UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

ALEXANDRE FLORES JOHN

Cluster Implementation for the Link Assessment Problem Finding a Heuristic to Estimate the Number of Swaps

Trabalho de graduação.

Trabalho realizado na
Technische Universität Kaiserslautern.

Orientador brasileiro:
Profa. Dra. Renata Galante

Orientador alemão:
Dipl.-Ing. C. Brugger

Porto Alegre
2015

1 Abstract

Recommendation Systems play a important role in the markets nowadays, they are responsible for offering new products to costumers based on this costumer's interest. For such a problem, called *Link Assessment Problem*, katharina A. Zweig proposed an algorithm, such algorithm was implemented on a cluster intended to run on several nodes in parallel. This thesis aims to reduce the time consuming of this implementation.
Swapping and co-occurrence calculation represent the major time consuming of this algorithm. The propose of this thesis is to show a runtime heuristic that tries to decrease the time consuming in the swapping part.
Besides a better understanding of the swapping procedure, the time consuming in swapping was reduced by up to 82% and the whole time computation by up to 30% using netflix datasets with 1k, 10k, 20k and 100k users. The tests were performed on a Intel Xeon architecture.

Sistemas de recomendação desempenham um importante papel no cenário de vendas, eles são responsáveis por oferecer novos produtos a consumidores tendo como base o interesse desse consumidor. Para tal problema, chamado de *Link Assessment Problem*, katharina A. Zweig propôs um algoritmo.
Tal algoritmo foi implementado em um cluster visando rodar em diversos nodos em paralelo. Esta tese tem por objetivo reduzir o tempo consumido por essa implementação.
Cálculos de swap e co-ocorrência representam o maior consumo de tempo desse algoritmo. A proposta dessa tese é demonstrar uma heuristica que execute em tempo de execução e diminua o tempo empregado em swaps.
Além de uma melhor compreensão do procedimento de swaps, o tempo de execução de swaps foi reduzido em até 82% e o tempo global de execução em até 30%. Para testes, forma usados datasets do Netflix de 1k, 10k, 20k and 100k usuários. Os testes foram executados em uma arquitetura Intel Xeon.

Lista de Figuras

Lista de Tabelas

Sumário

2 Introdução

O que segue é um resumo estendido em português para a Universidade Federal do Rio Grande do Sul do trabalho original em anexo. O trabalho de conclusão original, em inglês, foi apresentado na Technische Universität Kaiserslautern.
Tempo é uma variável crítica para algoritmos de big data, processamento com apenas um core torna-se ineficiente para esse fim. O tempo de computação está relacionado com energia, que por sua vez está relacionado com custos. Portanto quanto mais tempo o algoritmo levar, mais custos serão necessários. Nesse cenário, alternativas como algoritmos mais rápidos e paralelização são necessárias. Levando em conta isso, foi implementado, em c, a versão paralela do *Link Assessment Problem*, que é executado com diversos cores. Essa implementação é chamada de *Cluster Implementation*, pois foi desenvolvida visando a sua execução em um cluster com vários nodos. Embora essa implementação seja a mais rápida dentre as conhecidas, a medida de qualidade (PPV) mostra que o número de swaps é mais do que necessário. Disso surge a motivação para a criação de uma heuristica para selecionar um número adequado de swaps que ainda mostre uma boa qualidade de saída. Logo essa tese visa acelerar ainda mais a implementação do cluster.

3 Fundamentação Teórica

3.1 Encontrando Links Ocultos

Katharina A. Zweig propôs em [1] e [2] uma abordagem para encontrar links ocultos em grafos bipartidos, tal problema tem o nome de *Link Assessment Problem*. Um grafo bipartido é um grafo cujos vértices podem ser agrupados em dois conjuntos de interesses $\cup$ and $\vee$ (veja figura 1), exemplos são: proteínas que interagem com genes, usuários e preferências de filmes ... A ideia é criar, a partir do relacionamento das arestas, um único grafo mostrando a relação de apenas um lado de interesse. Por exemplo, em um grafo bipartido, teríamos usuários de um lado e filmes de outro. Em uma abordagem chamada *One-Mode Projection*, teríamos como saída a relação de filmes e pesos nas arestas demonstrando a intensidade de cada par de filmes. Os resultados obtidos podem ser usados para a criacão de uma sistema de recomendação, um sistema que sugere filmes (ou outro parâmetro de interesse) a usuários.
Por uma questão de testes, nesta tese serão usados grafos bipartidos do Netflix, um sistema de visualizacão de filmes online, onde os usuários podem, em uma escala de 1 a 5, estabelecer uma nota para o filme de sua preferencia. Portanto, os lados de interesse tratados serão usuários e filmes, o que buscamos é estabelecer uma relacão entre filmes.

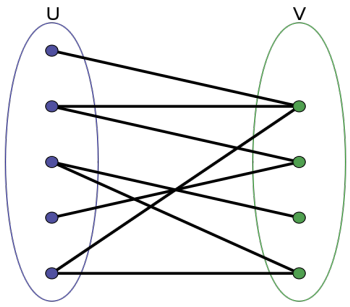

Figura 1: Grafo Bipartido

3.2 One-Mode Projection

Para entendermos como o One-Mode Projection é calculado, vamos considerar a figura 2, que mostra o diagrama de fluxo do algoritmo. Após ler como entrada um grafo bipartido de usuários e filmes, o algoritmo começa. Primeiro a co-ocorrência de todos os pares de filmes é calculada e armazenada, vamos chamar esse conjunto de valores de $cooc_{initial}$.

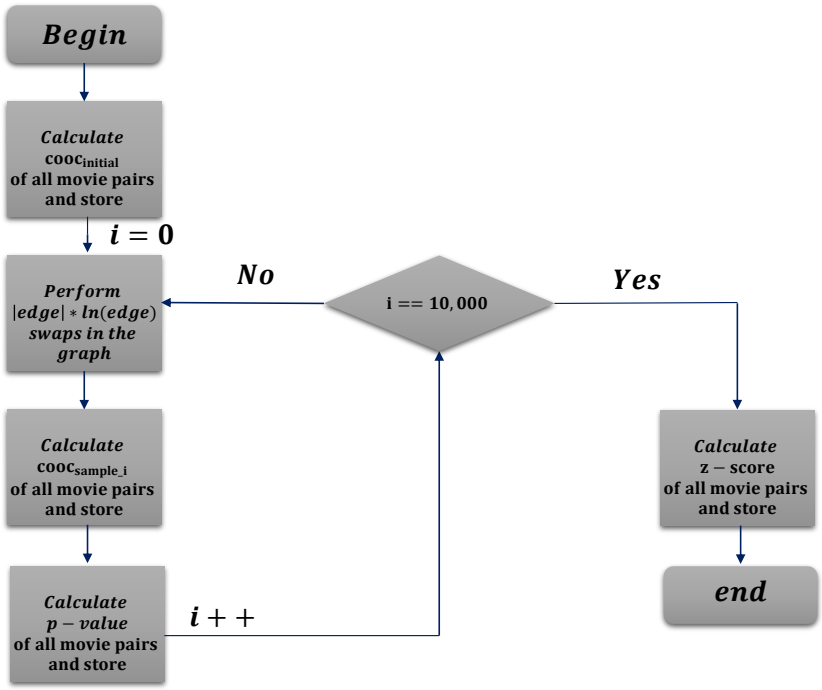

Figura 2: Algoritmo para o One-Mode Projection

A co-ocorrência, dada por um par de filmes, é nada mais do que o número de usuários em comum que gostam de ambos os filmes. Podemos ver um exemplo na figura 3, no qual a co-ocorrência de dois filmes, x e y, é calculada. O numero de usuários em comum que gostam de ambos os filmes está representado em verde claro e tem o valor de 6.
Após o cálculo de $cooc_{initial}$, começa-se o processo de amostragem, para isso o grafo bipartido fará trocas de filmes entre os usuários, trocas essas chamadas de *swaps*, cada amostra fará $|arestas| \times ln(|arestas|)$ swaps. Após cada

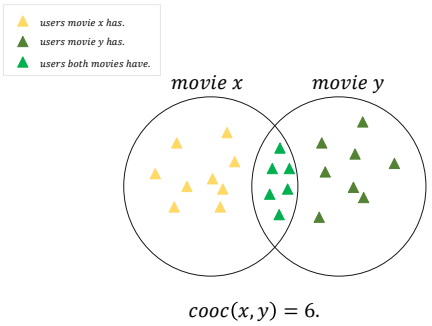

Figura 3: Exemplo do Cálculo da Co-ocorrência

amostragem, a nova co-ocorrência de todos os pares de filmes é novamente armazenada e serão coletadas 10,000 amostras. Portanto, ao final do algotimo, teremos 10,000 co-ocorrências de amostragem, para todos os pares de filmes, armazenadas. Para entendermos como os swaps são calculados, considere a matriz de adjacências mostrada na figura 4. Vamos chama-la de M. Primeiro, duas arestas são selecionadas rândomicamente. No exemplo, são elas $M(0,0)$ e $M(2,4)$, só será possível o swap se ambas arestas existirem, i.e se ambos tiverem o valor 1. E, além disso, $M(2,0) = M(0,4) = 0$, caso ambas as condições sejam satisfeitas, 1 swap é feito, ou seja $M(0,0) \leftarrow 0$, $M(2,4) \leftarrow 0$, $M(2,0) \leftarrow 1$ e $M(0,4) \leftarrow 1$. Swaps cujas condicões não são satisfeitas também contam como swap.

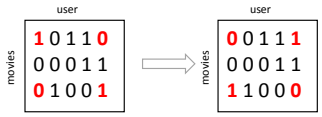

Figura 4: Exemplo do Cálculo de Swaps

Os coeficientes de similaridade de cada filme são p-value e z-score. O p-value é calculado ao final de cada amostragem e é definido como o número

de amostras os quais $cooc_{amostra} \geq cooc_{initial}$. Já o z-score é computado ao final da amostragem e é definido como

$$z{-}score(m_i, m_j) = \frac{cooc_{initial}(m_i, m_j) - cooc_{FDSM}(m_i, m_j)}{\sqrt{\sum_{i=1}^{amostras} \left(\frac{cooc^2_{amostra}(m_i,m_j)}{amostras}\right) - \sum_{i=1}^{amostras} \left(\frac{cooc_{amostra}(m_i,m_j)}{amostras}\right)^2}}$$

onde

$$cooc_{FDSM}(m_i, m_j) = \frac{1}{amostras} \sum_{i=1}^{amostras} cooc_{amostra_i}(m_i, m_j)$$

.Os Coeficientes representam a intensidade do relacionamento de cada par de filme, quanto maior z-score é, maior o grau de similaridade entre os filmes, quanto maior p-value é, menor o grau de similaridade entre os filmes.

3.3 Implementação do Cluster

A implementacão do Cluster [1] tem como objetivo paralelizar o algoritmo para o cálculo do *One-Mode Projection*. A ideia é dividir as amostras entre os nodos, de modo que cada nodo seja encarregado de computar $\frac{10{,}000}{n}$ amostras onde n é o número de nodos (Veja a figura 5).

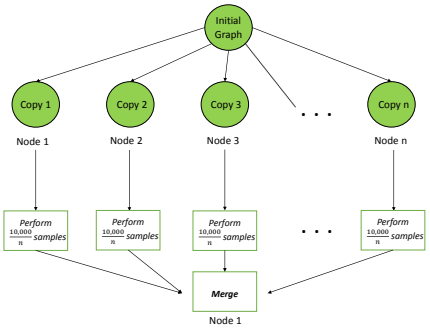

Figura 5: Implementação do Cluster

[1] Alexandre Flores John, Andre Lucas Chinazzo, source: https://git.rhrk.uni-kl.de/EIT-Wehn/ bigdata.graphs.

4 Phase Transition

4.1 PPV

O PPVk *(positive predicted value)* mede a qualidade da saída do *One-Mode Projection*. Ele recebe como entrada a saída do *One-Mode Projection* e um arquivo denominado *ground truth*.
O *ground truth* uma seleção feita por humanos do que seriam filmes similares. Por exemplo Senhor dos Anéis: o retorno do rei e Senhor dos Anéis: a sociedade do anel são filmes similares. Ele tem como saída um número de 0 a 1, mostrando o quão próximo da realidade a relação entre os filmes está.
Para entendermos o impacto do número de swaps por amostra na saída do algoritmo, foi plotado (figura 6), com 4 conjunto de dados do Netflix com 1k, 10k, 20k e 100k usuários, núumero de swaps dependendo da qualidade(PPV). Para todos os conjuntos de dados, para um número de swap por amostra menor que $usuario \times ln(usuario)$, a qualidade mantem-se em zero, após esse valor, começa a aumentar até estabilizar. O ponto de establização é menor que $arestas \times ln(arestas)$ swaps por amostra, mostrando que esse valor sugerido pelo algoritmo é mais que o suficiente. Esse experimento irá motivar a implementaçao de uma heuristica para determinar, entre $usuario \times ln(usuario)$ e $arestas \times ln(arestas)$, um número suficiente de swaps por amostra a ser computado.

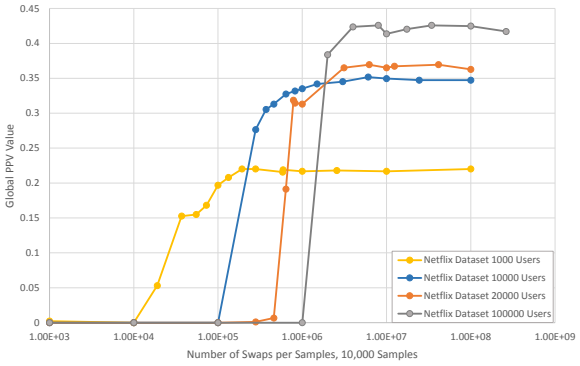

Figura 6: PPV sobre o número de swaps

4.2 Pares de Filmes com o mesmo *Degree*

Um experimento importante observado é que ao selecionar filmes com o mesmo *Degree*, i.e com o mesmo número de usuários relacionados, podemos observar, que para um determinado número de swaps por amostra, a $cooc_{FDSM}$ deve convergir para o mesmo valor. Esse número de swaps é o mesmo para o qual o valor de PPV é estável. Esse experimento é importante na medida que em tempo de execução não temos acesso ao ground truth. Porém podemos descobrir um valor estável para o PPV se observarmos filmes com o mesmo *Degree*.
A figura 7 mostra o experimento para o conjunto de dados de 10,000 usuários do Netflix. Podemos notar, como dito antes, que a establilidade do PPV está entre $usuario \times ln(usuario)$ e $arestas \times ln(arestas)$ e que a estabilidade também está relacionada com a convergência de filmes com mesmo *Degree*. Esse comportamento se repete para todos os outros conjuntos de dados.

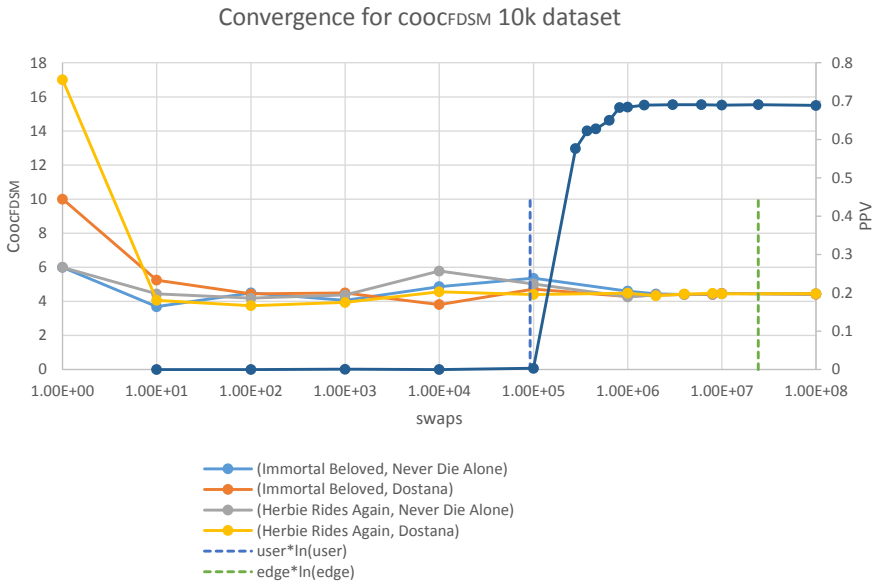

Figura 7: Relação entre o Convergimento da Co-ocorrência e a estabilidade do PPV

4.3 Heuristica Para o Número de Swaps

A heuristica implementada visa, com base na observação do PPV, diminuir o número de swaps por amostra, visto que $arestas \times ln(arestas)$ swaps são mais que suficiente.
A abordagem é selecionar conjuntos e filmes com o mesmo *Degree* e aumentar progressivamente o número de swaps até que os filmes atinjam a establilidade. O número de swaps que a heuristica terá como saída será esse mesmo valor para o qual os pares com o mesmo *Degree* atingiram establilidade.
Um ponto importante a se salientar é que a heuristica é um programa executado antes do programa principal (a implementação do cluster) e que ela executa o cálculo de co-ocorrência para apenas alguns pares de filmes(filmes com o mesmo *Degree*) tendo em vista que a co-ocorrência corresponde aproximadamente à metade do tempo de execução, a heuristica tende a executar mais rapidamente que o programa principal e que o programa principal será executado com um número entre $usuario \times ln(usuario)$ e $arestas \times ln(arestas)$.

4.4 Resultados

Os resultados da heuristica podem ser vistos na tabela 1, a primeira coluna representa o conjunto de dados utilizado. A segunda a diminuição do número de swaps se comparado com $arestas \times ln(arestas)$, tendo uma diminuição de até 19,2%.
Depois o melhoramento em tempo de swap se compararmos o tempo de execução com $arestas \times ln(arestas)$ e do tempo de execução ao executar a implementação do cluster com o número de swap por amostra descoberto pela heuristica, tendo uma melhora de até 83%. Isso significa que, se com $arestas \times ln(arestas)$ swaps por amostra o algoritmo leva 10h, com o novo número de swaps ele levará 1,7h.
A ultima coluna compara o erro entre o PPV obtido com $arestas \times ln(arestas)$ swaps por amostra e o número descoberto pela heuristica, o fato de o erro ser muito baixo, implica que o PPV obtido é muito próximo.

Conjunto de Dados	Diminuição do Número de Swaps	Melhoramento em Tempo de Swap	Erro de PPV para o Groud Truth Series
Netflix 1k Users	$19,2\times$	77%	-1%
Netflix 10k Users	$16,3\times$	83%	-1%
Netflix 20k Users	$15,7\times$	82%	$-,3\%$
Netflix 100k Users	$15,1\times$	82%	$-,2\%$

Tabela 1: Resultados Obtidos

5 Conclusões

A proposta dessa tese é mostrar uma heuristica para acelerar a implementação do cluster. Essa heuristica foi demonstrada e teve como melhoramento do tempo de swaps em até 83%. Além de estudos do número de swaps, o número de amostras vem sendo estudado. Tais estudos mostram que 10,000 amostras são mais do que o necessário.

University of Kaiserslautern

Department of Electrical Engineering and Information Technology

Microelectronic Systems Design Research Group

Bachelor Thesis

Cluster Implementation for the Link Assessment Problem

Finding a Heuristic to Estimate the Number of Swaps

Presented:	January 19, 2015
Author:	Alexandre Flores John
Research Group Chief:	Prof. Dr.-Ing. N. Wehn
Tutor:	Dipl.-Ing. C. Brugger

Statement

I declare that this thesis was written solely by myself and exclusively with help of the cited resources.

Kaiserslautern, January 19

Alexandre Flores John

1 Abstract

Recommendation Systems play a important role in the markets nowadays, they are responsible for offering new products to costumers based on this costumer's interest. For such a problem, called *Link Assessment Problem*, katharina A. Zweig proposed an algorithm, such algorithm was implemented on a cluster intended to run on several nodes in parallel. This thesis aims to reduce the time consuming of this implementation.
Swapping and co-occurrence calculation represent the major time consuming of this algorithm. The propose of this thesis is to show a runtime heuristic that tries to decrease the time consuming in the swapping part.
Besides a better understanding of the swapping procedure, the time consuming in swapping was reduced by up to 82% and the whole time computation by up to 30% using netflix datasets with 1k, 10k, 20k and 100k users. The tests were performed on a Intel Xeon architecture.

Contents

2 Introduction

Time is a critical variable when running big data algorithms, single core computation seems to be useless due time consuming. Computation time is strictly related to energy which in turn is related to costs, thus the more time the algorithm takes to finish, the more coasts are necessary. In this scenario, the necessity for alternatives such as faster algorithm and parallelization take place.

Concerning time consumption, there was implemented a c parallel version for the link assessment problem that runs on several cores meant to deal with big data, this implementation is called cluster implementation because of that it was build to run on a cluster with several nodes [1]. Even though it is the fastest among the known implementation, the quality measurement(called PPV) shows that the number of swaps used are more than enough. Regarding this fact, there comes the motivation to create a heuristic, that runs during runtime, to find out the better suitable swap number that still holds a good quality output. Hence this research aims to reduce even more the cluster implementation time consuming.

The input of the link assessment algorithm is a bipartite graph, while the output is a general graph. For the heuristic evaluation there were used four bipartite graphs datasets: 1,000 users with 10420 movies, 10,000 users with 15844 movies, 20,000 users with 16837 movies and 100,000 users with 17727 movies. These datasets are Netflix dataset, i.e a relationship between the Netflix users and the movies they like.

After running the heuristics for these datasets, both quality results as runtime reduction were satisfactory. The swapping time was reduced up to 82% and the whole program up to 30%.

The sections 3 and for will discuss the link assessment problem and its implementation. Section 4 will show the quality over swaps to motivate for the next sections to present the convergence test and heuristic respectively. Finally the last section will evaluate the heuristic implementation on the cluster implementation

[1] Alexandre Flores John, Andre Lucas Chinazzo, source: https://git.rhrk.uni-kl.de/EIT-Wehn/ bigdata.graphs.

3 Background

3.1 Link Assessment Problem

A Bipartite Graph is a graph whose vertices can be divided into two disjoint sets $\cup$ and $\vee$ (that is, $\cup$ and $\vee$ are each independent sets) such that every edge connects a vertex in $\cup$ to one in $\vee$. Vertex set $\cup$ and $\vee$ are often denoted as partite sets[2].
Many relationships can be modelled as bipartite graphs: proteins that interact with genes, costumers that buy products, users that rank movies, authors and their published articles and many others. This modelling can easily be applied to industry or scientific researches, for example the recommendation systems, i.e, systems that offer different products to the costumer according to his interest.
Most times we are only interested in one part of the of the graph. Suppose we have a recommendation system that offers to the users movies that this user has never watched, if exists a database with the relationship between only movies, the system can recommend movies that are strictly related to the movies this user likes, hence the user side is not necessary.
Regarding this motivation, [1] and [2] suggest an algorithm that builds a new graph based on the bipartite graph, such a method is called *one-mode projection*. This resulting graph has the combination of one side of interest and is weighted by two coefficients of similarity called: *p-value* and *z-score*. The higher *z-score* is, the higher the similarity is, however, the smaller *p-value* is, the higher similarity is between the pair.
In this thesis our main example will be users that rank movies, thus the resulting graph will be the weighted combination of movies. In order to calculate the *one-mode projection*, we have to take the following steps:

1. **Co-Occurrence Calculation** ($cooc_{initial}$)
   Let's consider we have a dataset with the following sets of users and movies, respectively: $\cup = \{u_1, u_2, ..., u_r\}$ and $\vee = \{v_1, v_2, ..., v_l\}$, the $cooc(v_i, v_j)$ is defined as the number of users in common v_i has with v_j. Considering the *Figure 1* below, let's have a look into the sets of users that are associated with movies. Let's define a function $\Xi(m_k)$, as the set of users that are associated with a given movie m_k:

[2] *http://en.wikipedia.org/wiki/Bipartite_graph*

- $\Xi(m1) = \{u1, u2, u3\}$.
- $\Xi(m2) = \{u3\}$.
- $\Xi(m3) = \{u1, u2\}$.
- $\Xi(m4) = \{u3\}$.

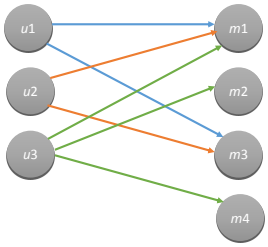

Figure 1: Co-Occurence Calculation Example

As defined, the $cooc(v1, v2) = 1$, for the reason that their user in common is only *u3*, however $cooc(m1, m3) = 2$, since they have *u1* and *u2* in common. In the beginning, we have to calculate and store all $cooc_{initial}(m_i, m_j)$, that are the *cooc* of all the combination of movies from the input bipartite graph.
Let's have a look at another example in order to define the co-occurence. The figure 2 shows two movies, x and y and the users they are related to, the number of users in common for x and y is 6, this amount of users in common represents the co-occurence.

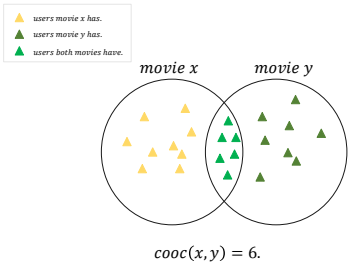

Figure 2: Co-Occurence Example

2. **Swap Sampling**
   In order to calculate the weights of the *one-mode projection*, it is necessary to shuffle the graph, this process is called *swap*. To understand how a swap is performed, consider a adjacency matrix $M(i, j)$, that represent movies x users, first we have to select 2 edges at random, let's say they have coordinates (a, b) and (c, d), if $M(a, b) = M(c, d) = 1$ and $M(b, c) = M(a, d) = 0$ they can be swapped. In this case $M(a, b) = M(c, d) = 0$ and $M(b, c) = M(a, d) = 1$. The swap sampling works as fallow for each sample:

   (a) $|edge| \times ln(|edge|)$ swaps are performed, where m represents the number of edges of the graph. The swaps that are not possible are also count as swaps, hence there is a high probability that there is less swaps than $|edge| \times ln(|edge|)$ per sample.

   (b) $cooc_{sample}$, which represents the co-occurence calculation right after swapping, is calculated and stored.

   10000 samples must be taken. Each random samples has the property to be FDSM, i.e *Fixed Degree Sequence Mode*.

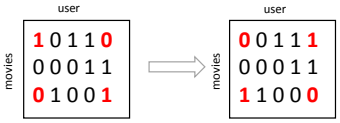

Figure 3: Swap Calculation Example

3. When all samples are already performed, we have to calculate $cooc_{FDSM}$ for each combination of movies. It's better defined as:

$$cooc_{FDSM}(m_i, m_j) = \frac{1}{samples} \sum_{i=1}^{samples} cooc_{sample_i}(m_i, m_j)$$

   where samples represents the total number of samples taken.

4. With the $cooc_{FDSM}$ for all combination of movies calculated, we can calculate their $leverage_{FDSM}$:

$$leverage_{FDSM}(m_i, m_j) = cooc_{initial}(m_i, m_j) - cooc_{FDSM}(m_i, m_j)$$

5. The $z - score$ can be calculated from $leverage_{FDSM}$ and the standard deviation of $cooc_{sample}$:

$$z{-}score(m_i, m_j) = \frac{leverage_{FDSM}(m_i, m_j)}{\sqrt{\sum_{i=1}^{samples} (cooc_{sample}(m_i, m_j) - cooc_{FDSM}(m_i, m_j))^2}}$$

$z{-}score$ can also be written depending on the $cooc_{sample}$ and $cooc^2_{sample}$:

$$z{-}score(m_i, m_j) = \frac{leverage_{FDSM}(m_i, m_j)}{\sqrt{\sum_{i=1}^{samples} \left(\frac{cooc^2_{sample}(m_i,m_j)}{samples}\right) - \sum_{i=1}^{samples} \left(\frac{cooc_{sample}(m_i,m_j)}{samples}\right)^2}}$$

6. $p{-}value$ is the number of samples that satisfies the inequation $cooc_{sample} \geq cooc_{initial}$.

The figure below shows the flowchart of the *Link Assessment Problem*. First the algorithm calculates the initial co-occurrence without any change and stores it for all pair of movies. After that there comes the sampling, where the original configuration will be shuffled 10,000 times, each time besides the p-value calculation and storage for each movie pair, the corresponding co-occurrence will be stored for later z-score calculation.

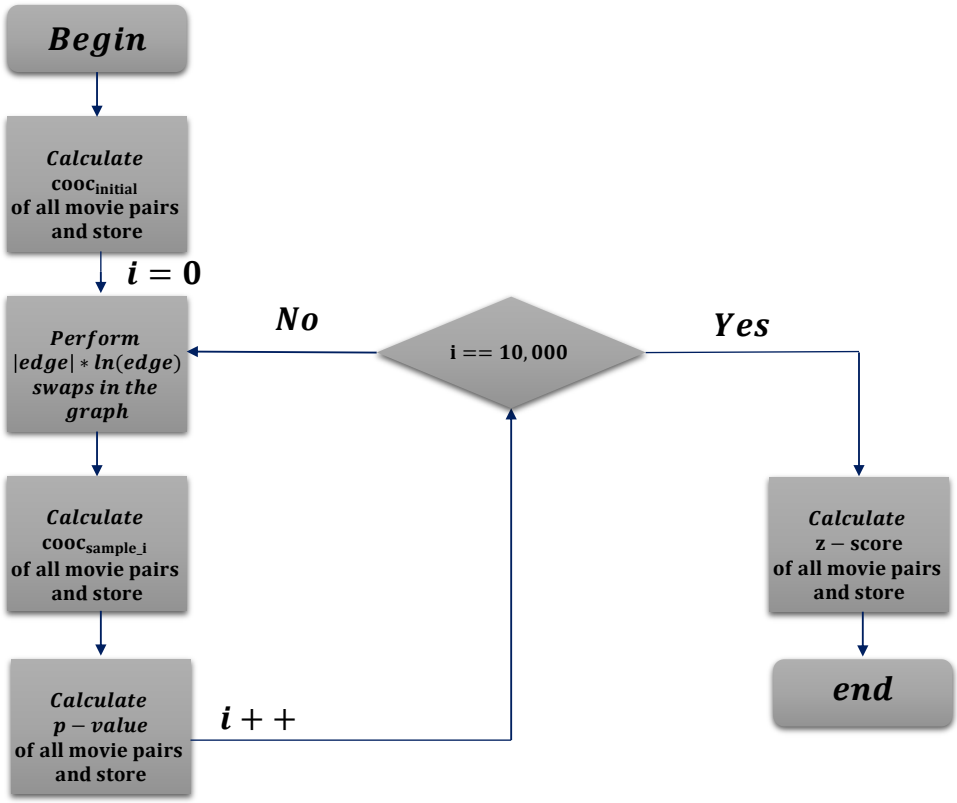

Figure 4: Link Assessment Flowchart

Even though [1] and [2] suggest that in order to have a randomized sample, it is needed $|edge| \times ln(|edge|)$, the quality measurement(PPV_k) shows its not necessary so much, this will be the main subject of this thesis. Regarding the number of samples, it's recommended 10,000 which is known as a safe number, in order to evaluate the heuristic this number will be used, since there is no reliable estimation for that so far.

3.2 Cluster Implementation

The cluster implementation, meant to run on a cluster with several nodes, is a c implementation and known as the fastest *Link Assessment Problem* implementation done so far. It was developed aiming parallelization as co-occurrence as swap, for the reason that both algorithm together correspond to more than 95% of the whole program time consuming. Even though it is fast, the quality measurement (shall be discussed in the next chapter) shows that $|edge| \times ln(|edge|)$ swaps per sample are more than enough, reducing this number is the thesis propose.
By means of parallelization, each process make a copy of the original graph, its threads will deal with parallelization algorithm for swap and co-occurrence calculation. Each process will be held by a node, which will perform $\frac{10{,}000}{n}$ samples where n is the number of nodes, hence the number of samples will be divided by the number of nodes(*See figure 5*).
To have a better understating of how the cluster implementation is computed, this chapter discusses its data structure, co-occurrence algorithm, swap algorithm and evaluation.

Listing 1: Graph Data Structure

```
typedef struct graph {
   char**   eventList;
   unsigned int*  actorAdjList;
   unsigned int*  actorAccumulatedDegrees;
   unsigned int*  actorEdgeMap;
   unsigned int*  adjMatrix;
   unsigned int   numberOfActors;
   unsigned int   numberOfEvents;
   unsigned int   numberOfEdges;
}GRAPH;
```

The `Graph Data Structure` stores the information related to the graph, i.e the relation between movies and users, let's have a look into the each of its components:

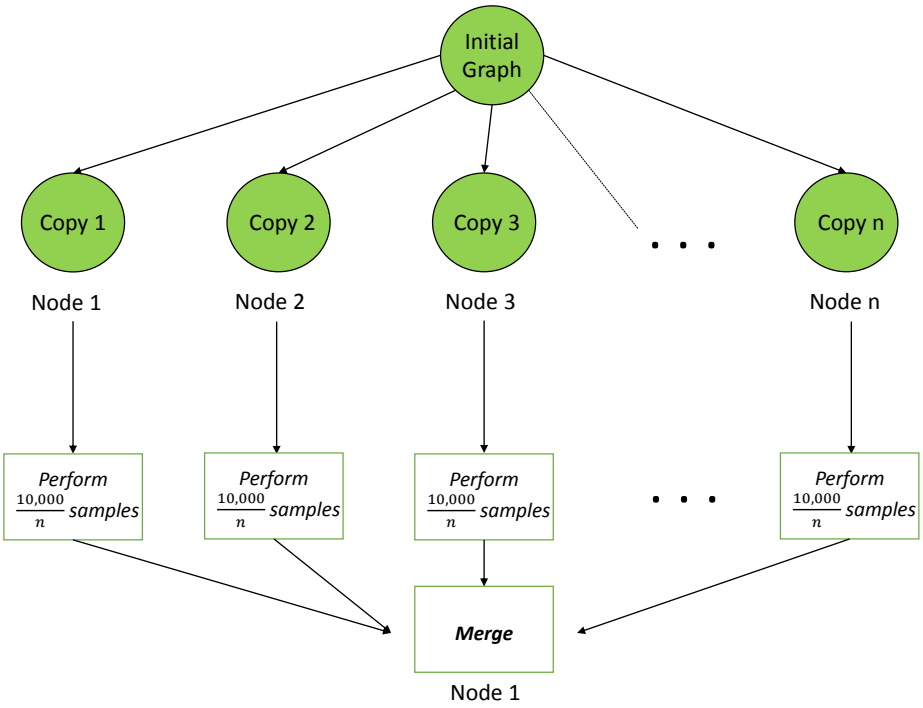

Figure 5: Cluster Implementation

1. `eventList`
   Movie map according to the bipartite graph. This structure is necessary, since the movies ids from the input are different from the program.

2. `actorAdjList`
   This scructure is responsible for storing the list of movies that belong to a given actor.

3. `actorAccumulatedDegrees`
   Stores the sum of the previous movies degrees, e.g, if we are interested in the degree number of a given movie m_k, we have to calculate the difference:
   $actorAccumulatedDegrees[m_k + 1] - actorAccumulatedDegrees[m_k]$

4. `actorEdgeMap`
   For each edge, this vector stores the users that belongs to this edge,

such a structure is useful for the swapping algorithm.

5. `adjMatrix`
   Is the binary matrix of edges in a way movie x user.

6. `numberOfActors, numberOfEvents and numberOfEdges`
   Store respectively the number of users, number of movies and number of edges from the bipartite graph.

`Temporary Data Structure` is the name given to the structure that stores the responsible variables to generate the output of the program:

Listing 2: Temporary Data Structure

```
typedef struct tmpresult {
   unsigned int** lastCooc;
   unsigned long** coocSum;
   unsigned long** coocSquareSum;
   unsigned int** pValue;
   double** zScore;
}TMPRESULT;
```

1. `lastCooc`
   Represents $cooc_{sample}$, i.e the calculated co-occurrence right after swapping.

2. `coocSum and coocSquareSum`
   They stores the cumulative $cooc_{sample}$ and $cooc^2_{sample}$, both used to calculate $z - score$.

3. `zScore and pValue`
   Are responsible for storing respectively the $z - score$ and $p - value$ results.

In order to compute the co-occurrence algorithm, it is necessary to use the `actorAdjList`. As explained before, the actor adjacency list stores the information related to a given user, i.e the movies this user ranked. One user is fixed and then the last movie and the first movie of his list are also fixed, the algorithm go from the end till the beginning of this user list incrementing his co-occurrence for each movie in this list, this is done for all movies in this user's list. The algorithm is shown below:

Listing 3: Co-occurrence Algorithm

```
1   for (j = last_movie; j > first_movie; j--)
2      for (i = j-1; i > first_movie-1; i--)
3         compute_cooc(i,j);
```

Let's consider an example for the co-occurrence calculation. Consider the `actorAdjList` shown below(*figure 6*), let's suppose we want to compute the co-occurrence for user id 5, the movies ids he ranked are 45, 54, 65 and 76. First we select movie id 76 and then go backwards, for each position of this vector we have to increment the co-occurrence, for movie 76 the following co-occurrences are incremented by 1:

- $cooc(76, 65)$
- $cooc(76, 54)$
- $cooc(76, 45)$

This procedure is made for movie ids 65 and 54. For movie id 65 we have the following co-occurrences incremented:

- $cooc(65, 54)$
- $cooc(65, 45)$

For movie id 54 only:

- $cooc(54, 45)$

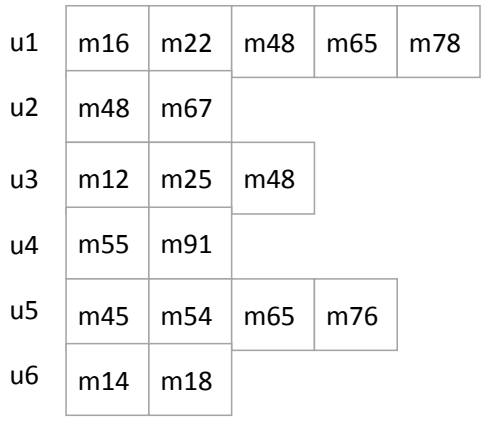

Figure 6: Actor Adjacency List Example

To perform swaps, the `adjMatrix` is used, it's done as the usual way, selecting 2 edges at random from the `edgeMap`, each one containing a movie and a user. Let's say A and B with respectively coordinates (i, j) and (l, k), the swap is done and computed as swap only if

$$\begin{cases} \texttt{adjMatrix}(i,j) = \texttt{adjMatrix}(l,k) = 1 \\ \texttt{adjMatrix}(i,k) = \texttt{adjMatrix}(j,l) = 0 \end{cases}$$

After swapping, the `actorAdjList` has to be modified.

Listing 4: Swap Algorithm

```
void randomSwap(){
   if (adjMatrix[i][j]!=1 || adjMatrix[l][k]!=1)
      return;
   else if(adjMatrix[i][k]!=0 || adjMatrix[j][l]!=0)
      return;
   else{
      adjMatrix[i][k]   = 1;
      adjMatrix[j][l]   = 1;
```

```
9         adjMatrix[i][j]    = 0;
10        adjMatrix[l][k]    = 0;
11        }
12 }
```

To evaluate the cluster implementation and the heuristic that will be presented, it was used the following good rating *Netflix Datasets*:

- $1k$ users
- $10k$ users
- $20k$ users
- $100k$ users

The experiments were ran on 2 x Intel® Xeon® Processor E5-2670 (20M Cache, 2.60 GHz, 8.00 GT/s Intel® QPI), using 16 cores. Each experiment using 10000 samples and $|edge| \times ln(|edge|)$ swaps per sample.

$Dataset$	$\frac{Swaps}{Sample}$	$Cooc_{time}$	$Swap_{time}$	$Total_{time}$
$1k$	2.5×10^6	0.56h	0.05h	0.62h
$10k$	2.4×10^7	3.32h	1.34h	4.67h
$20k$	4.9×10^7	5.54h	3.06h	8.52h
$100k$	2.6×10^8	22.42h	19.08h	41.54h

Table 1: Experiments time for 10,000 samples and $|edge| \times ln(|edge|)$ swaps

The heuristic presented in this thesis will suggest search for a better number for the swap number, based on the assumption that in order to have a reliable result quality it's not necessary to perform $|edge| \times ln(|edge|)$ swaps.

4 Phase Transition

Consider the Netflix dataset, which consist of users and movies, these users rank the movies with a range from 1 to 5, which represents very bad to very good movies in their point of view. This dataset, that will be our object of study in this thesis, is composed by 17,770 movies and 478,000 users, with the *Link Assessment* algorithm, we can make a *one-mode projection* of movies and stablish a relationship between these movies and tell how strong this relationship is. However how is it possible to be sure that all relationships with a huge similarities coefficients are reliable? Regarding this question, there comes a necessity to have a toll to measure the *Link Assessment* algorithm output.

4.1 PPV_k(Positive Predicted Values)

This tool is called PPV_k *(Positive Predictive Value)*, as defined in [3] and [4], it is equal to the *fraction of true positives among the k top-ranked actor similarities, where k is the number of elements in the ground truth* . It can be expressed as:

$$PPV_k = \frac{number\ of\ true\ positives}{number\ of\ true\ positives + number\ of\ false\ positives}$$

The ground truth is a movie x movie file, it is a selection by human of what movies are expected to be similar. The way that the PPV_k is calculated in the Link Assessment problem is by selecting from the output only the movies referenced by the ground truth, e.g if one edge of the ground truth is (1 2), all combination of 1 and 2 must be selected from the output. After that there will be a sort in order that the top of these selected edges have the smallest *p-value*, i.e the top holds the most similar pairs. Among the pairs that have the same *p-value*, there will be another sorting, now bringing to the top the highest *z-score* pairs. From this rank, there will be considered the top ground truth edges, i.e a selection of the same number of edges of the ground truth for the most similar pairs. The ppv will be given by the fraction of movies that are in this rank and in the ground truth divided by the number of edges of the ground truth.

4.2 PPV_k over Number of Swaps

To understand how swaps are tied to the output quality, the PPV_k was measured for different number of swaps as a parameter, using different Netflix Datasets (1k,10k and 20k users) and Movie Lens Dataset(9153 users, 15185 movies and 1077270 edges). The transition, which all of these experiments face, is called *phase transition*, in this transition, for all datasets, the PPV_k stays steady in zero till a given value, let's call this value $pt1_{dataset}$, where *dataset* is the dataset of interest. After $pt1_{dataset}$, the quality over swap will increase continuously till the point it is stabilized, let's call this point $pt2_{dataset}$, to this transition we call phase transition. In order to try to explain this transition, there was drew, for each plot, 2 additional lines, they represent $|user|ln(|user|)$ and $|edge| \times ln(|edge|)$.
The goal in drawing these lines is to show that in order to have a good quality output, it is not necessary $|edge| \times ln(|edge|)$ swaps. Let's take an example from the Movie Lens dataset (shown in figure 7). We can notice that $pt1_{MLens}$ is very close to $|user|ln(|user|)$ and $pt2_{MLens}$ takes place before $|edge| \times ln(|edge|)$. This behaviour is the same for all the datasets. Concerning this, we can make a heuristic interval, i.e the interval, which there is a good estimative for swap number and is less than $|edge| \times ln(|edge|)$. This interval is $[user|ln(|user|), edge|ln(|edge|)]$.

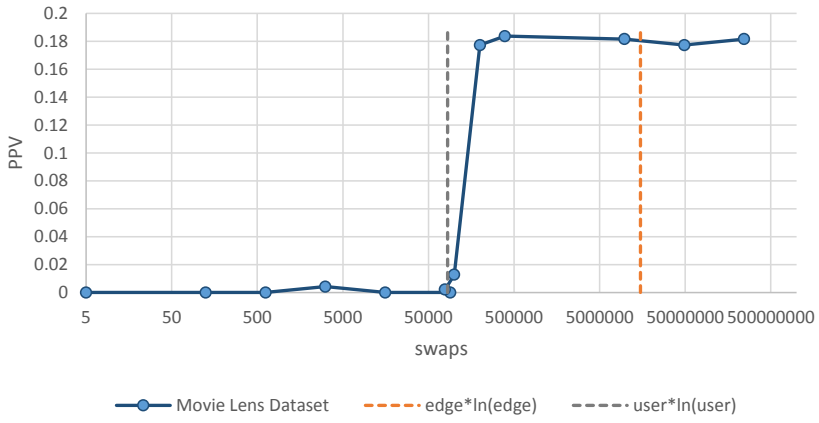

Figure 7: Movie Lens PPV over Swaps

Another important thing to notice is that the datasets will face the phase

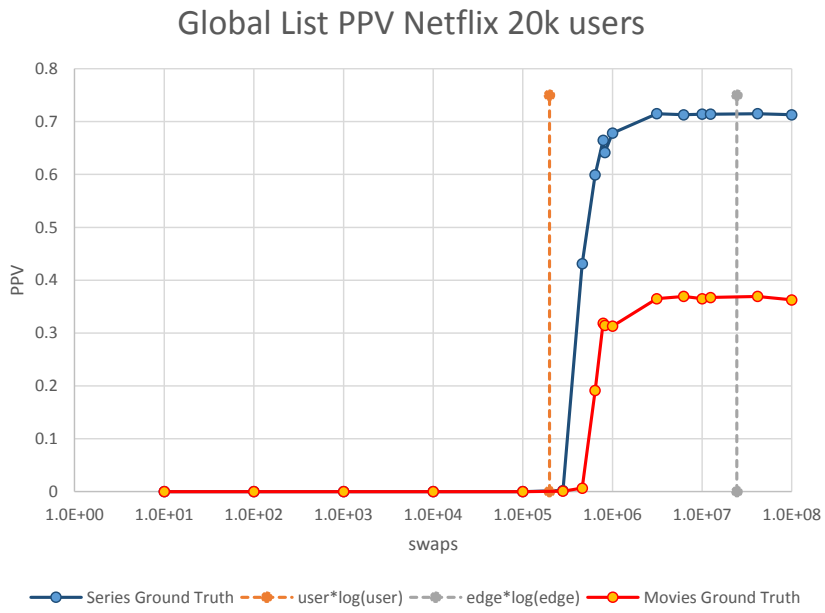

Figure 8: 20k Series vs Movie Ground Truth

transition at the same number of swaps for all ground truth used, i.e it is a fixed value (*see figure 8*). It's about 10^6 for the series and movies ground truth, the first represent similar series and the second similar movies.
Figure 9 shows the phase transition for the Netflix Datasets. They will have the same behaviour as Movie Lens, for $|user|ln(|user|)$ it's about pt_1 for these netflix dataset and it establishes for a swap number less than $|edge| \times ln(|edge|)$. We can also notice that the larger the number of users get, the more swaps are needed to cross the phase transition.
We can conclude from this experiments that for sure the optimal number of swaps, i.e the number of swaps that results in a good quality output, is between $|user|ln(|user|)$ and $|edge| \times ln(|edge|)$. This experiments will represent the heuristic interval, the interval which the heuristic will look for the optimal swap number.

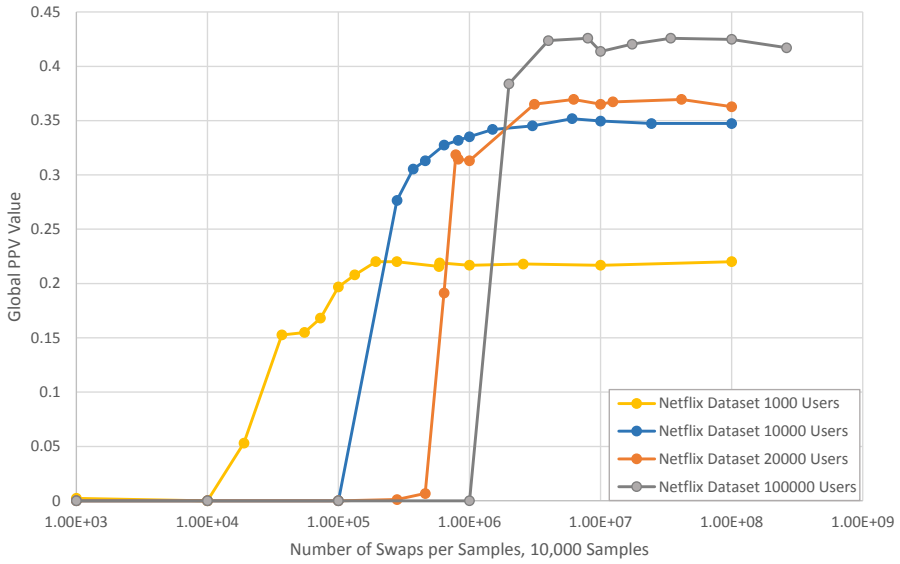

Figure 9: All Datasets PPV over Swaps

5 Testing the Graph Convergence

For a given number of swaps per sample, the $cooc_{FDSM}$ and the standard deviation of $cooc_{FDSM}$(let's say $stdev_{FDSM}$) have been converged for all the combination of movie pairs, i.e there is no use in increasing the swap number once this number is reached. The PPV_k also has converge for swap numbers equal or bigger than this number, we know that $|edge| \times ln(|edge|)$ is a sufficient number, however the PPV_k shows this number is more than enough.

The fact that $cooc_{FDSM}$ and $stdev_{FDSM}$ have converged implies that the graph is sufficiently randomized, thus swapping for bigger swap numbers than this ideal number will be only waste of time consuming. This section will show two systematic methods θ_m and θ_s that identify whether a graph has converged for a given swap number. The next section will apply this method to the heuristic for finding the swap ideal number.

5.1 Graph Convergence Premises

There will be three premisses to know whether a graph has converged or not. They will be evidences that the graph has converged for a given swap number. Let's present these premises:

1. $cooc_{FDSM}$ has converged for all movie pairs of the dataset.
2. $stdev_{FDSM}$ has converged for all movie pairs of the dataset.
3. The graph is sufficiently randomized.

Concerning these premises, we can demonstrate that the graph converges by applying the convergence tests for selected movie pairs. The convergence of $cooc_{FDSM}$ means that the mean of all the calculated $cooc_{sample}$ has the same value or a very close value if swapping with the ideal swap number or higher values, the same for $stdev_{FDSM}$. This because $p-value$ and $z-score$ depend on both values, once they have converged, the quality of the output will be reliable. The last premise is the main objective of the *link assessment problem*, i.e have enough samples of sufficiently randomized graphs in order to calculate the *one-mode projection*.

5.2 Premises Experiment

To analyse better these premises, there was done an experiment to test all of them . There were used three netflix datasets: 1k, 10k and 20k users with 10420, 15844, 16837 movies respectively.
For each dataset there was selected four movie pairs with the same degree, e.g having four movie pairs identified by *(movie1,movie2), (movie1, movie3), (movie4,movie2) and (movie4,movie3)*, the degree function $\xi(m_i)$, where m_i is a movie id, applied to these movies is :

- $\xi(movie1) = \xi(movie4)$
- $\xi(movie2) = \xi(movie3)$

For the reason that they have the same degree, once the graph is sufficiently randomized(*premise 3*), their $cooc_{FDSM}$ and $stdev_{FDSM}$ must converge to the same value(*premises 1 and 2*).
The experiments begin with 1 swap per sample, when $10,000$ samples are

taken, the $cooc_{FDSM}$ and $stdev_{FDSM}$ are calculated and the swap number multiplied by 10. This procedure is done till the swap number reaches a number bigger than $|edge| \times ln(|edge|)$. The results are shown in *figures 10,11,12,13,14 and 15*.

The general behaviour for all datasets tested is the same, for a swap number higher than $|user| \times ln(|user|)$ and smaller than $|edge| \times ln(|edge|)$, the dataset converges, i.e both $cooc_{FDSM}$ and $stdev_{FDSM}$ reach a stable value showing that it's not necessary $|edge| \times ln(|edge|)$ swaps for the sample to be randomized.

If we compare the swap number, for which $cooc_{FDSM}$ and $stdev_{FDSM}$ are stable between the PPV_k plot for any of the datasets, we will see that this number is very close to $p_{t2_{dataset}}$, i.e the swap number after crossing the *phase transition*, showing that the convergence is related to the output quality. This is shown in figure 10, when the movie pairs with the same degree reach the co-occurence convergence is exactly $pt2_{10k}$, demonstrating that there is a relationship between PPV_k and convergence.

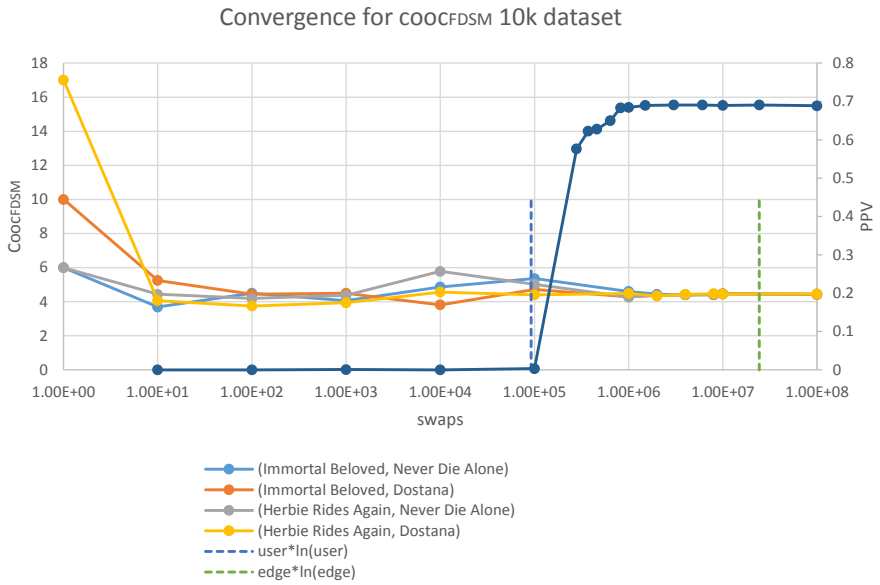

Figure 10: Relation Between Co-occurrence Convergence and PPV Convergence

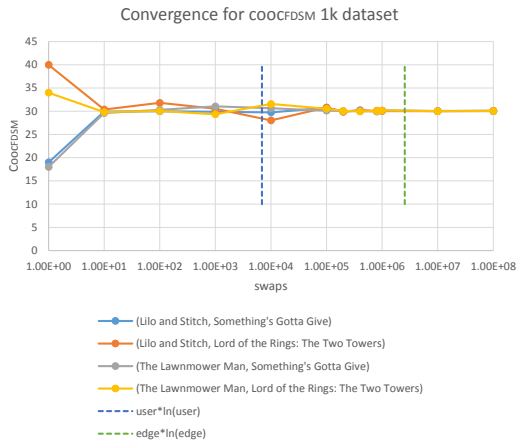

Figure 11: $cooc_{FDSM}$ over swaps (1k users)

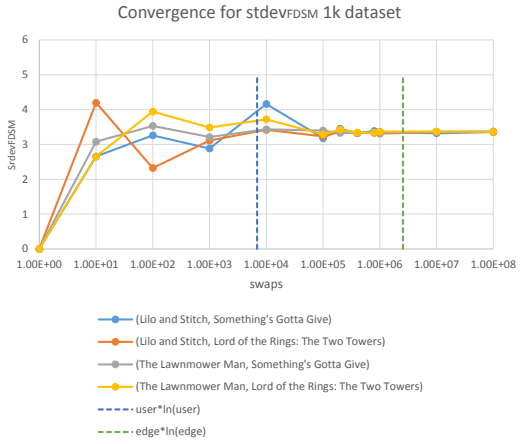

Figure 12: $stdv_{FDSM}$ over swaps (1k users)

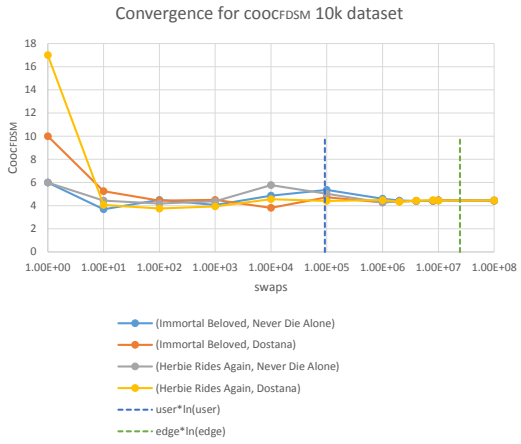

Figure 13: $cooc_{FDSM}$ over swaps (10k users)

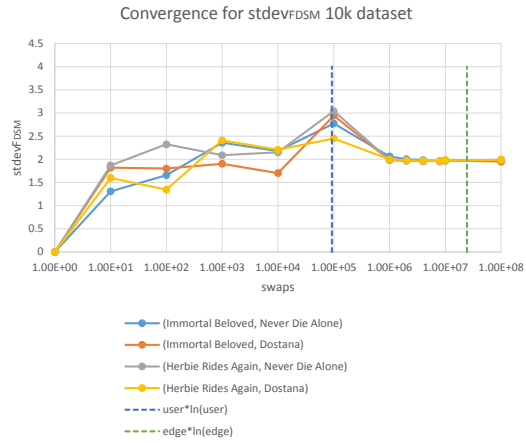

Figure 14: $stdev_{FDSM}$ over swaps (10k users)

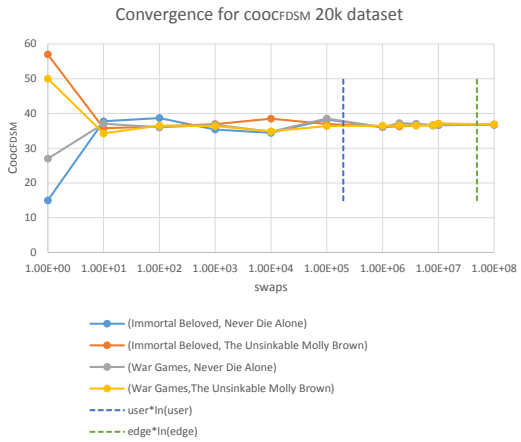

Figure 15: $cooc_{FDSM}$ over swaps (20k users)

Figure 16: $stdev_{FDSM}$ over swaps (20k users)

By the time the program is executing it is not possible to have information from the PPV_k to know whether the number of swaps performed per each sample is enough to have a good quality output. However it is possible to detect whether $co-occurrence_{FDSM}$ and $stdev_{FDSM}$ have converged or not, if so, it will imply that the PPV_k has also converged.
This result will be useful to create a heuristic to search for the optimal swap number per sample. Later there will be presented methods that test if $co-occurrence_{FDSM}$ and $stdev_{FDSM}$ have converged. These methods will be the most important tool to build the heuristic that searches for the optimal swap number.

5.3 Selection of Movie Pairs with Same Degree

In order to build the convergence tests, it is necessary to select some movies pairs. The convergence test proposed is to select 24 groups of movie pairs, let's say $G = \{g_0, g_1, ..., g_{24}\}$, where g_r is the group id with 4 movie pairs that have the same degree. This test will have the amount of 96 movie pairs, for that let's define a set of movie pairs $\Pi = \{\mu_0, \mu_1, ..., \mu_{95}\}$, where μ_n will represent a movie pair. The convergence test will be applied to each g_r , i.e all g_r will have their $cooc_{FDSM}$ and $stdev_{FDSM}$ compared after a given number of swaps. Relying on the premises, once the graph is sufficiently randomized(after a given number of swaps per sample), the movies with the same degree must converge to the same $cooc_{FDSM}$ and $stdev_{FDSM}$. This is what *table 2* shows, for the netflix dataset with 1k users, this table shows, for 6 groups(the total amount is 24), the ξ (degree) for each movie of the group, their initial $cooc_{initial}$ and $cooc_{FDSM}$ after a sufficient number of swaps per sample and using 10,000 samples is calculated. By doing that, $cooc_{FDSM}$ must converge to the same value.

g_r	m_i	m_j	$\xi(m_i)$	$\xi(m_j)$	$cooc_{init.}$	$cooc_{FDSM}$
g_0	Lilo and Stitch	Something's Gotta Give	52	249	19.0	30.08
	Lilo and Stitch	Lord of the Rings	52	249	40.0	30.10
	The Lawnmower Man	Something's Gotta Give	52	249	18.0	30.02
	The Lawnmower Man	Lord of the Rings	52	249	34.0	30.05
g_1	Something's Gotta Give	Dragonheart	249	69	29.0	38.76
	Something's Gotta Give	About Schmidt	249	69	27.0	38.83
	Lord of the Rings	Dragonheart	249	69	48.0	38.80
	Lord of the Rings	About Schmidt	249	69	25.0	38.79
g_2	Dragonheart	Rambo: First Blood	69	58	29.0	13.97
	Dragonheart	The Pacifier	69	58	21.0	13.94
	About Schmidt	Rambo: First Blood	69	58	8.0	13.98
	About Schmidt	The Pacifier	69	58	7.0	13.99
g_3	Rambo: First Blood	The Game	58	91	30.0	17.18
	Rambo: First Blood	Robin Hood	58	91	31.0	17.22
	The Pacifier	The Game	58	91	22.0	17.18
	The Pacifier	Robin Hood	58	91	22.0	17.17
g_4	The Game	Taking Lives	91	124	42.0	31.07
	The Game	Cool Hand Luke	91	124	37.0	31.06
	Robin Hood	Taking Lives	91	124	36.0	31.01
	Robin Hood	Cool Hand Luke	91	124	26.0	31.06
g_5	Taking Lives	The Deer Hunter	124	85	22.0	29.39
	Taking Lives	Cocoon	124	85	28.0	29.42
	Cool Hand Luke	The Deer Hunter	124	85	42.0	29.41
	Cool Hand Luke	Cocoon	124	85	26.0	29.45

*Table 2: Groups of movies with the same degree for 1k users netflix dataset, edge*ln(edge) swaps and 10,000 samples*

5.4 θ_m and θ_s Convergence Tests

There is two different convergence tests, they are θ_m and θ_s, the first is responsible for testing the convergence of $cooc_{FDSM}$ of each g_r of G, while the second is responsible to test $stdev_{FDSM}$ of the same groups of movie pairs.

These tests will generate a number, that compared to a threshold α, will tell if the swap number per sample is enough or not. The α threshold will be given as input, the more smaller this number is, the more close $cooc_{FDSM}$ or $stdev_{FDSM}$ of the movie pairs belonging to g_r will be. *Figures 17 and 18* show how is the behaviour of these test to the 10k netflix dataset compared with PPV_k for the same dataset. We can notice the more the quality establishes, the more θ_s and θ_m go close to zero, showing the relation between convergence and quality, i.e the outputs will have a good quality if the graph has already converged.

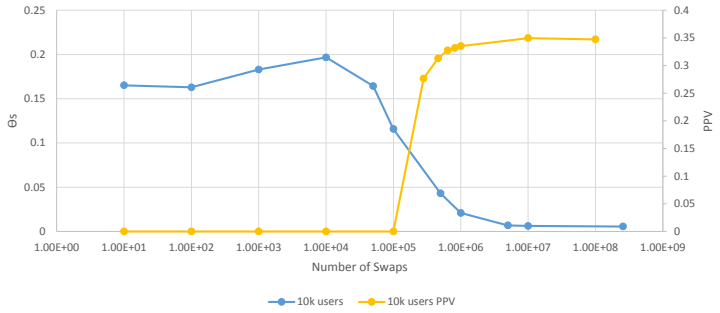

Figure 17: θ_s over swaps x PPV (10k users)

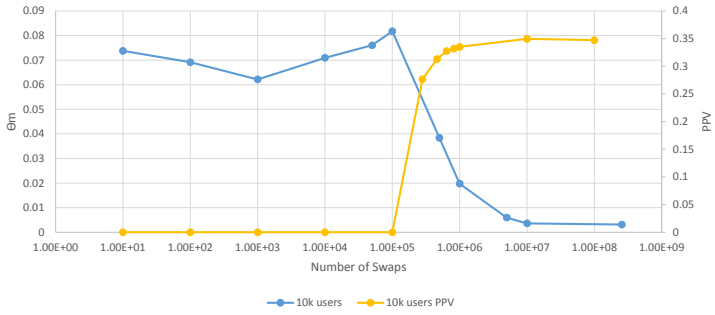

Figure 18: θ_m over swaps x PPV (10k users)

Let's define how to calculate θ_m and θ_s for each g_r of G. θ_m is defined as the mean of the standard deviation of each g_r $cooc_{FDSM}$ that belong to G divided by the mean of each g_r, the same for θ_s, however instead of $cooc_{FDSM}$, it is $stdev_{FDSM}$. In order to calculate θ_m and θ_s, let's define some auxiliar functions $\Upsilon_m(r)$ and $\Psi_m(r)$.
$\Upsilon_m(r)$ is defined as the sum of the $cooc_{FDSM}$ of each movie pair of the group r,

$$\Upsilon_m(r) = cooc_{FDSM}(\mu_{4\times r}) + cooc_{FDSM}(\mu_{4\times r+1}) + cooc_{FDSM}(\mu_{4\times r+2}) + cooc_{FDSM}(\mu_{4\times r+3})$$

The movie pair ids of this group are represented as μ_r. The $\Upsilon_m(r)$, however, represent the sum of the squares of $cooc_{FDSM}$ that belong to a group r and same notation for movie pairs,

$$\Psi_m(r) = cooc^2_{FDSM}(\mu_{4\times r}) + cooc^2_{FDSM}(\mu_{4\times r+1}) + cooc^2_{FDSM}(\mu_{4\times r+2}) + cooc^2_{FDSM}(\mu_{4\times r+3})$$

With $\Upsilon_m(r)$ and $\Psi_m(r)$ calculated for a group r, we can easily calculate the mean and the standard deviation of this group, both called $Mean_{PSD_m}(r)$ and $Std_{PSD_m}(r)$ respectively, here PSD represents *pairs with the same degree*. $Mean_{PSD_m}(r)$ is,

$$Mean_{PSD_m}(r) = \frac{\Upsilon_m(r)}{4}$$

The standard deviation is defined as

$$standard\ deviation\ = \frac{\sqrt{\sum_{i=1}^{N}(x_i - x_{mean})^2}}{N-1}$$

where X_i is each point and x_{mean} is the mean of these points. Let's try to expand $\sum_{i=1}^{N}(x_i - x_{mean})^2$:

$$\sum_{i=1}^{N}(x_i - x_{mean})^2 = (x_1 - x_{mean})^2 + (x_2 - x_{mean})^2 + ... + (x_N - x_{mean})^2$$

$$= \left(x_1^2 - 2 \times x_{mean} \times x_1 + x_{mean}^2\right) + \left(x_2^2 - 2 \times x_{mean} \times x_2 + x_{mean}^2\right) + ...+ \left(x_N^2 - 2 \times x_{mean} \times x_N + x_{mean}^2\right)$$

$$= (x_1^2 + x_2^2 + ... + x_N^2) - 2 \times x_{mean} \times (x1 + x2 + ... + x_N) + N \times x_{mean}^2$$

Considering our approach, we have $Std_{PSD_m}(r)$ as the standard deviation of the co-occurrence FDSM of all groups and the following terms:

1. $(x_1^2 + x_2^2 + ... + x_N^2) = \Psi_m(r)$
2. $(x1 + x2 + ... + x_N) = \Upsilon_m(r)$
3. $x_{mean} = Mean_{PSD_m}(r)$
4. $N = 4$

thus $Std_{PSD_m}(r)$ is

$$Std_{PSD_m}(r) = \sqrt{\frac{\Psi_m(r) - 2 \times \Upsilon_m(r) \times Mean_{PSD_m}(r) + 4 \times Mean_{PSD_m}^2(r)}{3}}$$

θ_m is the mean of the calculated normalization of each group r $\Gamma(r)$, thus with this normalizations yet calculated for each r, we calculate their mean, with r between the interval $[0, 23]$:

$$\theta_m = \frac{1}{24} \times \sum_{i=0}^{23} \left(\frac{Std_{PSD_m}(r)}{Mean_{PSD_m}(r)}\right)$$

θ_s is analogously to θ_m, however instead of $cooc_{FDSM}$, it is used $stdev_{FDSM}$, i.e θ_s is the mean of the normalized standard deviation of all $stdev_{FDSM}$ that belong to a group r, with r between $[0, 23]$. the mean of $stdev_{FDSM}$ of each group r of G is

$$Mean_{PSD_s}(r) = \frac{\Upsilon_s(r)}{4}$$

and its standard deviation,

$$Std_{PSD_s}(r) = \sqrt{\frac{\Psi_s(r) - 2 \times \Upsilon_s(r) \times Mean_{PSD_s}(r) + 4 \times Mean_{PSD_s}^2(r)}{3}}$$

where $\Upsilon_s(r)$ is

$$\Upsilon_s(r) = stdev_{FDSM}(\mu_{4\times r}) + stdev_{FDSM}(\mu_{4\times r+1}) + stdev_{FDSM}(\mu_{4\times r+2}) + stdev_{FDSM}(\mu_{4\times r+3})$$

And $\Psi_s(r)$

$$\Psi_s(r) = stdev^2_{FDSM}(\mu_{4\times r}) + stdev^2_{FDSM}(\mu_{4\times r+1}) + stdev^2_{FDSM}(\mu_{4\times r+2}) + stdev^2_{FDSM}(\mu_{4\times r+3})$$

Finally $\theta_s is$

$$\theta_s = \frac{1}{24} \times \sum_{i=0}^{23} \left(\frac{Std_{PSD_s}(r)}{Mean_{PSD_s}(r)} \right)$$

Both θ_m and θ_s test identify id the graph converged for a given number of swaps with respect to a threshold α given as input. These tests will be useful for the heuristic to define the number of swaps needed.

6 Heuristic for Swap Number Selection

The approach for the heuristic is running this algorithm first then the cluster implementation itself with the swap number found by the heuristic. The heuristic algorithm consists in running different number of swaps in some movie pairs selected from the dataset, calculate their co-occurence and then apply to these movies the convergence test θ_m or θ_s(shown on the previous section). The swap parameter will be changed by the heuristic till θ_m or θ_s reach a given threshold α. Note that the heuristic is restricted only in swapping time, since the co-occurence is only for a few movie pairs.
Since the number of samples is still unknown, both heuristic and cluster implementation will run with the number recommended 10,000 samples.

6.1 Searching for Swap Ideal Number

To understand how the heuristic works, let's imagine that the heuristic will walk through the PPV_k plot (shown on the section 4), the heuristic will start at a given swap and then apply the convergence test, once this test is not satisfied, the heuristic will select a bigger swap number and run the algorithm again till the test is satisfied. After the swap number is found, there will be a backwards walk in order to select the minimal value possible. The interval, which we will look for the swap ideal number was already defined, it is $[user| \times ln(|user|), |edge| \times ln(|edge|)]$ and the first swap number to be tested will be the middle point of this interval, i.e the geometric mean of this interval:

$$\sqrt{(|user| \times ln(|user|) \times (|edge| \times ln(|edge|))}.$$

For those searches we have to better define them:

1. **Forward Search**
   The forward will set a higher swap number if the current swap number does not satisfy α. It fixes the $|edge| \times ln(|edge|)$ as a limit and perform a geometric mean between this number and the current swap. The forward search is defined as:

$$F(k) = (|user| \times ln(|user|))^{\frac{1}{(2^{k+1})}} \times (|edge| \times ln(|edge|))^{\sum_{n=1}^{k+1} \frac{1}{(2^n)}}$$

where k is the number of forwards searches counted from the starting value.

2. **Backward Search**
   The backward search searches for a swap number that does not satisfy α. This function is harder to define, since it is tied to $F(k)$, for different number of k it has different values:

$$B(l) = \begin{cases} (|edge|ln(|edge|)^{\frac{1}{(2^{k+1})}}(|user|ln(|user|))^{\sum_{n=1}^{k+1}\frac{1}{(2^n)}},\ if\ k=0 \\ F(1)^{\frac{1}{(2^l)}}\ [(|user|ln(|user|))(|edge|ln(|edge|))]^{\frac{1}{2^{(l+1)}}}\ ,\ if\ k=1 \\ \sqrt{F(k)F(k-1)},\ if\ k>1\ and\ l=1 \\ [F(k)F(k-1)]^{\frac{1}{2^l}}\ F(k-1)^{\frac{1}{2^{(l-1)}}},\ if\ k>1\ and\ l>1 \end{cases}$$

6.2 Heuristic Example

Let's have a look at an example, *figure 20* show an example of how the heuristic algorithm works for $|user|\times ln(|user|) = 10^5$ and $|edge|\times ln(|edge|) = 10^7$. Suppose the black line are swap numbers, which will result in bad quality outputs (low PPV_k) and the green line swap numbers that will result in good quality outputs (high PPV_k). The heuristic has to select one point from the green line, for that it will start at the middle point of $|user|ln(|user|)$ and $|edge| \times ln(|edge|)$, in this case 10^6, and then apply, to a few combination of movies, the convergence test θ_m or θ_s (parameter chosen by the user). This test is not satisfied, since for that swap number the graph did not converged, i.e the resulting output concerning for this swap number results in low quality. Since the convergence test was not satisfied, the algorithm has to perform a forward search, i.e increase the swap number. For that, the algorithm calculates the geometric mean between $\sqrt{(|user|ln(|user|)(|edge|ln(|edge|))}$ and $|edge| \times ln(|edge|)$, as defined it is $F(1)$ and can be calculated as:

$$F(1) = (10^5)^{\frac{1}{2^2}}(10^7)^{\frac{1}{2^1}+\frac{1}{2^2}} = 10^{\frac{5}{4}+\frac{7}{2}+\frac{7}{4}} = 10^{\frac{13}{2}} \approx 3.16 \times 10^6$$

When applying the convergence test to movie selection for $F(1)$ swaps, it is not satisfied. It's necessary to increase more swaps per sample, for that the algorithm will run again with $F(2)$:

$$F(2) = (10^5)^{\frac{1}{2^3}}(10^7)^{\frac{1}{2^1}+\frac{1}{2^2}+\frac{1}{2^4}} = 10^{\frac{5}{8}+\frac{7}{2}+\frac{7}{4}+\frac{7}{8}} = 10^{\frac{27}{4}} \approx 5.6 \times 10^6$$

After applying the convergence test once again, it is satisfied so $F(2)$ swaps per sample are enough to generate a good quality output. However in order to reduce the cluster implementation as much as possible, we have to select the minimal swap number, hence we have to perform a search restricted to the limits $F(1)$ and $F(2)$, the first known as an insufficient swap number and the second as a sufficient swap number. In order find the swap ideal number within this interval, the algorithm calculates the $B(1)$ for $k = 2$:

$$B(1) = \sqrt{F(2)F(1)} = \sqrt{(5.6 \times 10^6) \times (3.16 \times 10^6)} \approx 4.2 \times 10^6$$

It still represents an output of good quality, i.e the selected movie pairs pass in the convergence test, consequently one more step back is necessary:

$$B(2) = [F(2)F(1)]^{\frac{1}{2^2}}\, F(1)^{\frac{1}{2(2-1)}} = \left[17.696 \times 10^{12}\right]^{\frac{1}{4}} (3.16 \times 10^6)^{\frac{1}{2}} \approx 3.64 \times 10^6$$

This swap point is inside the black area, thus the chosen swap number will be 4.2×10^6, for the reason that it is inside the green area and is the minimal swap number found.
With the heuristic output, the cluster implementation can run with the found swap number per sample. If we compare the usual value ($|edge| \times ln(|edge|)$) with the found value, we will have 10^7 against 4.2×10^6, the found value is ≈ 2.4 times smaller than the usual value.

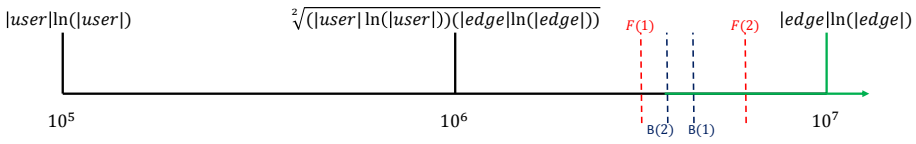

Figure 19: Heuristic Example

6.3 Theoretical Improvement

In order to calculate the heuristic theoretical improvement, let's first define the cluster implementation time consuming itself without the heuristic C_{time}:

$$C_{time} = cooc_{time} + swap_{time}$$

This time will be different for different number of samples and swaps, for now let's consider that both cluster implementation and heuristic run with 10,000 samples and $|edge| \times ln(|edge|)$ swaps per each sample. The heuristic time consuming will be very close to $swap_{time}$ when running with $|edge| \times ln(|edge|)$ swaps each sample, however the heuristic amount of swaps will be less than $|edge| \times ln(|edge|)$, thus the heuristic time consuming H_{time} will be:

$$H_{time} = \frac{amount\ of\ swaps\ during\ the\ heuristic \times swap_{time}}{|edge| \times ln(|edge|)}$$

When running the cluster implementation combined to the heuristic, we have to consider a new time consuming HC_{time} that will be given by $cooc_{time}$,H_{time} and the predicted swap number:

$$HC_{time} = cooc_{time} + H_{time} + \frac{predicted\ swap\ number \times swap_{time}}{|edge| \times ln(|edge|)}$$

The swap time improvement will depend on $F(k)$ and $B(l)$, for each dataset it will give a different improvement, since they will produce different k and l. Let's define a function $I(k,l)$ to calculate the swap program improvement, consisting of the combination of the heuristic and the cluster implementation:

$$I(k,l) = \left[1 - \left(\frac{F_{swaps}(k) + B_{swaps}(l) + predicted\ swap\ number}{|edge| \times ln(|edge|)}\right)\right]$$

where $F_{swaps}(k)$ is the amount of swaps performed by the forward search, it is given by:

$$F_{swaps}(k) = F(1) + F(2) + ... + F(k) = \sum_{n=1}^{k} F(n)$$

and analogously for $B_{swaps}(l)$

$$B_{swaps}(l) = B(1) + B(2) + ... + B(l) = \sum_{n=1}^{l} B(n)$$

7 Evaluation

7.1 Evaluation in Time Consuming

To evaluate the improvement in time consuming of the cluster implementation by using the heuristic, the cluster implementation was ran with the following inputs: the netflix dataset with 1,000, 10,000, 20,000 and 100,000 users taking 10,000 samples and $|edge| \times ln(|edge|)$ swaps per sample (usual parameters without the heuristic). The results are shown in *table 1*, section 3. There was established a comparison between these time consuming and the time consuming with respect to the heuristic plus the cluster implementation with the predicted swap number instead of $|edge| \times ln(|edge|)$ swaps per sample.
The metrics used to evaluate the improvement in time consuming are:

1. **Theoretical Swap Improvement**
   The theoretical swap improvement $I(k, l)$ is calculated based on the amount of swaps wasted during the forward and backward search (defined in section 6)

2. **Real Swap Improvement**
   The real swap improvement consider the swap time consuming for $|edge| \times ln(|edge|)$ swaps per sample and the time wasted in the heuristic plus the swapping time for the heuristic predicted value

   $$I_{swap_{real}} = \left[1 - \frac{Heuristic_{time} + swap_{time}(predicted\ swap\ number)}{swap_{time}(|edge| \times ln(|edge|))}\right]$$

   Where,

   - $Heuristic_{time}$ is the time waste to perform the heuristic
   - $swap_{time}(predictedswapnumber)$ is the time used for swapping by the cluster implementation, but instead of $|edge| \times ln(|edge|)$, the predicted value found by the heuristic.
   - $swap_{time}(edge| \times ln(|edge|))$ is the swapping time for the cluster implementation performing $|edge| \times ln(|edge|)$ swaps per sample, this is what *table 1* shows.

3. **Program Improvement**
   The program improvement tells a ratio of how much time consuming was decrease by the heuristic usage. It considers the $cooc_{time}$ of the cluster implementation that will be the same independent of the number of swaps, the time waste for the heuristic, the time used for swaping with the predicted value and total time for 10,000 samples each with $|edge| \times ln(|edge|)$. It is defined as:

$$I_{prog} = \left[1 - \frac{Heuristic_{time} + swap_{time}(predicted\ swap\ number) + cooc_{time}}{cooc_{time} + swap_{time}(|edge| \times ln(|edge|))}\right]$$

   where $cooc_{time}$ is the time wasted to perform the co-occurence calculation in the cluster implementation for all the movie pairs combination.

The heuristic was tested as for θ_m as for θ_s for different inputs, 1k,10k, 20k, 100k netflix datasets, and different values for the threshold α $(.01, .02$ and $.03)$. The cluster time consuming improvement by using the heuristic, both for θ_s and θ_m, is shown in *table 3* and *table 4*. The numbers generated by the heuristic, i.e the swap predicted number, when running the heuristic with θ_s are the same when running with θ_m, the only difference is the time the heuristic took to run, which is minimal difference.

$Dataset$	α	$H._{time}$	$Predicted\ number$	$Heuristic_{theo}$	$I_{swap_{real}}$	I_{prog}
$Netflix\ 1k\ users$	.01	63.38	584541	.38	.44	.04
$Netflix\ 1k\ users$	.02	33.49	133139	.88	.77	.07
$Netflix\ 1k\ users$	.03	33.65	133139	.88	.77	.07
$Netflix\ 10k\ users$	.01	2740.41	3016022	.35	.31	.09
$Netflix\ 10k\ users$	.02	542.08	1501075	.86	.82	.24
$Netflix\ 10k\ users$	.03	523.28	1501075	.86	.83	.24
$Netflix\ 20k\ users$	.01	5463.64	12408208	.30	.25	.088
$Netflix\ 20k\ users$	.02	1173.68	3124243	.85	.82	.30
$Netflix\ 20k\ users$	.03	1197.13	3124243	.85	.82	.30
$Netflix\ 100k\ users$	.01	41106.03	34276302	.32	.27	.12
$Netflix\ 100k\ users$	.02	7616.03	17387305	.85	.82	.38
$Netflix\ 100k\ users$	.03	7588.80	17387305	.85	.82	.38

Table 3: improvement in time consuming for θ_s

$Dataset$	α	$H._{time}$	$Predicted\ number$	$Heuristic_{theo}$	$I_{swap_{real}}$	I_{prog}
$Netflix\ 1k\ users$	.01	62.79	584541	.38	.44	.04
$Netflix\ 1k\ users$	.02	35.85	133139	.88	.75	.07
$Netflix\ 1k\ users$	.03	33.71	133139	.88	.76	.07
$Netflix\ 10k\ users$	.01	2520.51	3016022	.35	.32	.1
$Netflix\ 10k\ users$	.02	577.92	1501075	.86	.81	.24
$Netflix\ 10k\ users$	.03	596.30	1501075	.86	.81	.24
$Netflix\ 20k\ users$	.01	5498.24	12408208	.30	.25	.088
$Netflix\ 20k\ users$	.02	1173.68	3124243	.85	.82	.30
$Netflix\ 20k\ users$	.03	1197.13	3124243	.85	.82	.30
$Netflix\ 100k\ users$	.01	30906.04	34276302	.32	.42	.19
$Netflix\ 100k\ users$	.02	7487.22	17387305	.85	.82	.38
$Netflix\ 100k\ users$	.03	7515.84	17387305	.85	.82	.38

Table 4: improvement in time consuming for θ_m

The table 5 shows the times for co-occurence and swap for the cluster implementation after running the heuristic, i.e the cluster implementation with the predicted value. the metrics use this tables to get the $swap_{time}(predicted\ swap\ value)$ and $cooc_{time}$

	$\frac{SwapNumber}{Sample}$	$cooc_{time}$	$swap_{time}$	$Total_{time}$
$Netflix\ 1k\ users$	584541	2000.10	45.23	2049.30
$Netflix\ 1k\ users$	133139	1984.38	11.26	1999.08
$Netflix\ 10k\ users$	3016022	10602.07	595.39	11210.42
$Netflix\ 10k\ users$	1501075	11626.63	302.05	11943.41
$Netflix\ 20k\ users$	12408208	19683.46	2745.80	22450.83
$Netflix\ 20k\ users$	3124243	19972.95	690.23	20380.28
$Netflix\ 100k\ users$	34276302	80042.40	8991.31	89148.77
$Netflix\ 100k\ users$	17387305	80014.81	4459.50	84579.28

Table 5: Cluster implementation times after running the heuristic

7.2 Evaluation in Quality(PPV_k)

To measure the loss in quality when running the the heuristic with a swap less than $|edge| \times ln(|edge|)$, let's consider the error between the PPV_k for $|edge| \times ln(|edge|)$ swaps per sample and the PPV_k for less than this number:

$$\zeta(P) = \frac{|PPV_k(|edge| \times ln(|edge|)) - PPV_k(P)|}{PPV_k(|edge| \times ln(|edge|)}$$

Where, P is the predicted value found by the heuristic, $|PPV_k(|edge| \times ln(|edge|)$ is the PPV_k when running the cluster implementation without the heuristic and $|edge| \times ln(|edge|)$ swaps per sample, and $PPV_k(P)$ when running the cluster implementation with the *predicted swap value* found by the heuristic.
The tables 6 and 7 show the $|PPV_k(|edge| \times ln(|edge|)$ for the *Series* and *Movies Ground Truths*, respectively, with this values, we can calculate the error for the datasets and different ground truths. The results are shown in table 8, the highest error is 4% showing that the loss in quality is too small.

$Dataset$	$PPV_k(\|edge\| \times ln(\|edge\|))$
$1k\ users$	0.524711
$10k\ users$	0.688749
$20k\ users$	0.712934
$100k\ users$	0.726604

Table 6: $PPV_k(|edge| \times ln(|edge|))$ for the Series Ground Truth

$Dataset$	$PPV_k(\|edge\| \times ln(\|edge\|))$
$1k\ users$	0.216814
$10k\ users$	0.347345
$20k\ users$	0.362832
$100k\ users$	0.424779

Table 7: $PPV_k(|edge| \times ln(|edge|))$ for the Movies Ground Truth

$dataset$	$Predicted\ Values$	$\zeta(P)$ for the movies G. Truth	$\zeta(P)$ for the series G. Truth
$1k\ users$	584541	.010	.010
$1k\ users$	133139	.024	.040
$10k\ users$	3016022	.003	.006
$10k\ users$	1501075	.001	.015
$20k\ users$	12408208	.001	.012
$20k\ users$	3124243	.002	.003
$100k\ users$	34276302	.004	.002
$100k\ users$	17387305	.002	.010

Table 8: $\zeta(P)$ for the heuristic predicted values

8 Conclusion

The thesis propose is to show a heuristic to improve the cluster implementation. There were lots of efforts put in order to speed up the *link assessment problem* implementation, the cluster implementation shown in section 3 is the fastest software implementation so far done. However the PPP_k, for the datasets tested, shows it is not necessary to use $|edge| \times ln(|edge|)$ per sample in order to have a good quality output. These results motivated the study of the graph convergence and its relation with the PPP_k. From the necessity of measuring the graph convergence, there was defined θ_m and θ_s, both metrics identify the graph convergence by measuring the co-occurrence after sampling for movie pairs with the same degree. This convergence test, in turn, motivated the swap heuristic for finding a ideal swap number per sample.

To evaluate the heuristic results, there were established metrics both for PPV_k and reduction in time consuming. The first compares the swapping time for running the cluster implementation without any heuristic and when running the heuristic plus the cluster implementation, while the second calculates the error between the PPV_k with the predicted value found by the heuristic and PPV_k and $|edge| \times ln(|edge|)$.

The improvement in time consuming by using the heuristic was up to 82% in swap time and up to 38% considering the whole program, for the PPV_k the maximal error was 4%.

9 Appendix

9.1 Heuristic Code

The Heuristic implementation is composed of seven modules with different meanings, they are:

1. **Main**
   Responsible for managing all the other modules and control the forward and backward search.

2. **MPI Module**
   The Heuristic implementation works with the MPI and openMP application program interfaces(api), this structure manages the MPI api.

3. **Threads**
   This module controls the openMP treads and the program flow.

4. **Algorithm**
   Stands for the swap and co-occurrence algorithms.

5. **Heuristic**
   Perform all the calculation of $theta_m$ and $theta_s$ and select movie pairs with the same degree.

6. **Input Reader**
   Read the bipartite graph and store the information.

7. **Output**
   Creates a file with the $cooc_{FDSM}$, z-score and p-value for each movie pair.

The figure 20 shows the relationship between the modules. The main module is connected with all the other modules, it's responsible for modules initiation. Mpi module is connected with threads which in turn is connected with heuristic module.

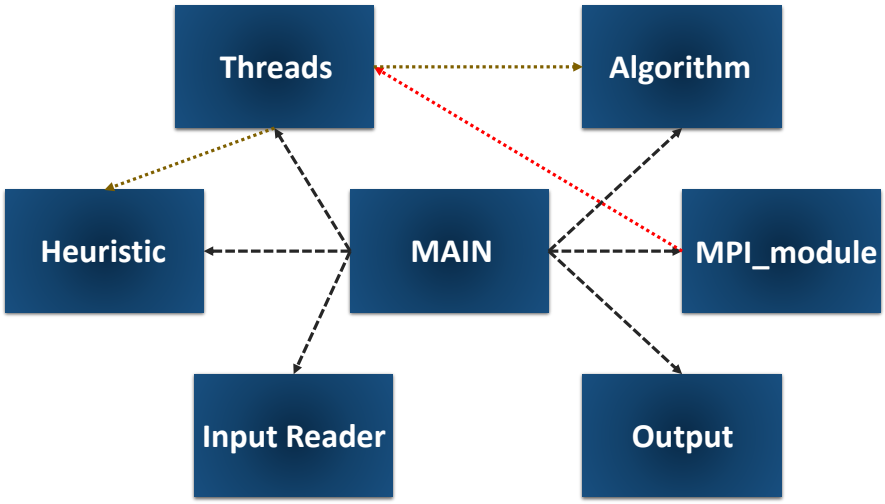

Figure 20: Heuristic Modules

9.1.1 Main Code

Listing 5: Main Code

```
#include "main.h"
double cumulatedSwaps;
void swapBackwardsRecSearch(int argc, char **argv,
    SWAP_HEURISTIC *h, MPIMODULE mpimodule,double error){
    if( h->sigmaM>error){
        cumulatedSwaps = cumulatedSwaps+h->limit2;
    }
    else{
        h= setSwap(h, h->limit1, h->currentSwap);
        cumulatedSwaps = cumulatedSwaps+h->currentSwap;
        mpiRun (argc, argv, mpimodule, h);
        swapBackwardsRecSearch(argc,argv,h,mpimodule, error);
    }
}

```

```
15 void swapForwardRecSearch(int argc, char **argv,
       SWAP_HEURISTIC *h, MPIMODULE mpimodule, double error){
16    if( h->sigmaM<error){
17       swapBackwardsRecSearch(argc,argv,h,mpimodule,error);
18       return h->currentSwap;
19    }
20    else{
21       h= setSwap(h, h->currentSwap, h->limit2);
22       cumulatedSwaps = cumulatedSwaps+h->currentSwap;
23       mpiRun (argc, argv, mpimodule, h);
24       swapForwardRecSearch(argc,argv,h,mpimodule,error);
25    }
26 }
27
28 int main (int argc, char **argv) {
29    ....
30    return 0;
31 }
```

9.1.2 Algorithm Code

Listing 6: Algorithm Code

```
1 #include "algorithm.h"
2 int cmpfunc (const void * a, const void * b){
3    return ( *(unsigned int*)a - *(unsigned int*)b );
4 }
5
6 void randomSwap (GRAPH* g, unsigned int* eventId, unsigned
      int* actorId, unsigned int* randNumber){
7    int numberOfBlocksPerActor = (g->numberOfEvents)/32 + 1;
8    int j, i;
9    unsigned int swap;
10   if ( g->adjMatrix [ actorId[0]*numberOfBlocksPerActor+
        eventId[1]/32 ] & (1 << eventId[1]%32) )//Check Non-
        Edge
```

```
11        return;
12    else
13        if ( g->adjMatrix [ actorId[1]*numberOfBlocksPerActor +
              eventId[0]/32 ] & (1 << eventId[0]%32) )//Check Non
             -Edge
14            return;
15        else {
16            g->adjMatrix [ actorId[0]*numberOfBlocksPerActor +
                 eventId[1]/32 ] |= (1 << eventId[1]%32);
                 //Set bit
17            g->adjMatrix [ actorId[1]*numberOfBlocksPerActor +
                 eventId[0]/32 ] |= (1 << eventId[0]%32);
                 //Set bit
18            g->adjMatrix [ actorId[0]*numberOfBlocksPerActor +
                 eventId[0]/32 ] &= (4294967295-(1 << eventId
                 [0]%32)); //Clear bit
19            g->adjMatrix [ actorId[1]*numberOfBlocksPerActor +
                 eventId[1]/32 ] &= (4294967295-(1 << eventId
                 [1]%32)); //Clear bit
20            swap = g->eventAdjList[ randNumber[1] ];//Swap edges
                  in adjacency list
21            g->eventAdjList[ randNumber[1] ] = g->eventAdjList[
                 randNumber[0] ];
22            g->eventAdjList[ randNumber[0] ] = swap;
23        }
24 }
25
26 void computeCoocJAVA(GRAPH* g, int eventId1, int eventId2,
       unsigned int** cooc) { // eventId2 > eventId1
27     int left_limit1  = g->eventAccumulatedDegrees[eventId1];
28     int right_limit1    = g->eventAccumulatedDegrees[eventId1
          +1];
29     int left_limit2  = g->eventAccumulatedDegrees[eventId2];
30     int right_limit2    = g->eventAccumulatedDegrees[eventId2
          +1];
31     int pos1 = left_limit1;
```

```
32      int pos2 = left_limit2;
33      while ( (pos1 < right_limit1) && (pos2 < right_limit2) )
           {
34          if (g->eventAdjList[pos1] < g->eventAdjList[pos2])
               pos1++;
35          else if (g->eventAdjList[pos1] > g->eventAdjList[pos2
               ]) pos2++;
36          else {
37         cooc[eventId1][eventId2-eventId1-1]++;
38              pos1++;
39              pos2++;
40          }
41      }
42  }
```

9.1.3 Threads Code

Listing 7: Algorithm Code

```
1  #include "threads.h"
2  void threadInit(OPENMP* openMP){
3     openMP->numberOfProcessors = omp_get_num_procs();
4     openMP->numberOfThreads = omp_get_num_threads();
5     openMP->threadId = omp_get_thread_num();
6  }
7
8  void threadCopyGraph (GRAPH* g){
9     ...
10 }
11
12 void threadRunCoocOriginal (GRAPH* g, unsigned int**
      originalCooc){
13    int i,j;
14    OPENMP openMP;
15    int eventChunk = (g->numberOfEvents-1) /16;
16    #pragma omp parallel shared(originalCooc, g) private(i, j)
```

```
17    {
18       #pragma omp for schedule(static,eventChunk)
19       for (i=0; i < g->numberOfEvents-1; i++){
20          for(j=i+1; j<g->numberOfEvents; j++){
21             computeCoocJAVA(g, i, j,originalCooc);
22          }
23       }
24    }
25 }
26
27 void swapParalell(GRAPH* g,OPENMP openMP,long seed,unsigned
       long numswaps){
28    unsigned int randNumber[2],i;
29    unsigned int eventId[2];
30    unsigned int actorId[2];
31    unsigned long swap;
32    gsl_rng *randGenerator[NUMTHREADS];
33    for (i=0; i<NUMTHREADS; i++) {
34       randGenerator[i] = gsl_rng_alloc (gsl_rng_mt19937);
35       gsl_rng_set(randGenerator[i], i + seed);
36    }
37    #pragma omp parallel shared( g ) private(openMP, swap,
          randNumber, eventId, actorId)
38    {
39             openMP.threadId = omp_get_thread_num();
40                for (swap=0; swap < numswaps; swap++) {
41                      randNumber[0]  = (unsigned int)
                          gsl_rng_uniform_int (randGenerator[
                          openMP.threadId], g[openMP.threadId].
                          numberOfEdges);
42                      randNumber[1]  = (unsigned int)
                          gsl_rng_uniform_int (randGenerator[
                          openMP.threadId], g[openMP.threadId].
                          numberOfEdges);
43                      actorId[0] = g[ openMP.threadId ].
                          eventAdjList[ randNumber[0] ];
```

```
44                      actorId[1] = g[ openMP.threadId ].
                            eventAdjList[ randNumber[1] ];
45                      eventId[0] = g[ openMP.threadId ].
                            eventEdgeMap[ randNumber[0] ];
46                      eventId[1] = g[ openMP.threadId ].
                            eventEdgeMap[ randNumber[1] ];
47                      randomSwap( &g[openMP.threadId], eventId
                            , actorId, randNumber);
48                  }
49      }
50  }
51
52  void Cooc_parallel(GRAPH *g, TMPRESULT* tmpResult, int
        graphNumber, SWAP_HEURISTIC *h){
53      unsigned int id1,id2,id3;
54      int eventChunk = (g->numberOfEvents-1) /16, i;
55      #pragma omp parallel shared(g, tmpResult) private(i)
56      {
57          #pragma omp for schedule(static,eventChunk)
58          for(i=0; i<g->numberOfEvents; i++)
59              qsort(&(g[graphNumber].eventAdjList[g->
                    eventAccumulatedDegrees[i]]), g->
                    eventAccumulatedDegrees[i+1] - g->
                    eventAccumulatedDegrees[i], sizeof(unsigned int),
                    cmpfunc);
60
61          #pragma omp barrier
62          #pragma omp for schedule(static,eventChunk)
63          for(i=0;i<48;i++){
64              id1= h->moviesIds[i][0];
65              id2= h->moviesIds[i][1];
66              id3= h->moviesIds[i][2];
67              if(id1>id2){
68                  computeCooc( &g[graphNumber], id2,id1,tmpResult->
                        lastCooc);
```

```
69              tmpResult->coocSum[id2][id1-id2-1]          +=
                    tmpResult->lastCooc[id2][id1-id2-1];
70              tmpResult->coocSquareSum[id2][id1-id2-1]    +=
                    tmpResult->lastCooc[id2][id1-id2-1]*tmpResult
                    ->lastCooc[id2][id1-id2-1];
71              tmpResult->lastCooc[id2][id1-id2-1]         = 0;
72          }
73          else{
74              computeCooc( &g[graphNumber], id1,id2,tmpResult->
                    lastCooc);
75              tmpResult->coocSum[id1][id2-id1-1]          +=
                    tmpResult->lastCooc[id1][id2-id1-1];
76              tmpResult->coocSquareSum[id1][id2-id1-1]    +=
                    tmpResult->lastCooc[id1][id2-id1-1]*tmpResult
                    ->lastCooc[id1][id2-id1-1];
77              tmpResult->lastCooc[id1][id2-id1-1]         = 0;
78          }
79
80          if(id1>id3){
81              computeCooc( &g[graphNumber], id3,id1,tmpResult->
                    lastCooc);
82              tmpResult->coocSum[id3][id1-id3-1]          +=
                    tmpResult->lastCooc[id3][id1-id3-1];
83              tmpResult->coocSquareSum[id3][id1-id3-1]    +=
                    tmpResult->lastCooc[id3][id1-id3-1]*tmpResult
                    ->lastCooc[id3][id1-id3-1];
84              tmpResult->lastCooc[id3][id1-id3-1]         = 0;
85          }
86          else{
87              computeCooc( &g[graphNumber], id1,id3,tmpResult->
                    lastCooc);
88              tmpResult->coocSum[id1][id3-id1-1]          +=
                    tmpResult->lastCooc[id1][id3-id1-1];
89              tmpResult->coocSquareSum[id1][id3-id1-1]    +=
                    tmpResult->lastCooc[id1][id3-id1-1]*tmpResult
                    ->lastCooc[id1][id3-id1-1];
```

```
90              tmpResult->lastCooc[id1][id3-id1-1]        = 0;
91          }
92       }
93    }
94 }
95
96 void threadRunSamples (GRAPH* g, ... , SWAP_HEURISTIC *h){
97    int graphNumber,row, col, i, j, sample;
98    int numsamples =((atoi(argv[3])+NUMTHREADS-1)/NUMTHREADS)*
         NUMTHREADS;
99    unsigned long swap,numswaps,burningSwaps = atoi(argv[5]);
100   unsigned long step = atol(argv[8]);
101   OPENMP openMP;
102
103   findMovieIds (h,g->eventAccumulatedDegrees,g->
         numberOfEvents,g->eventList);
104   swapParalell(g,openMP,seed, burningSwaps);
105
106   for (sample = 0; sample <numsamples;sample+=NUMTHREADS){
107      swapParalell(g,openMP,seed,step);
108      for (graphNumber=0; graphNumber<NUMTHREADS;graphNumber
            ++)
109         java_parallel(g, tmpResult, graphNumber,h);
110   }
111   stdevAndCoocCalc(h,sample,tmpResult->coocSum,tmpResult->
         coocSquareSum,step);
112 }
```

9.1.4 Heuristic Code

Listing 8: Heuristic Code

```
1
2 #include "../heuristic/heuristic.h"
3 int auxVector[25][3];
4 void sigmaCalc(SWAP_HEURISTIC *h){
```

```
5     int j=0,k=0,i=0;
6     double SumAux=0, SquareSumAux=0, meanAux=0, factor=4;
7     double cumulativeSTDstd=0,cumulativeSqrSTDstd=0;
8     double cumulativeSTDcooc=0,cumulativeSqrSTDcooc=0;
9
10    for(i=0;i<96;i=i+4){
11       for(j=i;j<i+4;j++){
12          SumAux=SumAux+h->coocMean[j];
13          SquareSumAux=SquareSumAux + h->coocMean[j]*h->
                coocMean[j];
14       }
15       meanAux  =SumAux/factor;
16       h->STD_coocMean[k]=stdevCalc(SquareSumAux, SumAux,
             factor);
17       cumulativeSTDcooc=cumulativeSTDcooc + h->STD_coocMean[k
             ]/meanAux;
18       cumulativeSqrSTDcooc=cumulativeSqrSTDcooc +(h->
             STD_coocMean[k]/meanAux)*(h->STD_coocMean[k]/meanAux
             );
19       SumAux=0;
20       SquareSumAux=0;
21       meanAux=0;
22       for(j=i;j<i+4;j++){
23          SumAux   = SumAux +h->stDeviation[j];
24          SquareSumAux   = SquareSumAux    +  h->stDeviation[j
                ]*h->stDeviation[j];
25       }
26       meanAux  =SumAux/factor;
27       h->STD_stdOfTheMean[k]=stdevCalc(SquareSumAux, SumAux,
             factor);
28       cumulativeSTDstd=cumulativeSTDstd + h->STD_stdOfTheMean
             [k]/meanAux;
29       cumulativeSqrSTDstd  =  cumulativeSqrSTDstd + (h->
             STD_stdOfTheMean[k]/meanAux)*(h->STD_stdOfTheMean[k
             ]/meanAux);
30       SumAux=0;
```

```
31        SquareSumAux=0;
32        meanAux=0;
33        k++;
34    }
35    h->sigmaM    =  (double)cumulativeSTDcooc/24.0;
36    h->sigmaS    =  (double)cumulativeSTDstd/24.0;
37    h->tauM      =  stdevCalc(cumulativeSqrSTDcooc,
          cumulativeSTDcooc,    (double)24.);
38    h->tauS      =  stdevCalc(cumulativeSqrSTDstd,
          cumulativeSTDstd,   (double)24.);
39 }
40
41 SWAP_HEURISTIC* setSwap(SWAP_HEURISTIC *h, unsigned long L1,
       unsigned long L2){
42    h->limit1    =  L1;
43    h->limit2    =  L2;
44    h->currentSwap =  (unsigned long) (sqrt(h->limit2*h->
          limit1));
45    return h;
46 }
47
48 SWAP_HEURISTIC* swapHeuristicInit(SWAP_HEURISTIC *h, unsigned
        long L1, unsigned long L2){
49    h=(SWAP_HEURISTIC*)malloc(sizeof(SWAP_HEURISTIC));
50    h->limit1    =  (unsigned long) (L1*log(L1));
51    h->limit2    =  (unsigned long) (L2*log(L2));
52    h->currentSwap =  (unsigned long) (sqrt(h->limit2*h->
          limit1));
53    return h;
54 }
55
56 double stdevCalc(double SquareSum, double Sum, double factor)
       {
57    return sqrt(( SquareSum - Sum*Sum/factor)/(factor-1.0));
58 }
59
```

```
60 void stdevAndCoocCalc(SWAP_HEURISTIC *h, int numsamples,
      unsigned int** coocSum,unsigned int** coocSquareSum,
      unsigned long numswaps){
61    int j=0,k=0, id1,id2,id3,i;
62    for(i=0;i<48;i++){
63       id1= h->moviesIds[i][0];
64       id2= h->moviesIds[i][1];
65       id3= h->moviesIds[i][2];
66       if(id1>id2){
67          h->coocMean[j]=      (double)coocSum[id2][id1-id2
               -1]/(double)numsamples;
68          h->stDeviation[j]=   stdevCalc((double)coocSquareSum
               [id2][id1-id2-1], (double)coocSum[id2][id1-id2
               -1], (double)numsamples);
69       }
70       else{
71          h->coocMean[j]=(double)coocSum[id1][id2-id1-1]/(
               double)numsamples;
72          h->stDeviation[j]=stdevCalc((double)coocSquareSum[
               id1][id2-id1-1], (double)coocSum[id1][id2-id1
               -1],(double)numsamples);
73       }
74       j++;
75       if(id1>id3){
76          h->coocMean[j]=   (double)coocSum[id3][id1-id3-1]/(
               double)numsamples;
77          h->stDeviation[j]=stdevCalc((double)coocSquareSum[
               id3][id1-id3-1], (double)coocSum[id3][id1-id3
               -1],(double)numsamples);
78       }
79       else{
80          h->coocMean[j]=(double)coocSum[id1][id3-id1-1]/(
               double)numsamples;
81          h->stDeviation[j]=stdevCalc((double)coocSquareSum[
               id1][id3-id1-1], (double)coocSum[id1][id3-id1-1],
                (double)numsamples);
```

```
82          }
83          j++;
84       }
85       sigmaCalc(h);
86    }
87
88    int degreehasBeenFound( int degree){
89       int i;
90       for(i=0;i<25;i++)
91          if(auxVector[i][0] == degree)
92             return 1;
93       return 0;
94    }
95
96    void findMovieIds (SWAP_HEURISTIC *h, unsigned int*
          eventAccumulatedDeg, int eventNumber, char** events){
97       int i,j,k=0, l=100/4;
98       unsigned int currentDegree, degAux1, degAux2;
99
100      for(i=0;i<25;i++){
101         auxVector[i][0]    = 0;
102         auxVector[i][1]    = 0;
103         auxVector[i][2]    = 0;
104      }
105         for(i=0;i<eventNumber;i++){
106            currentDegree=eventAccumulatedDeg[i+1]-
                  eventAccumulatedDeg[i];
107            if(currentDegree>50 && degreehasBeenFound(
                  currentDegree)==0 ){
108               for(j=i+1;j<eventNumber;j++){
109                  degAux1 = eventAccumulatedDeg[j+1]-
                        eventAccumulatedDeg[j];
110                  if(degAux1==currentDegree){
111                     auxVector[k][0]      = degAux1;
112                     auxVector[k][1]      = i;
113                     auxVector[k++][2] = j;
```

```
                    j = eventNumber+1;
                    if(k==25)
                        i=eventNumber+1;
                }
            }
        }
    }
    vectorCreate(h,eventAccumulatedDeg,events);
}

void vectorCreate(SWAP_HEURISTIC *h, unsigned int*
    eventAccumulatedDeg, char** events ){
    int i, j=0;

    for(i=0;i<25-1 ;i++){
        h->moviesIds[j][0]      =  auxVector[i][1];
        h->moviesIds[j][1]      =  auxVector[i+1][1];
        h->moviesIds[j++][2]    =  auxVector[i+1][2];
        h->moviesIds[j][0]      =  auxVector[i][2];
        h->moviesIds[j][1]      =  auxVector[i+1][1];
        h->moviesIds[j++][2]    =  auxVector[i+1][2];
    }
}
```

9.1.5 MPI Module Code

Listing 9: MPI Module Code

```
#include "mpiModule.h"
void mpiInit (int argc, char** argv, MPIMODULE* mpimodule){
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &mpimodule->procId);
        /* get current process id */
    MPI_Comm_size (MPI_COMM_WORLD, &mpimodule->numProcs); /*
        get number of processes */
}
```

```
7
8  unsigned int** movieStructureAloc(unsigned int** vector,
       unsigned int movies){
9     int row;
10    vector = (unsigned int**) malloc (( movies-1)*sizeof(
          unsigned int*));
11    for (row=0; row<movies-1; row++) {
12       vector[row] = (unsigned int*) malloc (( movies-1-row)*
             sizeof(unsigned int));
13    }
14    return vector;
15 }
16
17 void mpiRun(int argc, char** argv, MPIMODULE mpimodule,
       SWAP_HEURISTIC *h){
18    int row, col, i, j, k;
19    char inputFile[150];
20    OPENMP openMP;
21    threadInit (&openMP);
22    struct timespec start_read, end_read, seed;
23    double read_time=0;
24    GRAPH graph[NUMTHREADS];
25    clock_gettime(CLOCK_MONOTONIC, &start_read);
26    strcpy(inputFile, argv[1]);
27    strcat(inputFile,"/");
28    strcat(inputFile, argv[2]);
29    graph[0] = readInput(inputFile);
30    threadCopyGraph(graph);
31    unsigned int** originalCooc;
32    originalCooc = (unsigned int**) malloc ((graph[0].
          numberOfEvents-1)*sizeof(unsigned int*));
33    for (row=0; row<(graph[0].numberOfEvents-1); row++) {
34       originalCooc[row] = (unsigned int*) malloc ((graph[0].
             numberOfEvents-1-row)*sizeof(unsigned int));
35       if ( originalCooc[row] == NULL) {
36          exit(-1);
```

```
37        }
38    }
39
40    TMPRESULT tmpResult;
41    tmpResult.lastCooc = (unsigned int**) malloc ((graph[0].
          numberOfEvents-1)*sizeof(unsigned int*));
42    for (row=0; row<(graph[0].numberOfEvents-1); row++) {
43        tmpResult.lastCooc[row] = (unsigned int*) malloc ((
              graph[0].numberOfEvents-1-row)*sizeof(unsigned int))
              ;
44        if ( tmpResult.lastCooc[row] == NULL) {
45            exit(-1);
46        }
47    }
48
49    tmpResult.coocSum= (unsigned int**) malloc ((graph[0].
          numberOfEvents-1)*sizeof(unsigned int*));
50    for (row=0; row<(graph[0].numberOfEvents-1); row++) {
51        tmpResult.coocSum[row] = (unsigned int*) malloc ((graph
              [0].numberOfEvents-1-row)*sizeof(unsigned int));
52        if ( tmpResult.coocSum[row] == NULL ) {
53            exit(-1);
54        }
55    }
56
57    tmpResult.coocSquareSum = (unsigned int**) malloc ((graph
          [0].numberOfEvents-1)*sizeof(unsigned int*));
58    for (row=0; row<(graph[0].numberOfEvents-1); row++) {
59        tmpResult.coocSquareSum[row] = (unsigned int*) malloc
              ((graph[0].numberOfEvents-1-row)*sizeof(unsigned int
              ));
60        if ( tmpResult.coocSquareSum == NULL) {
61            exit(-1);
62        }
63    }
64    for (row=0; row<(graph[0].numberOfEvents-1); row++) {
```

```
65      for (col=0; col<(graph[0].numberOfEvents-1-row); col++)
            {
66          originalCooc[row][col]           = 0;
67          tmpResult.lastCooc[row][col]     = 0;
68          tmpResult.coocSum[row][col]        = 0;
69          tmpResult.coocSquareSum[row][col]   = 0;
70      }
71  }
72
73  threadRunCoocOriginal(&graph[0], originalCooc);
74  clock_gettime(CLOCK_MONOTONIC, &seed);
75  MPI_Bcast(&seed.tv_nsec, 1, MPI_LONG, 0, MPI_COMM_WORLD);
76  threadRunSamples(graph, &tmpResult, originalCooc, argc,
        argv, seed.tv_nsec + NUMTHREADS*mpimodule.procId,
        mpimodule.procId,h);
77  MPI_Barrier(MPI_COMM_WORLD);
78
79  if (mpimodule.numProcs != 1) {
80      if (mpimodule.procId == 0)
81          for (row=0; row<(graph[0].numberOfEvents-1); row++)
                {
82              MPI_Reduce(MPI_IN_PLACE, tmpResult.coocSum[row],
                    (graph[0].numberOfEvents-1-row), MPI_LONG,
                    MPI_SUM, 0, MPI_COMM_WORLD);
83              MPI_Reduce(MPI_IN_PLACE, tmpResult.coocSquareSum[
                    row], (graph[0].numberOfEvents-1-row),
                    MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
84          }
85      else
86          for (row=0; row<(graph[0].numberOfEvents-1); row++)
                {
87              MPI_Reduce(tmpResult.coocSum[row], tmpResult.
                    coocSum[row], (graph[0].numberOfEvents-1-row),
                     MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
88              MPI_Reduce(tmpResult.coocSquareSum[row],
                    tmpResult.coocSquareSum[row], (graph[0].
```

```
                numberOfEvents-1-row), MPI_LONG, MPI_SUM, 0,
                MPI_COMM_WORLD);
89          }
90      }
91
92      for(i=0;i<16;i++){
93          free(graph[i].eventAdjList);
94          free(graph[i].eventAccumulatedDegrees);
95          free(graph[i].eventEdgeMap);
96          free(graph[i].adjMatrix);
97
98          for (j=0; j<graph[0].numberOfEvents-1; j++)
99              free(graph[i].eventList[j]);
100
101         free(graph[i].eventList[j]);
102     }
103
104     for (j=0; j<graph[0].numberOfEvents-1; j++){
105         free(originalCooc[j]);
106         free(tmpResult.lastCooc[j]);
107         free(tmpResult.coocSum[j]);
108         free(tmpResult.coocSquareSum[j]);
109     }
110
111     free(originalCooc);
112     free(tmpResult.lastCooc);
113     free(tmpResult.coocSum);
114     free(tmpResult.coocSquareSum);
115
116     MPI_Barrier(MPI_COMM_WORLD);
117 }
118
119 void mpiFinalize (void)
120 {
121     MPI_Barrier(MPI_COMM_WORLD);
122     MPI_Finalize();
```

```
123 }
```

9.2 NetFlix Dataset PPV Plots

9.2.1 Movies Ground Truth

1k Users

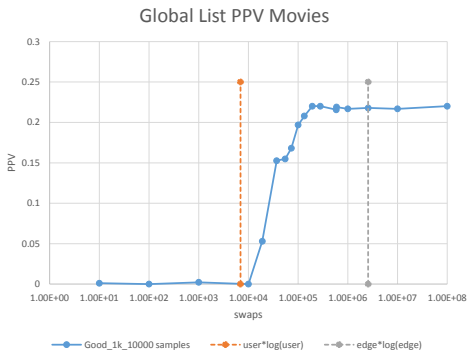

Figure 21: 1k Users Movies PPV

10k Users

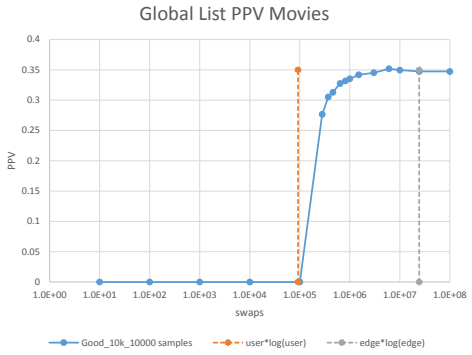

Figure 22: 10k Users Movies PPV

20k Users

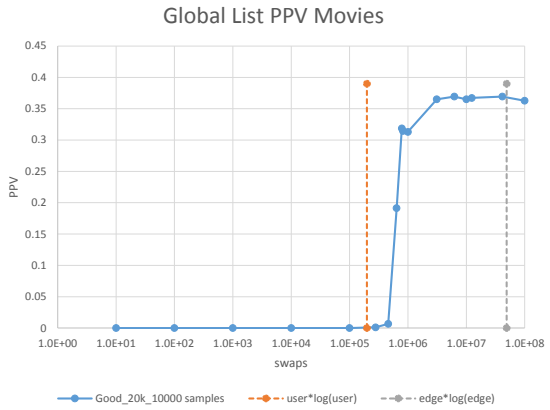

Figure 23: 20k Users Movies PPV

100k Users

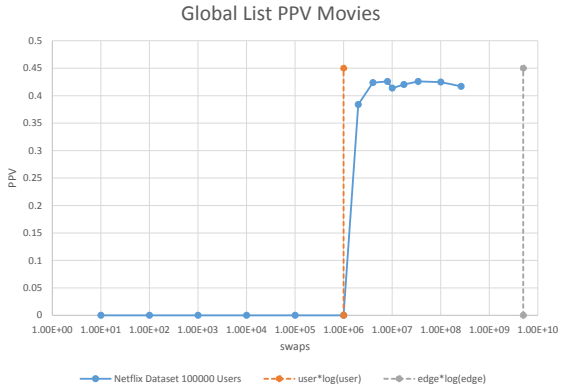

Figure 24: 100k Users Movies PPV

9.2.2 Series Ground Truth

1k Users

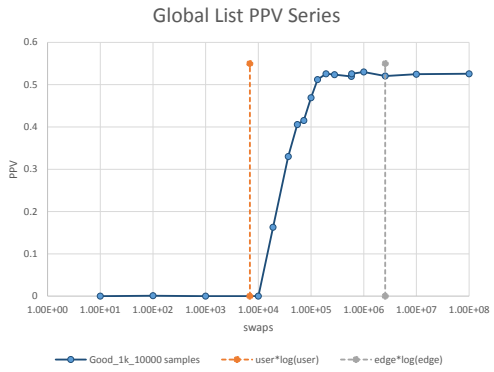

Figure 25: 1k Users Series PPV

10k Users

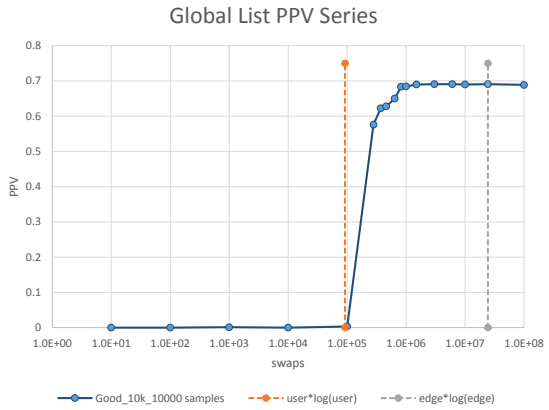

Figure 26: 10k Users Series PPV

20k Users

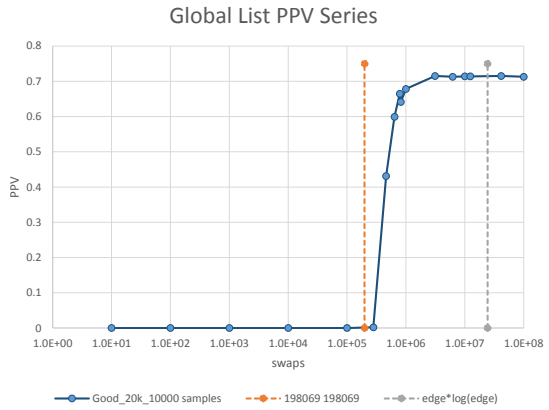

Figure 27: 20k Users Series PPV

References

[1] Katharina A. Zweig. How to forget the second side of the story - A new method for the one-mode projection of bipartite graphs. *Proceedings of the International Conference on Advances in Social Network Analysis and Mining, Odense, Denmark (ASONAM'10)(IEEE Computer Society)*, pages 200-207, 2010.

[2] Katharina A. Zweig, Michael Kaufmann. A systematic approach to the one-mode projection of bipartite graphs. *Social Network Analysis and Mining* Volume 1, Issue 3, pages 187-218

[3] Katharina A. Zweig, Emoke-Agnes Horvat. One-mode projections of multiplex bipartite graphs. *2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining.*

[4] Katharina A. Zweig, Emoke-Agnes Horvat, Andreas Spitz, Anna Gimmler, Thorsten Stoeck. The Link AssessMent Problem of Low Intensity Relationship in Complex Networks. *Submitted to Proceedings of the National Academy of Science of the United States of America.*

List of Figures

List of Tables