Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ КОМП'ЮТЕРНИХ НАУК ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Катедра «Комп'ютерна інженерія та програмування»

ЗВІТ

про виконання лабораторної роботи №5

з навчальної дисципліни «Алгоритми та структури даних»

Варіант 9

Виконав студент:

Ульянов Кирило Юрійович
Група: КН-1023b

Перевірив:

старший викладач

Бульба С.С.

Харків-2024

1 Мета роботи

Набуття і закріплення навичок програмування розміщення в пам’яті специфічних масивів.

2 Хід роботи

Розробити спосіб економного розміщення в пам’яті заданої розрідженої таблиці, де записані цілі числа. Розробити функції, що забезпечують доступ до елементів таблиці за номерами рядка і стовпця.

У програмі забезпечити запис і читання всіх елементів таблиці.

Визначити та порівняти час доступу до елементів таблиці при традиційному та економному поданні її в пам’яті. Зробити висновки.

Завдання обрати з табл. 5.1 згідно із своїм номером у журналі групи.

Моє завдання: нульові елементи розташовані на місцях з парними індексами рядків і стовпців.

2.1 Обраний підхід до побудови таблиці

Для вирішення моєї задачі я вирішив використовувати формат розріджених рядків (**CSR — Compressed Sparse Row**). Після дослідження інформації, я зрозумів що цей метод є ефективним для зберігання розрідженних таблиць та забезпечує оптимальний доступ до них по рядкам та стовбцях.

Формат CSR зберігає ненульові елементи з їхніми відповідними індексами стовпців і покажчиками на початок кожного рядка, що робить його особливо зручним для випадків, де нульові елементи мають специфічний розподіл.

2.2 Реалізація CSR таблиці

В приведеній нижче реалізації було створено міні-бібліотеку для перетворення звичайної матриці у CSR формат та також додаткові методи для роботи з ними.

Було реалізовано такі методи: **створення таблиці, отримання елемента за індексом, вивід таблиці, знищення таблиці.**

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  int *values;
  int *colIndex;
  int *rowPtr;
  int nonZeroCount;
} CSRMatrix;

CSRMatrix create(int **matrix, int rows, int cols) {
  CSRMatrix csr;
  int nonzero_elem_count = 0;

  for (size_t i = 0; i < rows; i++) {
    for (size_t j = 0; j < cols; j++) {
      if (!(i % 2 == 0 && j % 2 == 0) && matrix[i][j] != 0) {
        nonzero_elem_count++;
      }
    }
  }

  csr.values = (int *)malloc(nonzero_elem_count * sizeof(int));
  csr.colIndex = (int *)malloc(nonzero_elem_count * sizeof(int));
  csr.rowPtr = (int *)malloc((rows + 1) * sizeof(int));
  csr.nonZeroCount = nonzero_elem_count;

  int k = 0;
  csr.rowPtr[0] = 0; // start of first row
  for (size_t i = 0; i < rows; i++) {
    for (size_t j = 0; j < cols; j++) {
      if (!(i % 2 == 0 && j % 2 == 0) && matrix[i][j] != 0) {
        csr.values[k] = matrix[i][j];
        csr.colIndex[k] = j;
        k++;
      }
    }
    csr.rowPtr[i + 1] = k; // end of current row
  }

  return csr;
}

int get_element(CSRMatrix *csr, int row, int col) {
  for (size_t i = csr->rowPtr[row]; i < csr->rowPtr[row + 1]; i++) {
    if (csr->colIndex[i] == col) {
      return csr->values[i];
    }
  }
```

```
50
51    return (row % 2 == 0 && col % 2 == 0) ? 0 : 0;
52  }
53
54  void print(CSRMatrix csr, int rows, int cols) {
55    printf("\033[33mValues:\033[0m ");
56    for (int i = 0; i < csr.nonZeroCount; i++) {
57      printf("%d ", csr.values[i]);
58    }
59    printf("\n\033[33mColumn indexes for non-zero elements:\033[0m ");
60    for (int i = 0; i < csr.nonZeroCount; i++) {
61      printf("%d ", csr.colIndex[i]);
62    }
63    printf("\n\033[33mPointers to the beginning of each row in the values "
64           "array:\033[0m ");
65    for (int i = 0; i <= rows; i++) {
66      printf("%d ", csr.rowPtr[i]);
67    }
68    printf("\n");
69  }
70
71  void destroy(CSRMatrix *csr) {
72
73    if (csr->values != NULL) {
74      free(csr->values);
75      csr->values = NULL;
76    }
77    if (csr->colIndex != NULL) {
78      free(csr->colIndex);
79      csr->colIndex = NULL;
80    }
81    if (csr->rowPtr != NULL) {
82      free(csr->rowPtr);
83      csr->rowPtr = NULL;
84    }
85    csr->nonZeroCount = 0;
86  }
```

2.3 Реалізація звичайної матриці

Для порівняння роботи CSR матриці зі звичайною, я окремо розробив методи для роботи зі звичайною матрицею.

Було створено такі методи: **вивід матриці, заповнення випадковими цілими числами, знищення матриці.**

```
void print_matrix(int **matrix, int rows, int cols) {
  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
      if (matrix[i][j] == 0) {
        printf("\033[33m%d\033[0m ", matrix[i][j]);
      } else {
        printf("%d ", matrix[i][j]);
      }
    }
    printf("\n");
  }
}

void free_matrix(int **matrix) {
  for (int i = 0; i < N; i++) {
    free(matrix[i]);
  }
  free(matrix);
}

void fill_matrix(int **matrix) {
  for (int i = 0; i < N; i++) {
    matrix[i] = (int *)malloc(N * sizeof(int));
    for (int j = 0; j < N; j++) {
      matrix[i][j] = generateRandomInt(10, 80);
      if (i % 2 == 0 && j % 2 == 0) {
        matrix[i][j] = 0;
      }
    }
  }
}
```

2.4 Реалізація основної програми

Тут було реалізовано основну програму з використанням звичайної матриці та її парсингом у CSR матрицю. Було порівняно доступ до елементів та вивід представлення матриць у консоль.

```c
#include <stdio.h>
#include <stdlib.h>

#include "general_utils.h"
#include "sparse_table.h"

#define N 5

void print_matrix(int **matrix, int rows, int cols) {
  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
      if (matrix[i][j] == 0) {
        printf("\033[33m%d\033[0m ", matrix[i][j]);
      } else {
        printf("%d ", matrix[i][j]);
      }
    }
    printf("\n");
  }
}

void free_matrix(int **matrix) {
  for (int i = 0; i < N; i++) {
    free(matrix[i]);
  }
  free(matrix);
}

void fill_matrix(int **matrix) {
  for (int i = 0; i < N; i++) {
    matrix[i] = (int *)malloc(N * sizeof(int));
    for (int j = 0; j < N; j++) {
      matrix[i][j] = generateRandomInt(10, 80);
      if (i % 2 == 0 && j % 2 == 0) {
        matrix[i][j] = 0;
      }
    }
  }
}

void task1() {
  int **matrix = (int **)malloc(N * sizeof(int *));
  clock_t start_time, end_time;
  double access_time;

  highlightText("ORIGINAL MATRIX", "blue");

  printf("\n");
  fill_matrix(matrix);
  print_matrix(matrix, N, N);
  printf("\n");

  start_time = clock();
```

```
54      printf("Access element at \033[33m(3, 1)\033[0m: %d\n", matrix[3][1]);
55      printf("Access element at \033[33m(0, 0)\033[0m: %d\n", matrix[0][0]);
56      end_time = clock();
57      access_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
58      printf("\nTime taken to access elements of matrix: \033[32m%f\033[0m ms\
        n",
59              access_time);
60      printf("\n\n\n\n");
61
62      highlightText("ORIGINAL MATRIX CONVERTED TO SPARSE MATRIX", "blue");
63      printf("\n");
64
65      CSRMatrix csr = create(matrix, N, N);
66      print(csr, N, N);
67      printf("\n");
68
69      start_time = clock();
70      printf("Access element at \033[33m(3, 1)\033[0m: %d\n",
71             get_element(&csr, 3, 1));
72      printf("Access element at \033[33m(0, 0)\033[0m: %d\n",
73             get_element(&csr, 0, 0));
74      end_time = clock();
75      access_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
76      printf(
77          "\nTime taken to access elements of CSR matrix: \033[32m%f\033[0m ms
            \n",
78          access_time);
79
80      free_matrix(matrix);
81      destroy(&csr);
82  }
```

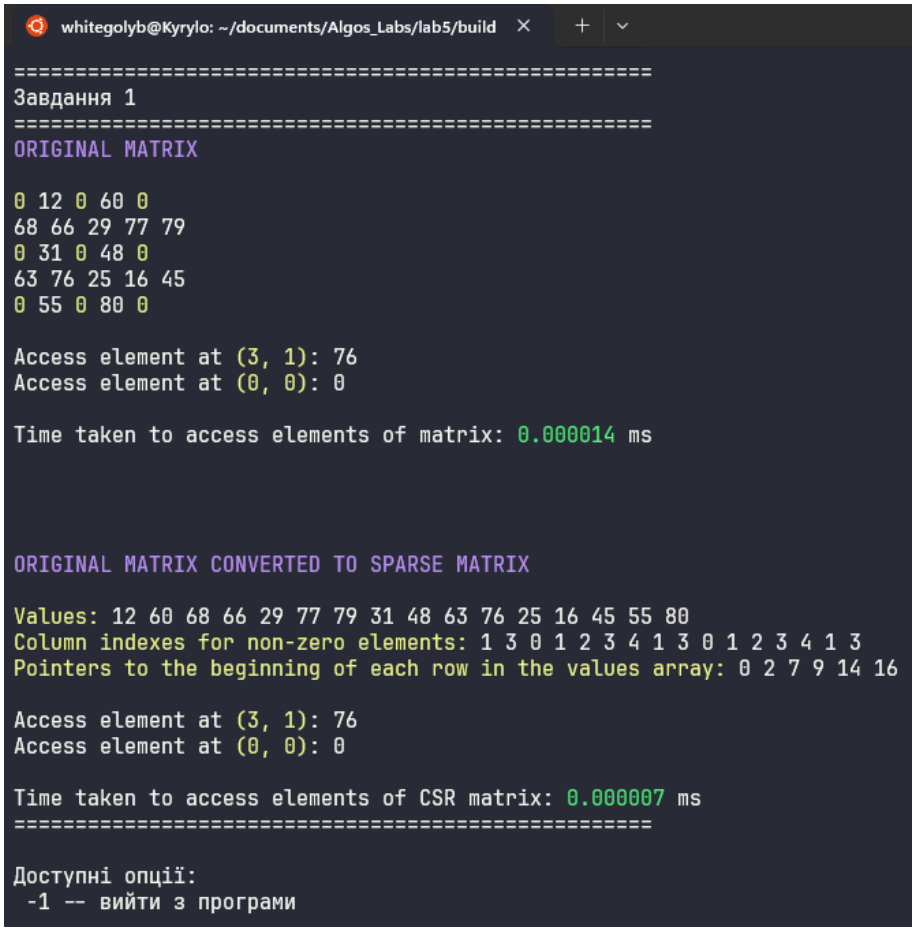

Рис. 1. Результат роботи програми

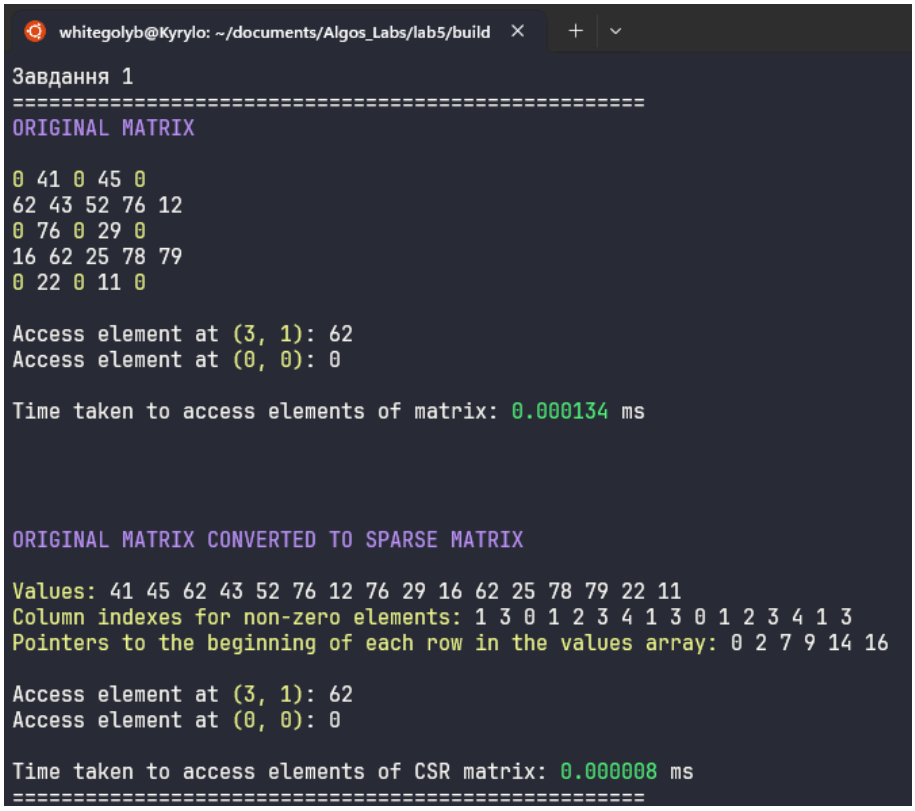

Рис. 2. Результат роботи програми

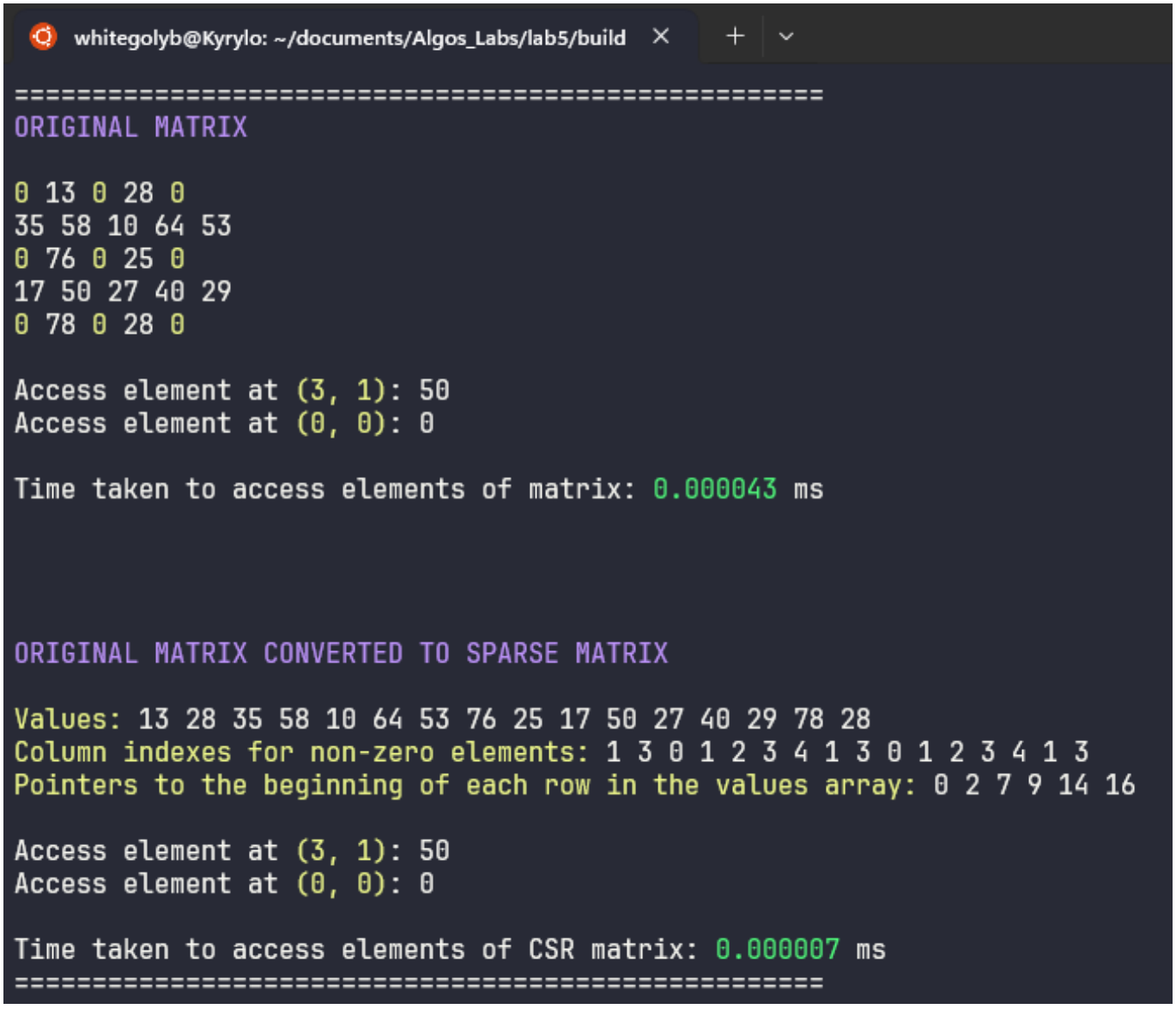

Рис. 3. Результат роботи програми

3 Висновки

Під час виконання лабораторної роботи, я розібрався з розрідженним представленням матриці та розібрався як її реалізувати для свого завдання.

Проаналізувавши отримані результати після завершення роботи програми, можна сказати що доступ до елементів звичайної матриці проходить довше ніж до розрідженної, але при певних умовах це може змінитися, бо доступ до елементів звичайної матриці буде швидкішим при більш великих матрицях, бо до елементів можна звертатися через арифметику вказівників, бо матриця зберігається неперервно у пам‘яті пристрою.

Таким чином можна зробити підсумок:

- Якщо матриця майже повністю заповнена ненульовими елементами, **звичайна матриця** краща для швидкого доступу.
- **Розріджена матриця** виграє, якщо більшість елементів дорівнюють нулю і потрібні операції з усіма ненульовими значеннями, а не точковий доступ до елементів.