University of St Andrews
School of Computer Science
Junior Honours

CS3052
Computational Complexity

Practical 1: Analysis of Matrix Multiplication

March 2018

150003431

Contents

1
Introduction

1.1 Overview

This practical covered the theory of asymptotic complexity and the assessment of the asymptotic behaviour of an algorithm derived from practical implementation. The given algorithm is analysed and improvements have been made by implementation of matrices and matrix multiplication algorithms. This report outlines my approach, research and analysis of the tasks.

1.2 Instructions

Code

The test suite is designed as a Maven project where source code and JUnit tests are located in relevant directories. All four test can be run on command line, inside "/practical1" where "pom.xml" file is located.

Data

All data used for analysis are located in "AnalysisData", which contains "Result" and "Plots" sub-directories. In general, data for testing will be randomly generated for every run of the test but new data sets can be created and written to a file using $DataTest.java$, if we would like to use same data set for multiple testings. Hence, "data" and "output" are empty directories for future test input and output storage.

Important Commands

Following commands can be run on command line (Terminal) in the lab machine:

- **mvn clean** Remove current compiled class and output files for a new build.
- **mvn clean test** The project will be rebuilt, and all tests will be run in the default settings. Note that running all four tests would take around an hour due to the size of inputs; hence running tests individually would be preferred.
- **mvn test -Dtest=${test_file_name}** Individually, the average time taken to run each test on a lab machine is the following:

  1. BasicMultiplierTest.java : 3 mins
  2. CompareMultipliers1Test.java : 5 mins
  3. CompareMultipliers2Test.java : 15 mins
  4. CompareMultipliers3Test.java : (random) 8 mins, (band) 4 mins, (triangular) 5 mins

  Explanations of the above tests will be in Task 2 and 4.
- **mvn test -Dtest=CompareMultipliers3Test -Dsparsity=${d} -DmatrixType=${i}**
  The command is used for advanced Testing (Task 4,5) where parameters are passed to define the characteristics of input matrices. In order to test on data sets that yet exist, create the data set first and run this command with corresponding parameter values.corresponding parameter values.
- **mvn test -Dtest=DataTest -DtestName=${testName} -Dsparsity=${d} -DmatrixType=${i}**
  The command is used for creating a new data set. -DtestName can take either "basic" or "comp" but default is set as "comp" if not provided: "basic" can be used to run *BasicMultiplier* and "comp" for any of *CompareMultiplier* Tests.
  -Dsparsity parameter controls the sparsity of a matrix and accepts a decimal of 2 decimal place within the range [0,1]. -DmatrixType determines the types of sparse matrix it will generate for testing. It accepts one of 0, 1 and 2. Please refer to Chapter 5 for further details.

2
Task 1

Analyse the Basic Algorithm for Matrix Multiplication for the worst-case time complexity. You may use the provided code in BasicMultiplier.java (or any alternative implementation you write or find on the web). This task is to give a theoretical worst-case guarantee, and will be assessed by the report.

The method multiply() in *BasicMultiplier.java* is analysed step by step as the following:

1. `int dim = a/getDim();`

   The integer assignment is a constant time operation, let this be c_1.

2. `IntMatrix result = new IntMatrix(dim);`

```
/**
 * Creates a square matrix with the given dimension.
 */
public IntMatrix(int dim) {
    this.values = new int[dim][dim];
}
```

   The constructor of an IntMatrix class instance creates a 2D array. Since Java automatically initialises every entry of the array with 0, this initialisation is n^2 operation.

3.
```
for(int i = 0; i < dim; i++) {
    for(int j = 0; j < dim; j++) {
        int sum = 0;
        for(int k = 0; k < dim; k++) {
            sum += a.get(i,k) * b.get(k,j);
        }
        result.set(i,j,sum);
    }
}
```

   This triple loop is where matrix multiplication happens. As we are interested in multiplying matrices of variable dimensions, the complexity of the algorithm can be formulated using the variable value *dim*. Let n be the value *dim*.

The most inner loop takes $c_2 n$ times where constant operations denoted by c_2 include GET operations of entry values and the subsequent arithmetic operations, taking n times. Additional operations such as integer initialisation and value assignment constituting the intermediate loop will take constant time of c_3. The outer double loop consists will loop over n^2 entries. Therefore, the combined time complexity of this triple loop by multiplying n^2 will be $(c_2 n + c_3)n^2$.

4.
```
public IntMatrix multiply(IntMatrix a, IntMatrix b) {
    int dim = a.getDim();
    IntMatrix result = new IntMatrix(dim);

    for(int i = 0; i < dim; i++) {
        for(int j = 0; j < dim; j++) {
            int sum = 0;
            for(int k = 0; k < dim; k++) {
                sum += a.get(i,k) * b.get(k,j);
            }
            result.set(i,j,sum);
        }
    }
return result;
}
```

Hence, the Basic Algorithm has an overall empirical time complexity:

$$\Theta(c_1 + c_3 n^2 + c_2 n^3)$$

Using a simplifying rule, the average case will be

$$\Theta(n^3)$$

. It is evident that the Basic Algorithm operates equally for any types of matrices: regardless of the positions or the number of zeros values, the algorithm will always traverse the whole $n \times n$ matrix. Hence, this concludes that the guaranteed worst-case complexity will be equal to its average case complexity since the dominant term will always be n^3.

3
Task 2

Analyse and estimate the average-case performance of the Basic Algorithm by generating random test inputs of various sizes and doing data analysis. You will have to generate harness code to generate random matrices, to find the time taken for the multiplication procedure, and then do some data analysis to find the behaviour as a function of input size. Submit your harness code, any scripts/spreadsheet input used for the data analysis, and the results of that data analysis. This task will be assessed by the report together with any supplementary material submitted.

3.1 Design and Testing

Testing is done at *BasicMultiplierTest* with parameterised inputs. Each run of this test suite will use data generated in *Data.java*, saved in "data" as *basic_ 0.75_ random_ input.csv*. It will record and write the average execution time of multiplication procedure to a CSV file. This output file will be named as *basic_ output.csv*.

Generation of Random Matrices

All the input data are randomly generated at the execution of the testing, while it is also possible to get pre-generated data set before any testing: the generation is done by running *DataTest.java*. The input file is read, by the method *getParams()* in *Utils.java*. The parameters are passed as an iterable list of lists of objects. For both cases, the generated matrices are passed according to the increasing value of the dimension. *getParams()* accepts multiple parameters for controlling data structure, a dimension, the number of matrices of equal dimension and a step size between two dimensions. In *getSparseMatrix()* method random matrices are generated for testing. The non zero entries are chosen using Random() and sparsity is set to 0.75 by default. One assumption made in the random matrices created is that they will only take integers values between 1 and 9 for non zero entries; this facilitates in removing this factor out of the complexity analysis.

Testing Strategy

At every iteration of parameters, the time taken for multiplication of two matrices will be recorded using *System.nanoTime*(). Only the time elapsed to run *BasicMultiplier.multipy()* will be recorded, which excludes the parameter preparation time.

A range of input size is chosen carefully by considering theoretical requirements and practical constraints:

- We are interested in the asymptotic behaviour of the algorithm as the dimension of an $n \times n$ matrix increases. Hence, to obtain an accurate estimation of the correlation of time and an input size, we would ideally prefer as many data points and a large value of dimension as possible.

- Practically, there exists constraints against implementing the desired range of input parameters. The test size should be manageable to be executed and similarly, such restriction in space implies that it is highly space-consuming and unrealistic to test on infinitely large input sizes. Therefore, there should a compromise reached to obtain a reliable result under the constraints.

- We are also interested in the average case, meaning the time taken for average sparse matrices to generate a product. Considering discrepancies which may exist in execution of different matrices of same dimension, the algorithm will be applied to multiple $n \times n$ matrices for every n. Thereby the computation time is summed to generate its average time elapsed for $n \times n$ matrix multiplication. This method will isolate n as a factor which should be considered in determining the effect of increasing input size on the algorithm execution time.

After several experiments, I have carefully chosen the range of dimension value n and some repetitions to test the Basic Algorithm; to discover both its general trend and asymptotic behaviour. The minimum is set to be 300 with a maximum value of n as 1000. The number of random matrices of a given dimension is set to be 20, with increasing dimension by a step size of 50. I have derived these values as a compromise of large values of dimension and maximum number of repetitions, which the program could accommodate. The output of the test is available at *basic_output.csv*.

3.2 Analysis

An analysis of test outputs is done by examining its graphical presentation. Two graphs are plotted for the average execution time against matrix dimension; time against n, and $log(t)$ against $log(n)$.

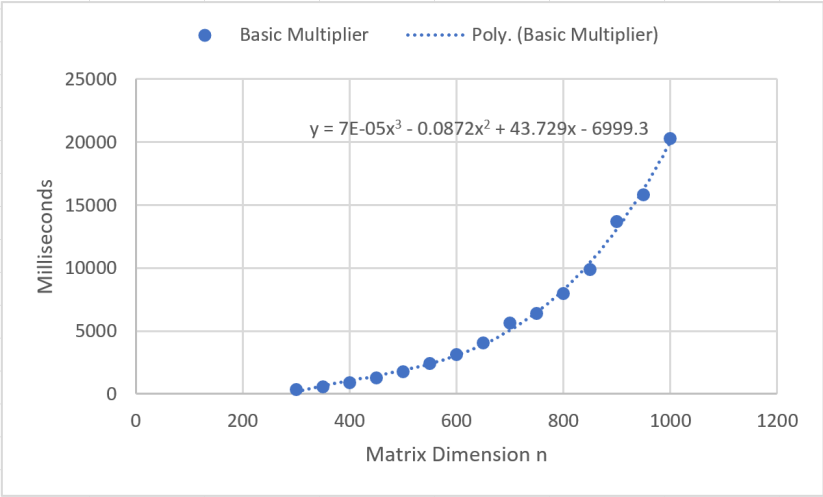

Figure 3.1: *A Plot of Execution Time of Basic Algorithm Against Increasing Matrix Dimension Size*

Figure 3.1 is a plot of average time taken for Matrix Multiplication in milliseconds against matrix dimension n. The curve chosen displays a polynomial, and it is evident that the time grows with n to the power of 3. The behaviour of t can be shown as a function of input size n; as shown in the figure. This plot visualises $O(n^3)$ behaviour of the algorithm which coincides with the initial analysis from Task 1 to a certain extent. However, we might not be satisfied whether this would be an asymptotic behaviour toward $\Theta(n^3)$ and this is when a log-log plot would be used for verification.

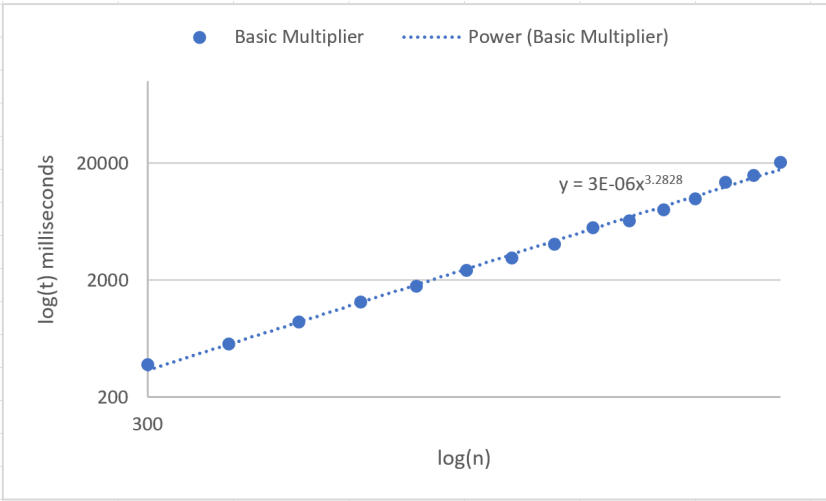

Figure 3.2: *A Log-Log Plot of Figure 3.1*

A log-log plot is plotted to examine its asymptotic behaviour. Assume that the dominant term in computation takes the form $t = an^b$ for large values of n. Then the plot

can be used to verify this assumption by displaying a linear correlation between the two variables, as a function $\log(t) = b\log(n) + \log(a)$. Note that the line of best fit is fitted along the data points for more significant values of n considering preponderant influence of dominant terms. If the points are derived from asymptotic behaviour, for large values of $\log(n)$ the points will correlate close to the line of best fit: shown in Figure 3.2.

Hence, the plots and the functions support the argument that the algorithm follows the average case time complexity of $\Theta(n^3)$ as estimated.

4
Task 3

Research data structures and algorithms that can save time taken, or memory used, or both, for sparse matrix multiplication. Note that we care about the sparse case, so you should think carefully about how you can save time or memory in the case most elements are zero. In your report, describe the results of your research and your sources. This must be backed up by an implementation of at least one idea, in any programming language of your choice. You can either write your own, or reuse code from elsewhere with proper attribution. The assessment will be based on the report.

4.1 Research

I have based my research on three areas.

- Finding alternative algorithms to improve time complexity of IntMatrix Multiplication.
- Finding alternative Matrix implementations using Arrays, to reduce space and time complexity.
- Finding an alternative data structure for Matrix implementation to reduce space complexity.

IntMatrix: Alternative Algorithms

An improved implementation ***improveMultiply()*** of multiplication algorithm is derived from the Basic Algorithm. The algorithm is improved by minimising the number of array entry access, by ignoring zero entries. An if statement is added to check the entries of a matrix A. Since this if statement comes before the innermost loop following can hold: if the value is zero, and there is up to x zero entries in the matrix A, we can reduce xn operations wasted in the Basic Algorithm.

An advanced approach ***AdvancedMultiplier*** shows that further optimisation can be made by understanding the layout of arrays in Java. It is an array of arrays, that is, a 1D array of references to other 1D arrays located elsewhere in memory. Figure 4.1 displays the structure visually, provided by [**gundersen**].

Therefore, unlike fast access to entries of 1D array, accessing a 2D array entry is relatively costly, due to that reference step to go through to find the inner array. In

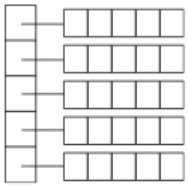

Figure 5: The layout of a two-dimensional Java arrays.

Figure 4.1: *Representation of 2D Array in Java*

addition, as with the preceded implementations, row-wise access is preferred over column-wise as we know that a 2D array is aligned along the row indices, reducing the delays caused by the column-wise access of the outer array. Hence, as argued by Gundersen,"traversing a 1D array instead of a 2D array could be a factor of two more efficient",[**gundersen**] supported by his original research source.[**shirazi**] Ultimately, in comparison to the Basic Algorithm, this advanced approach involves pure row-wise access of every inner array followed by if conditional for checking a non zero entry of the row. Time can be substantially saved by these two components.

Alternative Implementation Using Arrays

From my research, I have come up with three alternative implementations of a matrix. The design decisions explored ways of dropping entries with zeros but non zero values.

Jama

Jama is a Java Package for matrix representation and multiplication. Considering the similarity of Matrix representation with IntMatrix, its multiplication algorithm is an improvement from the Basic Algorithm, by the use of a 1D array multiplication operations. As shown below [**gundersen**], column access of outer 1D array is superceded by construction of 1D array of column index j for matrix B. Since the loop-order follows (j,i,k), the new matrix C is constructed column-by-column.[**gundersen**] Despite making some improvement in some extent, the column-wise construction mean the improvement would not fully offset its resulted downfall.

```
public Matrix times(Matrix B){
    Matrix X = new Matrix(m, B.n);
    double[][] C = x.getArray();
    double[] Bcolj = new double[n];
    for(int j=0; j < B.n; j++){
        for(int k = 0; k < n; k++){
            Bcolj[k] = B.A[k][j];
        }
        for(int i = 0; i<m; i++){
            double[] Arowi = A[i];
            double s = 0;
            for(int k=0; k<n; k++){
```

```
                    s += Arowi[k]*Bcolj[k];
                }
                C[i][j] = s;
            }
        }
        return X;
}
```

Java Sparse Arrays (JSA)

Since Jama package is applicable for both sparse matrix and dense matrix, it is indeed not the best implementation for sparse matrix; it retains zero values taking up space. The design improvement can be made regarding the storage of sparse matrix by using the concept of Java Sparse Arrays. This can be found in ***JavaSparseArray.java***. The concept can be best summarised as a collection of two 2D arrays for column indices and values of non zero entries of a sparse matrix. The original location of the non zero entry can be deduced. For a value x at the location (i, j), i will be the row index of index array, j is entry of ith row at some index k in the ith row of the column index array. Finally x will be located at (i, k) entry of value array. The resulting 2D arrays are therefore jagged arrays. The basic design of the class follows the structure of below, outlined by [**gundersen**], while my implementation will be using int array for Avalue:

```
public class JavaSparseArray{
        private double[][] Avalue;
    private int[][] Aindex;
    public JavaSparseArray(double[][] Avalue, int[][] Aindex){
        this.Avalue = Avalue;
        this.Bvalue = Aindex;
    }
    public JavaSparseArray times(JavaSparseArray B){...}
}
```

Space is saved in comparison to IntMatrix matrix; let nnz be the number of non zero entries. Then a JSA instance will typically use $2nnz + 2n$ of space locations[**gundersen**] for having two outer arrays each up to length n (dimension) and a total of $2nnz$ locations for storing values and column indices. Since $nnz < n^2$ by the sparsity of a sparse matrix, for large values of n, the memory used will be relatively improved.

Compressed Row Storage (CRS)

As with JSA, compressed row storage (CRS) also considers storing non zero entries only. The critical difference would be that CRS contains values and column indices continuously, each in a 1D array of length nnz. There is an additional 1D array of length $n + 1$ to store row pointer values. This is effectively the key to identify the row index of an entry. During the matrix traversal, the number of non zero entries will be counted and at the end of each row, this value will be recorded in the array. Hence, for i from 0 to n, the difference between the two row pointer entries $[i]$ and $[i + 1]$ is the number of non zero

entries at row i of the original matrix. The structure is also useful since the last entry of the array will always be nnz.

Implementation of CRS and CRSMultiplier can be found in ***CRS.java*** and ***CSV-Multiplier.java*** respectively. The design of CRS implementation should be attributed to [**patty**] as a source of pseudo code, which I extended as Java implementation. The concept is also explained by [**gundersen**] with an example below.

$$A = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix}. \tag{3}$$

The nonzero structure of the matrix A (3) stored in the CRS scheme:

```
double[] value = {10, -2, 3, 9, 3, 7, 8, 7, 3, 8, 7, 5, 8, 9, 9, 13, 4, 2, -1};
int[] columnindex = {0, 4, 0, 1, 5, 1, 2, 3, 0, 2, 3, 4, 1, 3, 4, 5, 1, 4, 5};
int[] rowpointer = {0, 2, 5, 8, 12, 16, 19};
```

Figure 4.2: *Example of CRS Representation of a Sparse Matrix*

Of the space used by the implementation, it uses even smaller than JSA, which is $2nnz+n+1$. In addition, the time complexity can also be reduced by its unique implementation of matrix multiplication algorithm. Multiplication process is done incrementally by row using a Sparse Accumulator. SPA is used as a temporary storage space for non zero entries of a row which is to be appended to the resulting matrix after every iteration of a row. Details of its implementation can be found in ***SPA.java***. Since CRS deals with 1D arrays only, the time taken for accessing entries will be quicker than JSA which also the number of nnz for the product is estimated to be around $1.5(A.nnz \times B.nnz)$[**patty**]. Its compactness and row-wise orientation for operations suggest CRS as a convincing option.

My implementation involves initialisation of 1D arrays up to n^2 entries, due to the unknown value of nnz for the product of two CRS matrices. At the end of multiplication, the arrays will be trimmed up to nnz. Such decision has been made since time complexity of this loop would be negligible in multiply algorithm complexity as the prominent term will be dominant.

Alternative Data Structure: HashMap

Another example of matrix implementation considering space efficiency is the use of a dictionary of keys. The example implementation I have found is written in Python[**scipy**]; which I have implemented using HashMap to emulate a dictionary of key and values. The key is a row-column index pair saved under Pair class; found in ***MapMatrix.java and Pair.java***. The design of Pair class is attributed to [**mapPair**]. The implementation also drops any zero entries as with others. I have found this implementation to be potentially useful considering its constant time of average $O(1)$ operation for GET and containsKey operations, while the space used will also be $O(n)$ on a hash table.

5
Task 4: Task 5

Investigate in what situations your implementation from Task 3 does better than the Basic Algorithm, and in what situations (if any) the Basic Algorithm wins. This will involve similar data analysis as Task 2, but you will have to decide the characteristics of the input you want to modify. Describe in the report your testing strategy and your results.

5.1 Testing Strategy

Input Matrices

For all the tests, matrices are passed as parameters and tested similarly as that from Task 2. The matrices will be randomly generated according to the set parameeter values. In this task, in order to analyse their asymptotic behaviour and persistent accuracy, default values are set as follows. The minimum n is chosen to be 500 with the maximum of 1000, compensated by increased value of repetition for each n of 25 different matrices. Sparsity is set to be 0.75 by default. Since the input will be the same for all three tests, this will minimise the effect of other variables but changes in algorithmic complexity for increasing values of n.

Test Classes

Test classes are arranged and tested as follows:

- **Test 1**: ***CompareMultipliers1Test.java*** tests three algorithms using random IntMatrix sparse matrices: Basic Algorithm, Improved algorithm (***improvedMultiply()***) and Advanced algorithm (***AdvancedMultiplier.multiply()***). Since the algorithms will get the same matrices at every iteration, the test will look at the effectiveness of each algorithm.

- **Test 2**: ***CompareMultipliers2Test.java*** now tests the Basic Algorithm applied to matrices under varying implementations. This test will typically look at the space complexity rather than the time complexity of algorithms for sparse matrix storage purposes. As an exception, CRS and Jama will be tested with the custom multiplication algorithms. Hence, four matrix implementations tested are: *JSA* and *MapMatrix* under *BasicMultiplier*, *CRS* under *CRSMultiplier* and *Jama Matrix* under its packaged built multiplication algorithm.

- **Test 3**: ***CompareMultipliers3Test.java*** now tests to find the optimal implementation of sparse matrices and multiplication algorithm by combining the optimal values. All five matrix implementations in the previous test will be used again, but this time, the multiplication algorithm will be the Advanced Algorithm based on our initial analysis that this will outperform Basic and Improved Algorithm implementations. Additionally, this test class will be used to test on varies parameter values controlling the characteristics of input matrices (Task 5).

5.2 Analysis

Test 1: Comparing Alternative Algorithms

Figure 5.1 shows that the improved and advanced algorithm implementations hugely outperform the Basic algorithm in average case of random matrices. We can see that the outcome can be justified theoretically by considering the time saved by minimising the number of operations. The Basic Algorithm will do all n^2 iterations and so as the arithmetic operations regardless of the entry values. On the other hand, both improved and advanced algorithms will bypass zero entries. Considering that the sparsity has been set as 0.75, the algorithms will conduct around $\frac{3}{4}$ fewer operations out of $c_2 n + c_3$ on average. Note that c_3 constant time assignment is now part of the innermost loop thereby time can be further reduced.

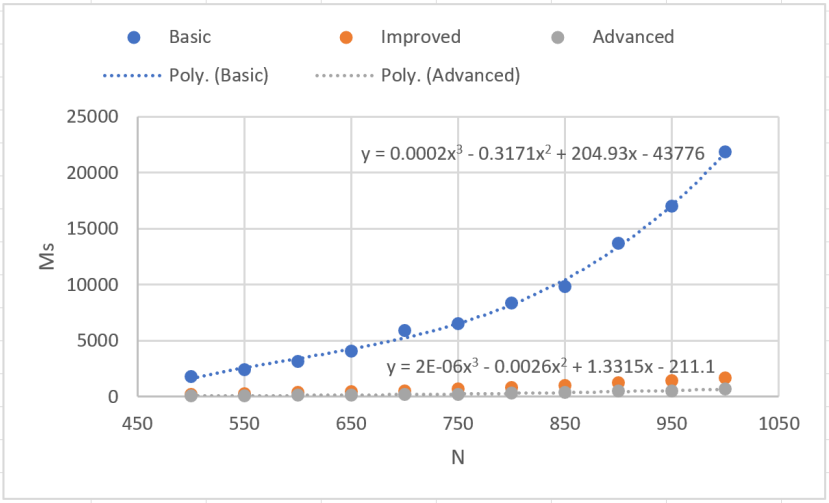

Figure 5.1: *A Plot of Execution Time of Basic and Alternative Algorithms Against Input Matrix Dimension N*

The plot also supports that Advanced algorithm further improves the time taken in comparison to the improved algorithm by 1D array operations, despite that the improvement is indistinguishable due to the worst performance by the Basic Algorithm. A 1D array access will reduce the time by a factor of 2 from 2D array access [**gundersen**] and that proportion of non zero entries is $\frac{1}{4}$. Therefore, in total, it will conduct at most $\left(\frac{1}{4}\right)\left(\frac{1}{2}\right)n^2$ operations compared to the Basic Algorithm. This calculation is evident from the functions shown in the Figure 5.1: the coefficient of the dominant term has dropped dramatically and contributions of lower order terms have also reduced.

Test 2: Comparing Alternative Implementations

Figure 5.2 shows the output of computation using random sparse matrices. This test aimed to assess the suitability of using alternative sparse matrix implementations while keeping the multiplication algorithm fixed, in an attempt to save memory used for matrix storage purposes. The time measured for JSA and MapMatrix includes not only the time taken for multiplication but also its conversion to IntMatrix.

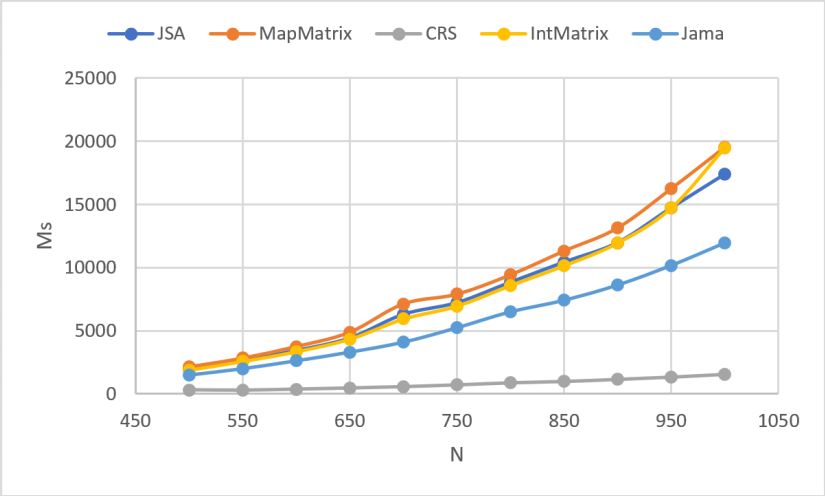

Figure 5.2: *A Plot of Execution Time of Basic Algorithms on Different Matrix Implementations*

Considering that JSA, MapMatrix and IntMatrix use the Basic Algorithm, the overlap in data points is understandable. The time spent on conversion is negligible for smaller values of n but would ultimately accumulate as n increases. The time complexity for the conversion of JSA to IntMatrix will be governed by the double loop giving $\Theta(n^2)$ after some simplification of $\Theta(c_2 + c_3 nnz)n^2 + c_1 n)$. It will always run n^2 times and the contribution will be approximately the same despite the if conditional in the inner loop because of the constant time operations.

Jama outperforms as expected from our analysis in Task 3; the total time is reduced approximately by a factor of 2 according to the plot. Furthermore, CRS is the only implementation here which uses its multiplication algorithm and has shown the best performance. Its multiplication algorithm involves an iteration of n times each of constant time operations in $CRSMultiplier$; such that the time spent in the inner double loop beats the Basic Algorithm by far since it only traverses along non zero elements according to the row pointer values and column indices. This is because the inner double loop will only iterate approximately $\left(\frac{3}{2}\right)\left(\frac{1}{4}\right)^2 n^2$ operations giving a total of $\Theta\left(\frac{3}{32}n^3\right)$ time complexity; due to 0.75 sparsity and the approximation that the product of matrices will have approximately $1.5 \times nnz^2$ nnz values.This implies that although the upper bound would be $O(n^3)$ but practically exhibiting much better performance.

Test 3: Finding the Optimal Implementation

Figure 5.3 shows how nnz number of operations done out of n^2 in the Advanced algorithm overtakes the full n^2 number of iterations taken in Jama as expected. It consistently

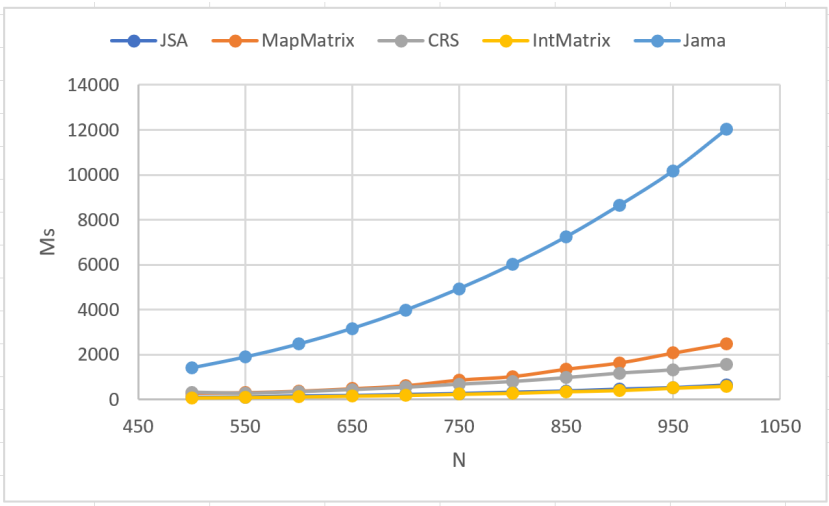

Figure 5.3: *A Plot of Execution Time of Advanced and Alternative Algorithms on Different Matrix Implementations*

exhibits $O(n^3)$ complexity. Hence, for the subsequent analysis of other implementations, time complexity for matrices of varying shape will not be measured for Jama.

In general, the average time taken has dramatically reduced between Figure 5.2 and Figure 5.3 by using the Advanced algorithm approach.

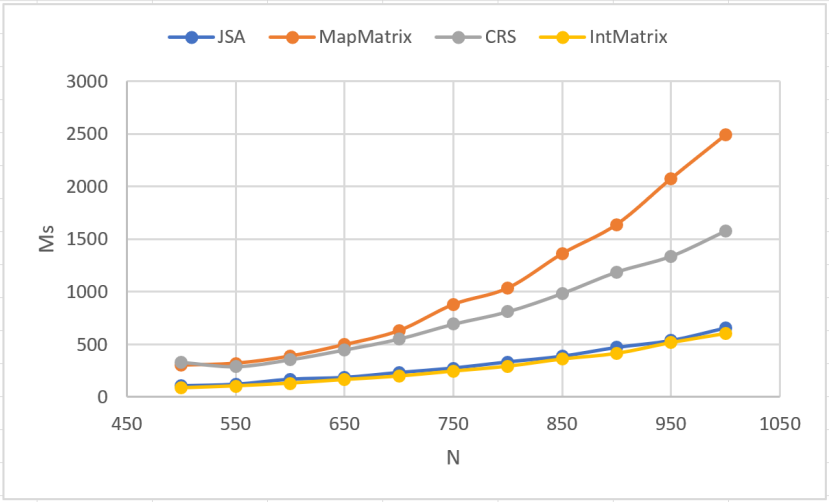

Figure 5.4: *A Plot of Figure 5.1 excluding Jama*

The argument from the previous section on MapMatrix is supported by Figure 5.4 where its growth deviates from the rest.

To evaluate the optimal algorithm and space-efficient matrix implementation, it would be necessary to explore the behaviours of algorithms under different conditions applied. Hence, further testing of the third comparison has been done in the next section.

Task 5: Additional Testing and Analysis

Give a theoretical justification for your results from Task 4, doing a complexity analysis based on input sizes and the characteristics of sparse matrices. This will be assessed by our judgement on how you understand the analysis, so simply copying down complexity analyses from published sources is unlikely to convince us that you understand the analysis and principles behind it.

I have used the following characteristics of sparse matrices to generate inputs using method $Utils.getParamsByCondition()$:

- **Matrix Dimension** n
  The dimension of a sparse matrix denoted as n is indeed the significant factor to consider in the analysis. And I have chosen the minimum and maximum value considering the memory allowed for matrix generation before testing.
- **Sparsity**
  As mentioned in the preceded section, sparsity denotes the proportion of zero entries of a matrix. The higher value of sparsity implies emptier the matrix will be. According to the value of sparsity, a sparse matrix can be generated by computing the corresponding number of non zero entries out of an $n \times n$ matrix. Sparsity variable is passed as double, while this factor was fixed as 0.75 to represent a random sparse matrix for testing done for Task 2 and first of Task 4 as explained previously.
- **Shape of a Sparse Matrix: Entry Positions**
  This is another variable controlled. Two other special cases of sparse matrices are considered.
  1. **Random Matrix [0]** Randomly chosen indices and entry values will generate a matrix.
  2. **Band Matrix [1]** A band matrix is a matrix where its non zero entries are along the main diagonal and one more on either side.
  3. **Upper Triangular Matrix [2]** A triangular matrix is a matrix where all non zero values appear above the main diagonal.

Test 3 Analysis Under The Controlled Environment

Some advanced testing are done by changing the values of sparsity and types of matrices used. More outputs which are not part of this report are accessible in "data" directory.

Increasing Sparsity

Comparison of Figure 5.3 with a plot 5.5:

An increase in sparsity from 0.75 to 0.9 for random matrices provided a similar pattern as the above Figure 5.3 in their asymptotic behaviour. The smaller value of nnz values gave an improvement in running time that is indistinguishable. This is reasonable since the contribution of nnz would be reduction in the number of constant time operations, overtaken by the dominant term.

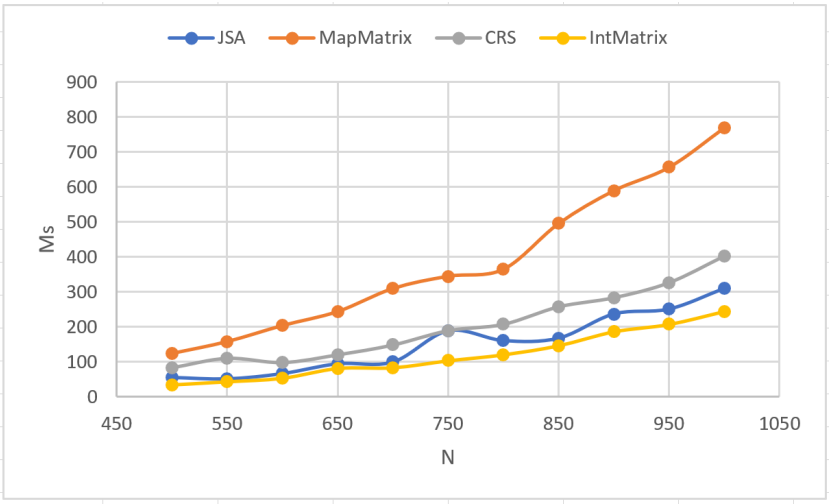

Figure 5.5: *A Plot of Execution Time of Multiplication of Random Matrices with Sparsity 0.9*

Changing the Shape of a Matrix

Comparison of Figure 5.4 with 5.6 and 5.7 :

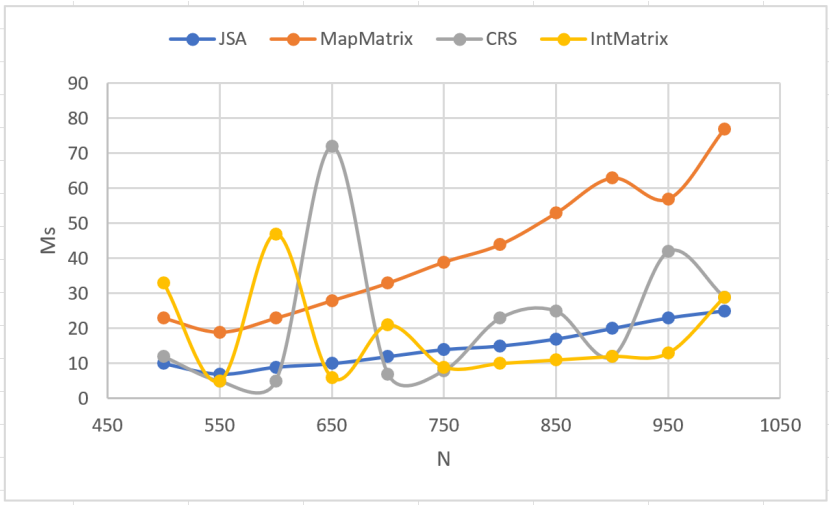

Figure 5.6: *A Plot of Execution Time of Multiplication of Band Matrices excluding Jama*

In general, the shortest execution time was required for band matrices among all three types of matrices. Despite some fluctations we encount from the plot, this is highly negligible. Nonetheless, JSA has consistently performed well for all three cases which again can be justified using its implementation, on the other hand IntMatrix implementation shows some fluctuations for Band Matrices, as shown in Figure 5.6. Performances of Advanced algorithms for IntMatrix and JSA are consistent for triangular matrices, however. The plots show that Advanced Multiplier can indeed accommodate all three types of sparse matrices, while the shortest times were taken in the band matrices.

According to the plots, CRS performs relatively well for triangular and randome matrices band matrices. This can be explained by CRS representation of matrix and its algorithm: incremental update made through row iterations will create the CRS matrix.

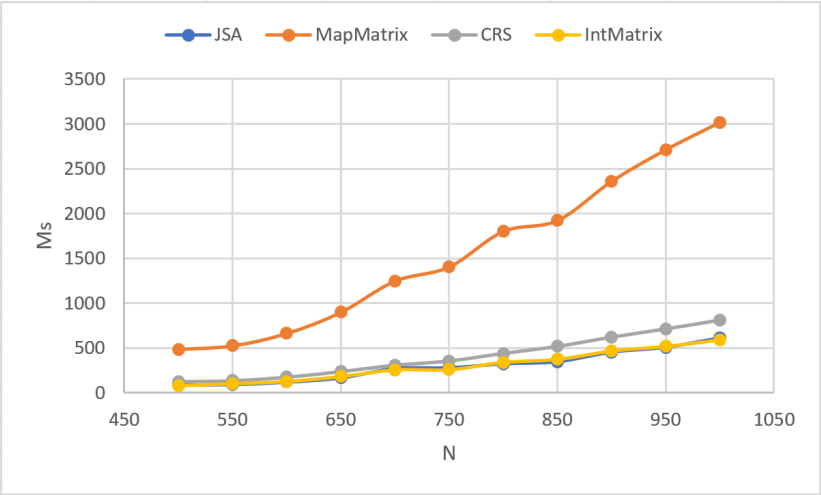

Figure 5.7: *A Plot of Execution Time of Multiplication of Upper Triangular Matrices excluding Jama*

For triangular matrices, a non zero value is likely to be followed by another non zero value leaving while bypassing any empty rows. Internally, this would reduce time in row iterations in comparison to band matrices.

The method of conversion for MapMatrix would theoretically also exhibit $\Theta(n^2)$, simplified from $\Theta(c_1+c_2nnz)n^2)$ regardless of the type of matrices it deals with. Combined with the complexity of the Advanced Algorithm, the overall contribution shows the costs in time, showing $O(n^3)$ performance. Figure 5.7 proves this argument; time is not saved for varying types of matrices it takes. Ultimately, MapMatrix implementation follows its theorectical complexity which others exhibit some improvement from its theorectical counterparts in terms of the growth.

5.3 Conclusion

Overall, this practical investigated complexity achieved in practical implementations in comparison to the theoretical complexity. The Basic Algorithm can be replaced with the optimal implementation of sparse matrix multiplication based on the justification made using the above analysis; memory efficient matrix implementation such as JSA can effectively accompany the Advanced Algorithm. Considering that Band matrix is the most common representation of a sparse matrix in practice, the Advanced algorithm can be regarded as the most probable approach to take; it has persistently shown optimal results against the others for various different cases. CRS could also be an option considering that it does not require any intermediate step for conversion by both memory and time efficient implementations. The final decision between the two would lie on the convenience

Bibliography

[1] Geir Gundersen, Trond Steihaug, *Data Structures in Java for Matrix Computations*, 2002, http://heim.ifi.uio.no/~nik/2002/Gundersen.pdf

[2] Jack Shirazi, *Java Performance Tuning*, O'Reilly, Cambridge, 2000

[3] Spencer Patty, *Math 639, Compressed Row Storage (CRS) Format for Sparse Matrices*, 2014, http://www.math.tamu.edu/~srobertp/Courses/Math639_2014_Sp/CRSDescription/CRSStuff.pdf

[4] https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.dok_matrix.html

[5] https://stackoverflow.com/questions/28953856/accessing-a-hashmap-value-with-a-custom-pair-object-as-a-key