3-Way Matching Problem Report

Nadeem Ahmad, Matthew Borkowski, Yousef EL-Qawasmi, Hatim Khan

Department Of Computer Science, Wilfrid Laurier University

CP493/CP395

Dr. Ilias Kotserias

April 26, 2023

Table of Contents

1. Preface

As organizations and individuals continue to accumulate and store massive amounts of data, the need to process and analyze this data efficiently and accurately has become increasingly pressing. Complications with processing large sets of data have been a significant area of research in computer science in recent years, as new technologies and methods are constantly being developed to compete with the sheer volume and complexity of data.

One important problem in big data processing is the 3-Way Matching Problem, where three text files are given containing billions of digits and at least one row in each text file sums to a given value lambda, λ. This problem has important implications in cryptography, coding theory and data analysis, as it is a fundamental problem in combinatorial optimization and has many practical uses in real-world scenarios.

In this research report, we present a detailed study of the 3-Way Matching Problem and propose a new approach for solving it efficiently in the context of big data processing. Our approach is based on a distributed algorithm using specialized hashing data structures in conjunction with multi-threading. The algorithm is designed to efficiently process large datasets and find all possible solutions to the 3-Way Matching Problem and improvements in processing large sets of data.

Our proposed approach has important implications for the development of more efficient algorithms for big data processing. It can be extended to other similar problems in combinatorial optimization and has the potential to be used in various practical applications. This research contributes to the ongoing efforts to advance the field of big data processing and to enable organizations and individuals to enable organizations and individuals to analyze large datasets and make more informed decisions.

2. Abstract

The 3-Way Matching Problem is a fundamental problem in computer science that involves parsing three text files to find a row in each that adds up to a given value λ. This problem has important applications in many areas, including cryptography, coding theory, and data analysis. In our research, we propose a novel approach for solving the 3-Way Matching Problem in the context of big data processing. The algorithm involves iterating through the different data sets and utilizes techniques such as hashing to store and manipulate the data. Its process is run for all the columns of the given files to identify and store all valid solutions found in the data sets. Our proposed algorithm is designed to efficiently process large datasets and find all possible solutions to the problem. The experimental results show that our proposed approach is highly efficient and can solve the matching problem for large datasets. The approach can also be easily extended to other similar problems in big data processing.

In conclusion, our proposed approach provides a novel solution to the matching problem in the context of big data processing. The approach is efficient, scalable, and can be easily extended to other similar problems. The results of this research have important implications for the development of more efficient algorithms to other related problems in data analysis and optimization.

3. Introduction

3.1 Problem Introduction

The 3-Way Matching Problem exists in number theory involving an integer constant λ and three files A, B and C with *k* columns and *n* rows and *x* elements, where $k \leq 50$, $\lambda < 1000$ and $x \leq 1000$. The goal of the problem is to produce an output which consists of three number lines L_A, L_B, L_C in files A, B, C such that

$$L_A[0] + L_B[0] + L_C[0] = \lambda, \dots, L_A[k] + L_B[k] + L_C[k] = \lambda \quad (1)$$

3.2 Research Objective

The 3-way matching problem in computer science can be illustrated with the following example: Suppose we have a block of memory with a size of 1000 Bytes. We want to allocate this memory to three different programs, where each program requires a specific amount of memory. Let's say program A needs 300 Bytes, program B needs 400 Bytes, and program C needs 300 Bytes. We must divide the block of memory into three parts of specified sizes such that each program gets the amount of memory it needs. In this case, we need to find a solution that allocates 300 Bytes to program A, 400 Bytes to program B, and 300 Bytes to program C, while also ensuring that the total allocated memory does not exceed the total available memory of 1000 bytes.

The 3-Way Matching Problem is a case in a class of given matching problems. Examples for other problems in this class would be 2-Way Matching and 4-Way Matching, both of which have existing solutions. Various approaches have been proposed to solve the 3-Way Matching Problem including recursion, dynamic programming, hashing, parallel backtracking, and binning. These approaches have been shown to be effective in generating all possible solutions for small sets of integers, but their efficiency decreases rapidly as the input size grows. Therefore, developing efficient algorithms for solving the matching problems is an area of research that requires attention as the amount of data in the world continues to increase.

In this report, we present an algorithm for solving the 3-Way Matching Problem based on a hashing approach. The algorithm iteratively goes through each row in the files, one column at a time. The approach allows us to significantly decrease the amount of lines in each file by weeding out lines which have integers which do not match with any other corresponding lines from other files. This allows the file sizes to be quickly shrunk on each iteration which, in turn, allows for the processing speed to increase significantly over time.

4. Previous Approaches

Several approaches have been proposed for solving this problem including dynamic programming, backtracking, and branching algorithms. However, these approaches can be computationally expensive and may not be suitable for large datasets. In this context, a new method based on hashing algorithms has been proposed for solving the 3-Way Matching Problem that offers a promising alternative for solving matching problems and can provide important insights for future research in this area.

4.1 Initial Approach

One possible solution we devised was the Hash-Threading Algorithm. The idea behind the algorithm was to use a combination of hashing and multi-threading to find potential line solutions within a file within a reasonable computation time. This was done by creating a 2D array which serve a similar purpose as a hashmap. The array would be used to store all the first column value indexes for File B. For example, the integer 12 found at position 7 in the column would be stored in the array located at index 7. This would give us the benefit of O(1) lookup and insertion for all elements.

The algorithm would continue to iterate through the first column of file A and C each in a nested manner to allow us to find all combinations of A and C. We could then take these values of the A and C columns and determine whether or not a potential B value existed or not with the following equation.

$$(\lambda - C[i]) - A[j] = B[k] \tag{2}$$

If the B value found by the equation existed in the hash that was created earlier, then a potential group of rows had been found which could serve as a solution to the problem. The indexes of each line were stored in a tuple for later use. Once all possibilities had been processed and stored, we would then use multi-threading to analyze each of the sets of lines. A tuple would be passed to each thread to determine if the three lines indexed by the tuple would be valid solutions. In the case that it was found a valid solution did not exist, the tuple was to be ignored and the thread was released to continue processing other rows. This approach presented a time complexity of O(n^2) initially and showed some promise with initial testing.

Issues arose further in testing as the size of the inputs increased rapidly and the algorithm was struggling to handle the increased workload. For example, when the file sizes of A, B and C were set to 92,000 rows, 72,00 rows and 72,000 rows respectively, over 40 billion potential combinations were found. If an assumption is made that the number of combinations increases in a proportional manner, file sizes approaching 10 million would have over 4.32*e*+12 potential combinations to verify. Even with multi-threading this would prove to be extremely inefficient. Furthermore, with further testing it was determined that the current rate of the algorithm would lead to multiple years of iteration to store information. It would be unreasonable to use the algorithm in any manner.

Several attempts were made to improve the processing time, but regrettably, they yielded limited success.. The amount of combinations required and the sheer quantity that was produced could not be overcome. The idea was finally abandoned when previous research titled *Reducing 3SUM to Convolution-3SUM* (Chan & He, 2020) revealed that a hard floor had been hit in terms of time complexity. deals with a very similar problem to 3-way matching called “3sum”. The best time complexity for an algorithm to solve the 3-sum problem was found by (Chan, 2019) which was $O((n^2/log^2 n)(log\ log\ n)^{O(1)})$ which reduces down to O(n^2). This revealed to us that there was no current method available to reduce the algorithm by degrees of magnitude and only marginal improvements could be made. For this reason this approach was abandoned and others were tested.

4.2 Pre-processing Approach

With the failure of the previous algorithm, a new one was designed with many of the old elements in an attempt to deal with the bottleneck issues that were previously faced. A pre-processing step was first implemented before any calculations could be made. The first columns of file A, B and C were taken and had all their values modulo 10 so that only the last digit of each value would be retained. This left us with three column elements that are less than or equal to 9 which was a significant improvement compared to the previous 1000 remaining elements. The numbers from the first columns would each be placed into separate two dimensional arrays with the element acting as the index and the element's position in the column the value being stored. With this, we would be able to see where each element is stored. For example, if you check the second index and find the string “4,32,17” you would know that element two was stored at indexes 4, 32 and 17. We would then go through the same process as the previous algorithm and run a double for loop that loops through all the values of the array that was made with the A and C files. Due to the pre-processing steps that were conducted, these arrays would only be of size 10, meaning that the loop will be almost instantaneous.

$$B = (10 + \lambda - A[i] - C[j])\%10 \tag{3}$$

The above equation then was used to get potential matches of indexes which may have the answer. This new equation follows the same principle from (2) except that it has to account for the pre-processing that was done and the carry any digits that came with it. In the case that the B value found is not an empty index on the B index the values were to be stored in a tuple which was also subsequently stored in an array for later use.

Another issue with this approach occurred when following branches to potential solutions. When run on the files provided we were left with 4 tuples. To explore all the valid paths permutations were to be done on each of the strings to index the A, B, and C values within each tuple. The permutations would be required for this which would require O(n!) runtime. Considering on average each of the individual strings would have an average length of 9200, just one tuple there would be over 7.78688*e*+14 possibilities. Even with multi-threading this would be extremely difficult to consider as a viable solution. Symmetry breaking was explored to test for opportunities to remove duplicates or redundancies in the solution, in order to bring down the run time, but did not help much with the problem that we faced.

If the permutation bottleneck was dealt with, a simple multi-threaded approach can be made to explore each path. A permutation would be taken and passed to a thread, and a concept of “rounds” would be implemented.

• Round 1: Check if the middle values the A row, B row and C row to see if they equal to the given λ. If they pass the round, the values are stored for the second round. If they do not pass the condition, the combination is discarded.
• Round 2: Check if the value from the middle to the end of the given A row, B row and C row to see if they equal to the original λ. If they pass the round and are stored for round two. If they do not pass, they are discarded
• Round 3: Check if the middle value from the start to the middle of the given A row, B row and C row is equal to the original λ. If they pass the round and are stored for round two. If not, they are discarded.

The round system can be repeated as many times as needed with as many rules as required. It would allow for the filtering a majority of the false positive row combinations. After all iterations, the threads can be run one last time to check all values of the few remaining row combinations. A match will be found if all the values sum to λ for a given row combination.

5. Current Algorithm

The current algorithm differs from previous attempts whenever combinations were calculated and then subsequently threaded. There was a tendency to produce such a large amount of possibilities and would very quickly go outside of the scope of what our computers were capable of. The new approach was an iterative process which aimed to do an approach in rounds, where each round was used to reduce the total number of rows in the file. This would have some heavy initial cost, but would lead to subsequent rounds being faster. This would allow for the algorithm to narrow down the number of potential solutions in a reasonable amount of time.

The first round would consist of selecting the first two columns from files A, B, and C. As previously mentioned, each would have the indices of its numbers stored in a hash of size 1999 with the elements acting as the keys to the hash. The pseudo hashes size was 1999 due to an assumption that the elements found within the files would be $-1000 \leq x \leq 1000$. These columns were nested loop through and used equation (2) to find all the potential paths for each. The tuples of elements were then stored for further processing. This would lead to two arrays being created which would then be further processed. Each array would have its tuples opened A, B, and C indexes separated into separate groups. The values in the hash that they would represent would then be retrieved and be placed into a set. This would be repeated for the second array, leaving 6 total sets 1 for column 1A, 1 for column 2A, 1 for column 1B, 1 for column 2B …etc.

We would then take the union of these sets as:

$$potential\,A = A1 \cup A2 \quad (4)$$

$$potential\,B = A1 \cup A2 \quad (5)$$

$$potential\,C = A1 \cup A2 \quad (6)$$

This would be the result at the end of the first round. The second round would follow the same process as the first, except column 3 would be used and the end column 3’s potential sets for A, B and C would be unioned with potential A, potential B and potential C to create a new set. The third round would take column 4 and continue the process until all columns were completed. In the end, the potential set would be used to go through the files A, B and C and create a new file based on the values that were stored in the potential set of each file. This would allow for a massive reduction in rows, making the files far easier to compute.

The approach was to use a second separate iterative strategy on the new set of files. Here, the first two columns would again be taken and hashed with equation (2) to check for valid values in the B file. The difference in the approach was that instead of checking for valid values separately, the process was done together. A match was only found in the case that both the given file's columns produced a result. An example can be seen below:

$B = \lambda - a[i][0] - c[j][0]$ ✓

$B = \lambda - a[i][1] - c[j][1]$

Whereas:

$B \neq \lambda - a[i][0] - c[j][0]$ ✗

$B = \lambda - a[i][1] - c[j][1]$

If a valid match was found the rows were added to a new set of files, thus filtering out any rows where there were not any matches.

Once this was completed for the first two rows, the process was then repeated on the first three of the new file, then the first four, and continue through the file. In theory, this would provide a viable solution to the three way matching problem.

In theory this second portion of the overall algorithm should work to continue and reduce the number of lines in a file until the only ones that are left provide the answer. Currently, there is an error that is instead causing all lines to be removed from the file. One thing to note is that although there are many steps involved in the code, the algorithms themselves are very quick to run. This is due to the heavy uses of sets and hashes, which helped bring down the time complexity significantly.

6. Results

We conducted experiments to evaluate the effectiveness of our proposed current algorithm. The main files that were used in testing were A, B and C files of lengths 92,000, 72,000 and 72,000 respectively. This was done because it was a large enough sample size to test to see if the algorithm was functional, without increasing the wait times that come with the execution time issues with larger files.

When the first round was run, there was a 99% reduction in the size of the files from 92,000, 72,000 and 72,000 to ≅ 2000, 700 and 700 respectively. This was extremely promising as it allowed us to get over the first round speed bottleneck. The second algorithm introduced a new issue when the first two columns were used as the filter. It would decrease the number of lines from all the files to 0, which was not a valid response. The reason for this bug's appearance has not yet been found, but we heavily believe that the theoretical portion of the algorithm is sound, with the problem being on application of the practical side.

Another recurring issue is with memory leakage. With larger files, the algorithm has a habit of slowing down and crashing the system it is running on. Thankfully, this is a trivial fix as when the code is ported over to C, we will have the ability to manage memory, preventing any issues. For this reason, the algorithm could not be tested against larger files.

Larger files were taken and broken down into 100,000 row sections to allow for multiple different testing sets. With each set that was tested, on average around a 90% reduction was seen after the first round was completed. Unfortunately, the same bug plagued all second round attempts regardless of the set. This is still fine as the first algorithm alone can be used on future files of the same type to consistently reduce file sizes for further processing down the line.

The algorithm's design gives it the potential to be extended to similar problems. For example, if 5 files need to be processed for a common sum, the first algorithm can be used to reduce file sizes to allow faster processing by other methods.

7. Future Directions

Moving forward, we need to address several key issues. The most pressing issue is a bug that causes all rows to filter out in the second algorithm. The error may be due to a string manipulation issue or an indexing error, leading to valid rows being eliminated from the pool. Fixing this error is the top priority, as without that the code is only good for removing large amounts of unwanted rows from the files. Once found, a solution would be very straightforward to find for reasonable file sizes of 1,000,000 - 10,000,000 rows.

Another area of improvement which was briefly mentioned earlier was that of converting the python code to C. The reason for this is due to both speed and memory. With C, the algorithm will run significantly faster as compared to python which takes far longer to do the same amount of work. C would also mean that multi-threading could be introduced creating further speed improvements. This would most likely be implemented in each round, where instead of processing the whole column, each thread would only process a small section of the given column and then combine their results in the end. Using this with the memory management freedom in C would allow a far faster and more robust program as compared to what can be done in python. However it should be noted that this change may require significant time and effort.

Finally, after this is completed the program can then be generalized for m-way matching problems. The approach that was used could be extended m-way's, meaning that many different future problems could be solved using a more general algorithm. If done correctly, the final algorithm would require little to no change to complete these problems in an efficient amount of time.

References

Chan, T. M. (2019). More logarithmic-factor speedups for 3sum, (median,+)-convolution, and some geometric 3sum-hard problems. *ACM Transactions on Algorithms*, *16*(1), 1–23. https://doi.org/10.1145/3363541

Chan, T. M., & He, Q. (2020). Reducing 3SUM to convolution-3sum. Symposium on Simplicity in Algorithms, 1–7. https://doi.org/10.1137/1.9781611976014.1