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 is splitting the data into bits that can be computed independently. Alternatively, another strategy offers an approximation of the result with the advantage of using little memory. In this paper we explore two such algorithms, comparing their running times and space usage. The first algorithm, designed by us, is specifically targeted for solving the problem at hand. This makes it possible to apply some seemingly small tweaks, which increase performance dramatically. The second is the Misra-Gries streaming algorithm. Furthermore, the concept of parallelism is explored in the context of our solution. It is shown that with very little additional code the consequtive algorithm can be easily parallelised. Which further increases performance and decreases memory requirements.

Keywords

IMDB, graph, triples 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 1 we examine the problem, its input and expected output. In section **??** 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 many of shortcomings of Misra-Gries. Further section **??** explores a way of improving our algorithm running time by running it in parallel. In section 3.5, we compare the two approaches, their running time, space usage, how they differ and why.

2 Problem Description

The **goal** is to find the three actors, whose movie count they have played together in is maximized among he 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, on an empty list if no three actors have played together in the same movie.

3 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.

3.1 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.

3.2 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 3 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

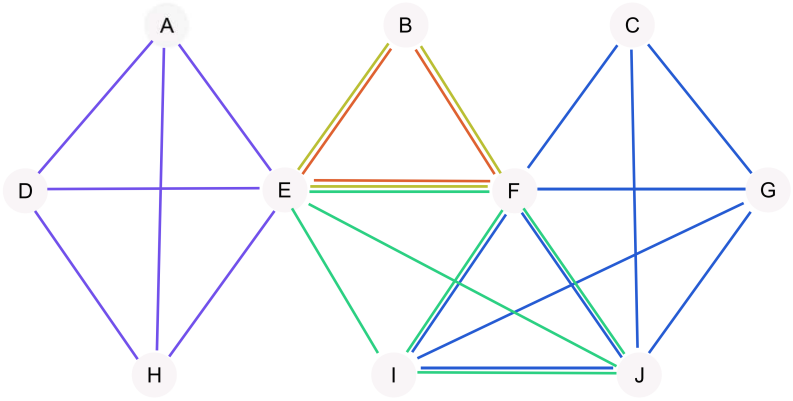

Figure 1: Graph Example

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

3.3 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 list 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;
```

3.4 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. To split data evenly, we just picked every ???? actor from the sorted list we have. That guaranties that workload on every part of data is similar.
We claim that we have no repetition of processing data. It is shown on the picture below:

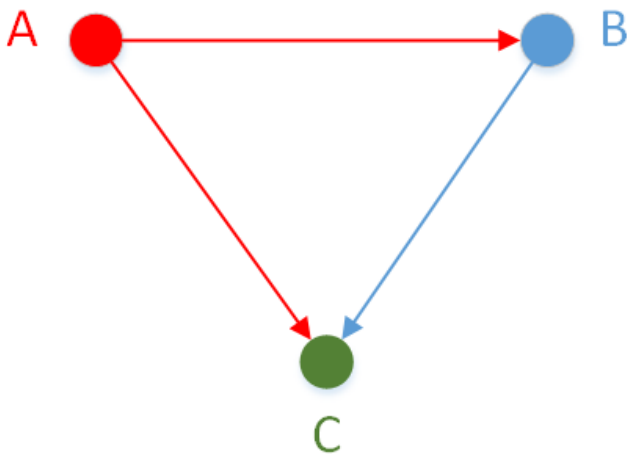

Figure 2: Threads Work Aggregation

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

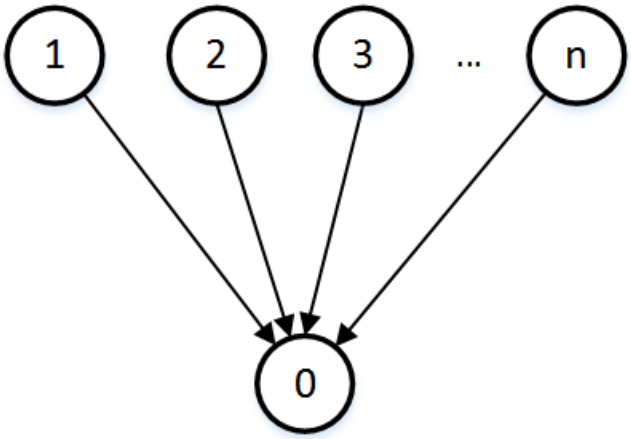

Figure 3: Threads Work Aggregation

The figure above models the aggregation done by process 0 of the work produced by processes from 1 to n.

3.5 Analysis

By not choosing to implement a traditional triangles counting, we avoided to examine three times the same triple of the actors and the number of their common movies they played in. To be able to avoid this, 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 needed. It is related to the process of looking for subsets of size two for each vertex. The complexity of the algorithm is $\sum_{i=1}^{n} \binom{k_i}{2}$ where k_i is size of A(v_i), v $\in$ V, which we can reduce to $\sum_{i=1}^{n} k_i^2$.
Comparison with Misra-Gries...

4 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. TO BE CONTINUED...

5 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 triple combination. Why we chose ours instead of misra gries

Back to the initial claim...

5.1 This is the conclusion text

5.2 Future Work

6 References