REKURENCYJNE ALGORYTMY MNOŜENIA MACIERZY

JAN SMÓŁKA

Abstract. Niniejszy dokument zawiera sprawozdanie z wykonania ćwiczenia 1. w ramach przedmiotu Algorytmy Macierzowe w semestrze zimowym roku akademickiego 2023/24.

1. Algorytmy

W ramach zadania zaimplementowano trzy algorytmy mnożenia macierzy: metodą Bineta, Strassena oraz sposobem odkrytym w 2021 roku przez model sztucznej inteligencji AlphaTensor, w ramach projektu DeepMind.

1.1. **Metoda Bineta.** Algorytm rekurencyjny operuje na macierzach klatkowych 2×2, jednak łatwo przekształcić go do postaci iteracyjnej dla macierzy kwadratowych o rozmiarze będącym wielokrotnością pewnej arbitralnej liczby, zwanej wielkością bloku. Macierze o rozmiarach poniżej wielkości bloku mnożone są algorytmem naiwnym przez procedurę `MULTIPLY_BLOCKS`.

Algorithm 1 Mnożenie macierzy klatkowych metodą Bineta

Require: $A, B \in M_{n\times n} \wedge m \mid n \wedge m \in \mathbb{N}_+$
Ensure: $C \in M_{n\times n}, n_b = m/n$

```
for i ← 1...n_b do
    for j ← 1...n_b do
        for k ← 1...n_b do
            MULTIPLY_BLOCKS(A[i,k], B[k,j], C[i,j])
        end for
    end for
end for

function MULTIPLY_BLOCKS(A[ ][ ], B[ ][ ], C[ ][ ])
    for i ← 1...m do
        for j ← 1...m do
            for k ← 1...m do
                C[i,j] ← A[i,k]Ḃ[k,j]
            end for
        end for
    end for
end function
```

Odwołanie za pomocą indeksu (np. $A[i,j]$) w głównej części algorytmu należy rozumieć jako klatkę A_{ij}, zaś w procedurze `MULTIPLY_BLOCKS` - jako liczbę pod indeksem.

1.2. **Metoda Strassena.** Algorytm Strassena nie poddaje się podobnemu przekształceniu co algorytm Bineta ze względu na nietrywialny charakter obliczeń, dlatego zostanie przedstawiony w postaci rekurencyjnej, dla macierzy klatkowych 2×2, równoważnych kwadratowym macierzom liczbowym o rozmiarach będących potęgami 2.

Algorithm 2 Mnożenie macierzy klatkowych metodą Strassena

Require: $A, B \in M_{2\times 2}(M_{n\times n}(\mathbb{R})) \wedge \exists k \in \mathbb{N}_+ : n = 2^k$
Ensure: $C \in M_{2\times 2}(M_{n\times n}(\mathbb{R}))$
  function STRASSEN(A, B)
    if $A, B \in M_{2\times 2}(\mathbb{R})$ **then**
      return $A\dot{B}$ // Multiply trivially
    end if

    $M_1 \leftarrow$ `STRASSEN`$(A[1,1] + A[2,2], B[1,1] + B[2,2])$
    $M_2 \leftarrow$ `STRASSEN`$(A[2,1] + A[2,2], B[1,1])$
    $M_3 \leftarrow$ `STRASSEN`$(A[1,1], B[1,2] - B[2,2])$
    $M_4 \leftarrow$ `STRASSEN`$(A[2,2], B[2,1] - B[1,1])$
    $M_5 \leftarrow$ `STRASSEN`$(A[1,1] + A[1,2], B[2,2])$
    $M_6 \leftarrow$ `STRASSEN`$(A[2,1] - A[1,1], B[1,1] + B[1,2])$
    $M_7 \leftarrow$ `STRASSEN`$(A[1,2] - A[2,2], B[2,1] + B[2,2])$

    $C \leftarrow$ `ZEROS`$(2,2)$
    $C[1,1] \leftarrow M_1 + M_4 - M_5 + M_7$
    $C[1,2] \leftarrow M_3 + M_5$
    $C[2,1] \leftarrow M_2 + M_4$
    $C[2,2] \leftarrow M_1 - M_2 + M_3 + M_6$

    return C
  end function

Dla prostoty zapisu algorytmu, wyniki pośrednie są zapisywane do macierzy pomocniczych $M_{1,\dots,7}$, zaś ostateczny wynik obliczenia do macierzy klatkowej $C_{2\times 2}$, jednak w rzeczywistej implementacji może istnieć możliwość optymalizacji tych kroków pod kątem inicjalizacji i wykorzystania pamięci.

1.3. **Metoda Deep Mind.** Algorytm ma postać analogiczną do metody Strassena. W każdej instancji rekurencji macierze wyjściowe dzielone są odpowiednio: A na 4×5, a B na 5×5 klatek. Wzory wyrażające wyniki pośrednie oraz klatki macierzy wynikowej przedstawione zostały na stronie projektu.

2. Implementacja

2.1. Metoda Bineta. Implementacja bezpośrednio odwzorowuje pseudokod. W kodzie występują wywołania funkcji pomocniczych o charakterze czysto technicznym.

```julia
function multiply(a::Matrix{<:Number}, b::Matrix{<:Number}, n_blocks::Union{Int, Symbol} = :auto)::Matrix{<:Number}
    n, m = size(a)
    m_b, p = size(b)

    if m ≠ m_b
        throw(ArgumentError("Matrix sizes don't match"))
    end

    if n_blocks === :auto
        n_blocks = max(n, m, p) ÷ 2
    end

    a, b = pad_to_common_square(n_blocks, a, b)

    m, _ = size(a)
    block_size = m ÷ n_blocks

    c = zeros(m, m)

    for i in 1:n_blocks
        for j in 1:n_blocks
            for k in 1:n_blocks
                multiply_blocks!(a, b, c, i, j, k, block_size)
            end
        end
    end

    return c[1:n, 1:p]
end
```

Figure 1. Funkcja realizująca metodę Bineta

```
function multiply_blocks!(
        a::Matrix{<:Number},
        b::Matrix{<:Number},
        c::Matrix{<:Number},
        block_i::Int,
        block_j::Int,
        block_k::Int,
        block_size::Int
)::Nothing
    for i in index_range(block_i, block_size)
        for j in index_range(block_j, block_size)
            for k in index_range(block_k, block_size)
                c[i, j] += a[i, k] * b[k, j]
            end
        end
    end
end
```

FIGURE 2. Funkcja odpowiedzialna za właściwą czynność mnożenia w metodzie Bineta

2.2. **Metoda Strassena.** Implementacja skupia się przede wszystkim na wzorach Strassena, pomijając całkowicie kwestie optymalizacji.

```
function multiply(a::Matrix{<:Number}, b::Matrix{<:Number}, threshold::Int = 2)::Matrix{<:Number}
    n, m = size(a)
    m_b, k = size(b)

    if m ≠ m_b
        throw(ArgumentError("Matrix sizes don't match"))
    end

    if min(n, m, k) ≤ threshold
        return a * b
    end

    a, b = pad_to_common_square(a, b)

    multiply_recursively(a, b)[1:n, 1:k]
end
```

FIGURE 3. Funkcja realizująca metodę Strassena

Funkcja pad_to_common_square sprowadza oba operandy do postaci macierzy kwadratowych o rozmiarach będących potęgami dwójki, przez dopisanie zer.

```julia
function multiply_recursively(a::MatrixLike, b::MatrixLike, threshold::Int = 2)::Matrix{<:Number}
    n, _ = size(a)

    if n ⩽ threshold
        return a * b
    end

    first, second = halves(a)

    @divide a
    @divide b

    p1 = multiply_recursively(a11 + a22, b11 + b22)
    p2 = multiply_recursively(a21 + a22, b11)
    p3 = multiply_recursively(a11, b12 - b22)
    p4 = multiply_recursively(a22, b21 - b11)
    p5 = multiply_recursively(a11 + a12, b22)
    p6 = multiply_recursively(a21 - a11, b11 + b12)
    p7 = multiply_recursively(a12 - a22, b21 + b22)

    c = zeros(n, n)

    c[first, first] += p1 + p4 - p5 + p7
    c[first, second] += p3 + p5
    c[second, first] += p2 + p4
    c[second, second] += p1 - p2 + p3 + p6

    return c
end
```

FIGURE 4. Pomocnicza, rekurencyjna funkcja w metodzie Strassena; implementuje właściwy algorytm

2.3. **Metoda Deep Mind.** Główną trudnością w implementacji jest duża liczba wzorów koniecznych do zapisania w kodzie. Na potrzeby wykonania zadania, formuły obecne na stronie projektu przekształcono z postaci graficznej do tekstu za pomocą narzędzia typu OCR. Następnie w programie zdefiniowano i zastosowano makro @declare_from_file, które umieszcza zawartość pliku tekstowego w kodzie źródłowym programu. Dla uproszczenia implementacji założono, że funkcja wykonuje mnożenie jedynie dla macierzy typu $A \in M_{4^k \times 5^k}$, $B \in M_{5^k \times 5^k}$, $k \in \mathbb{N}_+$.

```
function multiply(a::Matrix{<:Number}, b::Matrix{<:Number})::Matrix{<:Number}
    n, m = size(a)
    m_b, k = size(b)

    if m ≢ m_b
        throw(ArgumentError("Matrix sizes don't match"))
    end

    @assert is_power_of(n, 4) "n = $n"
    @assert is_power_of(m, 5) "m = $m"
    @assert is_power_of(k, 5) "k = $k"

    multiply_recursively(a, b)
end
```

FIGURE 5. Funkcja realizująca metodę Deep Mind

```
function multiply_recursively(a, b)
    n, m = size(a)

    if n ≤ 4 || m ≤ 5
        return a * b
    end

    n_chunk = n ÷ 4
    m_chunk = m ÷ 5

    @divide(a, 4, 5, n_chunk, m_chunk)
    @divide(b, 5, 5, m_chunk, m_chunk)

    @declare_from_file "meta/h.txt"
    @declare_from_file "meta/c.txt"

    c = zeros(n, m)

    @assign(c, 4, 5, n_chunk, m_chunk)

    return c
end
```

FIGURE 6. Właściwa implementacja metody Deep Mind

3. Koszt obliczeniowy

3.1. **Eksperyment.** By przekonać się jaka jest zależność czasu obliczeń oraz liczby poszczególnych rodzajów operacji zmiennoprzecinkowych od rozmiaru mnożonych macierzy, przeprowadzono testy dla następujących danych wejściowych: $A \in$

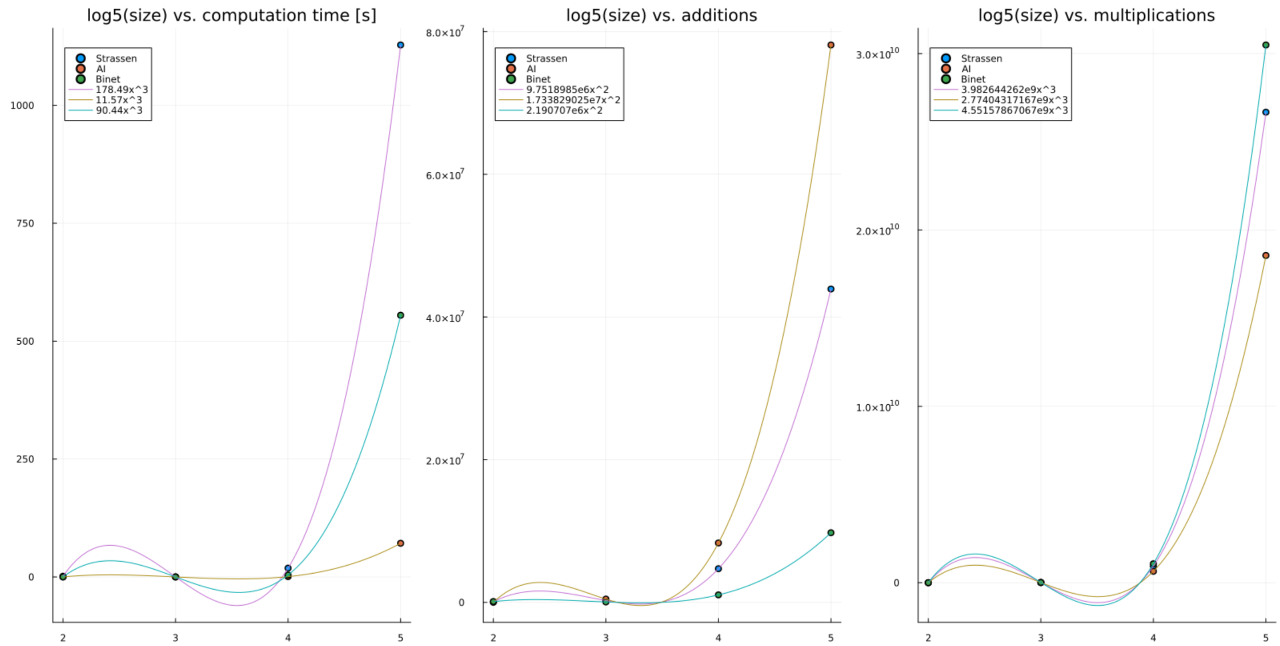

FIGURE 7. Czas działania i liczba operacji sumy i iloczynu w zależności od rozmiaru danych wejściowych, dla poszczególnych algorytmów wraz z aproksymacją wielomianową

$M_{4^k \times 5^k}(\mathbb{R}), B \in M_{5^k \times 5^k}(\mathbb{R}), k \in \{2, 3, 4, 5\}$. Z powodu ograniczonych możliwości sprzętowych, zakres potęg jest bardzo niewielki. Uzyskane wyniki obrazuje wykres na rysunku 7.

3.2. **Oszacowanie złożoności asymptotycznej.** Do zebranych danych zastosowano aproksymację wielomianową. Stopnie wielomianów aproksymujących wybrano zgodnie z teoretycznymi przewidywaniami. Dla czasu działania i liczby mnożeń wyniósł on 3, zaś dla liczby dodawań - 2.

Wnioski na temat wyniku oszacowania:

(1) **Jakość aproksymacji:** Ze względu na niewielką liczbę punktów, dopasowanie jest niskiej jakości, ze sporym overfittingiem. Właściwie uzyskano interpolację.
(2) **Współczynniki wielomianów:** Jako etykiety na wykresie przedstawiono współczynniki przy najwyższych potęgach, jako najbardziej znaczące dla oszacowania *asymtpotycznej* złożoności.

Biorąc pod uwagę powyższe uwagi, ogólna jakość oszacowania nie jest zbyt dobra, niemniej niesprzeczna z przewidywaniami teoretycznymi.