Assignment 4; Arithmetic Operations on Sparse Matrices

Edris Lutfi

1 Introduction

This file includes the explanation, the trials, and the code for Assignment 4.

2 Explanations

2.1 Explanation for `Makefile.c`

The Makefile helps build the C program, and do all the compilations all in one. It compiles the source code into an executable code for all files that require this step. It also compiles with flags to help detect any errors, and it first compiles the source files into object files as well. After, it links the objects files to create the final program, so that when the user enters "make", it does all the compilation and linking together efficiently. There is also a clean section to remove temporary files.

2.2 Explanation for `main.c`

This file performs all the operations as it pulls from the function.c file. First, it starts by setting up timing to measure the cpu time, so that it can print the cpu time later on. It check if the correct number of arguments are provided, if not, it prints an error and exits. Then, it opens the file specified by the user and reads and prints it out (and converts it into a CSR format). Depending on the number of arguments, it prints the matrix from the file and cpu time, or does all that and does the operation asked from the user (the user needs to request to print it by entering "1" for their fifth argument.

2.3 Explanation for `functions.c`

This code has many functions and the structure to perform many tasks, so that the main.c file can pull from it. For example, it read the matrix from a file in the Martix Market format and it into a CSR format for proper storage. There are also functions to add, subtract, and multiply two matrices. And, there is a function to transpore a matrix as well. This file handles memory allocation, error checking, and ensure that the resulting matrices contain only the non-zero entries.

2.4 Explanation for `functions.h`

This file sets up the bluepring for handling sparse matrices in the CSR format, whilst defining hoe to read, print and perform operations (that are used in function.c and main.c).

3 Tasks given in Assignment

3.1 Reading Files

Reading the file for `n3c4-b1.mtx`

```
Trying to open file: n3c4-b1.mtx
Number of non-zeros: 30
Row Pointer: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
Column Index: 4 5 3 5 2 5 1 5 0 5 3 4 2 4 1 4 0 4 2 3 1 3 0 3 1 2 0 2 0 1
Values: 1.000000 -1.000000 1.000000 -1.000000 1.000000 -1.000000 1.000000
-1.000000 1.000000 -1.000000 1.000000 -1.000000 1.000000 -1.000000 1.000000
-1.000000 1.000000 -1.000000 1.000000 -1.000000 1.000000 -1.000000 1.000000
-1.000000 1.000000 -1.000000 1.000000 -1.000000 1.000000 -1.000000
CPU time: 0.000130 seconds
```

Reading the file for `rel3.mtx`

```
Trying to open file: rel3.mtx
Number of non-zeros: 18
Row Pointer: 0 3 6 9 12 15 18 18 18 18 18 18 18
Column Index: 1 3 4 1 3 4 1 3 4 1 3 4 1 3 4 1 3 4
Values: 1.000000 -2.000000 1.000000 1.000000 -2.000000 1.000000 1.000000
-2.000000 1.000000 1.000000 -2.000000 1.000000 1.000000 -2.000000 1.000000
1.000000 -2.000000 1.000000
CPU time: 0.000115 seconds
```

Reading the file for `jgl009.mtx`

```
Trying to open file: jgl009.mtx
Number of non-zeros: 50
Row Pointer: 0 5 8 12 14 21 23 24 31 33
Column Index: 0 3 6 1 8 8 2 6 1 3 6 3 6 3 0 2 3 4 6 3 5 6 0 7 0 2 3 4 5 8 8 6
2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Values: 2.000000 1.000000 1.000000 2.000000 2.000000 2.000000 3.000000
3.000000 8.000000 3.000000 3.000000 4.000000 4.000000 5.000000 6.000000
```

```
6.000000 6.000000 6.000000 5.000000 6.000000 6.000000 6.000000 7.000000
7.000000 9.000000 9.000000 9.000000 9.000000 9.000000 8.000000 9.000000
8.000000 9.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
CPU time: 0.000206 seconds
```

3.2 Comparing CPU time with implementation in C vs Python

Table 1: Matrix multiplication (A × B) with CSR format

A × B	**A (dim and nnz)**	**B (dim and nnz)**	**CPU time (se**
`n3c4-b1` × `Trec6`	15×6 30	6x15 40	0.00134
`2cubes_sphere` x `rm_101492rows`	101492x101492 874378	101492x1000 1014920	6.954429
`tmt_sym` x `rm_726713rows`	726713x726713 2903837	726713x2000 14534260	35.356087
`StocF-1465` × `rm_1465137rows`	1465137x1465137 11235263	1465137x1000 1465137	15.802387

CPU time in Python for ...
n3c4-b1 x Trec6 = 0.0009034709999999989 seconds
2cubes_sphere x rm_101492rows = 0.4381497300000001 seconds.
tmt_sym x rm_rm_726713rows= Takes too long that kernel keeps on dying in my environment
StocF-1465 x rm_1465137rows = Takes too long that kernel keeps on dying in my environment

Although the cpu time for the first two, was faster when python had completed it, i believe there is bias in it, as in C, the program was printing much more and doing much more when it was giving its outputs (results for matrix A, results for matrix B, and results for the new matrix). Also, the code chunk that python had done, had included very few lines of code, and only that chunk was run. In addition, the last two multiplications, python on Jupyter notebook was not able to do as it took so long that the kernel would keep on shutting down. Therefore, I think the implementation in C should be more efficient and faster.

4 Appendix

In this section I will show you all the files for my code

`Makefile`

```
# Variables for compiler and flags
CC = gcc
CFLAGS = -Wall -O2



# Targets
all: main

# Building the final executable
main: functions.o main.o
 $(CC) $(CFLAGS) -o main functions.o main.o

# Compile functions.c
functions.o: functions.c functions.h
 $(CC) $(CFLAGS) -c functions.c -o functions.o

# Compile main.c
main.o: main.c functions.h
 $(CC) $(CFLAGS) -c main.c -o main.o

# Clean rule to remove object files and executable
clean:
 rm -f *.o main
```

`main.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "functions.h"
#include <string.h>
#include <time.h>

int main(int argc, char *argv[])
{
    // Start the clock before any major computation
    clock_t start_time, end_time;
    double cpu_time_used;
    start_time = clock();

    if (argc < 2 || argc > 5 || argc == 3)
    {
        fprintf(stderr, "Please use the correct format\n");
        return 1;
    }

    const char *filename = argv[1];
    printf("Trying to open file: %s\n", filename);

    CSRMatrix A;
    ReadMMtoCSR(filename, &A);

    if (argc == 2)
    {
        print_CSR_Matrix(&A);
        end_time = clock();
        cpu_time_used = ((double)(end_time - start_time)) /
           CLOCKS_PER_SEC;
        printf("CPU time: %f seconds\n", cpu_time_used);
        freeCSR(&A);
        return 0;
    }
    else if (argc == 4 && (strcmp(argv[2], "transpose") == 0))
    {
        CSRMatrix T = transposeCSR(&A);
        if (atoi(argv[3]) == 1)
        {
            printf("Matrix from %s\n", filename);
            print_CSR_Matrix(&A);
            printf("\n");
            printf("Transpose of matrix from %s\n", filename);
            print_CSR_Matrix(&T);
            printf("\n");
        }
        freeCSR(&A);
        freeCSR(&T);
        end_time = clock();
        cpu_time_used = ((double)(end_time - start_time)) /
           CLOCKS_PER_SEC;
        printf("CPU time: %f seconds\n", cpu_time_used);
        return 0;
    }

    const char *filename_2 = argv[2];
    CSRMatrix B;
    ReadMMtoCSR(filename_2, &B);
    CSRMatrix C;
    const char *calc = argv[3];

    if (argc == 5)
    {
        if (strcmp(calc, "addition") == 0)
        {
            C = addCSR(&A, &B);
        }
        else if (strcmp(calc, "subtract") == 0)
        {
            C = subtractCSR(&A, &B);
        }
        else if (strcmp(calc, "multiply") == 0)
        {
            C = multiplyCSR(&A, &B);
        }
        else
        {
            fprintf(stderr, "Please use one of the following when
               calculating: addition, subtract, multiply or
               transpose. \n");
            freeCSR(&A);
            freeCSR(&B);
            end_time = clock();
            cpu_time_used = ((double)(end_time - start_time)) /
               CLOCKS_PER_SEC;
            printf("CPU time: %f seconds\n", cpu_time_used);
            return 1;
        }
    }

    if (atoi(argv[4]) == 1)
    {
        printf("Matrix A:\n");
        print_CSR_Matrix(&A);
        printf("\n");
        printf("Matrix B:\n");
        print_CSR_Matrix(&B);
        printf("\n");
        printf("Resultant Matrix C:\n");
        print_CSR_Matrix(&C);
        printf("\n");
    }

    freeCSR(&A);
    freeCSR(&B);
    freeCSR(&C);

    end_time = clock();
    cpu_time_used = ((double)(end_time - start_time)) /
       CLOCKS_PER_SEC;
    printf("CPU time: %f seconds\n", cpu_time_used);

    return 0;
}
```

functions.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h> // ADDED FOR memcpy
#include "functions.h"
#define MAX_LINE_LENGTH 1024 // Define maximum line length for
   reading

void ReadMMtoCSR(const char *filename, CSRMatrix *matrix)
```

```
{
    FILE *file = fopen(filename, "r");
    if (file == NULL)
    {
        fprintf(stderr, "Failed to open file %s\n", filename); //
           handling input error
        return;
    }

    char line[1500];

    while (fgets(line, sizeof(line), file))
    {
        if (line[0] != '%')
            break; // skipping lines that start with % (comments)
    }

    sscanf(line, "%d %d %d", &matrix->num_rows, &matrix->num_cols,
       &matrix->num_non_zeros); // reading matrix dimensions and #
       of nonzero elements

    matrix->csr_data = (double *)malloc(matrix->num_non_zeros *
       sizeof(double));
    matrix->col_ind = (int *)malloc(matrix->num_non_zeros * sizeof(
       int));
    matrix->row_ptr = (int *)calloc(matrix->num_rows + 1, sizeof(
       int));

    if (matrix->csr_data == NULL || matrix->col_ind == NULL ||
       matrix->row_ptr == NULL) // incase memory allocation was
       failed
    {
        fprintf(stderr, "Failed to allocate memory.\n");
        fclose(file);
        return;
    }

    for (int i = 0; i <= matrix->num_rows; i++)
    {
        matrix->row_ptr[i] = 0; // initializing row_ptr
    }

    int row, col;
    double matrix_value; // initializing row, col, and value
    // read the data entries and put numbers in row_ptr
    while (fscanf(file, "%d %d %lf", &row, &col, &matrix_value) ==
       3)
    {
        row -= 1; // Convert to 0 based index
        matrix->row_ptr[row + 1] += 1;
    }

    // Accumulate the row pointers
    for (int i = 1; i <= matrix->num_rows; i++)
    {
        matrix->row_ptr[i] += matrix->row_ptr[i - 1];
    }

    fseek(file, 0, SEEK_SET); // going back to beginning of file

    while (fgets(line, sizeof(line), file))
    {
        if (line[0] != '%')
            break;
    }

    int *temp_row_ptr = (int *)malloc((matrix->num_rows + 1) *
       sizeof(int)); // allocating memory to store temp row
       pointers for matrix
    if (temp_row_ptr == NULL)
    {
        fprintf(stderr, "Failed to allocate memory.\n");
        fclose(file);
        return;
    }

    memcpy(temp_row_ptr, matrix->row_ptr, (matrix->num_rows + 1) *
       sizeof(int)); // copy memory from row_ptr to temp_row_ptr
```

```c
    // reading data and assigning values for csr_data and col_ind
    while (fscanf(file, "%d %d %lf", &row, &col, &matrix_value) ==
       3)
    {
        row -= 1;
        col -= 1; // Convert both rows and cols to 0 based index
        int index = temp_row_ptr[row]++;
        matrix->csr_data[index] = matrix_value;
        matrix->col_ind[index] = col;
    }

    free(temp_row_ptr);
    fclose(file);
}

void freeCSR(CSRMatrix *matrix)
{
    free(matrix->csr_data);
    free(matrix->col_ind);
    free(matrix->row_ptr);
}

void print_CSR_Matrix(const CSRMatrix *matrix)
{
    // Print the matrix details only once
    printf("Number of non-zeros: %d\n", matrix->num_non_zeros);
    printf("Row Pointer: ");
    for (int i = 0; i <= matrix->num_rows; i++)
    {
        printf("%d ", matrix->row_ptr[i]);
    }
    printf("\nColumn Index: ");
    for (int i = 0; i < matrix->num_non_zeros; i++)
    {
        printf("%d ", matrix->col_ind[i]);
    }
    printf("\nValues: ");
    for (int i = 0; i < matrix->num_non_zeros; i++)
    {
        printf("%.6f ", matrix->csr_data[i]);
    }
    printf("\n");
}

// Function to add two sparse matrices in CSR format
CSRMatrix addCSR(const CSRMatrix *A, const CSRMatrix *B)
{
    if (A->num_rows != B->num_rows || A->num_cols != B->num_cols)
    {
        fprintf(stderr, "Error: Matrix dimensions do not match.
           Make sure numbers of rows and columns match.\n");
        exit(EXIT_FAILURE);
    }

    CSRMatrix C;              // initialize C
    C.num_rows = A->num_rows; // Sum of matrices will have same
       dimensions as A and B
    C.num_cols = A->num_cols;

    // max num of nonzeros in matrix C will be the sum of nonzeroes
        in A and nonzeroes in B
    int max_non_zeros = A->num_non_zeros + B->num_non_zeros;

    C.row_ptr = (int *)calloc(C.num_rows + 1, sizeof(int));
    C.csr_data = (double *)malloc(max_non_zeros * sizeof(double));
    C.col_ind = (int *)malloc(max_non_zeros * sizeof(int));
    int *col_tracker = (int *)malloc(C.num_cols * sizeof(int));

    if (col_tracker == NULL || C.row_ptr == NULL || C.csr_data ==
       NULL || C.col_ind == NULL)
    {
        fprintf(stderr, "Failed to allocate memory.\n");
        free(C.csr_data);
        free(C.col_ind);
        free(C.row_ptr);
        free(col_tracker);
        exit(EXIT_FAILURE);
    }

    memset(col_tracker, -1, C.num_cols * sizeof(int)); // filling
       blocks of memory with -1
    // of memory that are being pointed at by the column_marker
       pointer. We fill in C.num_cols * sizeof(int) bytes of memory

    // using column_marker to more efficiently add

    int counter = 0;
    for (int i = 0; i < C.num_rows; i++) // iterating over rows
    {
        C.row_ptr[i] = counter;
        // iterating over all nonzero entries in the i-th row of A
        for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1]; j++)
        {
            int col = A->col_ind[j]; // copying all non-zero
               entries from A into C
            C.csr_data[counter] = A->csr_data[j];
            C.col_ind[counter] = col;
            col_tracker[col] = counter;
            // chaging certain entries from -1 to the value of the
               counter for any column that the value was copied
               from A to C

            counter += 1;
        }

        // iterating over nonzero entries in the i-th row of B
        for (int j = B->row_ptr[i]; j < B->row_ptr[i + 1]; j++)
        {
            int col = B->col_ind[j];
            // Any col where the column_marker is not -1 is a col
               where the value was copied from A to C
            if (col_tracker[col] != -1)
            {
                C.csr_data[col_tracker[col]] += B->csr_data[j];
            }
            else // creating new entries in C
            {
                C.csr_data[counter] = B->csr_data[j];
                C.col_ind[counter] = col;
                col_tracker[col] = counter;
                counter += 1;
            }
        }

        for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1]; j++)
        {
            int col = A->col_ind[j];
            col_tracker[col] = -1;
        }

        for (int j = B->row_ptr[i]; j < B->row_ptr[i + 1]; j++)
        {
            int col = B->col_ind[j];
            col_tracker[col] = -1;
        }
    }

    C.row_ptr[C.num_rows] = counter;

    // Allocating new arrays to filter out any added zeros
    double *correct_csr_data = (double *)malloc(counter * sizeof(
       double));
    int *correct_col_ind = (int *)malloc(counter * sizeof(int));
    int *correct_row_ptr = (int *)calloc(C.num_rows + 1, sizeof(int
       ));
    if (correct_csr_data == NULL || correct_col_ind == NULL ||
       correct_row_ptr == NULL)
    {
        fprintf(stderr, "Failed to allocate memory.\n");
        free(C.row_ptr);
        free(C.csr_data);
        free(C.col_ind);
        free(col_tracker);
        exit(EXIT_FAILURE);
    }
    int correct_num_values = 0;
    for (int i = 0; i < C.num_rows; i++)
    {
        correct_row_ptr[i] = correct_num_values;

        for (int j = C.row_ptr[i]; j < C.row_ptr[i + 1]; j++)
        {
            if (C.csr_data[j] != 0)
            {
                correct_csr_data[correct_num_values] = C.csr_data[j
                    ];
                correct_col_ind[correct_num_values] = C.col_ind[j];
                correct_num_values += 1;
            }
        }
    }

    correct_row_ptr[C.num_rows] = correct_num_values;

    free(C.csr_data);
    free(C.col_ind);
    free(C.row_ptr);
    free(col_tracker);

    C.csr_data = correct_csr_data;
    C.col_ind = correct_col_ind;
    C.row_ptr = correct_row_ptr;
    C.num_non_zeros = correct_num_values;

    return C;
}


// Subtract two CSR matrices
CSRMatrix subtractCSR(const CSRMatrix *A, const CSRMatrix *B)
{
    // Check if matrices have the same size
    if (A->num_rows != B->num_rows || A->num_cols != B->num_cols)
    {
        fprintf(stderr, "Error: Matrix dimensions do not match.\n")
           ;
        exit(EXIT_FAILURE);
    }

    CSRMatrix C;
    C.num_rows = A->num_rows;
    C.num_cols = A->num_cols;

    // Allocate memory
    C.row_ptr = (int *)calloc(C.num_rows + 1, sizeof(int));
                                  // Row pointers
    C.csr_data = (double *)malloc((A->num_non_zeros + B->
       num_non_zeros) * sizeof(double)); // Non-zero values
    C.col_ind = (int *)malloc((A->num_non_zeros + B->num_non_zeros)
        * sizeof(int));         // Column indices
    int *col_mark = (int *)malloc(C.num_cols * sizeof(int));
                                 // Track column indices

    if (!C.row_ptr || !C.csr_data || !C.col_ind || !col_mark)
    {
        fprintf(stderr, "Memory allocation failed.\n");
        free(C.row_ptr);
        free(C.csr_data);
        free(C.col_ind);
        free(col_mark);
        exit(EXIT_FAILURE);
    }
    memset(col_mark, -1, C.num_cols * sizeof(int)); // Initialize
       column tracker

    int count = 0; // Counter for non-zero entries
    for (int row = 0; row < C.num_rows; row++)
    {
        C.row_ptr[row] = count; // Start of row

        // Process matrix A
        for (int index = A->row_ptr[row]; index < A->row_ptr[row +
           1]; index++)
        {
            int col = A->col_ind[index];
            C.csr_data[count] = A->csr_data[index];
            C.col_ind[count] = col;
            col_mark[col] = count; // Update tracker
            count++;
        }

        // Process matrix B
        for (int index = B->row_ptr[row]; index < B->row_ptr[row +
           1]; index++)
        {
            int col = B->col_ind[index];
            if (col_mark[col] != -1)
            {
                C.csr_data[col_mark[col]] -= B->csr_data[index];
            }
            else
            {
                C.csr_data[count] = -B->csr_data[index];
                C.col_ind[count] = col;
                col_mark[col] = count; // Update tracker
                count++;
            }
        }

        // Reset tracker for next row
        for (int index = A->row_ptr[row]; index < A->row_ptr[row +
           1]; index++)
        {
            col_mark[A->col_ind[index]] = -1;
        }
        for (int index = B->row_ptr[row]; index < B->row_ptr[row +
           1]; index++)
        {
            col_mark[B->col_ind[index]] = -1;
        }
    }
```

```
    C.row_ptr[C.num_rows] = count; // End of last row

    // Remove zeros and update row pointers
    double *final_data = (double *)malloc(count * sizeof(double));
    int *final_indices = (int *)malloc(count * sizeof(int));
    int *final_ptr = (int *)calloc(C.num_rows + 1, sizeof(int));

    if (!final_data || !final_indices || !final_ptr)
    {
        fprintf(stderr, "Memory allocation failed.\n");
        free(C.row_ptr);
        free(C.csr_data);
        free(C.col_ind);
        free(col_mark);
        exit(EXIT_FAILURE);
    }

    int final_counter = 0;
    for (int row = 0; row < C.num_rows; row++)
    {
        final_ptr[row] = final_counter;
        for (int index = C.row_ptr[row]; index < C.row_ptr[row +
           1]; index++)
        {
            if (C.csr_data[index] != 0)
            {
                final_data[final_counter] = C.csr_data[index];
                final_indices[final_counter] = C.col_ind[index];
                final_counter++;
            }
        }
    }
    final_ptr[C.num_rows] = final_counter;

    // Free temporary memory
    free(C.csr_data);
    free(C.col_ind);
    free(C.row_ptr);
    free(col_mark);
    C.csr_data = final_data;
    C.col_ind = final_indices;
    C.row_ptr = final_ptr;
    C.num_non_zeros = final_counter;

    return C;
}


CSRMatrix multiplyCSR(const CSRMatrix *A, const CSRMatrix *B)
{
    // Check if matrices can be multiplied
    if (A->num_cols != B->num_rows)
    {
        fprintf(stderr, "Error: Matrices dimensions do not match
           for multiplication.\n");
        exit(1);
    }

    CSRMatrix result; // Result matrix
    result.num_rows = A->num_rows;
    result.num_cols = B->num_cols; // Result matrix dimensions
    result.row_ptr = (int *)calloc(result.num_rows + 1, sizeof(int)
       );
    if (result.row_ptr == NULL)
    {
        fprintf(stderr, "Failed to allocate memory.\n");
        exit(1);
    }

    int maxEntries = 160000000; // Estimated max non-zero entries
    result.csr_data = (double *)malloc(maxEntries * sizeof(double))
       ;
    if (result.csr_data == NULL)
    {
        fprintf(stderr, "Failed to allocate memory for data.\n");
        free(result.row_ptr);
        exit(1);
    }

    result.col_ind = (int *)malloc(maxEntries * sizeof(int));
    if (result.col_ind == NULL)
    {
        fprintf(stderr, "Failed to allocate memory for indices.\n")
           ;
        free(result.row_ptr);
        free(result.csr_data);
        exit(1);
    }

    int *columnFlags = (int *)malloc(result.num_cols * sizeof(int))
       ;
    if (columnFlags == NULL)
    {
        fprintf(stderr, "Failed to allocate memory for flags.\n");
        free(result.row_ptr);
        free(result.csr_data);
        free(result.col_ind);
        exit(1);
    }
    memset(columnFlags, -1, result.num_cols * sizeof(int)); //
       Initialize flags

    int entryCount = 0;
    for (int i = 0; i < A->num_rows; i++) // Process each row of A
    {
        result.row_ptr[i] = entryCount;

        for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1]; j++) //
           Process non-zeros in row of A
        {
            int a_col = A->col_ind[j];
            double a_val = A->csr_data[j];

            for (int k = B->row_ptr[a_col]; k < B->row_ptr[a_col +
               1]; k++) // Process non-zeros in column of B
            {
                int b_col = B->col_ind[k];
                double b_val = B->csr_data[k];
                if (columnFlags[b_col] < result.row_ptr[i]) //
                   Check if column already added
                {
                    columnFlags[b_col] = entryCount;
                    result.col_ind[entryCount] = b_col;
                    result.csr_data[entryCount] = a_val * b_val;
                    entryCount++;
                }
                else
                {
                    result.csr_data[columnFlags[b_col]] += a_val *
                       b_val; // Update existing entry
                }
            }
        }
        for (int jj = A->row_ptr[i]; jj < A->row_ptr[i + 1]; jj++)
        {
            int a_col = A->col_ind[jj];

            for (int kk = B->row_ptr[a_col]; kk < B->row_ptr[a_col
               + 1]; kk++)
            {
                int b_col = B->col_ind[kk];
                columnFlags[b_col] = -1; // Reset column flag
            }
        }
    }

    result.row_ptr[result.num_rows] = entryCount; // End of row
       pointers

    // Filter out zero entries
    double *cleanedData = (double *)malloc(entryCount * sizeof(
       double));
    int *cleanedColInd = (int *)malloc(entryCount * sizeof(int));
    int *cleanedRowPtr = (int *)calloc(result.num_rows + 1, sizeof(
       int));
```

```
if (cleanedData == NULL || cleanedColInd == NULL ||
   cleanedRowPtr == NULL)
{
    fprintf(stderr, "Failed to allocate memory for filtered
       data.\n");
    free(result.row_ptr);
    free(result.csr_data);
    free(result.col_ind);
    free(columnFlags);
    exit(1);
}

int nonZeroCount = 0;
for (int i = 0; i < result.num_rows; i++)
{
    cleanedRowPtr[i] = nonZeroCount;
    for (int j = result.row_ptr[i]; j < result.row_ptr[i + 1];
       j++)
    {
        if (result.csr_data[j] != 0)
        {
            cleanedData[nonZeroCount] = result.csr_data[j];
            cleanedColInd[nonZeroCount] = result.col_ind[j];
            nonZeroCount++;
        }
    }
}

cleanedRowPtr[result.num_rows] = nonZeroCount;
free(result.csr_data);
free(result.col_ind);
free(result.row_ptr);

result.csr_data = cleanedData;
result.col_ind = cleanedColInd;
result.row_ptr = cleanedRowPtr;
result.num_non_zeros = nonZeroCount;
```

```
    free(columnFlags);

    return result;
}
// Function to transpose a CSR matrix
CSRMatrix transposeCSR(const CSRMatrix *A)
{
    CSRMatrix T;                               // Create a new matrix for
       the transpose
    T.num_rows = A->num_cols;                  // Number of rows in T is
       the number of columns in A
    T.num_cols = A->num_rows;                  // Number of columns in T
       is the number of rows in A
    T.num_non_zeros = A->num_non_zeros; // Number of non-zero
       elements remains the same

    // Allocate memory for row_ptr, csr_data, and col_ind
    T.row_ptr = (int *)calloc(T.num_rows + 1, sizeof(int));
    if (T.row_ptr == NULL)
    {
        fprintf(stderr, "Failed to allocate memory.\n");
        exit(EXIT_FAILURE);
    }

    T.csr_data = (double *)malloc(T.num_non_zeros * sizeof(double))
       ;
    if (T.csr_data == NULL)
    {
        fprintf(stderr, "Failed to allocate memory.\n");
        free(T.row_ptr);
        exit(EXIT_FAILURE);
    }

    T.col_ind = (int *)malloc(T.num_non_zeros * sizeof(int));
    if (T.col_ind == NULL)
    {
        fprintf(stderr, "Failed to allocate memory.\n");
        free(T.row_ptr);
        free(T.csr_data);
        exit(EXIT_FAILURE);
    }

    // Count the number of entries in each column of A
    for (int i = 0; i < A->num_non_zeros; i++)
    {
        T.row_ptr[A->col_ind[i] + 1]++;
    }

    // Calculate row_ptr values for T
    for (int i = 1; i <= T.num_rows; i++)
    {
        T.row_ptr[i] += T.row_ptr[i - 1];
    }

    // Track current position in each row of T
    int *position_tracker = (int *)malloc(T.num_rows * sizeof(int))
       ;
    if (position_tracker == NULL)
    {
        fprintf(stderr, "Failed to allocate memory.\n");
        free(T.row_ptr);
        free(T.csr_data);
        free(T.col_ind);
        exit(EXIT_FAILURE);
    }
    memcpy(position_tracker, T.row_ptr, T.num_rows * sizeof(int));
       // Copy row_ptr to position_tracker

    // Fill csr_data and col_ind for T
    for (int i = 0; i < A->num_rows; i++)
    {
        for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1]; j++)
        {
            int col = A->col_ind[j];
            int index_in_T = position_tracker[col];

            T.csr_data[index_in_T] = A->csr_data[j];
            T.col_ind[index_in_T] = i;
            position_tracker[col]++;
        }
    }

    free(position_tracker);
    return T;
}
```

functions.h

```
 #ifndef FUNCTIONS_H
#define FUNCTIONS_H

// ###########################################################
// Do not change this part
typedef struct {
    double *csr_data;   // Array of non-zero values
    int *col_ind;       // Array of column indices
    int *row_ptr;       // Array of row pointers
    int num_non_zeros;  // Number of non-zero elements
    int num_rows;       // Number of rows in matrix
    int num_cols;       // Number of columns in matrix
} CSRMatrix;


void ReadMMtoCSR(const char *filename, CSRMatrix *matrix);
void print_CSR_Matrix(const CSRMatrix *matrix);
/* <Here you can add the declaration of functions you need.>
<The actual implementation must be in functions.c>
Here what "potentially" you need:
1. "addition" function receiving const CSRMatrix A, const CSRMatrix
    B, and computing C=A+B
2. "subtraction" function receiving const CSRMatrix A, const
   CSRMatrix B, and computing C=A-B
3. "multiplication" function receiving const CSRMatrix A, const
   CSRMatrix B, and computing C=A*B
4. "transpose" function receiving const CSRMatrix A, computing the
   transpose of A (C=A^T)
```

```
It is up to you how to save and return the product of each function
   , matrix C
*/

CSRMatrix addCSR(const CSRMatrix *A, const CSRMatrix *B);
// CSRMatrix addCSR(const CSRMatrix* A, const CSRMatrix* B);
CSRMatrix subtractCSR(const CSRMatrix* A, const CSRMatrix* B);
CSRMatrix multiplyCSR(const CSRMatrix* A, const CSRMatrix* B);
CSRMatrix transposeCSR(const CSRMatrix* A);
void freeCSR(CSRMatrix *matrix);

#endif
```