Programming Assignment #1 - Matrix Operations

Data Structure SWE2015-41, School of Software, SKKU
Susung Park (2014311254), School of Mechanical Engineering, SKKU

I. I'm busy. Just tell me how to use this.

OK, let's start from the conclusion.

```
1. Unzip the package and move into `ds_hw01` directory.
2. Just put your input file in `./io` folder, and execute the program
`./main`.
3. You'll get your output inside the same directory.
```

If you want to use a file other than `./io/hw1_input.txt`, give the name of file after the program name in the console. Even if custom file name is given, output will be created in the same directory. If input file does not exist, program raises error. Check the input file.

This code is written and verified on Linux OS (Ubuntu 18.04 & GCC 7.3.0). Please use a Linux machine to evaluate this program.

All includings are header-only. Use the following simple command to compile the program.
`gcc main.c -o main`
Optimization options might help improve execution speed, but it is not tested.

ONE THING TO BE CAUTIOUS: USE ` FOR TRANSPOSITION SYMBOL. OTHERWISE THE PROGRAM WON'T RECOGNIZE!

II. Objectives and Considerations

In this assignment, basic arithmetic and transposition operations of matrices are evaluated. While writing the codes, the followings are considered, with the priority follwing the numbers.

```
1. Reduction of time complexity
2. Reduction of space complexity
3. Capsulization & interoperability between different types of matrices
4. Modulization through utilization of functions
```

To do any arithmetic operation between two matrices, a function should be defined, such as `matrix_add(sparse_matrix a, dense_matrix b)`. Here, the data type of the parameters will be differ by the type of input. However, unfortunately, C language does not support function overloading, so we will have four separate functions with same function and different parameters, which will make the code cloggy. It would be much better to unify the data type of function parameters.

To resolve this problem, two unique `struct`s are defined; `SparseMatElem` and `SquareMatrix`. `SquareMatrix` is a C struct that contains **dimension**, **number of nonzeros**, and **pointer to matrix**, and `SparseMatElem` is a C struct that contains **row and column indices**, and the **value** for there.

One thing to point out is that `SquareMatrix` instance includes pointers for both sparse matrix and dense matrix. In some perspect this is merely a data overhead. However, considering the interoperability between `SquareMatrix` instances, this was inevitable. One more excuse for it is that, by setting one of these `NULL`, it can be used as a flag variable that indicates whether the `SquareMatrix` object is sparse or dense. This way, both sparse matrix and dense matrix can be capsulized as `SquareMatrix`.

Now, we consider about the operations between matrices.

III. Operations

Operations are given in a string form. Therefore, this formula should be parsed into set of operations. Fortunately, there is no condition to follow the priority rules, so serial reading is enough. Therefore, it would be efficient to create a separate matrix for the results so far, and do all the operations upcoming to that matrix. This will help both the workflow and data integrity of two originally-given matrices. Here, the '`+`' and '`*`' operators involve two terms, and '`' operator involves only one term.

To distinguish this diffefence, the variable `nOpTrigger` is used. When the formula parser met a variable, `+` operator, or `*` operator, it adds 1 to `nOpTrigger`. When the formula parser met the '`' operator, it adds 2 to `nOpTrigger`. When `nOpTrigger == 2`, the parser calls the appropricate function and triggers the calculation. This way, we can apply different action by type of operator.

Since there is no priority on operators, a 'buffer' for storing the results so far will be needed. Also, we would need a pointer that points to the upcoming operand. The matrix for 'result' will be called `matResult` and for 'upcoming operand' will be called `matNext` from now on.

1. Addition

Addition between two dense matrices is either easy; since the length of two matrices are equal, simply iterating two matrices and add the values is enough. Since both matrices are already confirmed that dense matrix representation is more efficient, the output could be represented in dense matrix form.

If `matResult` is dense and `matNext` is sparse, we simply can iterate through indices and values of `matNext`, and copy the value into proper locations.

Else if `matResult` is sparse and `matNext` is dense, we can convert `matResult` into dense matrix, and add the two dense matrices.

Dealing with addition between two sparse matrices is a real problem. It is somewhat similar to a *union* in set theories. We should merge two matrices, while finding and adding elements that have identical indices. Fortunately, thanks to the nature of the process creating sparce matrices, **both matrices are already sorted in column-major order**. So, we can simply consider this problem as merging two dictionaries.

When we try to merge the words in two dictionaries into one in lexiographical order (which actually seems quite silly), we might open the first page of both dictionaries and start comparing which word will come prior.

In the exactly same method, we can merge two sparse matrices.

2. Multiplication

For multiplication, a zero dense matrix was allocated at all cases. Maybe, for sparse * sparse case, comparing the indices and allocating new sparse matrix could have reduced space complexity. However, since comparing the indices is a task that costs `O(mn)`, running this operation just once is enough. Therefore, I just unified to allocate a dense matrix for all four cases.

3. Transposition

Transposition was realized exactly the same way discussed in class. Therefore, this has the time complexity of `O(n)` for sparse matrices, `O(n^2)` for dense matrices.

IV. Time & Space Compexity

Number of inputs can be considered in two ways, depending on the perspective. When the input is processed from the text file to matrix data, we will consider the 'input' as `n`, while actual number of inputs is `n^2`. When the input processing is finished and we deal with sparse matrices, actual number of data is `m`. Both perspectives will be discussed.

1. Parsing Inputs

As mentioned above, actual number of inputs is `n^2`. Since this program reads the file serially, operation time is proportional to the actual number of inputs. Therefore, for parsing inputs, we have time complexity `O(n^2)`.

2. Addition

- For *Dense* + *Dense* case, we simply sweep the whole of both matrices and add up each element. Therefore, for adding two dense matrices, we have time complexity `O(n^2)`.
- For *Sparse* + *Dense* case, we convert the sparse matrix into dense one and then add two dense matrices. Therefore, this case has `O(n^2)` time complexity just as *Dense* + *Dense* case.
- For *Dense* + *Sparse* case, we simply find the values in the proper index and add the value of sparse matrix. Therefore, this case has `O(m)` time complexity.
- For *Sparse* + *Sparse* case, we compare two sparse matrices and 'combine' them in lexiographical order. This is a serial task, so the time complexity is `O(m)`.

3. Multiplication

- For *Dense* * *Dense* case, we use the naive matrix multiplication algorithm, which costs `O(n^3)`.
- For *Sparse* * *Dense* case, we perform matrix multiplication in column-order. This leads to `n` operations for each `m` sparse matrix element. Therefore, the time complexity is `O(mn)`.
- For *Dense* * *Sparse* case, we perform matrix multiplication in row-order. This leads to `n` operations for each `m` sparse matrix element. Therefore, the time complexity is `O(mn)`.
- For *Sparse* * *Sparse* case, we iterate every combination of elements from both matrices, compare indices, and multiply if the indices are proper. Therefore, the time complexity is `O(m^2)`.

Here, one thing to consider is, time compexity does not follow space compexity. For a 1000 x 1000 matrix with 1E5 nonzeros, this is a sparse matrix but calculating as dense matrices is more efficient. The naive algotirhm has 1E9 computations, and the sparse matrix multiplication has time complexity of `O(t^2)`. Therefore, when nonzero elements exceed 31623 (theoretically), sparse calculation becomes more heavy.

4. Transposition

- For sparse matrices, method discussed in class, which has time complexity of `O(n)`, is applied.
- For dense matrices, swapping operation happens `n^2 - n` times, which corresponds to time complexity `O(n^2)`.

5. Overall Performance

For the worst case, which both matrices A and B are dense matrices and there is any multiplication operator, time complexity records `O(n^3)`. For the best case, which both matrices A and B are sparse and there are only addition or transposition, time complexity records `O(n)`.

In fact, as `n` goes larger, time-portion of I/O task rapidly increases. For a test case with 1000x1000 matrices, I/O operation took approximately 0.2 seconds, which is more than a half of total operation time. However, since I/O operation is inevitable and there is no measure to improve it, no need to consider I/O task.

Some benckmarks with 1000 x 1000 matrices are shown below.

- Sparse + Sparse: 136 ms
- Sparse * Sparse: 331 ms
- Sparse + Dense: 269 ms
- Sparse * Dense: 320 ms
- Dense + Sparse: 270 ms
- Dense * Sparse: 278 ms
- Dense + Dense: 337 ms
- Dense * Dense: 3870 ms

This benchmark is executed on my PC. The specifications are written below. System performance of the PC is relatively high, so more time might cost on computers with lower performance.

V. System Information

This code was written across two devices; laptop and PC. The hardware/software specs are shown below.

- Laptop: Intel Core(R) i7-6700HQ, 16G RAM, Ubuntu 18.04, GCC 7.3.0
- Desktop: AMD Ryzen 7 2700X, 32G RAM, Ubuntu 18.04, GCC 7.3.0