_2を用いて

$$T_1(n)=k_1 n[s], T_2(n)=k_2 \log_2 n[s]$$

と表せると仮定した。ここで、k_1は単純な整数の演算コスト、k_2は行列演算のオーバーヘッドを反映した定数である。本稿では$n=10^5$における測定値を基準に$k_1 \approx 1.2\times10^{-8}, k_2 \approx 3.0\times10^{-7}$と設定した。



第５節　考察

（１）実行時間の逆転

グラフの$n \leq 100$の領域に注目すると、$O(\log n)$の実行時間が$O(n)$の実行時間を上回っていることが確認できる。これは、$O(\log n)$解法が1回のステップごとに行列の乗算を行うため、$O(\log n)$の定数係数k_2が$O(n)$の定数係数k_1よりも大きくなるためである。

したがって、小規模なnに対しては、定数倍が支配的となり、単純な DP の方が高速になる場合がある。この事実は、実用的な実装において、入力の大きさに応じたアルゴリズムの使い分けが重要であることを示唆している。

（２）$O(n)$の限界と$O(\log n)$の優位性

$O(n)$の実行時間は$n = 10^9$を超えると急激に増加し、実用的な限界を迎える。仮に$n = 10^{18}$を$O(n)$で計算した場合、その実行時間は

$$T_1(10^{18}) \approx 1.2 \times 10^{-8} \times 10^{18}[s] = 1.2 \times 10^{10}[s]$$

と推定される。これは約380年に相当し、計算資源がいかに豊富でも現実的な時間内での計算は不可能である。対して$O(\log n)$解法は$n = 10^{18}$であっても$1.0 \times 10^{-5}[s]$未満で計算を完了しており、強力な手法であることが実証された。

第６章　結論

本研究では、動的計画法の計算量を削減する手法として行列累乗に着目し、その理論的背景の整理と実測による検証を行った。具体的には、線形漸化式を行列積の形に帰着させ、繰り返し二乗法を用いることで、計算量を$O(n)$から$O(k^3 \log n)$へと改善できることを示した。

実測の結果、$n = 10^{18}$という巨大な入力に対し、$O(n)$解法では約380年を要すると推定される計算が、行列累乗を用いた解法では$1.0 \times 10^{-5}[s]$未満で完了することを確認した。これは理論値通り$O(\log n)$の有効性を実証するものである。一方で、$n \leq 100$程度の小規模な入力においては、行列演算のオーバーヘッドにより$O(n)$解法の方が高速になるという逆転現象も観測された。

以上の結果から、行列累乗を用いた動的計画法の高速化は、大規模なnに対して極めて強力な手法であると結論付けられる。

しかし、常に最良の手法というわけではなく、入力の大きさに応じて、単純な DP と行列累乗を適切に使い分けることが、アルゴリズムの実装において重要である。

また、本手法の計算量は行列の大きさkに対して、$O\left(k^3\right)$で増加する。DP の状態数が行列の大きさに直結するため、状態数が多い問題設定では計算量が急激に高まることにも留意が必要である。

第 7 章　参考文献

[1] 秋葉拓哉・岩田陽一・北川宜稔 . プログラミングコンテストチャレンジブック . 株式会社マイナビ出版 , 2022 . [2] 高校数学の美しい物語 . “行列の積の定義とその理由 | 高校数学の美しい物語” . https://manabitimes.jp/math/1023 , (参照 2025-12-16) . [3] Codeforces . “Problem - E – Codeforces” . https://codeforces.com/contest/166/problem/E , (参照 2025-12-15) . [4] Codeforces . “Codeforces Round #113 (Div. 2) Tutorial – Codeforces” . https://codeforces.com/blog/entry/4173 , (参照 2025-12-15) .