Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Partitioning Billionscale Hypergraphs

Lukas Epple

Course of Study: Informatik

Examiner: Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel

Supervisor: Dipl.-Inf. Christian Mayer

Commenced: November 2, 2017

Completed: May 2, 2018

Abstract

Due to growing communication networks and the increasing size of social network datasets, the need for graph processing systems has increased. Since most communication services such as Whatsapp, Facebook or Reddit provide group functionalities, modeling such communication relationships as hypergraphs is straightforward. To proccess big hypergraphs, they need to be spread accross multiple machines, which makes partitioning the hypergraphs inevitable. Known hypergraph partitioning systems either are fast and produce partitionings of bad quality or they provide partitionings of good quality, but have a poor runtime. In this thesis, we present a hypergraph partitioning algorithm that achieves both, fast partitioning with high locality. The idea is simple but effective: the algorithm grows k disjoint subgraphs based on the neighbourhood relation and the degree distribution in the hypergraph. We performed extensive experiments and showed that our algorithm leads to perfectly balanced partitions with improved locality compared to state-of-the-art, while matching the fast runtime of streaming hypergraph partitioners.

Kurzfassung

Aufgrund wachsender Kommunikationsnetzwerke und der immer größer werdenden Datenmengen in sozialen Netzwerken hat sich die Nachfrage nach Graph-Verarbeitungs-Systemen deutlich erhöht. Fast alle modernen Kommunikationsnetzwerke, wie z.B. Whatsapp, Facebook oder Reddit bieten heutzutage Gruppenfunktionalitäten an, welche sich sehr einfach mit Hilfe von Hypergraphen modellieren lassen. Um diese großen Hypergraphen verarbeiten zu können, müssen diese durch Hypergraph-Partitionierung auf viele verschiedene Maschinen verteilt werden. Solche Partitionierungs-Algorithmen existieren bereits, bieten jedoch entweder eine schnelle Laufzeit und schlechte Ergebnisse oder gute Ergebnisse und eine schlechte Laufzeit. In dieser Arbeit wird ein neuartiger Partitionierungs-Algorithmus vorgestellt, welcher beides bietet, eine schnelle Laufzeit und gute Ergebnisse. Die dem Algorithmus zugrunde liegende Idee ist einfach aber effektiv: der Algorithmus baut k disjunkte Subgraphen anhand der Nachbarschafts-Information der einzelnen Knoten. Es wurden ausführliche Tests durchgeführt, die zeigen, dass die Ergebnisse dieses Algorithmus eine deutlich verbesserte Lokalität aufweisen im Vergleich zu bereits existierenden Algorithmen, wobei er dennoch eine bessere Laufzeit als diese aufweisen kann.

Contents

1 Introduction **13**

2 Problem Formulation **15**
2.1 Hypergraphs . . . 15
2.2 Graph Partitioning . . . 15
2.3 Benchmark Metrics . . . 16

3 Datasets **19**
3.1 Bipartite Datasets . . . 19
3.2 Reddit Hypergraph . . . 21

4 Background **23**
4.1 MinMax Streaming . . . 23
4.2 Multilevel Partitioning . . . 24

5 Hypergraph Partitioning with Neighbourhood Heuristic **27**
5.1 Original Neighbourhood Partitioning . . . 27
5.2 Primitive Hypergraph Node Partitioning with Neighbourhood Heuristics . . . 28
5.3 Optimization . . . 31

6 Evaluation **37**
6.1 Sum of External Degrees . . . 37
6.2 Edge Cut . . . 38
6.3 Balancing . . . 40
6.4 Runtime . . . 41

7 Conclusion **43**

Bibliography **45**

List of Figures

2.1 Hypergraph example 15

3.1 Stack Overflow edge distribution 19
3.2 Stack Overflow node distribution 19
3.3 Github edge distribution 20
3.4 Github node distribution 20
3.5 MovieLens edge distribution 21
3.6 MovieLens node distribution 21
3.7 Reddit edge distribution 21
3.8 Reddit node distribution 21

4.1 Multilevel partitioning summary[KK00] 24

5.1 Overview over the different sets managed while partitioning a hypergraph 30
5.2 Comparision of quality using the exact number of neighbours and caching the number of neighbours after the first calculation 31
5.3 Comparision of quality using the number of neighbours and accumulating the edgesizes of a node 32
5.4 Comparision of the runtime using the number of neighbours and accumulating the edgesizes of a node 32
5.5 Comparision of quality using different sizes of set $S \setminus C$ 34
5.6 Comparision of quality ignoring different sizes of edges during *expand* 35

6.1 Sum of external degrees compared for different edge sizes ignored when partitioning the different hypergraphs 38
6.2 Sum of External Degrees compared between different algorithms on different hypergraphs 39
6.3 Edge-Cut metric compared between different algorithms on different hypergraphs 40
6.4 Balancing compared between different algorithms on different hypergraphs . . . 41
6.5 Runtime compared between different algorithms on different hypergraphs 42

List of Tables

3.1 A table showing the used hypergraphs and information about their vertices, hyperedges and the number of edges in their corresponding bipartite graph 20

List of Algorithms

4.1 Greedy MinMax Streaming algorithm described in [AIV15] 23
5.1 Alloc algorithm for the neighbourhood edge partitioning algorithm[ZWL+17] . . 28
5.2 Neighbourhood expansion for the neighbourhood edge partitioning algorithm[ZWL+17] 28
5.3 Original neighbourhood edge partition algorithm for graphs[ZWL+17] 29
5.4 Alloc algorithm for the neighbourhood node partitioning algorithm 29
5.5 Neighbourhood node partitioning algorithm for graphs and hypergraphs 30
5.6 Node adding policy to decide which nodes are allowed to be in constant size $S \setminus C$ 33
5.7 Optimized *alloc* procedure adding only two nodes to S 34

1 Introduction

Today's communication services such as Whatsapp, Telegram or Facebook provide group functionalities with rather $1-n$ than $1-1$ communication patterns. Thus, increasingly data sets are modeled as graphs and hypergraphs which need to be processed. Due to the increasing size of the data sets and therefore also the hypergraphs, processing systems scale out processing by dividing the hypergraph into equally sized partitions [MMG+18; MMTR16; MTLR16; MTMR18]. When such partitions are made, the optimization goal is to minimize the numbers of hyperedges spreaded across multiple partitions and consequently the amount of communication between different machines.

This *balanced k-way partitioning problem* is NP-hard for graphs as well as for hypergraphs, which means calculating optimal solutions to this problem for this problem is not feasible for big graphs. Existing partitioning solutions are either slow and provide good quality cuts, or are fast and generate solutions of bad quality.

In this thesis, a new algorithm based on neighbourhood graph partitioning first proposed by Zhang et al.[ZWL+17] will be introduced. The algorithm grows k distinct node partitions by exploring the neighbourhood of the nodes already in the partition. This new algorithm will provide solutions to hypergraph partitioning of good quality while being scalable for huge hypergraphs. Using a naive neighbourhood graph partitioning algorithm as proposed by Zhang et al. does not scale well and the naive algorithm does not even return for hypergraphs which have as little as $\sim 50,000$ nodes in 24 hours. For this reason in this thesis further optimizations will be introduced in this thesis to make the algorithm viable for partitioning even huge graphs. This thesis will provide the following contributions:

- A hypergraph node partitioning algorithm derived from the edge partitioning algorithm proposed by Zhang et al. The new algorithm will grow k subgraphs based on the neighbourhood relation between nodes.
- Optimizations of the provided algorithm to make it feasable for partitioning huge hypergraphs.
- Two new hypergraphs based on Reddit comments. One hypergraph using comments as nodes and due to that having billion of nodes, one using subreddits as nodes and providing a usual distribution for hypergraphs, but being big enough to make existing algorithms fail when trying to process this graph.
- An evaluation of the new, optimized algorithm showing that it outperforms established hypergraph partitioning algorithms regarding relevant performance metrics when partitioning real world hypergraphs.

The rest of this thesis is organized as follows: Section 2 formally introduces hypergraphs, partitions and the graph partitioning problem. Furthermore, it will introduce and discuss different metrics to evaluate hypergraph partitionings. Section 3 introduces different datasets used to generate hypergraphs. These hypergraphs are later used to benchmark different partitioning algorithms.

This section also gives an insight into how the Reddit dataset has been used to create two huge hypergraphs. In Section 4 existing hypergraph partitioning algorithms are introduced and explained. Section 5 first describes the neighbourhood edge partitioning algorithm for graphs introduced by Zhang et al.[ZWL+17] and then introduces a hypergraph partitioning algorithm based on that algorithm. Since this naive hypergraph partitioning algorithm does not scale well, further optimizations for this algorithm are introduced in the section as well. This optimized new algorithm will be evaluated and compared to hypergraph partition systems intorduced in Section 4. For this, most of the hypergraphs introduced in Section 3 will be partitioned and the results evaluated in Section 6. The last section, Section 7 summarizes this thesis and gives an insight into future work related to the newly proposed algorithm.

2 Problem Formulation

2.1 Hypergraphs

Hypergraphs are a generalization of graphs, where edges do not connect only two but an arbitrary number of nodes. These edges are called *hyperedges*. Formally, a hypergraph H is a tuple (V, E) where V is a set of nodes and $E \subseteq \mathcal{P}(V) \setminus \emptyset$ which means, E is a set of non-empty node sets and therefore a subset of the powerset of V without the empty set. Figure 2.1 shows a visual representation of a hypergraph where $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$, $E = \{e_1, e_2, e_3, e_4\}$ and $e_1 = \{v_1, v_2, v_3\}$, $e_2 = \{v_1, v_4\}$, $e_3 = \{v_3, v_5, v_6\}$ and $e_4 = \{v_1, v_5\}$ Every graph can be represented by a hypergraph by using only hyperedges with the size of two. Also, every bipartite graph given as $G = (A, B, E)$ where A and B are two disjoint node sets and E is an edge set where every edge connects two nodes $x \in A$ and $y \in B$ can be repressented as a hypergraph. Such a graph can be converted to a hypergraph by using set A as nodes and set B as hyperedges. For every edge $(x, y) \in E$, the node $x \in A$ is connected in the hypergraph with hyperedge $y \in B$. If we proceed this way, no information about the origin bipartite graph is lost, which is why it also works the other way around. Every hypergraph $H = (V, X)$ can be transformed into bipartite graph $G = (A, B, E)$ when using the nodes $x \in V$ as nodes A and the hyperedges $y \in X$ as nodes B. Every node $x \in A$ is connected with another node $y \in B$ exacly when x was in the hyperedge y in the hypergraph which means $(x, y) \in E \Leftrightarrow \exists y \in X : x \in y$.

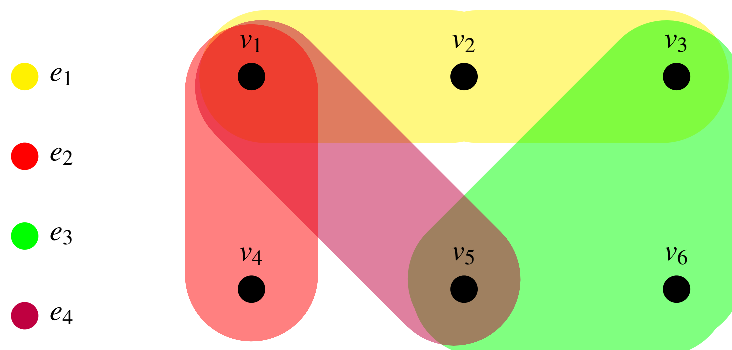


Figure 2.1: Hypergraph example

2.2 Graph Partitioning

When big graphs are processed, they need to be distributed accross several machines. Distributing the graph randomly leads to a high degree of duplication of either nodes, edges or both. which -in tern- necessitates high communication between the machines. Thus, algorithms for partitioning the

graph into smaller subgraphs so that the communication between the machines is minimized and the work is equaly distributed are needed. This problem is called the graph partitioning problem. The graph partitioning deals with splitting a graph $G = (V, E)$ into k subgraphs $(P_0, P_1, \ldots, P_{k-1})$ such that

$$G = \bigcup_{i=0}^{k-1} P_i \tag{2.1}$$

holds true, while globally minimizing some metrics. The graph partitioning problem is known to be NP-Complete [PQD+15] and since every graph can be represented as a hypergraph where hyperedges only hold two nodes, the hypergraph partitioning problem is also NP-Hard. Thus, finding an optimal solution which minimizes the metrics presented in section 2.3 is not reasonably feasible for big graphs, which is why existing solutions and our proposed algorithm are only able to find local optimal solutions for graphs as well as for hypergraph partitioning [MML17; MMT+18].

2.3 Benchmark Metrics

In order to compare different hypergraph partitioning algorithms, we need to provide a common set of benchmarks which evaluate the quality of a cut calculated by an algorithm for a given hypergraph and a number of partitions. This set of benchmarks needs to provide information about the following characteristics of a cut.

- **Runtime:** time the algorithm takes to calculate the cut.
- **Balancing:** balancing of the numbers of elements in the partitions
- **Quality:** communication later needed between machines because of commonly shared elements

2.3.1 Runtime

We will use the runtime of the algorithm as a benchmark on how much time the algorithm needs to calculate the cut for a given graph and a number of partitions. As some algorithms do calculations while the parsing of the graph and others do not, we will add the runtime of the parsers to read the given hypergraph into memory to the runtime needed to calculate the partitioning.

2.3.2 Balancing

Since we only compare algorithms which calculate edge partitionings on the given hypergraphs, we will benchmark the balancing of the number of nodes in the partitions. Let P_{max} be the partition with the highest number of nodes, P_{min} the one with the lowest and $|P_i|$ the number of nodes in partition P_i, then the quality B of the balancing is calculated as follows:

$$B = \frac{|P_{max}| - |P_{min}|}{|P_{max}|} \tag{2.2}$$

such that $0 \leq B \leq 1$.

A value of B near 1 means that there is a huge difference between the number of nodes of P_{max} and P_{min} and the cut is highly imbalanced, a value near 0 means the opposite.

2.3.3 Edge-cut

To measure the quality of a cut, the most straightforward way is to count the hyperedges which are spread on multiple partitions of the cut. This metric is called edge-cut metric and hMetis optimizes its cuts per default for it. This metric indirectly estimates the communication needed between the single partitions later. Since a single edge in a cut with k partitions can be spread across all those k different partitions and still be counted the same way as an edge spread across only 2 partitions, the edge-cut metric does neither directly nor reliably represent the amount of communication between the k partitions later.

2.3.4 Sum of External Degrees

Another metric to estimate how much partitions need to communicate later is called Sum of External Degrees. It estimates the communication by counting the hyperedges spread between different partitions. The external degree Ed of a partition P_i is the number of hyperedges which belong to partition P_i but the nodes of which are not all in partition P_i. With this definition of $Ed(P_i)$, the sum of external degrees $SoED$ of a cut with k partitions is defined as:

$$\text{SoED} = \sum_{i=0}^{k} Ed(P_i) \tag{2.3}$$

2.3.5 (k-1)-cut

Another metric to estimate the communication based on the number of hyperedges spread across multiple partitions is the (k-1)-cut metric. Let $H = (V, E)$ be a hypergraph with Edges E and vertices V, $(P_0, \ldots, P_{k-1})$ a valid cut of H into k partitions and $|P_i|$ the number of hyperedges which are in partition P_i. Then the (k-1)-cut metric is defined as follows:

$$\text{(k-1)-cut} = \left(\sum_{i=0}^{k} |P_i| \right) - |E| \tag{2.4}$$

Since

1. the Sum of External Degrees metrics is easier to calculate with the used datastructures and gives the same information as the (k-1)-cut metric about the communication between partitions
2. hMetis only provides the sum of external degrees metric out of the box

we will use the Sum of External Degrees in this thesis.

3 Datasets

In this chapter, different datasets will be introduced which are used to evaluate different partitioning algorithms. Since every bipartite graph can easily be converted into a hypergraph, most introduced datasets are bipartite graphs. The publicly available datasets are not big enough to provide a billion scale hypergraph, that is why we have used a billion scale reddit dataset to build a hypergraph large enough to evaluate if the proposed algorithms are able to do billion scale partitioning. Table 3.1 gives an overview of the generated hypergraph.

3.1 Bipartite Datasets

In this section, the various hypergraphs are presented which were built from online available bipartite graph data sets.

3.1.1 Stack Overflow

This bipartite graph data set provides a graph where each node is either a user or a post posted on Stack Overflow. Every user is connected with every post they have favorized. Thus, a user is never connected with another user nor a posting with another posting. Using the user nodes as nodes and posts as hyperedges, a hypergraph with 641.876 nodes and 545.196 hyperedges can be built[17b]. The resulting hypergraph has 15,772 components; this means it is not connected, which is not suprising considering the fact that posts exist which are favorized by exactly one user who has only one favorized post. This being said, the biggest component holds 96.2% of all nodes in the graph, and thus, the graph is not trivial to partition. Figure 3.1 and figure 3.2 show that the distributions of edges as well as nodes sizes in the built hypergraph follow the powerlaw.

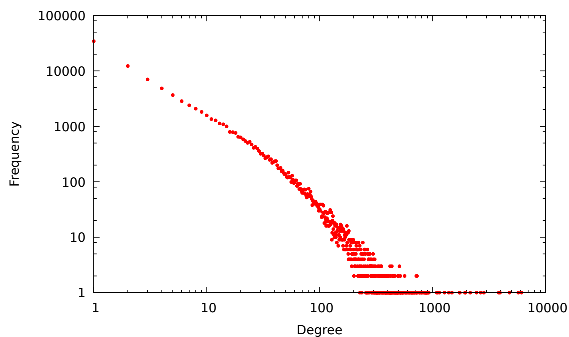


Figure 3.1: Stack Overflow edge distribution

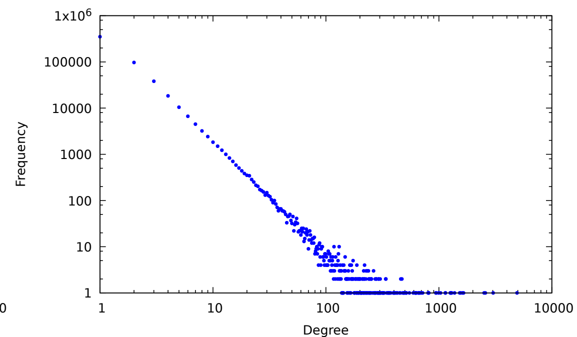


Figure 3.2: Stack Overflow node distribution

Dataset	Vertices	Hyperedges	#Vertices	#Hyperedges	#Edges	Source
Stack Overflow	Users	Posts	641,876	545,196	1,301,942	[17b]
MovieLens	Users	Movies	138,493	26,745	20,000,263	[Gro]
Github	Users	Projects	177,386	56,519	440,237	[17a]
Reddit	Subreddits	Authors	430,156	21,169,586	179,686,265	[Stu]

Table 3.1: A table showing the used hypergraphs and information about their vertices, hyperedges and the number of edges in their corresponding bipartite graph

3.1.2 Github

This bipartite dataset provides a graph where each node is either a user or a project. A user has an edge to a project if he is a member of the project. The dataset provides a graph with 177, 386 users and 56, 519 projects [17a]. The hypergraph built from the bipartite graph will use the users as nodes and the projects as hyperedges. When a user only is a member of a single project and this project only has one member, this member and project are building a component not reachable from the rest of the graph. Thus, it is not suprising that the resulting hypergraph has 15, 067 components and is not connected. The biggest component holds about 82.6% of the nodes in the graph, which means it is not trivial to perform a balanced node partitioning for the hypergraph. As Figure 3.3 and Figure 3.4 show, node and edge distributions are following the powerlaw for the built hypergraph.

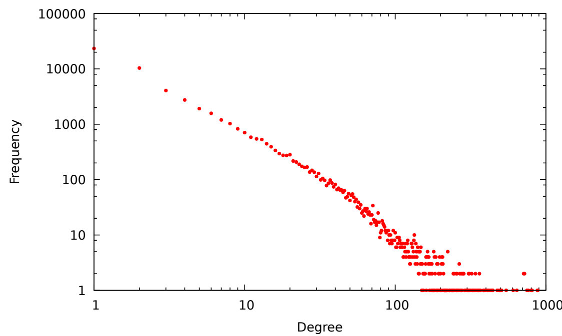


Figure 3.3: Github edge distribution

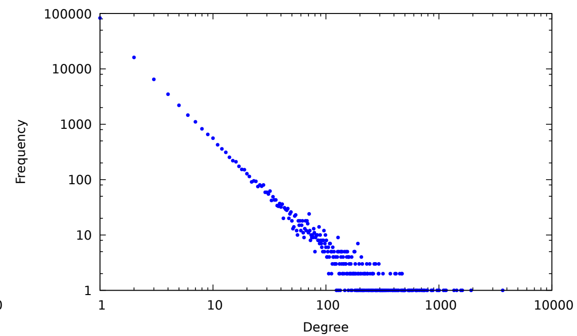


Figure 3.4: Github node distribution

3.1.3 MovieLens

The MovieLens dataset provides a bipartite graph with users and movies as nodes. Every user has edges to all movies he has rated, this way a user is never connected with another user and an edge is never connected with another edge, which makes the graph bipartite. The dataset has 138, 493 users who did 20, 000, 263 ratings for 26, 745 movies[Gro]. To concstruct a hypergraph from the dataset the user nodes were used to represent nodes and the movie nodes were used to represent hyperedges. Because of the high number of ratings made, the concstructed hypergraph is well connected and only consists of one component. Figure 3.6 shows that ever node is at least member of 20 hyperedges, which is also an indicator that the graph is highly connected. Aside from that, the distribution follows the powerlaw as well as the edge distribution does as Figure 3.5 shows.

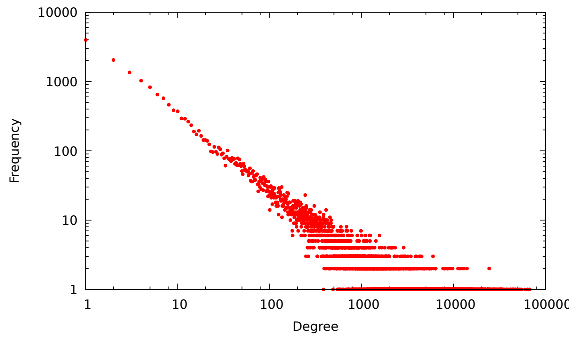


Figure 3.5: MovieLens edge distribution

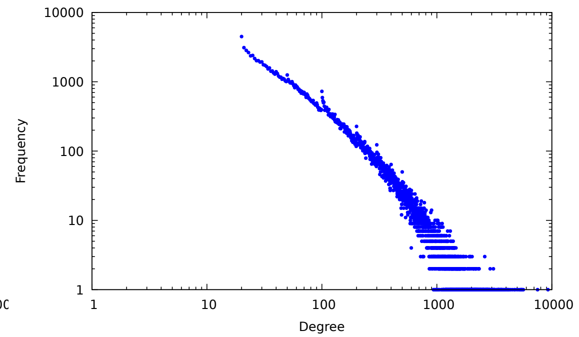


Figure 3.6: MovieLens node distribution

3.2 Reddit Hypergraph

Since the newly proposed algorithm in this thesis should be able to partition billion scale graphs, it is necessary to test on a much bigger hypergraph than the ones constructed from the bipartite datasets. Because of this, two new hypergraphs are introduced in this section: one hypergraph which will be used for comparing different algorithms and one with an unusual structure which will be used as a proof of concept for billionscale hypergraph partitioning. Reddit had about ~ 2.8 billion comments from 2005 to 2015 from $21,169,586$ authors in $430,156$ subreddits. When we use the authors and subreddits as nodes and connect every author with subreddits they commented in, a bipartite graph can be created. From this bipartite graph it is trivial to construct a hypergraph. In this case the subreddit nodes are used as nodes and the author nodes are used as hyperedges. This hypergraph has the edge and node distributions shown in Figure 3.7 and 3.8. The edge distribution follows the powerlaw, whereas nodes are not power-law distributed as the log-log curve does not show a straight line. Since this hypergraph is not billion scale either, another hypergraph was built from the dataset. When every comment is used as a node and every subreddit and author are used as a hyperedge, a billionscale hypergraph can be built by connecting every comment with his author and the subreddit it was posted in. The edge distribution of this new hypergraph follows the powerlaw, but the node distribution does not, because every node is connected with exactly two hyperedges. This graph has $\sim 2,8$ billion nodes and will be used later to prove that the proposed algorithm is able to partition billion scale hypergraphs. Because the node distribution is unusual for a hypergraph, it will not be used to compare different hypergraph partitioning algorithms.

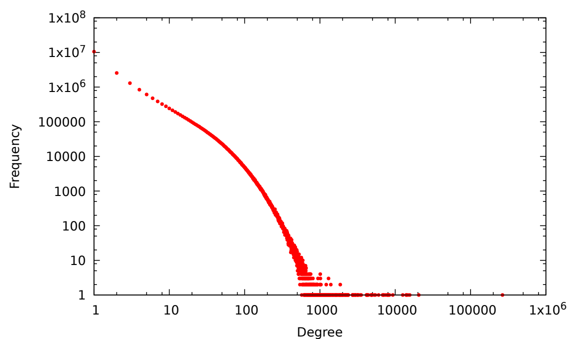


Figure 3.7: Reddit edge distribution

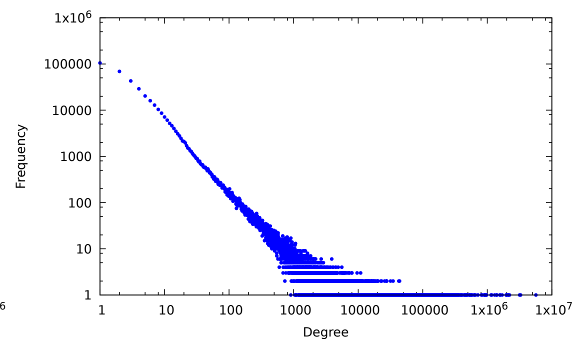


Figure 3.8: Reddit node distribution

4 Background

4.1 MinMax Streaming

Alistarh, Iglesias, and Vojnovic proposed different hypergraph streaming partitioning strategies in [AIV15]. These MinMax Streaming strategies can operate on a hypergraph with an unknown size or structure. The best strategy proposed in [AIV15] was the *greedy* strategy. The original greedy streaming algorithm expects a number of partitions k, a balancing constraint c and a source to stream the nodes of the hypergraph from. Algorithm 4.1 shows how greedy streaming works. First k empty partitions are initialized. This happens in line 2. Then, while new nodes v, which are connected with hyperedges R, are arriving, the newly arrived node is being added to the partition with which it has the most hyperedges in common and which does not violate the balancing constraint.

Algorithm 4.1 Greedy MinMax Streaming algorithm described in [AIV15]

```
 1: procedure GREEDYSTREAMING(k, c)
 2:     S_0, S_1, ...S_{k-1} ← ∅
 3:     while vertices are arriving do
 4:         v ← newly streamed vertex
 5:         R ← hyperedges v is connected with
 6:         I ← {i | ∀j : |P_i| ≤ |P_j| + c}
 7:         j ← arg min_{i∈I} |P_i ∪ R|
 8:         P_j ← P_j ∪ R
 9:     end while
10:     return S_0, S_1, ...S_{k-1}
11: end procedure
```

The original algorithm used a fixed size parameter c for the balancing. Since hypergraphs with different sizes are partitioned in this thesis, the original MinMax streaming algorithm has been rewritten to use a balancing constraint where the partitioning can only be imbalanced by 5% at any time. This way the parameter c can be omitted and does not need to be specified for every hypergraph.

Another problem the original proposed algorithm has is, that it uses edge balancing in its original form. Since we use node balancing for our algorithm, we implemented node balancing for the MinMax streaming as well. The resulting algorithm was almost always better than the one originaly introduced by Alistarh, Iglesias, and Vojnovic.

4.2 Multilevel Partitioning

The multilevel partitioning approach is another hypergraph partitioning approach introduced by Karypis and Kumar in [KK00]. For this algorithm it is crucial to have the whole hypergraph parsed into memory. In contrast to MinMax streaming, multilevel k-way partitioning takes a lot more time to process the hypergraph, but produces much better partitionings in regard to quality.

This approach processes the graph in three phases: the *coarsening* phase, the *initial partitioning* phase and the *uncoarsening* phase.

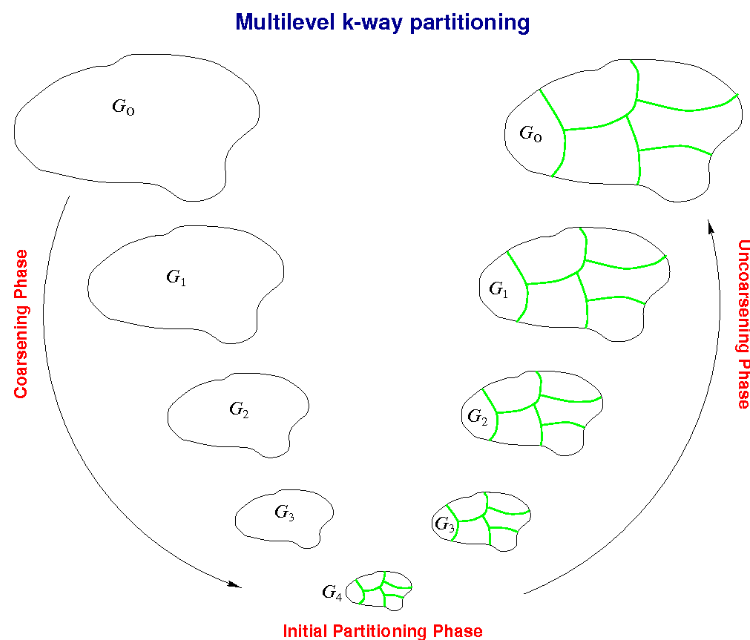


Figure 4.1: Multilevel partitioning summary[KK00]

In the *coarsening* phase, the hypergraph is gradually reduced by merging nodes together. That way hyperedges and the hypergraph itself are getting smaller and therefore it is much easier to calculate a good partitioning. Once this is done, the original hypergraph is reconstructed step by step from the smaller partitioned hypergraph. Figure 4.1 gives a rough idea of how the three different phases work together. During the *coarsening* phase it is crucial to merge nodes in a way that when partitioning the resulting smaller hypergraph, the partitioning is also valid and provides a good quality for the original hypergraph. For the *coarsening* phase different heuristics where propsed in literature to provide such a merging of nodes. For example *edge-coarsening*[KAKS99], *hyperedge-coarsening*[KAKS99] or *FirstChoice*[KK00].

In the second phase, the small hypergraph generated in the *coarsening* phase needs to be partitioned into k rough equaly sized partitions with good quality. In literature different approaches are proposed such as doing the *coarsening* phase until only k nodes are left and use them as initial partitions [KK00]. Other approach such as using different random partitionings of the small hypergraph and compare them to each other or exploring the neighbourhood of random selected nodes and use those neighbourhoods as partitionings as proposed in [KAKS99] are prefered and also used in hMetis.

In the last phase, the *uncoarsening phase* the partitioning of the smaller partitioned hypergraph is consecutively into another one, by uncoarsening it and optimizing the results with a refinement algorithm which optimizes the cut for a given metric such as the *Edge-Cut* or *Sum of External Degrees* metric discussed in Section 2.3. Different refinement algorithms were proposed by literature such as FM[KAKS99; KK00] or greedy refinement.

HMetis implements different multilevel partitioning systems with different algorithms used in the different phases [KAKS99; KK00]. For the evaluation and comparisions against our newly proposed algorithm we used the hMetis recusive bisecioning algorithm. We considered using the multilevel k-way partitioning system, but we were not able to partition big hypergraphs in a appropriate time to do so.

5 Hypergraph Partitioning with Neighbourhood Heuristic

Graph partitioning with neighbourhood heuristic was proposed by Zhang et al. for egde graph partitioning. In this thesis we used the neighbourhood heuristic to provide an algorithm which works as a scalabe node partitioning algorithm for hypergraphs with nodebalancing. In this chapter the original algorithm will be explained, and then transformed in such a way that the result will be a node partitioning algorithm for hypergraphs. Without further optimization this new algorithm does not scale well. That is why it will be optimized not only to run in $O(n)$ but also to be able to partition billion scale hypergraphs which will, in most cases, definitely produces better results than the existing solutions.

5.1 Original Neighbourhood Partitioning

In this section the original algorithm proposed by Zhang et al.[ZWL+17] will be explained. The algorithm successively constructs edge sets E_i in k iterations from a given graph in a way that those sets are a valid edge partitioning of the graph. When such a set E_i is constructed, two sets of nodes S and C are administered by the algorithm. Set S, is a set of nodes, which are good candidates for their edges being added to E_i and C, a set which contains the already worked off nodes. In every step a node $n \in S \setminus C$ is selected by the later described *neighbourhood expansion*. This node n is then added to set C and all neighbours $N(n)$ of node n are added to set S. All edges of this selected node n are added to the edge partition E_i. To make sure an edge is not distributed on more than one partition, each edge added to E_i will get deleted from the graph edge set E. If node n is not selected from S, it will also be added to S. Because of this, set S is the set of nodes which are connected with at least one edge in the current edge set E_i. Since a node gets never deleted from S or C and all nodes in C are included in S or added to S respectivly, $C \subseteq S$ is always true. In this thesis the procedure of adding new nodes or edges to a partition is called *alloc*. An example of how this *alloc* procedure can be implemented for the original neighbourhood partitioning algorithm can be seen in algorithm 5.1.

The *neighbourhood expansion* selects the node $n \in S \setminus C$ in a way that the number of neighbours of n which will be added to S is minimized, which means:

$$n = \underset{x \in S \setminus C}{\arg\min} |N(v) \setminus S| \tag{5.1}$$

Obviously, if $S \setminus C = \emptyset$, it is not possible to select such a node out of $S \setminus C$ and in this case a random node $n \notin C$ is selected from the given graph G. This procedure is called *neighbourhood expansion* or *expansion* respectivly. Algorithm 5.2 shows the *expand* procedure, which implements the *neighbourhood expansion* and shows how node n is selected for the original edge partitioning algorithm proposed by Zhang et al.

Algorithm 5.1 Alloc algorithm for the neighbourhood edge partitioning algorithm[ZWL+17]

```
 1: procedure ALLOC(n, E, E_k, C, S, δ)
 2:     S ← S ∪ {n}
 3:     C ← C ∪ {n}
 4:     for all x ∈ N(n) \ S do
 5:         S ← S ∪ {x}
 6:         for all y ∈ N(x) ∩ S do
 7:             if |E_k| ≤ δ then
 8:                 E_k = E_k ∪ {e_{x,y}}
 9:                 E ← E \ {e_{x,y}}
10:             end if
11:         end for
12:     end for
13: end procedure
```

Algorithm 5.2 Neighbourhood expansion for the neighbourhood edge partitioning algorithm[ZWL+17]

```
1: procedure EXPAND(S, C, G = (V, E))
2:     if S \ C = ∅ then
3:         return random node ∈ V \ S
4:     else
5:         return argmin_{x ∈ S\C} |N(v) \ S|
6:     end if
7: end procedure
```

Since neighbourhood expansion tries to minimize the number of new nodes added to S, nodes which are well connected with the current set S are prefered and the number of nodes $n \in S$ with edges to nodes $x \notin S$ is directly minimized. This results in a minimization of identic nodes on multiple partitions, which is exactly what is needed to provide a good edge partitioning. Later, when the neighbourhood heuristic for node partitioning of hypergraphs is used, advantage of those properties of nodes in set S and C will be taken. Algorithm 5.3 shows the whole edge partitioning algorithm when the previously in algorithm 5.1 and in algorithm 5.2 introduced *alloc* and *expand* procedures are used.

5.2 Primitive Hypergraph Node Partitioning with Neighbourhood Heuristics

In this section we will change the original edge partitioning algorithm proposed by Zhang et al. and described in section 5.1 in such a way that it will work on hypergraphs, the result being a node partitioning instead of the original edge partitioning. Since the new algorithm should be a node partitioning algorithm, saving the edges for the edge partition E_i in each iteration of the algorithm

Algorithm 5.3 Original neighbourhood edge partition algorithm for graphs[ZWL+17]

```
 1: procedure PARTITION(G = (V, E), p)
 2:     δ ← |E| / p
 3:     for k ← [0 .. p − 1] do
 4:         S, C, E_k ← ∅
 5:         while |E_k| ≤ δ ∧ E ≠ ∅ do
 6:             n ← EXPAND(S, C, G)
 7:             ALLOC(n, E, E_k, C, S, δ)
 8:         end while
 9:     end for
10:     return (E_1, . . . , E_{p−1})
11: end procedure
```

is no longer needed, which results in a new *alloc* procedure described in algorithm 5.4. This is the result of deleting the edge allocation part of the original *alloc* procedure descriped in algorithm 5.1 in lines 6 to 11.

The algorithm needs to provide a node partition V_i in each round. The nodes in this partition should preferably have few edges to nodes not in V_i and are not allowed to appear on any other node partition V_j for $j \neq i$. The new algorithm will use the node set C for this. Since C is built from nodes $n \in S$ in such a way that during the *neighbourhood expansion* as few as possible new nodes are added to S, C provides a set of nodes with few edges to nodes at least not in S. The algorithm could provide a better partitioning, when changing the *neighbourhood expansion* in a way that nodes with few neighbours not in C are prefered, but it will be changed anyway in a later section 5.3.1 due to the lacking performance of this form of *neighbourhood expansion*. For now we assume that using C as node partition will provide a reasonable partitioning of the given graph.

Since the algorithm needs to make sure a node $n \in V_i$ will never be assigned to any partition V_j for $j \neq i$, n is deleted from the given graph which is done in line 7 in algorithm 5.4.

Algorithm 5.4 Alloc algorithm for the neighbourhood node partitioning algorithm

```
1: procedure ALLOC(n, C, S, G = (V, E))
2:     S ← S ∪ {n}
3:     C ← C ∪ {n}
4:     for all x ∈ N(n) \ S do
5:         S ← S ∪ {x}
6:     end for
7:     V ← V \ {n}
8: end procedure
```

The new algorithm does perform a node partitioning, which is why the calculation of δ needs to be changed in order to dependend on the number of nodes in the given graph. Algorithm 5.5 does exactly this in line 2 and is the result of all changes to algorithm 5.3 discussed in this section. Line 7 calls the previously discussed *alloc* procedure. Line 6 calls the *expand* procedure which has been presented in section 5.1 and discussed in the current section. $C_0, \ldots, C_{p-1}$ in line 10 has the meaning of returning all sets C calculated in iteration 0 *to* $p - 1$

Algorithm 5.5 Neighbourhood node partitioning algorithm for graphs and hypergraphs

```
1:  procedure PARTITION(G = (V, E), p − 1)
2:      δ ← |V| / p
3:      for k ← [0 .. p] do
4:          S, C ← ∅
5:          while |C| ≤ δ ∧ V ≠ ∅ do
6:              n ← EXPAND(S, C, G)
7:              ALLOC(n, C, S)
8:          end while
9:      end for
10:     return (C_0, ..., C_{p−1})
11: end procedure
```

Figure 5.1 sketches the general framework for growing the core set of a single partition and how nodes are moved between the different sets.

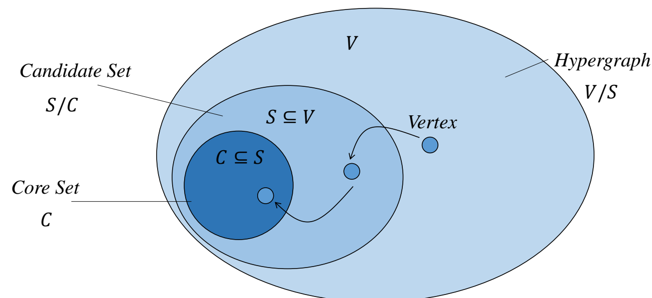


Figure 5.1: Overview over the different sets managed while partitioning a hypergraph

This new algorithm provides a node partitioning of a given graph. Since we only use nodes and neighbours of those nodes in the algorithm and since nodes as well as neighbours of a node are well defined for hypergraphs, the new algorithm can be used to perform node partitioning for hypergraphs.

There are several performance problems with this algorithm when hypergraphs are partitioned, such as the following: since the edges in a hypergraph are sets of nodes, a node in a hypergraph has usually more neighbours than in a normal graph. This is a problem because S grows in proportion to average number of neighbours a node has. When there is an edge with ever node in it, S is always the full graph. Because of this the results as well as the runtime of the algorithm are remarkably bad. An implementation of the algorithm did not return within 24 hours on hypergraphs with only $\sim 50,000$ nodes and edges. In the following sections these problems will be adressed and solved to make the current algorithm feasable for billion scale hypergraphs.

5.3 Optimization

In this section, optimization for the hypergraph node partitioning algorithm will be discussed. As said before, the naive hypergraph node partitioning algorithm presented in Section 5.2 does not scale very well. After the application of the optimizing factors discussed in this section, the algorithm will be able so scale in $O(n)$ and keep up with existing solutions.

5.3.1 Heuristic Optimization

Currently, during the *neighbourhood expansion* a node is ranked by the number of his neighbours $x \notin S$. In order to calculate this number, first the set of neighbours needs to be computed and then those neighbours are counted based on whether they are in S or not. This needs to be done for all nodes in S, everytime the *expand* procedure is called.

When a node has a only a few neighbours it can only have few neighbours not in S. Therefore, instead of comparing nodes based on the number of neigbours not in S, the absolute number of neighbours of a node can be used to decide if a node is a good candidate to expand. This is much easier to calculate because no set differences need to be calculated.

Everytime a new node is allocated to a partition, this node gets deleted from the graph. This means, due to that event the number of neighbours of any node in the graph can possibily change. Thus, caching the number of neighbours of each node gives other results than recalculating it everytime needed. After running tests using the exact and the cached version of neighbours on different graphs, it turns out that caching the number of neighbours does not affect the quality of the resulting partitioning, but reduces runtime by $\sim 50\%$. Figure 5.2 shows that there is now difference of quality between the cached and the exact number of neighbours, provided that the sum of external degrees metric discussed in section 2.3.4 is applied, for the partitioning of the Stack Overflow hypergraph discussed in section 3.1.1.

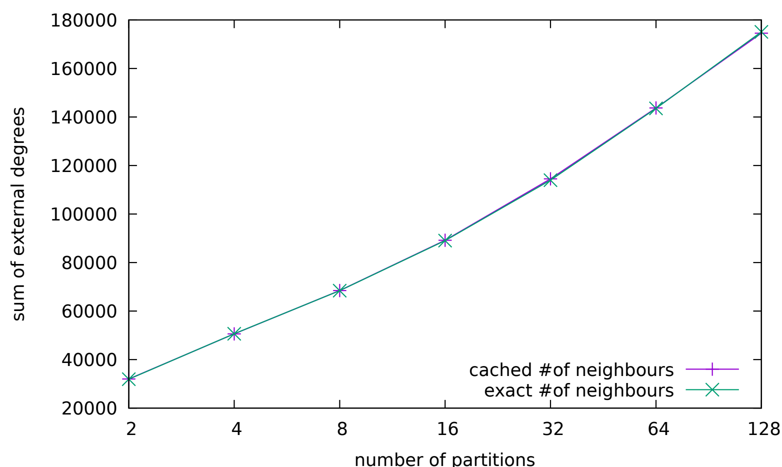


Figure 5.2: Comparision of quality using the exact number of neighbours and caching the number of neighbours after the first calculation

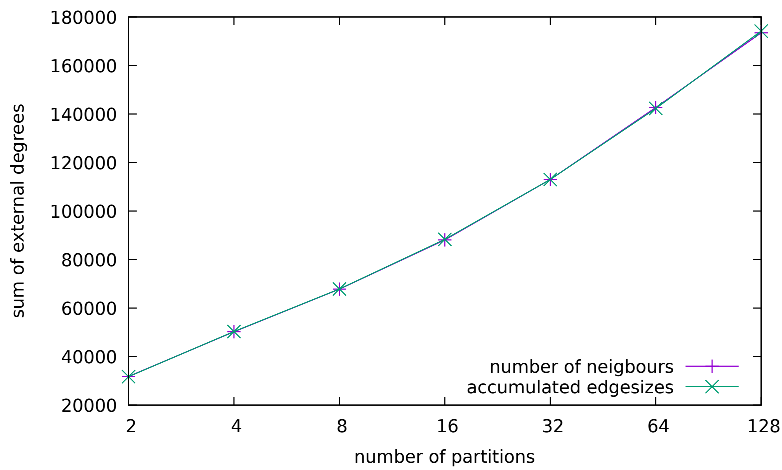


Figure 5.3: Comparision of quality using the number of neighbours and accumulating the edgesizes of a node

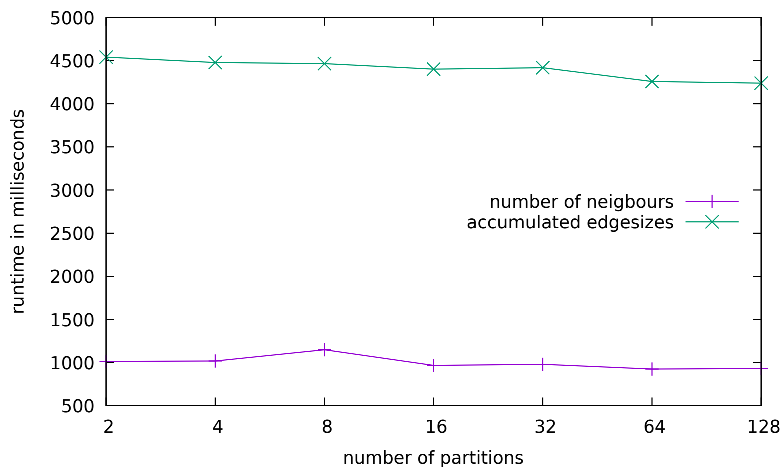


Figure 5.4: Comparision of the runtime using the number of neighbours and accumulating the edgesizes of a node

An even faster heuristic is to accumulate the sizes of edges a node is member of. This is not the same as the excat number of neighbours because neighbours of a node n can be spread across multiple edges shared with n. If one compares the two *expation* metrics on different graphs has shown that accumulating the edge sizes results in at least as good partitioning results as if the number of neighbours is used. Figure 5.3 shows that when the Github hypergraph introduced in section 3.1.2 is partitioned the quality of the results is not affected if one either uses the cached number of neighbours or the cached accumulated edgesizes. Moreover, figure 5.4 shows that using the accumulated edgsize metric can save up to 450% runtime.

Because of these results, instead of using the number of neighbours not in S for the *neighbourhood expantion*, the accumulated edgesizes calculated the first time when needed and then cached will be used in the algorithm. Due to these results, the number of neighbours not in S for the *neighbourhood expasion* will not be used. Instead, the accumulated edgesized calculated the first time needed and then cached will be used in the algorithm.

5.3.2 Restricting the Number of Nodes in Set $S \setminus C$

A issue of the current algorithm is that $S \setminus C$ can hold a lot of nodes, which is a problem because from these nodes the one with the smallest value acording to the heuristic discused in section 5.3.1 needs to be found to be added to C. To reduce the search time for the best node during the *neighbourhood expantion* $|S \setminus C|$ is restricted to a constant size. This results in constant search time for finding best node in $S \setminus C$. To make sure a constant size $S \setminus C$ does not only hold the worst possible candidates, a new *node adding policy* ensures only the best known candidates will be in $S \setminus C$. The *node adding policy* works as described in algorithm 5.6. It first unites the existing set $S \setminus C$ with the candidate nodes to get a set of all nodes which are candidates for the new set $S \setminus C$. It then sorts those nodes based on the heuristic introduced in 5.3.1 and then adds the best k nodes to the set $S \setminus C$, where k is the constant number of nodes restricting $S \setminus C$.

Algorithm 5.6 Node adding policy to decide which nodes are allowed to be in constant size $S \setminus C$

1: **procedure** addNodesToS($S, C, addCandidates$)
2:     $allNodes \leftarrow S \setminus C \cup addCandidates$
3:     *sort allNodes based on the in section 5.3.1 introduced policy*
4:     $S \leftarrow C \cup$ *first n nodes of allNodes*
5: **end procedure**

This makes sure that only the best k expation candidates are in $S \setminus C$. But it adds extra complexity and runtime when adding nodes to S, which will be solved in the next section. The question of how to choose k has been solved by running the algorithm on different graphs with different sizes of k. Figure 5.5 shows the quality mesured with the sum of external degrees metric discussed in section 2.3.4 of the results for different sizes of k when partitioning the Stack Overflow hypergraph introduced in section 3.1.1. The plot shows that $S \setminus C$ does not need to be high and is chosen to be 10 in this thesis, because the smaller $S \setminus C$ is, the less runtime is needed when chosing the next node during the *expation*. However, as the plot shows, choosing a bigger n does not affect the quality of the partitioning much.

5.3.3 Reducing the Number of Candidates for S

To reduce the cost of inserting a node into set S with the procedure introduced in 5.3.2 the number of nodes which will be added to S needs to be reduced. Since S only holds the best discovered candidates, it is not too bad if bad candidates are sometimes added to S, as long as not only bad candidates are added. Restricting the number of new nodes added to S to only two nodes, makes sure that everytime a node from S is added to C, two new candidates are available to refill $S \setminus C$. According to the *node adding policy* described in section 5.3.2 etter one of those two nodes will be added, for sure and the other one will be added if any node in $S \setminus C$ is worse. This makes sure

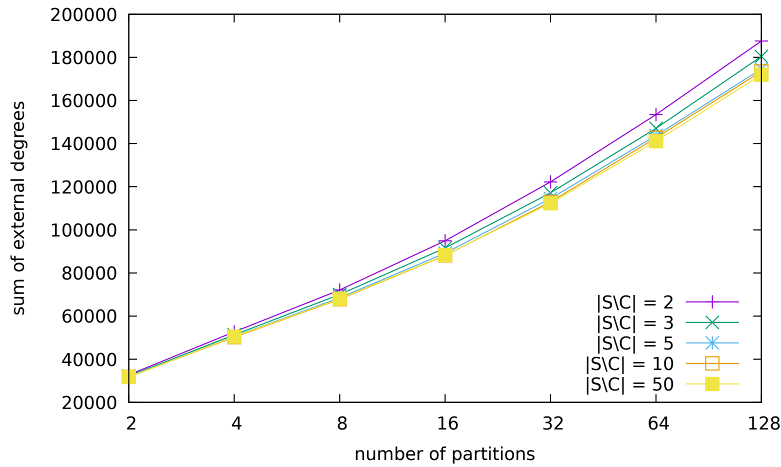


Figure 5.5: Comparision of quality using different sizes of set $S \setminus C$

that the quality of candidates in $S \setminus C$ increases during the execution of the algorithm. Since the *node adding policy* described in section 5.3.2 makes sure $S \setminus C$ only holds good candidates, the algorithm can randomly choose the two nodes which will be added to S from the neighbours of the node added to C. This new *alloc* procedure is also descriped in algorithm 5.7

Algorithm 5.7 Optimized *alloc* procedure adding only two nodes to S

1: **procedure** ALLOC$(n, C, S, G = (V, E))$
2: $S \leftarrow S \cup \{n\}$
3: $C \leftarrow C \cup \{n\}$
4: $addCandidates \leftarrow$ *select 2 random nodes from* $N(n) \setminus S$
5: ADDNODESTOS$(S, C, addCandidates)$
6: $V \leftarrow V \setminus \{n\}$
7: **end procedure**

5.3.4 Ignoring Edges while Selecting Candidates for S

The current algorithm has a weak point. When the algorithm selects two random neighbours of a node n as candidates to be added to S, those nodes can be connected with n through a big edge. this edge is almost surely also on another partition than the one currently built. This is certain to happen if the edge has more nodes than δ. This is why the algorithm should at least ignore edges $\geq \delta$ during the random selection of two candidates to be added to S. Tests have shown that, based on the graph, ignoring $5 - 50\%$ of the biggest edges in the graph considerably improves the quality of the partitioning. Figure 5.6 shows the differences in quality of partitionings of the hypergraph generated for the bipartite Github graph[17a]. In the figure the algorithm ignores different sizes of edges during *neighbourhood expansion*.

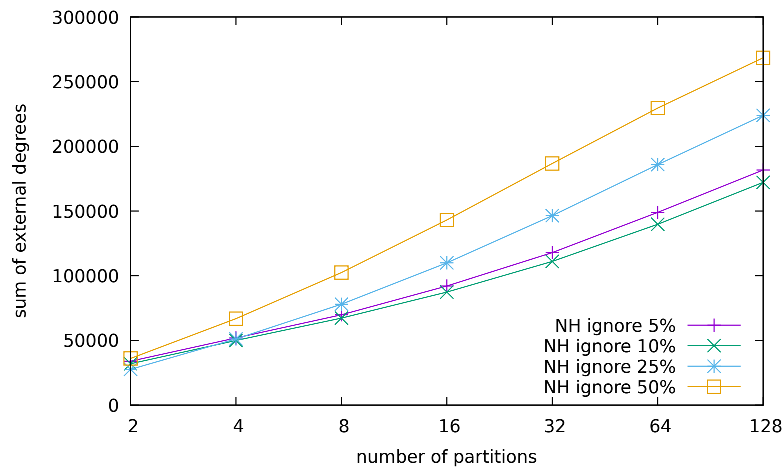


Figure 5.6: Comparision of quality ignoring different sizes of edges during *expand*

On different graphs a different number of edges ignored produces the best result. How to find a good value for this parameter is not covered by this thesis, but for the graphs introduced in section 3 the optimal parameters have been found and will be used in the following sections. With these optimizations a new node from a given hypergraph can be added to a partition in $O(1)$ and therefore the algorithm can partition a given given hypergraph $G = (V, E)$ in $O(|V|)$.

6 Evaluation

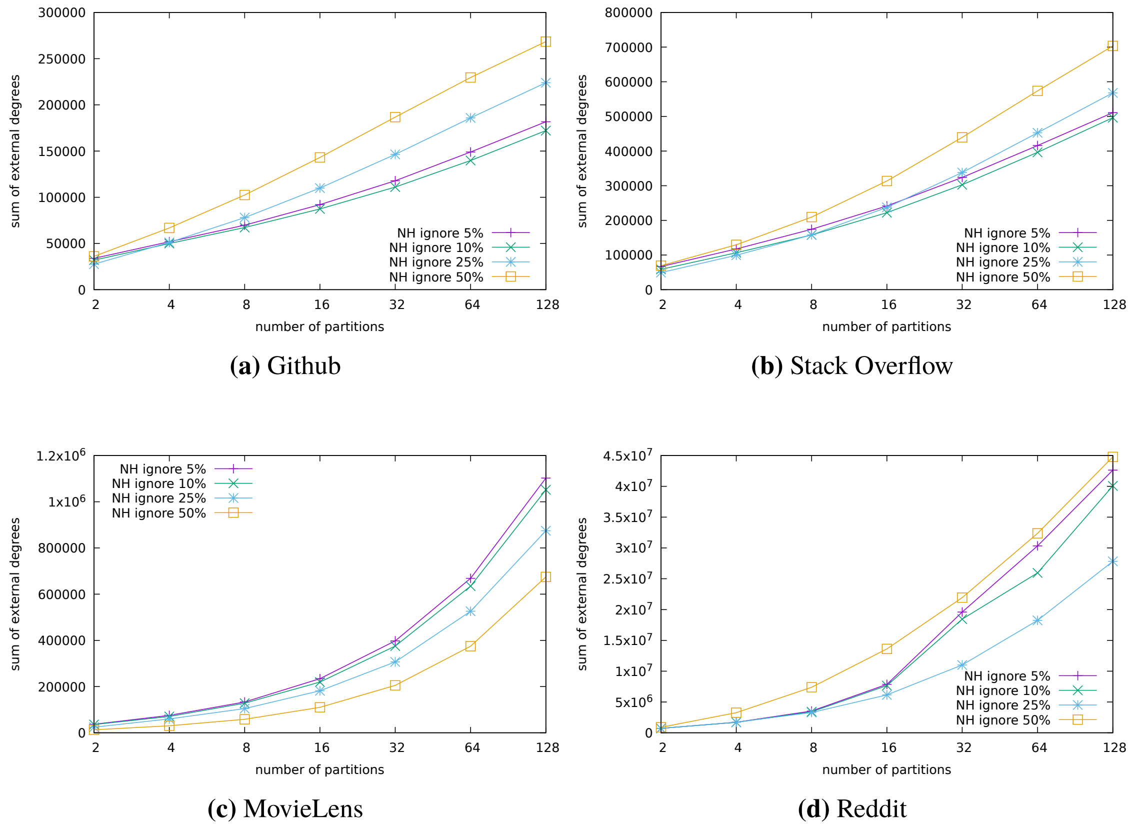
Based on the metrics introduced in Section 2.3, the newly proposed algorithm will be compared to other hypergraph partitioning systems. The implementation of the algorithm uses hashsets for all sets, and to avoid an $O(n)$ complexity when selecting random nodes, the first available node of those hashsets is chosen to make sure random node selection is possible in $O(1)$. The neighbourhood partitioning algorithm is compared with streaming algorithms presented in chapter 4 and hMetis. Since hMetis produces not strictly balanced cuts without further arguments, hMetis was benchmarked with default arguments and with an argument to make hMetis produce better balanced results. The algorithms are compared processing all hypergraphs presented in chapter 3 except hMetis, which did not return after 24 hours while trying to partition the reddit hypergraphs introduced in section 3.2. Since the neighbourhood partitioning algorithm tends to be sensitive to the percentage of the biggest edges ignored during the *neighbourhood expansion*, different values are tested for each hypergraph and the best result is used for the comparison with the other algorithms.

Experimental Setup All experiments have been performed on a shared memory machine with 4 x Intel(R) Xeon(R) CPU E7-4850 v4 @ 2.10GHz (4 x 16 cores) with 1 TB RAM.

6.1 Sum of External Degrees

In this section the different algorithms are compared based on how they perform in regard to the Sum of External Degrees metric introduced in section 2.3.4. HMetis has the option to optimize specifically for this metric, which was not possible for our hypergraphs, because hMetis did not return after 24 hours when doing so. Due to that hMetis was only instructed to perform partitionings optimized for the Edge-Cut metric introduced in Section 2.3.3.

Figures 6.1a - 6.1d show how the neighbourhood partitioning algorithm performs on the differnent hypergraphs with different numbers of edges ignored as explained in section 5.3.4 in regard to the Sum of External Degrees metric. As the figures show, on different graphs the algorithm needs to ignore a different percentage of the biggest hyperedges during the *neighbourhood expansion* to perform well. For all other comparisons between different partitioning algorithms, the number of edges ignored will be the one performing best in regard to the Sum of of External Degrees metric. Thus, when comparing different algorithms based on the Github or Stack Overflow hypergraphs, the biggest 10% of the hyperedges will be ignored during the *neighbourhood expansion*, based on the MovieLens hypergraph 50% and based on the Reddit hypergraph 25%. It is obvious that the optimal number of ignored biggest hyperedges depends on the density and the distribution of edge and node degrees of the hypergraph, but the calculation of the optimal percentage of the parameter are not covered by this thesis.



(a) Github

(b) Stack Overflow

(c) MovieLens

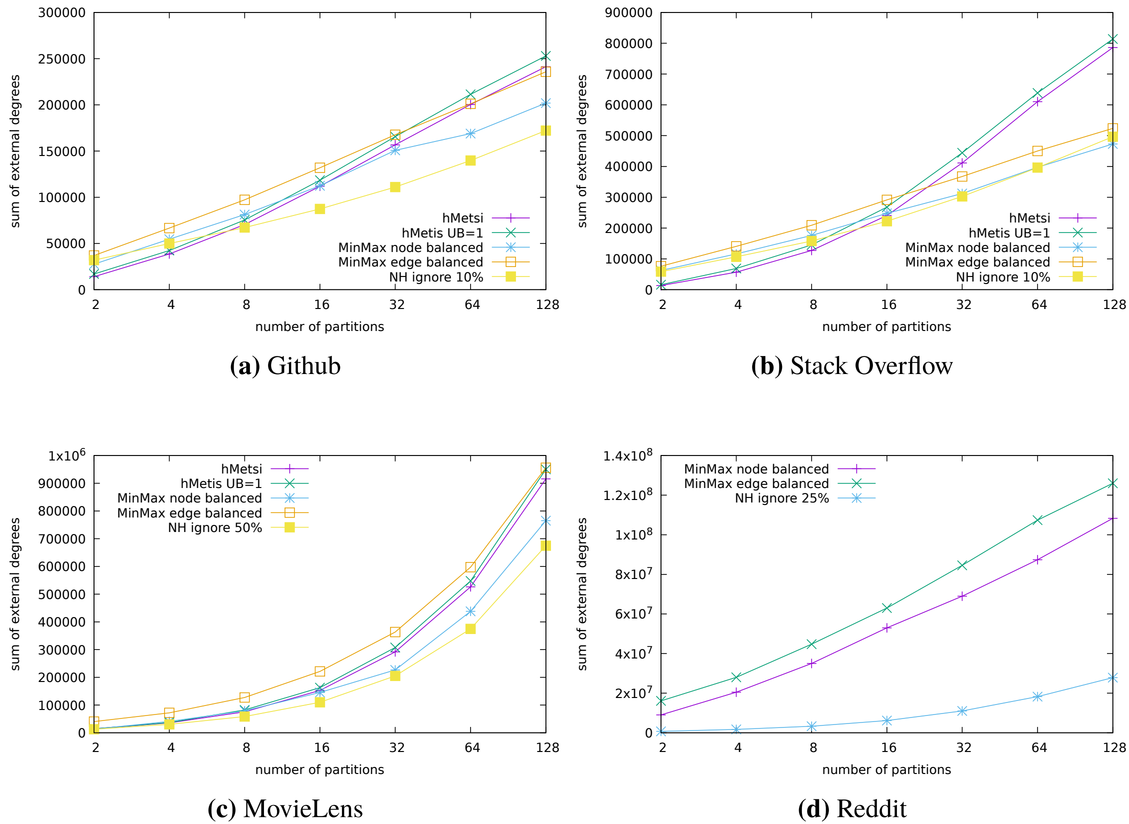
(d) Reddit

Figure 6.1: Sum of external degrees compared for different edge sizes ignored when partitioning the different hypergraphs

Figures 6.2a - 6.2d show the comparison of the neighbourhood partitioning algorithm with other hypergraph partitioning methods in regard to the Sum of External Degree metric which mesures the quality of the cut concerning the communication effort between machines later. It is remarkable that the neighbourhood partitioning algorithm described in this thesis scales better than hMetis regardless of whether hMetis produces balanced or unbalanced results. For few partitions, hMetis produces better results when partitioning the Stack Overflow or the Github hypergraphs, but for more than 8-16 partitions neighbourhood partitioning is the better choice. The new algorithm is also better than the streaming algorithms except for very few cases. Remarkably better results are being produced on the Reddit hypergraph when using the neighbourhood partitioning algorithm instead of the streaming algorithms.

6.2 Edge Cut

In this section, the different algorithms are compared to each other based on how they perform in regard to the Edge-Cut metric introduced in section 2.3.3. HMetis does in fact optimize results specifically for the Edge-Cut metric. As discussed in section 2.3 the Edge-Cut metric does not



(a) Github

(b) Stack Overflow

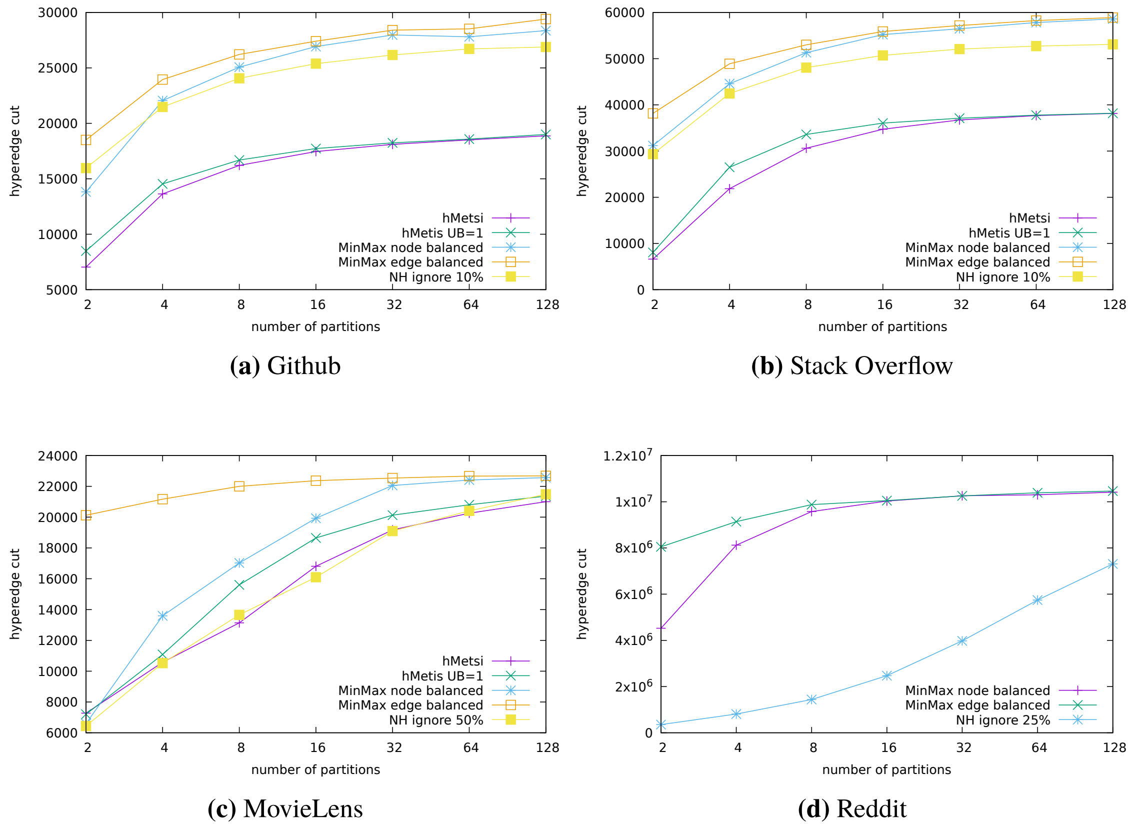
(c) MovieLens

(d) Reddit

Figure 6.2: Sum of External Degrees compared between different algorithms on different hypergraphs

directly model the communication between the machines later, which is why the Sum of External Degrees metric is much more important to be evaluated when speaking about the quality of results in terms of communication effort.

Figures 6.3a - 6.3d show how the different partitioning algorithms perform on different hypergraphs in regard to the Edge-Cut metric introduced in section 2.3.3. On the Stack Overflow and Github hypergraphs hMetis outperforms the other algorithms by magnitudes. When the MovieLens hypergraphs gets partitioned, the neighbourhood partitioning algorithm can compete with hMetis and sometimes even outperforms hMetis in regard to the edge cut metric. On all hypergraphs, the neighbourhood partitioning algorithm is superior to both streaming algorithms. Especially when the Reddit hypergraph is partitioned, the streaming algorithms are far worse than neighbourhood partitioning. As said, before hMetis is not able to partition the Reddit hypergraph, which is why a comparison between hMetis and the other algorithms is not possible for the Reddit hypergraph.



(a) Github

(b) Stack Overflow

(c) MovieLens

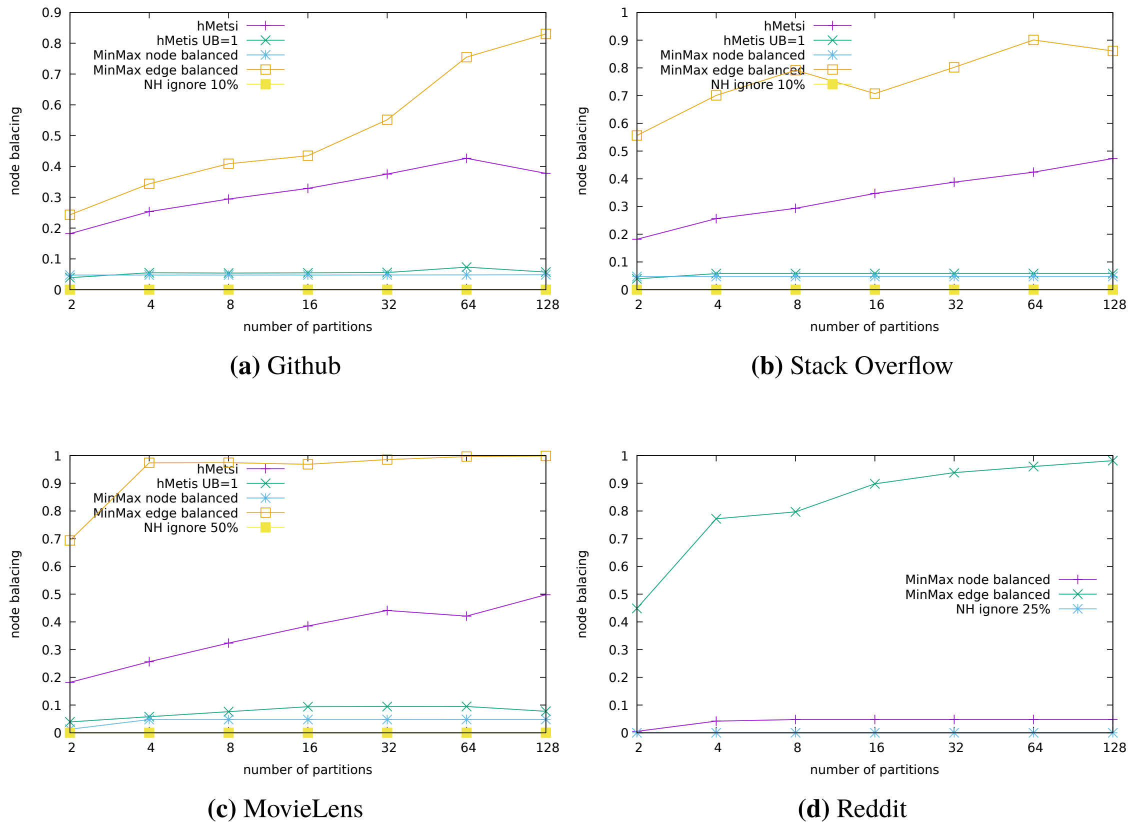
(d) Reddit

Figure 6.3: Edge-Cut metric compared between different algorithms on different hypergraphs

6.3 Balancing

In this section the different hypergraph partitioning systems are compared in regard to the question how well the resulting partitionings are node-balanced. Node-balancing is mesured by the metric introduced in section 2.3.2. Since the original MinMax streaming algorithm introduced by Alistarh, Iglesias, and Vojnovic uses hyperedge balancing[AIV15], this algorithm does not provide good node balancing at all.

In Figures 6.4a - 6.4d the different balancing behaviours of the different algorithms are displayed. As expected the MinMax streaming algorithm doing edge balancing, does not provide a good node balanced cut at all. The MinMax streaming algorithm using node base balancing, does provide partitionings which have 5% node inbalacing maximum. When hMetis is called with stricter balancing parameters it does as well provide most partitionings with approximately 5% imbalacing. If hMetis is not called with such a balancing parameter, however hMetis to does not limit the balancing of its results. All of those algorithms would provide very bad partitioning results, if imbalancing was restricted to 0%. In contrast to that, all partitionings calculated with the neighbourhood partitioning algorithm are almost perfectly balanced. As Figure 6.4 shows, the newly introduced algorithm is superior to all existing hypergraph partitioning algorithms, in



(a) Github

(b) Stack Overflow

(c) MovieLens

(d) Reddit

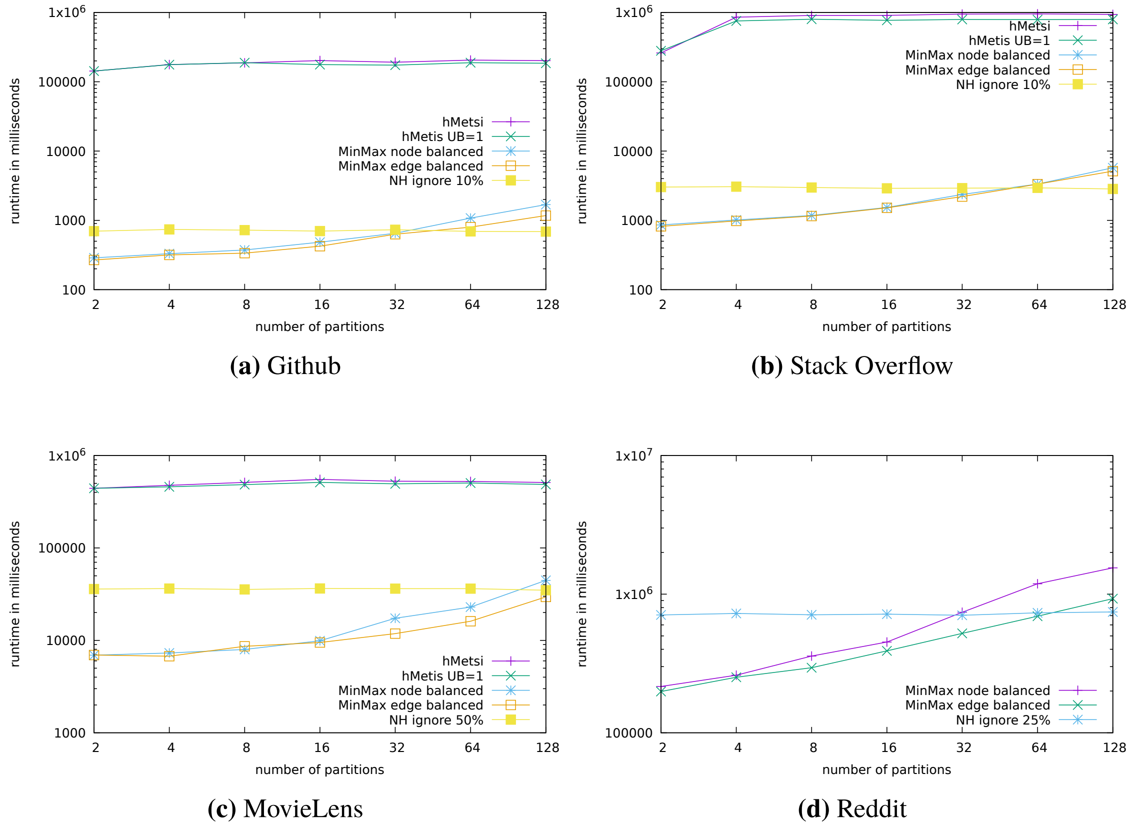
Figure 6.4: Balancing compared between different algorithms on different hypergraphs

terms of node balancing. That said, there are also applications that may benefit from node *and / or* hyperedge balancing [AIV15], which means depending on the application another hypergraph partitioning systems could be superior to the neighbourhood partitioning algorithm.

6.4 Runtime

Runtime is crucial for almost every algorithm. If an algorithm produces perfect results, but has a horrible runtime or does not scale, it is not useable for real world data. Since the *balanced k-way hypergraph partitioning problem* is NP-Hard and computable, an algorithm exists which produces perfect results in the regard to every metric described in section 2.3 except runtime. Briefly the reason why such an algorithm cannot be used, is its disastrous runtime.

Figures 6.5a - 6.5d show how the different hypergraph partitioning algorithms compare to each other regarding their runtime. Considering that the y-axes are logscale with base 10 we can see that hMetis is not competitive compared to the other algorithms at all. For every hypergraph the runtime of both hMetis variants is far worse than the ones of the streaming or neighbourhood partitioning systems. It is remarkable that the figure shows that the neighbourhood partitioning algorithms do scale well concerning the number of partitions calculated. In fact runtime, gets



(a) Github

(b) Stack Overflow

(c) MovieLens

(d) Reddit

Figure 6.5: Runtime compared between different algorithms on different hypergraphs

better with a higher number of partitions calculated. The streaming algorithms do not have this quality which is why the neighbourhood partitioning, even when having a worse runtime than the streaming algorithms for small numbers of partitions, surpasses them for higher numbers of partitions in terms of runtime. This quality makes the runtime of the neighbourhood partitioning algorithm well predictable when a partitioning for the same hypergraph has been calculated before. The billion scale hypergraph introduced in Section 3.2 build from Reddit comments was not used to create plots like the other datasets, since the partitioning this hypergraph takes about 8 hours for both the streaming systems and the neighbourhood partitioning algorithm. It is worth noting that both, the streaming algorithms as well as the neighbourhood partitioning algorithms were able to partition this billion scale hypergraph. Only few partitionings have been calculated for the billion scale hypergraph, but concerning quality the ones calculated with the neighbourhood partitioning algorithm have always been superior.

7 Conclusion

In this thesis hypergraph partitioning with the help of neighbourhood heuristics has been examined. The goal was to apply the idea of the graph edge partitioning algorithm proposed by Zhang et al.[ZWL+17] to hypergraph partitioning to be able to process even billion scale hypergraphs. Since the naive neighbourhood hypergraph partitioning algorithm was not able to scale, this thesis also covered optimizations to make the naive algorithm scale up to even partition billion scale hypergraphs.

As Section 6 shows, the newly proposed algorithm is superior to existing solutions in almost every respect. HMetis may produce better results regarding the *Edge-Cut* metric descriped in 2.3.3, but the new neighbourhood partition algorithm is almost alway better with regard to the *Sum of External Degrees* metric descriped in Section 2.3.4. Even hMetis optimizes per default directly for the *Edge-Cut* metric, tests on small graphs have shown, that when hMetis is called with parameters to optimize partitioning directly concerning the *Sum of External Degrees* metric, it is not able to produce better partitionings than the neighbourhood partitioning algorithm. Considering this our proposed algorithm is able to partition hypergraphs with a better runtime, quality and balancing than exisiting solutions. It is possible to partition huge hypergraphs, even those being billion scale. As mentioned in Section 6, it is possible to process the 2.8 billion node Reddit comment hypergraph descriped in Section 3.2 in less than $24h$, producing however better results than the other algorithms, which were able to process the hypergraph.

Future Work

As said in section 5.3.4, ignoring $X\%$ of the biggest hyperedges during the *neighbhbourhood expansion* improves the quality of the resulting cuts. Finding the perfect number of ignored hyperedges has not been dealt in this thesis. This problem needs to be addressed in future work.

In section 5.3.1 two different heuristics to rank nodes have been discussed, i.d. accumulated edge sizes and number of neighbours. Other than that another heuristic has been discovered meanwhile. The average hyperedge size of the hyperedges the node is connected to. This heuristic is very promising with regard to both runtime and quality of the result. Due to time constraints this has not been explored further.

Bibliography

[17a] *Github network dataset – KONECT*. Apr. 2017. URL: http://konect.uni-koblenz.de/networks/github (cit. on pp. 20, 34).

[17b] *Stack Overflow network dataset – KONECT*. Apr. 2017. URL: http://konect.uni-koblenz.de/networks/stackexchange-stackoverflow (cit. on pp. 19, 20).

[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, 40, 41).

[Gro] GroupLens. *MovieLens*. URL: https://grouplens.org/datasets/movielens/ (cit. on p. 20).

[KAKS99] G. Karypis, R. Aggarwal, V. Kumar, S. Shekhar. "Multilevel hypergraph partitioning: applications in VLSI domain." In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* (1999) (cit. on pp. 24, 25).

[KK00] G. Karypis, V. Kumar. "Multilevel k-way Hypergraph Partitioning." In: *VLSI Design* (2000), pp. 285–300. URL: http://dx.doi.org/10.1155/2000/19436 (cit. on pp. 24, 25).

[MMG+18] C. Mayer, R. Mayer, J. Grunert, A. Tariq, K. Rothermel. "Q-Graph: Preserving Query Locality in Multitenant Graph Processing." In: *Proceedings of the GRADES'18*. ACM. 2018, p. 7 (cit. on p. 13).

[MML17] R. Mayer, C. Mayer, L. Laich. "The tensorflow partitioning and scheduling problem: it's the critical path!" In: *Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning*. ACM. 2017, pp. 1–6 (cit. on p. 16).

[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: *Distributed Computing Systems (ICDCS), 2018 IEEE 38th International Conference on*. 2018 (cit. on p. 16).

[MMTR16] R. Mayer, C. Mayer, M. A. Tariq, K. Rothermel. "GraphCEP: Real-time Data Analytics Using Parallel Complex Event and Graph Processing." In: *DEBS*. Irvine, California, 2016. ISBN: 978-1-4503-4021-2 (cit. on p. 13).

[MTLR16] C. Mayer, M. A. Tariq, C. Li, K. Rothermel. "GrapH: Heterogeneity-aware graph computation with adaptive partitioning." In: *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*. IEEE. 2016, pp. 118–128 (cit. on p. 13).

[MTMR18] C. Mayer, M. A. Tariq, R. Mayer, K. Rothermel. “GrapH: Traffic-Aware Graph Processing.” In: *IEEE Transactions on Parallel and Distributed Systems* (2018) (cit. on p. 13).

[PQD+15] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, G. Iacoboni. “HDRF: Stream-Based Partitioning for Power-Law Graphs.” In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. CIKM ’15. Melbourne, Australia: ACM, 2015, pp. 243–252. ISBN: 978-1-4503-3794-6. DOI: 10.1145/2806416.2806424. URL: http://doi.acm.org/10.1145/2806416.2806424 (cit. on p. 16).

[Stu] Stuck_In_the_Matrix. *Reddit comment dataset*. URL: https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/ (cit. on p. 20).

[ZWL+17] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, Z. Li. “Graph Edge Partitioning via Neighborhood Heuristic.” In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’17. Halifax, NS, Canada: ACM, 2017, pp. 605–614. ISBN: 978-1-4503-4887-4. DOI: 10.1145/3097983.3098033. URL: http://doi.acm.org/10.1145/3097983.3098033 (cit. on pp. 13, 14, 27–29, 43).

All links were last followed on April 22, 2018.

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