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 parallelisation 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, parallelisation

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 3.1 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 4.4 explores a way of improving our algorithm running time by running it in parallel. In section 5, we present results of experiments we conducted on IMDB database.

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 h 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 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 $\sum_{i=1}^{m} \binom{a_i}{3} * h$ where h is size cache size. The complexity of the algorithm is calculated below.
t(m, a, h) $= \sum_{i=1}^{m} \binom{a_i}{3} * h \approx \sum_{i=1}^{m} a_i^3 * h \approx m * a_{max}^3 * h = \mathrm{O}(m * a_{max}^3 * h)$

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 of common movies between any given three actors, found so far.

4.2 Data structure

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

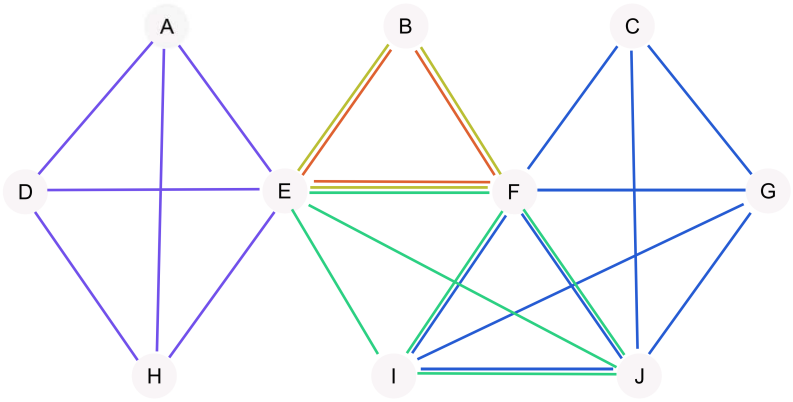

Figure 1: Graph Example

To clearly illustrate the problem, figure 1 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 initially 1 - a single movie common to two actors (vertices). The edge contains sorted list of the movies common to two actors adjacent that 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). For each vertex we generate all subsets of size two of the its adjacent edges. 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". This allows us not to look for triangles in traditional way. Instead, we look for pairs of edges, adjacent to a vertex, that have common movies.
We examine each pair of edges and we find the number of common movies that actors played together in. This is the intersection of the two subsets. Another optimisation can be applied here. The intersections, whose size is less than the max size found so far, can be ignored.
The operations described so far are summarized in the pseudocode 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 Parallelisation

Because of the fact the data set is huge, building graph for such an amount of data requires lots of memory. This introduced the need to split the data and make computations on separate parts in parallel. We took care of not losing any data and not processing any records multiple times.
Each separate group contains specific number of actors and outgoing edges. 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 by the Id of the first actor (in this actor with lower Id in an edge). We noticed that since an edge is incoming to the actor with higher Id, vertices with lower Ids have much more outgoing than incoming edges. So in order to have an even separation of computation, we couldn't just pick a contigious collection of vertices for a specific group.
For instance if we want to have ten groups, to split data evenly in each group, we just picked every tenth 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 outgoing edges.
We claim that we do not process same data multiple times. It is shown on figure 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 the green group we have only vertex C without any outgoing 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 models the aggregation of the work produced by processes from 1 to n. In our case depth is 1 and work is n.

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

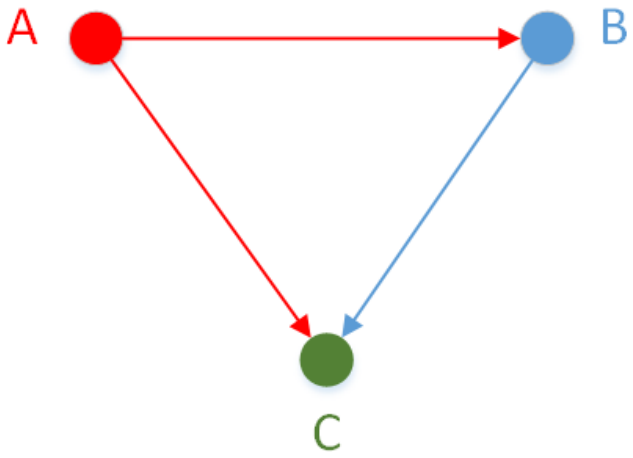

Figure 2: Vertex Triplet

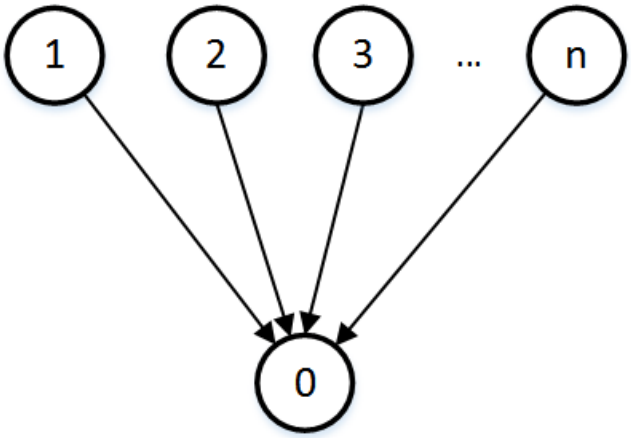

Figure 3: Threads Work Aggregation

machine has and amount of parallel processes we can run on it.

4.5 Analysis

By not opting for an implementation of a traditional triangles counting ($O(n^3)$), we avoided to examine the same vertex three times. To do so, we constructed a special data structure and by consolidating edges, we save space. 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.
There are two phases of the algorithm and we start with building a graph. A running

time for it is: $\sum_{i=1}^{m} m_i * d_{mi}$ where m_i is a number of movies that i-th actor was playing and d_{mi} is a number of actors playing in m_i-th movie. It results in $O(n * m_{max} * d_{max})$. Complexity of the algorithm highly depends on the 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 ∈ V. The complexity of the algorithm counted in the worst case is presented below. The worst case scenario is when the input 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 number and equal to 1274 indicating the highest amount of actors played in a movie. Let k be a constant number and equal to 15845 denoting the highest number of adjacent vertices a vertex had found in database. Let d be a constant number and equal to 909 indicating the highest number of movies that an actor played in. Number of movies in IMDB equals to 388126 and number of actors equals to 817718. For calculating running time of our approach we added running times of building a graph and making computations. The values obtained show that our

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	$\sum_{i=1}^{m} m_i * d_{mi} + \sum_{i=1}^{n} \binom{k_i}{2}$	$4.116 * e^{14}$

approach has smaller time complexity than Misra-Gries. For the worst case scenario it is approximately 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 computed for both algorithms: Misra-Gries and our algorithm. The experiments were run on the same machine. Environment specification:

Processor Intel Core i7-2760QM 2.20 GHz

Memory 8GB, 1,333MHz DDR3

Hard drive 1TB, 5,400rpm

5.1 Results for Misra-Gries

Table 2: Results for Misra-Gries algorithm

IMDB Input Size	Roles Number	Running Time (h)	Result Triplet
100%	48 967 421	38.6	{760909, 406612, 80307}

As predicted in section 3.2, the running time of Misra-Gries in our setup was expected to be long. The results in table 2 support this. The long running time comes from the fact that there are movies with a large number of actors. For these, the algorithm computes all possible triplets combinations. Furthermore, the bigger the cache, the longer the running times. The above experiments were performed with a chache size 20 000 triplets. More about the chache size in section 3.1

5.2 Results for our algorithm

Table 3: Results for our algorithm

IMDB Input Size	Roles Number	Building Time (sec)	Running Time (sec)	Result Triplet	Result Movies
100%	48 967 421	393.409	8.282	{150878, 215408, 215564}	130
50% (seg-0_2)	24 434 967	200.750	4.795	{150878, 215408, 215564}	130
50% (seg-1_2)	24 532 454	196.157	3.356	{157955, 651761, 329494}	101
25% (seg-0_4)	12 339 915	98.711	2.142	{33500, 316918, 761020}	84
25% (seg-1_4)	12 273 033	98.430	2.095	{41669, 341023, 338852}	94
25% (seg-2_4)	12 101 052	95.708	2.150	{150878, 215408, 215564}	130
25% (seg-3_4)	12 259 421	97.868	2.061	{157955, 651761, 329494}	101
10% (seg-0_10)	4 899 811	39.729	1.225	{33500, 316918, 761020}	84
1% (seg-0_100)	489 009	4.242	0.183	{33500, 316918, 761020}	84

The analysis made in section 4.5 predicted, the worts case running time is $O(n^3)$. As can be seen in table 3, the running times are obviously within the bounds. Meaning that doubling the problem size increases the computational time with a factor of less than 3.

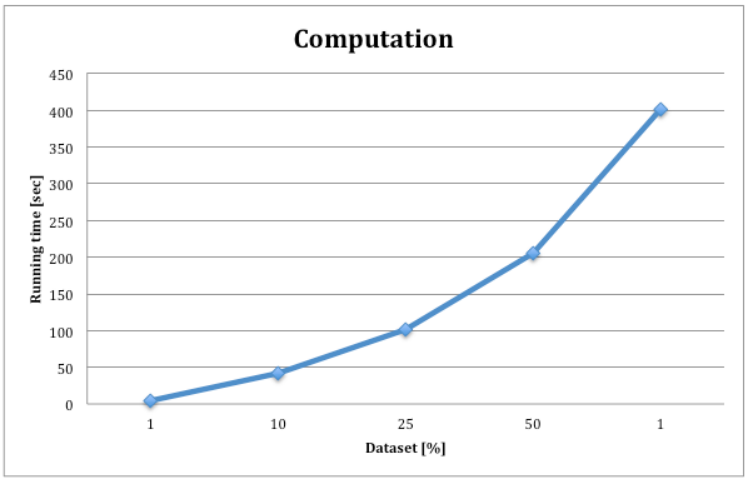

Figure 4: Algorithm Running Times

5.3 Parallelisation

As already mentioned, the graph is built in such a way, that the data is easy to split. This property allows for easy parallelisation. Parallelisation can be achieved by simply dividing the data into groups and processing them parallel. We ran our experiments on an 8-core machine, which allows for a maximum of 8 threads running simultaneously. This makes the algorithm almost 8 times faster. Table 4 show the time needed to process 100% of the data set with single thread and eight threads running concurrently.
Important to note here, is what happens in the case of when the data set does not fit in memory. We could have multiple parallel executions in this case. Each parallel execution will be performed sequentially. Take for example the case when the whole data set is of size 10GB, while we have 4GB available. We can have 3 sequential executions (2x4 + 1x2). Each of them will use all the available cores - will be executed in parallel.

Table 4: Results for Parallel Execution

Thread Count	Running Time (sec)
1	401.691
8	53.487

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

Possible future work might be exploring counting triangles algorithm. We can build graph in the same way we presented, but we analyze each triangle in the graph instead (a triangle represents a triplet of actors). Counting triangles method using MapReduce gives good results and might be an interesting alternative to our approach [1].

[1] http://theory.stanford.edu/ sergei/papers/www11-triangles.pdf

Appendix 1 – Code for Misra – Gries

Main:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MisraGries
{
   class Program
   {
      static void Main(string[] args)
      {
         MGAlgorithm algorithm = new MGAlgorithm (20000);

         algorithm.Process2();

         var v = algorithm.Cache.OrderByDescending(pr => pr.Value).First();

         Console.WriteLine(string.Format("Triple found: {0} {1} {2}",
            v.Key.Values.ElementAt(0),
            v.Key.Values.ElementAt(1),
            v.Key.Values.ElementAt(2)));

         Console.ReadLine();
      }
   }
}
```

Computation:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MisraGries
{
   class MGAlgorithm
   {
      private int cacheSize;
      private Dictionary<Triple, int> cache;

      public Dictionary<Triple, int> Cache { get { return cache; } }

      public DataProvider dt = new DataProvider();

      private int tripleCount = 0;
      private int percentCount = 0;
      private int[] percentages = new int[] { 1, 10, 25, 50, 100 };
      private const int totalTriples = 1925588228;

      public MGAlgorithm(int cacheSize)
      {
```

```
            this.cacheSize = cacheSize;
            cache = new Dictionary<Triple, int>(cacheSize);
        }

        public void Process(IEnumerable<int> actors)
        {
            if (actors.Count() < 3) { return; }

            foreach (Triple tr in GetTriples(actors))
            {
                Process(tr);
            }
        }

        public void Process2()
        {
            Stopwatch st = new Stopwatch();
            st.Start();
            Console.WriteLine("Started");

            List<Role> roles = new List<Role>();
            int prevMov = -1;

            foreach (var v in dt.GetRoles())
            {
                if (prevMov != v.MovieId)
                {
                    prevMov = v.MovieId;

                    if (0 < roles.Count)
                    {
                        //Debug.WriteLine("Processing movie " + ++movieCount);

                        foreach (var tr in GetTriples(roles))
                        {
                            Process(tr);
                            tripleCount++;
                            double nr = totalTriples * (percentages[percentCount] / 100.0);
                            if (nr <= tripleCount)
                            {
                                Console.WriteLine(
                                    string.Format("{0} % processed in {1} seconds", percentages[percentCount], st.
ElapsedMilliseconds / 1000));
                                percentCount++;
                            }
                        }
                        roles.Clear();
                    }
                }
                roles.Add(v);
            }
        }

        private void Process(Triple triple)
        {
            if (cache.ContainsKey(triple))
            {
                cache[triple]++;
            }
            else
```

```
            {
                cache.Add(triple, 1);

                if (cacheSize <= cache.Keys.Count)
                {
                    for (int i = 0; i < cache.Keys.Count; i++)
                    {
                        var key = cache.Keys.ElementAt(i);
                        cache[key]--;
                        if (cache[key] <= 0)
                        {
                            cache.Remove(key);
                        }
                    }
                }
            }
        }

        private IEnumerable<Triple> GetTriples(IEnumerable<int> actors)
        {
            for (int i = 0; i < actors.Count(); i++)
            {
                for (int j = i + 1; j < actors.Count(); j++)
                {
                    for (int k = j + 1; k < actors.Count(); k++)
                    {
                        yield return new Triple(actors.ElementAt(i), actors.ElementAt(j), actors.ElementAt(k));
                    }
                }
            }
        }

        private IEnumerable<Triple> GetTriples(IEnumerable<Role> roles)
        {
            for (int i = 0; i < roles.Count(); i++)
            {
                for (int j = i + 1; j < roles.Count(); j++)
                {
                    for (int k = j + 1; k < roles.Count(); k++)
                    {
                        yield return new Triple(
                            roles.ElementAt(i).ActorId,
                            roles.ElementAt(j).ActorId,
                            roles.ElementAt(k).ActorId);
                    }
                }
            }
        }
    }
}
```

Appendix 2 – Code for our algorithm

Main:

```
using Entities;
using System;
using System.Diagnostics;
using TriangleProblem.Utils;
using TriangleProblem.Utils.Computation;
using Utils.Input;

namespace TriangleProblem
{
   public class Program
   {
      const String FILE_PATH = "../../../../imdb/segments/all.csv";

      static void Main(string[] args)
      {
         FileParser fileParser = new FileParser(FILE_PATH);
         Stopwatch stopwatch = new Stopwatch();
         stopwatch.Start();
         Graph graph = fileParser.Parse();
         stopwatch.Stop();

         TimeSpan time = TimeSpan.FromMilliseconds(stopwatch.ElapsedMilliseconds);
         Console.WriteLine("Graph builded in (seconds): " + time.TotalSeconds);

         GraphManager manager = new GraphManager(graph);

         stopwatch.Reset();
         stopwatch.Start();
         Result result = manager.FindTreeActorsThatPlayedInMostMovies();
         stopwatch.Stop();

         foreach (Actor actor in result.Actors)
         {
            Console.WriteLine(actor.Id);
         }

         time = TimeSpan.FromMilliseconds(stopwatch.ElapsedMilliseconds);

         Console.WriteLine("total: "+ result.TotalMovieCount);
         Console.WriteLine("Time elapsed (seconds): " + time.TotalSeconds);
         Console.WriteLine("DONE");
         Console.ReadLine();
      }
   }
}
```

Bulding graph:

```
using Entities;
using Entities.DAL;
using Entities.Extensions;
using System;
using System.Collections.Generic;
using System.Linq;
```

```
namespace Utils.Input
{
   public class FileParser : IParser
   {
      private String FilePath { get; set; }

      public FileParser(String filePath)
      {
         FilePath = filePath;
      }

      public Graph Parse()
      {
         String[] deli = {","};
         String[] deliMovieTitle = {",'", "',"};
         Graph graph = new Graph();
         Roles roles = new Roles();

         using (FileInput input = new FileInput(FilePath))
         {
            String line;

            while ((line = input.ReadLine()) != null)
            {
               var array = line.Split(deli, StringSplitOptions.RemoveEmptyEntries);
               int actor1Id = int.Parse(array[0]);
               int actor2Id = int.Parse(array[1]);
               int movieId = int.Parse(array[2]);

               if (!graph.Actors.ContainsKey(actor1Id)) graph.Actors.Add(actor1Id, new Actor() { Id = a
ctor1Id });
               if (!graph.Actors.ContainsKey(actor2Id)) graph.Actors.Add(actor2Id, new Actor(){ Id=act
or2Id });
               if (!graph.Movies.ContainsKey(movieId)) graph.Movies.Add(movieId, new Movie() { Id
= movieId });

               Actor actor1 = graph.Actors[actor1Id];
               Actor actor2 = graph.Actors[actor2Id];
               Movie movie = graph.Movies[movieId];
               Edge edge;

               if (actor1.Edges.Exists(e => e.StartNode == actor1 && e.EndNode == actor2))
               {
                  edge = actor1.Edges.FirstOrDefault(e => e.StartNode == actor1 && e.EndNode == acto
r2);

               }
               else
               {
                  edge = new Edge() { StartNode = actor1, EndNode = actor2 };
                  actor1.Edges.Add(edge);
               }
               edge.CommonMovies.Add(movie);
            }
         }

         return graph;
      }
   }
}
```

Computations:

```
using Entities;
using System;
using System.Collections.Generic;
using System.Linq;

namespace TriangleProblem.Utils.Computation
{
   public class GraphManager
   {
      private Graph Graph { get; set; }

      public GraphManager(Graph graph)
      {
         Graph = graph;
      }

      public GraphManager()
      {
         RemoveEdges(Graph.Actors[0]);
      }

      public Result FindTreeActorsThatPlayedInMostMovies()
      {
         int movies_count = 0;
         Result result = new Result();

         foreach (Actor actor in new List<Actor>(Graph.Actors.Values))
         {
            for (int i = 0; i < actor.Edges.Count; i++)
            {
               List<Movie> movies1 = new List<Movie>(actor.Edges[i].CommonMovies);
               if (movies_count >= movies1.Count)
                  continue;
               for (int j = i + 1; j < actor.Edges.Count; j++)
               {
                  List<Movie> movies2 = new List<Movie>(actor.Edges[j].CommonMovies);
                  if (movies_count >= movies2.Count)
                     continue;
                  int count = CommonMovieSubsetCount(movies1, movies2);
                     if (movies_count < count)
                     {
                        movies_count = count;
                        result.Actors[0] = actor;
                        if (actor == actor.Edges[i].StartNode)
                        {
                           result.Actors[1] = actor.Edges[i].EndNode;
                        }
                        else
                        {
                           result.Actors[1] = actor.Edges[i].StartNode;
                        }
                        if (actor == actor.Edges[j].StartNode)
                        {
                           result.Actors[2] = actor.Edges[j].EndNode;
                        }
                        else
                        {
```

```
                        result.Actors[2] = actor.Edges[j].StartNode;
                    }
                }
            }

        }
        //RemoveEdges(actor);
    }

    result.TotalMovieCount = movies_count;

    return result;
}

public int CommonMovieSubsetCount(List<Movie> movies1, List<Movie> movies2)
{
    int p1 = 0;
    int p2 = 0;
    int count = 0;
    while (p1 < movies1.Count && p2 < movies2.Count)
    {
        if (movies1[p1].Id == movies2[p2].Id)
        {
            count++;
            if (p1 < movies1.Count)
            {
                p1++;
            }
            if (p2 < movies2.Count)
            {
                p2++;
            }
        }
        else if (movies1[p1].Id < movies2[p2].Id)
        {
            if (p1 < movies1.Count)
            {
                p1++;
            }
        }
        else if (movies2[p2].Id < movies1[p1].Id)
        {
            if (p2 < movies2.Count)
            {
                p2++;
            }
        }
    }

    if (p1 == movies1.Count)
    {
        for (int i = p2; i < movies2.Count; i++)
        {
            if (movies1[p1 - 1].Id == movies2[i].Id)
                count++;
        }
    }

    if (p2 == movies2.Count)
    {
```

```
        for (int i = p1; i < movies1.Count; i++)
        {
            if (movies2[p2 - 2].Id == movies1[i].Id)
                count++;
        }
    }
    return count;
}
```