Three actors playing together in most movies

Plamen Kmetski (pkme)
Kamil Androsiuk (kami)
Agnieszka Majkowska (amaj)

December 15, 2013

Abstract

Large data is difficult to process. The main difficulty comes from the fact it is not possible to store the data in its entirety in memory and directly manipulate it. A number of algorithms, each of them with its own strategy, are designed for working around this problem. One such strategy offers an approximation of the result with the advantage of using little memory. Alternatively, another strategy splits the data into bits that can be computed independently. In this paper we explore two such algorithms, comparing their running times and space usage. The first is the Misra-Gries streaming algorithm. The second algorithm, designed by us, is specifically targeted for solving the problem at hand. This makes it possible to apply some seemingly small tweaks in graph processing, which increase performance dramatically. Furthermore, the concept of parallelism is explored in the context of our solution. It is shown that with very little additional code the sequential algorithm can be easily parallelised.

Keywords

IMDB, graph, triplets count, streaming, Misra-Gries, parallelism

1 Introduction

The IMDB dataset provides data about actors, movies and relations between them. Based on this dataset a number of interesting properties can be extracted from the data. The point, however, is doing this in efficient manner.

In section 2 we examine the problem, its input and expected output. In section 3 we look at a standard approach for solving the problem - using the well known Misra-Gries streaming algorithm. We present the running time and space usage for various input sizes. In the next section we turn our attention to our algorithm that improves on the shortcomings of Misra-Gries. Further, section 4.4 explores a way of improving our algorithm processing time by running it in parallel. In section 4.5, we compare the two approaches, their running time, space usage and how they differ.

2 Problem Description

The **goal** is to find the three actors, whose movie count they have played together in is maximized among the whole dataset.

Input is a list of actors, movies and actor-movie pair, for each actor that has played in a movie.

Output is a list of actors with the desired property, or an empty list if no three actors have played together in the same movie.

3 Standard approach

We started with implementing a data streaming technique. Since our goal is to find heavy hitters, we chose Misra-Gries algorithm to save space and to process large amount of data fast.

3.1 Pseudocode

The idea of our version of Misra-Gries is to process data movie by movie. Each movie has a list of actors and while processing them we generate all triplets for each movie. Then we maintain a hash table which contains pairs of triplets and their respective count indicating relative number of occurences of the triplet. We add a triplet to the table if it does not appear there yet or we increment count of analyzed triplet. The result of the analysis is the triplet with the highest count. For more details we present pseudocode of the algorithm in paragraph below.
Let m denote number of movies and A(i) denotes list of actors playing in a given movie i. Let H be a hash table containing k pairs: triplet (t) and its count ($count_t$).

```
Algorithm 1: MisraGries()
1 FOREACH movie DO
2   FOR i ← 0 to size of A(movie) DO
3     FOR j ← i + 1 to size of A(movie) DO
4       FOR k ← j + 1 to size of A(movie) DO
5         IF {A(movie)[i],A(movie)[j],A(movie)[k]} ∈ H THEN
6           count[t] += 1;
7         ELSE
8           INSERT({A(movie)[i],A(movie)[j],A(movie)[k]}, count);
9         END IF
10        IF k < H.length THEN
11        FOREACH t IN H DO
12          count[t] -= 1;
13           IF count[t] = 0 THEN
14             H.REMOVE(t);
15           END IF
16         END FOREACH
```

```
17        END IF
18      END FOR
19    END FOR
20   END FOR
21 END FOREACH
22 RETURN H.MAX(count);
```

The algorithm allows to save memory since not store all processed data is stored. The efficiency and its running time highly depend on the cache size. The bigger it is, the slower algorithm computes and the more memory space we use. On the other hand with bigger table, we can work with bigger sample, which would result in an approximation that is closer to the actual answer.
The cache size was determined based on calculations and experiments. We estimated that with IMDB database we have nearly 2 billion triplets for all the movies. We ran multiple expreiments with different cache sizes, and the number we found most acceptable in terms of running times was 20 000 triplets. This represents 0.001% of the whole set. While the sample is relatively small, the running time with any bigger cache size would have been unacceptable. More on the running times in section 4.5

3.2 Analysis

Memory usage in Misra-Gries algorithm is very little, the space is needed only for the hash table. Complexity of the algorithm highly depends on the size of cache and the dataset to be processed. The bigger they are, the longer computation time is needed. The complexity of the algorithm is $\sum_{i=1}^{m} \binom{a_i}{3} * k$ where k is size cache size. We can reduce it to $\sum_{i=1}^{m} a_i^3 * k$.

4 Algorithm

The algorithm we are presenting works on two main steps:

- Build the data structure - the efficiency of the algorithm is determined by the data structure it runs on. On the other hand, the data structure is specifically designed to solve this problem.
- Traverse data structure and output result - the algorithm works by looping through all the actors and finding the most promising connections.

4.1 Notations

Let G = (V,E) be a weighted, directed simple graph and let n = $|V|$ and m = $|E|$.

A vertex v denotes an actor. Any edge e between vertices v_1 and v_2 denotes a set of movies these two actors have played together. Weight of the edge, W(e) denotes the size

of that set. An edge is always directed from the actor with lower Id to the actor with higher Id.

Denote by A(v) the set of adjacent edges to vertex v.

SET(e) is the set of (two) vertices adjacent to an edge e.

Unique$(v_1, v_2, ...v_n)$ - returns a set of unique elements.

MovieCount denotes the biggest number found so far of common movies between any given three actors.

4.2 Data structure

As mentioned in the previous section, the algorithm starts by first building the data structure. The data structure is a graph, where vertices represent actors and edges between them represent the movie(s) these actors played together in.

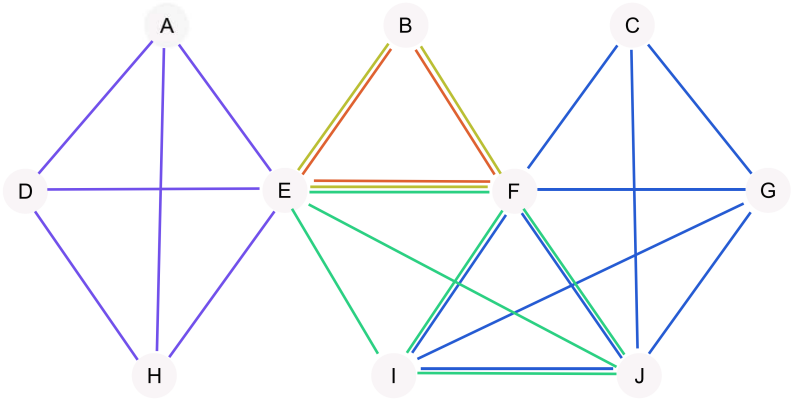

Figure 1: Graph Example

Figure 1 represents the data structure. To clearly illustrate the problem, the picture above represents a multi-graph, i.e. there can be multiple edges between two vertices. This is not the case of the actual data structure, however, as multiple edges are collapsed into a single one, where the weight is the sum of the weights of the original edges. The weight of an edge is the initially 1 - a single movie common to two actors (vertices). The edge contains sorted list of the movies common for two actors adjacent to the specific edge.

After the structure is constructed, the actual data processing takes place.

4.3 Pseudocode

Main algorithm we implemented is FindThreeActors. After constructing the graph, we iterate over all vertices (actors). We find for each vertex all subsets of size of two of the set of adjacent edges to that vertex. We take an advantage of the fact that if an actor 1 played together in the same movie "A" with an actor 2 and the actor 1 played together with an actor 3 in the movie "A", that means that the actor 2 and the actor 3 had to play together in the movie "A". So we decided not to look for triangles in traditional way, but we look for pairs of edges having movies in common being adjacent to a specific vertex.
We examine each pair of edges and we find the number of common movies that actors played in together (this is a common set of the subsets - the movies from two edges). We do that only if minimal weight of edges is higher than found (by now) maximum number of movies that actors played together.
If the solution is better that the already found, we save it. We continue searching and at the end, we remove any references to the adjacent edges to the analysed vertex. This will make future iterations faster, since less edges need to be examined. We go to the next iteration.
Pseudocode for the algorithm is presented below:

```
Algorithm 2: FindThreeActors(graph)
1 moviesCount ← 0;
2 {a1,a2,a3};
3 FOR v ∈ V DO
4   FOR i ← 0 to size of A(v) DO
5     e1←A(v)[i];
6     IF MoviesCount < W(e1) THEN
7       FOR j ← i + 1 to size of A(v) DO
8           e2←A(v)[j];
9           IF MoviesCount < W(e2) THEN
10               count ← CommonMovieSubsetCount(e1, e2);
11              IF moviesCount < count THEN
12                   movieCount ← count;
13                   {a1,a2,a3} ← Unique(SET(e1), SET(e2));
14              END IF
15           END IF
16       END FOR
17     END IF
18   END FOR
19 END FOR
20 RETURN moviesCount, {a1,a2,a3};
```

We designed CommonMovieSubsetCount that returns number of items that are common in 2 subset given as arguments. We take advantage of the fact that the lists are

sorted and we iterate over all the items in both lists in linear time.
The pseudocode is present below:

```
Algorithm 3: CommonMovieSubsetCount(movies1, movies2)
1 count ← 0;
2 p1 ← 0;
3 p2 ← 0;
4 WHILE p1 < W(movies1) AND p2 < W(movies2)
5   IF movies1[p1] = movies2[p2] THEN
6     INCREMENT(count);
7     INCREMENT(p1);
8     INCREMENT(p2);
9   ELSE IF movies1[p1] < movies2[p2]
10    INCREMENT(p1);
11  ELSE IF movies1[p1] > movies2[p2]
12   INCREMENT(p2);
13 END IF
14 RETURN count;
```

4.4 Parallelism

Since the IMDB database is enormous and building graph for such an amount of data, requires lots of RAM, we wanted to split the problem and make computations on separate parts in parallel. We took care of not loosing any data and not processing any data more times than once. The other important thing was to split data into parts that require the same amount of work for CPU.
Each separate group contains specific number of actors and edges coming out from them. The division is done by creating a list of edges (an edge contains 2 actors and a movie these 2 actors played together), where there is no repetition of edges (an edge is always directed from an actor with higher Id to an actor with lower Id). We sorted the list according to the Id of the first actor (in this actor with lower Id in an edge). We noticed that since an edge goes to the actor with higher Id, vertices with lower Ids have much more edges going out from it than getting in, so we could not just picked the first x actors from the list.
For instance if we want to have ten groups, to split data evenly in each group, we just picked every 10th vertex (actor) from the sorted list we have. That guaranties that workload on every part of data is similar, since each vertex has less and less edges coming out from it.
We claim that we do not process same data multiple times. It is shown on the drawing 2.

All three vertices are connected to each other and are placed in three different groups (red, blue and green). In group red we have a vertex A and two outgoing edges. In blue group we have a vertex B and one outgoing edge. In group green we have only vertex C without any outgoing edges. We can see that the presented triple will be analysed only

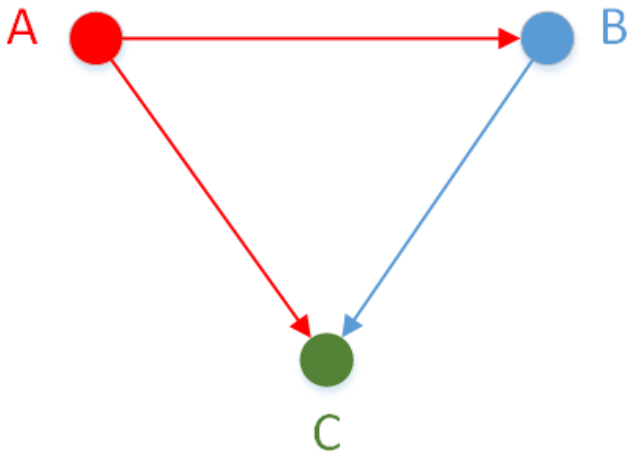

Figure 2: Vertex Triple

once when we will process the vertex A in group red.
The presented division allows us to run computations in parallel. Each thread results in a triple and number of movies actors played together in. After finishing computing all the data, we just need to aggregate the results and find the triple that played in the most number of movies. The figure 3 models the aggregation done by process 0 of the work produced by processes from 1 to n. In our case depth is 1 and work is n.

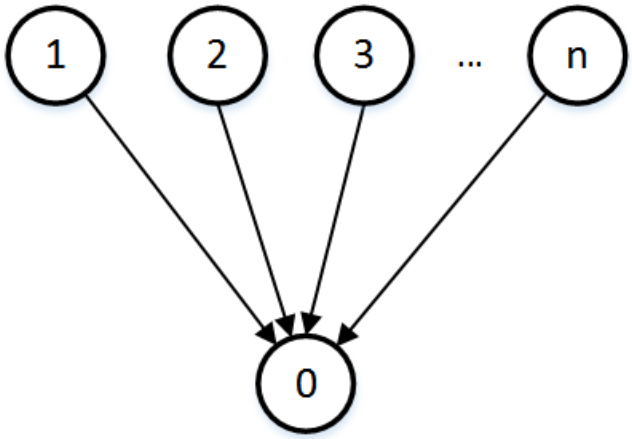

Figure 3: Threads Work Aggregation

We can run parallel computations on the same machine or on the separate machines. When we want to run it on the same machine, we have to consider amount of RAM the machine has and amount of parallel processes we can run on it.

4.5 Analysis

By not choosing to implement a traditional triangles counting $O(n^3)$, we avoided to examine three times the same vertex. To do so, we constructed a special data structure and by consolidating edges, we saved used space to save the data. In the worst case scenario, when each actor played together with another one (it is not realistic situation though) we will have n vertices and n*(n - 1)/2 edges.
Complexity of the algorithm highly depends on degree of vertices. The higher it is, the longer computation time is required. It is related to the process of looking for subsets of size two for each vertex. The running time of the algorithm is $\sum_{i=1}^{n} \binom{k_i}{2}$ where k_i is size of $A(v_i)$, v $\in$ V. The complexity of the algorithm counted for the worst case is calculated below. The worst case scenario is when an input for our algorithm is a full graph (all actors in the database played together).
$t(n,k) = \sum_{i=1}^{n} \binom{k_i}{2} \approx \sum_{i=1}^{n} k_i^2 \approx n * k_{max}^2 \approx n * (n-1)^2 = O(n^3)$
Comparing our approach to Misra-Gries algorithm presented in chapter 3, we can see that the complexity of our algorithm is lower. We present differences in computation time in table 1. The values are counted for the worst case scenario, meaning the worst value we found in dataset. Let a be a constant and equals to 1274, that denotes the highest amount of actors played in a movie. Let k be a constant and equals to 15845. It is the highest number of adjacent vertices a vertex had found in database. Number of movies in IMDB equals to 388126 and number of actors equals to 817718. The values

Table 1: Comparison of running times of Misra-Gries algorithm and our approach for the worst case scenario

Method	Running Time	Value
Misra-Gries	$\sum_{i=1}^{m} \binom{a_i}{3} * h$	$4.805 * e^{15} * h$
Our approach	$\sum_{i=1}^{n} \binom{k_i}{2}$	$4.106 * e^{14}$

obtained show that our approach has smaller time complexity than Misra-Gries. For the worst case scenario it is 10*h times.
For the parallel approach, the running time is $\sum_{i=1}^{n} \binom{k_i}{2}/x$ where x is number of threads and assuming that all the threads run simultanously. We ignore the time needed for aggregation after computing all the threads, since it is relatively small number.

5 Experiments

Some experiments were conducted to verify the correctness of the algorithm and its running time. The results were compared with the "naive" approach - a brute force algorithm that checks any possible combinations.

IMDB IS	Roles size	Bulding DS	Algorithm RT	Result	RM
100%		0 h	38.6 h	{760909, 406612, 80307}	15
100%	48 967 421	393.409 sec	8.282 sec	{150878, 215408, 215564}	130
50% (seg-0_2)	24 434 967	200.750 sec	4.795 sec	{150878, 215408, 215564}	130
50% (seg-1_2)	24 532 454	196.157 sec	3.356 sec	{157955, 651761, 329494}	101
25% (seg-0_4)	12 339 915	98.711 sec	2.142 sec	{33500, 316918, 761020}	84
25% (seg-1_4)	12 273 033	98.430 sec	2.095 sec	{41669, 341023, 338852}	94
25% (seg-2_4)	12 101 052	95.708 sec	2.150 sec	{150878, 215408, 215564}	130
25% (seg-3_4)	12 259 421	97.868 sec	2.061 sec	{157955, 651761, 329494}	101
10% (seg-0_10)	4 899 811	39.729 sec	1.225 sec	{33500, 316918, 761020}	84
1% (seg-0_100)	489 009	4.242 sec	0.183 sec	{33500, 316918, 761020}	84

6 Conclusion

While a streaming algorithm as Misra-Gries provides the advantage of less memory usage, it has one key disadvantage for our problem - the need to explore every possible triplet combination. In cases where the actors count of a movie is too big (1000+), this introduces an unacceptable running time. Our approach goes away with this by managing to explore a triplet only once. Furthermore, it allows for parallelisation of the computation by allowing subsets of the data to be computed independently.

6.1 Future Work

A possible Triangle counting with Hadoop

7 References