Mnożenie macierzy

Maciej Borowiec

26.10.2025

1 Algorytmy

1.1 Rekurencyjny algorytm Binét'a

1.1.1 Przebieg algorytmu

- Jeśli dowolny wymiar, dowolnej macierzy wejściowej, jest równy 1 - mnożymy macierze iteratywnie
- W przeciwnym wypadku:
  - Dzielimy macierze na podmacierze:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},\ B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

  - Obliczamy podmacierze macierzy wynikowej (mnożenie macierzy wykonujemy rekurencyjnie):

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

  - Zwracamy macierz wynikową:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

1.1.2 Kod

```python
def binet(A, B):
    flop_count = 0

    def f(A, B):
        nonlocal flop_count
        # initialize result matrix as zeros
        C = np.zeros((A.shape[0], B.shape[1]))
        # if matrices can be split further, split them
        if A.shape[0] > 1 and A.shape[1] > 1 and B.shape[1] > 1:
            # split matrices
            A11, A12, A21, A22 = split_matrix(A)
            B11, B12, B21, B22 = split_matrix(B)
            # calculate each part of the result recursively
            C11 = f(A11, B11) + f(A12, B21)
            C12 = f(A11, B12) + f(A12, B22)
            C21 = f(A21, B11) + f(A22, B21)
            C22 = f(A21, B12) + f(A22, B22)
            flop_count += el_count(C11) + el_count(C12) + el_count(C21) + el_count(C22)
            # write the result into new matrix
            half_r, half_c = C11.shape
            C[:half_r, :half_c] = C11
            C[:half_r, half_c:] = C12
            C[half_r:, :half_c] = C21
            C[half_r:, half_c:] = C22
        # if you can no longer split matrices, do normal matrix multiplication
        else:
            for r in range(C.shape[0]):
                for c in range(C.shape[1]):
                    for i in range(A.shape[1]):
                        C[r,c] += A[r,i]*B[i,c]
                        flop_count += 2
        # return resulting matrix
        return C

    return f(A, B), flop_count
```

Listing 1: Algorytm Binét'a

1.1.3 Oszacowanie złożoności obliczeniowej

Dla każdego wywołania funkcji wykonujemy 4 dodawania macierzy i 8 rekurencyjnych mnożeń macierzy o 2 razy mniejszych wymiarach. Zatem możemy zapisać złożoność obliczeniową jako:

$$T(n) = 8T(n/2) + 4n^2$$

$$T(1) = 1$$

Możemy zatem zapisać tę złożoność jako sumę:

$$T(n) = 1 \cdot 8^{\log_2 n} + \sum_{i=0}^{\log_2 n - 1} 8^i \cdot 4(\frac{n}{2^i})^2$$

Używając $a^{\log_c b} = b^{\log_c a}$ możemy uprościć równanie:

$$T(n) = n^{\log_2 8} + 4 \sum_{i=0}^{\log_2 n - 1} 8^i \cdot \frac{n^2}{4^i} = n^3 + 4n^2 \sum_{i=0}^{\log_2 n - 1} 2^i$$

Możemy teraz użyć wzoru na sumę ciągu geometrycznego:

$$T(n) = n^3 + 4n^2 \frac{1 - 2^{\log_2 n}}{1 - 2} = n^3 + 4n^2 \frac{1 - n}{-1} = 5n^3 - 4n^2$$

Zatem złożoność obliczeniowa wynosi $O(n^3)$.

1.2 Rekurencyjny algorytm Strassena

1.2.1 Przebieg algorytmu

- Dokonujemy paddingu - dodajemy zera do macierzy wejściowych, by miały one wymiar $2^n \times 2^n$
- Jeśli dowolny wymiar, dowolnej macierzy wejściowej, jest równy 1 - mnożymy macierze iteratywnie
- W przeciwnym wypadku:
  - Dzielimy macierze na podmacierze:
    $$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \; B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$
  - Obliczamy pomocnicze zmienne (mnożenie macierzy wykonujemy rekurencyjnie):
    $$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$
  - Obliczamy podmacierze macierzy wynikowej:
    $$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$
  - Zwracamy macierz wynikową:
    $$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

1.2.2 Kod

```python
def strassen(A, B):
    flop_count = 0
    # pad matrices
    n = A.shape[0]
    size_2n = 1
    while size_2n < n:
        size_2n *= 2
    A_padded = np.zeros((size_2n, size_2n))
    B_padded = np.zeros((size_2n, size_2n))
    A_padded[:n, :n] = A
    B_padded[:n, :n] = B

    def f(A, B):
        nonlocal flop_count
        n = A.shape[0]
        # initialize result matrix as zeros
        C = np.zeros((n,n))
        # if matrices can be split further, split them
        if n > 1:
            # split matrices
            A11, A12, A21, A22 = split_matrix(A)
            B11, B12, B21, B22 = split_matrix(B)
            # calculate needed values recursively
            M1 = f(A11+A22, B11+B22)
            M2 = f(A21+A22, B11)
            M3 = f(A11, B12-B22)
            M4 = f(A22, B21-B11)
            M5 = f(A11+A12, B22)
            M6 = f(A21-A11, B11+B12)
            M7 = f(A12-A22, B21+B22)
            flop_count += (n//2)**2 * 10
            # add values to create each part of result matrix
            C11 = M1 + M4 - M5 + M7
            C12 = M3 + M5
            C21 = M2 + M4
            C22 = M1 - M2 + M3 + M6
            flop_count += (n//2)**2 * 8
            # write the result into new matrix
            half_r, half_c = C11.shape
            C[:half_r, :half_c] = C11
            C[:half_r, half_c:] = C12
            C[half_r:, :half_c] = C21
            C[half_r:, half_c:] = C22
        # if you can no longer split matrices, it's 1x1 by 1x1
        else:
            C[0,0] = A[0,0]*B[0,0]
            flop_count += 1
        # return resulting matrix
        return C

    return f(A_padded, B_padded)[:n, :n], flop_count
```

Listing 2: Algorytm Strassena

1.2.3 Oszacowanie złożoności obliczeniowej

Dla każdego wywołania funkcji wykonujemy 18 dodawań/odejmowań macierzy i 7 rekurencyjnych mnożeń macierzy o 2 razy mniejszych wymiarach. Zatem możemy zapisać złożoność obliczeniową jako:

$$T(n) = 7T(n/2) + 18n^2$$

$$T(1) = 1$$

Możemy zatem zapisać tę złożoność jako sumę:

$$T(n) = 1 \cdot 7^{\log_2 n} + \sum_{i=0}^{\log_2 n-1} 7^i \cdot 18(\frac{n}{2^i})^2$$

Używając $a^{\log_c b} = b^{\log_c a}$ możemy uprościć równanie:

$$T(n) = n^{\log_2 7} + 18 \sum_{i=0}^{\log_2 n-1} 7^i \cdot \frac{n^2}{4^i} = n^{\log_2 7} + 18n^2 \sum_{i=0}^{\log_2 n-1} (\frac{7}{4})^i$$

Możemy teraz użyć wzoru na sumę ciągu geometrycznego:

$$T(n) = n^{\log_2 7} + 18n^2 \frac{1 - (\frac{7}{4})^{\log_2 n}}{1 - 2} = n^{\log_2 7} - 18n^2(1 - n^{\log_2 7/4})$$

$$T(n) = n^{\log_2 7} - 18n^2(1 - n^{\log_2 7/4}) = n^{\log_2 7} + 18n^{\log_2 7} - 18n^2 = 19n^{\log_2 7} - 18n^2$$

Zatem złożoność obliczeniowa wynosi $O(n^{\log_2 7}) \approx O(n^{2.8074})$.

1.3 Algorytm wykorzystujący faktoryzacje znalezione przez AI

1.3.1 Przebieg algorytmu

Algorytm dokonuje mnożenia rekurencyjnie za pomocą metody Binét'a, natomiast warunek stopu jest osiągany dla dowolnej faktoryzacji, którą znalazł Alpha Tensor. Wtedy wykorzystujemy dane na temat faktoryzacji, by obliczyć dane podmacierze, i co za tym idzie macierz wynikową.

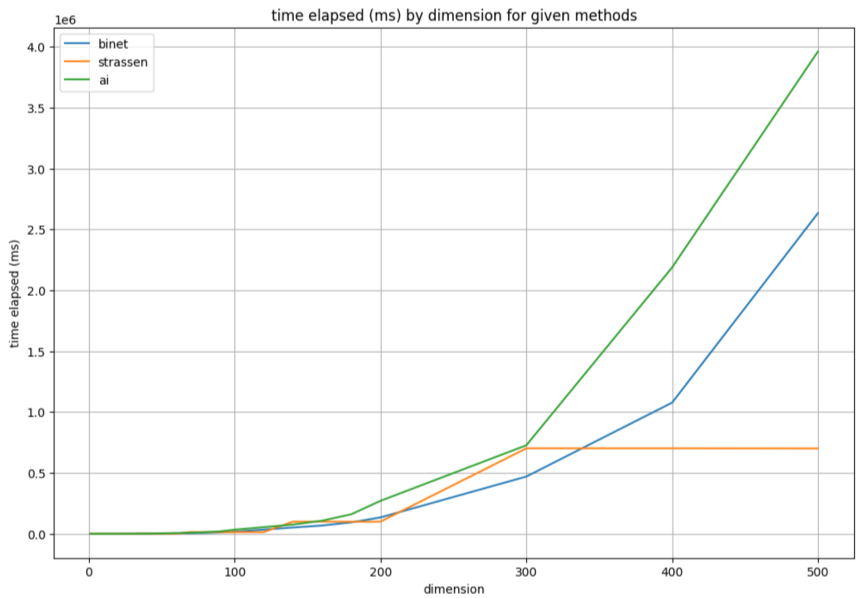
1.3.2 Kod

```python
def ai(A, B):
    flop_count = 0

    def f(A, B):
        nonlocal flop_count
        # initialize result matrix as zeros
        C = np.zeros((A.shape[0], B.shape[1]))
        shape_string = f"{A.shape[0]},{A.shape[1]},{B.shape[1]}"
        # if matrix sizes are in factorizations, multiply them
        if shape_string in FACTS:
            u, v, w = FACTS[shape_string]
            # create vectors representing matrices
            A_vec = A.reshape(-1)
            B_vec = B.reshape(-1)
            # calculate Ms
            A_part, f1 = mat_by_vec(u.T, A_vec)
            B_part, f2 = mat_by_vec(v.T, B_vec)
            M = A_part * B_part
            # create a vector of result and reshape it
            C_vec, f3 = mat_by_vec(w, M)
            flop_count += f1 + f2 + f3 + len(M)
            C = C_vec.reshape(A.shape[0], B.shape[1], order='F')
        # if they cannot be split further, multiply them normally
        elif A.shape[0] == 1 or A.shape[1] == 1 or B.shape[1] == 1:
            for r in range(C.shape[0]):
                for c in range(C.shape[1]):
                    for i in range(A.shape[1]):
                        C[r,c] += A[r,i]*B[i,c]
                        flop_count += 2
        # if not, split them using classic binet
        else:
            # split matrices
            A11, A12, A21, A22 = split_matrix(A)
            B11, B12, B21, B22 = split_matrix(B)
            # calculate each part of the result recursively
            C11 = f(A11, B11) + f(A12, B21)
            C12 = f(A11, B12) + f(A12, B22)
            C21 = f(A21, B11) + f(A22, B21)
            C22 = f(A21, B12) + f(A22, B22)
            flop_count += el_count(C11) + el_count(C12) + el_count(C21) + el_count(C22)
            # write the result into new matrix
            half_r, half_c = C11.shape
            C[:half_r, :half_c] = C11
            C[:half_r, half_c:] = C12
            C[half_r:, :half_c] = C21
            C[half_r:, half_c:] = C22
        # return resulting matrix
        return C

    return f(A, B), flop_count
```

Listing 3: Algorytm AI

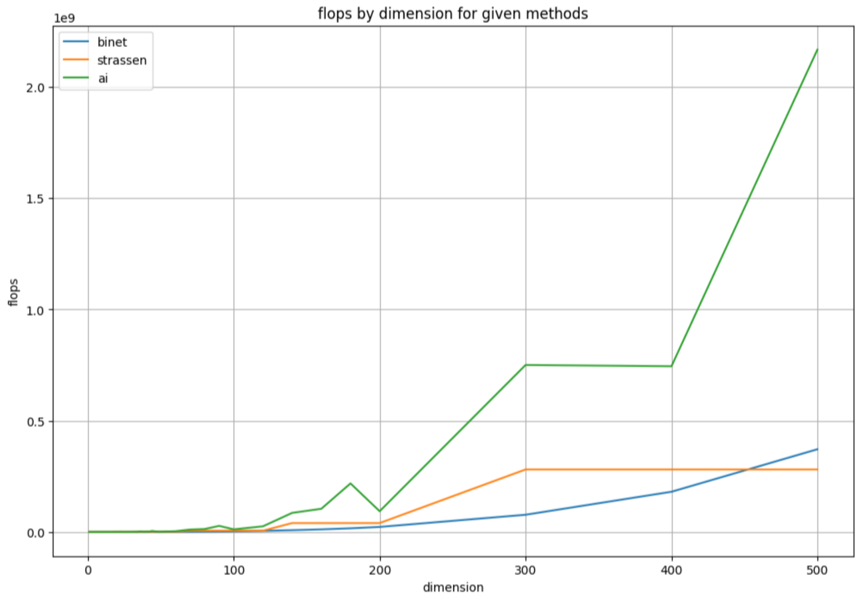
2 Porównanie algorytmów

Poniżej znajdują się 3 wykresy pokazujące działanie algorytmów w zależności od wymiarów macierzy. Są to wykresy pokazujące czas wykonania, liczbę operacji zmiennoprzecinkowych oraz zużytą pamięć dla każdego z 3 algorytmów. Algorytmy zostały uruchomione dla wszystkich wymiarów od 1 do 50, co dziesiątego wymiaru od 60 do 100, co dwudziestego wymiaru od 120 do 200 oraz wymiarów 300, 400 i 500.



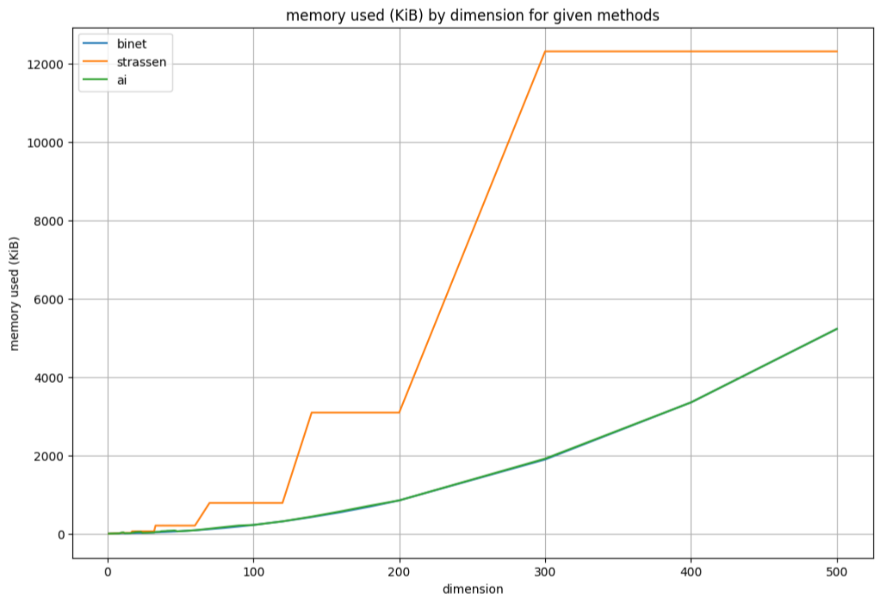
Rysunek 1: Czas wykonania algorytmów w zależności od rozmiaru macierzy

Czas wykonania dla największych macierzy okazał się być najlepszy dla algorytmu Strassena - jednakże widać, że on "skacze", ponieważ paduje macierze. Implementacja algorytmu Binét'a zaskakująco okazała się szybsza od algorytmu, który tak samo dzieli macierze, ale wykorzystuje lepsze faktoryzacje. Najprawdopodobniej jest to spowodowane nieoptymalnym mnożeniem rzadkiej macierzy przez wektor.



Rysunek 2: Liczba operacji zmiennoprzecinkowych algorytmów w zależności od rozmiaru macierzy

Liczba operacji zmiennoprzecinkowych dla algorytmu AI jest największa co nie jest zaskakujące patrząc na to, że był najwolniejszy. Ciekawe jest to, że algorytm Strassena dla wszystkich macierzy oprócz macierzy 500×500 wykonywał więcej operacji zmiennoprzecinkowych niż algorytm Binét'a, a dla tej największej różnica jest znacznie mniejsza od różnicy w czasie. Jest tak zapewne przez to, że algorytm Strassena dokonuje znacznie więcej dodawań niż mnożeń w porównaniu do algorytmu Binét'a, a dodawania są lżejsze dla sprzętu od mnożenia.



Rysunek 3: Zużycie pamięci algorytmów w zależności od rozmiaru macierzy

Dla algorytmów AI i Binét'a nie ma zauważalnej różnicy w użyciu pamięci, co nie jest zaskakujące. Oczekiwanym wynikiem również jest dużo większy przydział pamięci przez algorytm Strassena, gdyż dokonuje on paddingu macierzy.