# ARYTMETYKA MACIERZY HIERARCHICZNYCH

JAN SMÓŁKA

Abstract. Niniejszy dokument zawiera sprawozdanie z wykonania ćwiczenia 5. w ramach przedmiotu Algorytmy Macierzowe w semestrze zimowym roku akademickiego 2023/24.

## 1. Algorytmy

W ramach zadania zaimplementowano dwa algorytmy - dodawania i mnożenia H-macierzy. Ze względu na analogiczne implementacje, omówienie działania dodawania jest w poniższych rozdziałach omówione bardziej szczegółowo niż mnożenie.

### 1.1. **Dodawanie.**

Algorytm polega na wykonaniu działania blokowo. Rozmiary bloków są określone dla konkretnej pary H-macierzy przez rozmiary ich liści. Pseudokod nie zawiera wyszczególnionych wariantów liści, są one uwzględnione dopiero w implementacji.

**Algorithm 1** Suma H-macierzy

**Require:** $A, B \in HM_{n\times n}(\mathbb{R}) \wedge n \in \mathbb{N}_+$
**Ensure:** $A, B$ - H-Matrix nodes
  **function** ADD($A, B$)
    **if** $A.is\_divided$ **and** $B.is\_divided$ **then**
      **return** $Node$ { $add(A.ul, B.ul), add(A.ur, B.ur), add(A.ll, B.ll), add(A.lr, B.lr)$
  }
    **end if**
    **return** $Node$ { $A.matrix + B.matrix$ }
  **end function**

Dla prostoty w pseudokodzie wykorzystano notację przypominającą konstruktor obiektu Node. Gdy bieżąco dodawane węzły są podzielone, nastepuje wywołanie rekurencyjne dla poszczególnych dzieci, oznaczonych jako *ll* - *upper-left* itp. W przeciwnym wypadku składnikiem sumy jest co najmniej jeden liść. Dodajemy wówczas macierze w reprezentacji gęstej i konstruujemy z nich nowy węzeł, potencjalnie podzielony lub skompresowany za pomocą SVD. Należy mieć na uwadze wszelkie techniczne szczegóły procedury dodawania takich węzłów - np. gdy jeden składnik jest zerowy, wynikiem sumy jest drugi składnik, zaś gdy jeden ze składników nie jest liściem, musi zostać zrekonstruowany do postaci gęstej.

```
function add(a::HMatrix{T}, b::HMatrix{T})::HMatrix{T} where {T <: Number}
    @assert size(a) == size(b) "Mismatched dimensions - $(size(a)) + $(size(b))"

    rank = min(a.rank, b.rank)
    tolerance = max(a.tolerance, b.tolerance)

    sum = add(a.root, b.root, rank, tolerance)
    hmatrix_unchecked(sum, rank, tolerance)
end
```

FIGURE 1. Dodawanie - funkcja główna

1.2. **Mnożenie.** Metoda analogiczna, jednak w miejscu wywołania rekurencyjnego w algorytmie dodawania stosujemy schemat mnożenia Bineta, wykorzystując procedurę dodawania i rekurencyjne wywołanie mnożenia. Realizacja jest przedstawiona jako kod źródłowy.

## 2. IMPLEMENTACJA

Przedstawione w pseudokodzie algorytmy wraz z narzędziami służącymi do prezentacji wyników zaimplementowano w języku Julia, z wykorzystaniem implementacji struktury H-macierzy z zadania 3. Kod źródłowy niektórych funkcji pomocniczych dla algorytmu mnożenia nie został przytoczony ze względu na działanie całkowicie analogiczne do przedstawionych funkcji dla dodawania, o podobnych nazwach. Pominięte zostały również wszystkie funkcje pomocnicze, których charakter jest czysto techniczny - np. konstruktory obiektów.

## 3. WYNIKI DZIAŁANIA

3.1. **Testowanie algorytmów.** Algorytmy przetestowano na rzadkich macierzach losowych, z rozkładu jednorodnego $[0, 1]$, rozmiarów $256 \times 256$, z 1% niezerowych wartości.. Sprawdzono jedynie ich poprawność, z pominięciem pomiaru czasu działania. Kod testujący jest zawarty na ilustracji 7.

3.2. **Analiza wyników.**

(1) Suma - widać zachowanie cech topologicznych macierzy - pokrywające się duże "okienka" składników pozostają na swoich miejscach w macierzy sumy. Takie zachowanie wynika ze związku między dekompozycjami SVD składników i sumy - suma dekompozycji może być dobrym przybliżeniem dekompozycji sumy
(2) Iloczyn - zmiany w kompresji są znacznie bardziej gwałtowne, ze względu na możliwy zakres wartości iloczynu macierzy i nietrywialny związek rozkładu SVD czynników z dekompozycją iloczynu

```
function add(a::Node{T}, b::Node{T}, rank::Int, tolerance::T)::Node{T} where {T <: Number}
    if a.state == Zero
        deepcopy(b)

    elseif b.state == Zero
        deepcopy(a)

    elseif a.state == b.state == Divided
        children = Children{Node{T}}(
            add(a.children.ul, b.children.ul, rank, tolerance),
            add(a.children.ur, b.children.ur, rank, tolerance),
            add(a.children.ll, b.children.ll, rank, tolerance),
            add(a.children.lr, b.children.lr, rank, tolerance)
        )

        error = @sum(children, error)

        node_unchecked(
            Divided,
            a.rows,
            a.cols,
            nothing,
            children,
            error
        )

    elseif a.state ≠ Divided && b.state ≠ Divided
        add_nonzero_leaves(a, b, rank, tolerance)

    else
        add_nonzero_dense(a, b, rank, tolerance)
    end
end
```

FIGURE 2. Dodawanie - funkcja rekurencyjna - właściwa implementacja algorytmu

```
function add_nonzero_dense(a::Node{T}, b::Node{T}, rank::Int, tolerance::T)::Node{T} where {T <: Number}
    matrix = dense(a) + dense(b)
    one_to_n = 1:size(matrix, 1)

    node_from_slice(
        matrix,
        one_to_n,
        one_to_n,
        a.rows,
        a.cols,
        rank,
        tolerance
    )
end
```

FIGURE 3. Dodawanie - funkcja pomocnicza - dodawanie liści z konwersją do reprezentacji gęstej

```
function add_nonzero_leaves(a::Node{T}, b::Node{T}, rank::Int, tolerance::T)::Node{T} where {T <: Number}
    if a.state == b.state == Trivial
        matrix = a.svd + b.svd
        one_to_n = 1:size(matrix, 1)

        node_from_slice(
            matrix,
            one_to_n,
            one_to_n,
            a.rows,
            a.cols,
            rank,
            tolerance
        )

    elseif a.state == b.state == Compressed
        matrix = recompose(a.svd) + recompose(b.svd)
        one_to_n = 1:size(matrix, 1)

        node_from_slice(
            matrix,
            one_to_n,
            one_to_n,
            a.rows,
            a.cols,
            rank,
            tolerance
        )

    elseif a.state == Trivial && b.state == Compressed
        matrix = a.svd + recompose(b.svd)
        one_to_n = 1:size(matrix, 1)

        node_from_slice(
            matrix,
            one_to_n,
            one_to_n,
            a.rows,
            a.cols,
            rank,
            tolerance
        )

    elseif a.state == Compressed && b.state == Trivial
        matrix = recompose(a.svd) + b.svd
        one_to_n = 1:size(matrix, 1)

        node_from_slice(
            matrix,
            one_to_n,
            one_to_n,
            a.rows,
            a.cols,
            rank,
            tolerance
        )
    end
end
```

FIGURE 4. Dodawanie - funkcja pomocnicza - dodawanie liści, z rozróżnieniem przypadków

```
function multiply(a::HMatrix{T}, b::HMatrix{T})::HMatrix{T} where {T <: Number}
    @assert size(a, 2) == size(b, 1) "Mismatched dimensions - $(size(a)) * $(size(b))"

    rank = min(a.rank, b.rank)
    tolerance = max(a.tolerance, b.tolerance)

    product = multiply(a.root, b.root, rank, tolerance)
    hmatrix_unchecked(product, rank, tolerance)
end
```

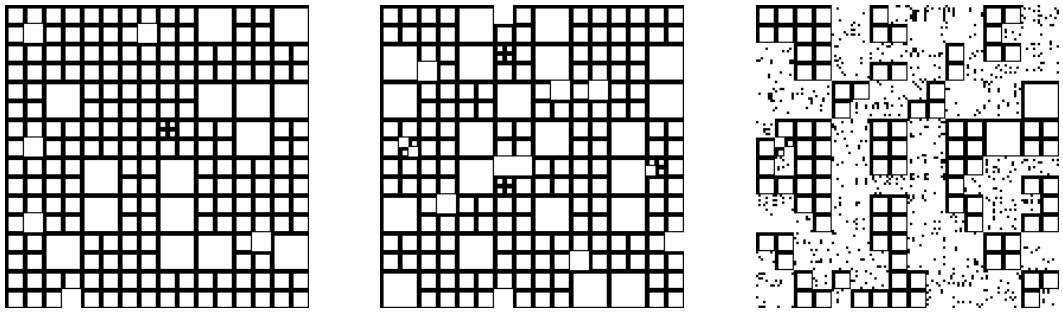
FIGURE 5. Mnożenie - funkcja główna

```
function multiply(a::Node{T}, b::Node{T}, rank::Int, tolerance::T)::Node{T} where {T <: Number}
    if a.state == Zero || b.state == Zero
        node_unchecked(
            Zero,
            a.rows,
            a.cols,
            nothing,
            nothing,
            zero(T)
        )

    elseif a.state == b.state == Divided
        # Recursive (Binet) block multiplication
        # Order of arguments does matter.
        # rows and cols of a result Node are always inherited from the first argument

        children = Children{Node{T}}(
            add( #                  ul
                multiply(a.children.ul, b.children.ul, rank, tolerance),
                multiply(a.children.ur, b.children.ll, rank, tolerance),
                rank,
                tolerance
            ),
            add( #                  ur
                multiply(a.children.ur, b.children.lr, rank, tolerance),
                multiply(a.children.ul, b.children.ur, rank, tolerance),
                rank,
                tolerance
            ),
            add( #                  ll
                multiply(a.children.ll, b.children.ul, rank, tolerance),
                multiply(a.children.lr, b.children.ll, rank, tolerance),
                rank,
                tolerance
            ),
            add( #                  lr
                multiply(a.children.lr, b.children.lr, rank, tolerance),
                multiply(a.children.ll, b.children.ur, rank, tolerance),
                rank,
                tolerance
            )
        )

        error = @sum(children, error)

        node_unchecked(
            Divided,
            a.rows,
            a.cols,
            nothing,
            children,
            error
        )

    elseif a.state ≠ Divided && b.state ≠ Divided
        multiply_nonzero_leaves(a, b, rank, tolerance)

    else
        multiply_nonzero_dense(a, b, rank, tolerance)
    end
end
```

FIGURE 6. Mnożenie - funkcja rekurencyjna - właściwa implementacja algorytmu; widoczne zastosowanie metody Bineta

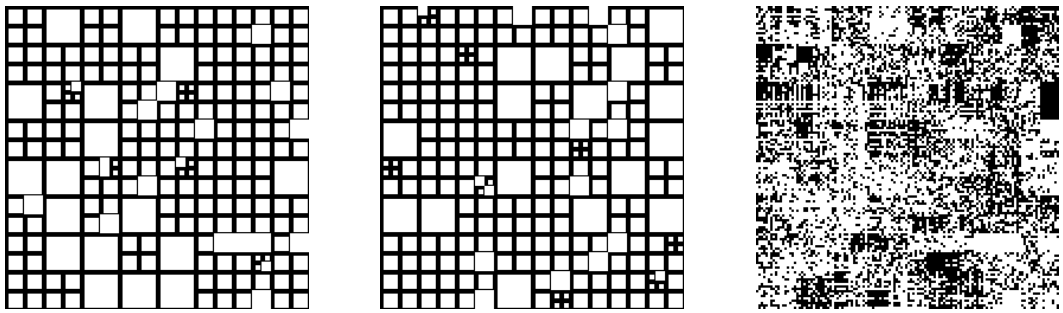
```
function fail_info(result::Matrix{Float64}, valid::Matrix{Float64})::String
    diff = result - valid
    error_min, error_max = extrema(diff)
    error_total = sum(diff)

    "error ∈ < $error_min, $error_max >, total = $error_total"
end

function test_add(size::Int)
    print("test_add($size): ")

    a = rand(size, size)
    b = rand(size, size)

    valid = a + b

    result = add(
        hmatrix(a, rank, threshold),
        hmatrix(b, rank, threshold)
    ) ▷ dense

    @assert all(valid .≈ result) fail_info(result, valid)
    println("passed!")
end

function test_multiply(size::Int)
    print("test_multiply($size): ")

    a = rand(size, size)
    b = rand(size, size)

    valid = a * b

    result = multiply(
        hmatrix(a, rank, threshold),
        hmatrix(b, rank, threshold)
    ) ▷ dense

    @assert all(valid .≈ result) fail_info(result, valid)
    println("passed!")
end
```

FIGURE 7. Funkcje sprawdzające poprawność implementacji



(A) Składnik (B) Składnik (C) Suma

FIGURE 8. Przykład sumy H-macierzy



(A) Czynnik (B) Czynnik (C) Iloczyn

FIGURE 9. Przykład iloczynu H-macierzy