DEGREE FINAL PROJECT

Distributed graph signal processing

Degree Programme: Grau en Enginyeria de Sistemes Audiovisuais
Author: Eduard Sanou
Directors: Antonio Ortega and Philippe Salembier
Year: 2013
## Revision history and approval record

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01/09/2013</td>
<td>Document creation</td>
</tr>
<tr>
<td>1</td>
<td>28/09/2013</td>
<td>Document revision</td>
</tr>
<tr>
<td>2</td>
<td>13/11/2013</td>
<td>Supervisor document revision</td>
</tr>
<tr>
<td>3</td>
<td>25/11/2013</td>
<td>Final revision</td>
</tr>
</tbody>
</table>

## DOCUMENT DISTRIBUTION LIST

<table>
<thead>
<tr>
<th>Name</th>
<th>e-mail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eduard Sanou</td>
<td><a href="mailto:eduardsanou@gmail.com">eduardsanou@gmail.com</a></td>
</tr>
<tr>
<td>Antonio Ortega</td>
<td><a href="mailto:antonio.ortega@gmail.com">antonio.ortega@gmail.com</a></td>
</tr>
<tr>
<td>Philippe Salembier</td>
<td><a href="mailto:philippe.salembier@upc.edu">philippe.salembier@upc.edu</a></td>
</tr>
</tbody>
</table>
Index

List of figures .............................................................. 5
Collaborations ............................................................ 6
Appreciation .............................................................. 7
Resum del treball .......................................................... 8
Resumen del proyecto .................................................... 9
Abstract ........................................................................ 10
1. Introduction ................................................................ 11
   1.1 Report structure ..................................................... 11
   1.2 Graph Theory ......................................................... 11
   i) Graph Fourier Transform ........................................ 13
   ii) Recommendation systems ...................................... 14
   1.3 Motivation for distributed approach ......................... 15
2. GraphLab framework .................................................. 17
   2.1 Introduction .......................................................... 17
   2.2 GAS approach ....................................................... 17
   2.3 Execution engines .................................................. 19
   2.4 Graph Partitioning and distributed processing .......... 19
   2.5 Requisites for distributed operations ....................... 21
   2.6 Example: kNN (k Nearest Neighbors) ....................... 21
3. Filtering ...................................................................... 23
   3.1 Introduction .......................................................... 23
   3.2 Polynomial approximation ...................................... 23
   3.3 Degree 3 Polynomials approach .............................. 26
   3.4 Chebyshev approach .............................................. 26
   3.5 Complexity analysis .............................................. 27
   3.6 Performance and scalability .................................... 29
4. Conclusions ................................................................ 32
   4.1 Project results ....................................................... 32
Distributed graph signal processing

4.2 Future work ........................................................................................................ 32

5. Annexes .................................................................................................................... 34
  5.1 Updates .................................................................................................................. 34
  5.2 kNN source code ................................................................................................. 34
  5.3 Chebyshev source code ...................................................................................... 49
  5.4 Degree 3 Polynomials source code .................................................................... 55
  5.5 Graph tools source code ..................................................................................... 61

6. References ................................................................................................................ 63
List of figures

Figure 1. Visual representation of a graph consisting of 3 nodes with two edges. 12
Figure 2. Adjacency Matrix ........................................................................................................... 12
Figure 3. Signal Vector .................................................................................................................. 12
Figure 4. Degree Matrix .............................................................................................................. 12
Figure 5. Matrix factorization used in recommendation systems .................................................. 14
Figure 6. In this figure the scope of data that can be accessed during the gather phase can be seen. After
the data from each neighbor is processed, the result is summed in parallel ..................................... 18
Figure 7. The gathered data is applied in the central vertex ............................................................. 18
Figure 8. The nodes connected to the central vertex are singled for future updates ......................... 19
Figure 9. GAS operations in a split part of the graph. In this case the graph was cut on the Y node,
creating a replica on each machine. Node Y has 8 neighbor nodes, 4 of them can be found
in the machine on the left while the others can be found on the machine on the right.
Each machine computes its own sum from the gather step. The results are summed on
both machines to obtain a common value to apply ......................................................................... 20
Figure 10. Visual representation of a part of the graph for the GAS operations ............................ 22
Figure 11. Filter function in frequency domain ............................................................................. 23
Figure 12. Frequency values of a graph signal .............................................................................. 23
Figure 13. Performance when varying the number of coefficients .................................................. 30
Figure 14. Performance when varying the percentage of connections .......................................... 30
Figure 15. Performance when varying the number of nodes .......................................................... 31
Collaborations
Appreciation

I would like to thank all the people that made this work possible. Antonio Ortega for giving me advice every time I needed, for helping me organizing all the work, for orienting me through all the concepts and for introducing me in the research field. Sunil K. Narang for being patient with me and for explaining me all the details of the different graph signal processing algorithms in the recommendation systems problem. I would like to thank Philippe Salembier for being the advisor for my project. Finally I also want to thank my family and my friends for giving me support when I needed it.
Resum del treball

Les representacions de dades mitjançant grafs suposa una tendència en investigació actualment, especialment per aquells problemes que tracten amb grans quantitats d’informació que necessiten ser analitzades i estudiades. Molts camps es poden beneficiar de l’ús de grafs. Recentment, el processat de senyal s’ha començat a aplicar en grafs, essent encara un concepte bastant nou que requereix ser explorat en més detall. Varies aplicacions del processat de senyal en grafs s’estan estudiant actualment, com per exemple resoldre el problema de Sistemes de Recomanació.

Aquest projecte extén la investigació feta en el camp del processat de senyal aplicat a grafs, però tenint en compte la tendència actual d’usar grafs grans que provenen de bases de dades massives. Per aquest cas, s’han desenvolupat frameworks que ens permeten operar amb els grafs en un entorn distribuït. GraphLab s’usarà en aquest projecte ja que és un framework lider en la computació distribuïda d’alt rendiment basat en grafs que s’ha usat ja en investigació per estudiar i aplicar algorismes en grans grafs.

En aquest projecte, una eina de processat de senyal específica ha estat estudiada i desenvolupada: la operació de filtrat. Dos implementacions diferents de tècniques de filtrat han estat desenvolupades, tenint en compte les necessitats de disseny per a una computació distribuïda. El rendiment i escalabilitat del filtrat distribuït per a grafs s’ha analitzat per poder observar com es comporta el framework amb diferents grafs, variant la mida i la connectivitat entre nodes.

El programa de filtrat proposat en aquest projecte ofereix una forma efectiva per a computar en grafs permetent múltiples aplicacions.

PARAULES CLAU: Teoria de Grafs, GraphLab, processat de senyal, filtrat, computació distribuïda
Resumen del proyecto

Las representaciones de datos mediante grafos suponen una tendencia en investigación actualmente, especialmente para aquellos problemas que tratan con grandes cantidades de información que necesitan ser analizadas y estudiadas. Muchos campos se pueden beneficiar del uso de grafos. Recientemente, el procesado de señal se ha empezado a aplicar en grafos, siendo aún un concepto bastante nuevo que requiere ser explorado en más detalle. Varias aplicaciones del procesado de señal en grafos se están estudiando actualmente, como por ejemplo resolver el problema de Sistema de Recomendaciones.

Este proyecto extiende la investigación hecha en el campo de procesado de señal aplicado en grafos, pero teniendo en cuenta la tendencia actual de usar grafos grandes provenientes de bases de datos masivas. Para este caso, se han desarrollado frameworks que nos permiten operar con grafos en un entorno distribuido. GraphLab se usará en este proyecto ya que es un framework líder en computación distribuida de alto rendimiento basado en grafos que se ha usado ya en investigación para estudiar y aplicar algoritmos en grandes grafos.

En este proyecto, una herramienta de procesado de señal específica ha sido estudiada y desarrollada: la operación de filtrado. Dos implementaciones diferentes de técnicas de filtrado han sido desarrolladas, teniendo en cuenta las necesidades de diseño para una computación distribuida. El redimiendo y escalabilidad del filtrado distribuido para grafos se ha analizado para poder observar como se comporta el framework con diferentes grafos, variando el tamaño y la conectividad entre nodos.

El programa de filtrado propuesto en este proyecto ofrece una forma efectiva para computar en grafos permitiendo múltiples aplicaciones.

PALABRAS CLAVE: Teoría de Grafos, GraphLab, procesado de señal, filtrado, computación distribuida
Abstract

Graph representation of data is a current trend in research nowadays, especially for those problems dealing with huge amounts of information that need to be analyzed and studied. Many fields can benefit from the use of graphs. Just recently, signal processing has begun to be applied to them, still being a fairly new concept that needs to be explored in more depth. Various applications of the signal processing applied on graphs are currently studied, such as solving the Recommendation System problem.

This project extends the work and research done in the field of signal processing applied to graph, but taking into account the current trend of using large graphs from massive datasets. For this case, frameworks have been developed that allow us to operate with graphs in a distributed environment. GraphLab will be used in this project as it is a leading high-performance distributed computation graph-based framework that has already been used in research to study and apply algorithms on large graphs.

In this project, a specific signal processing tool commonly used is studied and developed: the filtering operation. Two different implementations of a filtering technique are made, taking into account the needs for a distributed computation design. The performance and scalability of the distributed graph filter will be analyzed in order to observe how the framework behaves with different graphs, varying the size and the scarcity of them.

The filtering program proposed in this project delivers an efficient way to compute on a graph allowing multiple applications.

KEYWORDS: Graph Theory, GraphLab, signal processing, filtering, distributed computing
1. Introduction

In this chapter, the report structure will be discussed. After that, a brief introduction to Graph Theory will be made. Getting deeper into the subject of this work, the idea of signal processing on graphs will be introduced with specific details on frequency analysis for graphs. Along these lines a particular case will be mentioned: the Recommendation systems problem. Finally, taking into account current trends and needs, the motivation for a distributed approach will be discussed.

1.1 Report structure

This report is structured in 4 different chapters. The first one, the introduction, will give a brief explanation of the Graph Theory and the Signal Processing applied on Graphs, giving special attention to the Fourier Transform applied to signals on graphs. The motivation for this project will also be discussed. Chapter 2 will be about the GraphLab framework, which is used for the experiments in this project, with an explanation on how it works and other important information along with a practical example of the implementation of an algorithm. Chapter 3 deals with the main work of this project: a filtering algorithm that works in a distributed manner for graphs. For this, the needs for such an algorithm are discussed and finally, two approaches for its implementation are proposed. An analysis and performance test of the filtering algorithm will be made. The last chapter will provide conclusions with a discussion of the experimental results and possible future work on the topic.

As additional information, the annex contains the source code for the different algorithms developed in this project along with the tools used to perform the different experiments.

1.2 Graph Theory

Graph Theory consists on the study of graphs, which are mathematical structures which model relations between elements. A graph is made up of nodes (or vertices) and edges (or links) that connect them. A graph can be undirected, if for each pair of connected nodes there is no origin node and destiny node (the edge connects the nodes bidirectionally); or it can be directed: edges go from one node to another node. In this work, undirected graphs will be used. It is worth noticing that these kinds of graph have special algebraic properties that make it easy to work with them.1

Graphs can be used to model many kinds of relations between physical, social or information systems. Typical examples of graph usages are: to model physical devices in a network and their connectivity, to study and model molecules in chemistry, to model web pages and their connections through hyperlinks (used in the original Google algorithm: PageRank).
Signal Processing on Graphs

For signal processing purposes, we can attach values to the different elements of the graph. In this work, we will use a positive real value for each edge: edge weight, and a real value for each node, representing the signal on the graph. Such signal is going to be always discrete due to the nodal nature of the graph, but it can be taken from a continuous signal that needs to be modeled into a graph. For this case, a meaning of the weight links will need to be found.

A graph can be expressed as a matrix defining the weights between nodes (Adjacency Matrix) and a vector defining the values of the nodes (Signal Vector). This will allow us to write matrix expressions and operations to analyze and do algebraic calculations with the graph.

From the Adjacency Matrix, we can define the Degree Matrix, which shows the sum of weight for all edges connected to a certain node (we define this value as the degree of a node). For convenience, these values are stored in the diagonal of a matrix.

In the following picture, a visual representation of a 3 node graph can be seen, with some random values on its nodes and some random values for its edges.

![Figure 1. Visual representation of a graph consisting of 3 nodes with two edges.](image)

The previous graph can be expressed with matrices and vectors, as stated above:

\[
W = \begin{bmatrix}
0 & 0.3 & 0 \\
0.3 & 0 & 0.7 \\
0 & 0.7 & 0
\end{bmatrix}
\]

\[
f = \begin{bmatrix}
1.2 \\
0.3 \\
0.4
\end{bmatrix}
\]

**Figure 2.** Adjacency Matrix  \hspace{1cm}  **Figure 3.** Signal Vector

\[
D = \begin{bmatrix}
0.3 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 0.7
\end{bmatrix}
\]

**Figure 4.** Degree Matrix
Once the graph is defined in terms of matrices and vectors, a very interesting matrix can be defined: the Laplacian matrix, $L$, defined in 1.1. This matrix will be very useful for spectral analysis of the graph, as we will see in the Fourier Transform section.

From the Laplacian matrix, we can obtain the normalized Laplacian matrix, $\mathcal{L}$, using the degree matrix $D$, as shown in 1.2. Using the Normalized Laplacian matrix will make the frequency analysis easier by normalizing the frequency range from 0 to 2.

\begin{align*}
L &= D - W \quad (1.1) \\
\mathcal{L} &= D^{\frac{1}{2}}L D^{-\frac{1}{2}} \quad (1.2)
\end{align*}

For any undirected graph, the Laplacian matrix is symmetric and positive definite. This means that when applying a Singular Value Decomposition to the matrix as shown in 1.3, the eigenvectors will be orthogonal and the eigenvalues will be real and non negative.

\begin{equation}
\mathcal{L} = U \Lambda U^T \quad (1.3)
\end{equation}

i) Graph Fourier Transform

Using the SVD (Singular Value Decomposition) of the Normalized Laplacian matrix we can define a transform similar to the Fourier Transform:

- The eigenvectors will represent the frequency bases of the signal.
- The eigenvalues will represent the normalized frequency of each base.

Projecting the signal onto the eigenvectors gives us the signal in the frequency domain. This transform can be inverted in order to get the signal back in the graph domain as follows:

\begin{equation}
\hat{f} = U^* f \quad f = U \hat{f} \quad (1.4)
\end{equation}

With the Graph Fourier Transform we just defined, spectral analysis of the graph signal can be performed. This transform allows the modification of the energy found in each frequency for a given signal, leading to the concept of filtering. Many applications of signal filtering, such as interpolation, can now be applied to graphs. It is worth noticing that in order to apply the Fourier Transform, the SVD needs to be computed, which is an expensive
operation. Since the SVD is computed from the Laplacian matrix, which is different for every topology, each graph will have different frequency bases.

Other signal processing operations have been studied and adapted to work on graphs, like the convolution, modulation and dilation among others. These operations will not be discussed since they are not used in this project, but papers with extensive analysis on them can be found in the references.\(^2\)\(^3\)

### ii) Recommendation systems

In this project we focus on the problem of designing a Recommendation Systems. This problem consists on finding an algorithm that is able to predict the rating a user would give to an item (song, book, movie) using as background information the user characteristics (known ratings from the user) and items characteristics (other user’s ratings).

To recommend new items to the user, the unrated elements are predicted using the given algorithm and from all of them, the ones with higher predicted rating are recommended to the user. If the algorithm can predict the ratings accurately, the recommended elements will have a great chance of being liked by the user.

To solve this problem, a popular state of the art technique is the matrix factorization, with methods such as Non Negative Matrix Factorization seen for example in the Probabilistic Factorization Matrix algorithm.\(^4\)

In the matrix factorization approach, a matrix is built with all the ratings, where the rows represent the users and the columns represent the items. Since each user will have only rated a small subset of all available items, the corresponding matrix will be sparse, with a rank much lower than its size (zeros are used for unrated items). This matrix can be factorized into two smaller matrices, one representing the users’ characteristics and the other representing the movies characteristics. The algorithm used for the factorization of the matrix will try to minimize the error of the multiplication of the two small matrices to the rating matrix, but only taking into account the known ratings. With this approach, values will be obtained for the unknown ratings when multiplying the two small matrices. These values will be used as the ratings prediction.\(^5\)

![Figure 5. Matrix factorization used in recommendation systems.](image-url)
A different way to solve this problem can be developed by analyzing it from a Graph point of view. In this case, a graph can be built where each node represents an item (where the ratings from a given user will be the values of each node). The edge weights will represent the similarity between items, and can be computed using the ratings from users who have rated both items, for each pair of items. The following formula can be used to compute the weights (W_{ab} indicates the weight value from item “a” to item “b”. \( R_{ai} \) is the rating the user \( i \) gives to the item “a”, whereas \( R_{bi} \) is the rating the user \( i \) gives to the item “b”. \( n \) will be the number of users that have rated both items):

\[
W_{ab} = \frac{\sum_{i=1}^{n} R_{ai} \times R_{bi}}{\sqrt{\sum_{i=1}^{n} (R_{ai})^2} \times \sqrt{\sum_{i=1}^{n} (R_{bi})^2}} \tag{1.5}
\]

With this information, the rating prediction problem can be solved as an interpolation for all the unrated items. For this solution to work, the partial signal known in the graph (obtained from the rated items) should be a smooth signal. If this happens to be true, a low pass filter could be applied on the graph that would allow us to recover the unknown ratings. It will be important to design the filter with the right cut-off frequency in order to obtain accurate results in the interpolation. This specific problem will not be solved in this project although a filtering system will be designed that could be used for this case.

### 1.3 Motivation for distributed approach

Signal processing on graphs has been applied in practice to small graphs, but only a limited amount of work has been performed with large graphs, for which scalability issues could appear.

Operations on graphs can be expressed as matrix operations, which may not be suitable for distributed computation: to allow an algorithm to work in parallel, the computation needs to be broken into small independent parts. Certain matrix operations could not allow this. Other approaches may be needed to express signal processing operations on graphs in order to facilitate parallel processing.

The Fourier Transform appears in many signal processing algorithms. For graphs, we have seen that the normalized Laplacian matrix is used to obtain the signal in the frequency domain, by using the Graph Fourier Transform. Using the Laplacian matrix, \( \mathcal{L} \), in a distributed way needs to be studied.

These questions and several others will be studied and answered in this project.
Distributed graph signal processing

There is a current trend on working with large graphs; this makes it interesting to operate with them in a distributed way. Many massive datasets exist already and they are too big to be loaded on a single computer. Furthermore, just one computer would not have enough resources to process such graph.

Many examples can be found of these large datasets (They are not graphs per se, but can be expressed as such for data analysis and signal processing purposes):

- Netflix: >100,000 titles, 36.3 million subscribers
- Facebook: 1.1 billion active users per month
- Twitter: 140 million twits per day, 460,000 new users per month
- Amazon: 1.7 million eBooks

To deal with such big graphs, several frameworks have been developed. They allow parallel processing (taking advantage of the multi-core architecture of modern CPUs) and also distributed computing (the workload and memory requirements of the processing is spread over several machines in a network). When implementing an algorithm in a distributed framework the operations will need to be as local as possible. This will mean that for some current algorithms, local approximations will be needed. In this project, we use the GraphLab Framework, which will be introduced in the next chapter.
2. GraphLab framework

2.1 Introduction

GraphLab is a graph-based, high performance, distributed computation framework written in C++. The framework has been designed to allow a program to be completely scalable. The same written code can be used to run on a single machine as well as on a group of machines in a shared network in a distributed manner.

Programs using the GraphLab framework need to take into account that it has been designed to work with some considerations regarding the problem to be solved as it will be seen. In order to take advantage of the framework optimizations, the graphs used on the algorithms developed need to be sparse and have local dependencies: the processing is applied locally. Another consideration is that GraphLab has been designed to work with iterative algorithms. These algorithms could have a potentially asynchronous execution model in order to make the most of the framework.

The framework abstracts the user from the parallelism concerns. This allows us to focus on the algorithm that needs to be applied on the graph, without worrying about scalability or concurrency.

2.2 GAS approach

In order to apply the algorithms and make them compatible with a distributed framework, GraphLab uses the GAS approach for the implementation of the operations that work on a graph. GAS stands for Gather, Apply and Scatter. An algorithm needs to be defined using these steps in order to work with GraphLab. Algorithms in GraphLab are expressed as vertex programs, which are executed in parallel on each vertex and can interact with the neighboring vertices.

GAS Steps

In the first step, gather, at each node we can send a request to neighbors (other nodes at a one hop distance) to do some computation with their data (node data) and the data associated with the edge joining the neighbor node and the central node. A representation of this process applied on one node can be seen in Figure 6: Node Y is the central node, while the scope represents the neighboring area. Each neighbor node will obtain a result from the computation.
Figure 6. In this figure the scope of data that can be accessed during the gather phase can be seen. After the data from each neighbor is processed, the result is summed in parallel.

The results of the gather step are summed in parallel (as seen in Figure 6) and collected into one result in the apply step. Once the gathered data is accumulated in the central vertex, it will be available for computation. The data and values of the central node can be updated with the results, as it can be seen in Figure 7.

Figure 7. The gathered data is applied in the central vertex.

After the apply step, we can decide to signal the neighboring vertices so that they update their values, this is the scatter step. This decision can be made analyzing the data stored in the central node. In Figure 8 an example is shown where all the neighbor nodes are signaled.

These three steps can be understood better in practice with the example in section 2.6, in the implementation of the k nearest neighbors’ algorithm.
2.3 Execution engines

When GraphLab has loaded a vertex-program, vertices need to be signaled in order to execute the program on them. For this purpose, any given number of vertices in the graph can be signaled. In some cases, all the nodes will be signaled; in other cases, the graph may be split into training nodes and testing nodes, and only the testing nodes will be signaled. Once this has been done, they are added to an execution queue, and depending on the engine used, they can be processed in two different ways:

- Synchronized engine: In this mode, for each iteration of the vertex program, the engine will wait for the processing of all the vertices. After the processing operations on all vertices have finished, the next iteration will start. This will ensure the consistency of the values in the neighboring vertices. This mode is useful for algorithms that work with multiple steps, where for each step the algorithm will use results of the previous one.

- Asynchronous engine: In this mode, the iterations of the program will be processed on each vertex without waiting for the others to finish. In this case, the algorithm may finish earlier in a specific part of the graph. This mode is useful for convergence algorithms such as PageRank, where the values of each vertex are updated using the neighboring vertices values until they reach a stable state.

2.4 Graph Partitioning and distributed processing

Whereas graphs are usually cut on the edges, in order to work with distributed machines, GraphLab cuts the graph based on the vertices, creating replicas of the same split vertex in both sides of the cut, so that a copy of the node exists in all the machines where that particular vertex is needed.
For any split vertex, the gather step is going to happen on all the machines where that vertex has replicas, using the neighbors stored in each machine. Each machine will have a partial result from the collection of the neighbors they have access to. This partial result will be shared on the network so that each machine can sum the partial results and get a common value, which will be used on each machine during the apply step, as seen in Figure 9.

\[
(F_1 + F_2)(Y) \rightarrow Y
\]

**Figure 9.** GAS operations in a split part of the graph. In this case the graph was cut on the Y node, creating a replica on each machine. Node Y has 8 neighbor nodes, 4 of them can be found in the machine on the left while the others can be found on the machine on the right. Each machine computes its own sum from the gather step.

The results are summed on both machines to obtain a common value to apply.

For the distribution of the graph over the machines in the network, a partitioning will be made that minimizes the required network communication between machines. For this purpose GraphLab constructs a balanced partitioning of the data graph that minimizes the number of edges that cross over machines.

A two phased partitioning is applied in order to effectively balance the load on arbitrary cluster sizes. In the first phase, the graph is over-partitioned into k parts where k is much greater than the number of machines, using domain specific knowledge like planar embedding. Each part of the graph is stored in a temporary distributed storage system. A meta-graph is build with k vertices (corresponding to initial partitioning) and is then distributed over the physical machines by performing a fast balanced partitioning. Each machine constructs its own part of the graph using the partition of the meta-graph assigned and the information of the initial partitioning.

This two step partitioning allows reusing part of the computation when using a different number of machines, without requiring a full partitioning step. The performance obtained with this method is similar to a direct partitioning, allowing a faster performance.\(^{10}\)

More detailed information on the algorithms used for the different partitioning steps can be found in the OSIDI 2012 paper, which describes GraphLab 2.1, in section 4.\(^{11}\)
2.5 Requisites for distributed operations

After describing the workflow that GraphLab uses to implement algorithms, we can observe some restrictions and requirements needed for the algorithms we want to develop:

- Operations will need to be local. As it has been seen with the GAS model, for each vertex only the neighbor vertices can be accessed. This means that one hop operations need to be used.
- Inability to use the Laplacian matrix of the whole graph for each vertex. To obtain the Laplacian matrix we need access to the entire graph. GraphLab does not allow this. If the algorithm that needs to be implemented using the Laplacian matrix (a common thing with frequency related algorithms) we will need to find a solution. A local Laplacian matrix could be used: the values of neighbor vertices could be used if the operation allows so. This restriction gets more severe in case of wanting a Graph Fourier Transformation. For this, the SVD of the Laplacian matrix is needed, and it cannot be obtained with a local Laplacian matrix: the Fourier bases will not be the same. A one hop graph could be used for the Graph Fourier Transformation although this will lead to different results since the Laplacian matrix will be different.
- For certain algorithms that need access to the entire graph, approximations will need to be found taking into account only a small part of the graph.

Overall, it can be seen that vertex domain methods should be used.

2.6 Example: kNN (k Nearest Neighbors)

A simple algorithm will be discussed here to help understand how to implement an operation using the GAS approach. The chosen algorithm is fairly easy to understand in GAS terms.

In terms of Signal Processing on Graphs, kNN is a simple algorithm for interpolation. The value of each node can be approximated by the average of the neighbor values, using the edge weights to give more importance to certain nodes (2.1). The k value can be used to define a fixed number of neighbors to use for the average; the k neighbors with the highest edge weights will be used.

\[
\tilde{f} = \frac{\sum_{i=1}^{K} w_i f_i}{\sum_{i=1}^{K} w_i}
\]  
(2.1)
The kNN algorithm can be split using the GAS approach in the following way:

- **Gather**: For each connected node, multiply the edge weight with the value of the neighbor node.
- **Apply**: After getting the sum of the gathered values, it needs to be normalized by the sum of the weights of the edges connected to the central node (the degree of the interpolated node). The normalized result is stored in the central node as the interpolated value.
- **Scatter**: In this case, the algorithm only needs one iteration. No signaling of other nodes is needed here; this step can be skipped.

The source code of the kNN algorithm implemented using GraphLab can be found in the Annex section 5.1.
3. Filtering

3.1 Introduction

A very common operation in signal processing is the filtering. After learning about the Fourier Transform in Graph Theory, frequency analysis can be applied. This allows the design of filters in the frequency domain.

Currently, there exist many uses of filtering operations such as interpolation, signal smoothing, edge detection and wavelets among others. Various applications of filtering for common signals can also be applied to signals on graphs with interesting results. This is why it seems interesting to develop a framework to apply filters on graphs using the GraphLab framework and thus taking advantage of its parallel and distributed properties that allow it to work with large graphs.

A spectral filter for a graph can be defined as a function of lambda (where lambda is the discrete frequency for a given graph, and also corresponds to the eigenvalues obtained when doing the SVD of the Normalized Laplacian matrix). This can be seen in Figure 11.

In frequency domain, obtaining the filtered signal implies multiplying each frequency value of the signal (as seen in Figure 12) with the filter function at each given frequency (lambda). It can be seen that even though the filter function can be continuous, when we apply it, we use discrete values, since a signal on a graph is discrete due to its nature.

![Figure 11. Filter function in frequency domain](image1)

![Figure 12. Frequency values of a graph signal](image2)

3.2 Polynomial approximation

A given function describing a filter can be approximated using polynomials. Using the polynomial approximation gives us a great advantage: the filter can be applied in Graph Domain. There will be no need to use the Fourier Transform and this will allow a much faster implementation since no SVD (Singular Value
Decomposition) will need to be computed. The SVD is an expensive algorithm, with complexity \( O(N^3) \) where \( N \times N \) is the size of the matrix.

We can see how the polynomial approximation can be applied in Graph Domain in the following equations. In equation 3.1 it can be seen how the filter function is expressed as a polynomial in frequency domain. Equation 3.2 shows how this filter would be multiplied with the signal in frequency domain in order to obtain the result.

\[
h(\Lambda) = \sum_{i=0}^{N} \alpha_i \lambda^i \tag{3.1}
\]

\[
\hat{y} = h(\Lambda) \hat{f} \tag{3.2}
\]

In order to obtain the filtered signal in graph domain, first the signal needs to be projected on the eigenvectors. By doing this we obtain the signal in frequency domain, where the filter can be applied. Finally, the signal is projected back, getting the result in the graph domain. This can be seen in equation 3.3. This allows us to express the filter \( H \) that will be used in the graph domain.

Since the filter is independent of the eigenvectors, the eigenvectors can be brought into \( h(\lambda) \). This leads to expressing \( h \) as a function of the Normalized Laplacian matrix, like we see in equation 3.4.

After seeing this result, it can be seen that a polynomial filter in graph domain is expressed as polynomial function of the Laplacian matrix (equation 3.5). On equation 3.6 the factorization of the polynomial can be seen, leading to the expression using pairs of coefficients.

\[
y =Uh(\Lambda)U^T f = Hf \tag{3.3}
\]

\[
H = Uh(\Lambda)U^T = h(U\Lambda U^T) = h(L) \tag{3.4}
\]

\[
H = \sum_{i=0}^{N} \alpha_i L^i \tag{3.5}
\]

\[
H = \prod_{i=0}^{N} \left( a_i I + b L \right) \tag{3.6}
\]
From the final expression of the filter (equation 3.6), we can observe that not only it can be applied in the Graph Domain but it can be seen as an iterative local operation if we factorize the polynomial. This will fit with the distributed approach needs, and we will be able to implement it in the GraphLab Framework.

In equation 3.7 we see the operation that will need to be implemented, with $H$ expressed as a factorized polynomial in equation 3.8. For each iteration, each degree 2 polynomial will be multiplied with the signal, as it can be seen in equation 3.9, showing an example with a 3 node graph. The solution for the first value of the vector is shown in equation 3.10.

$$\mathbf{y} = \mathbf{Hf} \quad (3.7) \quad H = \prod_{i=0}^{N} (a_i I + b_i \mathbf{L}) \quad (3.8)$$

$$\begin{align*}
\mathbf{\tilde{V}}_1 &= \begin{bmatrix} a_0 + b_0 \\ -b_0 \frac{W_{12}}{\sqrt{d_1 d_2}} \\ -b_0 \frac{W_{13}}{\sqrt{d_1 d_3}} \end{bmatrix} \mathbf{V}_1 \\
\mathbf{\tilde{V}}_2 &= \begin{bmatrix} -b_0 \frac{W_{21}}{\sqrt{d_2 d_1}} \\ a_0 + b_0 \\ a_0 + b_0 \end{bmatrix} \mathbf{V}_2 \\
\mathbf{\tilde{V}}_3 &= \begin{bmatrix} -b_0 \frac{W_{31}}{\sqrt{d_3 d_1}} \\ -b_0 \frac{W_{32}}{\sqrt{d_3 d_2}} \\ a_0 + b_0 \end{bmatrix} \mathbf{V}_3
\end{align*} \quad (3.9)$$

$$\mathbf{\tilde{V}}_1 = (a_0 + b_0) \mathbf{V}_1 - b_0 \left( \frac{W_{12}}{\sqrt{d_1 d_2}} \mathbf{V}_1 - \frac{W_{13}}{\sqrt{d_1 d_3}} \mathbf{V}_3 \right) \quad (3.10)$$

It can be seen in the last equation (3.10) that to obtain the result for a node, only the neighbor nodes are needed: in case the node is not a neighbor, the weight will be 0 and thus its value and degree won’t make an impact on the result. This is a favorable result as it will allow an implementation of the algorithm in GraphLab.

One issue exists when factorizing the polynomials. The coefficients can be imaginary numbers. This could lead to inaccurate results and also could increase the number of operations needed. To solve this, two approaches can be made:

- Express the polynomial function as a multiplication of smaller polynomials of degree 3. This will allow us to express the filtering operation as an iterative two hops operation.

- Use Chebyshev filters. For any polynomial function, a Chebyshev factorization exists that uses real numbers as coefficients.
3.3 Degree 3 Polynomials approach

Factorizing the polynomial function into smaller degree 3 polynomials will lead to iterative operations that are not completely local (equations 3.11). But for each degree 3 polynomial, we can separate the operation into two different steps, where each step is local (one hop operation).

\[
y = \prod_{i=0}^{N'} (a_i' I + b_i' \mathcal{L}) f
\]

(3.11)

\[
y = \prod_{i=0}^{N} (a_i I + b_i \mathcal{L} + c_i \mathcal{L}^2) f
\]

(3.12)

\[
(a_i I + b_i \mathcal{L} + c_i \mathcal{L}^2) f = a_i f + b_i (\mathcal{L} f) + c_i (\mathcal{L} f)
\]

It has been seen in the previous section that when the Laplacian is multiplied by the signal values, the result can be obtained from each node using local operations.

It can be seen in the decomposition of each degree 3 polynomial that the operation can be broken into two steps (equation 3.12). The first will obtain the result of the first two elements and the partial result of the third one (Laplacian multiplied by f). On the second step, the partial result of the third element will be multiplied by \( c_i \) and by the Laplacian. Also during the second step, all the elements will be summed to obtain the result of the iteration.

The source code of the degree 3 polynomial filter implementation can be found in the annex in section 5.2.

3.4 Chebyshev approach

Chebyshev filters can be designed to be equivalent to any polynomial filter. The obtained Chebyshev filter will be of the same degree as the polynomial filter.\(^\text{12}\)

To find the Chebyshev coefficients from a polynomial filter fast methods exist, with complexity \( O(N^2) \) where \( N \) is the degree of the polynomial. Usually the degree of the polynomial will not be high enough to be a performance issue in here.

Chebyshev filters have the property that all the coefficients will be real and within a range value.
To apply the Chebyshev filters, a recursive iterative implementation exists that uses local operations and has the same complexity as applying a factorized polynomial filter.

The algorithm to apply the filter can be divided into two steps:

- **Initialization:** Several recursive values are initialized, and the first two coefficients are used.
- **Recursion:** For each coefficient starting from the third one, recursive iterations are performed that update the recursive values until a final result is obtained.

\[
\begin{align*}
T_0 &= f \\
T_1 &= \frac{\mathcal{L} - a_2 f}{a_1} \\
r_0 &= \frac{1}{2} c_0 T_0 + c_1 T_1 \\
T_{n+1} &= \frac{2}{a_n} (\mathcal{L} T_n - a_2 T_{n-1}) - T_{n-1} \\
r_n &= r_{n-1} + c_n + 1 T_{n+1}
\end{align*}
\]

The Chebyshev filter algorithm can be applied in GraphLab using the GAS approach: the operations dealing with the Normalized Laplacian can be applied locally (the Normalized Laplacian matrix multiplied with \( T_n \) is applied the same way as if it was multiplied by \( f \), like in the previous approach).

The source code of the Chebyshev filter implementation can be found in the annex in section 5.3.

### 3.5 Complexity analysis

Since both approaches can be split into iterative local operations, they can be implemented in GraphLab using the GAS approach.

The complexity of the Chebyshev algorithm will be analyzed here. The analysis will be done for the iterative recursion computations, since the initialization of the values corresponds to a small fraction of the computation that will not have much effect in the performance result. For the Degree 3 polynomials approach, the complexity would be similar, that is why it will not be studied.

Let’s consider that \( n_1 \) is the number of Chebyshev coefficients and \( n_2 \) is the number of nodes.
Gather step

In this step, the operation needed is the one that involves multiplying the Normalized Laplacian matrix with $T_n$, as it is the only operation that needs neighbor values. This operation can be seen in equation 3.14. The result of the first element in the vector can be seen in equation 3.15.

$$
\begin{bmatrix}
1 & -\frac{W_{ij}}{\sqrt{d_id_j}} & -\frac{W_{ij}}{\sqrt{d_id_j}} \\
-\frac{W_{ij}}{\sqrt{d_id_j}} & 1 & -\frac{W_{ij}}{\sqrt{d_id_j}} \\
-\frac{W_{ij}}{\sqrt{d_id_j}} & -\frac{W_{ij}}{\sqrt{d_id_j}} & 1
\end{bmatrix}
\begin{bmatrix}
T_1^n \\
T_2^n \\
T_3^n
\end{bmatrix}
$$

(3.14)

$$
\text{imp } 1 = T^2 n - \left( \frac{W_{12}}{\sqrt{d_id_j}} T^2 n + \frac{W_{13}}{\sqrt{d_id_j}} T^3 n \right)
$$

(3.15)

The operation that will be computed for each neighbor of each node is the following:

$$
\frac{W_{ij}}{\sqrt{d_id_j}} T^n
$$

The complexity of this computation $O(4)$ as it consists of 4 operations. This operation will be applied on each neighbor node of each graph node.

The number of edges in the graph can be expressed using the constant $c$, which represents the percentage of possible connections that exist between nodes. For a graph with $n_2$ nodes, there could be a maximum of $n_2^2$ connections. The total number of connections is divided by the number of nodes to make an average. The result is multiplied by 2 because the connections are bidirectional in this case:

$$
\frac{2 cn_2}{n_2} = 2 cn_2
$$

We can express the complexity of the gather operation on all the nodes in one iteration as: $O \left( 4 \times 2 cn_2 \times n_2 \right)$

The results of each gather operation will need to be summed for the apply step: $O \left( 2 cn_2 \times n_2 \right)$

Apply step

This part involves the following operations:

$$
T_{u+1} = \frac{2}{a_{u+1}} \left( c_{a_{u+1}} - a_{u+1} T_u \right) - T_{u-1}
$$

$$
r_u = r_{u-1} + c_{a_{u+1}} + T_{u+1}
$$

These operations will be applied on each node of the graph. The complexity will be: $O \left( 8 n_2 \right)$

Considering that these steps will be made for every coefficient of the Chebyshev filter, it can be seen that the complexity of the algorithm is:

$$
O \left( 10 cn_2 + 8 n_2 \right) n_1 = O \left( \left( 10 cn_2 + 8 n_2 \right) n_1 \right)
$$
From this expression, it can be seen that the algorithm scales linearly with the number of Chebyshev coefficients and exponentially (power of two) with the number of nodes, maintaining the same level of scarcity (connectivity).

It is worth mentioning that added to the obtained complexity expression, a function that depends on $n_1$ and $n_2$ should be taken into account, representing the operations needed by GraphLab to send messages over the machines and to synchronize the GAS processes. As it will be seen in the experimental results, the expression obtained models the performance and scalability properly, meaning that this added complexity does not affect the complexity analysis much.

3.6 Performance and scalability

Different performance tests have been run in order to analyze the time needed to apply the algorithms. For this purpose, different tools have been written to ease the creation of random graphs using parameters such as the number of nodes and the connectivity. The source code of these tools can be found in the annex, in section 5.4.

The tests have been run on a single machine with the following characteristics: 8 cores CPU and 53GB or RAM memory.

The multi core CPU will allow GraphLab to make use of the parallelization optimizations. It is also interesting to notice that having a machine with a high amount of RAM will allow us to load and process big graphs that would need more than one machine otherwise.

As an initial run, a graph with 50K nodes has been used, with 10% connectivity (125M edges). This connectivity represents an extreme case, rarely found in big graphs.

Applying a 64 coefficient Chebyshev filter on this graph has taken the following times:

- Time to load graph: 979.29 seconds
- Time to initialize algorithm: 11.31 seconds
- Time to finish all the iterations: 542.13 seconds

In these experiments, the graph is stored into two separate text files: one containing the values for each node (graph signal), and the other containing the weights between nodes when there is a connection (graph topology). Graphs could be stored in more efficient forms. It’s also worth mentioning that depending on the storage read
speed, the time to load the graph could be faster. This timing will not be taken into account when performing scalability tests.

For the processing part of the algorithm, it can be seen that the time to initialize the algorithm is much lower than the time to apply the iterations. Due to this, on the scalability tests only the iterations time will be used.

To analyze the scalability of the algorithm with different parameters, a script has been written to test different values within a range. The following results have been obtained:

![Figure 13. Performance when varying the number of coefficients](image13.png)

![Figure 14. Performance when varying the percentage of connections](image14.png)
We can observe that both the number of coefficients and the connectivity factor make the algorithm scale linearly (Figures 13 and 14). The results do not show a straight line but it is close enough to be considered linear scaling. In the case of the number of nodes, we see an exponential growth (Figure 15). These results coincide with the complexity analysis done in the previous section, so we can conclude that the analysis made was correct.
4. Conclusions

4.1 Project results

While giving an introduction of the Graph Theory and the emerging field of signal processing on graphs, this project has focused mainly on one specific application of signal processing applied on graphs in a distributed manner.

A filtering program has been created that allows us to work with large graphs in a distributed and efficient manner using the GraphLab framework, allowing multiple applications.

The scalability and performance analysis and tests of the filtering program show positive results when working with large graphs, since the computation needs of the algorithm scale properly with minimum performance issues due to the parallel nature of the implementation. Linear scalability is always desired for many algorithms, and in this case we were able to achieve regarding the number of edges and the number of coefficients for the filter. When incrementing the size of the graph, the growth of the complexity was exponential; but this happens because the number of connections in the graph increase exponentially when the number of nodes increases, if the scarcity is maintained. With this application, large graphs can be processed that could not be loaded in other environments such as Matlab. For graphs that happen to be too big, we are not limited to the resources of a single machine: more computers can be added in the network in order to share the processing costs.

4.2 Future work

Regarding to signal processing on graphs, there are still many things to study as it is a fairly new concept. Multiple applications can be found for this topic, as well as new problems requiring graph processing. More interestingly, these signal processing algorithms have not been tested much on large graphs; it would be interesting to see what new results can be obtained from this.

In the lines of this project, one first approach that could be taken would be to design specific Chebyshev filters for signal processing problems such as interpolation. In this case it would be worth comparing the results with other interpolation algorithms for graph signals. With an interpolation filter implemented using Chebyshev filters, the Recommendation System problem could be solved and compared with other Recommendation System algorithms. For that, the designing of the filter should be deeply studied since for each user a different filter may be required in order to get an accurate interpolation of the items ratings.

Many applications of filtering exist for regular signals. Filtering applied to a large Graph is a new concept that should be studied to find possible uses. For instances, it would be interesting to apply wavelets filters on large graphs and study the possible outcomes of the results.
As it has been seen in the introduction, filtering is just one example of signal processing. Many other algorithms exist in the signal processing field that could be used on Graphs. It would be very interesting to analyze these algorithms and find ways to implement them using the GAS approach. Currently, various toolkits exist in the GraphLab framework to deal with specific topics. A signal processing toolkit does not exist yet, so it could be developed with several algorithms. This toolkit could be incorporated into the GraphLab project so that anyone could use it.
5. Annexes

5.1 Updates

The name of the project has been changed from “Graph-based recommendation system” to “Distributed graph signal processing”.

The aim of the project has changed from implementing and analyzing a specific algorithm for recommendation systems to analyze the signal processing applied to graphs signals in a distributed manner, in a more broad way. All the results of the study of the recommendation systems are reflected in the current work.

5.2 kNN source code

The kNN program has been tested with movie ratings datasets. In order to apply the kNN algorithm, weights on the edges are needed (which are not found in the original dataset). For this purpose, the first two parts of the program have been created; they read the original database and create a graph using the weights described in section 1.3 ii. The kNN algorithm is implemented in the third part: knn3.cpp

```cpp
/**
 * File knn.cpp
 *
 * \brief The first step for KNN rating prediction
 *
 * This file contains the first step for KNN rating prediction. It reads the
 * input user ratings and creates outputs with: movies containing each user
 * rating in one line per movie (one file for training and one for validation)
 * and one file containing the movie connections (two movies are connected if at
 * least one same user has rated them both).
 *
 * #include <string>
 #include <graphlab.hpp>
 #include <boost/unordered_map.hpp>
 #include <list>
 #include <limits>

 unsigned int uimax = std::numeric_limits<int>::max();

 using namespace graphlab;

 const int SAFE_NEG_OFFSET = 2; // add 2 to negative node id
 // to prevent -0 and -1 which are not allowed

typedef boost::unordered_map<vertex_id_type, double> map;

/**
 * \brief The vertex data stores the movie rating information.
 */
 struct vertex_data {
```
/** brief The ratings each user has given to the movie */
map ratings;
map ratings_test;
bool is_movie;

vertex_data(bool is_movie = false): is_movie(is_movie) { }

/** brief Save the vertex data to a binary archive */
void save(graphlab::oarchive& arc) const {
    arc << ratings;
}
/** brief Load the vertex data from a binary archive */
void load(graphlab::iarchive& arc) {
    arc >> ratings;
}; // end of vertex data

/**
 * brief The edge data stores the weights between movies.
 */
struct edge_data : public graphlab::IS_POD_TYPE {
/**
 * brief The type of data on the edge;
 *
 * li *Train:* the observed value is correct and used in training
 * li *Validate:* the observed value is correct but not used in training
 */
    enum data_role_type { TRAIN, VALIDATE);

    /** brief The observed value for the edge */
    double obs;

    /** brief The train/validation/test designation of the edge */
    data_role_type role;

    /** brief basic initialization */
    edge_data(double obs = 0, data_role_type role = TRAIN) :
        obs(obs), role(role) { }
}; // end of edge data

typedef graphlab::distributed_graph<vertex_data, edge_data> graph_type;
typedef graph_type::vertex_type vertex_type;
typedef graph_type::edge_type edge_type;

/**
 * brief The graph loader function is a line parser used for
distributed graph construction. We load the data as user to movie edges,
where the edge values are the ratings from user to movie.
*/
bool graph_loader(graph_type& graph,
    const std::string& filename,
    const std::string& line) {

    // Determine the role of the data
    edge_data::data_role_type role = edge_data::TRAIN;
    if(boost::ends_with(filename,“.validate”))
        role = edge_data::VALIDATE;
    else if(boost::ends_with(filename, “.train”))
        role = edge_data::TRAIN;

    // Parse the line
    std::stringstream strm(line);
    graph_type::vertex_id_type source_id(-1), target_id(-1);
    double obs(0);
    strm >> source_id >> target_id >> obs;
Distributed graph signal processing

// map target id into a separate number space
//source_id = -(graphlab::vertex_id_type(source_id + SAFE_NEG_OFFSET));
//!!! This could be dangerous, will uimax be the same in different platforms???
source_id = uimax - source_id;

// Create an edge and add it to the graph
graph.add_vertex(source_id);
graph.add_vertex(target_id, vertex_data(true));
graph.add_edge(source_id, target_id, edge_data(obs, role));

return true; // successful load
} // end of graph_loader

class gather_type {
public:

map ratings; //Accumulate the user ratings
map ratings_test; //Accumulate the user ratings for testing purposes

/** \ brief basic default constructor */
gather_type() {
}
gather_type(vertex_id_type user, double rating) {
    ratings[user] = rating;
}
gather_type(vertex_id_type user, double rating, bool test) {
    if (test)
        ratings_test[user] = rating;
    else
        ratings[user] = rating;
} // end of constructor for gather type
gather_type(map rat) {
    //ratings.insert(rat.begin(), rat.end());
    ratings = rat;
}
/** \ brief Save the values to a binary archive */
void save(graphlab::oarchive& arc) const { arc << ratings; }
/** \ brief Read the values from a binary archive */
void load(graphlab::iarchive& arc) { arc >> ratings; }
/**
* \b brief joins two maps
*/
gather_type& operator+=(const gather_type& other) {
    ratings.insert(other.ratings.begin(), other.ratings.end());
    ratings_test.insert(other.ratings_test.begin(), other.ratings_test.end());
    return *this;
} // end of operator+=
}; // end of gather type
/**
* \b brief The first step saves all the user ratings in two different maps
* inside the vertex representing the movies.
*/
class vertex_program :
public graphlab::ivertex_program<graph_type, gather_type>,
public graphlab::IS_POD_TYPE { public:
/** The set of edges to gather along */
edge_dir_type gather_edges(icontext_type& context,
    const vertex_type& vertex) const {
    return graphlab::IN_EDGES;
}; // end of gather_edges

/** The gather function */
gather_type gather(icontext_type& context, const vertex_type& vertex,
    edge_type& edge) const {
if (edge.data().role == edge_data::TRAIN) {
    //printf("%i ", edge.source().id());
    return gather_type(edge.source().id(), edge.data().obs, false);
} else {
    return gather_type(edge.source().id(), edge.data().obs, true);
}
} // end of gather function

void apply(icontext_type& context, vertex_type& vertex,
    const gather_type& sum) {
    //for (map::iterator it; it != sum.ratings.end(); ++it)
    // graph.add_edge(vertex.id(), it->first);
    vertex.data().ratings = sum.ratings;
    vertex.data().ratings_test = sum.ratings_test;
} // end of apply

// No scatter needed. Return NO_EDGES
edge_dir_type scatter_edges(icontext_type& context,
    const vertex_type& vertex) const {
    return graphlab::NO_EDGES;
}

/**
 * \brief Signal all vertices on one side of the bipartite graph
 */
static graphlab::empty signal_right(icontext_type& context,
    const vertex_type& vertex) {
    if(vertex.num_out_edges() == 0)
        context.signal(vertex);
    return graphlab::empty();
} // end of signal_right

/**
 * \brief The second step saves a map in every user vertex containing all the
 * movies ID which that user has rated. (This will be used in the next step
 * in order to find the movie connections)
 */
class vertex2_program : public graphlab::ivertex_program<graph_type, gather_type>,
    public graphlab::IS_POD_TYPE {
public:
    /** The set of edges to gather along */
    edge_dir_type gather_edges(icontext_type& context,
        const vertex_type& vertex) const {
        return graphlab::OUT_EDGES;
    }; // end of gather_edges

    /** The gather function */
gather_type gather(icontext_type& context, const vertex_type& vertex,
        edge_type& edge) const {
        return gather_type(edge.target().id(), edge.target().id());
    } // end of gather function
void apply(icontext_type& context, vertex_type& vertex, 
    const gather_type& sum) {
  //for (map::iterator it; it != sum.ratings.end(); ++it)
  // graph.add_edge(vertex.id(), it->first);
  vertex.data().ratings = sum.ratings;
} // end of apply

// No scatter needed. Return NO_EDGES
edge_dir_type scatter_edges(icontext_type& context, 
    const vertex_type& vertex) const {
  return graphlab::NO_EDGES;
}

/**
 * \brief Signal all vertices on one side of the bipartite graph
 */
static graphlab::empty signal_left(icontext_type& context, 
    const vertex_type& vertex) {
  if(vertex.num_in_edges() == 0)
    context.signal(vertex);
  return graphlab::empty();
} // end of signal_left

/**
 * \brief The third and last step joins the maps from the user vertex which
 * contained the movies the user rated, into the vertex movie. This way, we have
 * now in each movie vertex a map containing all the movies ID it should be
 * connected to
 */
class vertex3_program : public graphlab::ivertex_program<
graph_type, gather_type>, public graphlab::IS_POD_TYPE {
public:

  /** The set of edges to gather along */
  edge_dir_type gather_edges(icontext_type& context, 
      const vertex_type& vertex) const {
    return graphlab::IN_EDGES;
  } // end of gather_edges

  /** The gather function */
  gather_type gather(icontext_type& context, const vertex_type& vertex, 
      edge_type& edge) const {
    return gather_type(edge.source().data().ratings);
  } // end of gather function

  void apply(icontext_type& context, vertex_type& vertex, 
      const gather_type& sum) {
    //for (map::iterator it; it != sum.ratings.end(); ++it)
    // graph.add_edge(vertex.id(), it->first);
    vertex.data().ratings = sum.ratings;
  } // end of apply

  // No scatter needed. Return NO_EDGES
  edge_dir_type scatter_edges(icontext_type& context, 
      const vertex_type& vertex) const {
    return graphlab::NO_EDGES;
  }

  /**
   * \brief Signal all vertices on one side of the bipartite graph
   */
  static graphlab::empty signal_right(icontext_type& context, 
      const vertex_type& vertex) {
if (vertex.num_out_edges() == 0) {
    context.signal(vertex);
    return graphlab::empty();
} // end of right_left
}; // end of als vertex program

/**
 * \brief Saves the users rating for each training user
 */
struct graph_writer {
    std::string save_vertex(graph_type::vertex_type vt) {
        std::stringstream strm;
        if (vt.num_out_edges() == 0) {
            strm << vt.id() << " ";
            for (map::iterator it = vt.data().ratings.begin(); it != vt.data().ratings.end(); ++it)
                strm << it->first << " " << it->second << " ";
            strm << "\n";
        }
        return strm.str();
    }
    std::string save_edge(graph_type::edge_type e) { return ""; }
}; // end of pagerank writer

/**
 * \brief Saves the users rating for each test user
 */
struct graph_test_writer {
    std::string save_vertex(graph_type::vertex_type vt) {
        std::stringstream strm;
        std::list<vertex_id_type> li;
        if (vt.num_out_edges() == 0) {
            for (map::iterator it = vt.data().ratings_test.begin(); it != vt.data().ratings_test.end(); ++it) {
                if (it->first != vt.id())
                    li.push_back(it->first);
            }
            li.sort();
            li.unique();
            for (std::list<vertex_id_type>::const_iterator it = li.begin(); it != li.end(); ++it) {
                strm << *it << " ";
            }
            strm << "\n";
        }
        return strm.str();
    }
    std::string save_edge(graph_type::edge_type e) { return ""; }
}; // end of pagerank writer
int main(int argc, char** argv) {
    graphlab::mpi_tools::init(argc, argv);
    graphlab::distributed_control dc;

    dc.cout() << "Loading graph.\n" << std::endl;
    graphlab::timer timer;
    graph_type graph(dc);
    graph.load("movielens/", graph_loader);
    dc.cout() << "Loading graph. Finished in "
               << timer.current_time() << std::endl;

    dc.cout() << "Finalizing graph.\n" << std::endl;
    timer.start();
    graph.finalize();
    dc.cout() << "Finalizing graph. Finished in "
              << timer.current_time() << std::endl;

    dc.cout() << "========== Graph statistics on proc " << dc.procid() << " ===============\n"        << "\nNum vertices: " << graph.num_vertices() << "\nNum edges: " << graph.num_edges() << "\nNum replica: " << graph.num_replicas() 
              << "Replica to vertex ratio: "
              << float(graph.num_replicas()) / graph.num_vertices() 
              << "------------------------\n"        << "Num local own vertices: " << graph.num_local_own_vertices() 
              << "Num local vertices: " << graph.num_local_vertices() 
              << "Replica to own ratio: " 
              << float(graph.num_local_vertices()) / graph.num_local_own_vertices() 
              << "Num local edges: " << graph.num_local_edges() 
              // << "\n\nBegin edge id: " << graph.global_eid(0) 
              << "\nEdge balance ratio: " 
              << float(graph.num_local_edges()) / graph.num_edges() << std::endl;

    dc.cout() << "Creating engine\n" << std::endl;
    graphlab::omni_engine<vertex_program> engine(dc, graph, "sync");

    // Signal all vertices on the vertices on the left (liberals)
    engine.map_reduce_vertices<graphlab::empty>(vertex_program::signal_right);

    // Run 1st engine
    dc.cout() << "Running ...\n" << std::endl;
    timer.start();
    engine.start();

    const double runtime = timer.current_time();
    dc.cout() << "-------------------------------------------\n"        << "Final Runtime (seconds): " << runtime 
              << "updates executed: " << engine.num_updates() << std::endl 
              << "Update Rate (updates/second): " 
              << engine.num_updates() / runtime << std::endl;

    // Save the final graph -------------------------------------------
    graph.save("out_rat", graph_writer(), 
                false, // do not gzip 
                true,  // save vertices 
                false); // do not save edges

    graph.save("out_test_rat", graph_test_writer(), 
                false, // do not gzip 
                true,  // save vertices
// do not save edges

dc.cout() << "Creating engine 2" << std::endl;
graphlab::omni_engine<vertex2_program> engine2(dc, graph, "sync");

// Signal all vertices on the vertices on the left (liberals)
engine2.map_reduce_vertices<graphlab::empty>(vertex2_program::signal_left);

// Run 2nd engine
dc.cout() << "Running ..." << std::endl;
timer.start();
engine2.start();

const double runtime2 = timer.current_time();
dc.cout() << "---------------------------------------------" << std::endl
  << "Final Runtime (seconds): " << runtime2
  << std::endl
  << "updates executed: " << engine2.num_updates() << std::endl
  << "Update Rate (updates/second): "
  << engine2.num_updates() / runtime2 << std::endl;

dc.cout() << "Creating engine 3" << std::endl;
graphlab::omni_engine<vertex3_program> engine3(dc, graph, "sync");

// Signal all vertices on the vertices on the left (liberals)
engine3.map_reduce_vertices<graphlab::empty>(vertex3_program::signal_right);

// Run 3rd engine
dc.cout() << "Running ..." << std::endl;
timer.start();
engine3.start();

const double runtime3 = timer.current_time();
dc.cout() << "---------------------------------------------" << std::endl
  << "Final Runtime (seconds): " << runtime3
  << std::endl
  << "updates executed: " << engine3.num_updates() << std::endl
  << "Update Rate (updates/second): "
  << engine3.num_updates() / runtime3 << std::endl;

graph.save("out_edg", graph_edge_writer(),
false, // do not gzip
true, // save vertices
false); // do not save edges

graphlab::mpi_tools::finalize();
return EXIT_SUCCESS;
}

knn2.cpp: The code corresponding to the algorithm implementation of the weight computation has been highlighted.

/*
 * file knn2.cpp
 *
 * brief The second step for KNN rating prediction
 *
 * This file contains the second step for KNN rating prediction. It reads the
 * input created by the first step and loads it as graph where each vertex
 * is a movie. It then calculates the weight of the edges between movies using
 * the vector cosine similarity. The result will be saved in a file.
 * This will output the weighted graph, which can be used for other algorithms.
 */
#include <string>
#include <list>
#include <graphlab.hpp>
#include <boost/unordered_map.hpp>
#include <math.h>

using namespace graphlab;

const int SAFE_NEG_OFFSET = 2; //add 2 to negative node id
//to prevent -0 and -1 which are not allowed

typedef boost::unordered_map<vertex_id_type, double> map;

/**
 * brief The vertex data stores the movie rating information.
 */
struct vertex_data {

/**
 * brief The ratings each user has given to the movie */
 map ratings;

vertex_data() { }
vertex_data(map ratings): ratings(ratings) { }

/**
 * brief Save the vertex data to a binary archive */
 void save(graphlab::oarchive& arc) const {
 arc << ratings;
 }
/**
 * brief Load the vertex data from a binary archive */
 void load(graphlab::iarchive& arc) {
 arc >> ratings;
 }
}; // end of vertex data

/**
 * brief The edge data stores the weights between movies.
 */
struct edge_data : public graphlab::IS_POD_TYPE {

/**
 * brief The type of data on the edge;
 *
 * li *Train:* the observed value is correct and used in training
 * li *Validate:* the observed value is correct but not used in training
 */
 enum data_role_type { TRAIN, VALIDATE };

/**
 * brief The observed value for the edge */
 double obs;

/**
 * brief basic initialization */
 edge_data(double obs = 0, data_role_type role = TRAIN) :
 obs(obs), role(role) { }
}; // end of edge data

typedef graphlab::distributed_graph<vertex_data, edge_data> graph_type;
typedef graph_type::vertex_type vertex_type;
typedef graph_type::edge_type edge_type;

/**
 * brief The graph loader function is a line parser used for
 * distributed graph construction.
 */
bool graph_vertex_loader(graph_type& graph,
 const std::string& filename,
 const std::string& line) {

// Parse the line
 std::stringstream strm(line);
 graph_type::vertex_id_type vt(-1);

 strm >> vt;
 map ratings;

 while(1) {
 graphlab::vertex_id_type user;
 double rating;
 strm >> user;
 strm >> rating;
 if (strm.fail())
 break;
 ratings[user] = rating;
}
University of Southern California and Universitat Politècnica de Catalunya

Distributed graph signal processing

```cpp
graph.add_vertex(vt, vertex_data(ratings));
return true; // successful load
} // end of graph_loader

bool graph_edge_loader(graph_type& graph,
            const std::string& filename,
            const std::string& line) {

    // Parse the line
    std::stringstream strm(line);
    graph_type::vertex_id_type vt(-1), tmp(-1);
    strm >> vt;
    while(1) {
        strm >> tmp;
        if (strm.fail())
            break;
        graph.add_edge(vt, tmp);
    }
return true; // successful load
} // end of graph_loader

/**
 * \brief Edge program to calculate the vector cosine similarity on all the
 * edges of the graph
 */
void weights_calc(edge_type& edge) {
map ma, mb;
float num = 0, den1 = 0, den2 = 0;
int num_rat = 0;
ma = edge.source().data().ratings;
mb = edge.target().data().ratings;
for (map::iterator it = ma.begin(); it != ma.end(); ++it) {
    if (mb.find(it->first) != mb.end()) {
        num_rat++;
        num += it->second * mb[it->first];
        den1 += it->second * it->second;
        den2 += mb[it->first] * mb[it->first];
    }
}
// Check number of common ratings between vertices
if (num_rat > 5) // Should be around 3-10
    edge.data().obs = num / (sqrt(den1) * sqrt(den2));
else
    edge.data().obs = 0;
}

/**
 * \brief The vertex data stores the movie rating information.
 */
struct graph_writer {
    std::string save_vertex(graph_type::vertex_type vt) {
        return "";
    }
    std::string save_edge(graph_type::edge_type ed) {
        std::stringstream strm;
        if (ed.data().obs > 0.01) {
            strm << ed.source().id() << " " << ed.target().id() << " "
                 << ed.data().obs << "\n";
        } else
            strm << "";
        return strm.str();
    }
}; // end of pagerank writer

int main(int argc, char** argv) {
    graphlab::mpi_tools::init(argc, argv);
    graphlab::distributed_control dc;
    dc.cout() << "Loading graph." << std::endl;
    graphlab::timer timer;
    graph_type graph(dc);
    graph.load("out_rat..", graph_vertex_loader);
    graph.load("out_edg..", graph_edge_loader);
    dc.cout() << "Loading graph. Finished in "
              << timer.current_time() << std::endl;
    dc.cout() << "Finalizing graph." << std::endl;
timer.start();
    graph.finalize();
    dc.cout() << "Finalizing graph. Finished in "
```
Distributed graph signal processing

```cpp
<< timer.current_time() << std::endl;

dc.cout() << "========== Graph statistics on proc " << dc.procid()
<< " ==============="  
<< "Num vertices: " << graph.num_vertices()
<< "Num edges: " << graph.num_edges()
<< "Num replica: " << graph.num_replicas()
<< "Replica to vertex ratio: "
<< float(graph.num_replicas())/graph.num_vertices()
<< "--------------------------------------------"
<< "Num local own vertices: " << graph.num_local_own_vertices()
<< "Num local vertices: " << graph.num_local_vertices()
<< "Replica to own ratio: "
<< float(graph.num_local_vertices()/graph.num_local_own_vertices())
<< "Begin edge id: " << graph.global_eid(0)
<< "Edge balance ratio: "
<< float(graph.num_local_edges())/graph.num_edges()
<< std::endl;

dc.cout() << "Calculating edge values." << std::endl;
timer.start();
graph.transform_edges(weights_calc);

knn3.cpp: The code corresponding to the kNN algorithm implementation has been highlighted (Gather and Apply steps).

```cpp
/** *
 * \file knn3.cpp
 * \brief The third step for KNN rating prediction
 * This file contains the third step for KNN rating prediction. It reads the
 * graph created by the second step and loads it as graph where each vertex
 * is a movie and the weight of the edges is the vector cosine similarity.
 * It will load into each vertex the test data. Then each user rating will be
 * predicted using the KNN algorithm. After that, the Mean Square Error will be
 * computed as an average between all the test ratings and the KNN predicted
 * ratings.
 * */
#include <string>
#include <list>
#include <graphlab.hpp>
#include <boost/unordered_map.hpp>
#include <math.h>
#include <boost/math/special_functions/round.hpp>
using namespace graphlab;
typedef boost::unordered_map<vertex_id_type, double> map;

/** 
 * \brief The vertex data stores the movie rating information.
 */
struct vertex_data {
    /** \brief The ratings each user has given to the movie */
    map ratings;
    /** \brief The predicted ratings found using KNN */
    map ratings_knn;
    vertex_data() {}
    vertex_data(map ratings): ratings(ratings) {}  
    /** \brief Save the vertex data to a binary archive */
```
void save(graphlab::oarchive& arc) const {
    arc << ratings;
}; // end of vertex data

/**
 * \brief The edge data stores the weights between movies.
 * 
 * struct edge_data : public graphlab::IS_POD_TYPE {
 **/
 * \brief The type of data on the edge;
 * 
 * //Train:* the observed value is correct and used in training
 * \li *Validate:* the observed value is correct but not used in training
 * 
 * enum data_role_type { TRAIN, VALIDATE };
 */
 /** \brief The train/validation/test designation of the edge */
 /** \brief basic initialization */
 edge_data(data_role_type role = TRAIN) :
 obs(obs), role(role) { }
}; // end of edge data

typedef graphlab::distributed_graph<
 vertex_data, edge_data>
 graph_type;
typedef graph_type::vertex_type
 vertex_type;
typedef graph_type::edge_type
 edge_type;

/**
 * \brief The graph loader function is a line parser used for
 * distributed graph construction.
 */
 bool graph_loader(graph_type& graph,
 const std::string& filename,
 const std::string& line) {

 // Parse the line
 std::stringstream strm(line);
 graph_type::vertex_id_type va(-1), vb(-1);
 float weight;
 strm >> va >> vb >> weight;
 if (weight > 0.1)
 graph.add_edge(va, vb, edge_data(weight));
 return true; // successful load
} // end of graph_loader

bool graph_test_loader(graph_type& graph,
 const std::string& filename,
 const std::string& line) {

 // Parse the line
 std::stringstream strm(line);
 graph_type::vertex_id_type vt(-1), user(-1);
 float rating;
 map ratings;
 strm >> vt;
 while(1) {
    strm >> user >> rating;
    if (strm.fail())
 break;
    ratings[user] = rating;
 } 
 if (ratings.size() >= 1)
 graph.add_vertex(vt, vertex_data(ratings));
 return true; // successful load
} // end of graph_loader

class gather_type {
 public:
 map ratings;
Distributed graph signal processing

map weights;
/** \brief basic default constructor */
gather_type() { }
gather_type(vertex_id_type user, double rating) {
  ratings[user] = rating;
}
gather_type(map rat, float wei) {
  //ratings.insert(rat.begin(), rat.end());
  ratings = rat;
  for (map::iterator it = ratings.begin(); it != ratings.end(); ++it)
    weights[it->first] = wei;
}
/** \brief save the values to a binary archive */
void save(graphlab::oarchive& arc) const { arc << ratings; }
/** \brief Read the values from a binary archive */
void load(graphlab::iarchive& arc) { arc >> ratings; }
/** * \brief Joins two maps(ratings) along with the weights for each user */
gather_type& operator+=(const gather_type& other) {
  map sum_ratings;
  map sum_weights;
  map other_ratings = other.ratings;
  map other_weights = other.weights;
  for (map::iterator it = ratings.begin(); it != ratings.end(); ++it) {
    sum_ratings[it->first] = 0;
    sum_weights[it->first] = 0;
  }
  for (map::const_iterator it = other_ratings.begin(); it != other_ratings.end(); ++it) {
    sum_ratings[it->first] = 0;
    sum_weights[it->first] = 0;
  }
  for (map::iterator it = sum_ratings.begin(); it != sum_ratings.end(); ++it){
    if (ratings.find(it->first) != ratings.end()) {
      sum_ratings[it->first] += ratings[it->first];
      sum_weights[it->first] += weights[it->first];
    }
    if (other_ratings.find(it->first) != other_ratings.end()) {
      sum_ratings[it->first] += other_ratings[it->first];
      sum_weights[it->first] += other_weights[it->first];
    }
  }
  ratings = sum_ratings;
  weights = sum_weights;
  return *this;
} // end of operator+=
}; // end of gather type
/** * \brief Compute the KNN for each rating in the vertices */
class knn_program :
  public graphlab::ivertex_program<graph_type, gather_type>,
          public graphlab::IS_POD_TYPE {
public:
/** The set of edges to gather along */
edge_dir_type gather_edges(icontext_type& context, const vertex_type& vertex) const {
  return graphlab::OUT_EDGES;
}; // end of gather_edges
/** The gather function */
gather_type gather(const context_type& context, const vertex_type& vertex, const edge_type& edge) const {
  map wei_rat;
  vertex_data etd = edge.target().data();
  for (map::iterator it = etd.ratings.begin(); it != etd.ratings.end(); ++it)
    wei_rat[it->first] = edge.data().obs * etd.ratings[it->first];
  return gather_type(wei_rat, edge.data().obs);
} // end of gather function
void apply(const context_type& context, vertex_type& vertex, const gather_type& sum) {
  //for (map::iterator it; it != sum.ratings.end(); ++it)
  //  graph.add_edge(vertex.id(), it->first);
  graphlab::apply_sum(sum.ratings, vertex.data().obs);
} // end of apply function
map norm_knn;
map sum_ratings = sum.ratings;
map sum_weights = sum.weights;
for (map::const_iterator it = sum.ratings.begin(); it != sum.ratings.end(); ++it) {
    //std::cout << "(" << vertex.id() << " " << sum_ratings[it->first] << " " << sum_weights[it->first] << ");"
    norm_knn[it->first] = sum_ratings[it->first] / sum_weights[it->first];
}
vertex.data().ratings_knn = norm_knn;
} // end of apply

edge_dir_type scatter_edges(icontext_type& context, const vertex_type& vertex) const {
    return graphlab::NO_EDGES;
} // end of knn vertex program
typedef graphlab::omni_engine<knn_program> engine_type;

/**
 * brief Compute the error between the KNN predicted value and the test rating
 */
float error_vertex_data(engine_type::icontext_type& context, const graph_type::vertex_type& vertex) {
    float err = 0, tmp;
    vertex_data vd = vertex.data();
    if (vd.ratings.size() > 0) {
        for (map::iterator it = vd.ratings.begin(); it != vd.ratings.end(); ++it) {
            //std::cout << "(" << vertex.id() << " " << vd.ratings[it->first] << " " << vd.ratings_knn[it->first] << ");"
            //Check if the KNN was properly computed or not (Maybe the vertex didn't have neighbours)
            if (vd.ratings_knn[it->first] < 0.1)
                tmp = 0;
            else
                tmp = (vd.ratings[it->first] - boost::math::round(vd.ratings_knn[it->first]));
            err += tmp * tmp;
        }
        if (isnan(err)) {
            //std::cout << "NaN ";
            return 0;
        } else
            return err / vd.ratings.size();
    } else
        return 0;
}

/**
 * brief Output the result error
 */
void print_finalize(engine_type::icontext_type& context, float total) {
    context.cout() << "Knn Average MSE: " << total / context.num_vertices() << "\n";
}

int main(int argc, char*** argv) {
    graphlab::mpi_tools::init(argc, argv);
    graphlab::distributed_control dc;
    dc.cout() << "Loading graph. " << std::endl;
    graphlab::timer timer;
    graph_type graph(dc);
    // Load the graph containing the weights and connections
    graph.load("out_fin_", graph_loader);
    // Load the test user ratings (not used to build the graph)
    graph.load("out_test_rat_", graph_test_loader);
    dc.cout() << "Loading graph. Finished in " << timer.current_time() << std::endl;
    dc.cout() << "Finalizing graph. " << std::endl;
    timer.start();
    graph.finalize();
    dc.cout() << "Finalizing graph. Finished in " << timer.current_time() << std::endl;
    dc.cout() << "\n\n Num vertices: " << graph.num_vertices();
    dc.cout() << "\n Num edges: " << graph.num_edges();
    dc.cout() << "\n Num replica: " << graph.num_replicas();
    dc.cout() << "\n Replica to vertex ratio: " << float(graph.num_replicas()) / graph.num_vertices();
}
```cpp
<
"\nNum local own vertices: " << graph.num_local_own_vertices()
"\nNum local vertices: " << graph.num_local_vertices()
"\nReplica to own ratio: " << (float)graph.num_local_vertices()/graph.num_local_own_vertices()
"\nNum local edges: " << graph.num_local_edges()
// "\nBegin edge id: " << graph.global_eid(0)
"\nEdge balance ratio: " << float(graph.num_local_edges())/graph.num_edges()
" std::endl;

dc.cout() << "Creating engine" << std::endl;
graphlab::omni_engine<knn_program> engine(dc, graph, "sync");
engine.signal_all();

// Run KNN
dc.cout() << "Running ..." << std::endl;
timer.start();
engine.start();

const double runtime = timer.current_time();
dc.cout() << "---------------------------------------------"
< "Final Runtime (seconds): " << runtime
< "std::endl"
< "Updates executed: " << engine.num_updates() << std::endl
< "Update Rate (updates/second): " << engine.num_updates() / runtime << std::endl;

engine.add_vertex_aggregator<float>("error",
    error_vertex_data,
    print_finalize);

engine.aggregate_now("error");
graphlab::mpi_tools::finalize();
return EXIT_SUCCESS;
```
5.3 Chebyshev source code

The Chebyshev program takes as input two files describing the graph (one describes the topology and the other one describes the signal on the graph) and one file with the Chebyshev coefficients. The result of the filtering is saved in a new file.

cheby.cpp: The code corresponding to the Chebychev filtering implementation has been highlighted (The initialization of the values and the iterative part can be observed).

```cpp
/**
 * file cheby.cpp
 *
 * brief Filtering operation using chebyshev coefficients
 *
 * This file contains the code to apply the filtering operation using the chebyshev coefficients as input.
 */

#include <string>
#include <list>
#include <graphlab.hpp>
#include <math.h>
using namespace graphlab;

/** brief Interval of operation */
double arange[] = { 0.0, 2.0 };
double a1 = (arange[1] - arange[0]) / 2;
double a2 = (arange[1] + arange[0]) / 2;

/** brief vector to store the chebyshev coefficients */
// Use some random values for testing purposes
//double coeff[] = {2.23, 5.23, 0.19, 8.39};
std::vector<double> coeff;
unsigned int coeff_len;

/** brief index for the current iteration */
//int ind = 0;

/**
 * brief The vertex data stores the movie rating information.
 */
struct vertex_data {
    /** brief node degree value */
    double degree;

    /** brief Values to store the temporal results of each iteration */
    double twf_old, twf_cur, twf_new;

    /** brief Signal value */
    double val;

    unsigned int counter;

    /** brief basic initialization */
    vertex_data() {};

    vertex_data(double val):
        val(val), counter(2) {};
```
Distributed graph signal processing

```cpp
/**
 * 
 */
void save(graphlab::oarchive& arc) const {
    arc << degree << twf_old << twf_cur << twf_new << val << counter;
}

/**
 * 
 */
void load(graphlab::iarchive& arc) {
    arc >> degree >> twf_old >> twf_cur >> twf_new >> val >> counter;
}

/**
 * 
 */
struct edge_data : public graphlab::IS_POD_TYPE {
    /**
     * 
     */
    double wei;

    /**
     * 
     */
    edge_data(double wei = 0) :
        wei(wei) { }
}; // end of edge data

typedef graphlab::distributed_graph<vertex_data, edge_data> graph_type;
typedef graph_type::vertex_type vertex_type;
typedef graph_type::edge_type edge_type;

/**
 * 
 */
bool graph_loader(graph_type& graph,
        const std::string& filename,
        const std::string& line) {
    // Parse the line
    std::stringstream strm(line);
    graph_type::vertex_id_type va(-1), vb(-1);
    double weight;

    strm >> va >> vb >> weight;
    if (weight > 0.1) {
        graph.add_edge(va, vb, edge_data(weight));
        graph.add_edge(vb, va, edge_data(weight));
    }
    return true; // successful load
} // end of graph_loader

bool graph_signal_loader(graph_type& graph,
        const std::string& filename,
        const std::string& line) {
    // Parse the line
    std::stringstream strm(line);
    graph_type::vertex_id_type vt(-1);
    double val;

    strm >> vt >> val;
    graph.add_vertex(vt, vertex_data(val));
    return true; // successful load
} // end of graph_signal_loader
```
bool filter_loader(graph_type& graph,
    const std::string& filename,
    const std::string& line) {

    // Parse the line
    std::stringstream strm(line);
    double val;
    while (1) {
        strm >> val;
        if (strm.fail())
            break;
        coeff.push_back(val);
    }
    coeff_len = coeff.size();

    return true; // successful load
} // end of graph_signal_loader

/**
 * \brief Saves the users rating for each training user
 */
struct graph_signal_writer {
    std::string save_vertex(graph_type::vertex_type vt) {
        std::stringstream strm;
        strm << vt.id() << " " << vt.data().val << "n";
        return strm.str();
    }
    std::string save_edge(graph_type::edge_type e) { return ""; }
}; // end of pagerank writer

/**
 * \brief Compute the degree of each node and store it
 */
class degree_program :
    public graphlab::ivertex_program<graph_type, double>,
    public graphlab::IS_POD_TYPE
{
public:

    /**< The set of edges to gather along */
    edge_dir_type gather_edges(icontext_type& context,
        const vertex_type& vertex) const {
        return graphlab::OUT_EDGES;
    }; // end of gather_edges

    /**< The gather function */
    double gather(icontext_type& context, const vertex_type& vertex,
        edge_type& edge) const {
        return edge.data().wei;
    } // end of gather function

    void apply(icontext_type& context, vertex_type& vertex,
        const gather_type& sum) {
        vertex.data().degree = sum;
    } // end of apply

    // No scatter needed. Return NO_EDGES
    edge_dir_type scatter_edges(icontext_type& context,
        const vertex_type& vertex) const {
        return graphlab::NO_EDGES;
    }
}; // end of degree program
/**
 * \brief Compute the initial values and do the first iteration of the Chebyshev filtering.
 */

class init_values_program :
  public graphlab::ivertex_program<graph_type, double>,
  public graphlab::IS_POD_TYPE
{
public:

/** The set of edges to gather along */
  edge_dir_type gather_edges(icontext_type& context,
                            const vertex_type& vertex) const {
    return graphlab::OUT_EDGES;
  } // end of gather_edges

/** The gather function */
  double gather(icontext_type& context, const vertex_type& vertex,
                edge_type& edge) const {
    return edge.data().wei / (std::sqrt(
        edge.target().data().degree * edge.source().data().degree))
      * edge.target().data().val;
  } // end of gather function

void apply(icontext_type& context, vertex_type& vertex,
           const gather_type& sum) {
    vertex.data().twf_old = vertex.data().val;
    vertex.data().twf_cur = (vertex.data().val - sum -
                            a2 * vertex.data().val) / a1;

    vertex.data().val = 0.5 * coeff[0] * vertex.data().twf_old
                        + coeff[1] * vertex.data().twf_cur;
  } // end of apply

// No scatter needed. Return NO_EDGES
  edge_dir_type scatter_edges(icontext_type& context,
                              const vertex_type& vertex) const {
    return graphlab::NO_EDGES;
  }
}; // end of init values program

/**
 * \brief Compute the next Chebyshev iterations.
 */

class cheby_program :
  public graphlab::ivertex_program<graph_type, double>,
  public graphlab::IS_POD_TYPE
{
public:

/** The set of edges to gather along */
  edge_dir_type gather_edges(icontext_type& context,
                            const vertex_type& vertex) const {
    return graphlab::OUT_EDGES;
  } // end of gather_edges

/** The gather function */
  double gather(icontext_type& context, const vertex_type& vertex,
                edge_type& edge) const {
    return edge.data().wei / (std::sqrt(
        edge.target().data().degree * edge.source().data().degree))
      * edge.target().data().twf_cur;
  } // end of gather function

```
void apply(icontext_type& context, vertex_type& vertex, const gather_type& sum) {
    vertex.data().twf_new = (2 / a1) * (vertex.data().twf_cur - sum) - a2 * vertex.data().twf_cur - vertex.data().twf_old;
    vertex.data().val = vertex.data().val + coeff[vertex.data().counter] * vertex.data().twf_new;
    vertex.data().twf_old = vertex.data().twf_cur;
    vertex.data().twf_cur = vertex.data().twf_new;
    vertex.data().counter++;
    if (vertex.data().counter < coeff_len)
        context.signal(vertex);
} // end of apply

// No scatter needed. Return NO_EDGES
edge_dir_type scatter_edges(icontext_type& context, const vertex_type& vertex) const {
    return graphlab::NO_EDGES;
} // end of cheby program

int main(int argc, char** argv) {
    graphlab::mpi_tools::init(argc, argv);
    graphlab::distributed_control dc;
    dc.cout() << "Loading graph." << std::endl;
    graphlab::timer timer;
    graph_type graph(dc);
    // Load the filter coefficients
    graph.load("coeff", filter_loader);
    // Load the graph containing the weights and connections
    graph.load("graph_topology", graph_loader);
    // Load the signal of the graph
    graph.load("graph_signal", graph_signal_loader);
    timer.start();
    graph.finalize();
    dc.cout() << "Finalizing graph. Finished in "
              << timer.current_time() << std::endl;
    dc.cout() << "Finalizing graph. Finished in "
              << timer.current_time() << std::endl;
    dc.cout() << "======= Graph statistics on proc " << dc.procid() << " ======
    dc.cout() << "Num vertices: " << graph.num_vertices() << std::endl;
    dc.cout() << "Num edges: " << graph.num_edges() << std::endl;
    dc.cout() << "Num replica: " << graph.num_replicas() << std::endl;
    dc.cout() << "Replica to vertex ratio: "
              << float(graph.num_replicas()) / graph.num_vertices() << std::endl;
    dc.cout() << "Num local own vertices: " << graph.num_local_own_vertices() << std::endl;
    dc.cout() << "Num local vertices: " << graph.num_local_vertices() << std::endl;
    dc.cout() << "Replica to own ratio: "
              << (float)graph.num_local_vertices() / graph.num_local_own_vertices() << std::endl;
    dc.cout() << "Num local edges: " << graph.num_local_edges() << std::endl;
}
Distributed graph signal processing

```cpp
// Edge id: 0
// Edge balance ratio: 
 float(graph.num_local_edges())/graph.num_edges()

dc.cout() << "Creating engine 1 (Calculate degrees)" << std::endl;
graphlab::omni_engine<degree_program> engine1(dc, graph, "sync");
engine1.signal_all();

// Calculate degrees
dc.cout() << "Running ..." << std::endl;
timer.start();
engine1.start();

double runtime = timer.current_time();
dc.cout() << "Final Runtime (seconds): " << runtime
          << std::endl
          << "Updates executed: " << engine1.num_updates() << std::endl
          << "Update Rate (updates/second): "
          << engine1.num_updates() / runtime << std::endl;

dc.cout() << "Creating engine 2 (Init values + 2 iterations)" << std::endl;
graphlab::omni_engine<init_values_program> engine2(dc, graph, "sync");
engine2.signal_all();

// Run init values + first 2 iterations
dc.cout() << "Running ..." << std::endl;
timer.start();
engine2.start();

dc.cout() << "Final Runtime (seconds): " << runtime
          << std::endl
          << "Updates executed: " << engine2.num_updates() << std::endl
          << "Update Rate (updates/second): "
          << engine2.num_updates() / runtime << std::endl;

dc.cout() << "Creating engine 3 (Chebyshev filtering iterations)" << std::endl;
graphlab::omni_engine<cheby_program> engine3(dc, graph, "sync");
engine3.signal_all();

// Run Iterative filtering operations
dc.cout() << "Running ..." << std::endl;
timer.start();
engine3.start();

dc.cout() << "Final Runtime (seconds): " << runtime
          << std::endl
          << "Updates executed: " << engine3.num_updates() << std::endl
          << "Update Rate (updates/second): "
          << engine3.num_updates() / runtime << std::endl;

graph.save("graph_filtered_signal", graph_signal_writer(),
false, // do not gzip
```

5.4 Degree 3 Polynomials source code

The Degree 3 Polynomials program takes as input two files describing the graph (one describes the topology and the other one describes the signal on the graph) and one file with the polynomials coefficients. The result of the filtering is saved in a new file.

poly.cpp: The code corresponding to the algorithm implementation has been highlighted (The two iterative steps can be observed)

```cpp
/**
 * \file poly.cpp
 * \brief Filtering operation using polynomial factorization (degree 3) coefficients
 * This file contains the code to apply the filtering operation using the degree 3 coefficients from a polynomial factorization.
 */
#include <string>
#include <list>
#include <graphlab.hpp>
#include <math.h>
using namespace graphlab;

// Use some random values for testing purposes
//double coeff[] = {2.23, 5.23, 0.19};
std::vector<double> coeff;
unsigned int coeff_len;

/** \brief index for the current iteration */
int ind = 0;

/** \brief The vertex data stores the movie rating information. */
struct vertex_data {
    /** \brief node degree value */
    double degree;

    /** \brief Values to store the temporal results of each iteration */
    double tmp, part_a, part_b;

    /** \brief Signal value */
    double val;

    /** \brief basic initialization */
    vertex_data() { }
}
vertex_data(double val):
```
Distributed graph signal processing

val(val) { }
/**
 * brief Save the vertex data to a binary archive */
void save(graphlab::oarchive &arc) const {
    arc << degree << tmp << part_a << part_b << val << counter;
}
/**
 * brief Load the vertex data from a binary archive */
void load(graphlab::iarchive &arc) {
    arc >> degree >> tmp >> part_a >> part_b >> val >> counter;
}; // end of vertex data
/**
 * brief The edge data stores the weights between movies.
 */
struct edge_data : public graphlab::IS_POD_TYPE {
    /**
     * brief the weight value for the edge */
    double wei;
    /**
     * brief basic initialization */
    edge_data(double wei = 0) :
        wei(wei) { }
}; // end of edge data
typedef graphlab::distributed_graph<vertex_data, edge_data> graph_type;
typedef graph_type::vertex_type vertex_type;
typedef graph_type::edge_type edge_type;
/**
 * brief The graph loader function is a line parser used for
 * distributed graph construction.
 */
bool graph_loader(graph_type& graph,
    const std::string& filename,
    const std::string& line) {
    // Parse the line
    std::stringstream strm(line);
    graph_type::vertex_id_type va(-1), vb(-1);
    double weight;
    strm >> va >> vb >> weight;
    if (weight > 0.1) {
        graph.add_edge(va, vb, edge_data(weight));
        graph.add_edge(vb, va, edge_data(weight));
    }
    return true; // successful load
} // end of graph_loader

bool graph_signal_loader(graph_type& graph,
    const std::string& filename,
    const std::string& line) {
    // Parse the line
    std::stringstream strm(line);
    graph_type::vertex_id_type vt(-1);
    double val;
    strm >> vt >> val;
    graph.add_vertex(vt, vertex_data(val));
    return true; // successful load
bool filter_loader(graph_type& graph,  
const std::string& filename,  
const std::string& line) {  
    // Parse the line  
    std::stringstream strm(line);  
    double val;  
    while (1) {  
        strm >> val;  
        if (strm.fail())  
            break;  
        coeff.push_back(val);  
    }  
    coeff_len = coeff.size();  
    return true; // successful load  
}  

/**  
* \brief Saves the users rating for each training user  
*/  
struct graph_signal_writer {  
    std::string save_vertex(graph_type::vertex_type vt) {  
        std::stringstream strm;  
        strm << vt.id() << " " << vt.data().val << "\n";  
        return strm.str();  
    }  
    std::string save_edge(graph_type::edge_type e) { return ""; }  
};  
/**  
* \brief Compute the degree of each node and store it  
*/  
class degree_program : public graphlab::ivertex_program<graph_type, double>,  
public graphlab::IS_POD_TYPE  
{  
    public:  
        /** The set of edges to gather along */  
        edge_dir_type gather_edges(icontext_type& context,  
const vertex_type& vertex) const {  
            return graphlab::OUT_EDGES;  
        }  
        // end of gather_edges  
        /** The gather function */  
        double gather(icontext_type& context, const vertex_type& vertex,  
edge_type& edge) const {  
            return edge.data().wei;  
        }  
        // end of gather function  
        void apply(icontext_type& context, vertex_type& vertex,  
const gather_type& sum) {  
            vertex.data().degree = sum;  
        }  
        // end of apply  
        /* No scatter needed. Return NO_EDGES  
edge_dir_type scatter_edges(icontext_type& context,  
const vertex_type& vertex) const {  
        return graphlab::NO_EDGES;  
    }  
}  
// end of graph_signal_loader
Distributed graph signal processing

/**
 * \brief Compute the binomial operations step_a.
 */
class degree3poly_a_program {
public graphlab::ivertex_program<graph_type, double>,
public graphlab::IS_POD_TYPE
{
public:

/** The set of edges to gather along */
edge_dir_type gather_edges(icontext_type& context,
const vertex_type& vertex) const {
    return graphlab::OUT_EDGES;
}; // end of gather_edges

/** The gather function */
double gather(icontext_type& context, const vertex_type& vertex,
edge_type& edge) const {
    return edge.data().wei / (std::sqrt(
        edge.target().data().degree * edge.source().data().degree)) * edge.target().data().val;
} // end of gather function

void apply(icontext_type& context, vertex_type& vertex,
const gather_type& sum) {
    vertex.data.part_a = (coeff[ind] + coeff[ind + 1]) * vertex.data.val;
    - coeff[ind + 1] * sum;
    vertex.data.tmp = vertex.data.val - sum;
}; // end of degree3poly step_a program

/**
 * \brief Compute the binomial operations step_b.
 */
class degree3poly_b_program {
public graphlab::ivertex_program<graph_type, double>,
public graphlab::IS_POD_TYPE
{
public:

/** The set of edges to gather along */
edge_dir_type gather_edges(icontext_type& context,
const vertex_type& vertex) const {
    return graphlab::OUT_EDGES;
}; // end of gather_edges

/** The gather function */
double gather(icontext_type& context, const vertex_type& vertex,
edge_type& edge) const {
    return edge.data().wei / (std::sqrt(
        edge.target().data().degree * edge.source().data().degree)) * edge.target().data().tmp;
} // end of gather function

void apply(icontext_type& context, vertex_type& vertex,
const gather_type& sum) {
    vertex.data.part_b = coeff[ind + 2] * (vertex.data.tmp - sum);
    vertex.data.val = vertex.data.part_a + vertex.data.part_b;
} // end of apply

// No scatter needed. Return NO_EDGES
edge_dir_type scatter_edges(context_type& context,
    const vertex_type& vertex) const {
    return graphlab::NO_EDGES;
}
}; // end of degree3poly step_b program

int main(int argc, char** argv) {
    graphlab::mpi_tools::init(argc, argv);
    graphlab::distributed_control dc;

dc.cout() << "Loading graph." << std::endl;
    graphlab::timer timer;
    graph_type graph(dc);
    // Load the filter coefficients
    graph.load("coeff", filter_loader);
    // Load the graph containing the weights and connections
    graph.load("graph_topology", graph_loader);
    // Load the signal of the graph
    graph.load("graph_signal", graph_signal_loader);
    dc.cout() << "Loading graph. Finished in "
      << timer.current_time() << std::endl;

dc.cout() << "Filter lenght: " << coeff_len << std::endl;

dc.cout() << "Finalizing graph." << std::endl;
    timer.start();
    graph.finalize();
    dc.cout() << "Finalizing graph. Finished in "
      << timer.current_time() << std::endl;

dc.cout() << "========== Graph statistics on proc " << dc.procid() << " ===============
\n| Num vertices: | Num edges: | Num replica: | Replica to vertex ratio: | Num local own vertices: | Num local edges: | Edge balance ratio: |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>graph.num_vertices()</td>
<td>graph.num_edges()</td>
<td>graph.num_replicas()</td>
<td>float(graph.num_replicas()) / graph.num_vertices()</td>
<td>graph.num_local_own_vertices()</td>
<td>graph.num_local_edges()</td>
<td>float(graph.num_local_edges()) / graph.num_edges()</td>
</tr>
</tbody>
</table>

    std::endl;

dc.cout() << "Creating engine 1 (Calculate degrees)" << std::endl;
    graphlab::omni_engine<degree_program> engine1(dc, graph, "sync");
    engine1.signal_all();

    // Calculate degrees
    dc.cout() << "Running ..." << std::endl;
    timer.start();
Distributed graph signal processing

distributed_graph_signal_processing()

double runtime = timer.current_time();
dc.cout() << "----------------------------------------------------------"
   << std::endl
   << "Final Runtime (seconds): " << runtime
   << std::endl
   << "updates executed: " << engine1.num_updates() << std::endl
   << "Update Rate (updates/second): "
   << engine1.num_updates() / runtime << std::endl;

dc.cout() << "Creating engine 2 iteration step_a" << std::endl;
graphlab::omni_engine<degree3poly_a_program> engine2(dc, graph, "sync");

dc.cout() << "Creating engine 3 iteration step_b" << std::endl;
graphlab::omni_engine<degree3poly_b_program> engine3(dc, graph, "sync");

for (unsigned int i = 0; i*3 < coeff_len; i++) {
  engine2.signal_all();

  // Run step_a
  dc.cout() << "Running step a..." << std::endl;
timer.start();
engine2.start();

  runtime = timer.current_time();
dc.cout() << "----------------------------------------------------------"
   << std::endl
   << "Final Runtime (seconds): " << runtime
   << std::endl
   << "updates executed: " << engine2.num_updates() << std::endl
   << "Update Rate (updates/second): "
   << engine2.num_updates() / runtime << std::endl;

  engine3.signal_all();

  // Run step_b
  dc.cout() << "Running step b..." << std::endl;
timer.start();
engine3.start();

  runtime = timer.current_time();
dc.cout() << "----------------------------------------------------------"
   << std::endl
   << "Final Runtime (seconds): " << runtime
   << std::endl
   << "updates executed: " << engine3.num_updates() << std::endl
   << "Update Rate (updates/second): "
   << engine3.num_updates() / runtime << std::endl;
  ind++;
}

graph.save("graph_filtered_signal", graph_signal_writer(),
false, // do not gzip
true, // save vertices
false); // do not save edges

graphlab::mpi_tools::finalize();
return EXIT_SUCCESS;}
5.5 Graph tools source code

`mega_graph.py`: This tool generates a random graph using as parameters the number of nodes and the connectivity percentage. The output is saved in two files: one describing the topology and the other one describing the signal on the graph.

```python
#!/usr/bin/env python
import sys, random

size = int(sys.argv[1])
conn = float(sys.argv[2])

print 'Generating signal...

# write signal file
with open("graph_signal.txt", 'w') as f:
    for i in range(size):
        f.write(str(i + 1) + ' ' + str(random.uniform(0, 10)) + '
')

print 'Generating topology...

# links = set()
# n_links = (conn * (size**2))
with open("graph_topology.txt", 'w') as f:
    while len(links) < n_links:
        a = random.randint(1,size)
        b = random.randint(1,size)
        #a = round(random.random()*(size-1))+1
        #b = round(random.random()*(size-1))+1
        if a != b:
            links.add((a,b))
        for link in links:
            wei = random.random()
            f.write(str(link[0]) + ' ' + str(link[1]) + ' ' + '{0:.2f}'.format(wei) + '
')
```

`scale_test.sh`: This tool runs the Chebyshev filtering program iterating over different values for the number of coefficients in the filter, the percentage of connectivity in the graph and the number of nodes in the graph. The time spent for each test is saved in a file.

```bash
#!/bin/bash

echo -n "" > scale_res2.txt
# N coeff, 50000 nodes, 1% conn
./mega_graph.py 50000 0.01
for coeff in $(seq 10 10 100)
do
cat _coeff_1000.txt | cut -d " " -f1-$coeff > coeff.txt
./cheby > tmp_out.txt
echo 50000 0.01 $coeff >> scale_res2.txt
cat tmp_out.txt | grep "Finished in" >> scale_res2.txt
cat tmp_out.txt | grep "Final Runtime" >> scale_res2.txt
done

# N conn, 50000 nodes, 64 coeff
for conn in $(seq 0.005 0.005 0.05)
do
./mega_graph.py 50000 $conn
cat _coeff_1000.txt | cut -d " " -f1-64 > coeff.txt
```
Distributed graph signal processing

```bash
./cheby > tmp_out.txt
echo 50000 $conn 64 >> scale_res2.txt
cat tmp_out.txt | grep "Finished in" >> scale_res2.txt
cat tmp_out.txt | grep "Final Runtime" >> scale_res2.txt
done

# 1% conn, N nodes, 64 coeff
for nodes in $(seq 5000 5000 50000)
do
  ./mega_graph.py $nodes 0.01
  cat _coeff_1000.txt | cut -d " " -f1-64 > coeff.txt
  ./cheby > tmp_out.txt
  echo $nodes 0.01 64 >> scale_res2.txt
  cat tmp_out.txt | grep "Finished in" >> scale_res2.txt
  cat tmp_out.txt | grep "Final Runtime" >> scale_res2.txt
done
```
6. References

1. C. Godsil, G. F. Royle, Algebraic Graph Theory. Springer, 2001


11. J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs”, OSDI, 2012