CS3052 Practical One

150001088

12 March 2018

Contents

1 Overview

This practical addresses the widely known issues of the time and space complexity in matrix multiplication with a focus on sparse matrices. This document tries to give an introduction into the topic and then presents an approach how to improve the naive version of matrix multiplication. In the first two parts of this report the focus is on worst and average time complexity of the native algorithm, the third and fourth part cover memory and time complexity of sparse matrix multiplications and the improvements thereof.

1.1 Reproducabilty

All my experiments are reproducible with the code submitted. When the main method is run all relevant csv files for this analysis are created. However, they all carry a "_2" extension to their name to avoid confusion with the submitted files. Please note that the duplication of code is intended in this case and is used for clarification purposes. This document has been created with R markdown and all R code snippets are echoed in the submission.

2 Worst case analysis

Looking at the given matrix multiplication algorithm we see that the function has the following operations in this order:

1. Assignments takes a time.
2. Triple for loop with each going from 1 to n giving it $O(n^3)$. In the second loop there are n^2 assignments of time b and then there are n assignments in the innermost loop and a multiplication of c.
3. Total time is $O(a + c * n + b * n^2 + c * n^3)$
4. Using the simplifications rules I only need to take the maximum of these and can drop constants, so the time is $O(n^3)$

Hence, the worst case time complexity for this matrix multiplication algorithm is $O(n^3)$.

3 Average case analysis

3.1 Methodology

To analyze how the behavior of the algorithm changes with input size, the class *GenerateTestData_Task2* has been created. In this class Java's random function with the seed 1234 is used to generate the entries of the matrices and allow the tests to be reproduced. The results will be saved in a csv file, which is then used for further analysis in R. The data will be accumulated in a libre office calc document for manual inspection.

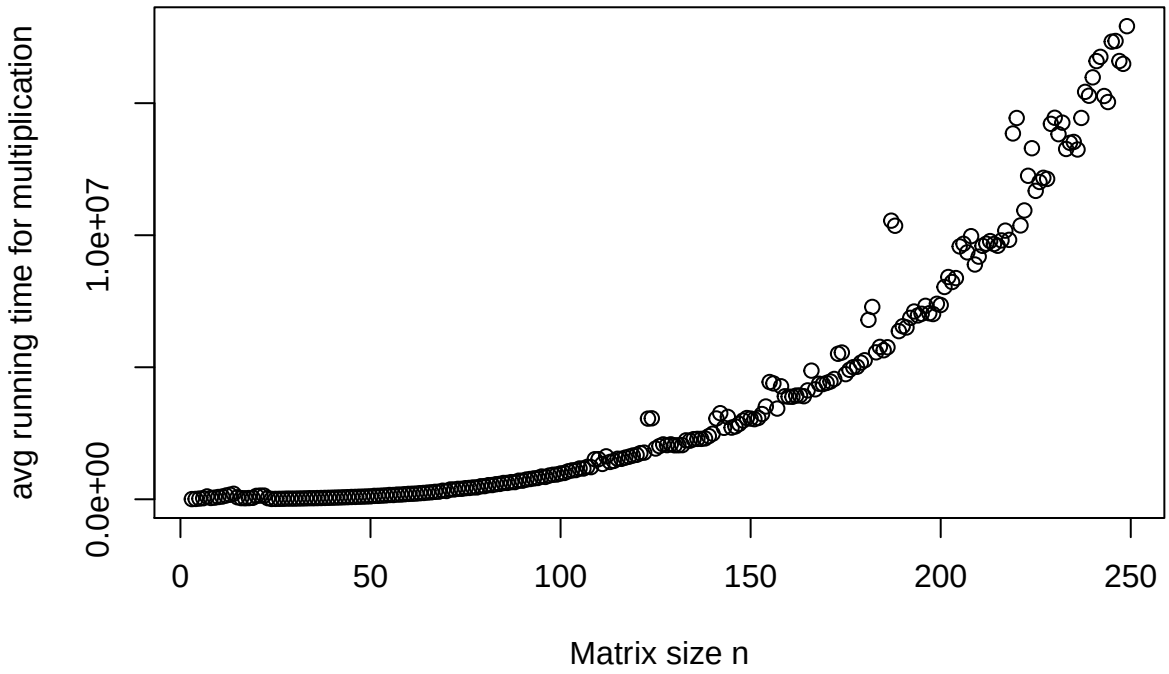
For the Analysis matrices with dimension 3 to 249 in size were used, plus a native Java time taking method to measure execution time when multiplied. For each size there will be ten samples.

3.2 Analysis

```
# data generated by "generateTestData_base_case(1234,3,250)"
df = read.csv("r_analysis.csv")
```

The file *r_analysis.csv* contains the average running times for the matrices sized between 3 and 249. To get a first feel for the data, it is visualized in the plot below.

```r
# plots running time of native algorithm vs time
plot(df$mat_size,
     df$avg.times,
     xlab="Matrix size n",
     ylab = "avg running time for multiplication")
```



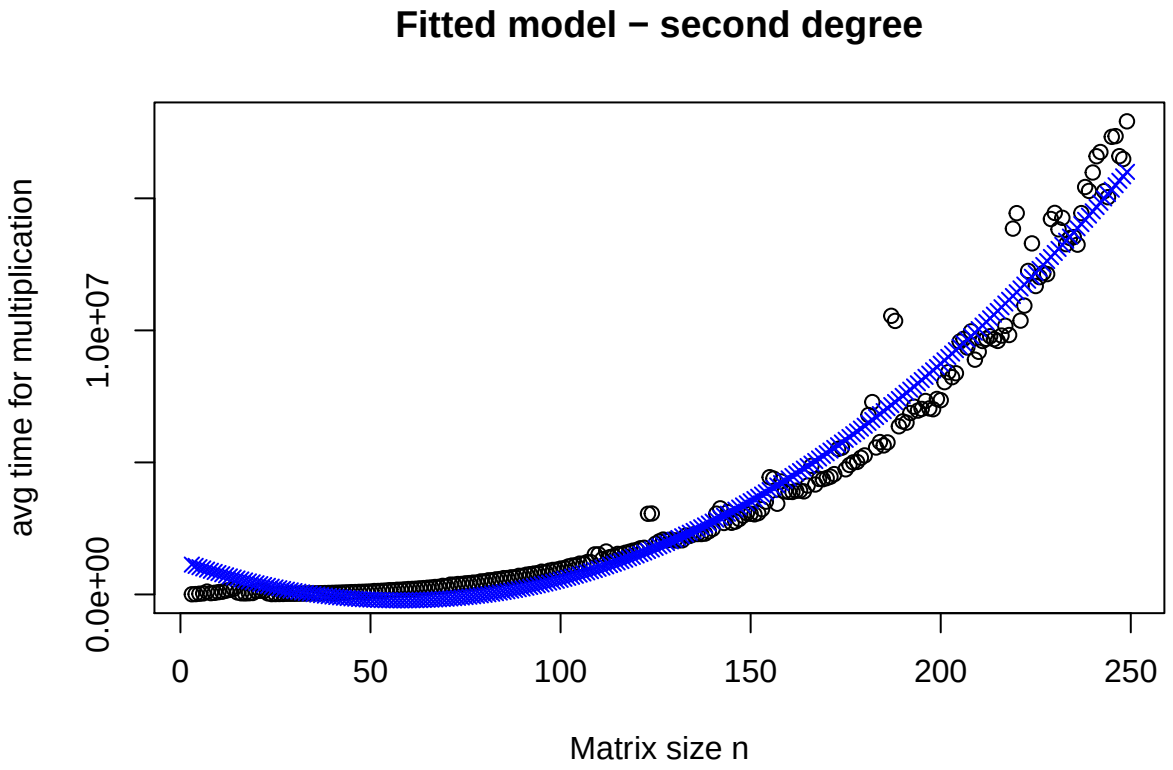
There are a few irregularities in the data, however, these will be caused by other processes on the machine slowing down the multiplication and should not impact the analysis.

As a next step a polynomial can be fitted to the data and the statistical significance of the factors can be tested. A polynomial of second, third and fourth degree will be fitted.

```r
# df contains two columns: avg.times and mat_size from the experiments
# linear model degree 4
fit.four = lm(df$avg.times ~ poly(df$mat_size,4))
# linear model degree 3
fit.three = lm(df$avg.times ~ poly(df$mat_size,3))
# linear model degree 2
fit.two = lm(df$avg.times ~ poly(df$mat_size,2))
```

The quadratic model can be plotted on top of the data points to get a feel for the fit and to visualize the estimates.

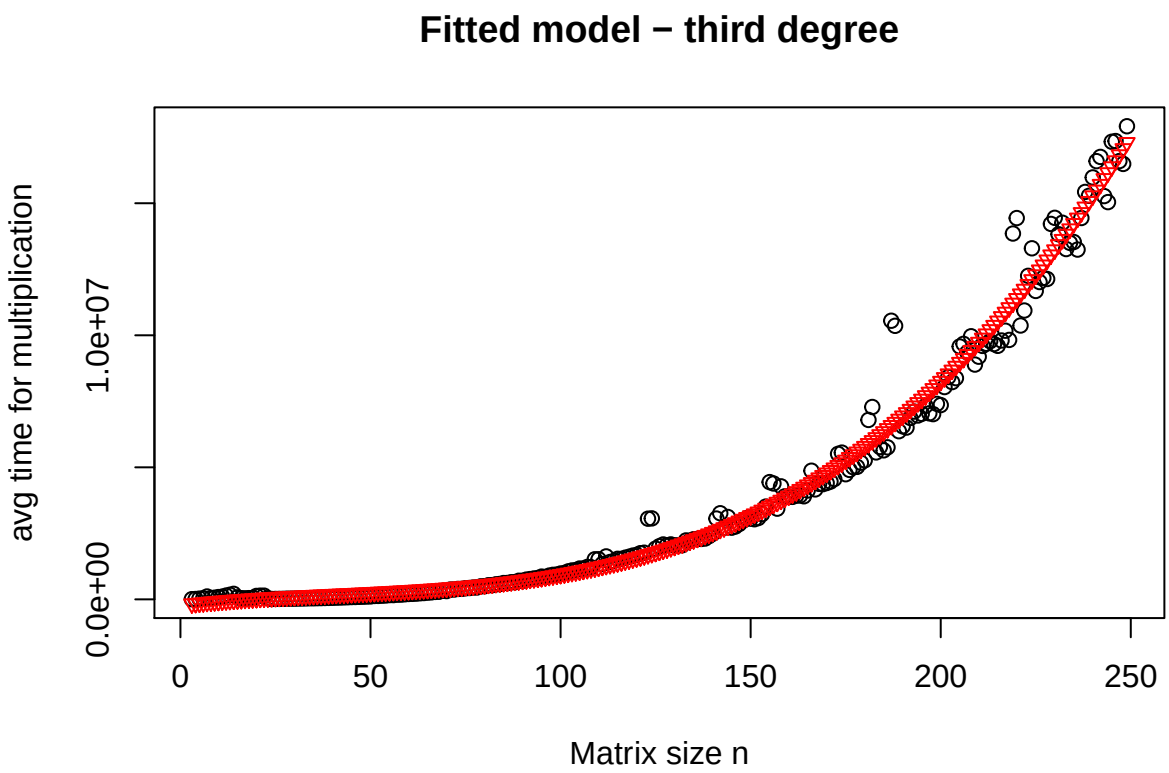
```r
# plots the quadratic fitted model
plot(df$mat_size,
     df$avg.times,
     xlab="Matrix size n",
     ylab = "avg time for multiplication",
     main="Fitted model - second degree")
points(df$mat_size,
       fitted(fit.two),
       col="blue",pch=4)
```



This appears to be a good fit, however, the dip between 0 and 100 seems to be unnatural for an increase in problem size. It also appears that between sizes 150 and 200 the fitted values are quite far off compared to the observed values.

The fitted model of third degree appears to offer a much better fit from a visual inspection below.

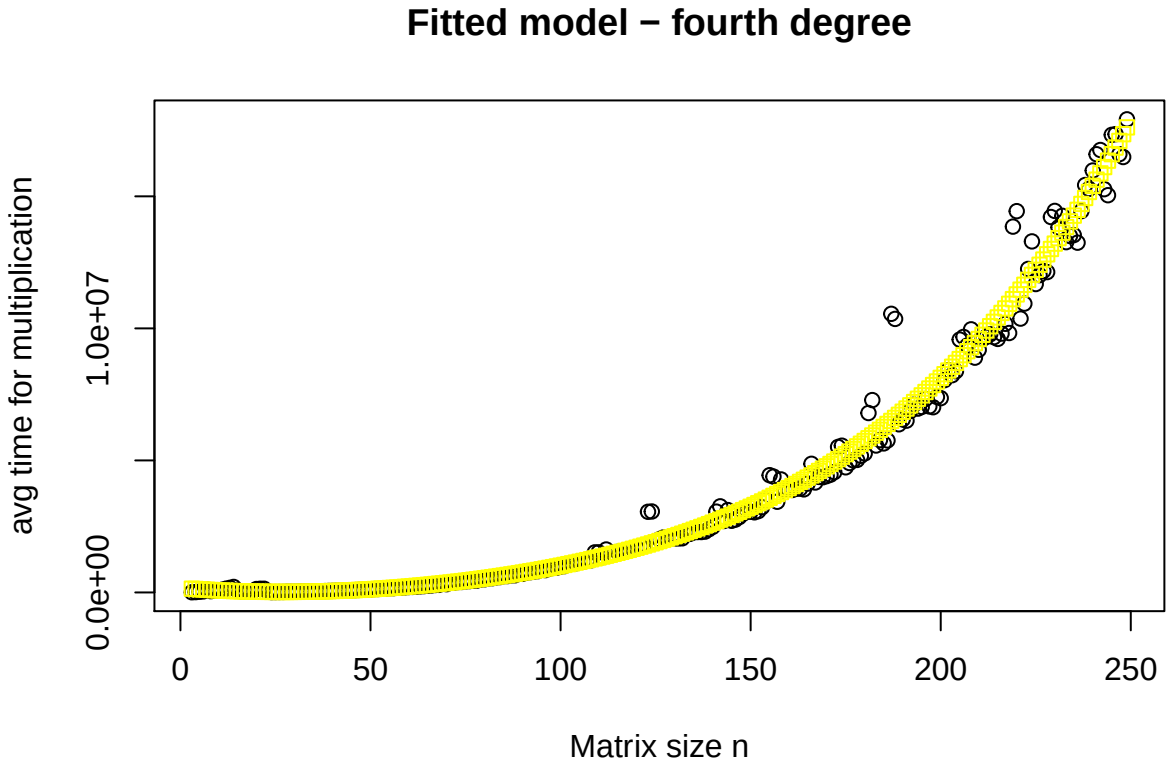
```
# plots the cubic fitted model
plot(df$mat_size,
     df$avg.times,
     xlab="Matrix size n",
     ylab = "avg time for multiplication",
     main="Fitted model - third degree")
points(df$mat_size,
       fitted(fit.three),
       col="red",pch=6,cex=0.8)
```



Especially between 0 and 150 the fitted values seem to align with the observed values perfectly and also the trend for larger matrices seem to be captured better than in the quadratic model.

Naturally the model of fourth degree should offer a better fit, as more factors allow a better fit for an observation so it is no surprise that also this model captures the general trend very well.

```
# plots the model of fourth degree
plot(df$mat_size,
     df$avg.times,
     xlab="Matrix size n",
     ylab = "avg time for multiplication",
     main="Fitted model - fourth degree")
points(df$mat_size,
       fitted(fit.four),
       col="yellow",pch=0)
```



However, the difference between the third and the fourth degree models seems to be smaller than for the quadratic and cubic model. To gather more evidence what model should be preferred, one could rely on an analysis of variance (ANOVA) test.

As we have nested models in this case, we can conduct a simple anova test between the different models. In particular, the test aims to detect if an extra parameter improves the fitted model significantly, without risking to over-fit the model.

Comparing the quadratic and the cubic model:

```
#compares qudaratic and cubic model
anova(fit.two,fit.three)
```

```
Analysis of Variance Table

Model 1: df$avg.times ~ poly(df$mat_size, 2)
Model 2: df$avg.times ~ poly(df$mat_size, 3)
  Res.Df        RSS Df  Sum of Sq      F    Pr(>F)
1    244 1.5237e+14
2    243 8.9865e+13  1 6.2501e+13 169.01 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The finding that the parameter in dispute, the parameter raised to the third power, is highly significant should be no surprise, considering the difference was clearly visible in the plot. Hence, we would prefer the cubic model over the quadratic model.

Comparing the cubic and the model of fourth degree:

```
# compares model of third and fourth degree
anova(fit.three, fit.four)
```

```
Analysis of Variance Table
```

```
Model 1: df$avg.times ~ poly(df$mat_size, 3)
Model 2: df$avg.times ~ poly(df$mat_size, 4)
  Res.Df        RSS Df  Sum of Sq      F   Pr(>F)
1    243 8.9865e+13
2    242 8.6986e+13  1 2.8791e+12 8.0098 0.005043 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This case is slightly more complicated. While there is still a significance on the 0.001% level, which would be considered highly significant in cases where the underlying process is assumed to follow a random distribution, however, in this case, it can be assumed that the underlying process is not random, but follows a strict mathematical function and only the errors are normally distributed. Under this assumption, one can consider the cubic model superior as it captures the general trend well, which is what a big-O analysis aims to do.

Based on the knowledge of Task 1 and the plots above, there is a second way to obtain an indicator of average time complexity. To get the second indicator, we can simply take the second, third and fourth root of the execution time and take the difference between the largest and the smallest size to see what power drives the time complexity.

```r
#used for the table below
rootdiff <- function(x){
  root.first = df$avg.times[1]**(1/x)
  root.last = df$avg.times[247]**(1/x)
  return(round(root.last-root.first,digits=0))
}
```

n^{th} root	Difference last-first
2	4144
3	242
4	56

Note that for the third root the difference is roughly 242 between the matrices sized 3 and 249, while the problem size increases by 246, about the same magnitude. For polynomials in general, a problem of size $n+246$ should take $(n+246)^3$ time, compared to a problem of size n taking n^3. The difference can explained through the machine, involved constants, polynomial factors of lower degree and other processes on the machine. As stated in lectures before, we are interested in the overall trend, hence the conclusion that this process is very likely to be $O(n^3)$. While this works for a cubic polynomial in this case, we see that for both the second and fourth root the differences are far from 246. This indicates that this is not very likely to be neither $O(n^2)$ nor $O(n^4)$. It can be expected that increasing the roots further, will decrease the difference, leaving $O(n^3)$ the best estimate.

Based on ANOVA, the visualizations, the theoretical analysis in Task 1 and the conclusion that the running time of the native matrix multiplication algorithm follows a cubic polynomial, it can be assumed that the algorithm has $O(n^3)$ time complexity in its average case.

4 Time and memory complexity

While the chapters above cover the cases for dense matrices, the following paragraphs focus on sparse matrix multiplication. This analysis will be split into two parts. Firstly, space complexity will be analysed and an improvement for the given data structures will be suggested, afterwards a suggestion for time complexity will be introduced. These implementations will the be tested in the last chapter.

4.1 Suggestions to improve memory complexity

There multiple ways to approach this issue and this practical introduces a common ones which will ultimately lead to lower memory complexity. At first it is crucial to point out that the major pain point of the current implementation is the storage of zero entries. To reduce the number of zeros, one can make use of the fact, that matrices can be compressed. Compressing describes the process to drop entries which are zero and storing the values along other information in a new format. There are multiple ways to compress a matrix, these include compressed columns, compressed rows or block compressed row storage. For the purpose of illustration this practical will present a way to compress a sparse matrix into a compressed row format. In 2002 Gundersen and Steihaug [1] introduced in their paper different ideas how to store matrices in smaller memory. They suggest a compressed row data structure in which three 1D arrays are maintained. These include:

1. non-zero values array
2. column index array, noting the column in which the non-zero value was stored
3. row pointer array, containing the index at which a new row begins in the values array if transformed back into 2D representation.

For a NxN matrix with m non-zero entries, the given standard implementation requires $n * n = n^2$ storage space, however, the compressed row implementation requires m units of storage for the first array, similarly m units of storage for the second array, as they have the same size and ultimately $n + 1$ units of storage for the third entry, as there should be as many entries as there are rows plus an additional one for the last entry. Adding these up, we only need $m + m + n + 1$ storage, which can be simplified to $2 * m + n + 1$, which is significantly less than n^2 in the standard implementation as we know $m << n^2$, $n < n^2$ and by simplification rules we can drop constants leaving growth with a maximum of degree one.

4.2 Implementation of the suggested data structure

In a first implementation the idea to drop all zero entries and 2D arrays was picked up. In the given implementation, which can be found in *CRS.java*, three arrays are maintained, which will be filled with the values passed in as an IntMatrix as given in the specs. Please note, that the time complexity of this parsing method is not great, but this wasn't the purpose of this implementation as the focus is the memory. All the code for this part has been created by the student and is printed below.

```
public class CRS {
    int[] value;
    int[] column_index;
    int[] row_pointer;

    public CRS(IntMatrix matrix){
        this.row_pointer = new int[matrix.getDim()+1];
        int non_zero = get_non_zero(matrix);
        this.column_index = new int[non_zero];
        this.value = new int[non_zero];
        int k = 0;
        int t = 0;
```

```
        row_pointer[t]=0;
        t++;
        for (int i = 0; i < matrix.getDim() ; i++) {
            for (int j = 0; j < matrix.getDim() ; j++) {
                if(matrix.get(i,j)!=0){
                    value[k] = matrix.get(i,j);
                    column_index[k] = j;
                    k++;
                }
            }
            row_pointer[t] = k;
            t++;
        }
    }

    private int get_non_zero(IntMatrix matrix){
        int k =0;
        for (int i = 0; i < matrix.getDim(); i++) {
            for (int j = 0; j < matrix.getDim(); j++) {
                if(matrix.get(i,j)!=0){
                    k++;
                }
            }
        }
        return k;
    }
}
```

4.3 Suggestion to improve time complexity

As discussed widely in the first two tasks the time complexity of the standard algorithm for dense matrices will always be $O(n^3)$, however, considering sparse matrices now, we can come easily to the conclusion that many calculations can in fact be skipped. All shortcuts can be drawn from the knowledge that $x * 0 = 0$ for all x. The biggest barrier now is how to make use of this fact.

The most intuitive approach to this problem is almost certainly the one to skip zero entries, for which there are multiple approaches. However, this approach might decrease running time as shown later on, this is due to the way 2D arrays are implemented in Java. One main discovery is that row wise and column wise multiplication actually have different running times. In a 2D array, the implementation is a first array containing a pointer to a second array somewhere on the disk, however, these pointers may point to locations far apart which leads to increased access times through read and write proccesses and decreased efficiency (p. 2/3, Gundersen and Steinhaug, 2002).

For example the algorithm given from class is a mixed approach, it is neither column only nor row only. Based on the paper I will adapt a row only approach conbined with skipping zero elements, which is can also be found here at link or at [2].

As none of the approaches make use of the suggested compressed row data structure above this remains an option as well. While CRS matrix multiplications are mentioned in multiple papers, none of them offer a working solution, this lead me to code up my own implementation However, the problem with this approach will be that the layout of the resulting matrix will not be known until the matrix has actually been calculated. While we know the dimension in a full matrix, we don't know the form of the compressed row matrix. There are three options:

1. The whole system works with compressed row structures and the resulting matrix needs to be transformed.
2. The whole system works with standard matrices and they need to be parsed into CRS form before calculating the product.
3. The system supports both structures. Passing in compressed row matrices when a multiplication is needed, but accepts a standard 2D matrix as a results.

While option three is very likely to be the fastest, the use cases need to be carefully considered. Taking the case that just the result is nedded in some form, this is the go to solution, however, when the result also needs to make efficient use of memory one might prefer option one, event though the extra transformation will increase the running time. Option two is an option when the user doesn't worry about memory, but only about the multiplication and the parsing is expected to be fast.

4.4 Implementation of the suggested improvement

The implementations of my algorithms can be found in *BasicMultiplier* in the methods *multiply_crs* for the CRS algorithm which simply returns a full 2D matrix but takes compressed row matrices,*multiply_skip_zero_changed_iterations* for the more advanced zero skipping algorithm and *multiply_skip_zero* for the native approach when skipping zero entries. Also interesting in the skipping zero case is the fact, that the sum is not calculated in one for loop, but rather build incrementally over the course of the whole calculation. The CRS algorithm is something the student came up with after studying the data structure in Gundersen and Steinhaug, similar with the native skip zero algorithm, which was a first approach.

The CRS algorithm:

```
public IntMatrix multiply_crs(CRS a, CRS b) {
        int dim = a.row_pointer.length-1;
        IntMatrix result = new IntMatrix(dim);
        for (int i = 0; i < a.row_pointer.length-1; i++) {
            for (int k = a.row_pointer[i]; k <a.row_pointer[i+1]; k++) {
                for (int j = b.row_pointer[a.column_index[k]];
                j < b.row_pointer[a.column_index[k]+1];j++){
                    result.set(i,b.column_index[j],
                    result.get(i,b.column_index[j])+a.value[k]*b.value[j]);
                }
            }
        }
        return result;
    }
```

The pure row approach including skipping zeros:

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

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

```
        return result;
    }
```

The native approach including skipping zeros:

```
public IntMatrix multiply_skip_zero(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++) {
                    if(a.get(i,k)==0||b.get(k,j)==0){
                        continue;
                    }
                    sum += a.get(i,k) * b.get(k,j);
                }
                result.set(i,j,sum);
            }
        }
        return result;
    }
```

The running times will be studied in the next section.

5 Data Analysis of improved algorithms

In the first half the memory complexity of the compressed row format for sparse matrices is tested. In the second part of this section the focus is on the time complexity of the suggested algorithms and their response to change in sparsity and size. To allow randomness for numbers to be multiplied and to account for the right number of non-zero entries a method has been implemented which works on a uniform distribution. Please see the code below for the function creating sparse matrices in all subsequent experiments.

```
public IntMatrix new_sparse_matrix(Random generator, int size, double sparsity){
        IntMatrix Matrix = new IntMatrix(size);
        boolean is_zero;
        int entry;
        for (int i = 0; i <size ; i++) {
            for (int j = 0; j <size ; j++) {
                entry = 0;
                is_zero = new Random().nextInt(100)<sparsity*100;
                if(is_zero){
                    entry = generator.nextInt();
                }
                Matrix.set(i,j,entry);
            }
        }
        return Matrix;
    }
```

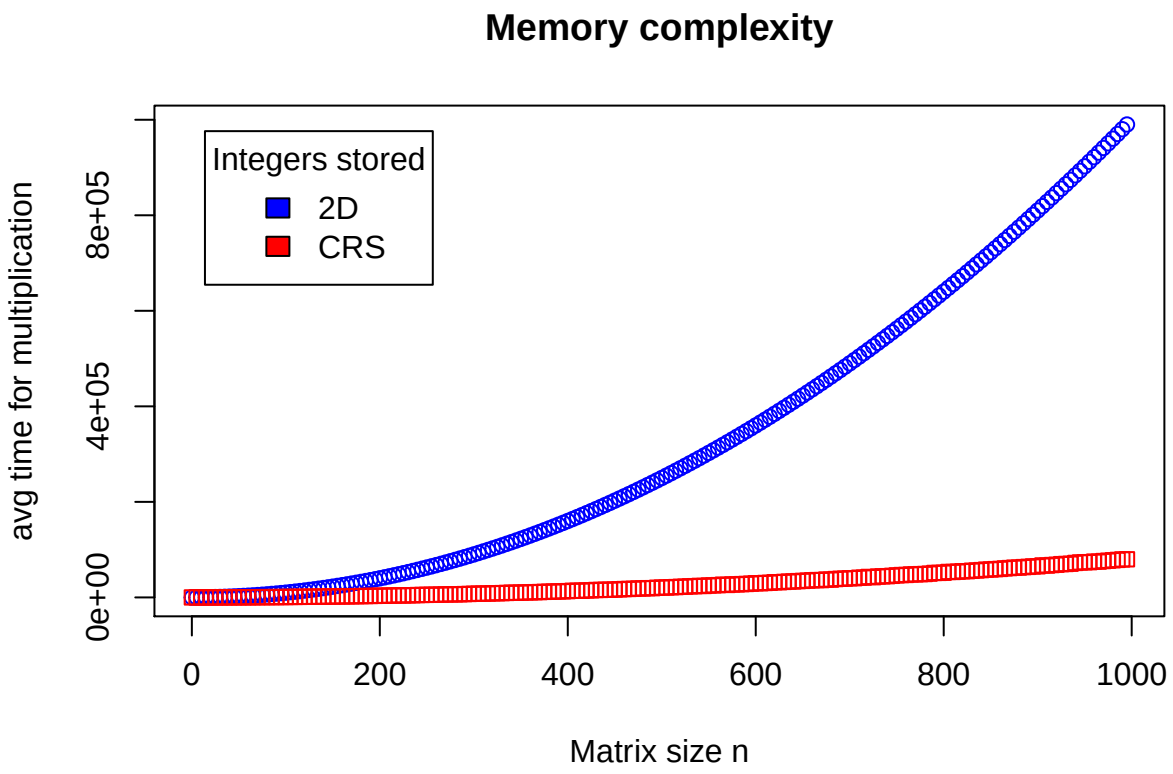
5.1 Memory complexity analysis

Running the compressed row matrices code and comparing the actual sizes on memory is fairly complicated with Java, as Java has multiple wrappers and also optimizes some types. The most intuitive version of comparing is what I rely on for this analysis: The total count of numbers which needed to be stored. Please check my class *generateNumberCounts* for the relevant code. This approach has obvious weaknesses:

1. The sizes of the arrays are not taken into account.
2. The size on memory might be different.

For larger sizes the first point becomes less relevant, so it can be discarded for larger matrices. Problem 2 is something for which there is no work around. The graph below gives a rough indication how much space both represenation need for matrices with 4% non-zero entries and increasing size.

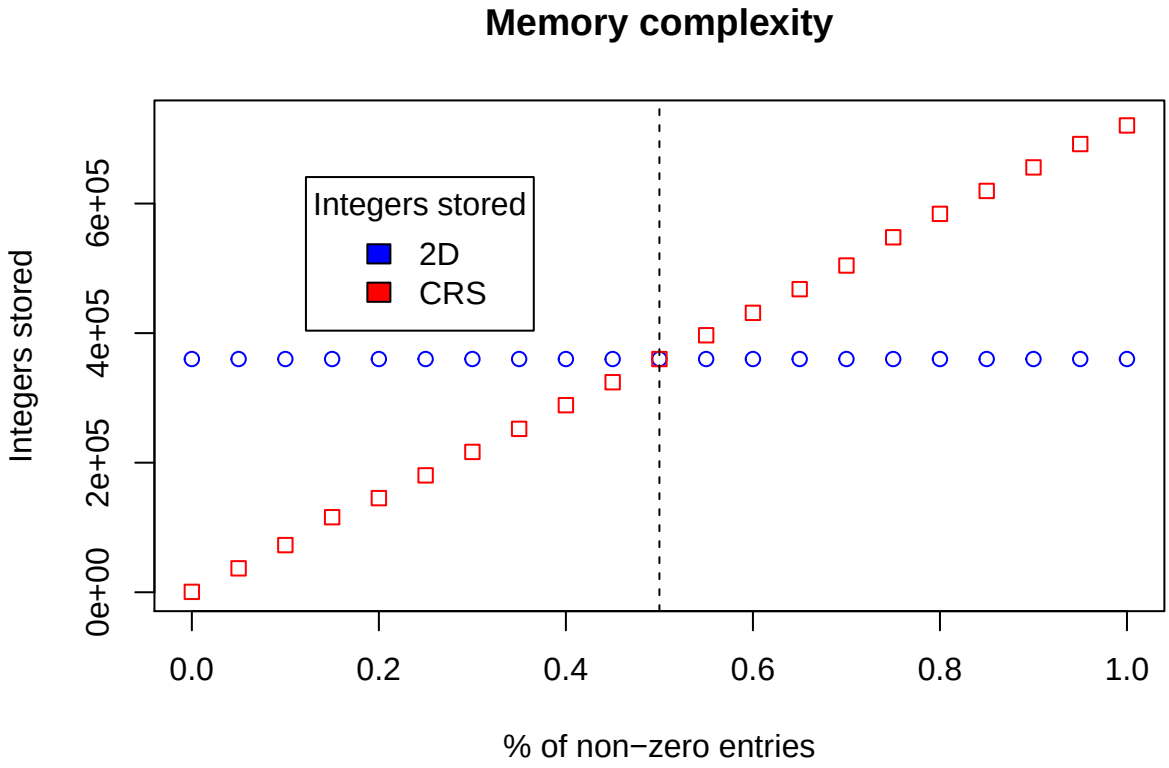
```
# data generated by generate_sizes(1234,0,1000,0.04)
df_size = read.csv("r_size.csv")
plot(df_size$size,
     df_size$numbers_stored_standard,
     xlab="Matrix size n",
     ylab="avg time for multiplication",
     col="blue",
     main="Memory complexity")
legend("topleft",
       inset = 0.05,
       title = "Integers stored",
       legend = c("2D","CRS"),
       fill = c("blue","red"))
points(df_size$size,
       df_size$numbers_stored_crs,
       col="red",pch=0)
```



Clearly this is not a actual representation of memory on the disk or during run time as my CRS representation has multiple array objects which are maintained, which also need further space in storage. However, after all this does give a feel how easy memory space can be reduced by transforming a single matrix into multiple arrays without loss of information.

Now taking size 600 as an example and increasing the share of non-zero elements, we can observe that the compress row data format will in almost 50% of the cases cases store less integers than the 2D representation.

```
# data generated by generate_sizes_sparsity(1234,600,0)
df_size = read.csv("r_size_sparsity.csv")
plot(df_size$sparsity,
     df_size$numbers_stored_standard,
     xlab="% of non-zero entries",
     ylim=c(0,730000),
     ylab="Integers stored",
     col="blue",
     main="Memory complexity")
legend("topleft",
       inset = 0.15,
       title = "Integers stored",
       legend = c("2D","CRS"),
       fill = c("blue","red"))
points(df_size$sparsity,
       df_size$numbers_stored_crs,
       col="red",pch=0)
abline(v=0.5, lty=2)
```



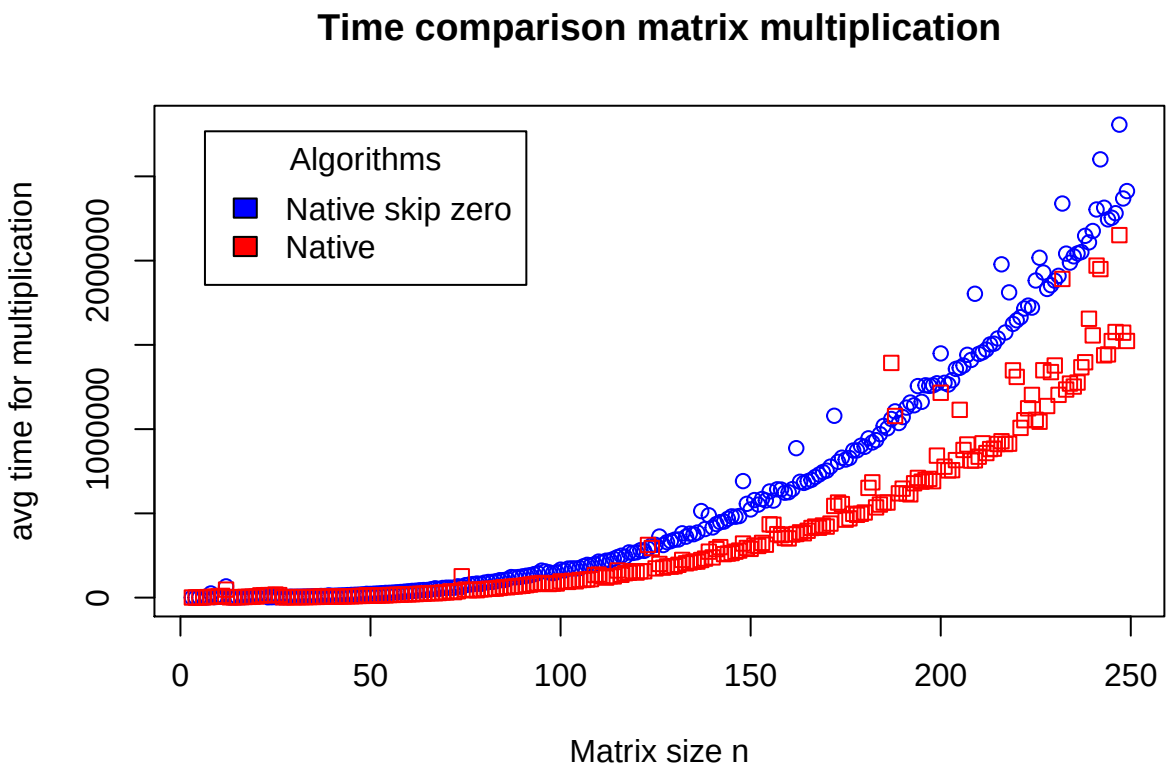
Noteworthy is the fact that for cases in which non-zero entries exceed 50% of all values, the compressed row format will in fact store more values than the 2D array representation. This can be easily be deduced from the fact that for every new non-zero value the value of the column needs to be stored, leading to two new entries in the compressed row format for every new non-zero entry and there will always be a constant added to the total number of integers, namely the dimension, representing the size of the row-pointer array. Hence the crossover point will be slightly below 50% and the 2D representation beats the compressed row format in all dense matrix cases.

5.2 Time complexity analysis

The most common approach will almost certainly be to skip a calculation when one of the integers is zero. While one might be fast to assume that this little tweak will yield an improvement of time, this assumption is in fact wrong as shown by the following experiment which operates on a level of 4% non-zero entries.

```
# data generated by generateTestData_skip_zero(1234,3,250,0.04)
df_skip = read.csv("r_basic_skip_zero.csv")
plot(df_skip$mat_size,
     df_skip$avg_skip_zero,
     xlab="Matrix size n",
     ylab="avg time for multiplication",
     col="blue",
     main= "Time comparison matrix multiplication")
points(df_skip$mat_size,
       df_skip$avg_native,
       col="red",pch=0)
legend("topleft",
       inset = 0.05,
       title = "Algorithms",
       legend = c("Native skip zero","Native"),
```

```
       fill = c("blue","red"))
```



An interesting find in this case is that for small matrices skipping zero entries is in fact faster, this changes for $sizes > 25$. This observation can be explained by considering the expected value where n is the size. For $n > 25$ the expected value is larger than 1, which means most matrices contain at least one non zero entry, which leaves the algorithm to check both for a zero and multiply which adds steps and increases the time measured. While this might surprise, it is not so difficult to explain. The three for loops will run inevitably as the if statement is inside the last for loop, which means that the time saved is only the multiplication itself, which has less significance on the running time the larger the matrix and the more zero entries there are. For all non-zero cases, an extra step is added and hence the factor of the running time is changed slightly to the worse. Hence the time complexity here is also $O(n^3)$ even for sparse matrices.

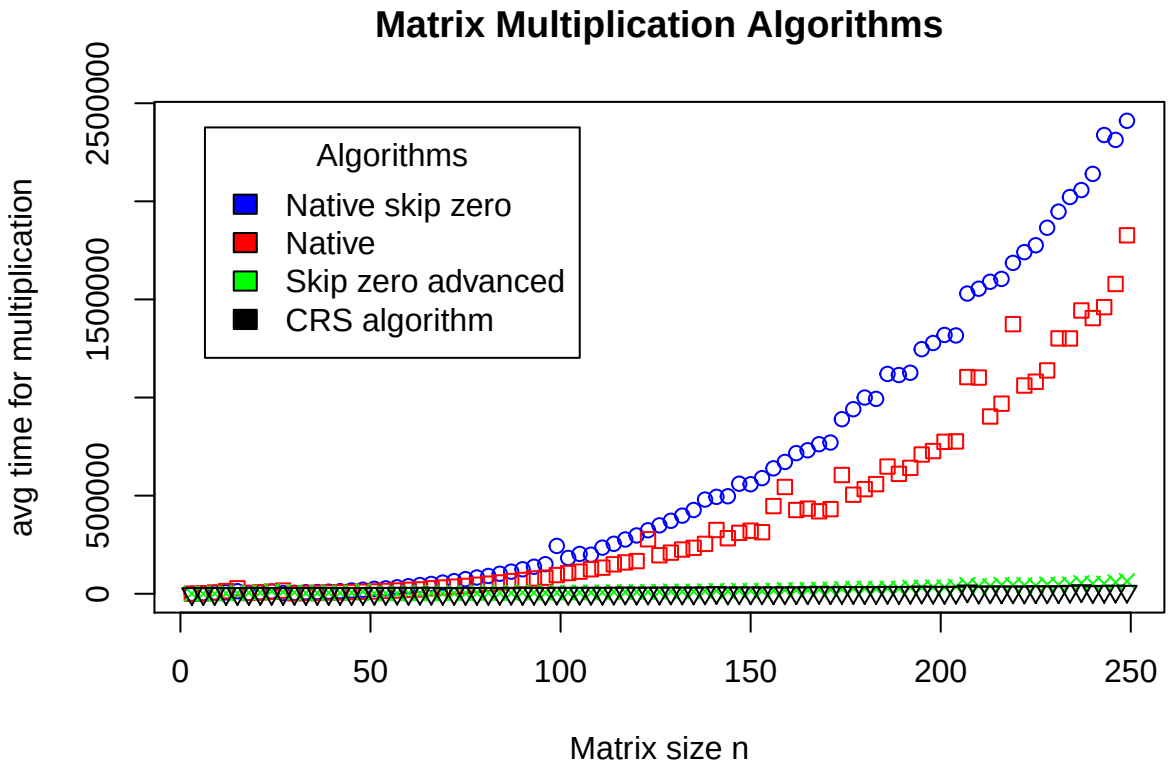
However, once the columns are switched and the result of the matrix muliplication is build in a different fashion; the running time improves massively and simiarly the algorithm for CRS matrices offers a strong improvement as shown below:

```
# data generated by generateTestData_all_alg(1234,3,250,0.04)
df.skip.adv = read.csv("r_all_alg.csv")

plot(df.skip.adv$mat_size,
     df.skip.adv$avg_skip_basic,
     xlab="Matrix size n",
     ylab="avg time for multiplication",
     col="blue",
     main="Matrix Multiplication Algorithms")
points(df.skip.adv$mat_size,
       df.skip.adv$avg_native,
       col="red",pch=0)
points(df.skip.adv$mat_size,
       df.skip.adv$avg_skip_advanced,
```

```
       col="green",pch=4)
points(df.skip.adv$mat_size,
       df.skip.adv$time_in_ns_crs,
       col="black",pch=6)

legend("topleft",
       inset = 0.05,
       title = "Algorithms",
       legend = c("Native skip zero","Native","Skip zero advanced","CRS algorithm"),
       fill = c("blue","red","green","black"))
```



In this graph all three suggested improvements of the algorithms are shown. The native skipping approach can be discarded from now on as it doesn't offer any further interesting insights. However, the advanced skipping zero algorithm and the CRS algorithm both show a substaintal improvement over the native matrix multiplication algorithm.

For the row wise skipping zero algorithm the exceution time is massively reduced, which can be deduced from two factors, firstly, a whole loop iteration of the size of the dimension can be skipped. Note that the experiment was run with a 4% nen-zero entries setting, this means that for roughly 96% of all cases the inner loop and the multiplication can be skipped. This makes then algorithm follow a quadratic polynomial in 96% of all cases, which ultimately leads to a worst case complexity of $O(n^3)$ for dense matrices as all three loops will be exceuted, but an average time complexity of $O(n^k)$ with $k + t = 3$ with $t > 0, 3 > k > 0$ and t very small for sparse matrices. The fewer non-zero elements the matrices have, the better for the execution time.

For the CRS algorithm the case looks slightly different. Let's assume a matrix with dimension m and n non-zero entries for matrix A and k non-zero entries in Matrix B in $A * B = C$. Taking a closer look at the implementation yields the following observation:

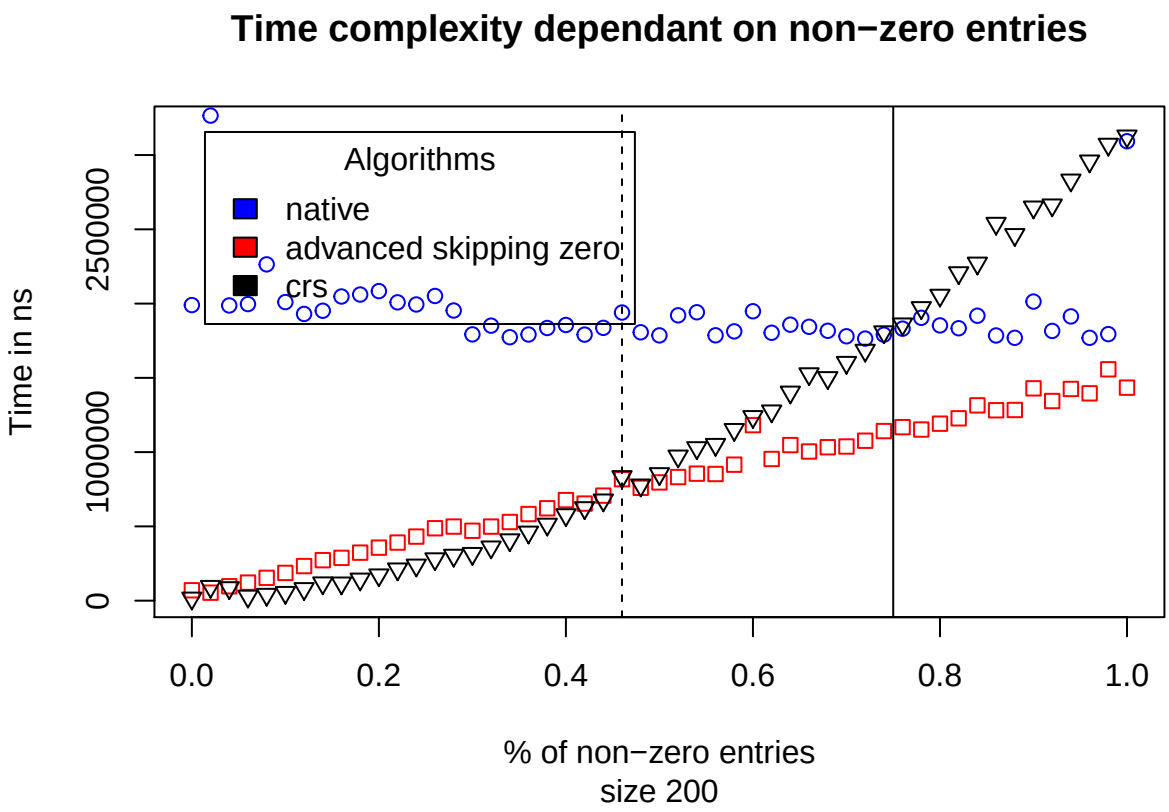
1. The outer loop has time complexity $a * m$, where $a = const$.
2. The middle loop has time complexity $b * t$ where $b = const$ and $n \geq t \geq 0$.
3. The inner most loop has time complexity $c * r$ where $c = const$ and $k \geq r \geq 0$.

4. Let the multiplication and memory access take d where $d = const$.

This can be merged into $O(d + (a * m) * (b * t) * (c * r)))$. This becomes $O(m * t * r)$ when the constants are dropped. However, we know that in the worst case $m = t = r$ which would lead to the time complexity for the native algorithm, the average case for sparse matrices looks different. In this case it can be assumed that $t << m$ and $r << m$, which leads to a time complexity smaller than $O(n^3)$.

It is worth noting that both the CRS and the advanced skipping algorithm are highly dependant on the sparsity of the matrices. The graph below shows the significance of this parameter. All times have been calculated using matrices with size 200.

```
# data generated by generateTestData_time_sparsity(1234,300,0.00)
df_time = read.csv("r_time_sparsity.csv")
plot(df_time$sparsity,
     df_time$time_in_ns_native,
     xlab="% of non-zero entries",
     ylim=c(16000,3200000),
     ylab="Time in ns",
     sub="size 200",
     col="blue",
     main="Time complexity dependant on non-zero entries")
legend("topleft",
       inset = 0.05,
       title = "Algorithms",
       legend = c("native","advanced skipping zero","crs"),
       fill = c("blue","red","black"))
points(df_time$sparsity,
       df_time$time_in_ns_advanced,
       col="red",pch=0)
points(df_time$sparsity,
       df_time$time_in_ns_crs,
       col="black",pch=6)
abline(v=0.46, lty=2)
abline(v=0.75, lty=1)
```



This plot highlights interesting features of the algorithms involved. Firstly, the native algorithm is not dependent on sparsity at all, which is not surprising considering it will iterate through all entries no matter what. The advanced skipping zero algorithm on the other hand is asymptotically growing towards the native case the more non zero entries there are. It is important to highlight, that on average the advanced skipping zero algorithm will beat the native algorithm given with the practical as the native algorithm doesn't follow a pure row based calculation as established above.

The CRS algorithm is highly reactive to non zero entries in the multiplication involved. At around 46% non zero entries it looses its edge compared to the skipping zero algorithm (please see the dashed line). This is can be deduced from the fact that the arrays in the CRS object are scattered around memory and the access times start to outweight the slight advantages of the CRS algorithm. At around 75% non zero entries the CRS algorithm also starts to perform worse than the native approach. This is can be deduced from the fact that the CRS algorithm has more read and write proccess across arrays, so even though the time complexity is $O(n^3)$ for dense matrices the involved constants force it to perform worse than the native case.

6 Conclusion

The native algorithm is may seem like an easy way to multiply matrices, but it turns out that even changing the order of the loops has an extensive impact on running times without changing the computational complexity. However the average case complexity can easily be improved by adding a check for a zero to the algorithm. This concept works best for sparse matrices but will have an edge over the native approach for almost completely dense matrices. Compressing a sparse matrix is a good and effective way to save memory space up to matrices with around 50% non-zero entries. The advantages become more extreme the fewer non-zero entries there are and the bigger the matrix is. Multiplying CRS matrices is extremely fast if the user is happy with a non-compressed result, however, as soon as it needs to be compressed the running time will increase. Another approach to the one in this practical is to guess the expected non-zero values and compute straight into a new CRS matrix. Overall, it was shown that there are many possebilities to effectively improve storage and computation of sparse matrices.

7 Sources

[1] Gundersen and Steihaug, 2002, Data Structures in Java for Matrix Computations, University of Bergen

[2] https://www.programcreek.com/2014/10/leetcode-sparse-matrix-multiplication-java/