

Institute of Parallel and Distributed Systems

University of Stuttgart  
Universitätsstraße 38  
D-70569 Stuttgart

Masterarbeit

## **Massive Hypergraph Partitioning**

Heiko Geppert

**Course of Study:** Softwaretechnik  
**Examiner:** Prof. Dr. Kurt Rothermel  
**Supervisor:** Dr. rer. nat. Sukanya Bhowmik

**Commenced:** April 16, 2019  
**Completed:** October 16, 2019



## Abstract

The hypergraph model is a more generic version of the graph model. Instead of edges connecting only two vertices, hyperedges can connect an arbitrary number of vertices. This way, the hypergraph model has a higher expressiveness than the original graph model. It can model group memberships in social media or group chats in messengers easily. In the past years both, social media and instant messengers, gained much more popularity. The increased number of users and the thus increasing traffic result in larger hypergraph data sets, which can not be processed by a single machine. Hence, the hypergraphs need to be partitioned, thereby maintaining a low communication overhead in the later processing and having balanced partition sizes to ensure efficient parallelization. There are already many hypergraph partitioning algorithms so far, yet they are often limited in their parallelization, which is needed to process massive hypergraphs. A known paradigm which can be parallelized easily is label propagation. We created three approaches called *Credit Label Propagation (CLP)*, *Split Credit Label Propagation (SCLP)* and *Split Credit Label Propagation – Credit Value Compensation (SCLP-cc)* based on label propagation introducing new balancing punishments to improve the partitions balance leading to a better scalability. Thereby we introduced a credit value to regulate the growth of the label clusters. In the later algorithms we modified the credit value handling to prevent clusters from becoming too large, by regulating the amount of credit value available in the system. Further, we created a novel hypergraph partitioning algorithm called *Gravity Expansion* based on graph visualization algorithms. The hypergraphs vertices are placed in the center of a two-dimensional space. Next they are expanded into the space while keeping connected vertices close due to vertex movements via barrier jumps. After the expansion two gravity holes are placed which absorb and therefore partition the vertices. During the absorbing the vertices are further expanded to react to previous assignments. The *SCLP-cc* algorithm outperformed the *Label Propagation Partitioning* algorithm from the *hyperX* framework in terms of balance and cut size. Our *Gravity Expansion* algorithm outperformed *hMetis* when creating many subgraphs. In addition, it outperformed the publicly available version of *Hype* in terms of the cut size on some input graphs. Summarized, we provide new approaches for label propagation partitioning algorithms to increase the balancing. Further, we created a novel scalable partitioning algorithm which introduces the field of visualization algorithms to the partitioning problem.

## Kurzfassung

Das Hypergraph-Modell ist eine Verallgemeinerung des Graph-Modells. Hyperkanten verbinden eine beliebige Anzahl an Knoten, anstatt nur zwei wie es beim Graph-Modell der Fall ist. Somit kann das Hypergraph-Modell mehr Datenstrukturen darstellen. Gruppenmitgliedschaften in sozialen Netzwerken oder Gruppenkommunikationen in Sofortnachrichtendiensten können einfach modelliert werden. In den letzten Jahren gewannen sowohl Soziale Netzwerke als auch Sofortnachrichtendienste sehr viel Beliebtheit. Steigende Nutzerzahlen und der somit steigende Datenverkehr führten zu größeren Hypergraphen, die nicht mehr von einer einzelnen Maschine verarbeitet werden können. Daher müssen die Hypergraphen partitioniert werden, wobei ein geringer Kommunikationsoverhead bei der späteren Verarbeitung und gleichmäßige Partitionsgrößen, um effiziente Parallelisierbarkeit zu ermöglichen, benötigt werden. Es gibt bereits viele Partitionsalgorithmen für Hypergraphen, aber diese sind meist in ihrer Parallelisierbarkeit limitiert, die wiederum notwendig ist um riesige Hypergraphen effizient zu verarbeiten. Ein bekanntes parallelisierbares Paradigma ist Label Propagation. Wir entwickelten drei Ansätze basierend auf Label Propagation, die neue Möglichkeiten vorstellen, die Balance der Partitionen zu verbessern und somit die Parallelisierbarkeit verbessern. Namentlich sind dies *Credit Label Propagation (CLP)*, *Split Credit Label Propagation (SCLP)* und *Split Credit Label Propagation-Credit Value Compensation (SCLP-cc)*. Wir nutzten einen Kreditwert, um das Wachstum der Label-Cluster zu regulieren. Bei einigen Ansätzen modifizierten wir den Umgang mit dem Kreditwert, um zu verhindern, dass einzelne Cluster zu groß werden, indem wir die global verfügbare Menge an Kreditwert regulierten. Zudem entwickelten wir einen neuen Partitionierungsalgorithmus basierend auf Graphvisualisierungsalgorithmen für Hypergraphen namens *Gravity Expansion*. Dabei werden die Knoten des Hypergraphen zunächst in der Mitte eines zweidimensionalen Raums platziert. Von dort aus expandieren diese in den Raum, wobei verbundene Knoten mittels Barrier Jumps nahe beieinanderbleiben. Nach dem Expandieren werden zwei Gravitationspunkte erstellt, die Knoten in ihrer Nähe absorbieren und den Hypergraphen somit partitionieren. Während der Absorbierungsphase expandieren die Knoten weiter, um auf die bisherige Partitionierung zu reagieren. Der *SCLP-cc* Algorithmus übertraf den *Label Propagation Partitioning* Algorithmus aus dem *hyperX* Framework bezüglich der geschnittenen Hyperkanten und der Balance der Partitionsgrößen. Unser *Gravity Expansion* Algorithmus übertraf *hMETIS* wenn besonders viele Partitionen angelegt wurden. Zudem übertraf er bezüglich der geschnittenen Hyperkanten bei einigen Eingabedaten die öffentlich verfügbare Version von *HYPE*. Zusammengefasst präsentieren wir neue Ansätze im Bereich der Label Propagation Algorithmen, die die Balance der Partitionsgrößen verbessern. Außerdem stellen wir einen neuen skalierbaren Partitionierungsalgorithmus vor, der Visualisierungsalgorithmen auf das Partitionierungsproblem anwendet.

# Contents

<b>1</b>	<b>Introduction</b>	<b>15</b>
<b>2</b>	<b>Background</b>	<b>19</b>
2.1	Graph And Hypergraph Models . . . . .	19
2.2	Graph Processing . . . . .	20
2.3	Massive Hypergraph Partitioning Framework . . . . .	23
<b>3</b>	<b>Related Work</b>	<b>27</b>
<b>4</b>	<b>Problem Formulation</b>	<b>31</b>
<b>5</b>	<b>Algorithms</b>	<b>35</b>
5.1	Label Propagation . . . . .	35
5.2	Graph Layout Partitioning . . . . .	42
<b>6</b>	<b>Evaluations</b>	<b>53</b>
6.1	Environment . . . . .	53
6.2	Data Sets . . . . .	54
6.3	Label Propagation . . . . .	55
6.4	Gravity Expansion . . . . .	60
6.5	Discussion . . . . .	71
<b>7</b>	<b>Conclusion</b>	<b>73</b>
	<b>Bibliography</b>	<b>75</b>



## List of Figures

2.1	A undirected graph on the left side and a directed graph on the right side. . . . .	19
2.2	A visualization of a hypergraph with 11 vertices and 6 hyperedges. . . . .	21
2.3	The moreno-crime graph visualized using different layout algorithms; <i>Fruchterman-Reingold</i> on the left, <i>forceAtlas2</i> in the center and <i>openOrd</i> on the right. . . . .	22
2.4	Architecture overview of our partitioning framework . . . . .	24
5.1	Example graph . . . . .	39
5.2	SCLP example. Seed placement on the upper left, Starting state in the upper right, first iteration in the lower left, second iteration in the lower right . . . . .	41
5.3	The same graph twice, one time with a good graph layout and one time with a random layout . . . . .	43
5.4	Barrier Jump example . . . . .	44
5.5	Gravity hole placement example on the moreno-crime graph . . . . .	47
5.6	GE example. The graph is shown in a pretty visualization in a), the expansion, absorbing and reexpansion steps afterwards. . . . .	48
6.1	Cut size comparison of the LP algorithms on the connected GitHub graph . . . . .	56
6.2	Maximal imbalance comparison of the LP algorithms on the connected GitHub graph . . . . .	56
6.3	Runtime comparison of the LP algorithms on the connected GitHub graph . . . . .	57
6.4	Cut size comparison of the LP algorithms on the connected stackoverflow graph . . . . .	58
6.5	Maximal imbalance comparison of the LP algorithms on the connected stackoverflow graph . . . . .	58
6.6	Cut size comparison of the <i>SCLP</i> algorithm with different iteration parameters on the connected GitHub graph . . . . .	59
6.7	Maximal imbalance comparison of the <i>SCLP</i> algorithm with different iteration parameters on the connected GitHub graph . . . . .	59
6.8	Cut size comparison of <i>GE</i> with different initial iterations on the moreno-crime graph . . . . .	61
6.9	Runtime comparison of <i>GE</i> with different initial iterations on the moreno-crime graph . . . . .	61
6.10	Cut size comparison of <i>GE</i> with different reexpansion rates on the moreno-crime graph . . . . .	62
6.11	Runtime comparison of <i>GE</i> with different reexpansion rates on the moreno-crime graph . . . . .	63
6.12	Cut size comparison of <i>GE</i> with different grid sizes on the moreno-crime graph . . . . .	64
6.13	Runtime comparison of <i>GE</i> with different grid sizes on the moreno-crime graph . . . . .	64
6.14	Cut size comparison of several random jump modifications on the moreno-crime graph . . . . .	65
6.15	Runtime comparison of several random jump modifications on the moreno-crime graph . . . . .	66
6.16	Cut size comparison of several <i>GE</i> configurations on the GitHub graph . . . . .	67

6.17	Runtime comparison of several <i>GE</i> configurations on the GitHub graph . . . . .	67
6.18	Cut size comparison of several <i>GE</i> configurations on the stackoverflow graph . .	68
6.19	Runtime comparison of several <i>GE</i> configurations on the stackoverflow graph . .	68
6.20	Cut size comparison of several algorithms on the GitHub graph . . . . .	70
6.21	Runtime comparison of several algorithms on the GitHub graph . . . . .	70
6.22	Cut size comparison of several algorithms on the stackoverflow graph . . . . .	71



## List of Tables

2.1	Variables used in this thesis . . . . .	22
6.1	Servers used for the evaluation . . . . .	53
6.2	graphs used for the evaluation . . . . .	55



## List of Algorithms

5.1	High level CLP algorithm . . . . .	36
5.2	Seed initialization for CLP . . . . .	37
5.3	Vertex program for CLP . . . . .	37
5.4	High level SCLP algorithm . . . . .	38
5.5	<i>selectLabel</i> method of the SCLP algorithm . . . . .	40
5.6	<i>splitCreditValue</i> method of the SCLP algorithm . . . . .	40
5.7	Rebalancing procedure of the SCLP-cc algorithm . . . . .	42
5.8	High level gravity expansion bi-partitioning algorithm . . . . .	45
5.9	Gravity Expansion vertex program . . . . .	46
5.10	Weighted Centroid Calculation . . . . .	50



# Acronyms

**CLP** Credit Label Propagation. 36

**CV** credit value. 36

**GE** Gravity Expansion. 44

**LP** Label Propagation. 35

**LPP** Label Propagation Partitioning. 35

**SCLP** Split Credit Label Propagation. 38

**SCLP-cc** Split Credit Label Propagation - Credit Value Compensation. 42



# 1 Introduction

In the last years the digitization increased massively. The number of people using the digital world and the connectivity grew just as well. From this evaluation arise new challenges in handling the data provided by this new world.

Graphs are a generic data structure capable of representing data elements, which are somehow connected. A graph consists of vertices and edges connecting the vertices. This way, the data of many domains like social networks, food chains, electrical power grids, web pages on the internet, router level connectivity of the internet, scientific paper citations as well as many more domains can be stored and processed [New18]. For example, the vertices of a graph could be used to represent users of a social network and the edges to represent friendship relations between them. However, the graph model has its limits. Each edge in a graph connects exactly two vertices. When representing group memberships, we can use bipartite graphs, where we have two types of vertices. One type would be users and the other type groups. An edge would then represent a group membership. One drawback of this representation is the increasing number of vertices and especially edges [BC08]. Another drawback is that bipartite graphs originally do not allow vertices of the same type to be directly connected by an edge, but only via a vertex of the other type. These drawbacks can be avoided, when not using the standard graph model, but the more generic hypergraph model. A hypergraph is a graph, where edges can connect an arbitrary number of vertices. The edges in a hypergraph are called hyperedges. Within this graph model, a group can be represented by a single hyperedge, which connects all its members. We can represent the same data using way less edges, because we have only one edge per group and can still directly connect vertices if necessary. In the real world hypergraphs can be used to represent group memberships in social networks like Facebook or participation in Reddit's subreddits. Further, it can be used for communication structures in messengers like WhatsApp, where the vertices are the users and an edge represent a chat. Group chats can have multiple members, while direct chats are represented as hyperedges with just two members. On all these graphs, we can use graph algorithms like centrality metrics to find the best influencer for a specific target audience or use epidemic simulations for how rumors and other information spread in a community [New18]. Another domain are product recommendations, where every vertex might be a user and the edges the products they bought. When a vertex (user) is connected with another vertex via multiple edges (products), there is a good chance they have common product interests and suggesting the first user some products the second bought might be a good idea.

Social networks like Facebook or messengers like WhatsApp continue to grow and so does the data they need to handle. The larger the data set, the harder the methods of analysis become. Considering the millions of users which are accessing data simultaneously, it is obvious that a single machine can not handle the load. Further, the users are spread all over the world and therefore most users would be too far away from the single machine to maintain a feasible latency, even if the machine could handle the load. Hence, there is a need for multiple machines and data replication just to be able to handle the load of the users and to keep the latency low. Yet, there is an additional problem. The

data set itself is way too large to be stored on a single machine, especially when the data is kept in the RAM for faster processing [AS12]. The solution is to partition the data. This way, several machines hold pieces of the data set and together they have the whole set. Summarized, a distributed system is needed, spread across the globe replicating the data as well as splitting it to handle the size of the data set and the amount of users. All machines in the system have to keep in sync somehow and still need to provide short query processing and latency times. The algorithm to partition the data set can have a great impact on the resulting performance. Whenever a server needs some data it does not hold, it needs to send a request to a machine that holds the data and the data needs to be sent back to the first machine via the network. This takes obviously more time than processing local data would have taken. So finding cuts which result in as little server to server communication as possible is crucial. If we got such cuts, we can store the data as close to the users as possible and most user requests can be handled without any inter-server communication. This way, the scaling becomes better, because we do not need as much disk space or RAM anymore and less bandwidth. However, we need some preprocessing steps to partition the data. Hence, we need some computational time and resources beforehand, to save them later. Our goal is therefore to create partitioning algorithms which are fast, scalable and yield good results. There is no perfect solution yet, since the problem of (hyper)graph partitioning is known to be NP-hard [AR06]. Thereby all partitioning algorithms are doing a trade-off between a good partitioning quality, a short runtime and their scalability.

When looking at the just named attributes involved in the trade-off, some are pretty clear. The runtime is the overall time needed to cut a graph in multiple sub-graphs. Scalability consists of two attributes: parallelization and low memory usage. The parallelization is the degree by which the partitioning can be performed by multiple cores or even machines simultaneously. In the easiest case we have no parallelization and therefore no need to worry about race conditions and parallelization overhead. However, this would increase the runtime. When multiple CPU cores can work together on the problem, the runtime benefits a lot. Meanwhile, the partitioning quality might suffer from the parallelization, since the decisions of every core are not as well-informed as in the single core case. The parallelization is therefore another trade-off between runtime and partitioning quality. However, some algorithms perform pretty well when parallelized, especially when using iterative algorithms, where a single core would take the same decisions as multiple cores. Low memory usage relates to the amount of RAM needed by the algorithm. The overall goal is to stay in a feasible range of RAM required. When holding the whole graph and lots of metrics, the amount becomes quite large, but allows to make very informed decisions. Simply reading the graphs vertices in a stream and assigning them randomly needs nearly no RAM. However, the result will be of bad quality. Speaking of it, the quality of a partitioning can be measured by the overhead to be expected due to inter-server communication. Within this thesis we partition hypergraphs by assigning vertices to partitions. Hyperedges can span multiple partitions. Whenever a server of the distributed system processing the partitioned graph later needs to access data via such an edge, we need inter-server communication, because another member of the system has to send the required information via the wire. The quality of a partitioning can therefore be measured by the number of edges spanning more than one machine and the number of machines involved per edge. This represents how much information has to be shared between the machines later to make the processing possible and also how well each machine in the distributed system can work on its own.

Previous partitioning algorithms have many problems like missing scalability [KK00; MMB+18] or the algorithm trades too much partitioning quality for scalability [JQY+18]. Some also try to solve the partitioning problem by solving a load balance problem [KKP+17]. This is totally fine for the result and has some advantages like the possibility of distributed partitioning. However, it also inherits



---

is drawbacks, since it needs inter-server communication during the partitioning already. Further, many partitioning algorithms proposed by science so far solve the partitioning problem in a limited way, like only on very small graphs or with highly imbalanced partitions. The label propagation algorithms for graph partitioning, without actually doing load shedding, so far do not create balanced partitions. Further, they are mostly created for regular graphs, rarely for hypergraphs. We introduce new penalty functions combined with credit values to improve the partition's balancing. By sticking to plain label propagation we ensure a high degree of parallelization. Also we test our algorithms with large real world hypergraphs.

The field of graph visualization yields a lot of algorithms [BEGT13; FR91; JVHB14; MBKB11]. These algorithms take a graph and try to create a visualization feasible for humans to understand the graph structure. Many of these algorithms layout the vertices force-based [BEGT13; FR91; JVHB14]. When looking at good visualizations of smaller graphs, humans can easily identify communities and see where a cut might be promising. To the best of our knowledge, layout algorithms have not been used for graph or hypergraph partitioning so far. However, the geographically layout of the vertices might yield good results for graph partitioning, even in dense graphs. So we propose a new partitioning algorithm called *Gravity Expansion*, which uses ideas from the graph visualization and adapts them to graph partitioning. The *openOrd* algorithm was used as inspiration to a great deal [MBKB11]. We exploited barrier jumps and the iterative workflow of force-based layout algorithms. Yet, the gravity expansion algorithm is no visualization algorithm and therefore not designed with the goals of creating a layout fit for humans, but to partition hypergraphs. Within this thesis we make the following contributions:

- three label propagation algorithms for hypergraph partitioning with different penalty functions to enforce balanced partitions
- a novel hypergraph partitioning algorithm based on graph visualization
- all our approaches are scalable with an arbitrary number of CPU cores
- a comparison of our algorithms with state-of-the-art solutions.

The thesis has the following structure: we present the background in Chapter 2. Related work is discussed in Chapter 3. Chapter 4 defines the problem and the metrics used formally. Afterwards we present the partitioning algorithms in Chapter 5. The chapter is divided in two parts. In the first part (Section 5.1), we focus on label propagation, starting with a recap of the algorithm we used as base-line for label propagation, followed by our algorithms. The second part (Section 5.2) presents our novel hypergraph partitioning algorithm based on graph visualization. The algorithms' performances are evaluated and discussed in Chapter 6. Finally, we conclude the thesis in Chapter 7 and will also illustrate the future work.

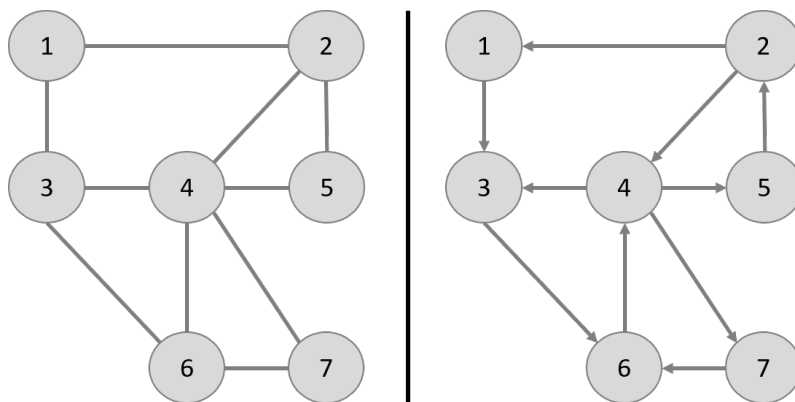


## 2 Background

Within this chapter, we explain the graph model as well as the hypergraph model. Further, we discuss what partitioning means, what a graph layout is and how these can be created, using force-based algorithms. Finally, we speak about graph processing frameworks and introduce the partitioning framework we build for this thesis.

### 2.1 Graph And Hypergraph Models

This section presents the fundamentals of graphs and hypergraphs. The graph model is a generic data structure for somehow associated data points. This can be people which are friends with other people or also geographic points which are connected via roads, to name just two examples. In the graph model we call the data points *vertices* or sometimes *nodes*. Within this thesis we will stick to the term *vertices*. The associations between the vertices are called *edges* or *links*. Every edge links exactly two vertices. Vertices can be distinguished by their id, edges either by an id or by the ids of the vertices it connects. A graph  $g$  might be denoted as  $G(V, E)$  with  $V$  being a set of vertices and  $E \in V \times V$  being a set of edges. The graph model distinguishes between *directed* and *undirected* graphs. Undirected graphs do not care about the order of an edges vertices. An edge  $e_1 = v_1 \times v_2$  would be the same as  $e_2 = v_2 \times v_1$ , hence the condition  $e_1 = e_2$  holds for undirected graphs. In a directed graphs, each edge has a direction. This can be used for road maps to be able to model one-way roads. The edges  $e_1$  and  $e_2$  are not equal within a directed graph, because one goes from  $v_1$  to  $v_2$  and the second in the opposite direction. Figure 2.1 shows two graphs with the same vertices. However, the left graph is undirected, while the right one is directed. As it can be seen, the paths (series of adjacent edges between two vertices) connecting two vertices can vary a lot due to this distinction.



**Figure 2.1:** A undirected graph on the left side and a directed graph on the right side.

Furthermore, graphs can be *weighted*, where every edge has a weight. This can represent some sort of cost, like the distance between two geographical points or the time needed to travel. One example where weighted graphs are used for, are shortest path calculations in the context of navigation.

The number of edges a vertex is connected to is referred to as *degree*. Within a directed graph, we distinguish between an *in-degree* and an *out-degree*. In many graph structures vertices with a high degree are in some way important. This can be used for example to find important influencer in a social graph or relevant websites in a web graph. There are more metrics to determine the importance of a vertex like the page rank, closeness centrality or betweenness centrality [BP98; New18], yet we do not use them within this thesis, since the degree is the easiest metric to calculate.

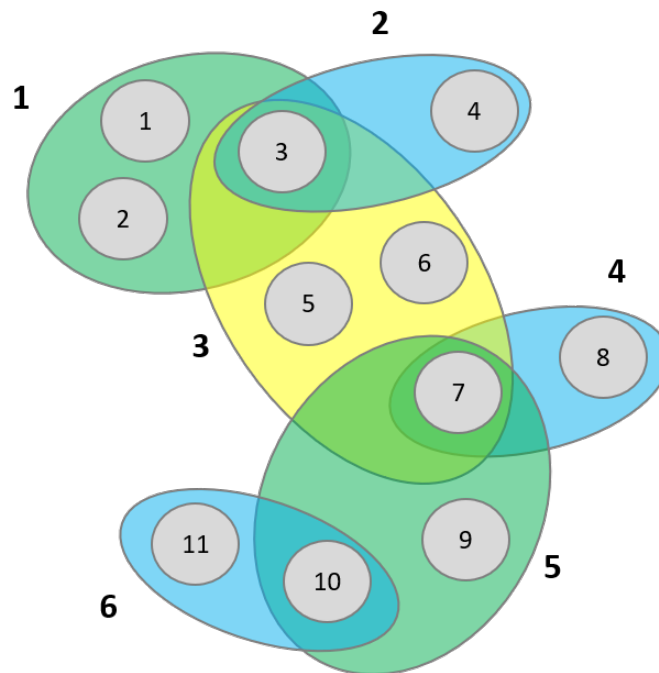
Bipartite graphs are a special form of graphs. They have two sets of vertices and the edges always connect two vertices from distinct sets. It can be defined as  $G(U, V, E)$  where  $U$  and  $V$  are sets of vertices. Thereby a single vertex  $v$  can be either in  $U$  or in  $V$  but not in both. The condition  $U \cap V = \emptyset$  holds. The edge set can be defined as  $E = U \times V$  for the undirected case. There is more to know about bipartite graphs, however this is sufficient to understand this thesis.

A problem of the graph model is the limitation of the edges to connecting only two vertices. Bipartite graphs solve the problem by introducing a second set of vertices. Hence, the vertices from the second set can be used to cluster all edges, which should be connected. This way we can model group memberships and similar things. However, a group with  $n$  members results in one vertex and  $n$  edges. For strongly connected structures with many overlapping groups, the bipartite model creates a large overhead due to the additional edges. An even more generic model than the graph model is the hypergraph model. A Hypergraph  $h$  can be defined as  $H(V, E)$  where  $V$  is a set of vertices, like in the graph model.  $E$  is a set of hyperedges, where a hyperedge  $e$  can be defined as  $e \subset V$ . This way a hyperedge can connect an arbitrary number of vertices. Figure 2.2 shows a hypergraph with 11 vertices and 6 hyperedges. Hypergraphs can also be weighted or directed, resulting in a start set and a target set of the hyperedge. However, the hypergraphs used within this thesis are considered to be unweighted and undirected, because we focus on hypergraphs from social domains, which are undirected most of the times. Directed graphs are more an issue for workflow graphs. We ignore weighted graphs for now to keep the problem simpler.

## 2.2 Graph Processing

There are several ways to process a graph as well as some preprocessing steps. The next section discusses partitioning (preprocessing) as well as visualization and label propagation (processing). Further, established graph processing frameworks will be marked out.

Graph partitioning is the act of cutting a graph structure in several subgraphs which united yield the original graph. More formally, the problem of partitioning a hypergraph into  $k$  disjoint subsets (partitions) is known as *balanced  $k$ -way hypergraph partitioning* [KK00]. Thereby the partitions need to be equally loaded and the communication between the partitions during a graph algorithm later is to be minimized. Graph partitioning is known to be a NP-hard problem [AR06]. Hence, all algorithms are just approximations. A partition is referred to as  $p$  and the set of all Partitions as  $P$ . Thereby the condition  $\{p_1 \cup p_2 \cup \dots \cup p_k\} = h$  holds. A collection of variables often used within this thesis along with an explanation of the variable is provided in Table 2.1. In regular graph partitioning, there is a distinction between two types of graph cuts. There is the edge cut, where

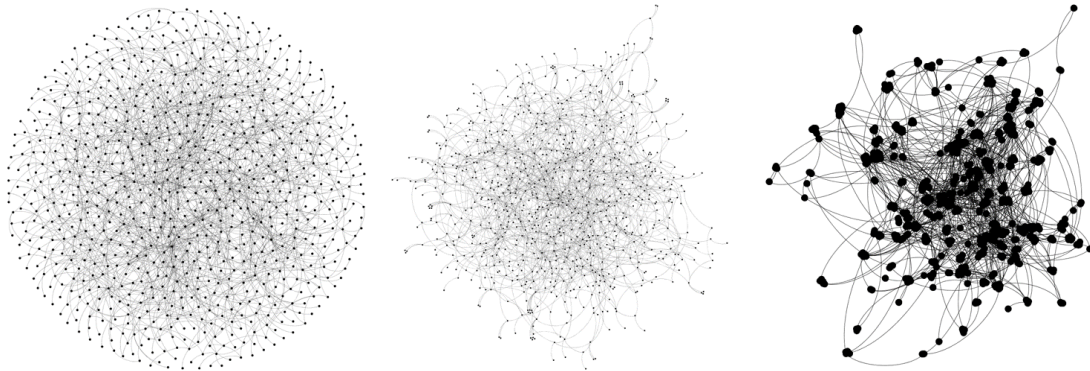


**Figure 2.2:** A visualization of a hypergraph with 11 vertices and 6 hyperedges.

edges can connect vertices which are located on two different partitions. Further, there are vertex cuts, where edges are assigned to only one partition. A vertex which has edges on several partitions is replicated on these partitions. In the edge-cut scenario, the inter-server communication arises from accessing neighbor data, when the neighbor is stored on a different machine. When using vertex cut, it arises when applying new values to a vertex which is replicated to several machines. Since we are performing hypergraph partitioning, the decision between edge-cut and vertex-cut becomes less relevant. The role of vertices and hyperedges can be swapped easily, since both are connected to an arbitrary number of the opposite element. Hence, every vertex can be interpreted as a hyperedge and every hyperedge as a vertex. As a result, every hypergraph partitioning algorithm which does not perform hybrid cuts can be used as a vertex-cut or as an edge-cut algorithm. We decided to perform edge-cuts, since they seem more natural to the human mind than the more abstract vertex-cuts. Independent of the type of cut, there are two major goals when partitioning a graph. First, the quality of the cut should be good. What type of quality can depend on the domain. However, in most cases a low cut size is required. This means the number of vertices replicated or the number of edges which are cut is to be minimized. There are several metrics about the cut size. We discuss those metrics in Chapter 4. The second goal is the balance. In most cases we want partitions of equal size or at least similar size. A little imbalance might be allowed, to gain a better quality. Sometimes we also want partitions of different sizes, if we have machines with unequal computational power. In this case the partition sizes should reflect the capabilities of the underlying machines. In Chapter 4 the balance is formally defined and different balancing metrics are discussed.

Within this thesis we are not working on graph visualization. However, we are using several features from graph visualization algorithms and therefore introduce the basics of the field. Graph visualization deals with the problem of finding a layout for graphs humans can interpret and take as

$h$	a hypergraph
$V$	set of vertices in $h$
$v$	a vertex in $V$
$E$	a set of hyperedges in $h$
$e$	a hyperedge in $E$
$P$	partitions of $h$
$p$	a partition in $P$ , $p \subset V$
$ p $	the number of vertices on partition $p$
$p.adjacentEdges$	set of hyperedges connected to at least one vertex in $p$
$n$	the number of partitions ( $ P $ )
$v.edges$	the hyperedges connected to vertex $v$
$v.degree$	the number of hyperedges connected to vertex $v$
$e.vertices$	the vertices adjacent to hyperedge $e$
$e.cardinality$	the number of vertices adjacent to hyperedge $e$

**Table 2.1:** Variables used in this thesis**Figure 2.3:** The moreno-crime graph visualized using different layout algorithms; *Fruchterman-Reingold* on the left, *forceAtlas2* in the center and *openOrd* on the right.

much correct information out of the visualization as possible. Figure 2.3 shows the moreno-crime graph [16a] three times using the *Fruchterman-Reingold* algorithm [FR91] on the left, *force-atlas2* [JVHB14] in the center and *openOrd* [MBKB11] on the right. Different algorithms attach importance on different features, like uniform vertex distance and ordering, showing all vertices or maintaining the a structure arising from the connectivity. The choice of the layout depends on the task ahead.

Many layout algorithms use a force-based paradigm [BEGT13; FR91; JVHB14], where every vertex wields an attraction force towards the vertices it is connected to and a repulsive force towards the rest of the vertices. This way connected vertices are drawn together and especially cliques are placed in immediate neighborhood. The forces can further be modified if the edges are weighted or depending on the vertex importance. Force-based algorithms calculate the forces from and on all vertices first. When all calculations are done, the vertices positions are updated accordingly. This is repeated iteratively until it converges or a limit is reached.

There are much more algorithms concerning graphs and hypergraphs. Since (some) programmers need to work on them often, there are several frameworks for graph processing like *graphX*<sup>1</sup>, *giraph*<sup>2</sup> or *graphLab*<sup>3</sup>. They are all established frameworks designed to process regular graphs. Considering hypergraphs, there are some scientific projects like *hyperX* [JQY+18] and *mesh* [HHS+19]. These frameworks can load (hyper)graphs and perform algorithms on them. A common programming model in the context of (hyper)graphs is the *vertex centric programming model*, also known as *think-like-a-vertex* model [MWM15]. Thereby, problems are solved from a vertex point of view. Each vertex can have a value (data in arbitrary form), which is updated iteratively based on its neighbors values. Global super steps keep the vertices in sync, since the next update may not be performed until all vertices have completed their previous update. It is also possible to not perform an update on a vertex in an iteration, if no neighbor has changed. This way programs can improve their runtime by denoting the vertices as *active*, if they need to perform an update in the next super step. Thanks to the *think-like-a-vertex* model, it is quite easy to write parallel programs for graph algorithms, making it easy to reach high concurrency, which is required to handle massive graphs.

An example for a graph algorithm is the label propagation algorithm. Every vertex in the graph can have a label. The vertices update their label in every super step according to their neighbors. More precisely, each vertex adopts the label appearing most within its neighbors. However, the basic label propagation can be extended using edge weights or other fitness functions, to modify the label adoption. Classification [SSUB11] and prediction tasks [ZWHS15] are some examples, where label propagation can be used.

## 2.3 Massive Hypergraph Partitioning Framework

Within the scope of this thesis, we created a small hypergraph partitioning framework. In this section we will explain how it works, what it is capable of and where its limits are. The partitioning framework is available on GitHub at <https://github.com/gepperho/HypergraphPartitioner>.

We wrote it in Java, since it is platform independent, can be debugged easily and most importantly, it provides a very comfortable way of writing parallel programs using its stream feature and lambdas (introduced in Java 8). Figure 2.4 shows an overview of our partitioning framework including the tasks done in every step of the pipeline. In the beginning, the command-line arguments are parsed and the graph file is read. We assume to have the graph already loaded, when we start partitioning. Streaming algorithms can be simulated by iterating the vertices or edges only once, but the graph is still loaded beforehand. Hence, they still need the RAM to hold the graph instead of building it while partitioning. In the partitioning step, the partitioning algorithm specified via the command line is called. The partitioning can be done in parallel, if the partitioning algorithm is implemented accordingly. Currently, we support the following partitioning algorithms:

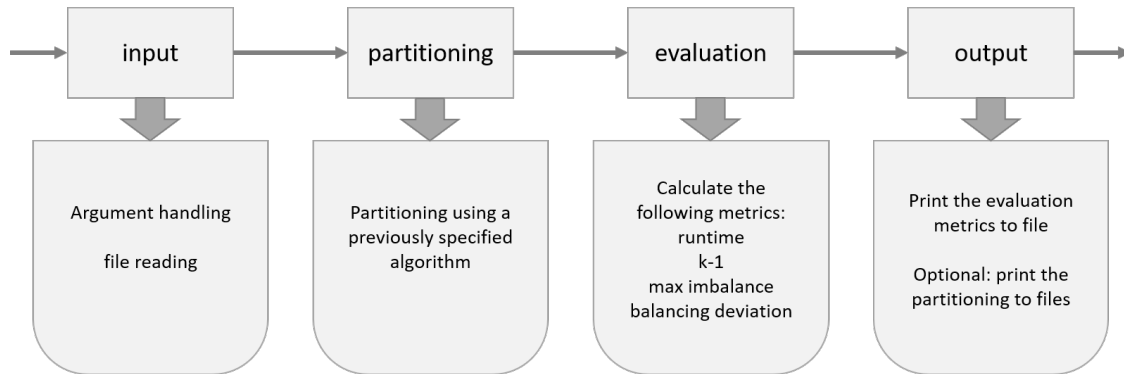
- Min-Max NB - [AIV15; MMB+18]
- HYPE - [MMB+18]

---

<sup>1</sup><https://spark.apache.org/graphx/>

<sup>2</sup><https://giraph.apache.org/>

<sup>3</sup><https://turi.com/>



**Figure 2.4:** Architecture overview of our partitioning framework

- connected component - places each connected component on a separate partition (we used this to create the reduced graphs mentioned in Section 6.2).
- Label Propagation Partitioning - [JQY+18], Section 5.1.1
- Credit Label Propagation - Section 5.1.2
- Split Credit Label Propagation - Section 5.1.3
- Split Credit Label Propagation - Credit Value Compensation - Section 5.1.4
- Gravity Expansion - Section 5.2.2

The evaluation step takes the resulting partitioning and calculates several metrics (partitioning time, k-1 cut, maximal imbalance and balancing deviation). The k-1 cut is a metric concerning the quality of the cut, while the maximal imbalance considers whether the partitions are equally loaded. In contrast to the maximal imbalance, the balancing deviation considers all partitions and not only the largest and the smallest. However, the balancing deviation does not tell much about the total imbalance, since it is some kind of average value. We will discuss the metrics in detail in Chapter 4. Finally, the metric results are printed on the command line and into a file. If specified, the partitioning results are also written into files, where every file represents a partition and holds the vertex ids of the vertices it stores.

We did stick to the object oriented programming as much as possible, to create a framework, which is easy to understand and maintain. Hence, we used an object based model structure. We have an object for the *hypergraph*, holding many *hyperedge* and *vertex* objects. The vertex objects have a *vertexData* and a *nextVertexData* field to allow iterative algorithms. Both fields are of the type *object* and have to be casted, when used in a partitioning strategy. During the partitioning, these objects are also stored (referenced) in *partitioning* objects, which inherit from the *hypergraph* object. This way, each partition, can be partitioned again easily, when using bi-partitioning algorithms.

To extend the framework with own partitioning strategies, one has to create a new class for the partitioning algorithm implementing the *PartitioningStrategy* interface. The interface provides the method signatures for the partitioning method as well as some methods yielding the strategy name and parameters used for the programs output. The partition method has the hypergraph and the number of partitions as parameters and returns a list of partitions. How the partitioning within is



performed is totally up to the strategy. In most strategies we used vertex streams yielded by the hypergraph and defined our vertex programs on them. To use it, the new strategy also has to be added into the *PartitioningStrategyFactory*.

Summarized, we built a general purpose hypergraph partitioning framework. Due to the decision to stick to OOP and use Java as programming language, we enhanced a great understandability, maintainability and extensibility, since the object oriented paradigm is well-known to most programmers as well as the Java programming language. These decisions make sense, because we aimed to create a proof of concept for our approaches and make the knowledge available, rather than creating a high performance tool nearly no one understands besides us. As a result of Java, the framework's performance is not very good, as the Java Runtime Environment is not the place for high performance code. Yet, we think the benefits of our framework justify the decision. In future work, we can build specific, high-performance software for a single algorithm. Because we can drop the general purpose there, it should be possible to write better suited code in that case.



## 3 Related Work

A lot of work has been done in graph partitioning already, as well as in hypergraph partitioning. *hMetis* [KK00] is one of the most successful hypergraph partitioning frameworks and still used as benchmark in current publications. It is a multi-level partitioning framework applying several coarsening steps and partitioning the coarse grain hypergraph afterwards. During the uncoarsening the partitioning is further optimized. However, *hMetis* has a pretty long runtime and is single threaded. The *zoltan* toolkit [CA99] provides a parallel multi-level hypergraph partitioning algorithm. However, there is no easy-to-use version of their hypergraph partitioning algorithm, as in *hMetis*<sup>1</sup>. That is why we did not use the *zoltan* partitioning tool as a benchmark.

A more recent algorithm is the *Social Hash Partitioner* [KKP+17] developed by Facebook employees. It is a scalable partitioning framework which can partition even very large graphs. The *Social Hash Partitioner* uses initial hashing and performs load shedding/optimizing afterwards. Hence, it is very scalable, since the actual optimization is performed as a distributed algorithm. They introduce a *probabilistic fanout* to optimize the number of servers requested to answer a query. Further, the partitioning runtime can be limited easily by defining a maximal number of optimization iterations. A related algorithm for regular graph partitioning is *Q-Graph* [MMG+18]. Thereby, the partitioning goal is not just a balanced partitioning with a low overall edge cut, but to minimize the number of queries cut by the partitioning. This way the runtime of a query can be minimized, so the partitioning focuses on a low latency for specific tasks instead of a general purpose partitioning.

In past years several linear runtime streaming algorithms emerged. Their runtime is way smaller than the runtime of multi-level approaches like *hMetis* or *metis* (for regular graphs) [Gep17; KK00; KK95; SK12]. However, their partitioning quality lags behind the computational complex multi-level algorithms partitioning quality. One streaming algorithm we like to highlight is the *Min-Max* algorithm from Alistarh et al. [AIV15]. A stream of vertices is partitioned by assigning each vertex to the partition it has the most hyperedges in common. To ensure a balanced output, the difference of hyperedges assigned to the partitions must not be larger than a slack parameter, which is suggested to be 100. Within the work of Mayer et al. [MMB+18] an alternative version of the *Min-Max* algorithm, called *Min-Max NB* (node balanced), is proposed. Instead of balancing the number of hyperedges associated with each partition, they balance the number of vertices (also with a slack parameter of 100). They state, the algorithm performs better in terms of the cut size metric they used in [MMB+18], which is also the metric we use in this thesis (in the original *Min-Max* Paper a different metric was used).

Most streaming approaches are only used when the partitioning has to be done quickly, when load shedding is applied anyway or the graph changes constantly. *ADWISE* [MMT+18] is a compromise between partitioning results and runtime in regular graph partitioning. The algorithm can adapt (even during runtime) to meet time constraints. Like other streaming algorithms they read the graph

---

<sup>1</sup><http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview>

and place the read edges. However, in *ADWISE* a window of edges is read and the currently best fitting gets assigned. Uninformed decisions can be delayed to favor more informed assignments. By increasing and decreasing the window size, the partitioning time can be controlled.

The idea of neighborhood expansion was recently implemented with good results in *HYPE* [MMB+18]. They took a seed vertex and grew the partition around the seed, focusing on adding neighbors from small hyperedges. This led to low cut sizes and perfectly balanced partitions. The algorithm was performing quite fast, yet it is single threaded. Hence, it does not scale well with large graphs. They also provide an easy-to-use implementation of *HYPE* on github<sup>2</sup>.

*HyperX* [JQY+18] is a general-purpose hypergraph processing framework on top of Apache Spark. It uses a label propagation algorithm called *Label Propagation Partitioning* for hypergraph partitioning. They try to gain balance in terms of the vertices as well as the edges, since their framework supports vertex programs as well as edge programs. Their partitioning algorithm is a pretty basic label propagation algorithm with an additional balancing function. While the algorithm has a low complexity and short runtimes, the balancing is not as good and the low cut size is bought with high imbalance. Another framework is *MESH* [HHS+19]. It is comparable to *HyperX* and they provide an interesting discussion about graph processing in general within their publication.

In 'How to Partition a billion-node graph' [WXS+14] they provide a multi-level label propagation algorithm for regular graph partitioning. They use the idea of weighted vertices and edges, which derive from a coarsening step. Further, they rely on *metis* [KK95] for the final partitioning. Ugander et al. designed a label propagation algorithm which creates balanced partitions [UB13]. They use the label propagation to determine which vertices should be moved to a different partition. Afterwards they create and solve a relocation linear program to finally swap several vertices. This way the algorithm is basically a load shedding algorithm, and they use a lot of global knowledge between the actual label propagation steps. This limits the parallelization and introduces a lot of overhead, especially when partitioning large graphs. Barber et al. introduced several punishment functions to keep the partitions more balanced [BC09]. But they did not use very large graphs within their original evaluations and their punishment functions were designed for community detection, not for graph partitioning. Furthermore, all label propagation algorithms mentioned so far except for the *Label Propagation Partitioning* are designed for regular graphs.

The field of (regular) graph visualization is also well researched. Fruchterman and Reingold created a graph visualization algorithm for undirected graphs [FR91]. They use a force-directed placement aiming for a uniform edge length. *OpenOrd* is a toolbox for graph layout [MBKB11]. They created a scalable, parallel algorithm with several phases, for different layout goals like expanding the graph or fixing the final vertex placement. In the initial liquid phase they try to move apart disconnected components. Afterwards the graph is spread out in the expansion phase. The *openOrd* algorithms further phases (Cool-down, Crunch and Simmer) are used to improve the result of the expansion in terms of the visualization. By using features like barrier jumps [DWB01] they have a pretty good runtime compared to other force based layout algorithms as well as decent results. A common layout algorithm is *Force Atlas 2* [JVHB14] which is well-known due to the *Gephi* tool [gep11]. They combine several techniques like degree-dependent repulsive force to an algorithm yielding good layouts. However, it was not designed for huge graphs, so it is not scalable enough for our purposes. Bannister et al. developed a layout algorithm using social gravity to draw social graphs

---

<sup>2</sup><https://github.com/mayerrn/hype>

---

[BEGT13]. They do not simply use the vertex degree to determine the vertex impact on the graph, but its centrality. This way, more meaningful metrics than the degree metric are used to determine the importance of a vertex.

Overall, the investigation showed that while there exist multiple algorithms, many of them lack parallelization and the capability to handle massive graphs while still exploiting the benefits of algorithm complexities which are above linear runtimes. The label propagation algorithms, which provide parallelization, often lack balanced results. Further, there are visualization algorithms providing ways of parallelization which were not applied to hypergraph partitioning yet.



## 4 Problem Formulation

Within this chapter, we formally define the hypergraph partitioning problem addressed in this work. Further, we present and discuss the metrics we used.

Back in Chapter 2 we defined a hypergraph  $h$  as  $H(V, E)$ .  $V$  was a set of vertices and  $E$  a set of hyperedges. A hyperedge  $e \in E$  was defined as  $e \subset V$ . A balanced  $k$ -way hypergraph partitioning was defined as a set of partitions  $P = \{p_1, p_2, \dots, p_k\}$  where  $p_1 \cup p_2 \cup \dots \cup p_k = h$ . In other words, the graph is cut in several subgraphs, which combined result in the original hypergraph.

There are several metrics to describe a partitioning in terms of several qualities. We can classify these into two groups we are interested in: cut size metrics and balancing metrics. The cut size relates to the number of hyperedges which are cut by the partitioning. When more hyperedges are cut, graph algorithms later will have more effort collecting information and its runtime increases, due to synchronization overhead. The first cut size metric we discuss is the *number of cut hyperedges*. It denotes how many hyperedges had to be cut for the partitioning without considering how often an edge has been cut (see Equation (4.1)). The *number of cut hyperedges* can be used to get an overview how many of the edges needed to be cut and therefore how much of the graphs data will be synchronized later. However, it does not reflect the total amount of data, which will be send via the network, since much of it might be send to several partitions. The *sum of external degrees* is another cut size metric defined in Equation (4.2). It is calculated by summing all hyperedges per partition, which are also connected to another partition. This way, a hyperedge, which is adjacent to multiple partitions, is counted multiple times. In contrast to the *number of cut hyperedges* the volume of the synchronization effort can be seen in the metric, yet there is no way to tell if some edges are split several times or if many edges are split just once. Finally, there is the  *$k-1$  cut*. This metric is closely related to the *sum of external degrees*, since it sums all hyperedges adjacent to a partition (note, all adjacent edges, not only the outgoing which are also connected to another partition). Equation (4.3) shows a definition of the metric. The difference of the  *$k-1$  cut* is that the total number of hyperedges is subtracted in the end. A hyperedge, which is not cut, will be added once during the summing and be removed again in the end. Thus, the metric represents only the additional overhead produced by the partitioning.

$$\text{number of cut hyperedges} := \sum_{e \in E} \begin{cases} 1 & \exists u, v \in e \mid u \neq v \wedge u \in p_i \wedge v \in p_j \wedge i \neq j \\ 0 & \text{else} \end{cases} \quad (4.1)$$

$$\text{sum of external degrees} := \sum_{p \text{ in } P} |\{e \in p.\text{adjacentEdges} \mid \exists u, v \in e \wedge u \neq v \wedge u \in p \wedge v \in p_i \wedge p \neq p_i\}| \quad (4.2)$$

$$k-1 \text{ cut} := \left( \sum_{p \in P} |p.\text{adjacentEdges}| \right) - |h.E| \quad (4.3)$$

When considering the presented metrics, the *number of cut hyperedges* is too limited, since we get no reliable statement for the total network overhead appearing later. Both, the *sum of external degrees* and the *k-1 cut* overcome this drawback. Hence, we did not further consider the *number of cut hyperedges*. The other two metrics are quite similar in the expressiveness and both are used to quantify partitioning quality [KK00; MMB+18]. Yet, we favor the *k-1 cut* metric, because it represents the introduced overhead in a more deterministic fashion. When an additional hyperedge is cut, the *sum of external degrees* is increased by 2, if the same edge is cut again, the metric is further increased by 1. Hence, the *sum of external degrees* introduces some imprecision and can not be interpreted clearly. The *k-1* metric does not have this drawback and we used it therefore.

The cut size is not the only relevant metric for the partitioning. Assigning the whole hypergraph to one partition would result in a perfect cut size, since no edge would be cut. Same holds for assigning all vertices of a connected component on the same partition. Yet, we want to cut the hypergraph in several equally sized partitions to process it in parallel. For most data sets we will have to cut some edges anyway to achieve balanced partitions. Hence, we need a balancing metric, to decide if the vertices are distributed equally across the partitions. Many algorithms assign vertices and are bound by a balancing constraint [AIV15; MMT+18]. Mayer et al. used the constraint shown in Equation (4.4) to ensure an imbalance below  $\tau$ . Their constraint can be reduced to dividing the size of the largest partition by the size of the smallest, to get one number representing the balance.

$$\forall i, j \in P, |P_i| > |P_j| : \frac{|P_j|}{|P_i|} > \tau \quad (4.4)$$

An addition to the previous reduction, is the *maximal imbalance*, we defined in Equation (4.5). The beginning is the same as before, however we subtract 1 from the result. This way, we get the difference, i.e., how much larger the largest partition is than the smallest. A value of 1 would mean it has twice the size (it is 100% larger), while a value of 2 would mean it is thrice as large (200% larger). The benefit of this metric is we get a direct answer to the question, how much more time the largest partition will probably need to compute its task compared to the smallest. This also relates to the computing time, which is lost due to bad parallelism. A drawback is that we consider only the largest and the smallest partition. All the other partitions might be nearly as loaded as the largest, resulting less idle time, but they also might be as loaded as the smallest, resulting in many machines being idle while waiting for the last one.

$$\text{maxImbalance}(P) = \frac{\max_{p \in P} |p|}{\min_{p \in P} |p|} - 1 \quad (4.5)$$

The *balancing deviation* is a simple metric to compensate the drawback of the *max imbalance* metric. It is the standard deviation of the partitions sizes and therefore calculated as shown in Equation (4.6).  $\bar{p}$  is the average partition size, which can be calculated as  $\bar{p} = \frac{|V|}{|P|}$ , where  $|P|$  is the number of partitions. The *balancing deviation* tells nothing about the actual balance or imbalance, but just the deviation of the partitions sizes. Hence, it can be consulted, to get more information about a partitioning, when we already have proper balancing metric.



---


$$\text{balancingDeviation}(P) = \sqrt{\frac{1}{|P|} \sum_{p \in P} (\bar{p} - |p|)^2} \quad (4.6)$$

Within this thesis the *maximal imbalance* is the balancing metric of our choice. Our label propagation algorithms are not compatible with the previous balancing constraint formula, since they work completely in parallel and there is no dedicated vertex assignment phase, where such a constraint could be implemented. Our other algorithm creates vertex balanced partitions without such a constraint. Hence, in all cases we can only afterwards calculate the balancing and compare it with other algorithms. Thereby the *maximal imbalance* suits our purpose quite well, since it is very susceptible to extreme values (partitions with too much as well as too few vertices). This susceptibility is needed, since we want to avoid extreme values as much as possible. If one partition is loaded less than the rest, algorithms performed on the graph later would need less time on this partition. Hence, it would have to wait until the remaining partitions are also processed.

In the worst case the partition is loaded more than the other partitions, because then all other partitions need to wait until the machine processing the largest partition is finished. This way an arbitrary number of machines would have to idle, because of one imbalanced partition. This would decrease the degree of the parallelism and waste a lot of computational time. Our framework also calculates the *balancing deviation*, to compensate the shortcomings of the *maximal imbalance*. However, we did not use it in our evaluations, due to its limited expressiveness. It is just in the framework for completeness and because other users might need it.

The problem we address in this thesis is to cut hypergraphs in several vertex balanced subgraphs. Thereby we try to minimize the *k-1 cut size* (Equation (4.3)) between the subgraphs, while also minimizing the *maximal imbalance* (Equation (4.5)).



## 5 Algorithms

This chapter proposes and explains several algorithms and approaches to solve the problems defined in Chapter 4. Thereby we also briefly introduce the algorithms our ideas were inspired from and provide the basics about the group of algorithms.

### 5.1 Label Propagation

Label propagation (LP) is a known graph algorithm in the field of graph processing. It assigns labels to vertices and updates them according to their neighbors [RAK07]. Some examples, where label propagation is used, are to solve classification [SSUB11] and prediction tasks [ZWHS15]. In this thesis we want to use the community detection ability of label propagation for cluster aware hypergraph partitioning. This way we keep vertices from strongly connected clusters on the same partition and reduce the overall cut size by cutting through sparse graph areas. Further, we use the fact that label propagation algorithms can be parallelized easily to maintain short runtimes.

#### 5.1.1 Label Propagation Partitioning

The Label Propagation Partitioning (LPP) was introduced by Juang et al. in [JQY+18]. Since our label propagation algorithms are based on their idea, we briefly discuss *LPP* in this section. *LPP* is used as the standard partitioning algorithm in the general-purpose hypergraph processing framework *hyperX*. The algorithm was designed to balance vertices as well as hyperedges, which differs a little from our defined partitioning goals. The idea is still applicable to our problem, since *LPP* assigns the edges based on their adjacent vertices. Hence, there is no edge balancing enforcement within the algorithm.

Initially, every vertex receives a random label. This way *LPP* can also handle disconnected graphs, which makes sense, since it needs to be able to process all types of hypergraphs in the *hyperX* framework. The vertex labels are later mapped to the workers (partitions). In each iteration, every vertex calculates its new label ( $L(v)$ ) via Equation (5.1). For each possible label ( $\in [1, n]$ ), the number of neighbors carrying the label are counted and multiplied by a balancing factor. This balancing factor reduces the value of large clusters, while increasing the value of small ones. The effect is gained by the exponent, which is negative for clusters larger than the average cluster size resulting in a factor smaller than 1. In small clusters the exponent is positive and the factor therefore larger than 1. The balancing factor is calculated with the Euler's number ( $e$ ) to the power of the square of the balancing difference. Thereby  $\bar{A}$  denotes the average number of vertices per cluster and  $A_i$  the number of vertices holding the label  $i$  (cluster  $i$ ). The rest of the symbols can be taken from the table in Table 2.1. Mind the difference between an edge ( $e$ ) and the Euler's number ( $e$ ).

$$L(v) = \arg \max_{i \in [1, \dots, n]} (|e \mid e \in v.edges \wedge L(e) == i| * e^{\frac{\bar{A}^2 - A_i^2}{\bar{A}^2}}) \quad (5.1)$$

### 5.1.2 Credit Label Propagation

Based on *LPP* [JQY+18], we developed the Credit Label Propagation algorithm (CLP). It is a combination of *Label Propagation Partitioning* and the credit method ([BB06]). By introducing a credit value (cv) to each labeled vertex, we try to improve the vertices label decisions and balancing of the partitions. Local clusters should be on the same partition later, since there will be much traffic between them with high probability. So, we try to pass cv, from a vertex to its neighbors based on their locality. Neighboring vertices, which are connected via small edges, tend to be good candidates for the same partition, because cutting those edges can be avoided, while most of the bigger edges are to large and interconnected with other large edges to be not partitioned. With the cv approach, a vertex considers a neighbor from a low cardinality edge more important than neighbors from a high cardinality edge. Further, we use only a small set of seeds to allow each label to grow into the graph. *LPP* initialized every vertex with a value. By using only a few seeds getting an initial value, we want to increase the locality of the labels and prevent the algorithm from growing several small clusters at different graph areas. Due to the small-world properties of the graph, local clusters should naturally get the same label assigned, as the labels grow into the graph.

So we first place several seeds throughout the graph. Afterwards we run our label propagation algorithm for several iterations. After the label propagation, the clusters are accumulated resulting in a new, smaller, weighted graph. We used a greedy algorithm to partition the new graph (min-max algorithm [AIV15] with balancing constraint [MMB+18]). Algorithm 5.1 shows a high level representation of the *CLP* algorithm.

---

#### Algorithm 5.1 High level CLP algorithm

---

```

1: Input Hypergraph  $h$ , number of partitions  $n$ 
2: initializeSeed( $n * m$ )
3: for  $i$  in range(1, 10) do
4:   for all vertices  $v$  in  $h$  do
5:      $v.vertexProgram()$ 
6:   end for
7: end for
8: accumulateClusters()
9: partitionClusters()

```

---

Initially some vertices are assigned an (initially) unique label and a credit value (CV) of 1. The label will later be used to cluster the vertices. Its credit value is a metric about how well the label fits the vertex. We use  $n * m$  seeds, where  $n$  is the number of desired partitions and  $m$  an arbitrary factor (we used 512).  $m$  is needed to get many smaller clusters which can be partitioned later. Using only  $n$  seeds did not work, since clusters can vanish during the algorithm if all their vertices are overwritten by another label. In some test runs more than 80% of the labels vanished in the process. For simplicity, the initial vertices are chosen at random. To improve the seeds, we walk a few steps after the random pick, to get seeds with a desired degree. The desired degree might depend on the

algorithm used and the problem to be solved. For the *CLP* algorithm, the degree of the seed did not have a significant impact. In Section 5.1.3 the choice of seeds is discussed further, since it had more of an impact there. To gain a seed within the desired degree range, the *move* operation can be performed several times. If the resulting seed still does not have the desired degree, repeating the algorithm with a new random seed might help. Yet, clear limits in move operations and taking new seeds need to be defined, to avoid an endless initialization phase. In our case, we used at most three steps and retried the whole procedure only twice per seed.

---

**Algorithm 5.2** Seed initialization for CLP
 

---

```

1: Input number of seeds: n
2: for n times do
3:   seed ← random_vertex()
4:   if seed.degree is to small or to high then
5:     move(seed)
6:   end if
7:   seed.setNewLabel()
8:   seed.cv ← 1
9: end for

```

---

After the seed initialization, the vertices pass their label and some *cv* to their neighbors iteratively (see Algorithm 5.3). Thereby, the credit value of a vertex is always between 0 and 1. A vertex sends its *cv* reduced based on Equation (5.2). The formula is meant to benefit small edges (with a low cardinality), since we want those to be clustered. Thereby we set  $a = 0.2$  and  $b = 60$  to ensure a curve progression, where small edges have a value close to one and as they get larger, the value moves asymptotically towards 0. Those coefficients might profit from more fine-tuning, however, we did not see a huge benefit in investing much time in it. In summary, vertex  $v$  sends *cv* to its neighbors, thereby the *cv* is reduced by a factor between 0 and 1. After several iterations, the vertex programs are stopped. We used 10 in our evaluations, since using more iterations had no significant impact on the partitioning. Due to the high interconnection within our (connected) graphs, most of the vertices are reached within two or three iterations and the label propagation would converge at some point.

$$R(e) = 1 - \frac{a * (e.cardinality - 2)^2}{a * (e.cardinality - 2)^2 + b} \quad (5.2)$$

---

**Algorithm 5.3** Vertex program for CLP
 

---

```

1: Input received labels and cv's as tuple list
2: labelMap ← new Map()
3: for all input tuples t do
4:   labelMap[t.label] +=  $\frac{t.cv}{list.size}$ 
5: end for
6: label ← arg max(labelMap)
7: cv ← max(labelMap)
8: sendToAllEdges(label, cv)

```

---

### 5.1.3 Split Credit Label Propagation

The plain Credit Label Propagation has the drawback that it just reduces the order of the graph and a partitioning algorithm is still needed afterwards. Furthermore, many clusters are lost during the label propagation, since they are completely overwritten by another label. During our evaluations, sometimes more than 80% of the clusters vanished in the process and even when setting  $m$  to 1, some labels still vanished. So, we created a new approach called Split Credit Label Propagation (SCLP), where labels can't disappear and no seed multiplier is necessary. In *CLP* the vertices reduced their cv and sent a new cv share created out of nowhere to their neighbors. *SCLP*, on the other hand, limits the vertices to sending portions of their own cv to neighbors. When a vertex changes its label in a later iteration, its old cv is sent back to the old cluster. This way labels can not vanish during the partitioning, because the overall cv per cluster stays the same and the smaller the cluster is, the more cv each member has in average. Overwriting members of small clusters becomes much harder and up to a point even impossible (a cv of 1 can never be overwritten, since sending vertices keep a part of their cv and the summarized cv of a cluster is always 1). Further, the vertices are mapped to a partition according to the label they hold after the last iteration, so we do not need another partitioning algorithm afterwards.

Algorithm 5.4 shows the high level *SCLP* algorithm. The *initializeSeed* method is the same as used by *CLP* (see Section 5.1.2). Within the *applyReceivedCv* method, all cv arrived in the iteration is added to the vertex current cv.

---

**Algorithm 5.4** High level SCLP algorithm

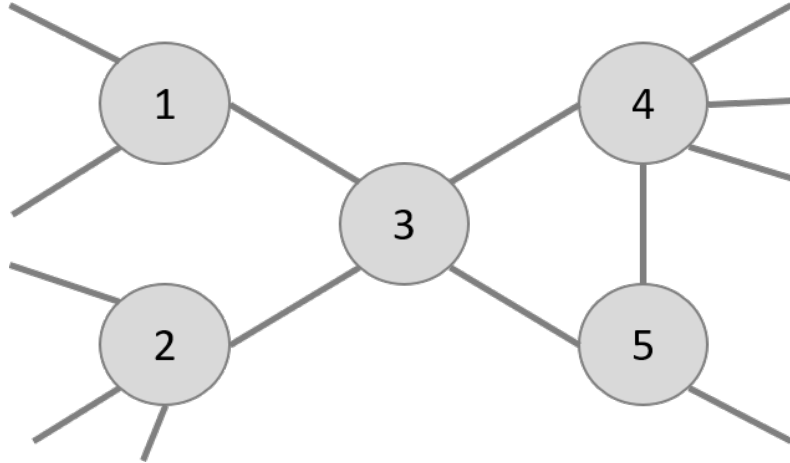
---

```
1: Input Hypergraph  $h$ , number of partitions  $n$ , number of iterations  $it$ 
2: initializeSeed( $n$ )
3: for  $i$  in range(1,  $it$ ) do
4:   for all vertices  $v \in h.vertices$  do
5:     selectLabel( $v$ )
6:   end for
7:   for all vertices  $v \in h.vertices$  do
8:     splitCreditValue( $v$ )
9:   end for
10:  for all vertices  $v \in h.vertices$  do
11:    applyReceivedCv( $v$ )
12:  end for
13: end for
14: return accumulated clusters
```

---

To ensure there is no cv not lost during partitioning, the algorithm needs some global knowledge. First, each cluster has a master vertex. Initially the seed is used, later the vertex with the highest cv. Whenever a vertex changes its label, its old cv is split and sent to all neighbors still having the old label. If there is no such neighbor anymore, the cv of the vertex is sent to the master. This way each cluster always has a summarized cv of 1 and small clusters can not be overwritten by larger clusters.

We illustrate this with an example. For simplicity, we use a regular graph (it works analog for hypergraphs). Imagine we have a graph with 5 vertices connected as seen in Figure 5.1, where vertex 1, 2 and 3 have the label A and vertex 4 and 5 have the label B. Further, we assume vertex 3



**Figure 5.1:** Example graph

will swap its label in the next iteration to have label B. Vertex 1 and 2 keep their labels. In this case, vertex 3 sends half of its old  $cv$  to vertex 1 and 2 respectively. This way its whole (old)  $cv$  is preserved within the cluster, while vertex 3 can get a new label and receive new  $cv$  from vertex 4 and 5.

Now imagine vertex 1 and 2 have the label A, 4 and 5 have the label B again and vertex 3 has a completely different label C. Again, vertex 3 will take label B in the next iteration. However, there are no neighbors with the old label around (since 1 and 2 have a different label this time). To prevent the loss of vertex 3's  $cv$  while still allowing it to switch the label, the old  $cv$  has to be sent to the clusters master (e.g. vertex 6, which is far away). Hence, the total  $cv$  of each cluster stays the same in all iterations.

The *selectLabel* method (Algorithm 5.5) serves the same purpose as in *CLP*. Since we already have some global knowledge now, we use the balancing function of *LPP* [JQY+18] (see Equation (5.3)), where we need to know the cluster sizes. For simplicity and to avoid race conditions, we use the cluster sizes after the previous iteration. Thereby  $\bar{A}$  is the average number of vertices per cluster, while  $A_p$  is the number of vertices in the cluster  $p$ . Additionally, the  $cv$  sent by a vertex is equally split between all edges and within the hyperedge again, equally split between all vertices. Like in *CLP*, half of the  $cv$  stays with the vertex. The  $cv$  calculation within the *selectLabel* method is just an approximation, since the  $cv$  is only sent to neighbors which will have the same label in the next iteration. In the approximation, we expect the  $cv$  to be sent to all neighbors. However, a vertex can receive  $cv$  from a neighbor multiple times in one iteration via several edges.

$$balancingFactor(p) = e^{\frac{\bar{A}^2 - A_p^2}{\bar{A}^2}} \quad (5.3)$$

Algorithm 5.6 shows the process of the  $cv$  being split and sent to the vertex neighbors. First, it is determined whether the vertex label has changed or not. If the label stays the same, the vertex adds half of its  $cv$  to the *cvPool* and sets the *inspectedLabel* to its next label (which is also the current label). If the label has changed, the vertex add all its  $cv$  to the *cvPool* and the *inspectedLabel* is set to the old, redeemed label (lines 7-9). Afterwards the vertex checks, if there are still neighbors labeled with the old label. In case there is none left, the  $cv$  is sent to the master of the old labels cluster and the method ends (line 10-14). In every other case, the vertex looks at its adjacent edges

**Algorithm 5.5** *selectLabel* method of the SCLP algorithm

---

```

1: Input vertex  $v$ 
2:  $labelMap \leftarrow new Map()$ 
3: for all edges  $e$  in  $v.adjacentHyperedges$  do
4:   for all vertices  $n$  in  $e.adjacentVertices \setminus v$  do
5:      $labelMap[n.label] += balancingFactor(v.label) * \frac{n.cv}{2 * n.degree * e.size}$ 
6:   end for
7: end for
8:  $v.nextLabel \leftarrow \arg \max(labelMap)$ 
9:  $updateMaster(v)$ 

```

---

for edges having label neighbors (vertices, where the next label is the same as the *inspectedLabel*). For each of those edges, an *edgeCvPool* is calculated (line 19) and each label neighbor of this edge gets the same share of the *edgeCvPool*.

**Algorithm 5.6** *splitCreditValue* method of the SCLP algorithm

---

```

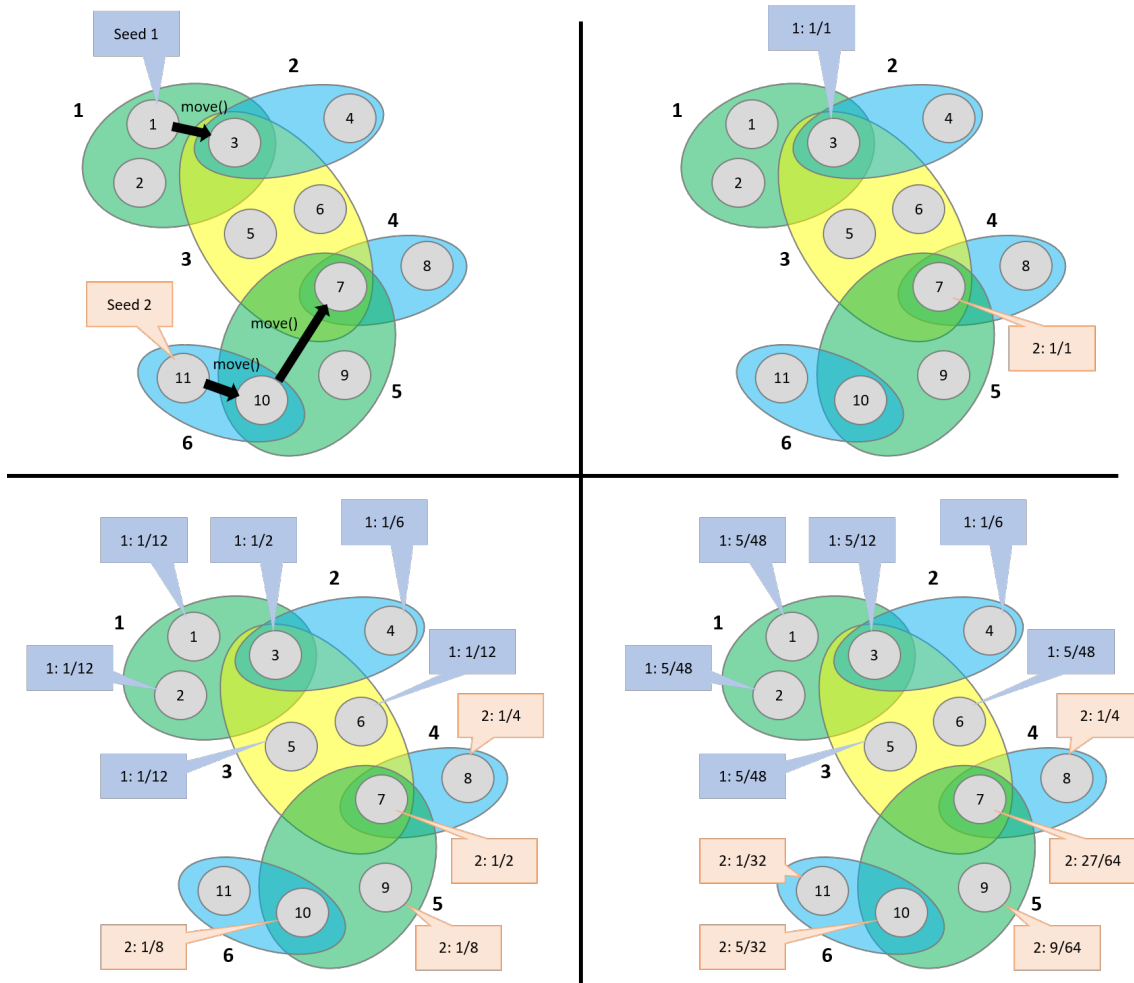
1: Input vertex  $v$ 
2: if  $v.label == v.nextLabel$  then // usual case
3:    $cvPool \leftarrow \frac{v.cv}{2}$ 
4:    $v.cv \leftarrow \frac{v.cv}{2}$ 
5:    $inspectedLabel \leftarrow v.nextLabel$ 
6: else // special cases
7:    $cvPool \leftarrow v.cv$ 
8:    $v.cv \leftarrow 0$ 
9:    $inspectedLabel \leftarrow v.label$  // use the old label to return the cv of the old label
10:  if  $|\{v_i \in e.vertices \mid v_i.nextLabel = v.nextLabel\}| == 0$  then //  $v$  has no neighbors with
    the old label
11:     $sendCvToMaster(v.label, cvPool)$  // the old label is isolated
12:    return
13:  end if
14: end if
15:  $labelNeighborEdges \leftarrow \{e \in v.adjacentEdges, \exists v' \in e \setminus \{v\} \mid v'.nextLabel = inspectedLabel\}$ 
16: for all edges  $e$  in  $labelNeighborEdges$  do
17:    $edgeCvPool = \frac{cvPool}{|labelNeighborEdges|}$ 
18:    $labelNeighbors = \{v' \in e.vertices \mid v'.nextLabel = inspectedLabel\}$ 
19:    $passedCv \leftarrow \frac{edgeCvPool}{|labelNeighbors|}$ 
20:   for all vertices  $n \in labelNeighbors$  do
21:      $sendCv(n, passedCv)$ 
22:   end for
23: end for

```

---

Figure 5.2 shows an example how the *SCLP* algorithm works on a small graph. We have a hypergraph with 11 vertices and 6 hyperedges. The vertices have a degree between 1 and 3. In the upper left picture, we see the seed initialization. Vertex 1 and 11 have been chosen randomly, both having a





**Figure 5.2:** SCLP example. Seed placement on the upper left, Starting state in the upper right, first iteration in the lower left, second iteration in the lower right

degree of 1. However, we want vertices with a high degree to be the seeds. Hence the *move* method is called. The seed is changed from vertex 1 to 3 and from 11 via 10 to vertex 7. The vertices 3 and 7 have the highest degree in this graph. In the upper right picture vertex 3 gets the label '1' assigned, shown in the blue speech bubble, while 7 gets the label '2', shown in the orange speech bubble. Both get a cv of 1.0. These colors are used later on for an easy differentiation of the current label, which is also shown in the speech bubble along with the cv. In the lower left we can see the graph after the first *SCLP* iteration. The labels have been spread to the seeds neighbors. In the last picture in the lower right, we see the graph after the second *SCLP* iteration. All vertices have a label now and most have already done at least one round of sending cv to their neighbors. When we take this iteration as the last one, we have a cut in the middle of the graph cutting just one edge. Since the graph is connected there is no better cut size to be achieved and the partitions size differs by just one vertex.

### 5.1.4 Split Credit Label Propagation – Credit Value Compensation

For further improvement of *SCLP*'s balancing, we thought of many optimizations like transferring the *cv* only via a portion of the most promising edges, creating twice as much clusters and merging the largest with the smallest, sending *cv* probabilistic depending on the cluster size and several other balancing optimizations. The optimization which seemed to have the greatest impact was the Credit Value Compensation (*SCLP-cc*), where every few iterations the *cv* of the cluster gets evenly redistributed within the cluster. Further, the total *cv* of the cluster is reduced during the rebalancing according to the cluster size. *SCLP-cc* works mostly like *SCLP*. The only difference is we add a rebalancing procedure after every 3rd iteration.

Algorithm 5.7 shows the rebalancing procedure. First we determine the largest cluster. Afterwards a *cvPool*, holding the whole *cv* of each cluster, is calculated (line 4-7). A cluster's total *cv* depends on its size compared to the largest cluster. The smaller a cluster is, the more total *cv* it gets. Additionally, we introduce a base value (we used 0.1), to avoid that the largest cluster ends up with a *cv* of zero. The final *cvPool* value of cluster *i* calculated as the Sum of the *baseValue* and the one minus the base value multiplied by the difference of the largest cluster size ( $|p_{max}|$ ) and the size of cluster *i* ( $|p_i|$ ) normalized via the largest cluster size. The equation can be taken from line 6.

The frequency of the rebalancing is crucial, since the benefits can not be exploited, if not performed, while exaggerating results in oscillating cluster sizes with a slowly growing imbalance. During our evaluations, we came to the conclusion, that rebalancing after every third iteration yields reasonable results. Further, it is important to make sure the final result is not taken from an iteration right after a rebalancing, since the cluster sizes tend to over fit in those iterations. After another iteration, the partitioning is back to a good deviation. We used 11 iterations and therefore 3 rebalancings.

---

**Algorithm 5.7** Rebalancing procedure of the *SCLP-cc* algorithm

---

```

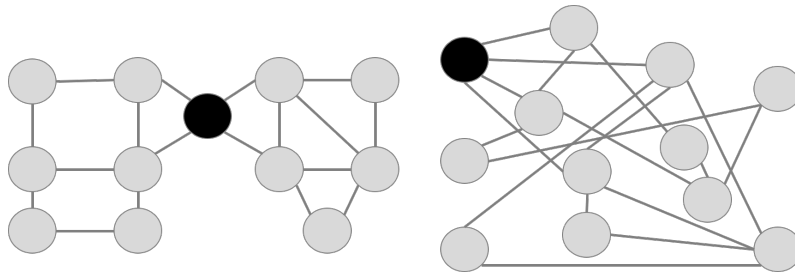
1: Input graph g, partitions P
2:  $p_{max} \leftarrow \arg \max(|P|)$ 
3:  $cvPools \leftarrow newArray[numberOfPartitions]$ 
4:  $baseValue \leftarrow 0.1$ 
5: for i in range(1, numberOfPartitions) do
6:    $cvPools[i] \leftarrow baseValue + (1 - baseValue) * \frac{|p_{max}| - |p_i|}{|p_{max}|}$ 
7: end for
8: for all vertices v  $\in$  g do
9:    $v.cv \leftarrow \frac{cvPools[v.label]}{|p_{v.label}|}$ 
10: end for

```

---

## 5.2 Graph Layout Partitioning

The idea behind graph layout/graph visualization is to draw graphs in a way, that humans can get as much information about the data as possible. Since we are familiar with two-dimensional data (e.g. pictures, texts, sketches, maps), graphs are often visualized in two dimensions. Adding a third dimension can help to visualize more of the structure (e.g. the Swiss roll data set shown in



**Figure 5.3:** The same graph twice, one time with a good graph layout and one time with a random layout

[MBKB11]), however reducing the visualization to two dimensions is easier to show on a screen or piece of paper. When visualizing graphs, it is common to place connected vertices close to each other, since they belong together somehow. Further, the graph is drawn as planar as possible, since it is much easier to process and understand for humans, if the edges do not cross. The vertex size often varies based on the vertex importance, to make it easy to pin point important vertices and to avoid getting confused with the mass of irrelevant ones.

When we are able to create visualizations preserving the graph structure, it can be quite easy to partition it. Figure 5.3 shows the same graph twice. When looking at the left graph, it is pretty easy to see that the best way to partition the graph would be cutting the black vertex. Cutting the black vertex is also the best choice for the right graph, since they are identical. However, it is hard to see, due to the chaotic graph layout. We try to make use of the simple and intuitive partitioning choice provided by good layout algorithms.

Creating good visualizations is no easy task, however there are several algorithms to create graph layouts like *Fruchterman-Reingold* [FR91], *ForceAtlas2* [JVHB14] and *OpenOrd* [MBKB11]. Further, there are layout algorithms for social graphs (which mostly follow a power law) [BEGT13]. To the best of our knowledge, such algorithms have not yet been applied to the balanced k-way partitioning problem. In this section we introduce a modified version of the *OpenOrd* algorithm for hypergraph partitioning.

### 5.2.1 Basics

The following subsection covers the fundamental mode of operation of many graph layout algorithms and a technique called barrier jump, for avoiding local minima. Both is required to some extent by our algorithm. Basic graph layout algorithms work in force-based manner [BEGT13; FR91; JVHB14]. They iteratively move the vertices into position. In the beginning they are either randomly distributed or all on the same spot. In every iteration, every vertex calculates the forces impacting it. Those forces are the attraction force from the vertex neighbors, pulling the vertex close to them and the repulsive force from all other vertices, pushing it away from all other vertices. Thereby the distance between the two vertices is taken into account. Based on the forces from all vertices in the graph, a vertex movement is calculated and performed. This process is repeated until it converges or a limit of iterations is reached. Different layout algorithms introduce or combine further features like a vertex weight [BEGT13], hierarchical force-calculation [BH86] or uniform edge lengths [FR91] to improve the results.

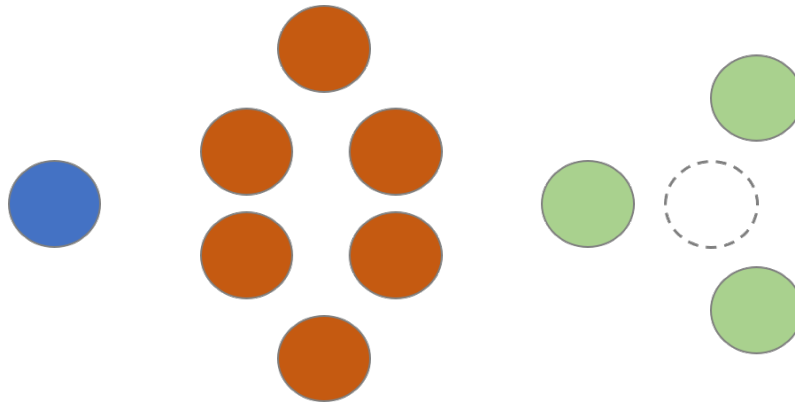


Figure 5.4: Barrier Jump example

### Barrier Jump

Since force-directed layout algorithms move the vertices small distances iteratively, they get easily stuck in a local minimum. The barrier jump was introduced to avoid this shortcoming [DWB01].

When looking at Figure 5.4, there is a blue vertex on the left side. Let's assume this vertex is connected to all green vertices on the right side, but not to the red vertices in the center. Without loss of generality, let's assume the red and green vertices do not move during the layout algorithm. We want to have connected vertices close to each other and vertices which are not linked should have some distance as described in Section 5.2.1. Naive force-directed algorithms would push the vertex as close to the red vertices as possible, due to the green vertices attraction force, but not through them, since it would be repulsed by the red vertices repulsion force. However, it is obvious that the vertex would perfectly fit in the marked spot on the right end of the graph, to be as close to its neighbors as possible. Barrier jump is a technique to move vertices across such barriers. Thereby, the centroid of the vertex neighbors is calculated by taking the average of the neighbors coordinates and the vertex is moved directly there. This way, the repulsive force of the barrier is ignored and the vertex jumps to a better position – in this case to the marked spot. Further, the barrier jump has the advantage of using only the vertex neighbors to calculate the new position instead of all vertices in the graph, which is a great benefit considering the algorithms runtime complexity. This technique is applied by the *OpenOrd* algorithm, which uses only barrier jumps and random jumps for the graph expansion, making it more scalable with larger graphs.

### 5.2.2 Gravity Expansion

The gravity expansion (GE) is a bi-partitioning algorithm, we created by modifying the *OpenOrd* visualization algorithm [MBKB11]. To create more than two partitions, the algorithm can be applied several times. *OpenOrd* has an expansion phase, where the graphs vertices are spread in a two-dimensional room while trying to keep connected vertices close to each other. These properties are also desired in hypergraph partitioning, so we used it as a baseline for our algorithm. We use *openOrds* expansion phase, to scatter the vertices in a two-dimensional space within a few iterations. Thereby, barrier jumps and random jumps are used, to get good vertex movements due to the power of two random choices (however, one is not pure random) [RMS01]. We then evaluate those two

jumps with respect to the graph density at these points. Afterwards we place two 'gravity holes', which absorb their closest vertex alternating. As soon as a specified number of vertices are absorbed (specified via parameter), another iteration of the expansion is performed, to improve the vertex placement.

---

**Algorithm 5.8** High level gravity expansion bi-partitioning algorithm

---

```

1: Input graph  $g$ , number of iterations  $it$ , reexpansion rate  $reExp$ , gridSize  $gs$ 
2:  $initializeDensityGridAndFallOff(gs)$ 
3:  $initializeVertices()$ 
4: for  $i$  in range(1,  $it$ ) do
5:    $g.expand()$ 
6: end for
7:  $gravityHoles[] \leftarrow createGravityHoles()$ 
8: for  $i$  in range(1,  $g.order$ ) do
9:    $gravityHoles[i \bmod 2].absorbClosestVertex()$ 
10:  if  $i \bmod reExp == reExp - 1$  then
11:     $g.expand()$ 
12:  end if
13: end for
14: Return  $gravityHoles$ 

```

---

Algorithm 5.8 shows the high level gravity expansion algorithm. In the beginning we create a density grid. The density grid is an accumulation of the vertex density in the two dimensional space. Each cell in the grid represents the vertex density in the according space area. Since storing the density for an endless space would be much more complex, we limit the two-dimensional space according to the density grid. Choosing good values for the grid is important, since the expansion result suffers, if it is too small or too large. The runtime and space requirements also grow with the grid size. We will show the impact of the grid size in Section 6.4.1. Further we calculate a fall-off matrix, which holds the density impact of vertices. A vertex does not only impact the density inside the cell it is placed, but also the density within a specified radius of cells. The impact decreases the further away the cell is from the actual vertex placement. Both, the density grid and the fall-off matrix arise directly from the *OpenOrd* implementation in the *Gephi* tool [gep11].

After the grid and fall-off initialization, all vertices are placed in the center of the space and the density grid is updated accordingly. As soon as all vertices are placed, an initial expansion phase starts (line 4-6). The loop can be performed in parallel. In the *expand* method, the vertex program of all vertices is called (see Algorithm 5.9). The vertex program calculates a barrier jump and a random jump and takes the better fitting one. If the vertex has no neighbors, the result of the barrier jump is its current position. For both, the barrier jump position and the random jump position, the weighted distance is calculated [MBKB11], which is the summed square distance to each neighbor multiplied by an edge weight (see Equation (5.4)). As an edge weight, we used  $\frac{1}{edgeSize}$ , which is less of an weight, but a normalization so every edge has the same impact independent of the edge size.  $d(x, y)$  denotes the distance between the coordinates  $x$  and  $y$ . In our case, between the calculated point and the neighboring vertex. For the final fitting, the weighted distance is added to the position's density, which is multiplied with a density weight (we used 2 to increase the impact of the density grid). The formula can be seen in Algorithm 5.9, line 3 and line 5. Afterwards the vertex

takes the position with the better (smaller) fitting and updates the density grid accordingly (line 6-11). Hence, the complexity of a single expansion round in a regular graph would be  $O(2 * |E|)$ , since for every vertex we have to look at every neighbor.

---

**Algorithm 5.9** Gravity Expansion vertex program
 

---

```

1: Input vertex  $v$ 
2:  $centroid \leftarrow calculateCentroid(v.neighbors)$ 
3:  $centroidFitting \leftarrow weightedDistance(v,centroid) + densityWeight * densityGrid(centroid)$ 
4:  $random \leftarrow calculateRandomJump(v)$ 
5:  $randomFitting \leftarrow weightedDistance(v,random) + densityWeight * densityGrid(random)$ 
6: if  $centroidFitting < randomFitting$  then
7:    $v.position \leftarrow centroid$ 
8: else
9:    $v.position \leftarrow random$ 
10: end if
11:  $updateDensityGrid(v)$ 

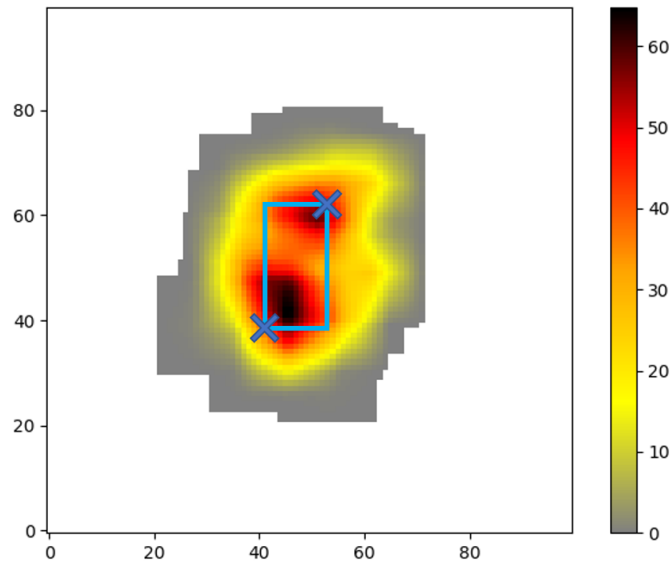
```

---

$$weightedDistance(v, pos) = \sum_{e \in v.edges} \left( \sum_{u \in e.vertices \setminus v} \frac{1}{e.size} * d(pos, u_{coordinates})^2 \right) \quad (5.4)$$

As soon as the expansion phase is finished, two gravity holes are created. Those will later absorb the vertices and thus create the bi-partitioning. For the placement of the gravity holes, we take the densest 10% of the density grids cells and draw a rectangle around it. An example can be seen in Figure 5.5. The areas with a density of zero are white, areas with a very low density gray, and as the density increases, the colors move to yellow, orange, red and finally black, for the densest parts in the graph. There are two black spots marking very dense areas. A larger one is in the bottom left of the center and a smaller one in the upper right. Those are part of the densest 10% and therefore enclosed by the rectangle. We place one gravity hole in at the corner of the minimal coordinates and one at the corner of the maximum coordinates (marked as blue crosses). When looking at Figure 5.5, a diagonal split between the two dense areas seems like a good choice. By placing the gravity holes on the rectangles corners, we ensure a split close the very dense areas while both gravity holes have many vertices around. Further we avoid absorbing ill-fitting vertices, which are pushed far away in the beginning of the partitioning. This leaves us with many good options even if we already absorbed several vertices and splits the dense areas early on avoiding the bad decisions from strategies like 'avoid big' [SK12]. Having the less connected vertices left at the end of the partitioning allows us to assign them without suffering from very poor choices, when vertices are assigned to a partition not as fitting, to ensure the balancing. By using only the dense part for the gravity hole positioning, we further avoid that one gravity hole is much further away from the whole graph than the other one.

Afterwards the vertices are absorbed one by one (Algorithm 5.8, line 8-13). Thereby, the gravity holes absorb their closest vertex alternating. The absorbed vertices are moved to the gravity hole and set as inactive (they will not be moved anymore in later expansions). The absorbing is interrupted in a given interval, specified via parameter, to perform another expansion call (line 10-12). The so

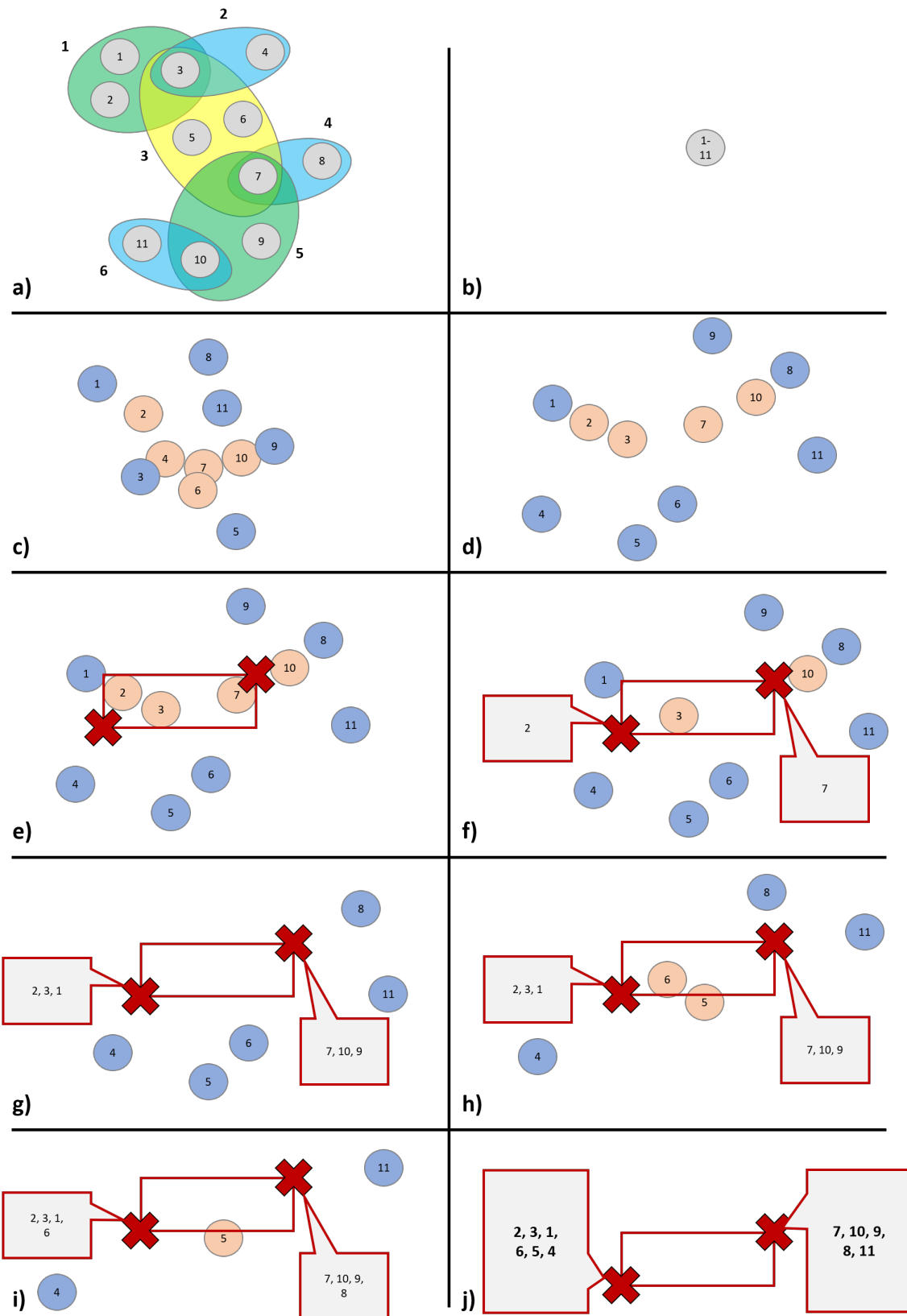


**Figure 5.5:** Gravity hole placement example on the moreno-crime graph

called reexpansion rate should not be too small, since it has a major impact on the overall runtime, especially on large graphs. Yet, the outcome benefits from a low reexpansion rate. Finding a good balance between quality and runtime is the key to this strategy. Finally, the gravity holes contents are returned as partitions and the process can be repeated on them, if more than two partitions are desired.

This procedure has the drawback, that only partitionings with  $2^n$  partitions can be created. For an arbitrary number of partitions, the balancing needs to be modified, to enforce screwed partitions, which result in balanced partitions after the last recursion step. This might be done in future work.

For better understanding, we created a small example how the *Gravity Expansion* algorithm works. Figure 5.6 shows several steps of the algorithm. In the upper left at a) the graph which is used in the example is shown. It is the same as in the example of Section 5.1.3. The first step of the *GE* algorithm is placing all vertices in the center of the space (see b). Next, the graph expansion starts. Vertices moved by a random jump are colored blue, those placed via barrier jump are colored orange for better comprehension. As it can be seen, the random jumps are expanding the graph, while the barrier jumps are in the center of the graph to keep the vertices close to their expanding neighbors. Without loss of generality, we performed the vertex functions sequentially ordered by the vertex ids. It is important to keep in mind that updates already performed in an iteration affect the decisions of the vertices handled later in the same iteration. That is why some vertices were already placed with a barrier jump in c) and they all jumped to a different location. Without these live updates, barrier jumps in the first iteration would have been jumping to the current position. After a some expansion rounds the graph could look as in d). The next step is placing the gravity holes. Since the graph is pretty small, we used more than just 10% (which would have been one vertex) for the density rectangle. The gravity holes are shown as red marks in e). Afterwards, the gravity holes start absorbing the vertices nearby in f) and g). h) shows the graph after a reexpansion step. In this step, the live updates can be seen pretty good. Vertex 5 made a barrier jump and was placed near to the lower rectangle border, since the old position of vertex 6 pulled it downwards. Afterwards



**Figure 5.6:** GE example. The graph is shown in a pretty visualization in a), the expansion, absorbing and reexpansion steps afterwards.



vertex 6 performs a barrier jump and is placed inside the rectangle, close to the lower gravity hole (where vertex 3 is placed). If both had been placed agnostic to each other they might have been placed right in the center between the two gravity holes. Due to how the beginning of the expansion went, they are now placed closer to a gravity hole, making it easier to assign them to a partition. In i) the next absorbing step is shown. Finally, j) shows the resulting partitioning.

### Optimizations

The Graph Expansion algorithm has several drawbacks. First, there is the long runtime due to several initial iterations with calculations on all vertices neighbors ( $O(2|E|)$  for each iteration) and the same calculations on all vertices (which are still active) and their neighbors during the reexpansion. Second, the memory usage may also not be ignored. Especially in our java implementation, objects are created for each edge and each vertex, where the vertex holds an additional object for its vertex data (due to a generic implementation). Last, the algorithm is randomized and the outcome is therefore nondeterministic.

In this subsection we will discuss some optimizations for the Gravity Expansion algorithm. The first is *modified random jump*, where we adapt the random jump length, to ensure a certain minimal length, to avoid very small steps. *Weighted centroid* is the name of the second optimization. Thereby, we try to improve the barrier jumps, to increase the impact of neighbors from small edges. Finally, we tried to reduce the two-dimensional space to one dimension after the initial expansion phase, to reduce the complexity and runtime.

#### *Modified Random Jump*

The length of the random jumps were only bound in terms of the maximal value, but not in terms of the minimal value. Hence, the jump could be anywhere between zero and the maximum (the so called temperature [MBKB11]). However, jumps close to zero do not have much potential for proper expanding. They are fine, if the layout is relevant, especially at the end of the layouting for fine granular vertex positioning. Yet, we are not interested in such a detailed positioning and therefore do not want such small jumps. So we changed the number range of the random value from  $[0, temperature)$  to  $[temperature/2, temperature)$ .

We also tried other modifications like adding two random values to gain a Gaussian curve or using small random values instead of large ones. However, using the long jumps had slightly better results (evaluated on a small graph).

#### *Weighted Centroid*

The barrier jump considers every edge evenly important. However, partitioning strategies like HYPE [MMB+18] try to assign vertices of small edges on the same partition, since cutting large edges is often unavoidable. Meanwhile, cutting small ones can be avoided very often. So we tried to increase the importance of small edges by increasing the influence neighbors from small edges have on the centroid.

Algorithm 5.10 shows how the weighted centroid is calculated. The algorithm takes a vertex and the current partition (graph) as inputs. The partition is needed due to an implementation detail. In our implementation of GE the vertex objects keep all their neighbors linked in the recursive bisection steps. Hence, when looping the neighbors later, we need to check if the current neighbor is actually stored on this partition or was stored elsewhere in a previous bisection. This check can be omitted, if the graph structure can be lost during the partitioning. In the beginning we define a new vector named *centroid* and a variable *normalization*. Next we loop all adjacent edges of the input vertex. For each edge we calculate an edge weight (line 5), which will later be the weight of the neighbors connected via this edge. Then we loop all adjacent vertices of the edge, skipping the input vertex and all vertices which are not stored in the current partition. In line 10 and 11, we add the edge weight to the normalization and add the neighbors position multiplied by the edge weight to the centroid. Before finally returning the calculated centroid, it has to be normalized by dividing each element of the vector by the *normalization*.

---

**Algorithm 5.10** Weighted Centroid Calculation

---

```
1: Input vertex  $v$ , partition  $p$ 
2:  $centroid \leftarrow 0,0$ 
3:  $normalization \leftarrow 0$ 
4: for all  $e \in v.edges$  do
5:    $edgeWeight \leftarrow \frac{\max |v.edges|}{|edge|}$ 
6:   for all  $v' \in e.vertices$  do
7:     if  $v' == v \ || \ v' \notin p.vertices$  then
8:       continue
9:     end if
10:     $normalization += edgeWeight$ 
11:     $centroid += edgeWeight * v'_{coordinates}$ 
12:   end for
13: end for
14: return  $\frac{1}{normalisation} * cenroid$ 
```

---

It should be noted that a neighbor linked to a vertex via multiple edges is included multiple times into the calculation. The classic centroid calculation did consider every neighbor just once. Due to this property as well as the increased computational effort, the partitioning takes longer when using the weighted centroid optimization.

### *One Dimensional Partitioning*

The *Gravity Expansion* algorithm is quite computational heavy, since the vertex centroids have to be calculated for every unassigned vertex in every expansion round. Due to the size of the input data, most of the time is spend in the absorbing and reexpanding phase. To be more precise, the calculation of the weighted distance of the random jump and the barrier jump takes a lot of time (measured with VisualVM<sup>1</sup>). So we tried to decrease the number of calculations to be performed, by not using a two-dimensional space, but a one dimensional space. However, we still want to have the ability to cluster well connected nodes in the beginning, which can be done pretty good in a

---

<sup>1</sup><https://visualvm.github.io/>

two-dimensional space, since there is enough room for each cluster to grow into. That is why we decided for this optimization to do a two-dimensional, initial expansion phase and map the vertex coordinates to a one dimensional space when the absorbing starts.

After the first initial expansion rounds, which are done on a two-dimensional grid, we calculate the placement of the gravity holes. Their placement calculation is done as before, we take the densest parts of the graph and draw a rectangle around it. In the not optimized version the gravity holes would be placed on two corners of the rectangle and the absorbing and reexpansion phase would start. Using the one dimensional optimization however, we determine on which dimension the distance between the gravity holes is longer. This is the dimension we want to keep, to make sure the gravity holes are not to close to each other. For every vertex and the gravity holes we now remove the coordinate of the dimension we do not need. The density grid and the fall-off matrices are recalculated as one dimensional vectors. Furthermore, the optimization decreases the RAM required during the absorbing and reexpanding. As soon as the mapping to one dimension has finished, the algorithm carries on as described in Section 5.2.2, however using a single number as coordinate.



## 6 Evaluations

The next section covers the empirical evaluation of the algorithms presented in Chapter 5. First, we present the server environment, benchmark algorithms, and briefly recap our implementation. Second, the data sets are introduced. Third, we compare the label propagation algorithms with each other. Fourth, the Gravity Expansion is evaluated and compared to state-of-the-art algorithms. Finally, we provide a discussion of the algorithms and their performance.

### 6.1 Environment

The evaluations have been performed using two types of servers. Small evaluations on the moreno-crime graph [16a] have been performed on the vssdn server and the other evaluations have been evaluated on the vsflash server. The server specs can be taken from Table 6.1.

	vssdn	vsflash
CPU	Intel Xeon W-2145 (8x3.7GHz)	2x AMD EPYC 7401 (96x2.0GHz)
RAM	32GB	256GB
OS	Ubuntu 18.04.3 LTS	Ubuntu 18.04.2 LTS

**Table 6.1:** Servers used for the evaluation

To compare our algorithms we used several existing algorithms as benchmark. Our label propagation algorithms were compared to the *Label Propagation Partitioning* algorithm from the *hyperX* framework [JQY+18]. It is an established label propagation algorithm able to handle large graphs. Hence, it is a fair benchmark for the label propagation algorithms, because it has the same limitations and capabilities inherited from the label propagation paradigm. *Gravity Expansion* creates balanced partitions, thus we wanted to compare it to state-of-the-art algorithms also creating balanced partitions or at least algorithms holding balancing constraints. The first benchmark is *Min-Max NB*, a modified version of the plain *Min-Max* algorithm [AIV15] which achieves better results on the *k-1* metric [MMB+18]. It is a streaming algorithm providing an upper bound for the cut size, since the algorithm is performing pretty well while being in a lower class of complexity. The second benchmark is *hMetis* [KK00], because it is a well studied algorithm which is used as benchmark in many publications. *hMetis* is an older hypergraph partitioning algorithm, but still creating very good cuts. However, it is pretty complex and will therefore be a lower bound considering the cut size. Finally, we used *HYPE* [MMB+18] as a benchmark for our results, considering it influenced our algorithms to a great extent. Further, *Hype* was shown to provide very good partitionings while running on a single thread.

We implemented our algorithms in a new general purpose graph partitioning framework, that we created for this thesis. The framework is written in Java and can be extended to support different types of partitioning algorithms. It is important to state, that our implementation was a proof of concept and therefore is a prototype implementation. The framework can be used to implement parallel partitioning strategies easily using parallel streams. We described it in detail in Section 2.3. Summarized, the framework has two major limitations. The first is our workflow and model. We assume that initially the whole graph is loaded. Afterwards the partitioning is performed on the graph, then the metrics are calculated and finally the result is returned. Hence, it can only simulate streaming partitioning algorithms. For the sake of the partitioning quality this is no problem, because streaming algorithms can simply ignore the additional information the framework provides. The model can limit the framework usage, since we assume a vertex centric programming paradigm. This paradigm can optionally be ignored. In this case every vertex has a bit of unused overhead due to some flags and pointers. The second major limitation is the framework's performance. The decision to write it in Java and to stick to OOP in favor of understandability and maintainability worsened the overall runtime of the framework. Further, the Java Runtime Environment and especially the memory handling of Java do not perform well with large amounts of data and do not provide short runtimes.

In addition to our framework we used the stand-alone implementations of *hMetis* and *Hype*. However, they are not written in Java (or a comparable language using an additional runtime environment) and they are optimized quite a lot. This leads to huge differences in the runtime just because of the implementation. To overcome this bias we also implemented *Hype* within our framework. This way we can evaluate our algorithms against the *original Hype* implementation as well as the one deriving from the algorithm, as it is described in the paper. Our *Hype* implementation uses all optimizations presented in the paper. Yet, we did not add any optimizations which might be in the *original Hype* code, since they were not documented. By doing so we try to avoid biases based on the programming language or implementation skills. When using both, our *Hype* implementation and the original one, we can try to sketch the potential of our algorithms, when implementing them in an optimized fashion.

## 6.2 Data Sets

We used different graphs for our evaluations. An overview can be seen in Table 6.2. All graphs are available online within the konect network collection [Kun13]. The moreno-crime graph was mainly used for debugging and to evaluate optimization benefits. For the evaluation of the label propagation algorithms we reduced the graphs to the largest connected component, since our label propagation algorithms can not handle disconnected graphs with the current implementation. These modified graphs are denoted with the suffix *con* for 'connected'. We use them only for the LP evaluation and not within the *Gravity Expansion* evaluation, to make it easier to reproduce our Gravity Expansion results and to stay closer to the original real world graphs.

graph	#vertices	#hyperedges	source	type
moreno-crime	829	551	[16a; Kun13]	interaction
github	56.519	120.867	[Cha09; Kun13]	collaboration
github_con	39.845	99.907	[Cha09; Kun13]	collaboration
stackoverflow	545.195	96.678	[16b; Kun13; Sta11]	rating
stackoverflow_con	524.670	80.492	[16b; Kun13; Sta11]	rating

**Table 6.2:** graphs used for the evaluation

### 6.3 Label Propagation

The next section covers the performance of the label propagation algorithms introduced in Section 5.1. Thereby we compare the *LPP* from the HyperX paper [JQY+18] with our algorithms *CLP*, *SCLP* and *SCLP-cc*. It is important to keep in mind, that we had to modify the original graphs to a reduced version only consisting out of the largest connected component, because our LP algorithms can not handle disconnected graphs. Yet this is not much of a problem, because partitioning disconnected graphs is quite easy, since the disconnected components can simply be placed on different partitions without any punishment in the cut size. It is further important to keep in mind, that we used 11 iterations on *SCLP-cc*, while the rest of the algorithms used only 10 as suggested in [JQY+18]. We will show the impact of multiple iterations using the example of *SCLP* later in this section.

In Figure 6.1, we can see a comparison of the cut sizes of the LP algorithms on the connected GitHub graph. The  $k-1$  metric we used to measure the cut size represents the number of additional partitions the hyperedges connect. It is assumed that every edge is assigned to one partition and the  $k-1$  metric increases by one for every additional partition a hyperedge connects. *LPP* has the lowest cut size, followed by *SCLP-cc*, *SCLP* and *CLP*. Yet, there is some distance between *LPP* and the rest. However, when looking at the imbalance in Figure 6.2, we see that *LPP* has the largest imbalance. Remember, an imbalance value of 1 denotes the largest partition has twice the size of the smallest, a value of 2 denotes it has thrice the size and so on. *LPP* is above 1 early on, while *SCLP-cc* *CLP* stay below 1 with 32 partitions and less, only 64 partitions are heavily imbalanced there. Hence, the good cut sizes of *LPP* is bought with by increasing the partitions imbalance. The balancing of the plain *SCLP* is also not good, however the credit value compensation optimization fixed this issue (while also improving the cut size). When finally looking at Figure 6.3, we see that *LPP*'s runtime is way smaller than those of the other algorithms. *CLP*'s runtime increases with the number of partitions, because the greedy assignment becomes harder the more partitions there are. *SCLP* and *SCLP-cc* seem to become slightly faster with increasing partitions. This might be related to decreasing synchronization times, when updating the labels masters, since the partitions are smaller and fewer vertices have to check and update the same master and therefore produce less synchronization overhead.

When looking at the LP evaluations on the stackoverflow graph, we see a slightly different picture (see Figures 6.4 and 6.5). *CLP* has a relatively bad balancing resulting in a low cut size. Meanwhile, *LPP* has an imbalance mostly comparable to *SCLP*, resulting in a cut size close to *SCLP*. The imbalance for *SCLP* on two partitions as well as the imbalance of *SCLP-cc* on two and four partitions is so low, it is not even shown in Figure 6.5. In terms of the cut size, *LPP* and *SCLP* are quite

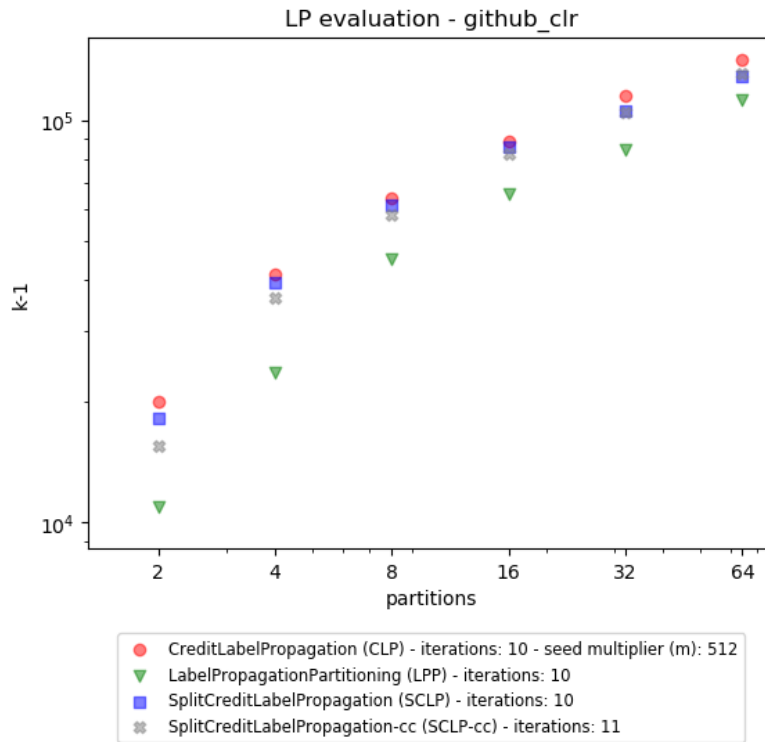


Figure 6.1: Cut size comparison of the LP algorithms on the connected GitHub graph

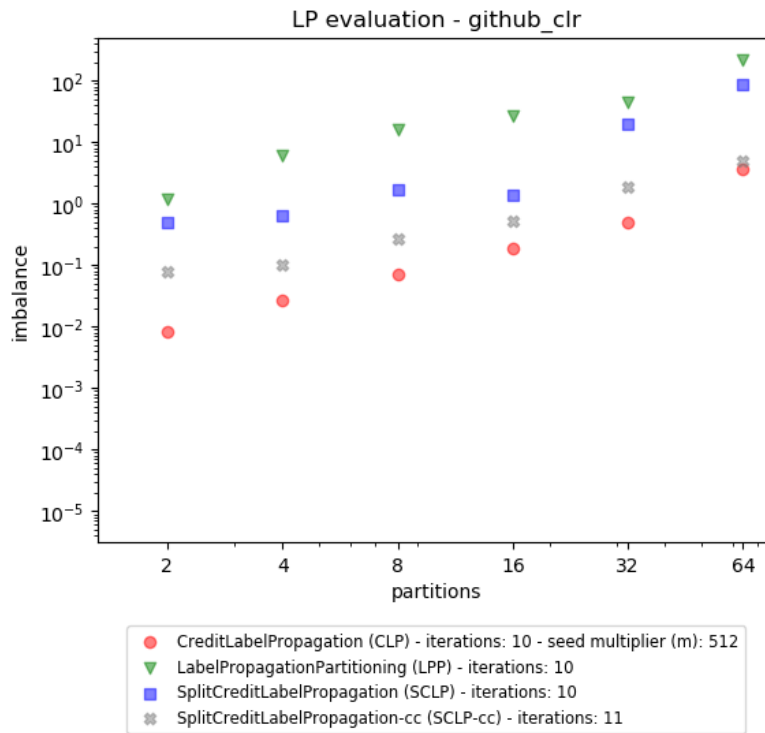
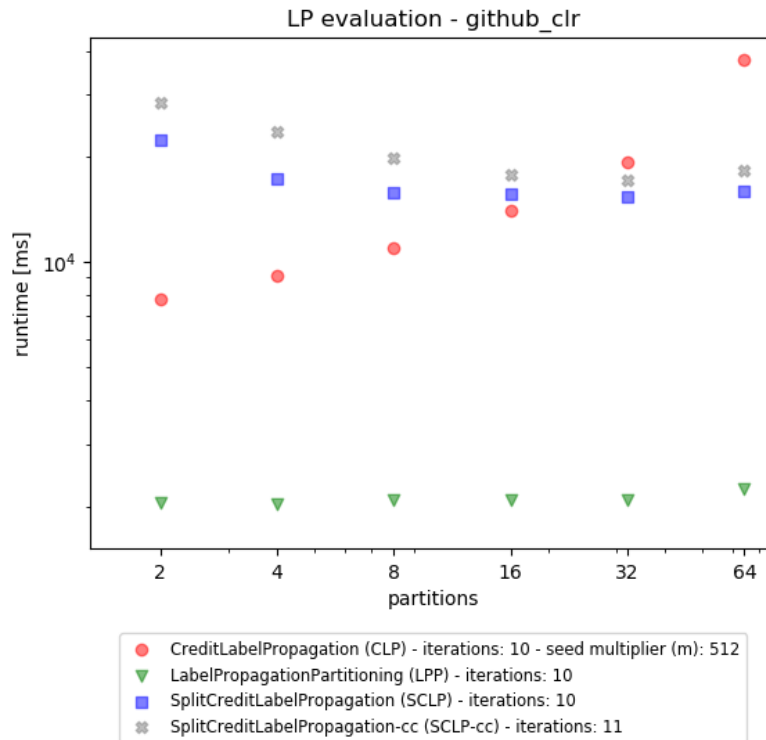


Figure 6.2: Maximal imbalance comparison of the LP algorithms on the connected GitHub graph





**Figure 6.3:** Runtime comparison of the LP algorithms on the connected GitHub graph

similar, while *SCLP-cc* is slightly better (see Figure 6.4). Sometimes *CLP* is much better due to bad balancing. The runtime differences are comparable to those from the GitHub evaluation, so we omitted the plot.

*SCLP-cc* uses a fixed number of 11 iterations (see Section 5.1.4). When comparing it to the other label propagation algorithms it has therefore an iteration more than the rest of the algorithms. To create a fair comparison in terms of the runtime, we should have also evaluated the LP algorithms using 11 iterations. However, there is not much benefit in using more than 10 as Jiang et al. stated in [JQY+18] for their *LPP* algorithm and Figure 6.6 shows for the *SCLP* algorithm. The number of iterations had no visible effect on the cut size as can be seen in Figure 6.6. The imbalance differs between the number of iterations (see Figure 6.7), however there is no pattern visible, whether a lower or a higher number of iterations benefits the balancing. Therefore, we decided to compare it in favor of the other LP algorithms runtime, since they do not rely on these eleventh iteration.

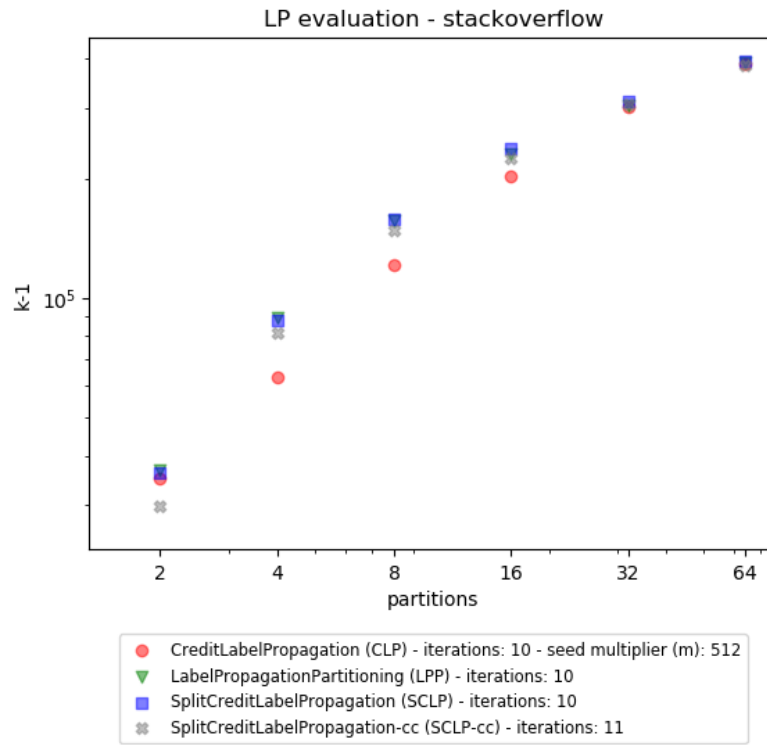


Figure 6.4: Cut size comparison of the LP algorithms on the connected stackoverflow graph

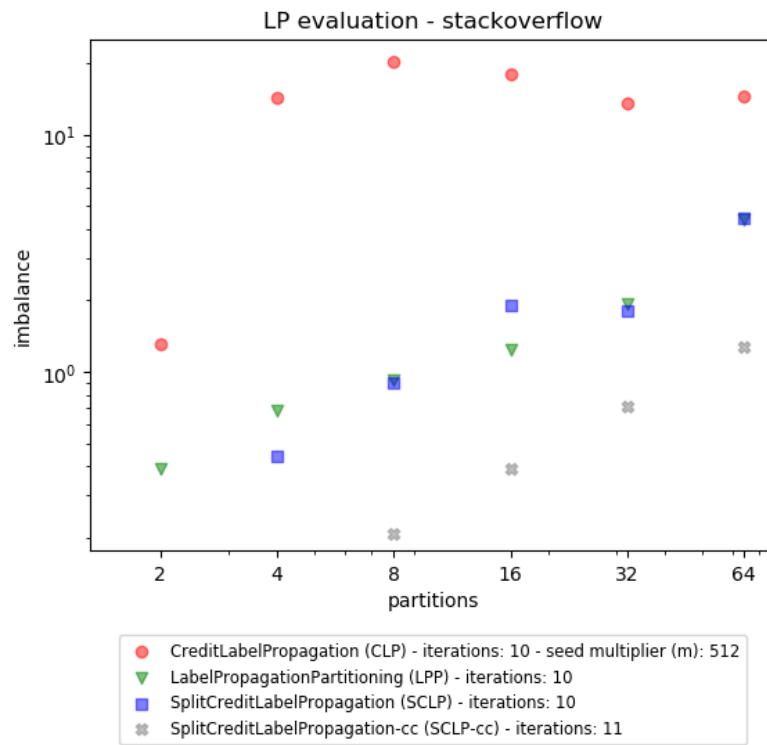
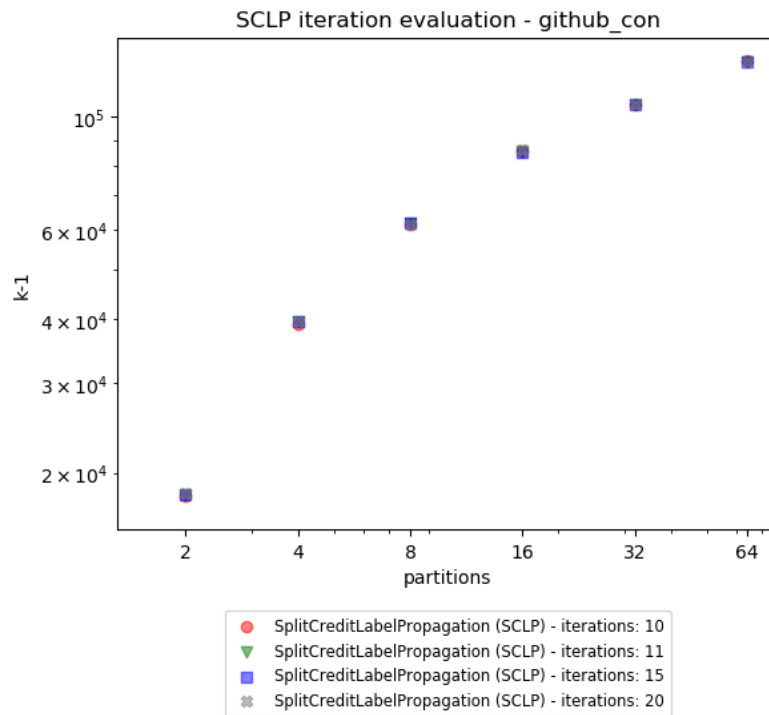
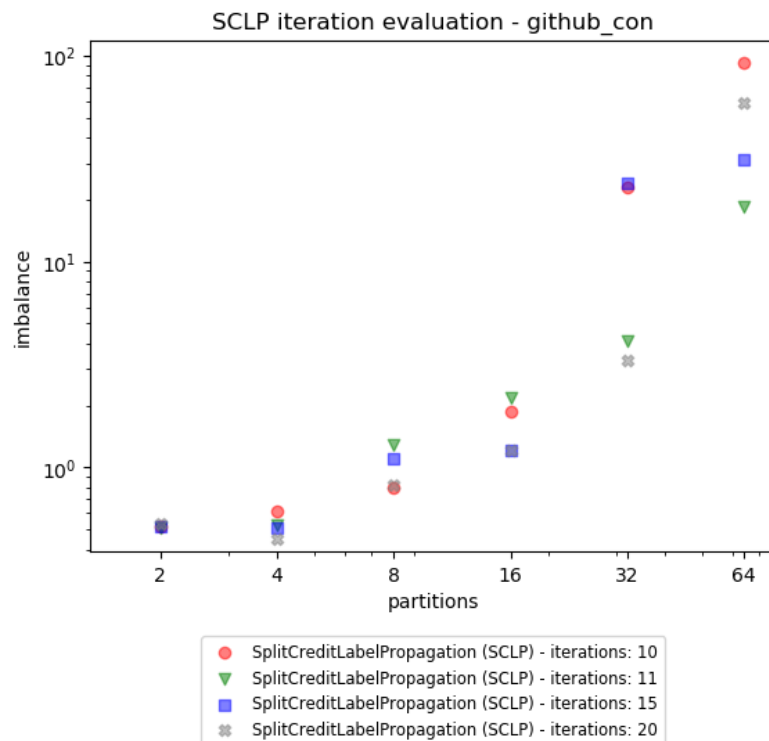


Figure 6.5: Maximal imbalance comparison of the LP algorithms on the connected stackoverflow graph



**Figure 6.6:** Cut size comparison of the *SCLP* algorithm with different iteration parameters on the connected GitHub graph



**Figure 6.7:** Maximal imbalance comparison of the *SCLP* algorithm with different iteration parameters on the connected GitHub graph

## 6.4 Gravity Expansion

This section covers the Gravity Expansion evaluation. First, we discuss the impact of the algorithms parameters. Afterwards we show the effect from the algorithm optimizations. Finally, *GE* is evaluated against other state of the art algorithms.

### 6.4.1 Parameters

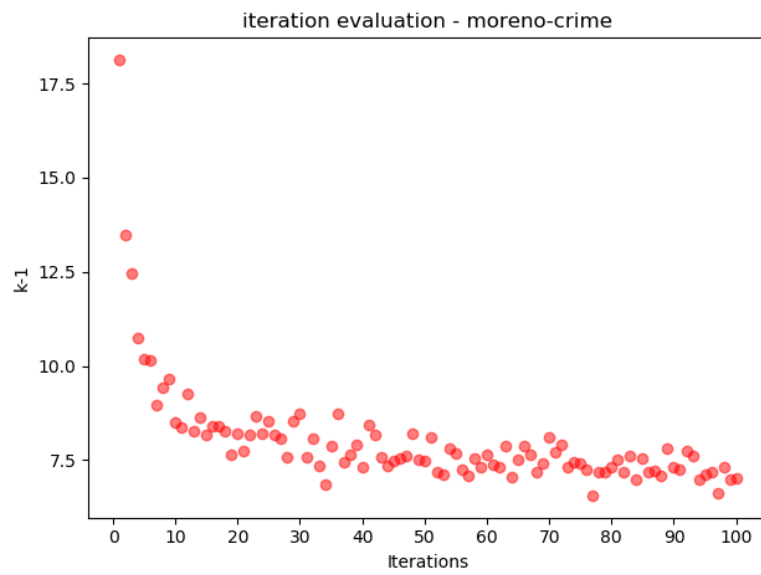
The Gravity Expansion is highly parameter dependent. Using the wrong values can result in bad cut sizes or a very long runtime. There is no one size fits all configuration for *GE*. Depending on the required properties, the algorithm can be configured to have a lower runtime, a slightly lower RAM requirement or a lower cut size, when trading for runtime. Hence, the impact of the parameters should be known when using *GE*. In this first section we use the small moreno-crime graph to show the parameters impact in a fine-grain manner. First, we show the effects of the initial iterations, afterwards from the reexpansion and finally the grid size. It is important to remember, that other graphs, especially other graph types, might be impacted to a different extent by the parameters. Especially the graphs density can have a huge impact, since the expansion runtime depends on the graphs connectivity because the vertex neighbors are accessed for the centroid calculation.

#### Iterations

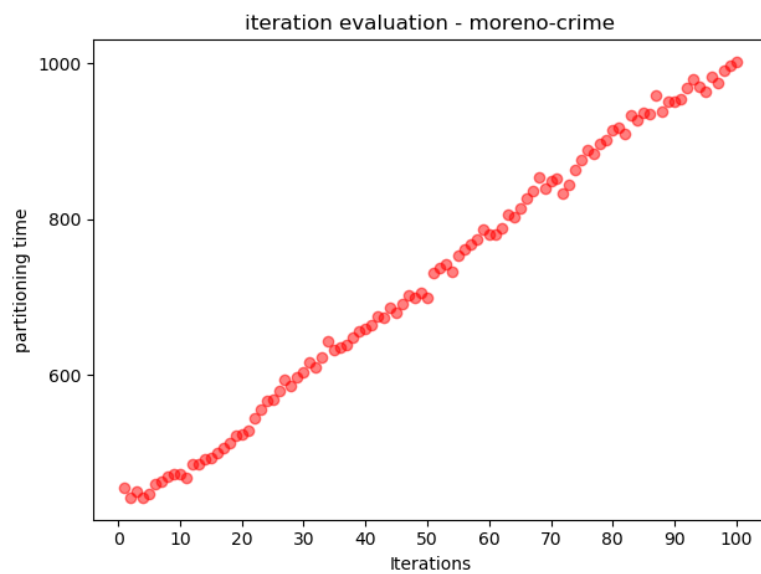
The iteration parameter determines how many expansion steps are executed before the absorbing starts. Figure 6.9 shows the runtime influence with different numbers of iterations on the moreno-crime graph. Obviously the runtime increases with the number of iterations. When assuming a regular graph, then each iteration would be in  $O(2 * |E|)$ , as every vertex needs to access all its neighbors positions. So every regular edge is followed twice. When now transferring to hypergraphs, every hyperedge is followed quadratic to its cardinality ( $e.cardinality^2$ ), because every vertex iterates the edge once for all neighbors. Thus summing the runtime to  $\sum_{e \in E} e.cardinality^2$ . However, the calculation for regular graphs is much easier to understand and imagine. When looking at the cut size in Figure 6.8, we can see the cut size decreases quickly in the beginning and behaves asymptotically afterwards. However, there is some random noise in the distribution. So we are not able to tell exactly which values are good and which not. Hence, we want to use an amount of iterations which is as small as possible, while still yielding decent results. In our evaluations we took 10 iterations for the moreno-crime graph, because the curve starts the asymptotic behavior around 10 and 25 for the other graphs, to have a little bit offset, since they are much larger than the morneo-crime graph.

#### Reexpansion Rate

During the absorbing phase, the *GE* algorithm performs further expansion steps. These are done after a number of absorbed vertices. This number is called the reexpansion rate. Since most graphs are large, the reexpansion rate can have a great influence on the runtime, because every expansion round is in  $O(2 * |E|)$  (w.l.o.g. assuming we have a regular graph), as discussed before. However, the actual runtime is reduced, since the already absorbed vertices are skipped, because they do not



**Figure 6.8:** Cut size comparison of *GE* with different initial iterations on the moreno-crime graph



**Figure 6.9:** Runtime comparison of *GE* with different initial iterations on the moreno-crime graph

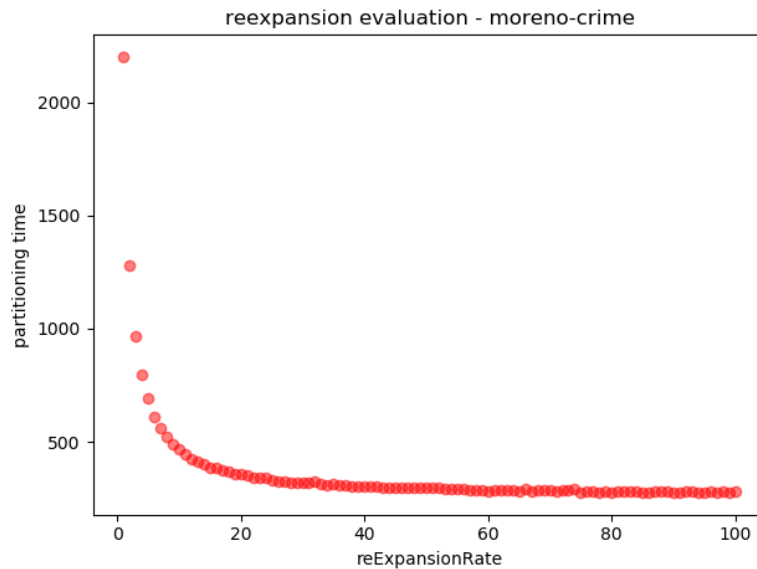


**Figure 6.10:** Cut size comparison of *GE* with different reexpansion rates on the moreno-crime graph

change their position anymore. Performing too much reexpansions results in a much longer runtime (see Figure 6.11). Nevertheless, the cut size benefits from a low reexpansion rate, as can be seen in Figure 6.10. When examining the cut size evaluation, the data points suggest some sub linear growth. However, the growth is almost linear within the investigated scope. When now looking at the runtime, we see an asymptotic behavior, which seems to shrink with an inverse quadratic function. So increasing the reexpansion rate seems not to be much of a big deal when using larger reexpansion rates. Yet, we need to find a sweet spot to gain a feasible runtime, while maintaining a good cut size. We decided to use a reexpansion rate of 10 for the very small moreno-crime graph, 100 for the GitHub graph and 10.000 for the stackoverflow graph. When using this values, the initial iterations take just a small amount of time compared to the later reexpansions. Further, there is still enough room to adapt the vertex placement during the absorbing while maintaining a feasible runtime.

### Grid Size

The grid size determines how much space there is for the graph to expand. A low grid size results in a geographically dense graph with a lot of pressure between the vertices. A high grid size on the other hand, can result in a wide spread graph, where smaller connected components can move far away from the graphs core. *GE* needs to store a density value as well as a lock for each cell of the grid. This way the grid size parameter has a direct impact on the RAM used, besides the RAM already used to store the graph itself. Yet, we need to point out that within our Java implementation the RAM needed for the graph itself was the dominant factor, due to our object oriented implementation. Figure 6.12 shows the impact of the grid size on the cut size when partitioning the moreno-crime graph. The first value with a very small grid size (25) is very bad. The next at 50 is still not perfect, but way better. If we would increase the number of samples in the low grid size area, we would most likely see more bad values when using very small grids. Hence, it is crucial for the partitioning



**Figure 6.11:** Runtime comparison of *GE* with different reexpansion rates on the moreno-crime graph

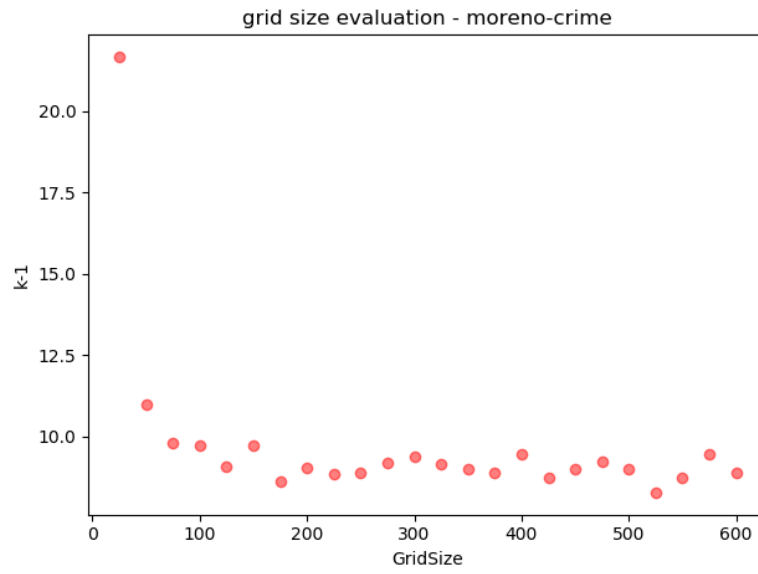
to provide a grid large enough to get a decent result. Afterwards the cut size behaves a bit blurry. We can not determine one good value just based on the cut size metric. However, after a grid size of 425, the runtime starts increasing again, suggesting a parabola behavior (see Figure 6.13). The initially decreasing runtime is a result from the reduced number of parallel write accesses on the same density grid cells. The later increasing might result from the overall memory management overhead, since quite a lot of memory is used. Summarized, we want a grid, which is large enough to yield decent results, while not being too small or too large to result in a bad runtime and high memory consumption. We did some coarse grain evaluations to determine the grid sizes to use within the thesis. So we ended up using 300 for the moreno-crime graph, 850 for the GitHub graph and 900 for the stackoverflow graph (when using a reexpansion rate of 10.000). There is some room for future work investigating good grid sizes and creating a generic formula to use based on the input graph and other inputs.

### 6.4.2 Optimizations

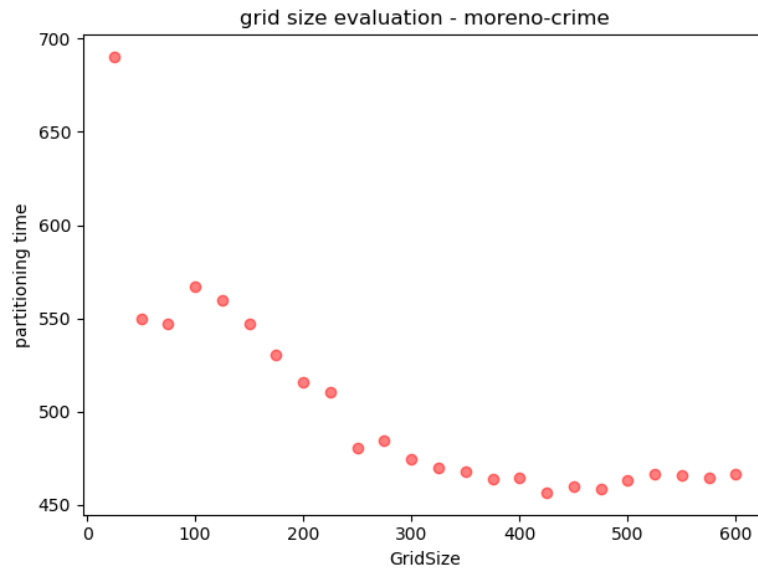
The optimizations discussed in Section 5.2.2 are subject to the next section. First we discuss the modified random jump compared with other random jump modification. Afterwards the weighted centroid and reduction to one dimension are covered.

#### Modified Random Jump

In this section, we investigate the effect of several modifications on the random jump behavior. Figures 6.14 and 6.15 show the cut size and the runtime of four different random jumps on the moreno-crime graph. First, the classic random jump as it is used within the *openOrd* implementation. Second the Gaussian random jump, where we add two random values divided by two, to increase the probability of values in the middle of the number range. Third long jumps, where the lower half

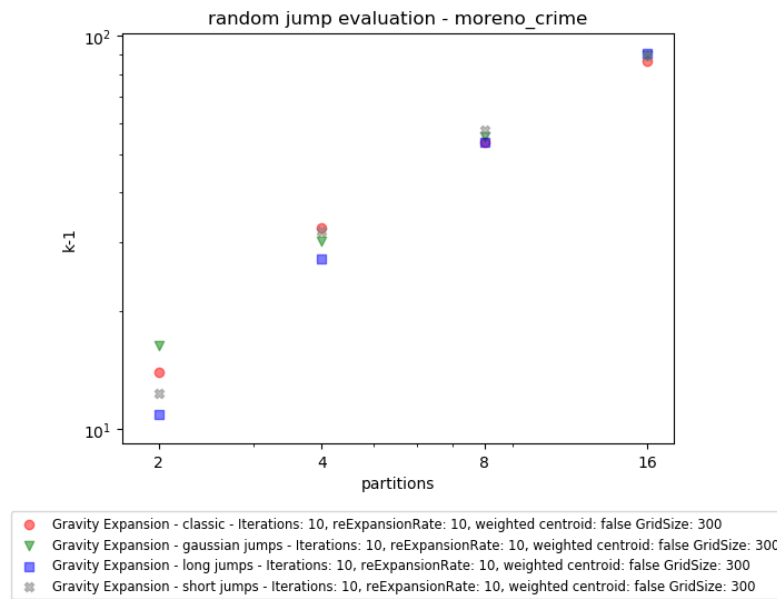


**Figure 6.12:** Cut size comparison of *GE* with different grid sizes on the moreno-crime graph



**Figure 6.13:** Runtime comparison of *GE* with different grid sizes on the moreno-crime graph





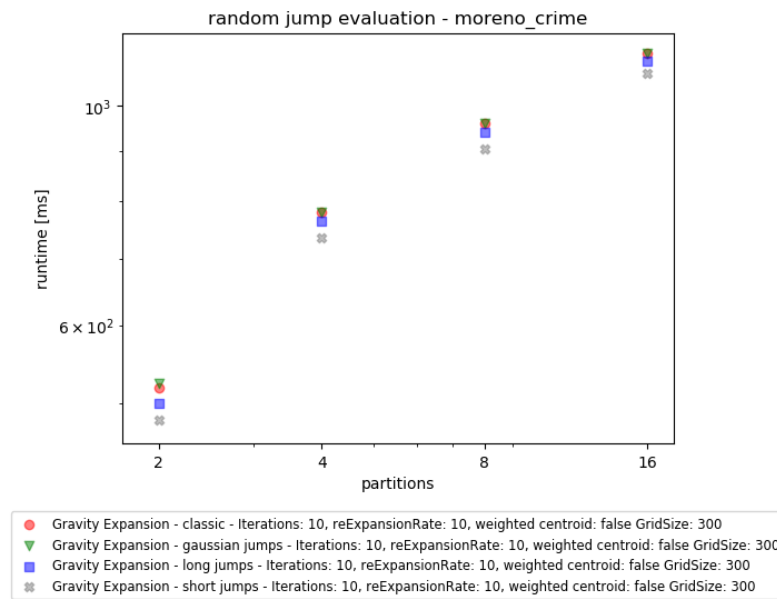
**Figure 6.14:** Cut size comparison of several random jump modifications on the moreno-crime graph

of the number range is omitted, to ensure a certain distance is traveled by the random jump. Finally, short jumps, where the upper half of the number range is omitted, which is basically a classic random jump with having a smaller maximal value. The partitions balance is equal in all cases due to the implicit balancing of the gravity expansion and therefore not shown. In Figure 6.14 we see the cut sizes when using the different random jumps. When creating several partitions (in this case 8 or more), the random jumps do not have a large impact anymore. However, the moreno-crime graph is small and cutting it into more than 4 partitions does not make much sense, since the subgraphs expanded in the last bisections are too small for reliable results. When looking at 2 or 4 partitions, the long random jumps provide the best cut sizes. When also considering the runtime in Figure 6.15, we can see that the runtime differs just a bit and the long random jumps seem to have an average runtime compared to the other techniques. Hence, we used the long random jumps for the *Gravity Expansion* algorithm.

### Weighted Centroid and GE-1d

The weighted centroid optimization (*wc*) as well as the reduction to one dimension (*1d*) are binary decisions. Hence, they can be turned on or off. Therefore, they can also be combined. That is why both their performances are evaluated within this subsection.

When looking at Figure 6.16, we can see the cut sizes on the GitHub graph. If we compare the classic *GE* with the classic *GE* using the weighted centroid, we see that the weighted centroid optimization improves the cut size. We confirm this by comparing *GE-1d* with *GE-1d* using the weighted centroid. Again, the weighted centroid optimization leads to an improved cut size. Figure 6.17 shows the runtime of these configurations. The weighted centroid configurations are not only better in terms of the cut size, they are also faster than the not optimized case. However, for *GE-1d* the difference becomes negligible with an increasing number of partitions. The difference in the runtime arises

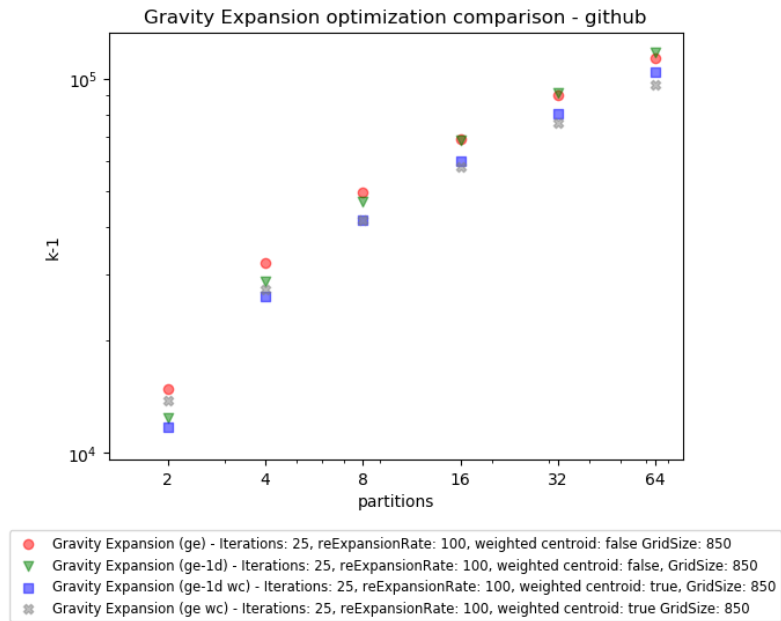


**Figure 6.15:** Runtime comparison of several random jump modifications on the moreno-crime graph

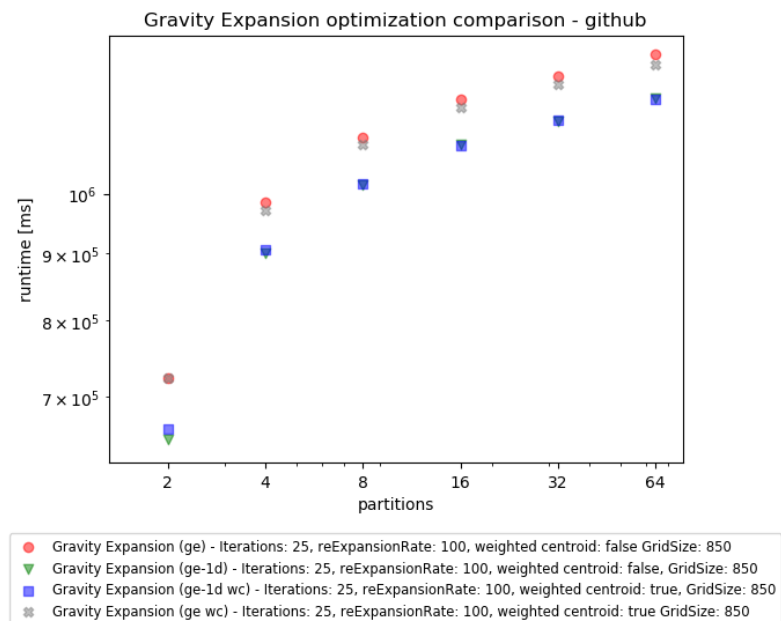
from an implementation detail, where our framework collects the vertex neighbors within the vertex function for the not optimized function. Hence, in every iteration, every vertex has to do a lookup, which of its neighbors are part of the current bisection. Using a caching mechanism would most likely result in a shorter runtime.

When now comparing the reduction to one dimension to the classic *GE*, we see that *GE-1d* benefits in terms of the runtime (see Figure 6.17). Further, it slightly improves the cut size when creating two partitions. As the number of partitions increases, the classic *GE* becomes better than *GE-1d*. Or in other words, the cut size of *GE-1d* grows faster when increasing the number of partitions, than the cut size of the classic *GE* algorithm. This observation is independent of the weighted centroid optimization. *ge-1d* handles large partitions better than small ones on the GitHub graph. Hence, one might think about mixing them in future work.

When now comparing their performance on the stackoverflow graph (see Figures 6.18 and 6.19), we see a different result. The weighted centroid optimization still has better cut sizes in both cases. Yet, the reduction to one dimension has a slightly worse cut size compared to the classic *GE*. Thereby, the one dimensional reduction with the weighted centroid is still better than the plain *GE* algorithm. When comparing the algorithms runtime, we also see some differences. The weighted centroid optimization leads to longer runtimes on the stackoverflow graph. Further, the reduction to one dimension has a longer runtime in most cases. The different results on the two data sets might be a result to their properties. The GitHub graph has about 56k vertices and about 120k hyperedges. However, when considering it as a bipartite graph, it would have about 440k edges. This results in an average degree of about 7.8 hyperedges. The stackoverflow graph on the other hand has about 545k vertices and 96k hyperedges. Yet, as a bipartite graph it would have 1300k edges resulting in an average vertex degree of about 2.4 hyperedges. Hence, the connectivity of the GitHub graph is higher. Due to the higher number of neighbors, the reduction to one dimension might benefit, since dividing the coordinates to be considered per neighbor in half has a larger impact.



**Figure 6.16:** Cut size comparison of several *GE* configurations on the GitHub graph



**Figure 6.17:** Runtime comparison of several *GE* configurations on the GitHub graph

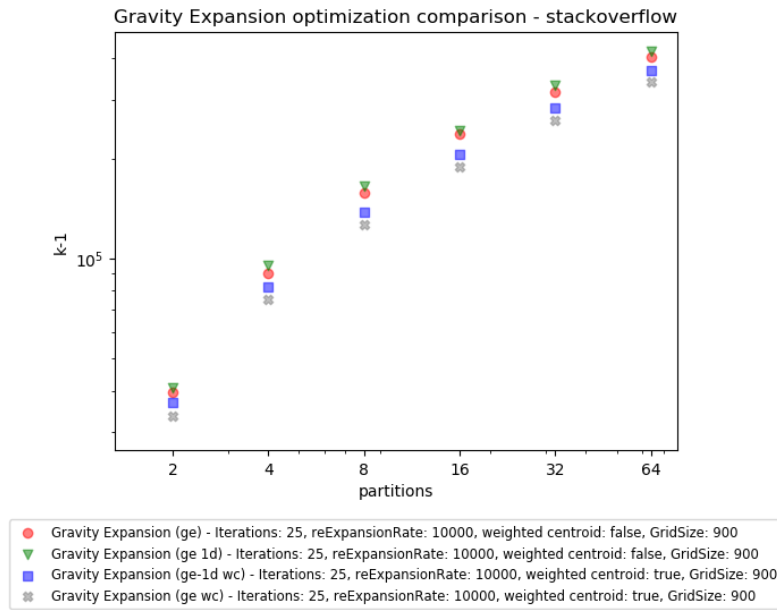


Figure 6.18: Cut size comparison of several *GE* configurations on the stackoverflow graph

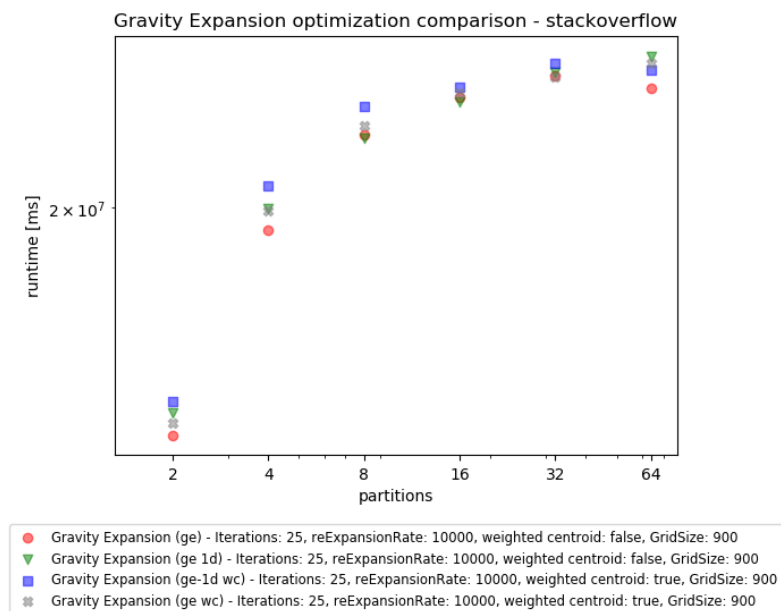


Figure 6.19: Runtime comparison of several *GE* configurations on the stackoverflow graph

### 6.4.3 Comparing GE with state of the art algorithms

In the previous subsections we evaluated the *Gravity Expansion* itself with its parameters and optimizations. This subsection covers the comparison of *GE* with state of the art partitioning algorithms. Thereby, we use *Hype* [MMB+18] (using the original as well as our own implementation), *Min-Max NB* ([AIV15; MMB+18]) and *hMetis* [KK00] as benchmarks.

Figure 6.20 shows the cut sizes on the GitHub graph. Our implementation of *Hype* has the best cut size. *hMetis* comes on the second place followed by the *original Hype* implementation and the *Gravity Expansion*. As the number of partitions increases, the cut size of all algorithms increases. However, we can see that some algorithms scale better with more partitions. In the end the *Gravity Expansion* algorithms nearly reaches the performance of our *Hype* implementation, beating *hMetis* and the *original Hype* implementation in terms of the cut size. The huge difference between the two *Hype* implementations derives from optimizations in the *original Hype* code, which are not documented in the paper. As seen already, the *one-dimensional Gravity Expansion* performs better on the GitHub graph than the *classic Gravity Expansion*. Since the *GE* algorithms and *Hype* implementations yield balanced partitions and *hMetis* uses a balancing constraint (set to 0.1, which was the minimum it could handle, resulting in values around 0.4%) as well as *Min-Max NB* (100 vertices) we omit to show a plot for the balance. The runtimes are shown in Figure 6.21. *Min-Max NB* and the *original Hype* implementation are not shown, since their runtime was much faster (300-1000 ms for *Min-Max NB* and about 800 ms for the *original hype* implementation). From the seen algorithms, *hMetis* and *GE* have a similar growth when increasing the number of partitions. However, currently *hMetis* is about one order of magnitude faster. When comparing our Java implementation of *Hype* to *GE* they start having the same runtime for the bisection. With an increasing number of partitions *Hype* becomes faster. This is a contradiction to the *Hype* paper where they state a runtime independent of the number of partitions [MMB+18]. This difference arises from our implementation, since we use hash-maps which become less efficient the more elements they hold. The more partitions we create, the smaller the hash-maps become during the neighborhood expansion. Hence, our implementation becomes faster as the number of partitions increases. The original implementation must have had a hash-map implementation which was independent of the input size.

The cut size behavior differs on the stackoverflow graph (see Figure 6.22). Initially, *hMetis* provides the best cut size. As before, it does not scale very well with an increasing number of partitions. Hence, our *Hype* implementation has the best cut size above four partitions. *GE* and *Min-Max NB* are quite close to each other. Further, they scale better than the rest of the algorithms, yet at 64 partitions they are still worse than our *Hype* implementation. The *original Hype* implementation has the worst cut size on the stackoverflow graph, which is only outnumbered by *hMetis* on 64 partitions. The balancing and runtime behavior is mostly the same as on the GitHub graph. The runtimes have just been longer due to the larger graph size so we omitted the plots. As already discussed in Section 6.4.2 the GitHub graph has a higher average vertex degree than the stackoverflow graph. Due to the differences in the graph structure, the *Gravity Expansion* and *Min-Max NB* algorithms might benefit from the decreased number of neighbors (fewer vertices to be regarded for a barrier jump or a lower number of neighbors already assigned), hence outperforming the *original Hype* implementation.

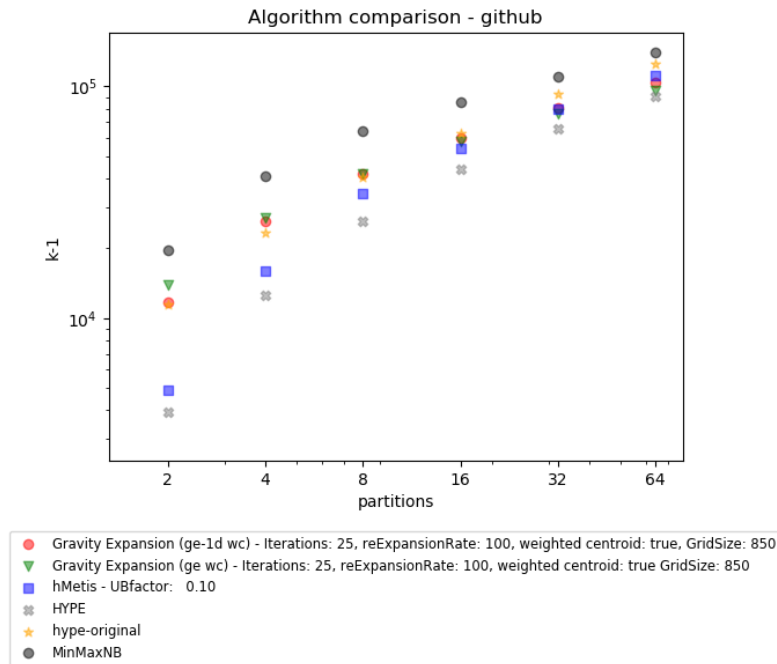


Figure 6.20: Cut size comparison of several algorithms on the GitHub graph

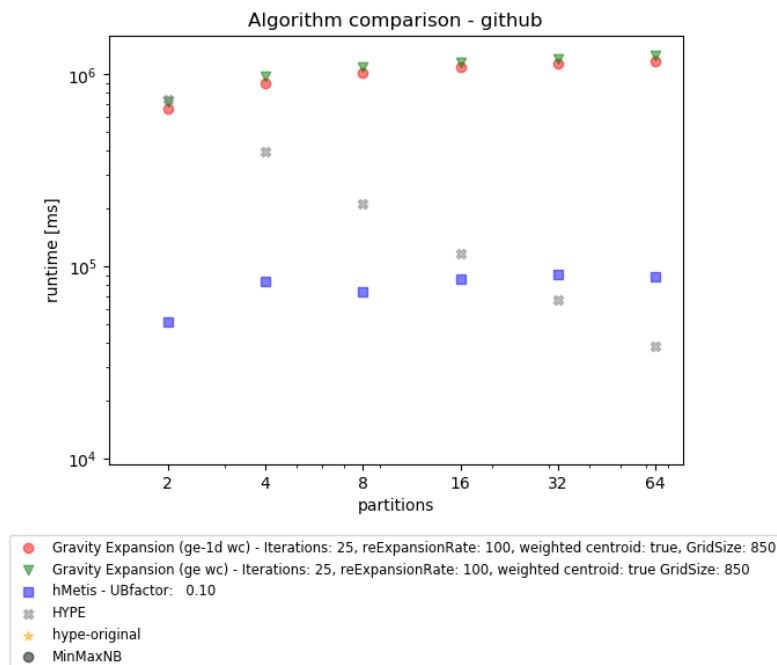


Figure 6.21: Runtime comparison of several algorithms on the GitHub graph

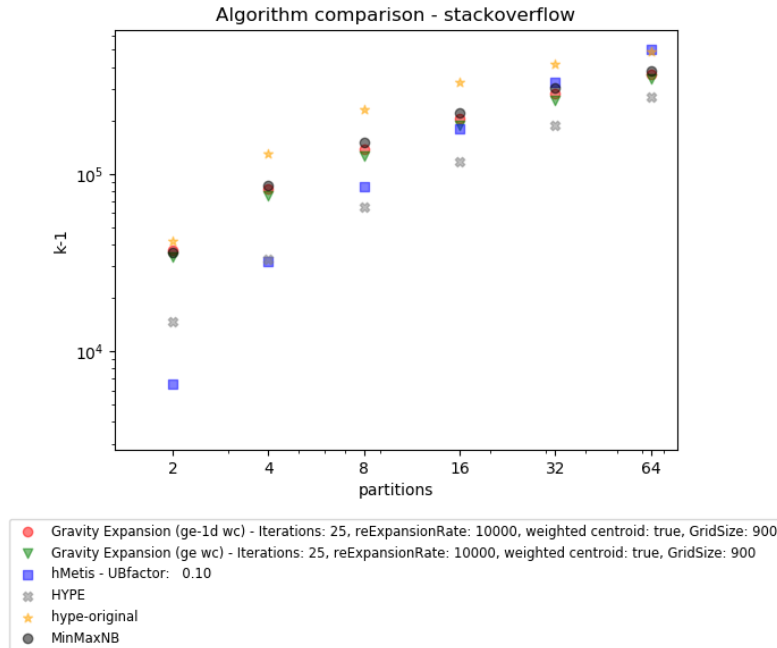


Figure 6.22: Cut size comparison of several algorithms on the stackoverflow graph

## 6.5 Discussion

In the following we discuss the algorithms performances, thereby looking closely on our approaches. We further present use-cases for the different algorithms.

Our label propagation algorithms have outperformed the *LPP* algorithm we used as a benchmark in terms of the balancing. On the stackoverflow graph we have even been able to outperform it in terms of the cut size. However, we have to keep in mind that our LP algorithms trade their improved cut size and balance for additional computation time while *LPP* sometimes has good cut sizes at the cost of the balance. Further, the balance of the LP algorithms is still not perfect, since we are not able to define a balancing constraint which is enforced. Hence, we did not compare them against balanced algorithms like *hMetis* or *Hype*. Finally, they are currently unable to partition disconnected graphs. Summarized, our LP algorithms can currently only be used in very limited situations. A fixed version of the algorithms would be fit for fast partitioning on highly parallel systems. The LP algorithms are therefore a good default partitioning choice for parallel hypergraph processing frameworks (as *LPP* is currently used). This way, users without much experience simply trying out graph processing would have a fast algorithm which would not delay the actual work afterwards much. For large scale analytics some additional partitioning overhead might be worthwhile.

Some promising results were shown by our *Gravity Expansion* algorithm when outperforming the *original Hype* implementation in terms of the cut size on the stackoverflow graph and scaling better with many partitions than *hMetis*. The degree of parallelization is theoretically very high, making it a good candidate for handling massive graphs. In practice however, the parallelization during the absorbing phase was way below the parallelization degree during the expansion. The overall runtime in the current implementation is not feasible. Yet, it is comparable to the runtime of our *Hype* implementation. The optimized *original Hype* implementation performs way faster.

Hence, there might be room for significant runtime improvements in our *GE* implementation too. Summarized, the *GE* approach needs some more effort to perform at its best. When optimized it might become a valid approach for massive graph partitioning.

Considering our benchmark algorithms, *Min-Max NB* had good results when taking the runtime into account. To perform a very fast partitioning or to add deltas to an existing partitioning it is therefore a good choice. *hMetis* had very good results when not creating too many partitions. Due to its single threaded nature combined with its complexity class it is not feasible for massive graphs. However, those had not been much of a concern at the time *hMetis* was developed. The performance of *Hype* varies. Our implementation had very good results but an awful runtime. The original implementation had a very good runtime but the partitioning result varied a lot. Yet it is currently the best choice to partition massive graphs even if it is also single threaded.



## 7 Conclusion

This last chapter concludes our findings and provides some perspectives for future work.

In this thesis we tried to find a way to partition massive hypergraphs in a balanced fashion while maintaining low cut sizes. We provided three label propagation algorithms, thereby augmenting our baseline algorithm (*LPP*) in terms of the balancing. This was done by combining the idea of label propagation with the credit method to control the label growth. The first LP algorithm we created was the *Credit Label Propagation*, which places labels on several seeds giving each of the seeds a credit value of 1. Afterwards the algorithm starts its label propagation where the vertices send their label with a reduced amount of credit value to all their neighbors. Next, every vertex adopts the label it got the most summarized credit value in this iteration. This process is repeated several times. Finally, the vertices holding the same label are clustered and this coarsened graph is partitioned using a greedy strategy. The second LP algorithm was *Split Credit Label Propagation*. Within the *SCLP* algorithm every LP iteration is divided in three phases. First every vertex decides which label it will hold in the next iteration based on its neighbors and their credit value. In the second phase, every vertex splits its own cv and sends portions of it to its neighbors holding the same label in the next iteration. The important difference to the *CLP* algorithm is that the global cv of every label stays 1 throughout the algorithm. In *CLP* new cv was created in every iteration. The third phase of the *SCLP* label propagation iteration is adding all received cv to its own pool. After several iterations, all vertices holding the same label are assigned to the same partition. The last LP algorithm was called *Split Credit Label Propagation – Credit Value Compensation*. *SCLP-cc* is an optimized version of *SCLP*. The vertices behave like in *SCLP*, however the vertex cv is periodically equalized. This means the cv of all vertices with the same label is split equally among all of them. Further, the global cv is reduced, if the associated cluster is too large. The credit value compensation helps to avoid hoarding cv in the center of the label cluster and therefore reduces the probability the rim of the cluster changes the label repetitively.

Further, we created a new bi-partitioning algorithm called *Gravity Expansion* based on graph visualization techniques. In the beginning all vertices are placed in the center of a two-dimensional space. Iteratively those vertices move to more fitting places. Thereby every vertex calculates a random jump and a barrier jump to their neighbors centroid and takes the jump which fits better. The neighbors centroid is the center of all neighbors position calculated via the means of the coordinates dimensions. The fitting of a jump is determined with a density value. The two-dimensional space is fragmented into a coarse grid. When a vertex is inside a grid cell, it increases the density of the cell and several cells around decreasingly. The less dense a cell is the more favorable it is to place a vertex there. Performing one jump for every vertex is one expansion round. When several expansion rounds are done, the absorbing phase starts. Thereby the densest 10% of the grid are determined and a mental rectangle is placed around it. Two gravity holes are placed on two diagonal corners. During the absorbing the gravity holes swallow the vertex closest to them thereby assigning them to a partition. Within the absorbing phase further expansion rounds are performed to react on the previous assignments. Already assigned vertices are not moved anymore, however their position

(which is the same as the position of the gravity hole which absorbed them) is used to determine the centroids for their neighbors which are not assigned yet. When all vertices are assigned, the bisection is done. To create more than two partitions, the process can be repeated on the subgraphs.

Considering our label propagation algorithms, we have been able to improve the balancing compared to *LPP*. We have also been able to improve the cut size on some data sets without exchanging a lower cut size for larger imbalances. However, the runtime of our algorithms were longer than the runtime of *LPP*. Further, they are currently not able to partition disconnected graphs. The *Gravity Expansion* had promising results, yet the current implementation is too slow due to several design choices in our framework. Nevertheless, *GE* was able to outperform *hMetis* when partitioning a graph into very many subgraphs and it was able to outperform the *original Hype* implementation in terms of the cut size on the stackoverflow graph. All our LP algorithms as well as the *Gravity Expansion* succeeded in terms of the scalability, since they can be scaled to an arbitrary degree.

In future work we might fix the balancing of our label propagation algorithms and enable them to handle disconnected graphs. One approach might be to place connected components and use the LP algorithms afterwards to split some connected components to gain a balanced partitioning. Concerning *GE*, we could create a new high-performance stand-alone implementation overcoming the limits of our framework. A fixed version might be fit for large scale graph analytics, if the resulting partitioning time is feasible. Further, we could create a version of *GE* which creates more than two partitions by creating more gravity holes and evaluate it against the bisection version. Additionally, it might be promising to look into creating a parallel version of *Hype*. This way we would have an additional benchmark with a well performing partitioning strategy, which should then also be scalable to handle even larger graphs. Even though our algorithms were designed to partition hypergraphs, they have no feature limiting them to this graph model. Hence, evaluating them against regular graph partitioning algorithms could show more of their potential.

## Bibliography

- [16a] *Crime network dataset – KONECT*. Sept. 2016. URL: [http://konect.uni-koblenz.de/networks/moreno\\_crime](http://konect.uni-koblenz.de/networks/moreno_crime) (cit. on pp. 22, 53, 55).
- [16b] *Stack Overflow network dataset – KONECT*. Oct. 2016. URL: <http://konect.uni-koblenz.de/networks/stackexchange-stackoverflow> (cit. on p. 55).
- [AIV15] D. Alistarh, J. Iglesias, M. Vojnovic. “Streaming Min-max Hypergraph Partitioning”. In: *Advances in Neural Information Processing Systems* 28. Ed. by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, R. Garnett. Curran Associates, Inc., 2015, pp. 1900–1908. URL: <http://papers.nips.cc/paper/5897-streaming-min-max-hypergraph-partitioning.pdf> (cit. on pp. 23, 27, 32, 36, 53, 69).
- [AR06] K. Andreev, H. Racke. “Balanced Graph Partitioning”. In: *Theory of Computing Systems* 39.6 (Nov. 2006), pp. 929–939. ISSN: 1433-0490. DOI: [10.1007/s00224-006-1350-7](https://doi.org/10.1007/s00224-006-1350-7). URL: <https://doi.org/10.1007/s00224-006-1350-7> (cit. on pp. 16, 20).
- [AS12] D. Altinbuken, E. G. Sirer. *Commodifying replicated state machines with openreplica*. Tech. rep. 2012 (cit. on p. 16).
- [BB06] M. Ben-Ari, M. Ben-Ari. *Principles of concurrent and distributed programming*. Pearson Education, 2006 (cit. on p. 36).
- [BC08] G. Buehrer, K. Chellapilla. “A scalable pattern mining approach to web graph compression with communities”. In: *Proceedings of the 2008 International Conference on Web Search and Data Mining*. ACM. 2008, pp. 95–106 (cit. on p. 15).
- [BC09] M. J. Barber, J. W. Clark. “Detecting network communities by propagating labels under constraints”. In: *Physical Review E* 80.2 (2009), p. 026129 (cit. on p. 28).
- [BEGT13] M. J. Bannister, D. Eppstein, M. T. Goodrich, L. Trott. “Force-Directed Graph Drawing Using Social Gravity and Scaling”. In: *Graph Drawing*. Ed. by W. Didimo, M. Patrignani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 414–425. ISBN: 978-3-642-36763-2 (cit. on pp. 17, 22, 29, 43).
- [BH86] J. Barnes, P. Hut. “A hierarchical O(N log N) force-calculation algorithm”. In: *nature* 324.6096 (1986), p. 446 (cit. on p. 43).
- [BP98] S. Brin, L. Page. “The anatomy of a large-scale hypertextual web search engine”. In: *Computer networks and ISDN systems* 30.1-7 (1998), pp. 107–117 (cit. on p. 20).
- [CA99] U. V. Catalyurek, C. Aykanat. “Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication”. In: *IEEE Transactions on parallel and distributed systems* 10.7 (1999), pp. 673–693 (cit. on p. 27).
- [Cha09] S. Chacon. *The 2009 GitHub Contest*. online. July 2009. URL: <https://github.com/blog/466-the-2009-github-contest> (cit. on p. 55).

- [DWB01] G. S. Davidson, B. N. Wylie, K. W. Boyack. “Cluster stability and the use of noise in interpretation of clustering.” In: *infovis*. 2001, pp. 23–30 (cit. on pp. 28, 44).
- [FR91] T. M. Fruchterman, E. M. Reingold. “Graph drawing by force-directed placement”. In: *Software: Practice and experience* 21.11 (1991), pp. 1129–1164 (cit. on pp. 17, 22, 28, 43).
- [gep11] gephi. *Gephi - The Open Graph Viz Platform*. online. 2011. URL: <https://github.com/gephi/gephi> (cit. on pp. 28, 45).
- [Gep17] H. Geppert. “Scalable hypergraph partitioning”. B.S. thesis. University of Stuttgart, 2017 (cit. on p. 27).
- [HHS+19] B. Heintz, R. Hong, S. Singh, G. Khandelwal, C. Tesdahl, A. Chandra. “Mesh: A flexible distributed hypergraph processing system”. In: *arXiv preprint arXiv:1904.00549* (2019) (cit. on pp. 23, 28).
- [JQY+18] W. Jiang, J. Qi, J. X. Yu, J. Huang, R. Zhang. “HyperX: A Scalable Hypergraph Framework”. In: *IEEE Transactions on Knowledge and Data Engineering* (2018) (cit. on pp. 16, 23, 24, 28, 35, 36, 39, 53, 55, 57).
- [JVHB14] M. Jacomy, T. Venturini, S. Heymann, M. Bastian. “ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software”. In: *PLOS ONE* 9.6 (June 2014), pp. 1–12. doi: [10.1371/journal.pone.0098679](https://doi.org/10.1371/journal.pone.0098679). URL: <https://doi.org/10.1371/journal.pone.0098679> (cit. on pp. 17, 22, 28, 43).
- [KK00] G. Karypis, V. Kumar. “Multilevel k-way hypergraph partitioning”. In: *VLSI design* 11.3 (2000), pp. 285–300 (cit. on pp. 16, 20, 27, 32, 53, 69).
- [KK95] G. Karypis, V. Kumar. “METIS—unstructured graph partitioning and sparse matrix ordering system, version 2.0”. In: (1995) (cit. on pp. 27, 28).
- [KKP+17] I. Kabiljo, B. Karrer, M. Pundir, S. Pupyrev, A. Shalita. “Social hash partitioner: a scalable distributed hypergraph partitioner”. In: *Proceedings of the VLDB Endowment* 10.11 (2017), pp. 1418–1429 (cit. on pp. 16, 27).
- [Kun13] J. Kunegis. “Konec: the koblenz network collection”. In: *Proceedings of the 22nd International Conference on World Wide Web*. ACM. 2013, pp. 1343–1350 (cit. on pp. 54, 55).
- [MBKB11] S. Martin, W. M. Brown, R. Klavans, K. W. Boyack. “OpenOrd: an open-source toolbox for large graph layout”. In: *Visualization and Data Analysis 2011*. Vol. 7868. International Society for Optics and Photonics. 2011, p. 786806 (cit. on pp. 17, 22, 28, 43–45, 49).
- [MMB+18] C. Mayer, R. Mayer, S. Bhowmik, L. Epple, K. Rothermel. “HYPE: Massive Hypergraph Partitioning with Neighborhood Expansion”. In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 458–467 (cit. on pp. 16, 23, 27, 28, 32, 36, 49, 53, 69).
- [MMG+18] C. Mayer, R. Mayer, J. Grunert, K. Rothermel, M. A. Tariq. “Q-graph: preserving query locality in multi-query graph processing”. In: *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. ACM. 2018, p. 6 (cit. on p. 27).

- [MMT+18] C. Mayer, R. Mayer, M. A. Tariq, H. Geppert, L. Laich, L. Rieger, K. Rothermel. “Adwise: Adaptive window-based streaming edge partitioning for high-speed graph processing”. In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2018, pp. 685–695 (cit. on pp. 27, 32).
- [MWM15] R. R. McCune, T. Weninger, G. Madey. “Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing”. In: *ACM Computing Surveys (CSUR)* 48.2 (2015), p. 25 (cit. on p. 23).
- [New18] M. Newman. *Networks*. Oxford university press, 2018 (cit. on pp. 15, 20).
- [RAK07] U. N. Raghavan, R. Albert, S. Kumara. “Near linear time algorithm to detect community structures in large-scale networks”. In: *Physical review E* 76.3 (2007), p. 036106 (cit. on p. 35).
- [RMS01] A. W. Richa, M. Mitzenmacher, R. Sitaraman. “The power of two random choices: A survey of techniques and results”. In: *Combinatorial Optimization* 9 (2001), pp. 255–304 (cit. on p. 44).
- [SK12] I. Stanton, G. Kliot. “Streaming graph partitioning for large distributed graphs”. In: *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2012, pp. 1222–1230 (cit. on pp. 27, 46).
- [SSUB11] M. Speriosu, N. Sudan, S. Upadhyay, J. Baldridge. “Twitter Polarity Classification with Label Propagation over Lexical Links and the Follower Graph”. In: *Proceedings of the First Workshop on Unsupervised Learning in NLP*. EMNLP ’11. Edinburgh, Scotland: Association for Computational Linguistics, 2011, pp. 53–63. ISBN: 978-1-937284-13-8. URL: <http://dl.acm.org/citation.cfm?id=2140458.2140465> (cit. on pp. 23, 35).
- [Sta11] Stack Exchange Inc. *Stack Exchange Data Explorer*. online. 2011. URL: <http://data.stackexchange.com/> (cit. on p. 55).
- [UB13] J. Ugander, L. Backstrom. “Balanced label propagation for partitioning massive graphs”. In: *Proceedings of the sixth ACM international conference on Web search and data mining*. ACM. 2013, pp. 507–516 (cit. on p. 28).
- [WXS14] L. Wang, Y. Xiao, B. Shao, H. Wang. “How to partition a billion-node graph”. In: *2014 IEEE 30th International Conference on Data Engineering*. IEEE. 2014, pp. 568–579 (cit. on p. 28).
- [ZWHS15] P. Zhang, F. Wang, J. Hu, R. Sorrentino. “Label propagation prediction of drug-drug interactions based on clinical side effects”. In: *Scientific reports* 5 (2015), p. 12339 (cit. on pp. 23, 35).

All links were last followed on October 09, 2019.



### **Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature