Labolatorium 4

Autorzy:

Patryk Klatka
Wojciech Łoboda

Algorytm permutacji macierzy - Minimum degree

Algorytm minimum degree służy do znajdowania takiej permutacji macierzy która minimalizuje 'fill-in' - ilość zerowych elementów macierzy które w trakcje wykonywania pewnych algorytmów zmieniają wartość na niezerową. Jedym z zastosowań permutacji minimum degree jest wykorzystanie jej przed wykonaniem rozkładu Choleskiego, co usprawnia działanie tego algorytmu.

Pseudokod:

Dla symetrycznej macierzy M o rozmiarze $n \times n$:

- $G_0 = (V, E)$ ($|V| = n$, jeżeli i-tym wierszy i j-tej kolumnie jest wartość niezerowa to między wierchołkami i i j jest krawędź).
- R = []
- Dla i = 1 do i = n:
  - v - węzeł z G_{i-1} o najmiejszym stopniu.
  - $G_i = G_{i-1}$ - ale bez wierchołka v
  - R.append(v)
- R - wynikowa permutacja wierchołków.

Implementacja

```
from math import inf

def minimum_degree_permutation(matrix):
    n, m = matrix.shape
    adj_matrix = {i:set() for i in range(n)}
    for i in range(n):
        for j in range(m):
            if matrix[i][j] != 0 and i != j:
                adj_matrix[i].add(j)
    permutation = []
    for i in range(n):
        deg_min = m+1
        for v, adj in adj_matrix.items():
            if len(adj) < deg_min:
                v_min = v
                deg_min = len(adj)
```

```
        for v in adj_matrix:
            adj_matrix[v] = adj_matrix[v].difference([v_min])
        for u in adj_matrix[v_min]:
            adj_matrix[u] = (adj_matrix[u].union(adj_matrix[v_min].difference([u]))
        adj_matrix.pop(v_min)
        permutation.append(v_min)
    return permutation

def permutate(matrix, permutation):
    new_matrix = matrix.copy()
    for i in range(len(permutation)):
        if i == permutation[i]:
            continue
        new_matrix[i,:] = matrix[permutation[i],:].copy()
    matrix = new_matrix.copy()
    for i in range(len(permutation)):
         if i == permutation[i]:
             continue
         new_matrix[:,i] = matrix[:,permutation[i]].copy()
    return new_matrix

def apply_permutation(matrix, algorithm):
    p = algorithm(matrix)
    return permutate(matrix, p)
```

Algorytm permutacji macierzy - Cuthill-McKee i reversed Cuthill-McKee

Algorytm Cuthill-McKee to kolejna metoda znajdowania permutacji macierzy które minimalizują 'fill-in'. Algorytm przekształca rzadką symetryczną macierz do do macierzy wstęgowej, algorytm jest wariantem BFS-a. Reversed Cuthill-McKee jest tym samym algorytmem z odwrócaną kolejnością wierchołków w znalezionej permutacji.

Pseudokod:

Dla symetrycznej macierzy M o rozmiarze $n \times n$:

- $G = (V, E)$ ($|V| = n$, jeżeli i-tym wierszy i j-tej kolumnie jest wartość niezerowa to między wierchołkami i i j jest krawędź).
- R = [x] - x to wierchołek o najmniejszym stopniu z G
- Dla i = 1, 2, ..., dopóki $|R| < n$:
    - $A_i = Adj(R_i) \setminus R$
    - $sort(A_i)$ - rosnąco po minimalnym przodku (odwiedzonym sąsiadem z najmniejszym indeksem w R oraz po stopniu jezeli minimalni przodkowie są tacy sami).
    - R.append(A_i)
- R - wynikowa permutacja wierchołków.

Implementacja

```
from queue import Queue
```

```python
def cuthill_mckee(matrix):
    n, m = matrix.shape
    ADJ = {i:set() for i in range(n)}
    for i in range(n):
        for j in range(m):
            if matrix[i][j] != 0 and i != j:
                ADJ[i].add(j)
    deg_min = inf
    for v, adj in ADJ.items():
        if len(adj) < deg_min:
            v_min = v
            deg_min = len(adj)
    R = []
    pred = [inf for _ in range(n)]
    visited = [False for _ in range(n)]
    pred[v_min] = 0
    Q = Queue()
    Q.put((0, len(ADJ[v_min]), v_min))
    while not Q.empty():
        _, _, v = Q.get()
        if visited[v]:
            continue
        visited[v] = True
        p = len(R)
        R.append(v)
        for u in ADJ[v]:
            if visited[u] == False:
                if pred[u] > p:
                    pred[u] = p
                Q.put((pred[u], len(ADJ[u]), u))
    return R

def reversed_cuthill_mckee(matrix):
    return cuthill_mckee(matrix)[::-1]
```

Kompresja macierzy do macierzy hierarchicznej (poprzedni lab)

```python
from sklearn.utils.extmath import randomized_svd
import numpy as np
import numpy as np
import matplotlib.pyplot as plt
from collections import deque


class MatrixTree:
    def __init__(self, matrix, row_min, row_max, col_min, col_max):
        self.matrix = matrix
        self.row_min = row_min
        self.row_max = row_max
        self.col_min = col_min
        self.col_max = col_max
        self.leaf = False
        self.children = None


    def compress(self, r, eps):
```

```python
        U, Sigma, V = randomized_svd(self.matrix[self.row_min:self.row_max, self.cc
        if self.row_min + r == self.row_max or Sigma[r] <= eps:
            self.leaf = True
            if not self.matrix[self.row_min:self.row_max, self.col_min: self.col_ma
                self.rank = 0
            else:
                self.rank = len(Sigma)
                self.u = U
                self.s = Sigma
                self.v = V
        else:
            self.children = []
            row_newmax = (self.row_min + self.row_max)//2
            col_newmax = (self.col_min + self.col_max)//2
            self.children.append(MatrixTree(self.matrix, self.row_min, row_newmax,
            self.children.append(MatrixTree(self.matrix, self.row_min, row_newmax,
            self.children.append(MatrixTree(self.matrix, row_newmax, self.row_max,
            self.children.append(MatrixTree(self.matrix, row_newmax, self.row_max,

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

    def decompress(self, output_matrix):
        if self.leaf:
            if self.rank != 0:
                sigma = np.zeros((self.rank,self.rank))
                np.fill_diagonal(sigma, self.s)
                output_matrix[self.row_min:self.row_max, self.col_min: self.col_max
            else:
                output_matrix[self.row_min:self.row_max, self.col_min: self.col_max
        else:
            for child in self.children:
                child.decompress(output_matrix)

    def compute_compression_ratio_(self):
        if self.leaf:
            x = self.row_max - self.row_min
            y = self.col_max - self.col_min
            if self.rank != 0:
                return self.rank * len(self.u) * 2 + self.rank, x * y
            else:
                return 0, (x * y)
        v, s = 0, 0
        for child in self.children:
            res = child.compute_compression_ratio_()
            v += res[0]
            s += res[1]
        return v, s

    def compute_compression_ratio(self):
        v, s = self.compute_compression_ratio_()
        return s / v

def draw_tree(root, title=''):
    image = np.ones(root.matrix.shape)*255
    Q = deque()
    Q.append(root)
```

```
    while Q:
        v = Q.pop()
        if v.leaf:
            image[v.row_min:v.row_max, v.col_min:v.col_min+v.rank] = np.zeros((v.ro
            image[v.row_min:v.row_min+v.rank, v.col_min:v.col_max] = np.zeros((v.ra
            image[v.row_min, v.col_min:v.col_max] = np.zeros((1,v.col_max - v.col_m
            image[v.row_max-1, v.col_min:v.col_max] = np.zeros((1,v.col_max - v.col
            image[v.row_min:v.row_max,v.col_min] = np.zeros(v.row_max-v.row_min)
            image[v.row_min:v.row_max,v.col_max-1] = np.zeros(v.row_max-v.row_min)
        else:
            for child in v.children:
                Q.append(child)

    plt.imshow(image, cmap="gist_gray", vmin=0, vmax=255)
    plt.title(title)
    plt.show()

def compress_matrix(M):
    k = M.shape[0]
    eps = randomized_svd(M, n_components=k, random_state=0)[1][k - 1]
    root = MatrixTree(M, 0, k, 0, k)
    root.compress(1, eps)
    return root
```

Generacja i kompresja macierzy opisującej topologie siatki 3d

```
import numpy as np

def generate(k):
    n = 2 ** (3 * k)
    M = np.zeros((n, n))
    jumps = [0, -1, 1, -2**(k), 2**(k), -2**(k*2), 2**(k*2)]
    for v in range(n):
        for j in jumps:
            if v + j >= 0 and v + j < n:
                M[v][v + j] = 1
    return M * np.random.random((n, n))
```

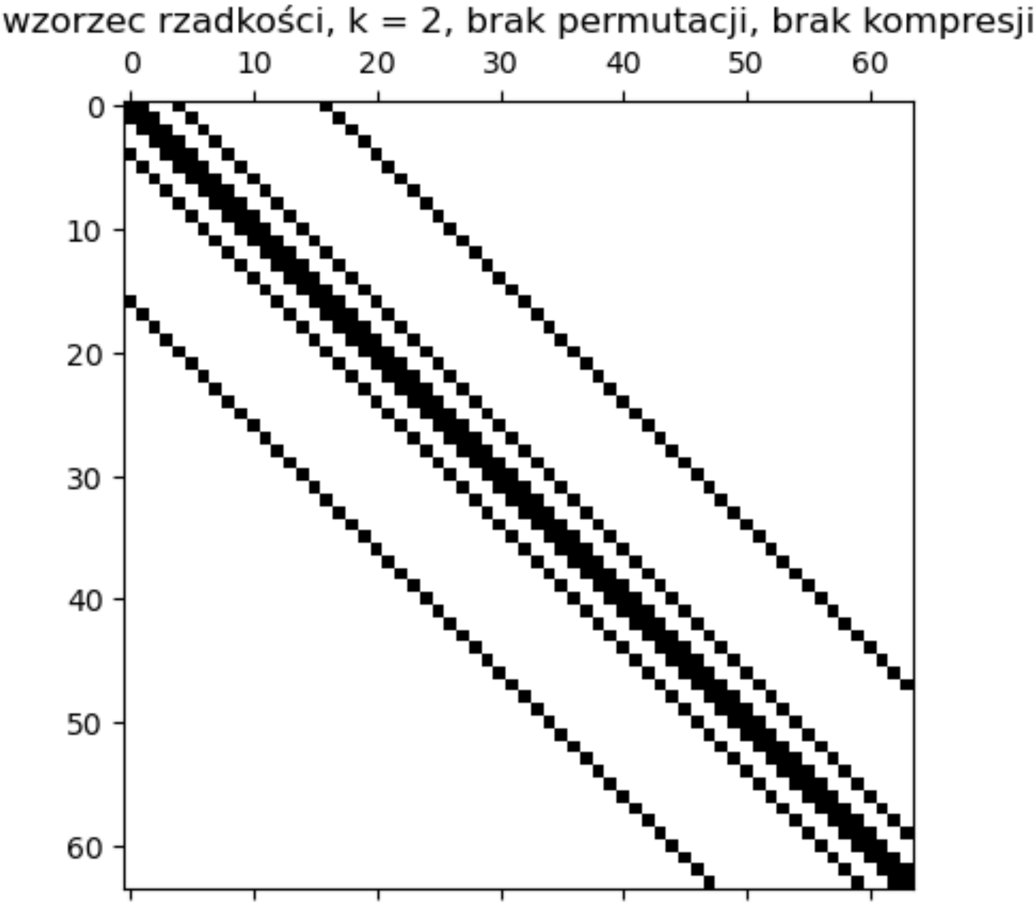
```
test_matrices = {k:generate(k) for k in [2, 3, 4]}
test_matrices_compressed = {k:compress_matrix(test_matrices[k]) for k in [2, 3, 4]}
test_matrices_perm = {(k, algo):apply_permutation(test_matrices[k], algo) \
                      for k in [2, 3, 4] for algo in [minimum_degree_permutation, c
test_matrices_perm_compressed = {(k, algo):compress_matrix(apply_permutation(test_m
                                 for k in [2, 3, 4] for algo in [minimum_degree_per
```

Wizualizacja

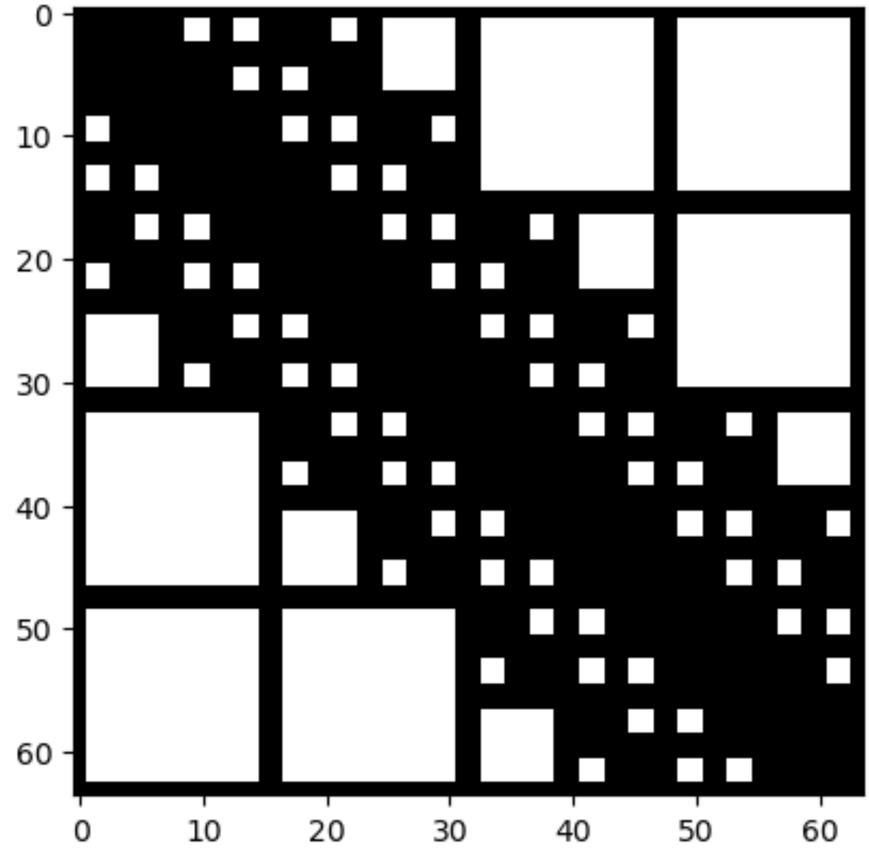
```
from matplotlib.pyplot import spy
import matplotlib.pyplot as plt

def draw_sparsity_pattern(M, title):
    fig, ax = plt.subplots()
    ax.set_title(title)
    ax.spy(M)
    plt.show()
```

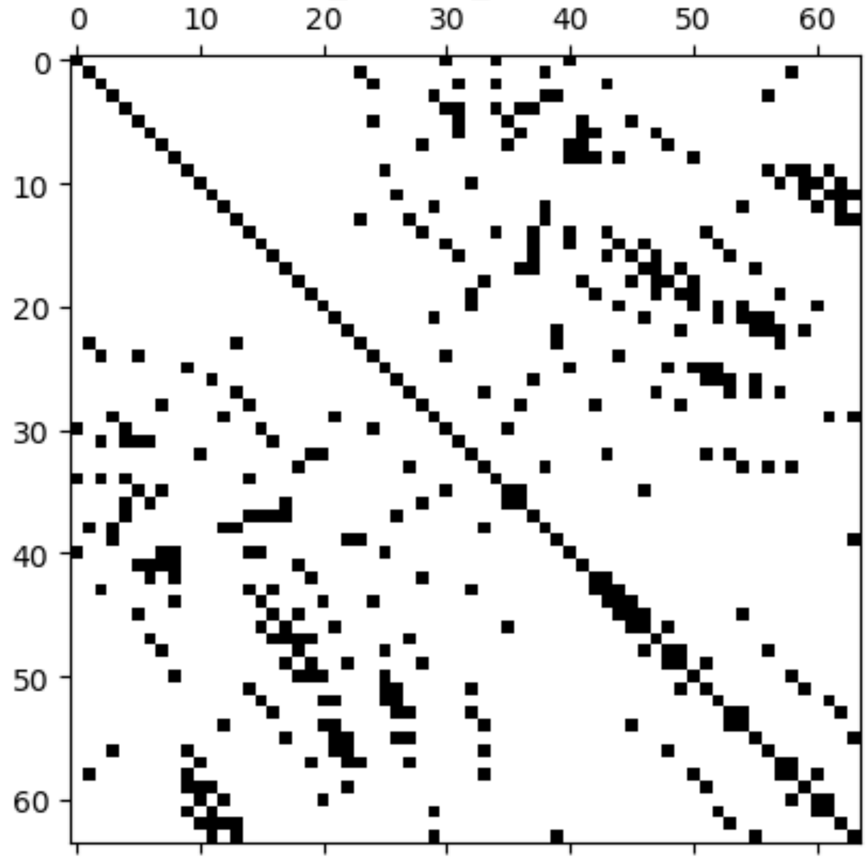
```
for k in [2, 3, 4]:
    M = test_matrices[k]
    draw_sparsity_pattern(M, f'wzorzec rzadkości, k = {k}, brak permutacji, brak ko
    draw_tree(test_matrices_compressed[k], title=f'wizualizacja macierzy, k = {k}, 
    for algo in [minimum_degree_permutation, cuthill_mckee, reversed_cuthill_mckee]
        draw_sparsity_pattern(test_matrices_perm[(k, algo)], title=f'wzorzec rzadko
        draw_tree(test_matrices_perm_compressed[(k, algo)], title=f'wzorzec rzadkoś
```



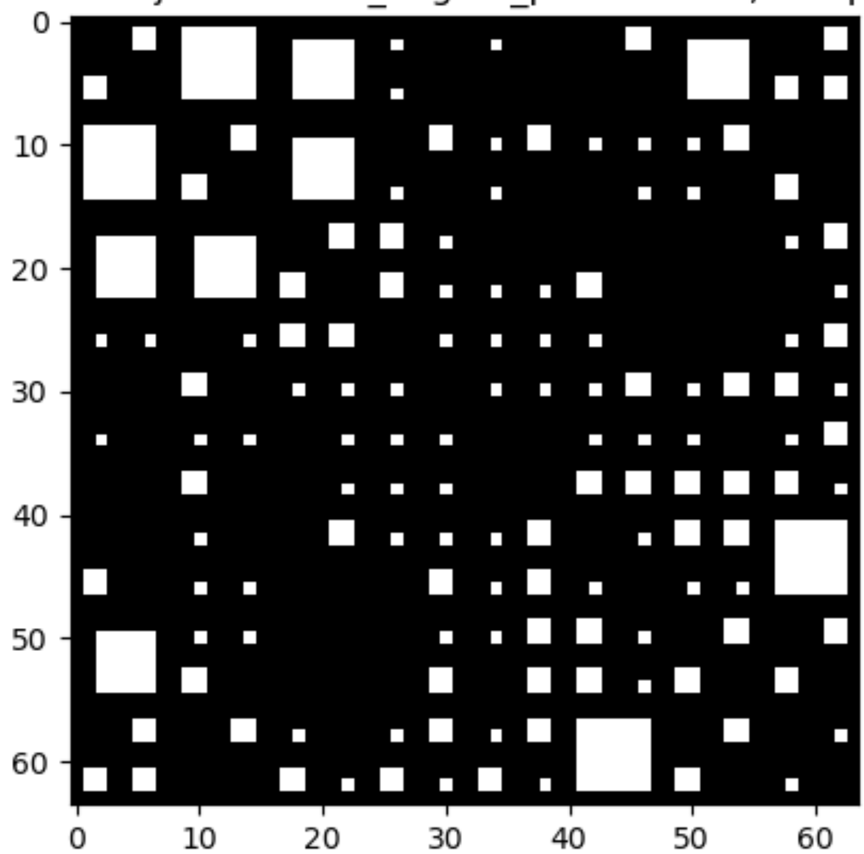
wizualizacja macierzy, k = 2, brak permutacji, kompresja



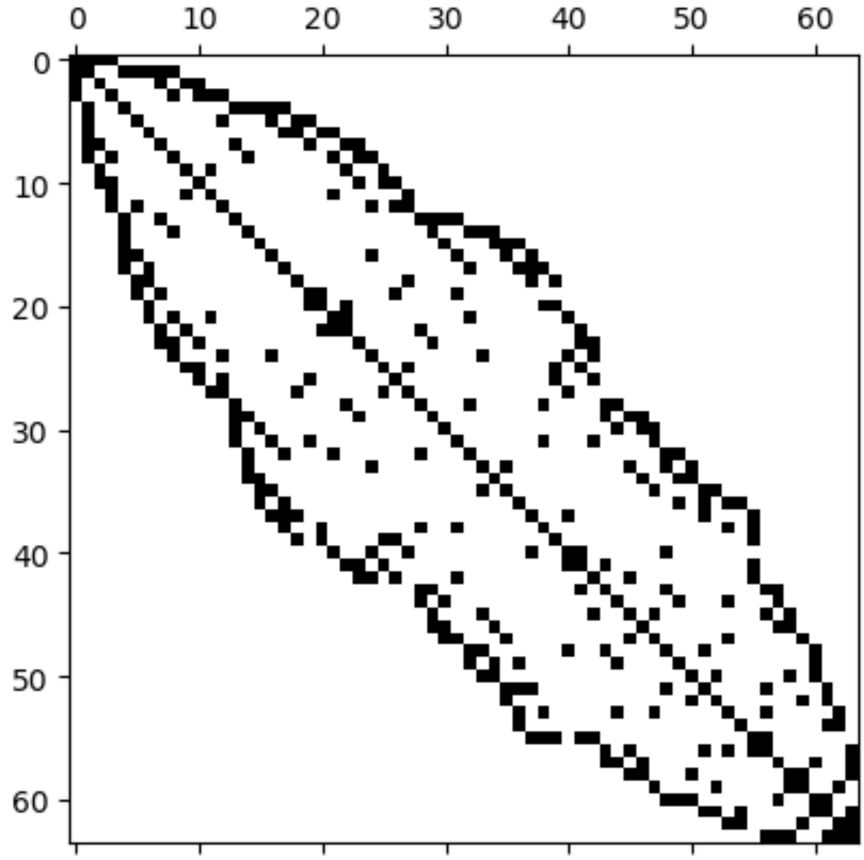
wzorzec rzadkości, k = 2,
permutacja:minimum_degree_permutation, brak kompresji

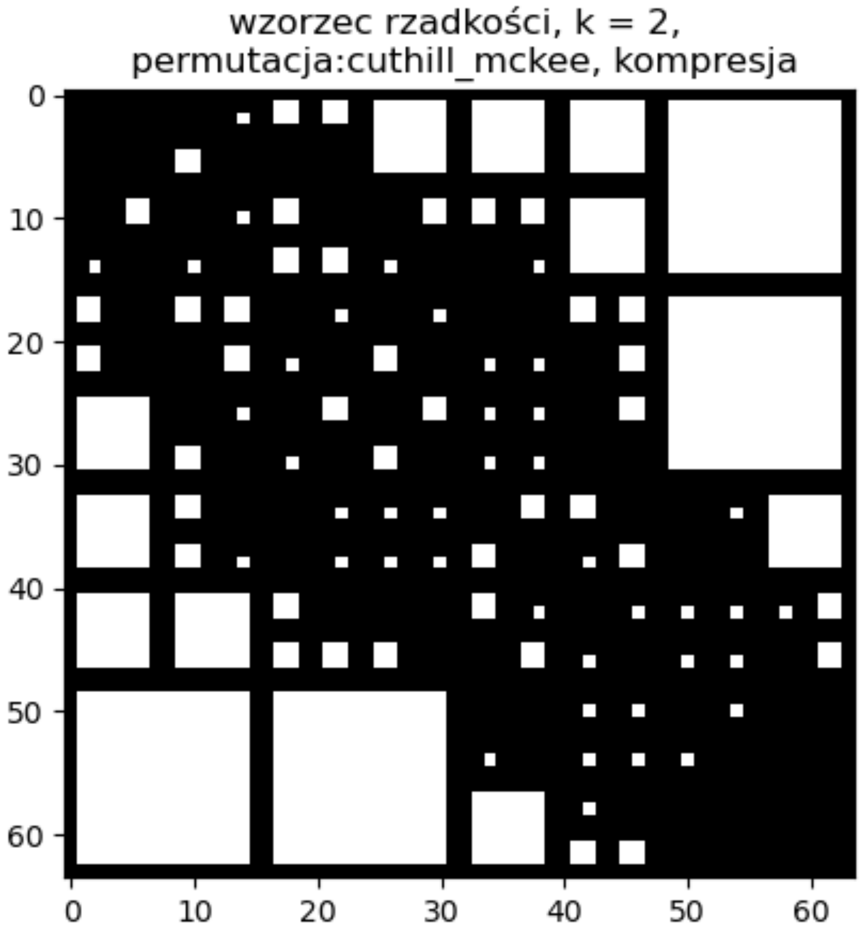


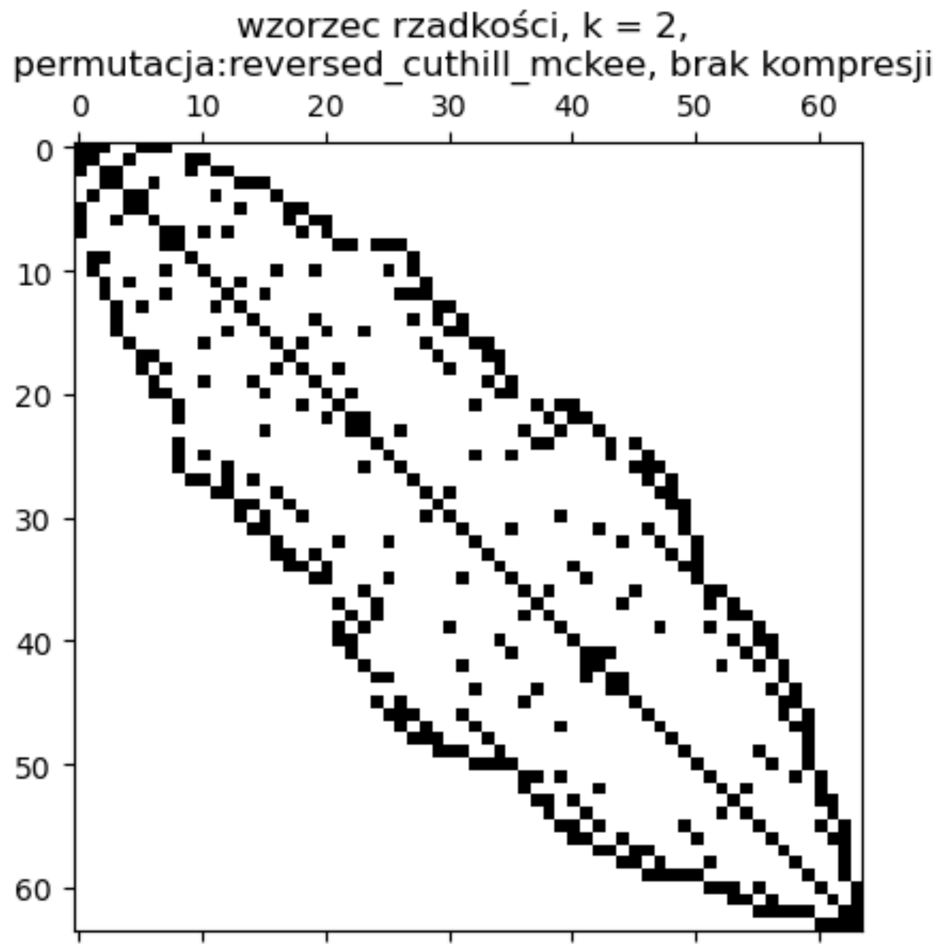
wzorzec rzadkości, k = 2,
permutacja:minimum_degree_permutation, kompresja

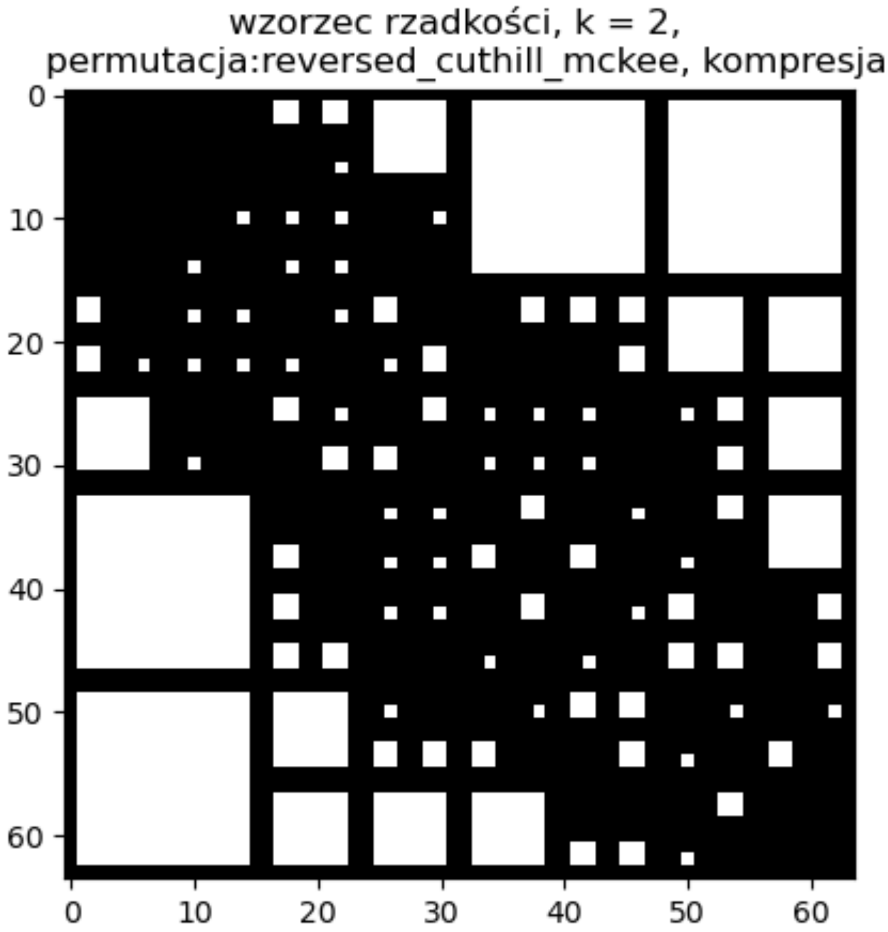


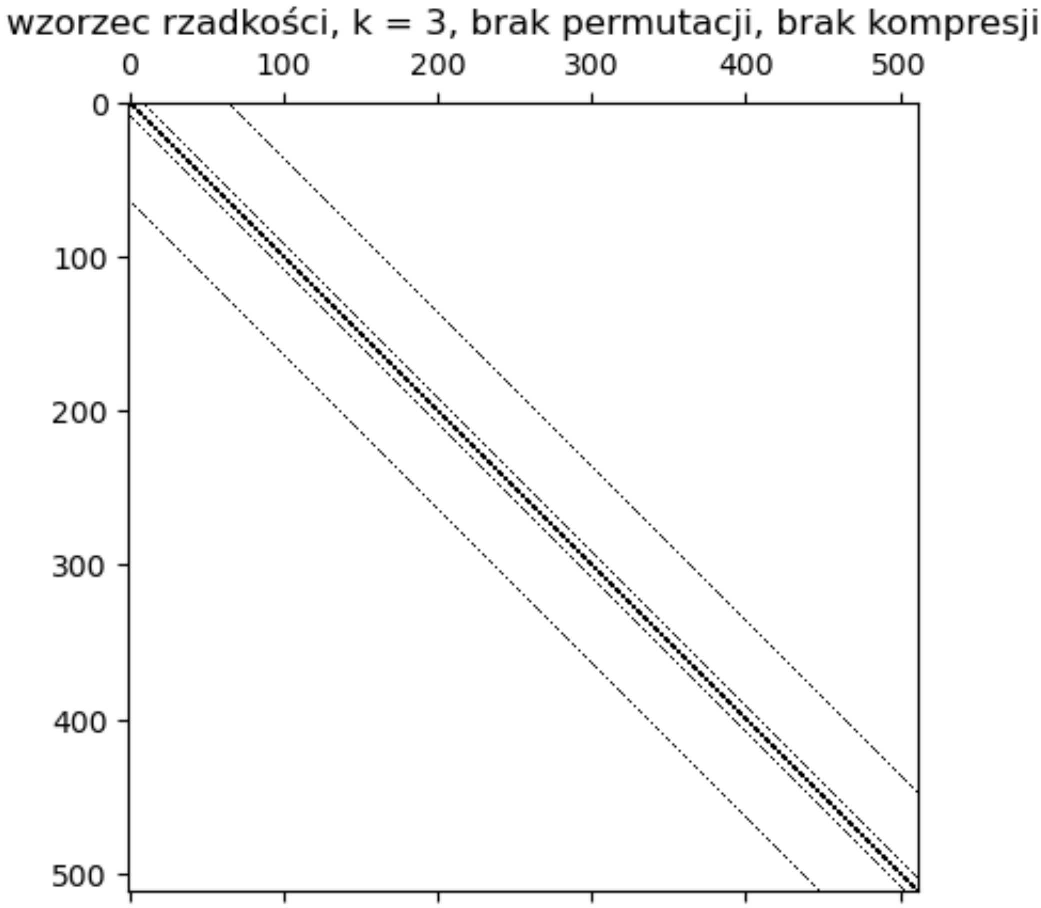
wzorzec rzadkości, k = 2,
permutacja:cuthill_mckee, brak kompresji



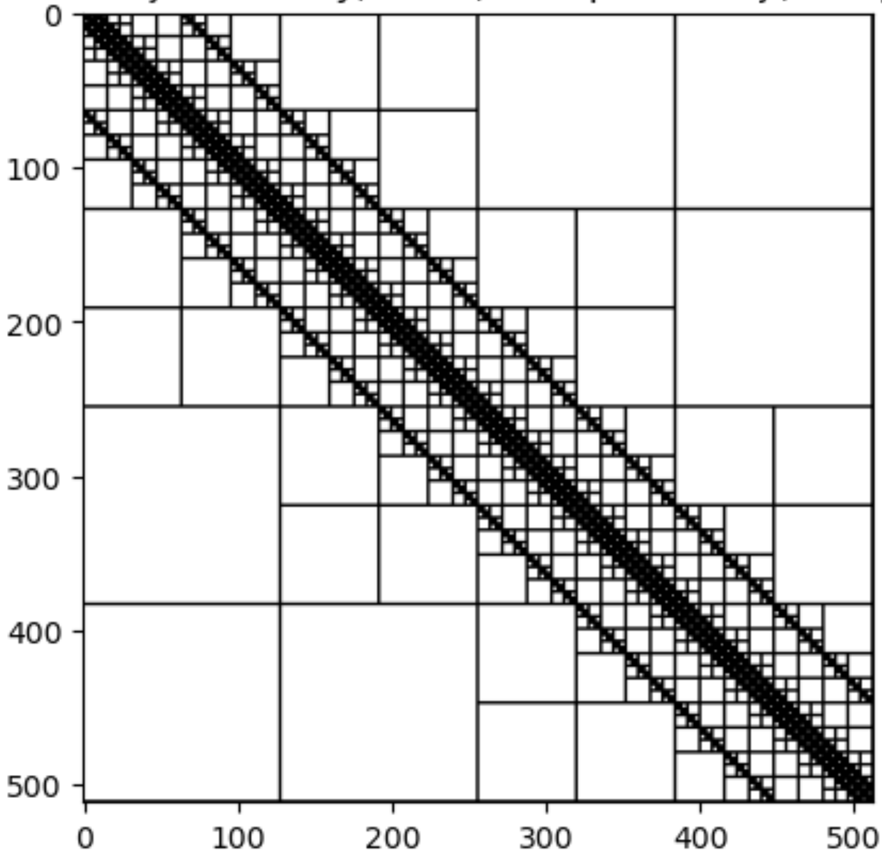

wzorzec rzadkości, k = 2,
permutacja:cuthill_mckee, kompresja


wzorzec rzadkości, k = 2,
permutacja:reversed_cuthill_mckee, brak kompresji

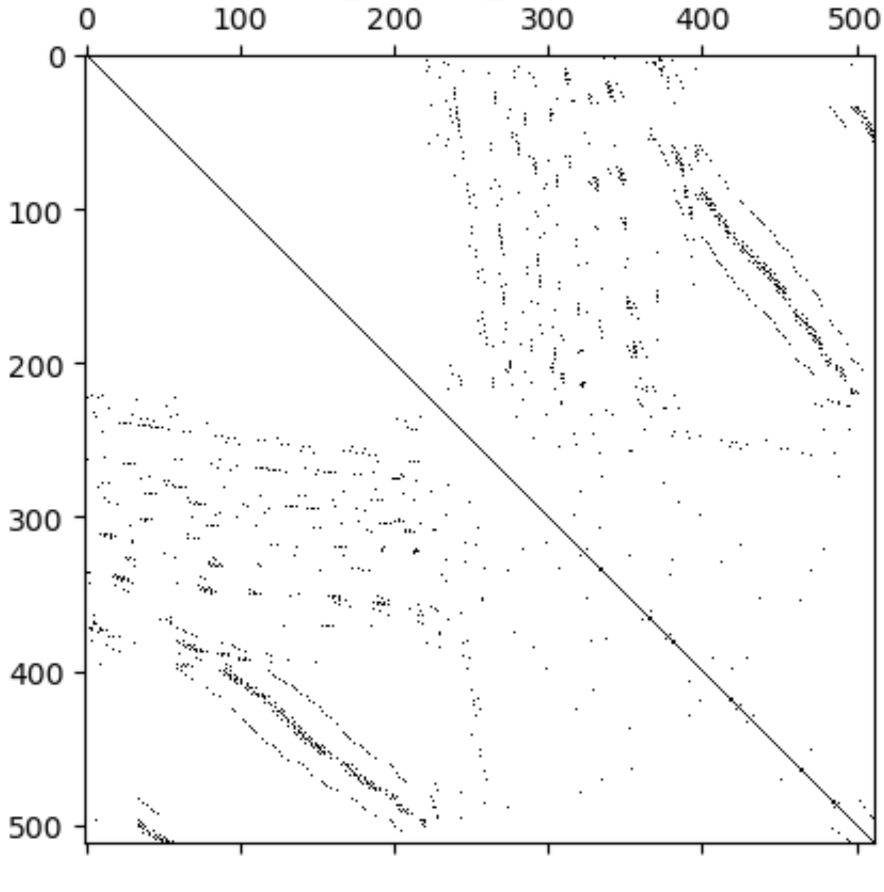

wzorzec rzadkości, k = 2,
permutacja:reversed_cuthill_mckee, kompresja
0
10
20
30
40
50
60


wzorzec rzadkości, k = 3, brak permutacji, brak kompresji
0
100
200
300
400
500

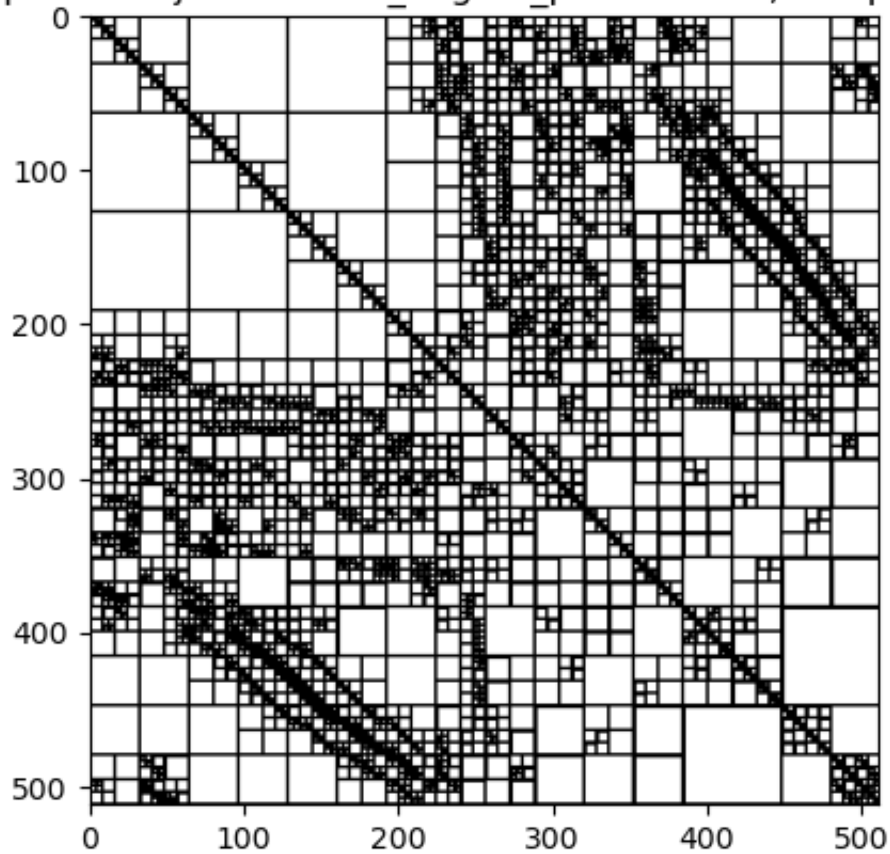
wizualizacja macierzy, k = 3, brak permutacji, kompresja



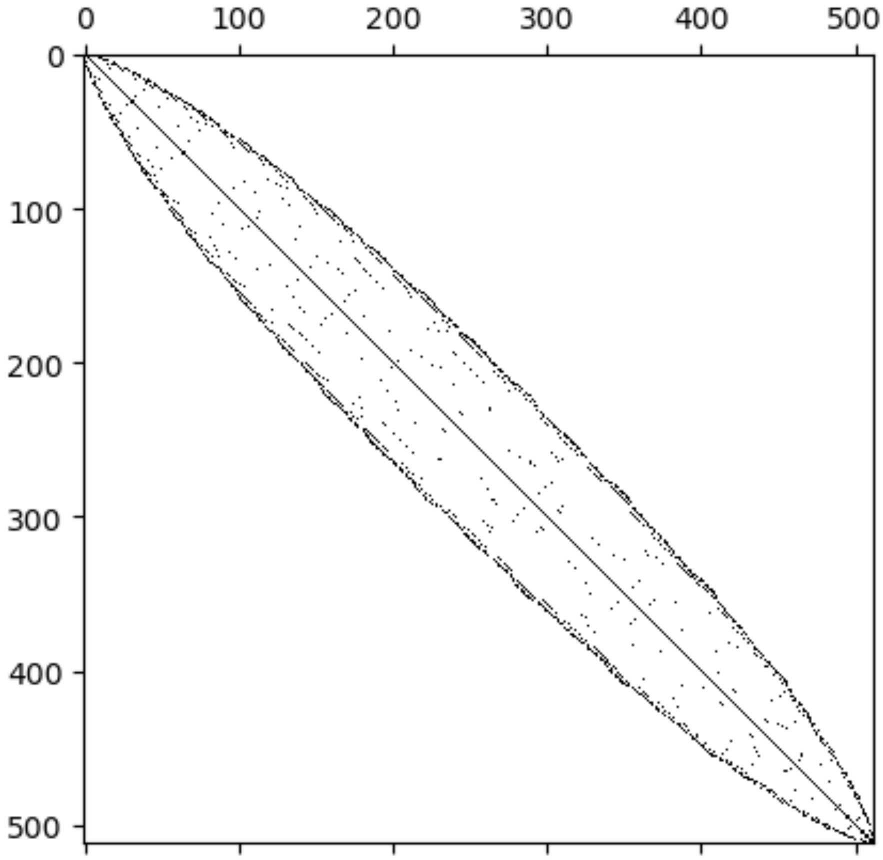
wzorzec rzadkości, k = 3,
permutacja:minimum_degree_permutation, brak kompresji



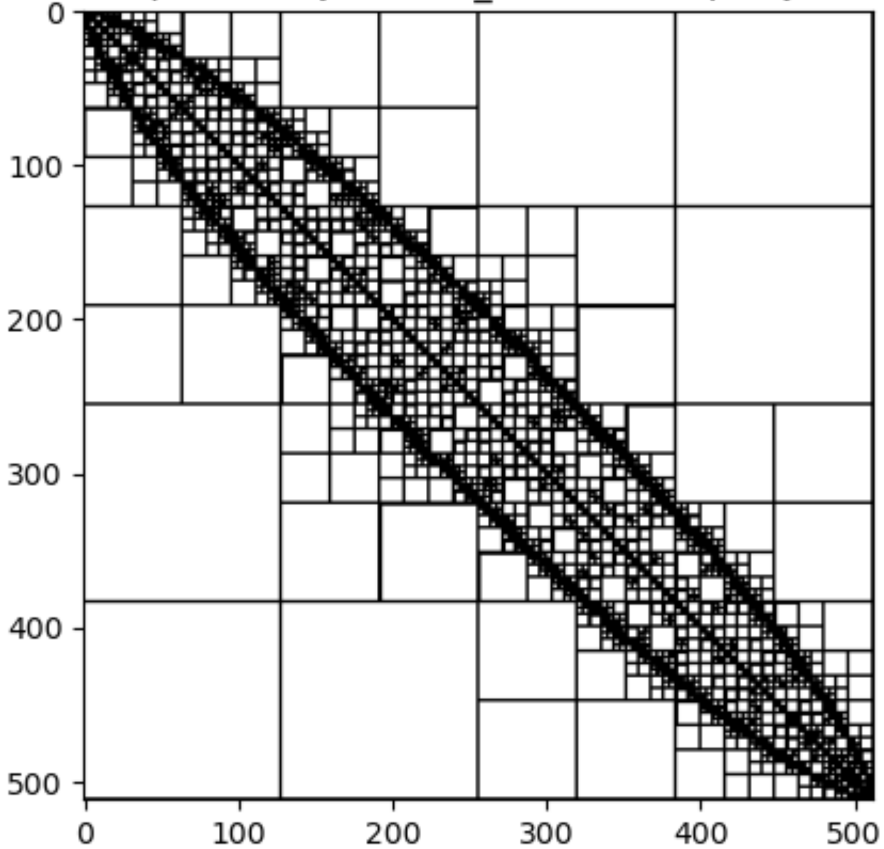
wzorzec rzadkości, k = 3,
permutacja:minimum_degree_permutation, kompresja



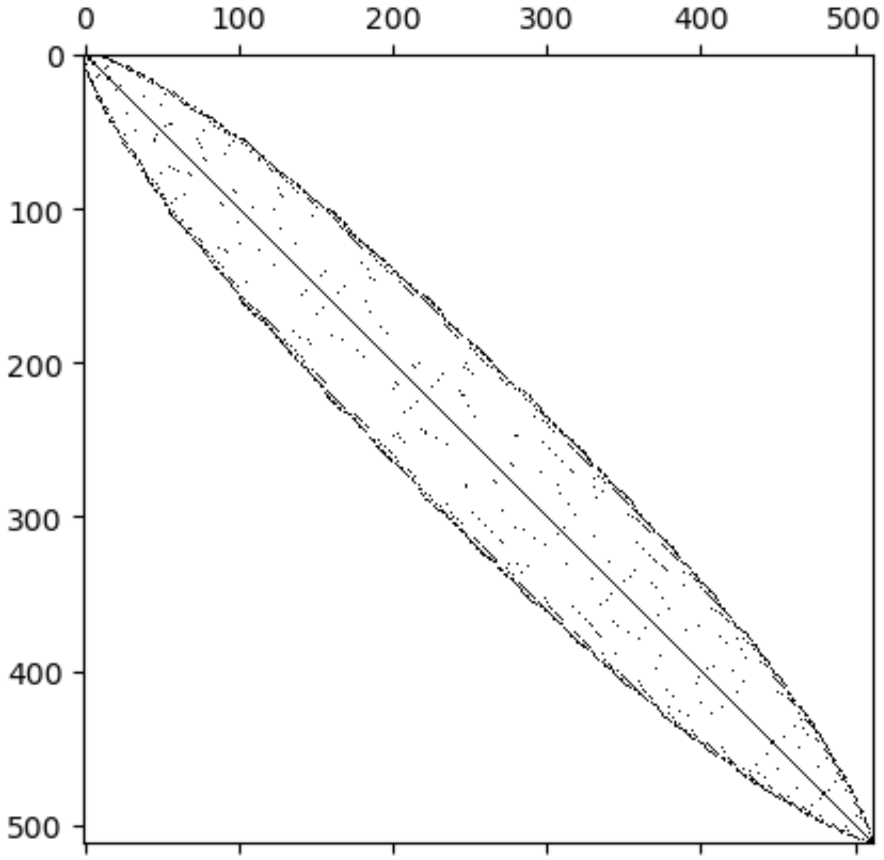
wzorzec rzadkości, k = 3,
permutacja:cuthill_mckee, brak kompresji

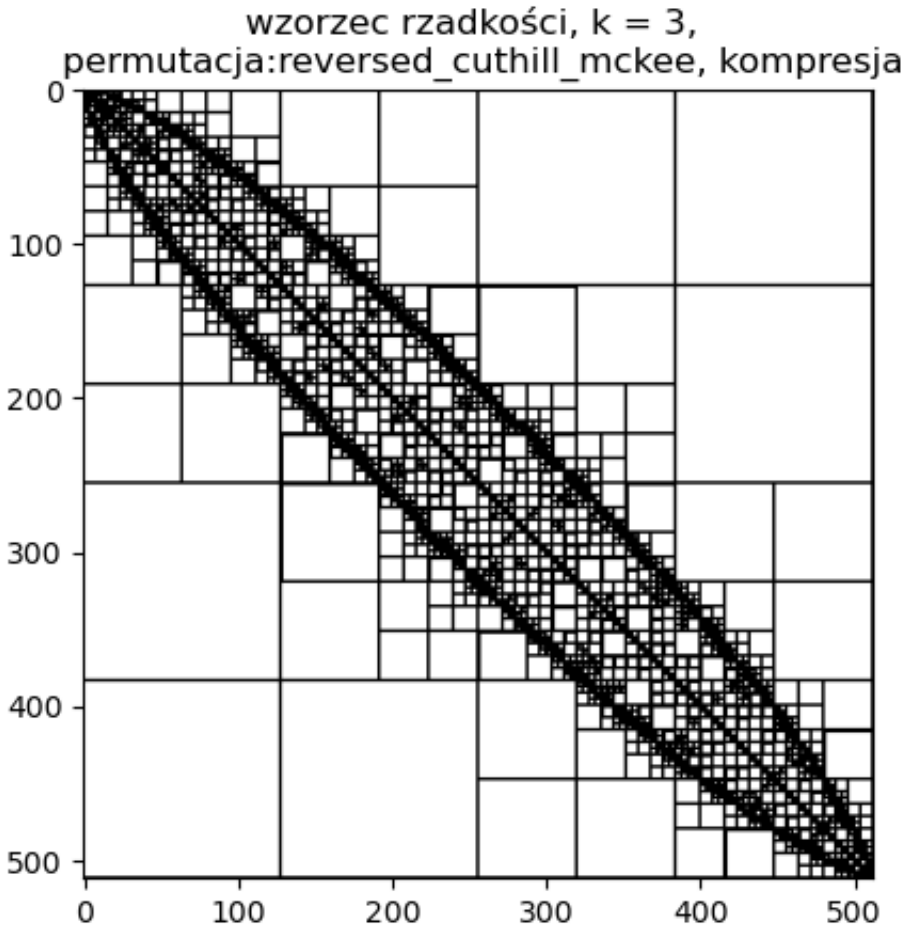


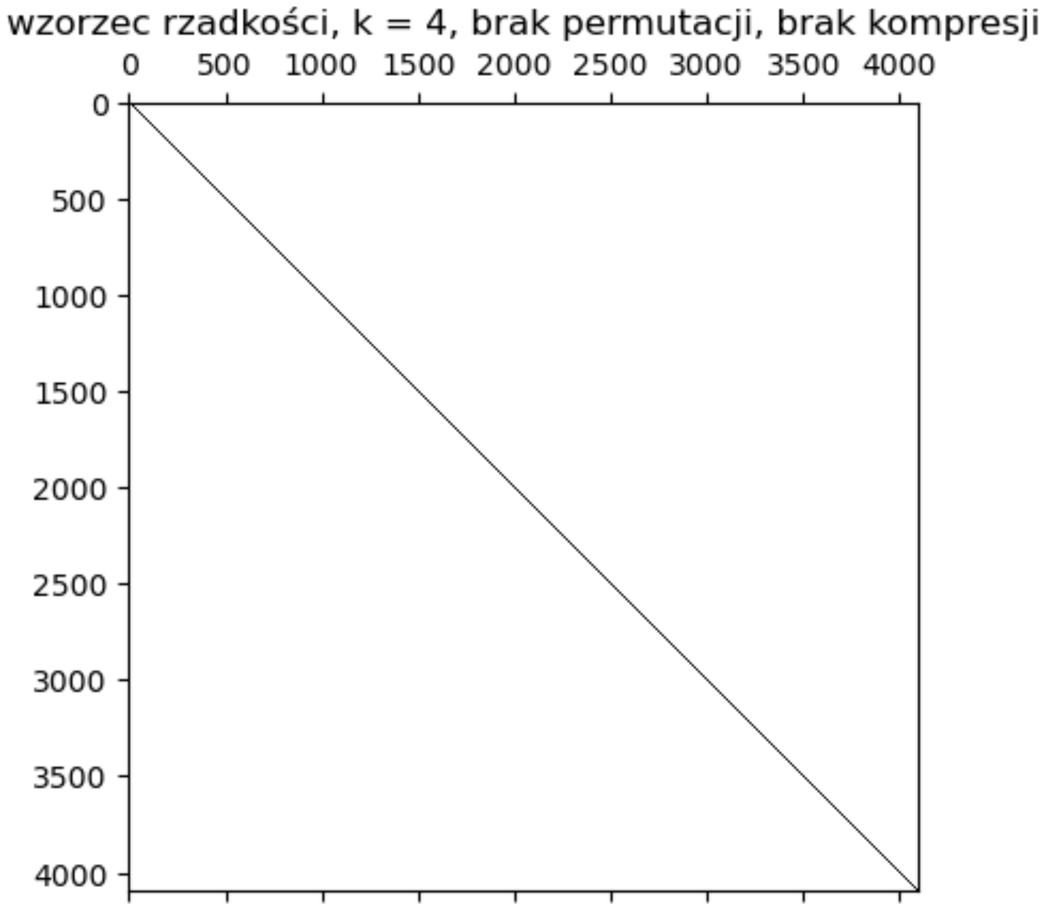
wzorzec rzadkości, k = 3,
permutacja:cuthill_mckee, kompresja

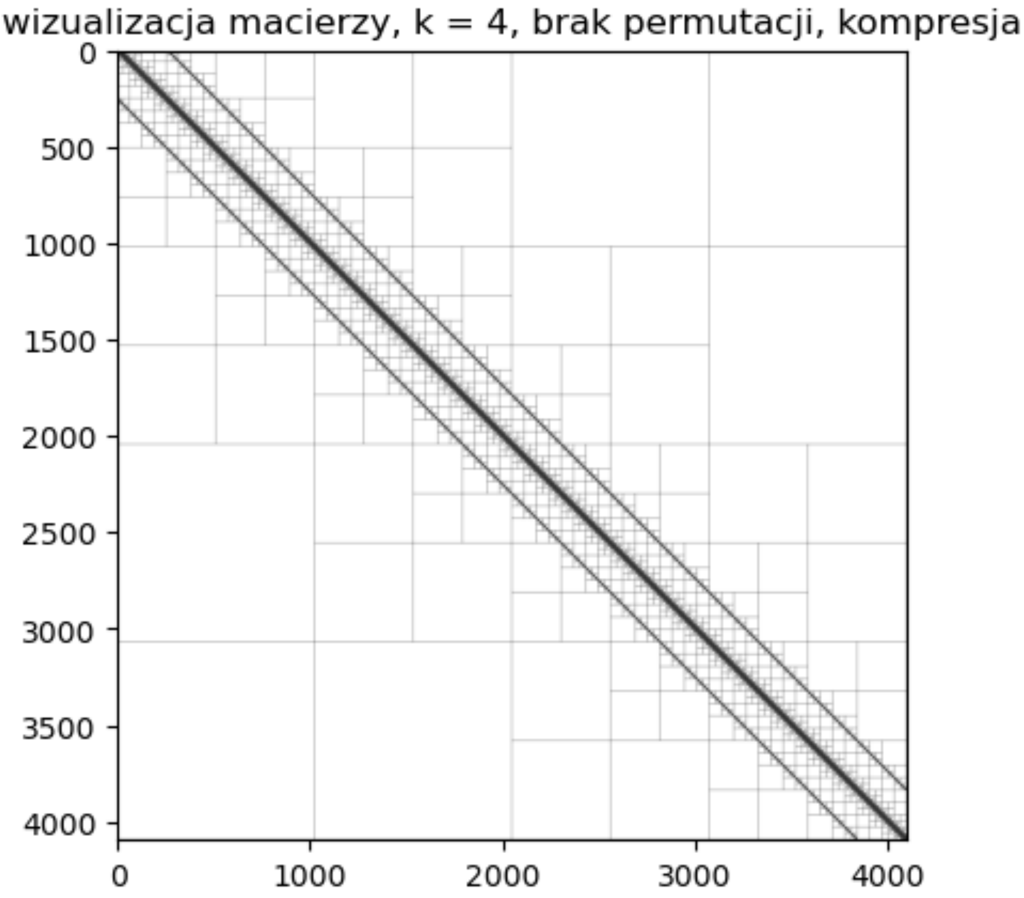


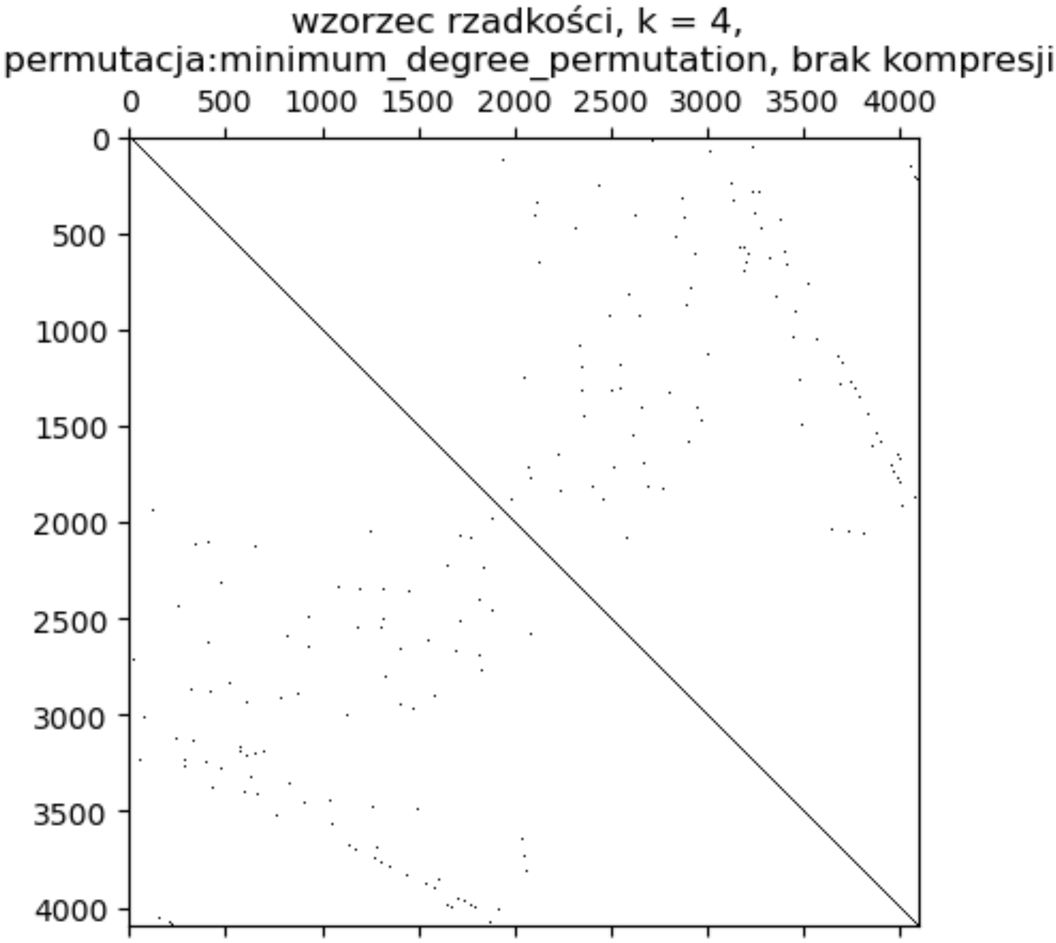
wzorzec rzadkości, k = 3,
permutacja:reversed_cuthill_mckee, brak kompresji

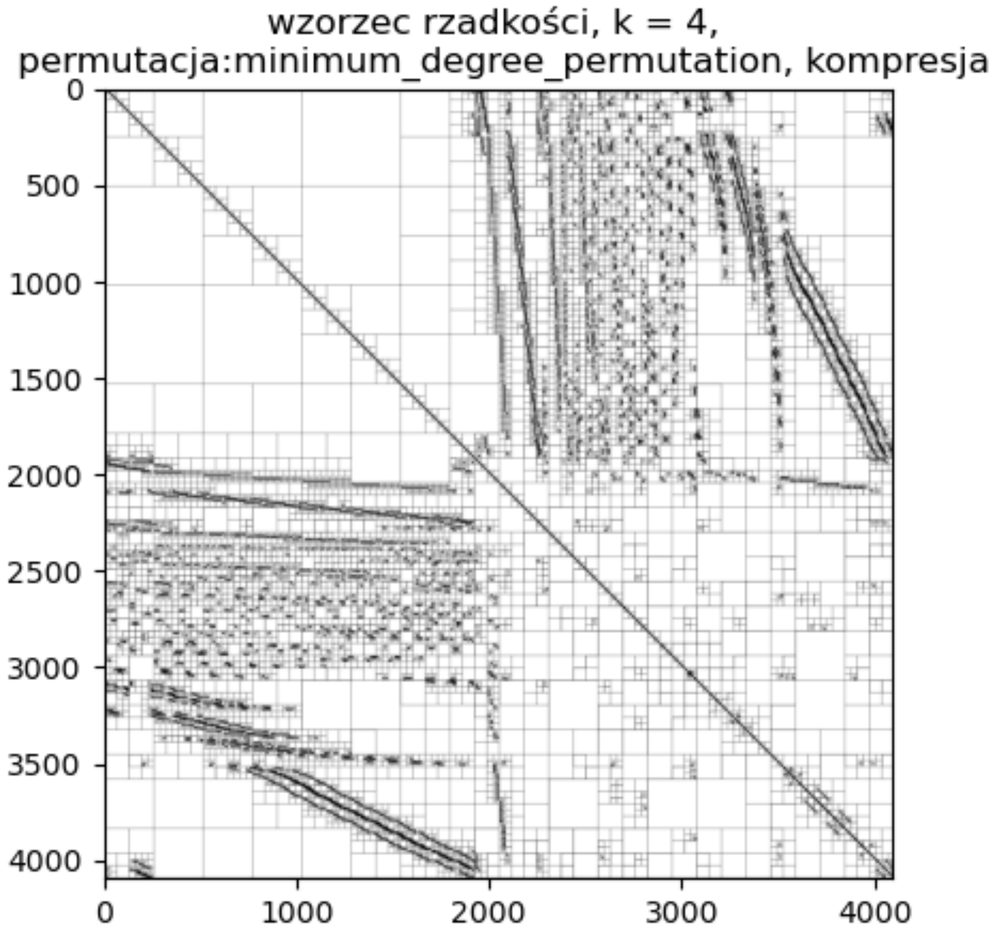


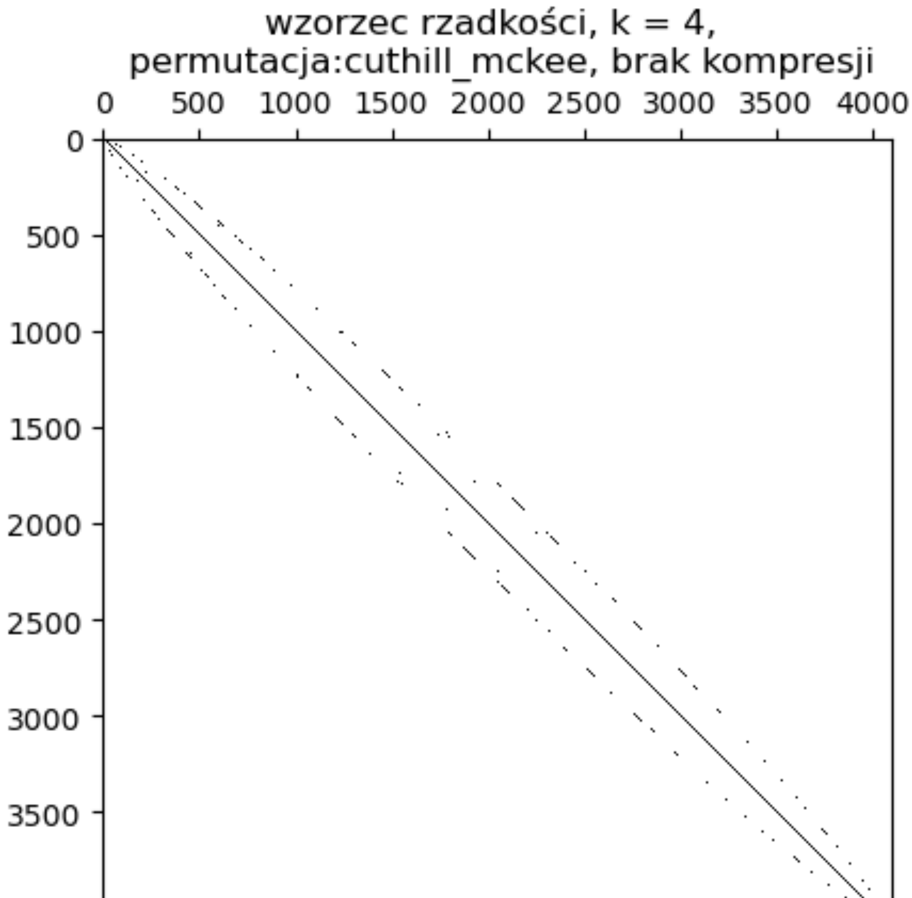

wzorzec rzadkości, k = 3,
permutacja:reversed_cuthill_mckee, kompresja
0
100
200
300
400
500
0
100
200
300
400
500

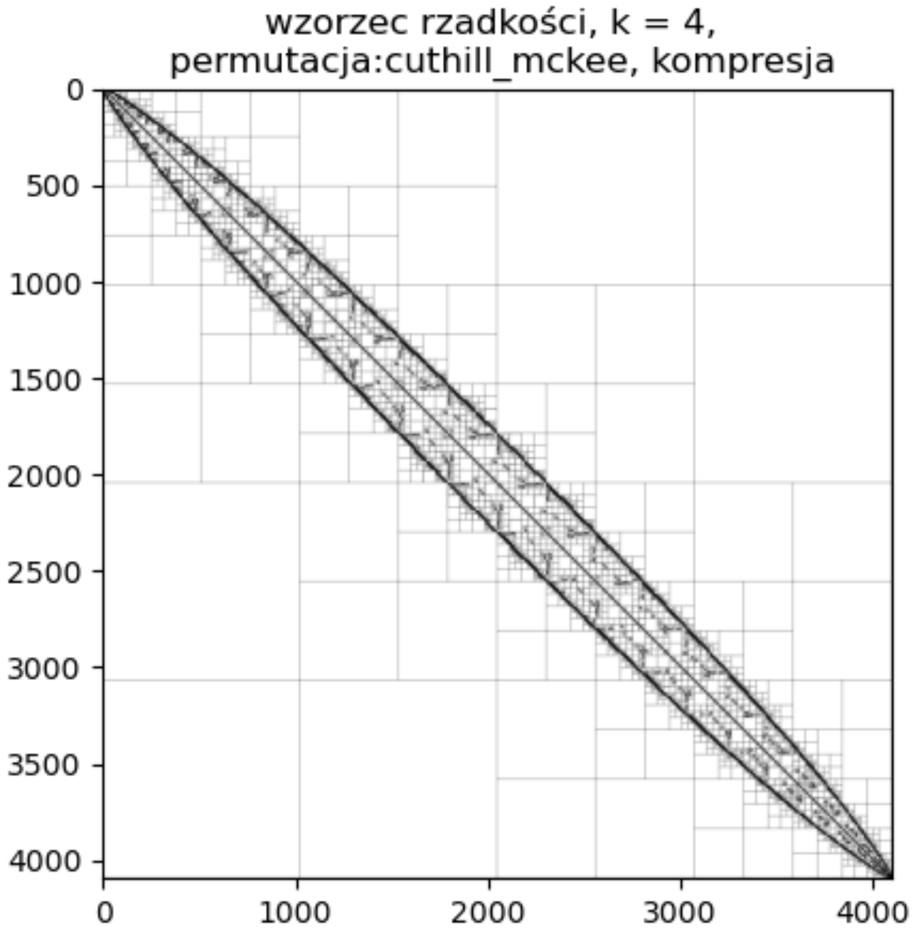

wzorzec rzadkości, k = 4, brak permutacji, brak kompresji
0
500
1000
1500
2000
2500
3000
3500
4000
0
500
1000
1500
2000
2500
3000
3500
4000

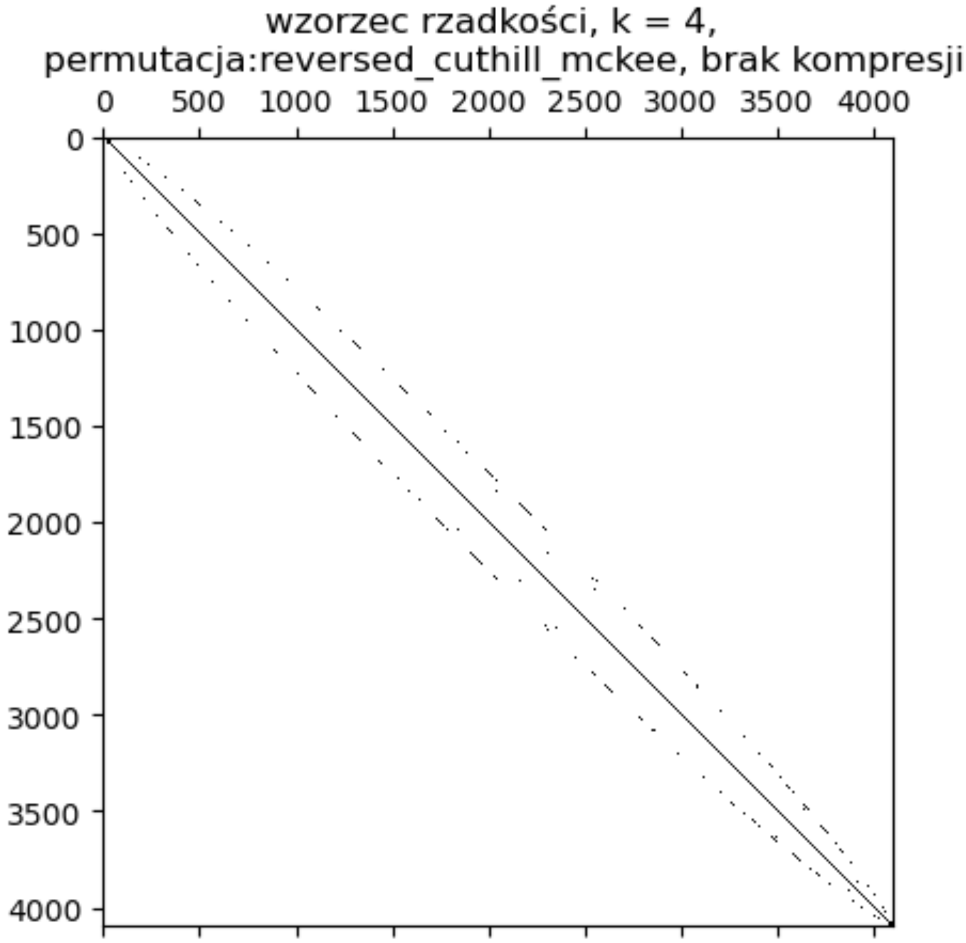

wizualizacja macierzy, k = 4, brak permutacji, kompresja
0
500
1000
1500
2000
2500
3000
3500
4000
0
1000
2000
3000
4000

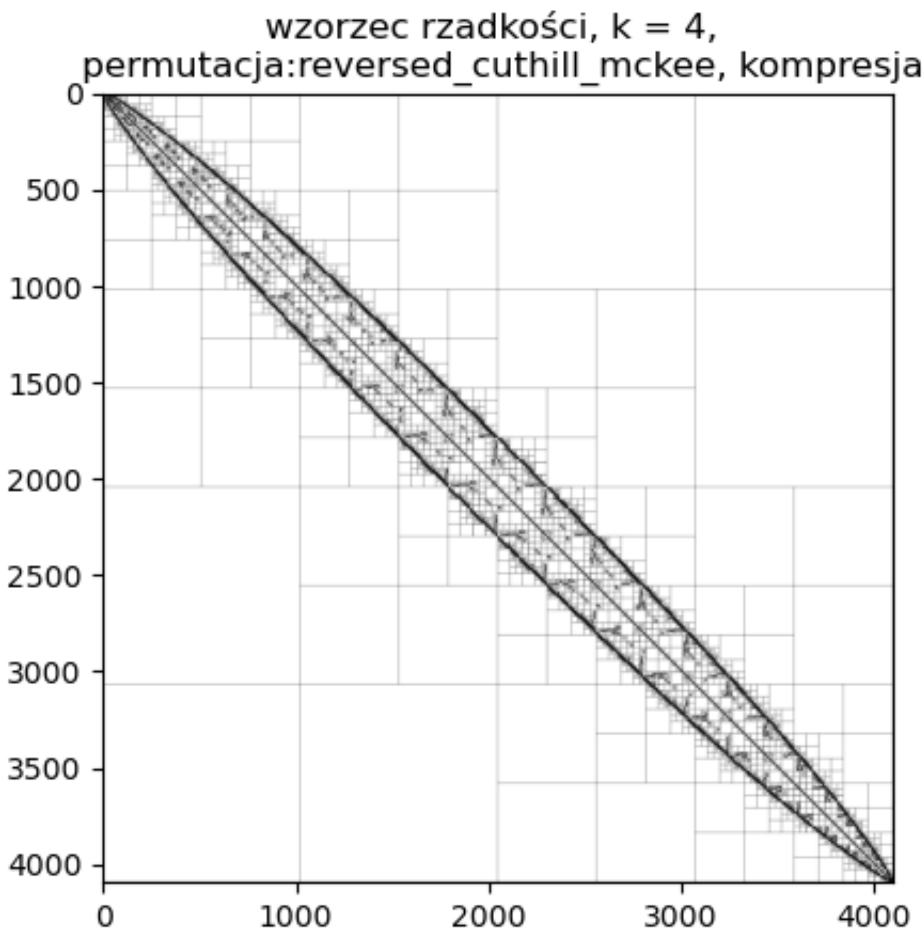

wzorzec rzadkości, k = 4,
permutacja:minimum_degree_permutation, brak kompresji
0
500
1000
1500
2000
2500
3000
3500
4000
0
500
1000
1500
2000
2500
3000
3500
4000


wzorzec rzadkości, k = 4,
permutacja:minimum_degree_permutation, kompresja
0
500
1000
1500
2000
2500
3000
3500
4000
0
1000
2000
3000
4000


wzorzec rzadkości, k = 4,
permutacja:cuthill_mckee, brak kompresji
0
500
1000
1500
2000
2500
3000
3500
4000
0
500
1000
1500
2000
2500
3000
3500
4000


wzorzec rzadkości, k = 4,
permutacja:cuthill_mckee, kompresja
0
500
1000
1500
2000
2500
3000
3500
4000
0
1000
2000
3000
4000


wzorzec rzadkości, k = 4,
permutacja:reversed_cuthill_mckee, brak kompresji
0
500
1000
1500
2000
2500
3000
3500
4000
0
500
1000
1500
2000
2500
3000
3500
4000



Porównanie stopnia kompresji macierzy

```
import pandas as pd

df = pd.DataFrame(columns=['k', 'ratio - brak permutacji',
                           'ratio - mdp', 'ratio - cuthill_mckee',
                           'ratio - reversed cuthill_mckee'])

for k in [2, 3, 4]:
    rc = test_matrices_compressed[k].compute_compression_ratio()
    new_row = {'k':k, 'ratio - brak permutacji': rc}
    for algo, name in [(minimum_degree_permutation, 'ratio - mdp'),
                       (cuthill_mckee, 'ratio - cuthill_mckee'),
                       (reversed_cuthill_mckee, 'ratio - reversed cuthill_mckee')]:
        rc = test_matrices_perm_compressed[(k, algo)].compute_compression_ratio()
        new_row[name] = rc
    df.loc[len(df)] = new_row
```

```
df.style.hide(axis='index')
```

Out[ ]:

k	ratio - brak permutacji	ratio - mdp	ratio - cuthill_mckee	ratio - reversed cuthill_mckee
2	2.461538	1.210760	1.566348	1.566348
3	16.468401	5.693090	9.449355	9.449355
4	115.104805	29.378357	73.624645	73.624645

Wnioski

Na podstawie wygenerowanych wizualizacji można stwierdzić, że wzorzec rzadkości dla macierzy przed kompresją dla każdego z zastosowanych algorytmów permutacji odpowiada wizualizacji skompresowanej macierzy hierarchicznej. Wzorzec dla macierzy bez permutacji pokazuję że wartości niezerowe są rozłożone regularnie w okolicy przekątnej macierzy. Dla algorytmu minimal degree widać że wartości niezerowę są bardziej 'rozrzucone' i znajdują się też dalej od przekątnej, wzór wydaje się być uporządkowany. W przypadku Cuthill–McKee i Reversed Cuthill–McKee we wzorcu widoczna jest wstęga wokół przekątnej co sugeruje poprawne działanie algorytmu. Stopień kompresji macierzy okazał się zależeć istotnie od zastosowanego algorytmu permutacji, najepszy stopeń udało się w każdym przypadku osiągnąć dla wersji bez permutacji, oba warianty algorytmu Cuthill–McKee osiągały podobne wartości i były lepsze od najsłabszego w tym przypadku algorytmu minimal degree.