

Algorytmy Macierzowe

Sprawozdanie nr 3

4.12.2024

Mateusz Król, Natalia Bratek

gr. 3

Spis treści

1. [Polecenie](#)
2. [Rekurencyjna kompresja macierzy](#)
 1. [Opis algorytmu](#)
 2. [Pseudokod](#)
 3. [Fragment kodu](#)
3. [Rekurencyjna dekompresja macierzy](#)
 1. [Opis algorytmu](#)
 2. [Pseudokod](#)
 3. [Fragment kodu](#)
4. [Wizualizacja](#)
5. [Wykresy i bitmapy](#)
 1. [Wartości osobliwe](#)
 2. [Nieskompresowana bitmapa](#)
 3. [Kompresja](#)
 1. [Wariant 1](#)
 2. [Wariant 2](#)
 3. [Wariant 3](#)
 4. [Wariant 4](#)
 5. [Wariant 5](#)
 6. [Wariant 6](#)
6. [Bibliografia](#)

1. Polecenie

- Proszę wybrać ulubiony język programowania.
- Proszę wybrać ulubioną kolorową bitmapę np. 500×500
- Proszę zamienić bitmapę na 3 macierze Red Green Blue (wartości z przedziału [0,255])
- Proszę napisać rekurencyjną kompresję macierzy z wykorzystaniem częściowego SVD dla wybranych parametrów
 $\delta = \text{najmniejsza wartość osobliwa}$ (wyrzucamy mniejsze) i $b = \text{maksymalny rank}$ (liczba wartości osobliwych)
- Proszę zaimplementować rysowacz skompresowanej macierzy
- Proszę zaimplementować rysowacz skompresowanej bitmapy

2. Rekurencyjna kompresja macierzy

2.1 Opis algorytmu

Algorytm polega na hierarchicznej kompresji macierzy z wykorzystaniem częściowego SVD. Dzieli macierz na coraz mniejsze bloki, aż każdy blok będzie można przybliżyć macierzą o niskim ranku i z wymaganą dokładnością. Najpierw, dla danego bloku macierzy, algorytm oblicza SVD przy użyciu metody randomized_svd. Jeśli blok spełnia warunek dopuszczalności, algorytm oznacza go jako liść w drzewie, zapisując wynikowe macierze U, S, i V. Jeśli nie, blok zostaje podzielony na cztery mniejsze części, które są następnie przetwarzane w ten sam sposób. Proces jest powtarzany rekurencyjnie, aż każdy blok spełni kryterium przybliżenia lub osiągnie minimalny rozmiar.

2.2 Pseudokod

```
compress(r, epsilon):

    Wyodrębnij bieżący blok macierzy
    matrix_block = matrix[t_min:t_max, s_min:s_max]

    if blok jest zerowy:
        oznacz blok jako liść
        ustaw rank na 0
        zwróć blok

    oblicz dekompozycję SVD
    (U, Sigma, V) = RandomizedSVD(matrix_block, r + 1)

    if blok spełnia warunek dopuszczalności
        zapisz U[:, :r], Sigma[:r], V[:r, :]
        oznacz blok jako liść
        zwróć blok
    else:

        Stwórz listę children
        Dodaj do children: CompressTree(matrix, t_min, new_t_max, s_min, new_s_max, r,
        epsilon)
        Dodaj do children: CompressTree(matrix, t_min, new_t_max, new_s_max, s_max, r,
        epsilon)
        Dodaj do children: CompressTree(matrix, new_t_max, t_max, s_min, new_s_max, r,
        epsilon)
        Dodaj do children: CompressTree(matrix, new_t_max, t_max, new_s_max, s_max, r,
        epsilon)

        Rekurencyjnie przetwórz dzieci
        dla child w children:
            wywołaj compress dla child z parametrami (r, epsilon)

    zwróć blok z children
```

2.3 Fragment kodu

```
def is_admissible(self, U, S, V, r, epsilon):
    if self.t_min + r == self.t_max or S[r] <= epsilon:
        self.rank = r
        self.set_leaf(U[:, :r], S[:r], V[:r, :])
        return True
    return False

def set_leaf(self, U, S, V):
    self.is_leaf = True
    self.U = U
    self.S = S
    self.V = V

def compress(self, r, epsilon):
    matrix_block = self.matrix[self.t_min:self.t_max, self.s_min:self.s_max]

    if np.sum(matrix_block) == 0:
        self.rank = 0
```

```

        self.is_leaf = True
        self.zeros = True
        return self

U, Sigma, V = randomized_svd(matrix_block, n_components=r + 1, random_state=0)
sigma = np.diag(Sigma)
if self.is_admissible( U,Sigma,V, r, epsilon):
    return self

else:
    self.children = []
    new_t_max = (self.t_min + self.t_max) // 2
    new_s_max = (self.s_min + self.s_max) // 2
    row_splits = [(self.t_min, new_t_max), (new_t_max, self.t_max)]
    col_splits = [(self.s_min, new_s_max), (new_s_max, self.s_max)]

    for t_min, t_max in row_splits:
        for s_min, s_max in col_splits:
            child = CompressTree(self.matrix, t_min, t_max, s_min, s_max)
            self.children.append(child)

    for child in self.children:
        child.compress(r, epsilon)

```

3. Rekurencyjna dekompresja macierzy

3.1 Opis algorytmu

Decompress rekonstruuje oryginalną macierz z jej skompresowanej reprezentacji w drzewie. Jeśli węzeł drzewa jest liściem, funkcja sprawdza czy reprezentuje blok zerowy. Jeśli tak, zwraca blok wypełniony zerami, a w przeciwnym razie odtwarza blok macierzy z zapisanych macierzy U, S i V przy użyciu operacji macierzowych. Dla węzłów wewnętrznych, które mają dzieci, funkcja inicjalizuje pustą macierz o odpowiednich wymiarach, a następnie wypełnia jej części poprzez wywołanie funkcji decompress rekurencyjnie dla każdego dziecka. Każde dziecko odpowiada jednemu z czterech podbloków macierzy, które są umieszczane w odpowiednich miejscach macierzy wynikowej.

3.2 Pseudokod

```

decompress():

    if węzeł jest liściem:
        if blok jest zerowy:
            return macierz zerowa o wymiarach (t_max - t_min, s_max - s_min)
        else:
            odtwórz macierz
            macierz = U * diag(S) * V
            zwróć macierz o wymiarach (t_max - t_min, s_max - s_min)

    zinicjalizuj macierz decompressed_matrix o wymiarach (t_max - t_min, s_max - s_min)
    oblicz połowę zakresów:
        half_row = (t_max + t_min) // 2
        half_col = (s_max + s_min) // 2

    dla każdego dziecka (child) w children:

        if indeks dziecka == 0:
            Wypełnij górny lewy kwadrant:
            decompressed_matrix[:half_row - t_min, :half_col - s_min] = child.DEOMPRESS()

```

```

if indeks dziecka == 1:
    Wypełnij górny prawy kwadrant:
    decompressed_matrix[:half_row - t_min, half_col - s_min:] = child.DECOMPRESS()

if indeks dziecka == 2:
    Wypełnij dolny lewy kwadrant:
    decompressed_matrix[half_row - t_min:, :half_col - s_min] =
child.DECOMPRESS()

if indeks dziecka == 3:
    Wypełnij dolny prawy kwadrant:
    decompressed_matrix[half_row - t_min:, half_col - s_min:] =
child.DECOMPRESS()

zwróć decompressed_matrix

```

3.3 Fragment kodu

```

def decompress(self):
    if self.is_leaf:
        if self.zeros:
            return np.zeros((self.t_max - self.t_min, self.s_max - self.s_min))
        return (self.U @ np.diag(self.S) @ self.V).reshape(self.t_max - self.t_min,
self.s_max - self.s_min)

    nrows = self.t_max - self.t_min
    ncols = self.s_max - self.s_min
    decompressed_matrix = np.zeros((nrows, ncols))

    half_row = (self.t_max + self.t_min) // 2
    half_col = (self.s_max + self.s_min) // 2
    for i, child in enumerate(self.children):
        if i == 0:
            decompressed_matrix[:half_row - self.t_min, :half_col - self.s_min] =
child.decompress()
        elif i == 1:
            decompressed_matrix[:half_row - self.t_min, half_col - self.s_min:] =
child.decompress()
        elif i == 2:
            decompressed_matrix[half_row - self.t_min:, :half_col - self.s_min] =
child.decompress()
        elif i == 3:
            decompressed_matrix[half_row - self.t_min:, half_col - self.s_min:] =
child.decompress()

    return decompressed_matrix

```

4. Wizualizacja

4.1 Rysowacz skompresowanej macierzy

```

class CompressTreeStructureVisualizer:
    def __init__(self, compress_tree):

```

```

        self.compress_tree = compress_tree

    def visualize_tree_structure(self):
        rows, cols = self.compress_tree.matrix.shape
        structure = np.ones((rows, cols))

    def mark_blocks(node):
        if node.is_leaf:
            if not node.zeros:
                structure[node.t_min:node.t_min + node.rank,
                          node.s_min:node.s_max] = 0

                structure[node.t_min :node.t_max,
                          node.s_min:node.s_min + node.rank] = 0
        else:
            for child in node.children:
                mark_blocks(child)

    mark_blocks(self.compress_tree)

    plt.figure(figsize=(10, 10))
    plt.imshow(structure, cmap="gray", interpolation="nearest")
    plt.title("Tree Structure Visualization (U, S, V blocks)")
    plt.axis("off")
    plt.show()

```

4.2 Rysowacz skompresowanej bitmapy

```

def reconstruct_block(node):
    if node.is_leaf:
        if node.zeros:
            return np.zeros((node.t_max - node.t_min, node.s_max - node.s_min))
        else:
            return node.U @ np.diag(node.S) @ node.V
    else:
        raise ValueError("Node is not a leaf.")

class CompressTreeBitmapVisualizer:
    def __init__(self, compress_tree):
        self.compress_tree = compress_tree

    def reconstruct_matrix(self):
        matrix = np.zeros_like(self.compress_tree.matrix)
        stack = [self.compress_tree]

        while stack:
            node = stack.pop()
            if node.is_leaf:
                block = reconstruct_block(node)
                matrix[node.t_min:node.t_max, node.s_min:node.s_max] = block
            else:
                stack.extend(node.children)

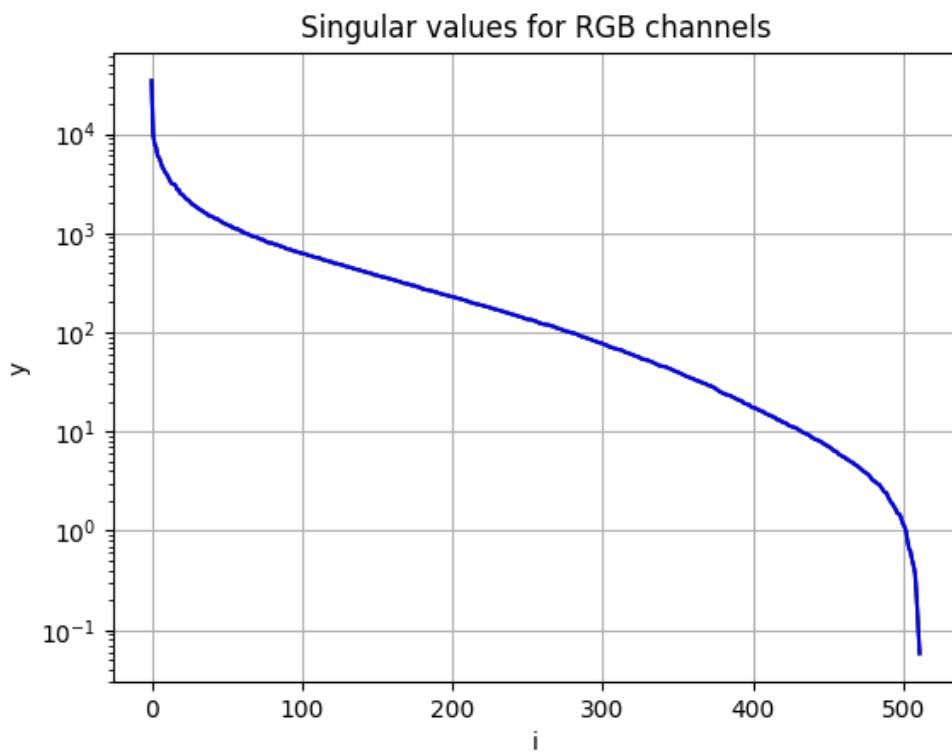
        return matrix

    def draw_bitmap(self):
        matrix = self.reconstruct_matrix()
        return matrix

```

5. Wykresy i bitmapy

5.1 Wartości osobliwe



5.2 Nieskompresowana bitmapa



Wymiary: 512x512



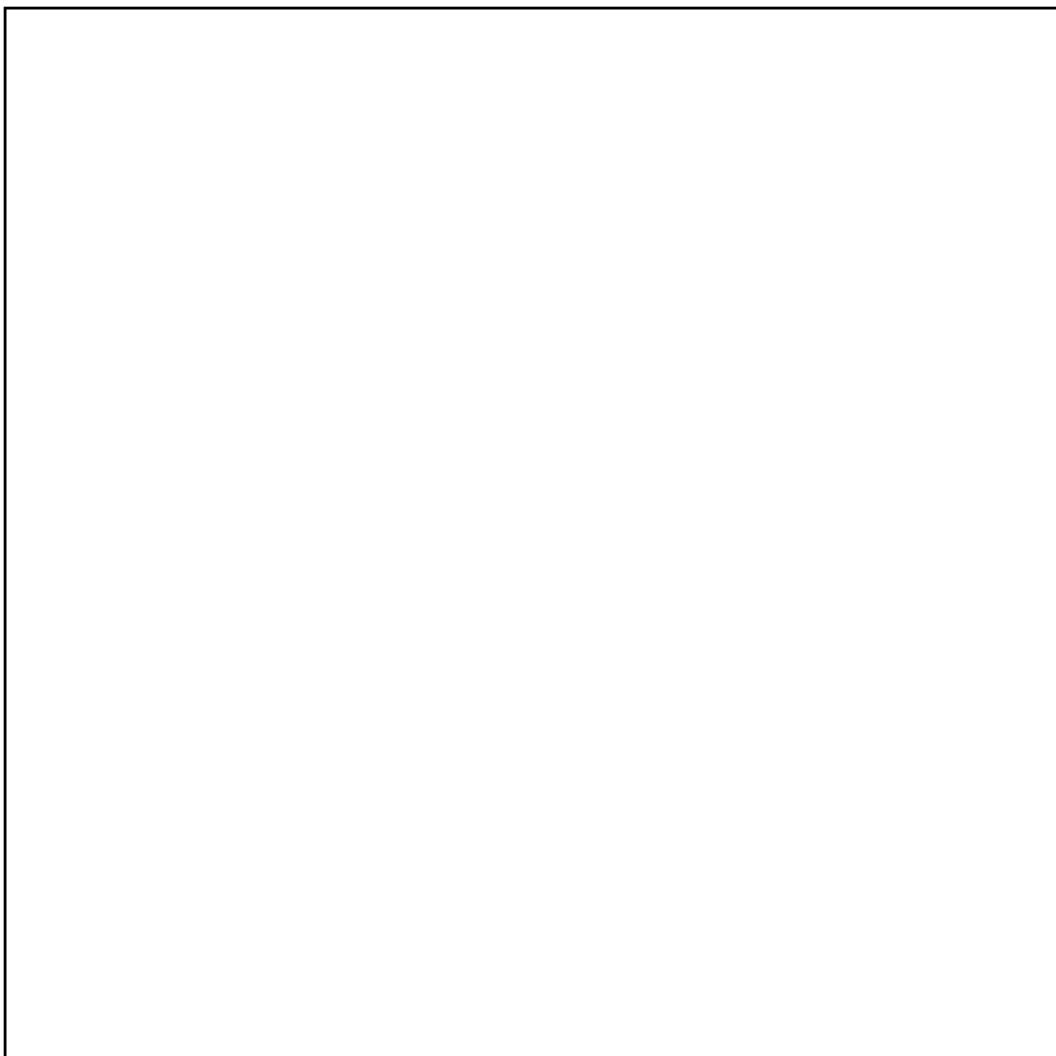
5.3 Kompresja

5.3.1 Wariant 1

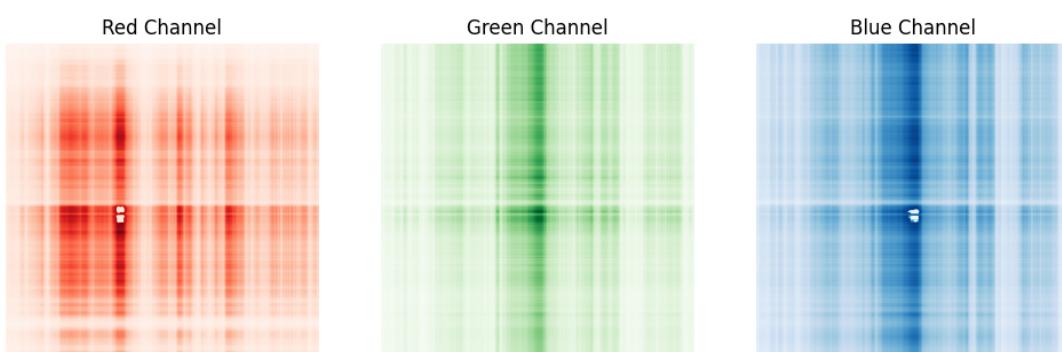
- $r = 1$
- $\text{epsilon} = \sigma_1$

Macierze kompresji są identyczne dla R, G, B:

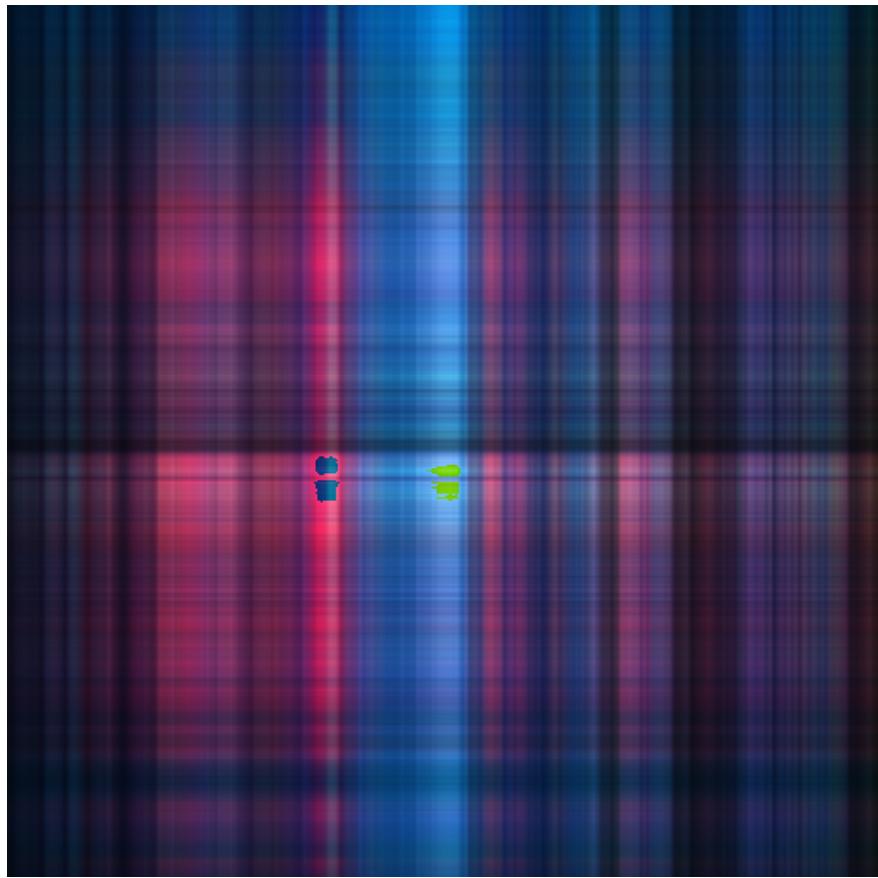
Tree Structure Visualization (U, S, V blocks)



Wynikowe bitmapy R, G, B:



Połączona bitmapa:



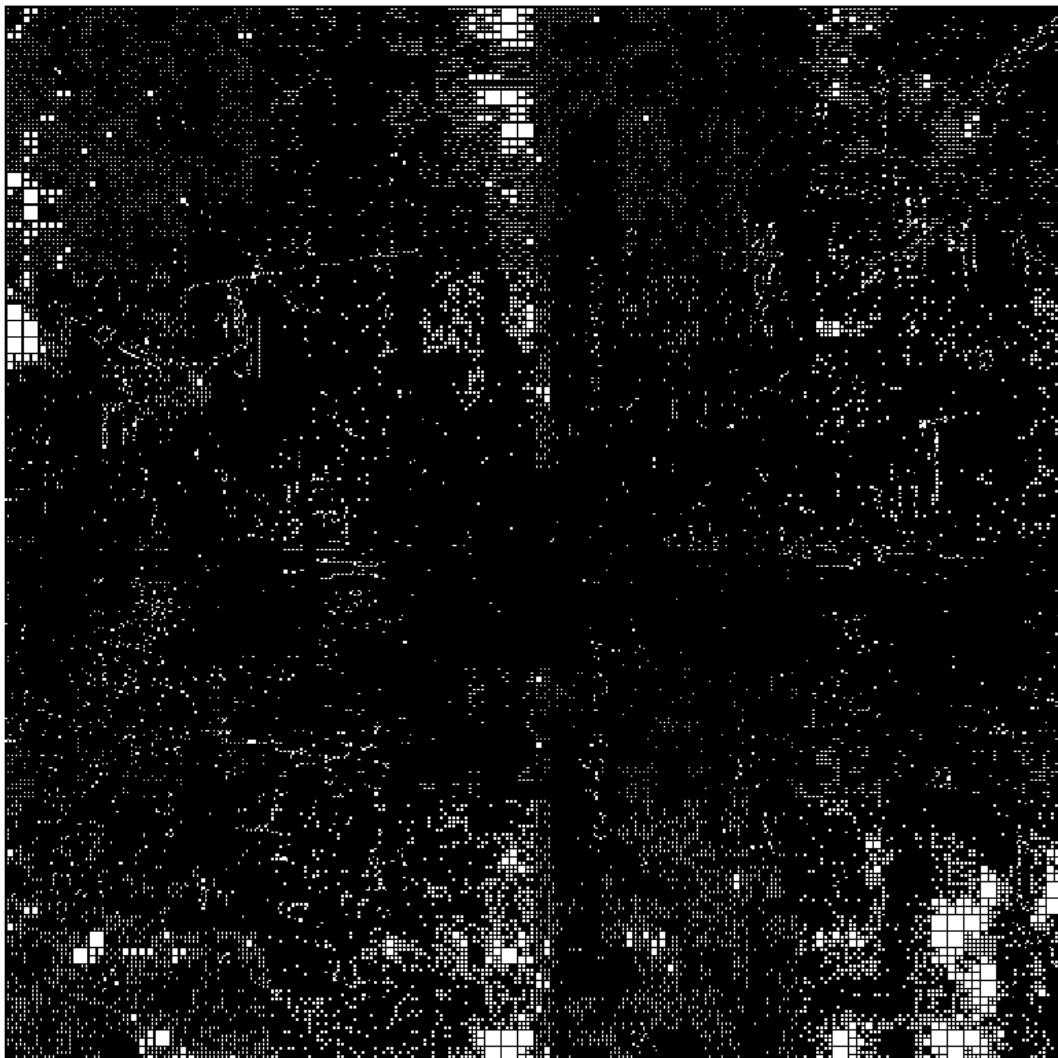
5.3.2 Variant 2

- $r = 1$
- $\epsilon = \sigma_{512}$

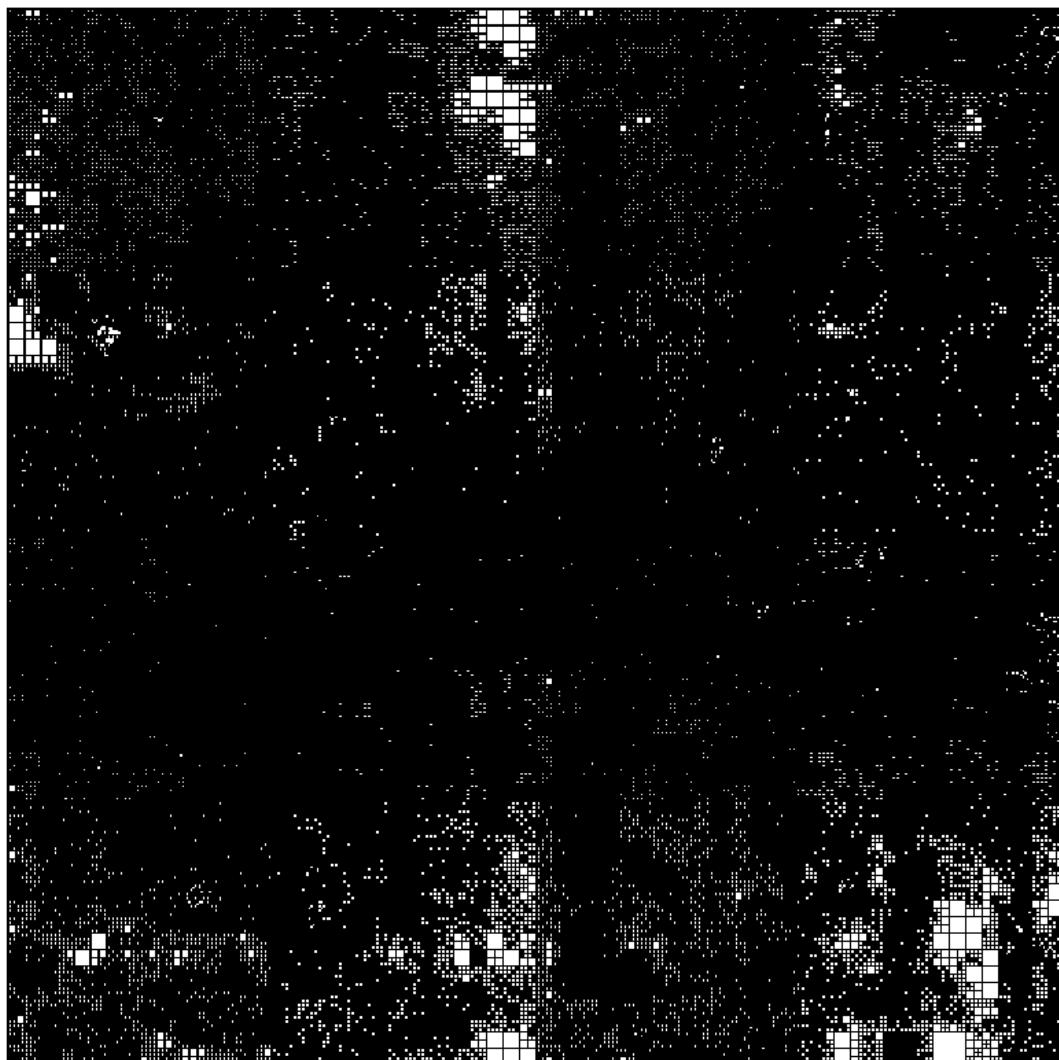
Macierze kompresji dla R, G, B:



Tree Structure Visualization (U, S, V blocks)



Tree Structure Visualization (U, S, V blocks)



Wynikowe bitmapy R, G, B:



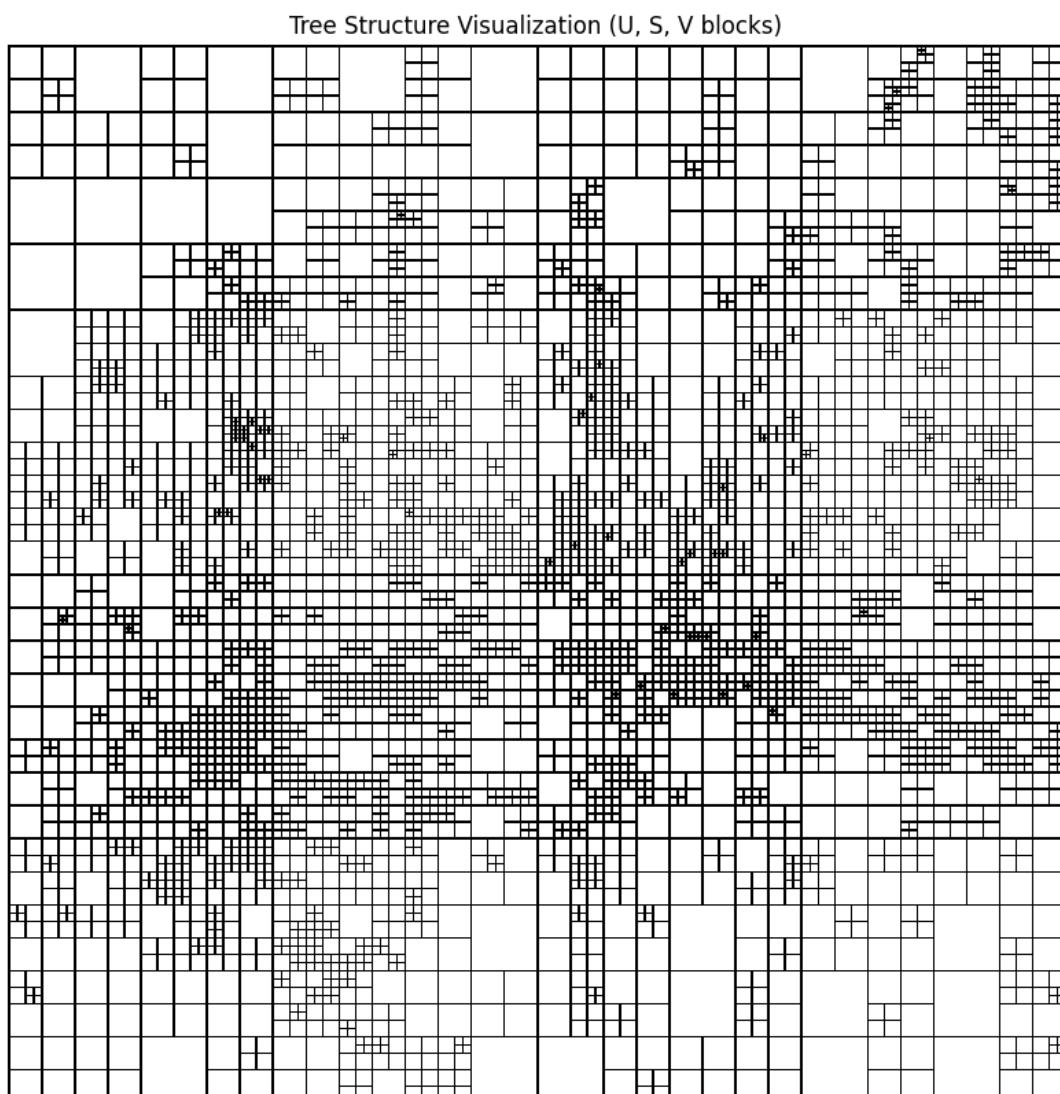
Połączona bitmapa:



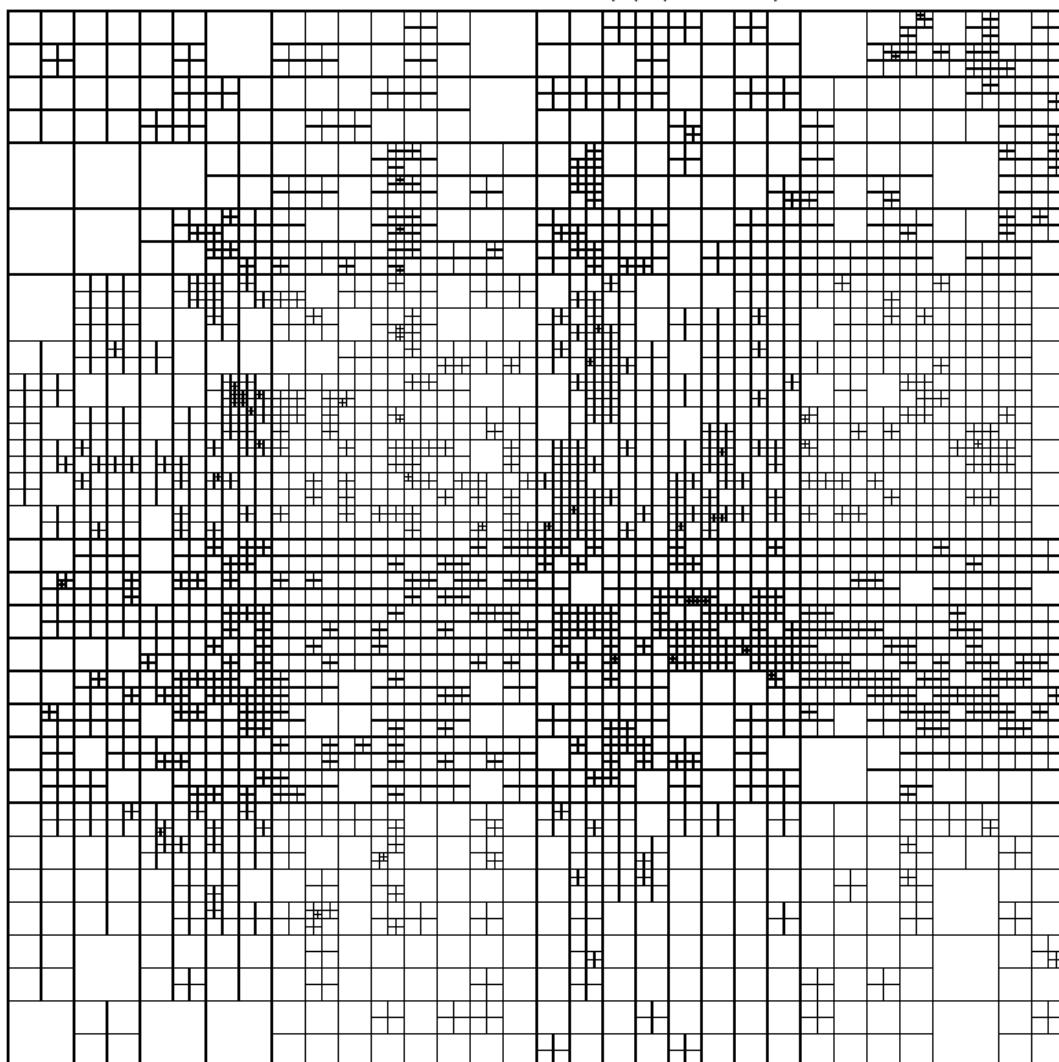
5.3.3 Variant 3

- $r = 1$
- $\text{epsilon} = \sigma_{256}$

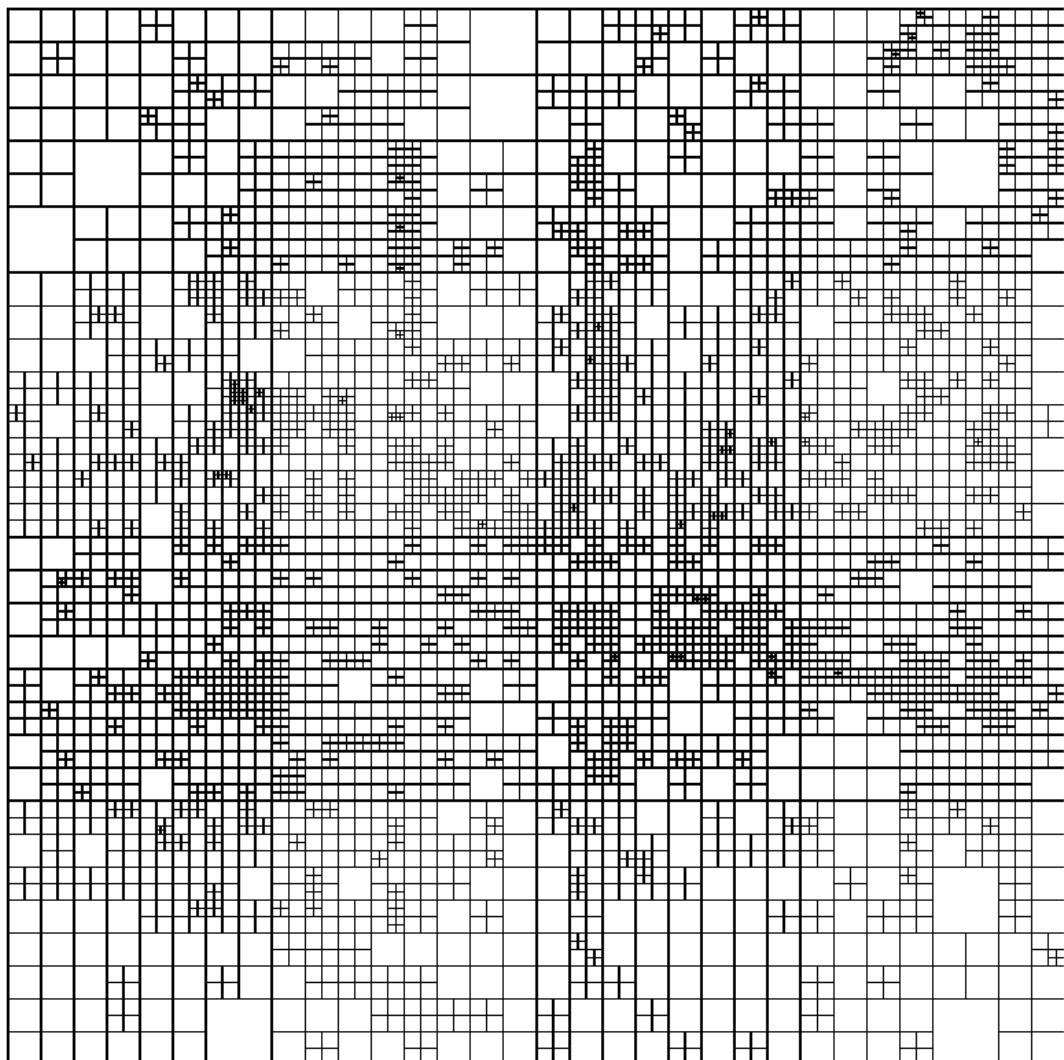
Macierze kompresji dla R, G, B:



Tree Structure Visualization (U, S, V blocks)



Tree Structure Visualization (U, S, V blocks)



Wynikowe bitmapy R, G, B:



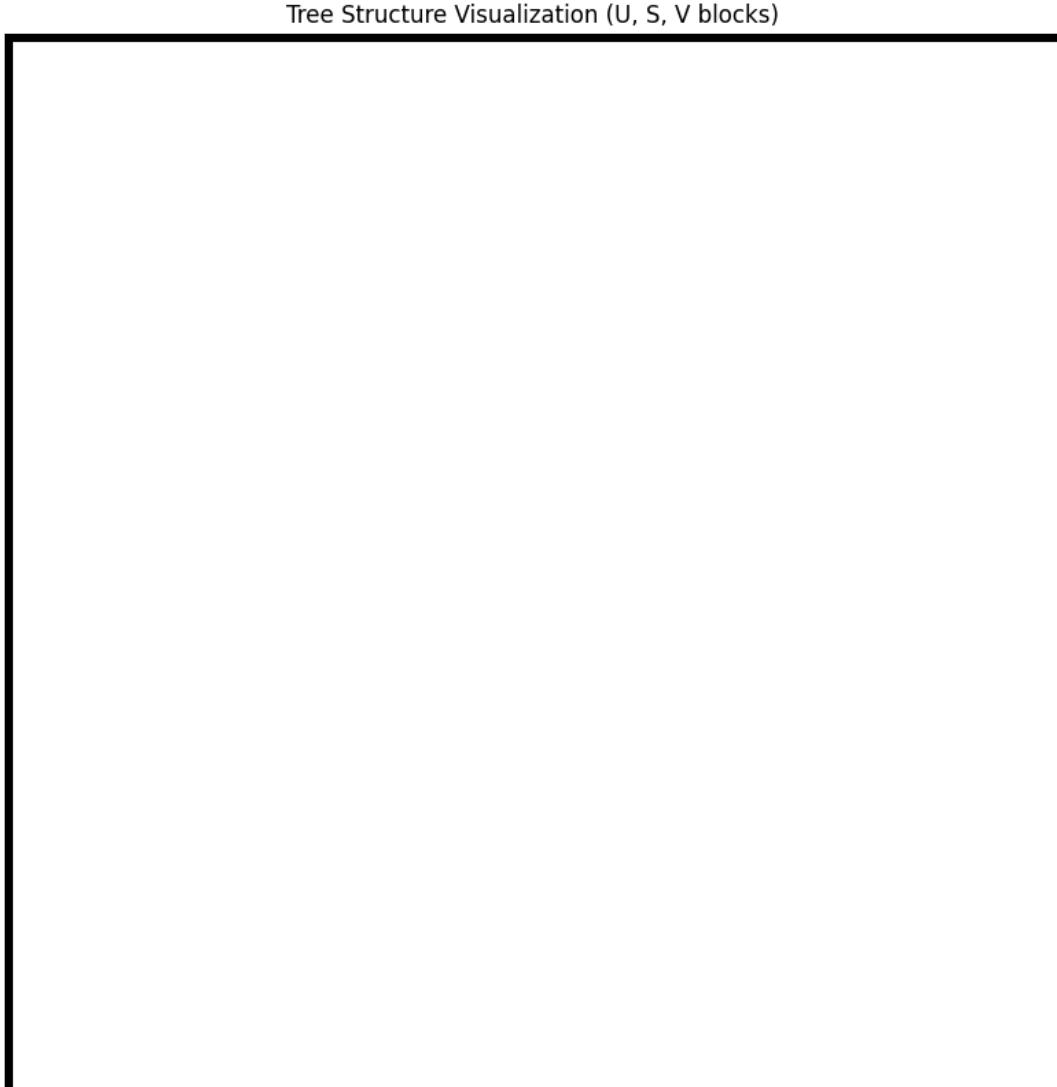
Połączona bitmapa:



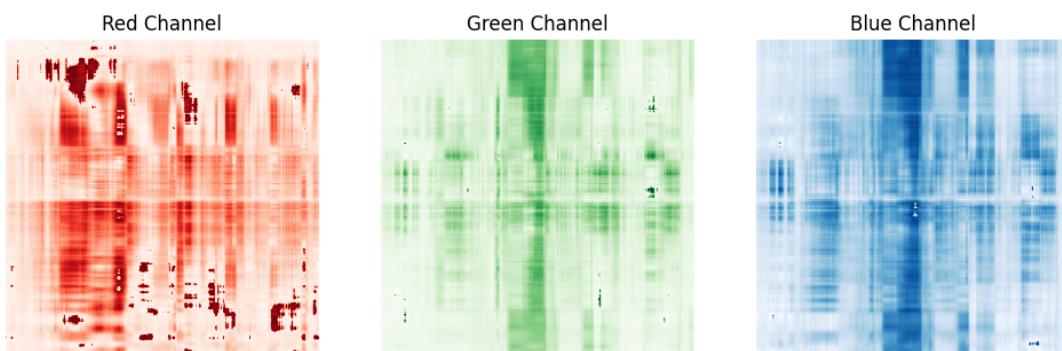
5.3.4 Wariant 4

- $r = 4$
- $\text{epsilon} = \sigma_1$

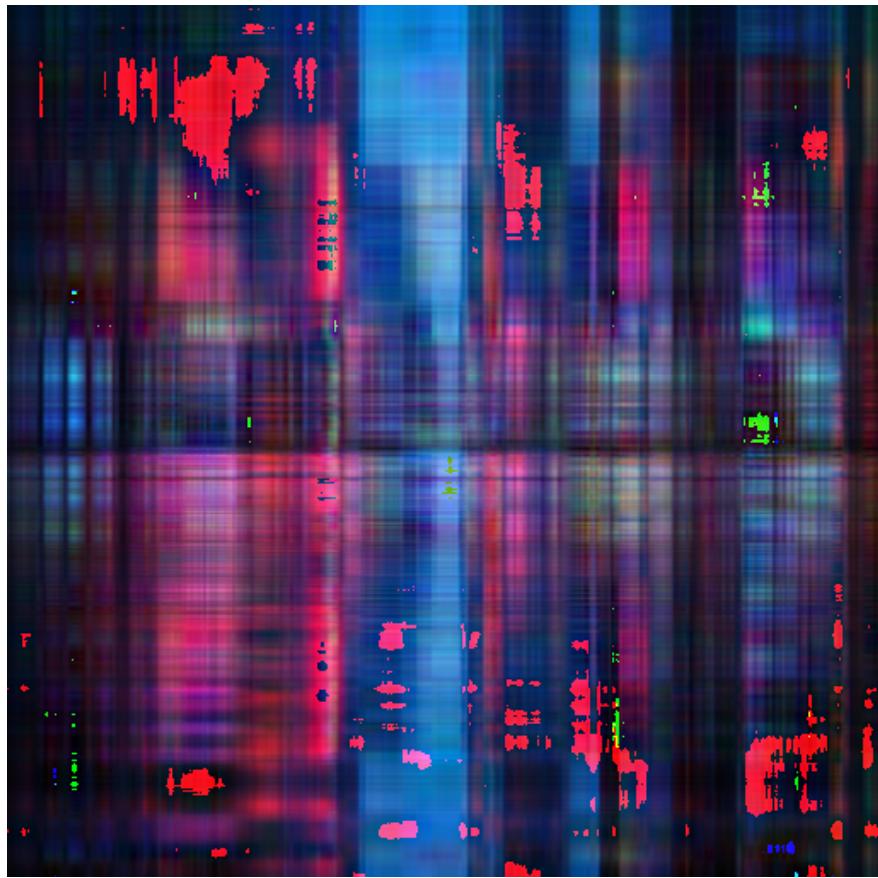
Macierze kompresji są identyczne dla R, G, B:



Wynikowe bitmapy R, G, B:



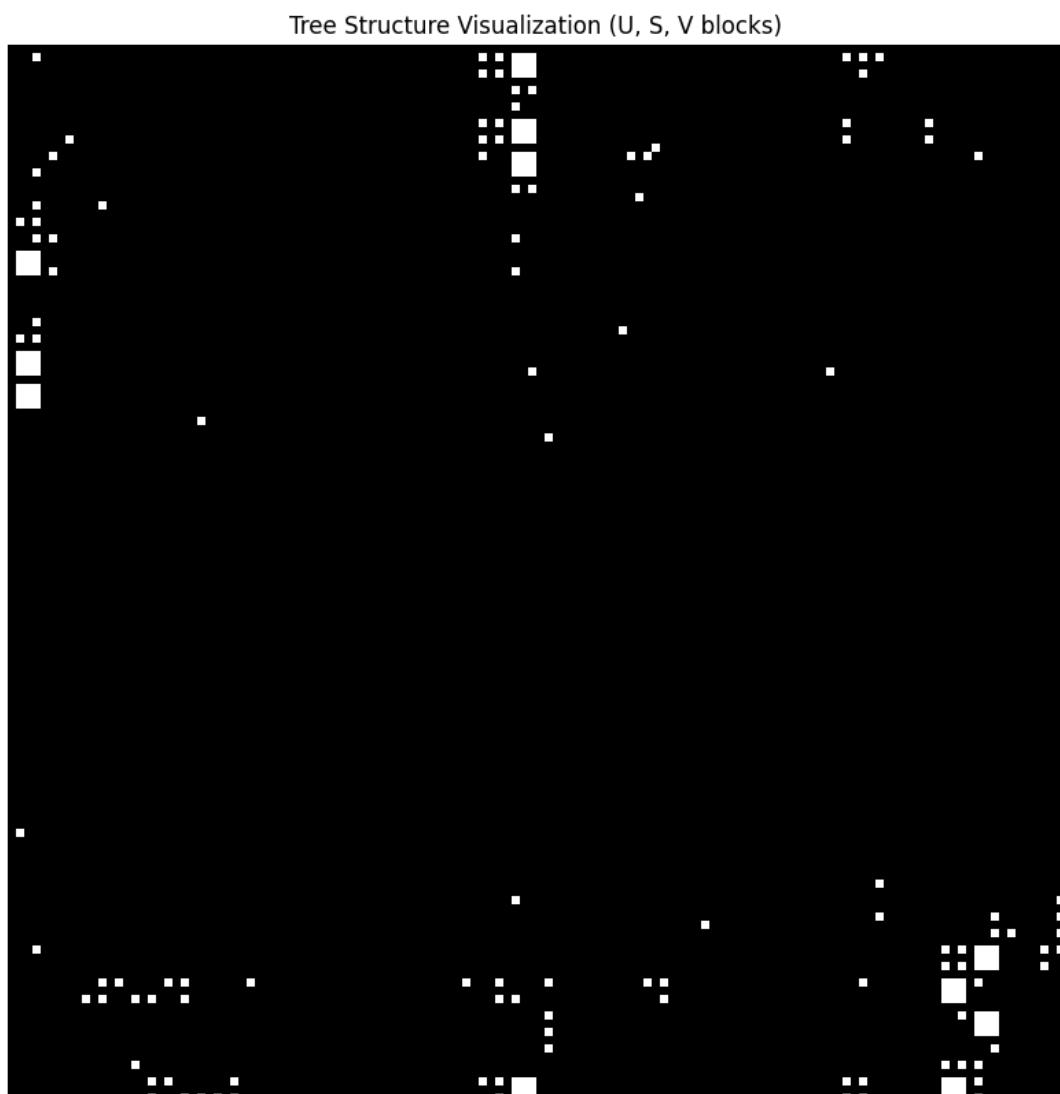
Połączona bitmapa:



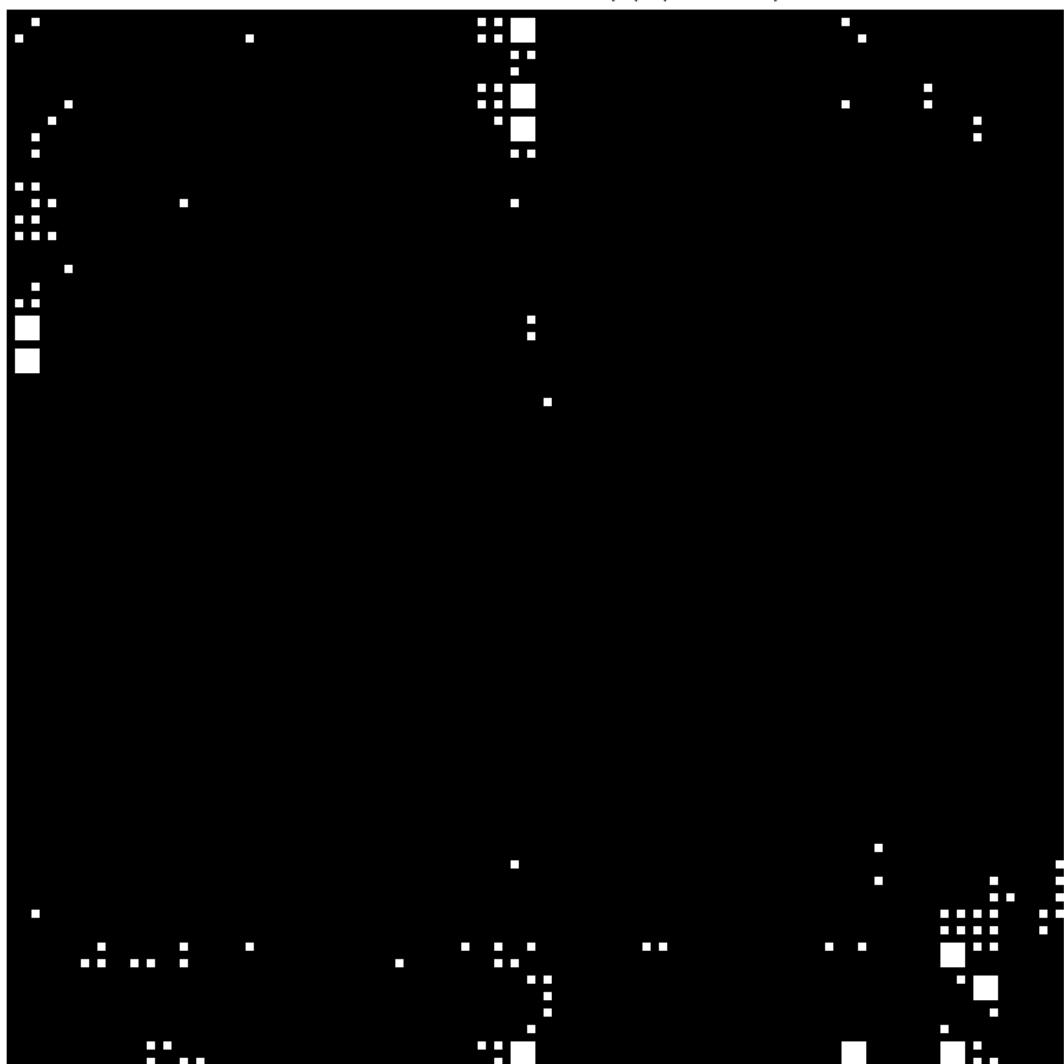
5.3.5 Variant 5

- $r = 4$
- $\epsilon = \sigma_{512}$

Macierze kompresji dla R, G, B:



Tree Structure Visualization (U, S, V blocks)



Tree Structure Visualization (U, S, V blocks)



Wynikowe bitmapy R, G, B:



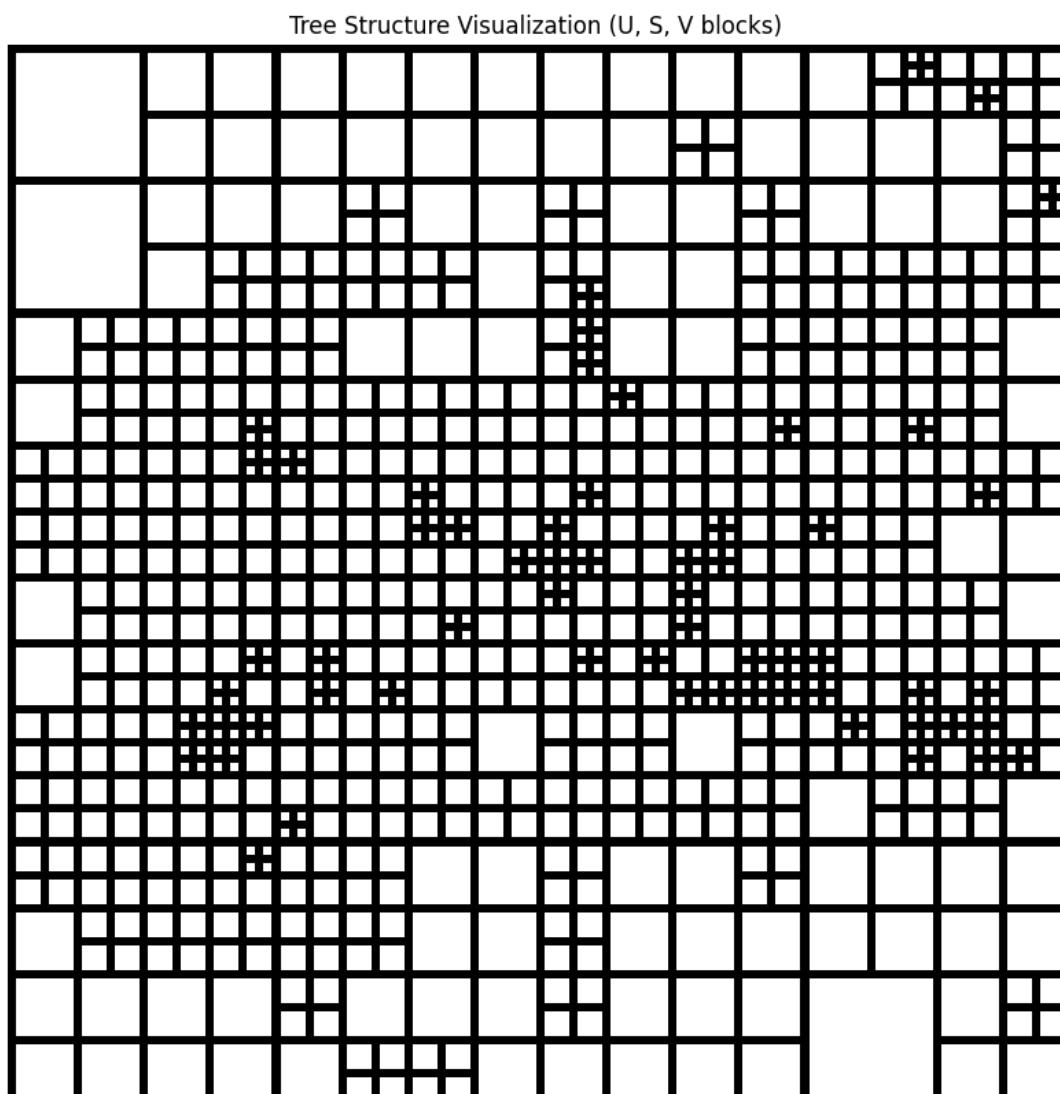
Połączona bitmapa:



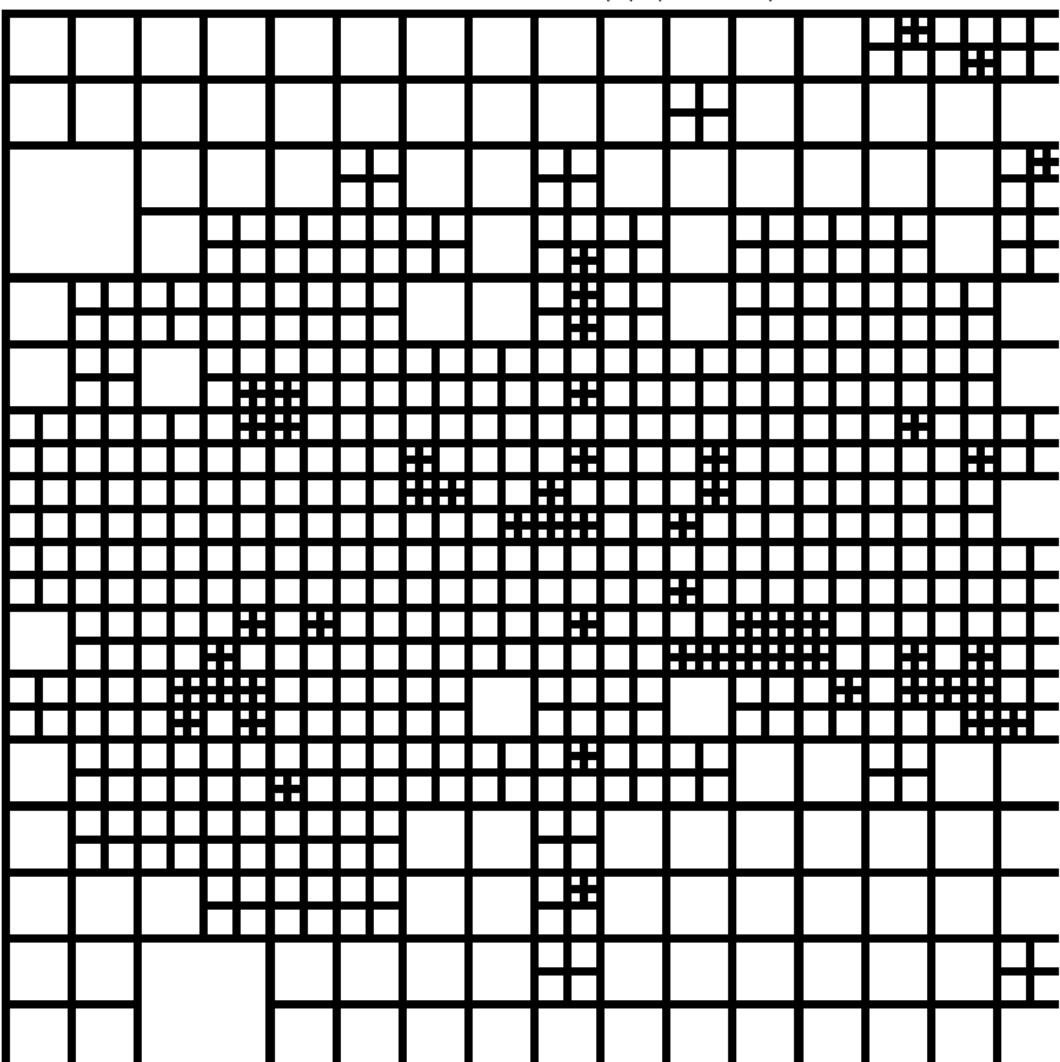
5.3.6 Variant 6

- $r = 4$
- $\text{epsilon} = \sigma_{256}$

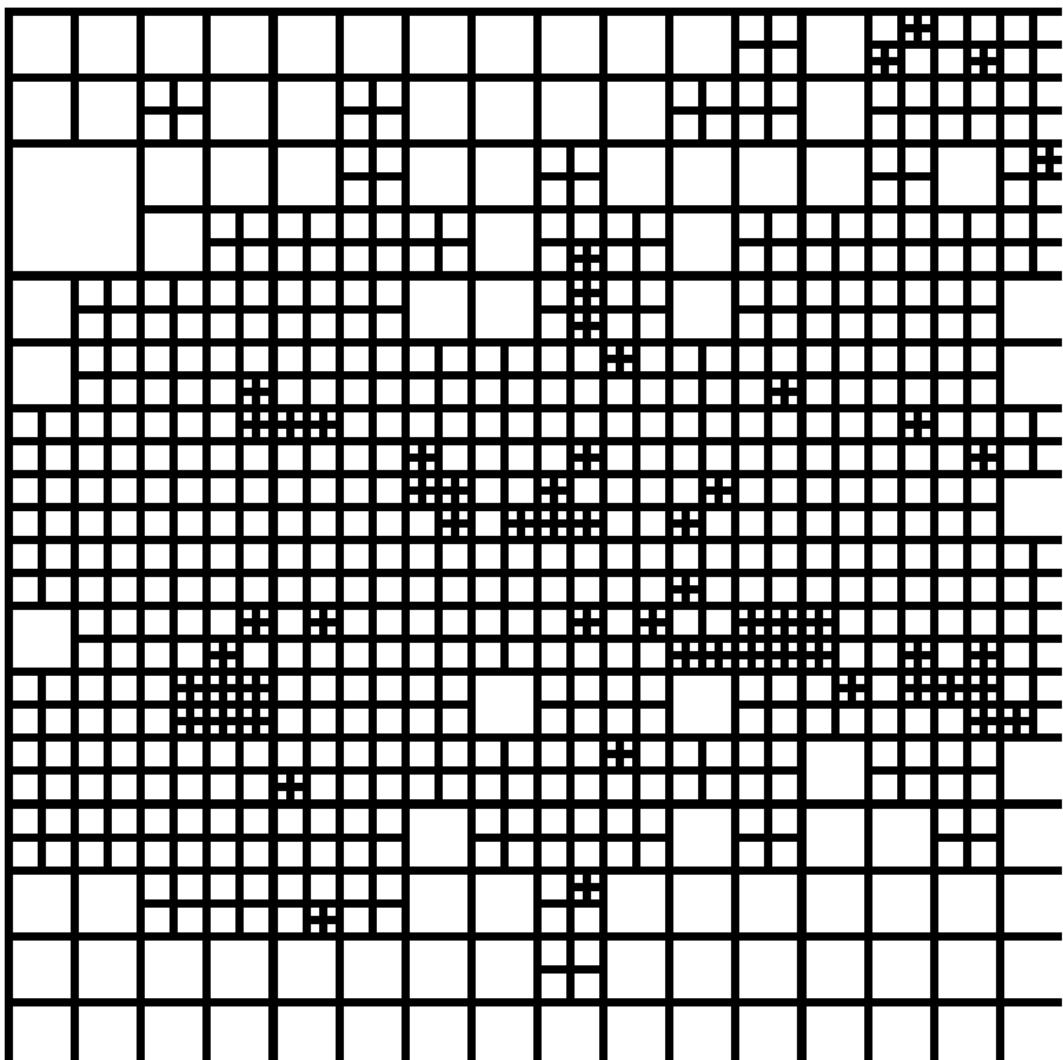
Macierze kompresji dla R, G, B:



Tree Structure Visualization (U, S, V blocks)



Tree Structure Visualization (U, S, V blocks)



Wynikowe bitmapy R, G, B:



Połączona bitmapa:



6. Bibliografia

- Wykłady prof. dr hab. Macieja Paszyńskiego (<https://home.agh.edu.pl/~paszynsk/RM/RachunekMacierzowy1.pdf>)