DEFRAGMENTING SOCIAL NETWORKS

by

Robert Stephenson Lindquist, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
December 2017

Committee Members:

Dan E. Tamir, Chair

Daniela Ferrero

Mina Guirguis

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Tamir for his tireless efforts over the last few years towards this goal of mine. I would also like to thank Dr. Ferrero and Dr. Guirguis for their help in this process. Finally, I would like to thank Dr. Sherry Lindquist, Ed. D. for her help in grammatical revision.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

The size of modern Online Social Networks requires splitting user data across multiple servers. Viewing these networks as graphs, with users as vertices and friendships as edges, this split is a form of graph partitioning. Studies have shown that high-quality partitioning can improve performance. Consequently, this thesis considers two measures of partitioning quality, the exact solutions to which are NP-complete. The first, Minimum Cut, minimizes the number of edges with vertices on different partitions, known as the edge cut. The second, MIN_REPLICA, gives each master vertex a local replica of each neighbor that is on different partition, and rearranges the vertices to minimize the number of replicas while providing a minimum level of redundancy.

Traditional partitioning algorithms have either failed to take into account the unique structure of social graphs, or the dynamic nature of the graph, resulting in either a large edge cut or a high number of vertex replicas, and most cannot run in a distributed fashion, which is necessary for graphs with billions of vertices.

This thesis paper presents two sets of methods that hybridize existing graph partitioning solutions. Each of these consists of an event-driven local algorithm and a periodic global algorithm, to minimize edge cut or replication overhead while maintaining partitions of roughly equal size and avoid excess inter-partition vertex movement. This is analogous to the approach taken by file systems in using a greedy online algorithm to place a file in a particular location quickly, and periodically seeking a better arrangement during periods of low activity, e.g. when users are asleep.

This thesis finds that hybridizing can reduce the runtime in the Minimum Cut problem, and can improve partition quality in the MIN_REPLICA problem.

1. INTRODUCTION

Online Social Networks, or OSNs, are an area of significant interest [1]. Ranging from professional networks like LinkedIn to the extracurricular networks like Facebook, in the past decade the number and size of these networks has exploded. This rapid growth has brought challenges, particularly the need to scale up infrastructure designed for a single server to run in a distributed fashion; Twitter and Facebook have had to change their architecture several times, while Friendster's failure can be attributed to its inability to scale [2, 3].

Treating social networks as graphs, horizontal scaling is a form of graph partitioning. The primary cost induced by such partitioning is that edge traversals that had previously been local may now cross partitions, leading to either expensive inter-server communication or significant replication overhead.

There are two established problems that address this. The first, Minimum Cut, minimizes the edge cut of the resulting partitions, defined as the number of edges with endpoints in separate partitions. MIN_REPLICA takes a different tack, adding a second set of vertices known as replicas, which are copies of the pre-existing vertices, which it calls masters. MIN_REPLICA minimizes the number of replicas, as long as each master vertex has at least k replicas, and has all of its neighbors on its master partition, either as replicas or masters.

Minimum Cut with similarly-sized partitions is an NP-complete graph problem [4], as is MIN_REPLICA [3]. Numerous heuristics including diffusion-based clustering, divide-and-conquer methods, and schemes that make use of geometric data for the vertices, have been presented for Minimum Cut, [5]. However, the traditional approaches

have not considered some phenomena relevant to OSNs. First, they typically ignore the effects of graph dynamism, which consist of adding or deleting vertices, edges or partitions. Second, they typically ignore the distinct properties of social graphs, which include power-law vertex degree distribution, wherein the number of vertices of degree k, i.e. having k neighbors, is proportional to $k^{-\alpha}, 2 \leq \alpha \leq 3$; a lack of well-defined large clusters of 100 vertices or more; and assortivity, wherein the degree of a vertex is positively correlated with the degrees of its neighbors [6].

The problem addressed in this thesis is finding heuristic solutions to Minimum Cut and MIN_REPLICA for dynamic graphs modeled after social networks, with minimal vertex movement. This thesis hypothesizes that by making regular, greedy decisions on vertex placement following befriending, and then periodically performing a more complex repartitioning, these solutions can maintain an edge cut or replication overhead significantly smaller than current solutions. This is analogous to the defragmentation process that file systems use to compensate for undesirable qualities of the greedy algorithms they use to place files initially.

This thesis present four hybrid solutions that take elements from both Pujol et al.'s SPAR and a distributed graph partitioning algorithm. The two minimum cut solutions are Jabar, which adds SPAR's frequent rebalancing to Ja-be-Ja's global repartitioning, and Hermar, which similarly adds SPAR-style rebalancing to Hermes. The two MIN_REPLICA solutions are Spaja, which adds Ja-be-Ja's simulated annealing algorithm to SPAR, and Sparmes, which adds Hermes's Lightweight Repartitioner to SPAR.

The results show that Jabar improves upon Ja-be-Ja while reducing the cost considerably, Spaja improves upon SPAR moderately at a corresponding increase in cost, and that Sparmes improves upon SPAR dramatically at a moderate cost. This demonstrates that some forms of hybridization can improve partition quality without increasing cost, suggesting that hybridization is a field worth pursuing.

The rest of the thesis is structured as follows. Chapter 2 provides background information. Chapter 3 reviews the existing literature. Chapter 4 presents a detailed explanation and analysis of each of the selected or hybrid solutions and the experimental setup. Chapter 5 shows the experimental results. Chapter 6 evaluates those results. Chapter 7 presents conclusions and areas for future work.

2. BACKGROUND

Graph Theory

Types of Graphs

A graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ consists of a non-empty set of vertices $\mathbb{V}$, and a set of edges $\mathbb{E}$ joining pairs of vertices. For example, finite state machines interpret $\mathbb{V}$ as a set of states, and $\mathbb{E}$ as the transitions between them.

In general, a graph may have multiple, distinct edges between any two vertices u, v, or between v and itself, and each edge can have a numerical value, such as a weight, associated with its traversal. A graph is simple if it has at most one unordered, unweighted edge between any pair of distinct vertices u, v, and no edges from v to itself, known as loops. Thus, in simple graphs, an edge e is uniquely defined by its endpoints, and named uv. Simple graphs are rich enough to model a wide variety of phenomena in social networks, so this thesis focuses on them.

Connectivity

The vertices $u, v \in \mathbb{V}$ are neighbors if $uv \in \mathbb{E}$, denoted $u \leftrightarrow v$. The neighborhood of a vertex v is the set of all neighbors of v, denoted as $nbhd(v)$, and the degree of v is the cardinality of the set $nbhd(v)$ and it is written as $deg(v)$. A graph $\mathbb{G}$ is connected when for each pair of different vertices u and v there exists a path from u to v in $\mathbb{G}$.

A path p from vertex u to vertex v is a sequence of edges $uw, wx, \ldots, zv$ where each edge shares a vertex with its predecessor, and $paths(u, v)$ is the set of all paths from u to v. The length of p, written $len(p)$, is the number of edges along p.

$minPaths(u, v)$ is the subset of $paths(u, v)$ where $len(p)$ is minimum. Similarly, $dist(u, v)$ is the aforementioned minimum $len(p)$ across $paths(u, v)$, also known as the distance.

Partitioning and Edge Cuts

Let k be a nonnegative integer and $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ be an undirected simple graph. A k-partition of $\mathbb{G}$ is a set P of k nonempty subsets of $\mathbb{V}$ with no overlap and full coverage. That is, $P = \{V_1, V_2, \ldots, V_k\}$ such that $V_1 \cup V_2 \cup \ldots \cup V_k = \mathbb{V}$, and that $\forall i, j, 1 \leq i < j \leq k, V_i \cap V_i = \emptyset$. Each V_i is called a partition, and the notation $partition(v) = i$ indicates that $v \in V_i$. The weight of a partition V_i is the number of vertices it contains, written $|V_i|$. The imbalance of a given partition V_i is its weight divided by the average partition weight, or $|V_i|/(|\mathbb{V}|/k)$.

The edge cut of P, written $cut(P)$, is the set of edges $uv\ s.t.\ partition(u) \neq partition(v)$. The cross-partition cut $cut(V_i, V_j)$ is the set of all edges $uv \in \mathbb{E}\ s.t. partition(u) = i \wedge partition(v) = j$. A low $|cut(V_i, V_j)|$ indicates that P preserves the structure of $\mathbb{G}$ well.

The Minimum Edge Cut Problem

Let k be a positive integer greater than one, and $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ be an undirected simple graph. Find a partitioning P with partition sizes no more than γ times the average and the smallest possible edge cut. That is, find $\min|uv \in \mathbb{E}\ s.t. partition(u) \neq partition(v)|$ subject to the constraint that $\forall V_i \in P, |V_i| < \gamma * |\mathbb{E}|/k$.

The MIN_REPLICA Problem

Let k be a nonnegative integer, M be an integer greater than 1, and $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ be an undirected simple graph. Find M-partition P, and $R(v) \to \{r_1, r_2, \ldots, r_i\}, v \in \mathbb{V}, 1 \leq r_j \leq M$, known as the *replicas* of v, where $\sum_{v \in \mathbb{V}} |R(v)|$ is minimum, subject to the constraints that $\forall V_i \in P, \lfloor \mathbb{E}/M \rfloor \leq |V_i| \leq \lceil \mathbb{E}/M \rceil$, known as *master distribution*, $|R(v)| \geq k$, known as *k-redundancy*, and $\forall v \in \mathbb{V}, \forall u \in nbhd(v), partition(v) \in R(u) \cup \{partition(u)\}$, known as *neighbor-collocation*.

Graph Partitioning Algorithms

Overview

Numerous algorithms for graph partitioning are defined in the literature [5]. Most of these are centralized, meaning that they require a full, global view of the state of the graph and must be run on a single machine or suffer a performance hit for nonlocal access. A growing number, though, are decentralized, allowing them to be run on multiple machines with little to no handicap.

Some of these algorithms create a new partitioning from scratch, while others incrementally improve an existing partitioning. The former are partitioners, while the latter are repartitioners, emphasizing their awareness of the preexisting partitions.

While graph partitioning can reduce the cost of operating a system, partitioning itself incurs a cost when moving vertices from one partition to another, which requires costly inter-server communication. This movement is either physical, which involves sending a vertex and all its data to a new partition, or logical, in which only some of the

data is moved, typically during the repartitioning process. Because physical movement is the easiest to compare across different algorithms, it is the focus of the cost estimates. Below are several relevant graph algorithms from the literature; see Appendix F for pseudocode and algorithms.

Kernighan-Lin Partition Improvement

Kernighan-Lin, abbreviated KL, is a centralized repartitioner that iteratively improves partitions by exchanging vertex pairs between partitions to reduce edge cut [7]. In its simplest form, it considers a graph with a 2-partition, known as a bipartition.

The fundamental operation of KL is the exchange, or swap, of two vertices $a \in A, b \in B$, which moves a to B, and b to A, and can reduce the overall edge cut. KL refers to the gain, which is the reduction in edge cut from a single swap, formulated as $D(a) + D(b) - 2{\times}c(a,b)$. Here, $D(a) = |nbhd(a) \cap B| - |nbhd(a) \cap A|$, $D(b) = |nbhd(b) \cap A| - |nbhd(b) \cap B|$, and $c(a,b) = 1$ if $ab \in E$; $c(a,b) = 0$ otherwise. In the case of weighted graphs, $c(a,b)$ replaces 1 with the weight of the edge ab.

Each iteration of KL finds vertices $a \in A, b \in B$ such that swapping a and b produces the greatest gain across all unmarked vertices. Next, KL appends a to list a_v, b to b_v, and their gain to g_v. This is a form of logical movement, in that KL has reassigned vertices without physically moving them. It marks them and does not visit them again.

Once it marks all vertices, it finds the shortest subsequence $g_v[1..i]$ with maximal sum, and swap $a_v[1..i]$ and $b_v[1..i]$, a physical movement cost of 2i. Note that by not terminating at the first nonpositive element of g_v, KL avoids some local optima.

Fidducia-Mattheyses Partition Improvement

Fidducia-Mattheyses, abbreviated FM, is a newer centralized repartitioner that makes two major changes to KL [9]. First, FM relaxes the strict balance requirements of KL, allowing partitions to be up to γ times the average partition weight. Second, whereas KL swaps pairs of vertices across partitions, FM moves individual vertices across partitions. As with KL, FM focuses on the bipartition case.

Each iteration of FM finds the unmarked vertex v with the most to gain from moving to the other partition. FM defines $gain(a), a \in A$ the same as $D(a)$ in KL, except that it equals $-\infty$ if B would overweight, that is $|B| + 1 > \gamma \times ((|A| + |B|)/2)$. FM marks v and records its gain in the list $GAIN$. As with KL, this is logical movement.

Once FM has marked all of the vertices, it finds the smallest subsequence $GAIN[1..i]$ with maximal sum and physically moves the corresponding vertices to the other partitions, resulting in a physical movement cost of i.

Multi-Level Schemes

If a human were to partition a graph, she might zoom out, partition at a gross level, and then zoom in to refine the partition. Multi-level schemes formalize that process, starting with a coarsening phase, contracting the graph by merging neighboring vertices while maintaining their original connectivity [5]. Next, they partition the coarse graph. Finally, they expand the contracted vertices and refine the partition via KL/FM or another similar partition improvement method.

One particularly successful implementation of such schemes is the METIS family of algorithms, which includes the distributed paraMETIS [10]. Like KL and FM, METIS

is centralized, though it differs in that it ignores preexisting partitions, and is thus a partitioner, not a repartitioner.

Ja-be-Ja Partitioning

Ja-be-Ja is a decentralized graph partitioning algorithm that generates new partitions from scratch [11]. It uses a KL-type algorithm with simulated annealing to get around local optima, allowing swaps of vertices resulting in slightly worse edge cut [11].

Prior to the first iteration, Ja-be-Ja randomly assigns each vertex a logical partition. This initial, random partitioning is what makes Ja-be-Ja a partitioner instead of a repartitioner: the partitioning it iteratively improves is not the current partitioning. Moreover, because Ja-be-Ja uses multiple restarts, it randomly repartitions before each restart. Additionally, it sets the temperature of the iteration to the initial temperature before each restart, and decrease it by a fixed ΔT after each iteration, until it drops to 1.

In each iteration, Ja-be-Ja looks for a swap partner for each vertex a with the highest gain relative to the temperature. Because it uses simulated annealing, however, it uses a somewhat different gain formula from KL. Let x_{ab} be the number of neighbors of a on b's logical partition, T_r be the temperature of the iteration, and α be power to which it raises the comparison function. The gain is $\left(T_r \times ({x_{ab}}^{\alpha} + {x_{ba}}^{\alpha})\right) - ({x_{aa}}^{\alpha} + {x_{bb}}^{\alpha})$. Note that if T_r is 1 and α is 1, this is quite similar to KL's formula.

For performance reasons, Ja-be-Ja does not consider every potential swap partner of a. Instead, it uses a partner selection policy that considers a fixed number k of candidates. The local partner selection policy considers k vertices on a's physical partition, the random policy looks for k candidates on any partition, and the hybrid policy

first performs the local policy, and if no suitable partner is found proceeds to the random policy; Ja-be-Ja prefers local partners because their operations are less expensive. Regardless of the policy, Ja-be-Ja will only swap if the best partner has a positive gain.

Ja-be-Ja terminates as soon as the algorithm has converged, and no more swaps occur. Because it allows multiple restarts, however, it may repeat the process several times, and choose the resulting partitioning with the lowest edge cut across all restarts, if it improves upon the initial partitioning.

Hermes's Lightweight Repartitioner

Hermes's Lightweight Repartitioner is a distributed, FM-type repartitioner that maintains a small amount of auxiliary data [12]. That is, each vertex "knows" how many neighbors it has in each partition, the weight of each partition, and the total weight of the graph as a whole. Like FM, it uses an imbalance constraint γ, and it also defines k as the maximum number of vertices that can move from a partition in one iteration stage. As in FM, the gain of vertex a moving from partition A to$B$ is how many more neighbors a has in partition B than in A.

In each iteration, Hermes asks each partition for vertices with the most to gain by moving to other partitions. To avoid oscillation, Hermes breaks each iteration into two stages, and asks for at most k candidates per partition per stage. In the first stage, vertices can only move to partitions with larger partition IDs, and vice versa in the second stage. At the end of each stage, Hermes logically migrates the chosen vertices and updates the auxiliary data in each partition. Because each stage executes independently on each partition, which is why Hermes is a distributed repartitioning algorithm.

As noted above, each partition may provide up to k candidates per stage. How many it does provide depends on the imbalance of the partition and whether the gains are positive. Overweight partitions provide the top k candidates regardless of whether their gain is positive. Underweight partitions provide no candidates. Properly-weighted partitions return the top k candidates that have positive gain.

Hermes keeps iterating until convergence, at which point no partition offers any candidates in either stage. Once it converges, Hermes physically moves each vertex to its logical partition. Note that convergence is not guaranteed, and in correspondence with Nicoara, he confirmed a particular adversarial case.

Social Networks

Definition

A social network can be represented as a graph $\mathbb{G}$ where $\mathbb{V}$ represents the users, and $\mathbb{E}$ the friendships between them [13]. Depending on the network, $\mathbb{E}$ may be directed, as in following in Twitter, or undirected, as in friendship in Facebook.

Power Law Degree Distribution

Studies indicate that the distribution of $deg(v)$ follows power law $p_k \propto k^{-\alpha}, 2 \leq \alpha \leq 3$, where p_k is the number of vertices with degree k [13, 14]. This yields a long tail with an appreciable number of vertices of very high $deg(k)$. This can lead to small-world properties; i.e.,$avg\big(dist(u,v)\big)$ is quite short, as shown in Milgram's six degrees of separation, where randomly-selected people were often separated by no more than six relationships, e.g. a path of length ≤ 6 [15].

Assortivity

Further studies of social networks show that $deg(v)$ is positively correlated with $avg\big(deg(u)\big), u \in nbhd(v)$, known as assortivity [13]. For example, researchers with many collaborations tend to collaborate with others with many, while those with few work with others with few. This may be unique to social networks.

Lack of Well-Defined Clusters

Evidence suggests that social networks lack well-defined large clusters [6]. While Leskovec et al. found that tight clusters exist up to a size of about 100 vertices, seeking clusters beyond that size results in a "roughly inverse relationship between community size and optimal community quality" [6, p1].

Graph Dynamism

Numerous algorithms treat graphs as static objects, where vertices may not be added or deleted, nor may edges or partitions. However, partitioning schemes for social graphs must consider these graphs' dynamic nature to avoid suboptimal performance. Furthermore, while vertex movement is a concern in any iterative partitioning algorithm, it is especially problematic in dynamic graphs. Note that the literature sometimes refers to oscillation, which is the movement of vertices back and forth between partitions, instead of vertex movement.

Expected All-Neighbors Query Delay

Online social networks frequently query all 1-hop neighbors of a particular vertex, such as when a user logs in and sees all of her friends' updates, or when an advertiser wishes to determine whether any of a user's friends like a particular movie or lifestyle product for cross-marketing. Because of its cost and ubiquity, it can be useful to model its expected delay. Andrew Charneski presents a number of models over at the Simia Cryptus blog; of particular interest is the log-normal distribution [17].

The delay of any multi-request is simply the maximum delay of all of the subrequests. In many cases, the size of the request and response are immaterial, so the delay of any request is indistinguishable from the ping delay. Thus, the log-normal distribution provides an estimate of that delay by returning the maximum of k random draws from the distribution. Performing this estimate across all vertices in the system, taking the arithmetic mean of the results, provides an acceptable approximation of the expected value of any all-neighbors request.

It is worth noting that, because the delay is determined by the maximum subdelay, 1-hop queries hitting fewer partitions tend to be faster, regardless of the edge cut. To demonstrate, let vertices u, v on partition 1 each have twenty neighbors on other partitions. If u has one neighbor each on partitions 2-21, and v has all twenty neighbors on partition 2, the delay for u would likely be greater, even though the same number of edges are cut. As a result, minimizing the edge cut does not necessarily minimize expected delay, so experiments are necessary to show that the minimum cut problem is an acceptable proxy for query delay.

Neighbor Collocation

MIN_REPLICA reduces delay by replicating the neighbors of a vertex that are in separate partitions [3]. In practice, Benevenuto et al. show that "80% of all accesses remain within a 1-hop neighborhood in the social network", so neighbor collocation reduces inter-server traffic while replicating only a small fraction of the graph [16, p11]. Furthermore, it helps distributed programming resemble the single-server case [3].

Newman's Models of Social Networks

As Newman states, "There is empirical evidence that clustering in networks arises because the vertices are divided into groups [19, 20], with a high density of edges between members of the same group, and hence a high density of triangles, even though the density of edges in the network as a whole may be low." [18, p1]

Accordingly, his model consists of five parameters:

- N: the number of vertices
- M: the number of groups
- p: the probability that two vertices within a given group are neighbors.
- r_m : A probability distribution for the number of groups for a vertex. For example, the probability that a vertex is in three groups is r_3.
- s_n: A probability distribution for the number of vertices in a particular group. For example, the probability that a group contains 5 vertices is s_5.

This thesis uses a simple probability distribution: that the probability any given vertex is in any given group is q. Formally, this produces a binomial distribution for r_m centered about $q \times M$, and another binomial distribution for s_n centered about $q \times N$.

SPAR's Model of Social Network Middleware

Introduction

In Pujol et al.'s view, there are six high-level operations that a social network middleware solution must perform [3]:

- Add_User(u)
- Remove_User(u)
- Befriend(u, v)
- Unfriend(u, v)
- Add_Partition(p)
- Remove_Partition(p)

To these, this thesis adds a seventh that appears to be implicit in some of the papers to more effectively test the central thesis.

- Downtime()

This thesis's implementations of various solutions all adhere to this high-level interface to facilitate comparisons between them.

When adding master or replica vertices to the system, the most straightforward option is a water-filling strategy. Much like water fills the lowest places, so too does a water filling strategy place vertices on the partition with the lowest number of master vertices.

Of the algorithms defined in this section, only SPAR has a fully-articulated plan for each of the major operations, with the exception of downtime. Table 1 formalizes SPAR's actions in each of these cases; see the Solutions section for an explicit explanation of the bare minimum operations.

SPAR's Actions

Table 1 – SPAR's Actions

Operation	Action
Add_User(u)	Add u to the partition with the fewest masters
Remove_User(u)	Bare Minimum
Befriend(u, v)	Runs the Befriending algorithm below
Unfriend(u, v)	Runs the Unfriending algorithm below
Add_Partition(p)	Bare Minimum
Remove_Partition(p)	Runs the Partition Removal algorithm below
Downtime()	No Action

Pujol et al.'s explanation of SPAR's major components is straightforward enough to reproduce below. Note that they use the terms "node" instead of "vertex" and "server" instead of "partition". Finally, note that, as a solution to the Min_Replica problem, SPAR guarantees a minimum redundancy of k for some nonnegative integer k.

SPAR's Edge Addition [Befriending Algorithm]

> When a new edge is created between nodes u and v, the algorithm checks whether both masters are already co-located with each other or with a master's slave. If so, no further action is required.
> If not, the algorithm calculates the number of replicas that would be generated for each of the three possible configurations: 1) no movements of masters, which maintains the status-quo, 2) the master of u goes to the partition containing the master of v, 3) the opposite.
> Let us start with configuration 1). Here, a replica is added if it does not already exist in the partition of the master of the complementary node. This may result in an increase of 1 or 2 replicas depending on whether the two masters are already present in each other's partitions. This can occur if nodes v or u already have relationships with other nodes in the same partition or if there already exist extra slaves of v or u for redundancy.
> In configuration 2), no slave replicas are created for u and v since their masters will be in the same partition. However, for the node that moves, in this case u, we will have to create a slave replica of itself in its old partition to service the master

> of the neighbors that were left behind in that partition. In addition, the masters of these neighbors will have to create a slave replica in the new partition – if they do not already have one– to preserve the local semantics of u. Finally the algorithm removes the slave replicas that were in the old partition only to serve the master of u, since they are no longer needed. The above rule is also subject to maintaining a minimum number of slave replicas due to the K redundancy: the old partition slave will not be removed if the overall system ends up with less than K slaves for that particular node. Configuration 3) is complementary to 2). The algorithm greedily chooses the configuration that yields the smallest aggregate number of replicas subject to the constraint of load-balancing the master across the partitions. More specifically, configuration 2) and 3) also need to ensure that the movement does not cause load unbalancing. That is, this movement either happens to a partition with fewer masters, or to a partition for which the savings in terms of number of replicas of the best configuration to the second best one is greater than the current ratio of load imbalance between partitions. [3, p378]

SPAR's Edge Removal [Unfriending Algorithm]

> When an edge between u and v disappears, the algorithm removes the replica of u in the partition holding the master of node v if no other node requires it, and vice-versa. The algorithm checks whether there are more than K slave replicas before removing the node so that the desired redundancy level is maintained. [3, p379]

SPAR's Partition Removal [Server Removal Algorithm]

> When a server is removed, whether intentionally or due to a failure, the algorithm re-allocates the N/M master nodes hosted in that server to the remaining $M - 1$ servers equally. The algorithm decides the server in which a slave replica is promoted to master, based on the ratio of its neighbors that already exist on that server. Thus, highly connected nodes, with potentially many replicas to be moved due to local data semantics, get to first choose the server they go to. The remaining nodes are placed wherever they fit, following simple water-filling strategy. [3, p379]

A Note on the Necessity of the Partition Removal Algorithm

The purpose of the server removal algorithm is not to prevent data loss, but to improve data availability. Given that a commercial OSN likely has a high-quality data backup solution, it should be possible to restore the full state of that server from disk,

after provisioning a new server. However, as restoring from disk is quite slow, it can reduce the availability of that server's data for a significant amount of time. Thus, it can be much faster to promote replicas to masters, as the whole operation occurs in memory.

A Note on Data Consistency

Any system with data replication must ensure consistency. SPAR accomplishes this with a Replication Manager that must "propagate the writes that take place on a user's master to all her [replicas] using an eventual consistency model, which guarantees that all the replicas will – with time – be in sync." [3, p383]. The authors further note that having a single master eliminates additional sources of inconsistencies [3].

3. RELATED WORK

Several solutions have been proposed to handle partitioning of dynamic graphs, but few of them tackle both the dynamism and unusual characteristics of social graphs while maintaining low edge cut, and those that do could be improved. This thesis presents some that form a foundation upon which it shall build.

Pujol et al. present SPAR, a middleware for handling horizontal graph scaling through partitioning and replication [3]. They eschew clever high-level partitioning, as they believe current algorithms fail to support incremental, or dynamic operation, are not robust to input conditions for clustering/community detection, and that minimizing the cut may be counterproductive. SPAR generally enforces strict balance requirements, but does allow deviation from hill-climbing by permitting vertices to migrate to higher-loaded partitions when, "[T]he savings in terms of number of replicas ... is greater than the current ratio of load imbalance between partitions." While SPAR is effective, its avoidance of high-level partitioning leaves room for substantial reductions in edge cut, possibly with little additional overhead.

Mondal and Deshpande make greedy decisions about migrating vertices at the cluster level [19]. Their work is effective, but by making a series of locally-optimal choices without regard to the global optimum, they risk getting stuck in local optima. Additionally, their approach assumes relatively well-defined large clusters, which is generally not applicable to social graphs [6].

Rahimian et al. present Ja-be-Ja, meaning swap in Persian, a distributed approach to large-scale balanced graph partitioning [11]. Its partitions are independent, self-governing processes that can swap vertices with their neighbors greedily, akin to KL. Ja-

be-Ja enforces strict partition balancing and avoids local optima by using simulated annealing to allow temporary deviations from optimality. They compare their strategy to METIS and other well-established algorithms, and find that it works well on various graphs, including social networks. They do not, however, attempt to handle dynamism or make any small-scale, low-complexity improvements.

Vaquero et al. present their lightweight repartitioner, which uses an FM-type distributed algorithm for iterative partitioning [20]. They migrate individual vertices to the partition with maximum gain each iteration, and avoid oscillation by making vertices roll the dice before migrating. They enforce balance constraints by making pessimistic assumptions about balancing, allocating only $remaining\_capacity/k$ possible moves between any pair of the k partitions. Their results are impressive, but they do not consider partition addition or deletion, nor do they take a global view of the system.

Nicoara et al. present Hermes, which differs from Vaquero et al. algorithm in its balance and oscillation-avoidance techniques, as well as its ability to handle vertex-weighted graphs [12]. It avoids oscillation by allowing vertices to move only to partitions with higher IDs in odd iterations, and then to those with lower IDs during even ones. Hermes explicitly maintains a small amount of information about neighboring vertices for balance constraints, decrying more global views as requiring "information ... proportional to the [number of edges] of the graph." Hermes avoids local optima by allowing a specific imbalance factor γ between partitions. As with Vaquero et al. work, their results are promising, but they do not handle partition deletion/addition, and they could benefit from adding frequent, low-complexity optimizations.

4. METHODOLOGY

Solutions Overview

The solutions developed in this thesis promote efficient OSN operation through maximizing partition quality and minimizing vertex movement, all without full knowledge of the graph, meaning they are decentralized. The minimum cut problem defines quality as a low edge cut, while the MIN_REPLICA problem defines quality as a low number of replicas.

The solutions to each of these problems fall into one of four categories:

- Dummy solutions that do the bare minimum to maintain a system that obeys the problem constraints. These provide a minimum performance baseline that any solution should be able to beat.
- State-of-the-art centralized solutions. These are impractical for large systems, but provide a good standard of comparison.
- Decentralized solutions previously described in the literature. These provide a goal for the hybrid solutions to beat. See Appendix B for differences between the original papers and the implementations.
- Hybrids of existing solutions, which are the primary contribution.

Please note that this thesis simulates the behavior of these solutions, rather than implementing them through a distributed graph database like Neo4j. The scale of OSN middleware systems means that testing and implementation of such a distributed system requires the use of dozens of rack-mounted or cloud-based servers. Unfortunately, rack-mounted systems are prohibitively expensive, and all common cloud-based system treat

persistent storage as one shared pool, which precludes individual partitions from having their own set of vertices.

Thus, simulation was the only viable option. As a result, the results do not show the actual performance overhead of the solutions, only approximate it though counting vertex movement and estimating all-neighbors query delay.

Minimum Cut Solutions Details

Summary

- Dummy: Dummy. Performs no optimizations, and redistributes vertices upon partition failure using a water-filling strategy.
- METIS: Centralized [10]. Frequently repartitions the graph from scratch using the METIS algorithm.
- Ja-be-Ja: From the literature [11]. Runs Ja-be-Ja's simulated annealing partitioning algorithm upon downtime.
- Hermes: From the literature [12]. Runs Hermes's Lightweight Repartitioner upon vertex addition.
- Jabar: Hybrid. Adds SPAR-like rebalancing upon befriending to on Ja-be-Ja, and the partitioning is incremental and runs only one iteration.
- Hermar: Hybrid. Adds SPAR-like rebalancing upon befriending to Hermes.

Bare Minimum Actions

This thesis considers the actions in Table 2 to be the bare minimum for maintaining a correct, coherent, consistent system. Note that the "lightest partition" has the fewest vertices.

Table 2 – Bare Minimum Actions for Minimum Cut Solutions

Operation	Action
Add_User(u)	Register u in the system and place on the lightest partition
Remove_User(u)	Unfriend all of u's neighbors, remove it from its partition, and remove it from the system
Befriend(u, v)	Add u to v's neighbors list, and vice versa
Unfriend(u, v)	Remove u from v's neighbors list, and vice versa
Add_Partition(p)	Register p in the system, but do not add anything to it
Remove_Partition(p)	Migrate all vertices on p to the lightest partition at the time, and remove p from the system
Downtime()	Do nothing

Dummy

Table 3 – Dummy's Actions

Operation	Action
Add_User(u)	Bare Minimum
Remove_User(u)	Bare Minimum
Befriend(u, v)	Bare Minimum
Unfriend(u, v)	Bare Minimum
Add_Partition(p)	Bare Minimum
Remove_Partition(p)	Bare Minimum
Downtime()	No Action

As stated in Table 3, dummy literally does bare minimum necessary to meet the criteria. For example, when two vertices become neighbors, it simply updates each vertex to include the other's ID in its neighbor list.

METIS

Table 4 – METIS's Actions

Operation	Action
Add_User(u)	Bare Minimum
Remove_User(u)	Bare Minimum
Befriend(u, v)	Run the METIS repartitioning algorithm, 10% of the time
Unfriend(u, v)	Bare Minimum
Add_Partition(p)	Bare Minimum
Remove_Partition(p)	Bare Minimum
Downtime()	Run the METIS partitioning algorithm

Ja-be-Ja

Table 5 – Ja-be-Ja's Actions

Operation	Action
Add_User(u)	Bare Minimum
Remove_User(u)	Bare Minimum
Befriend(u, v)	Bare Minimum
Unfriend(u, v)	Bare Minimum
Add_Partition(p)	Bare Minimum
Remove_Partition(p)	Bare Minimum
Downtime()	Run Ja-be-Ja's Simulated Annealing Partitioning Algorithm

Hermes

Table 6 – Hermes's Actions

Operation	Action
Add_User(u)	Run Hermes's Lightweight Repartitioner
Remove_User(u)	Bare Minimum
Befriend(u, v)	Bare Minimum
Unfriend(u, v)	Bare Minimum
Add_Partition(p)	Bare Minimum
Remove_Partition(p)	Bare Minimum
Downtime()	No Action

Jabar

Table 7 – Jabar's Actions

Operation	Action
Add_User(u)	Bare Minimum
Remove_User(u)	Bare Minimum
Befriend(u, v)	Rebalance in the SPAR Style (see below)
Unfriend(u, v)	Bare Minimum
Add_Partition(p)	Bare Minimum
Remove_Partition(p)	Bare Minimum
Downtime()	Run a single-start, incremental version of the Ja-be-Ja Partitioning algorithm (see below)

Jabar befriending uses a SPAR-style model to determine whether to migrate either u or v if they are on different partitions. Because Ja-be-Ja does not allow imbalance, the Jabar algorithm swaps pairs of vertices instead of moving individual ones. For each of u, v, Jabar randomly selects k vertices on the other partition and uses a modified version of the FIND_PARTNER algorithm to find the best swap candidate. If both find a suitable candidate with positive gain, the one with the greatest gain is selected for a swap. If only one has a suitable candidate with positive gain, that is chosen for a swap. Finally, if no suitable candidates are found with positive gain, no swap occurs. For an explicit algorithm, see Appendix A.

Jabar repartitioning differs from Ja-be-Ja's method as follows. First, it takes only a single iteration of the partitioning algorithm. While it could have multiple restarts, preliminary experimental results for this thesis showed that without the random initialization, additional iterations provided little to no benefit, and were quite expensive. Second, instead of initializing partitions to random values, Jabar keeps them at each vertex's current partition, which allows for incremental partitioning; this makes Jabar a

repartitioner, rather than a partitioner. Finally, Jabar use Ja-be-Ja's random partner selection method instead of the hybrid model. Because Jabar does not assign logical partitions randomly, the initial logical partitions are the same as the physical ones. Thus, it would take significant movement before much difference arises between logical and physical partitions, reducing any advantage of seeking local partners.

Hermar

Table 8 – Hermar's Actions

Operation	Action
Add_User(u)	Run Hermes's Lightweight Repartitioner
Remove_User(u)	Bare Minimum
Befriend(u, v)	Rebalance in the SPAR Style (see below)
Unfriend(u, v)	Bare Minimum
Add_Partition(p)	Bare Minimum
Remove_Partition(p)	Bare Minimum
Downtime()	No Action

Hermar's befriending algorithm is based on SPAR's. It calculates the gain that u would experience by moving to v's partition, and vice versa, as defined in the Lightweight Repartitioner. Whichever vertex has greater gain is migrated to the other's partition, if the gain is positive. For an explicit algorithm, see Appendix A.

MIN_REPLICA Solutions Details

Summary

- RDUMMY, short for Replica Dummy: Dummy. Does the bare minimum.
- RMETIS, short for Replica METIS: Centralized [10]. Frequently repartitions the graph from scratch using the METIS algorithm and generates new replicas to meet k-replication and neighbor-collocation requirements.
- SPAR: From the literature [3]. Takes a light approach to management, optimizing upon befriending, where it considers moving each vertex to the partition of the other, and upon partition removal, where it attempts to move vertices to partitions where they already have a replica and neighbors.
- Spaja: Hybrid. Adds Jabar-style repartitioning upon downtime to SPAR.
- Sparmes: Hybrid. Adds Hermes-style repartitioning upon downtime to SPAR.

Bare Minimum Operations

This thesis considers the actions in Table 9 to be the bare minimum for maintaining a correct, coherent, consistent system with k-replication and neighbor collocation. Note that the lightest partition has the fewest masters.

Table 9 – Bare Minimum Actions for MIN_REPLICA Solutions

Operation	Action
Add_User(u)	Register u in the system, place u on the lightest partition, add k replicas to random partitions.
Remove_User(u)	Unfriend all of u's neighbors and their replicas, remove all of u's replicas, remove u from its partition, and remove u from the system.
Befriend(u, v)	Add u to v's neighbor list and those of v's replicas, and vice versa.
Unfriend(u, v)	Remove u from v's neighbor list and those of v's replicas, and vice versa. If u and v are on separate partitions, remove replicas of each from the other's partition if k-replication and neighbor collocation allow it. Note that this is the SPAR unfriending algorithm.
Add_Partition(p)	Register p in the system, but do not add anything to it.
Remove_Partition(p)	For each u on p, select the lightest partition where u has a replica, if one exists, and promote that replica to a master. If the minimum number of replicas is 0, a partition may need to be selected at random to receive a new copy. Add replicas as necessary to achieve neighbor collocation. Add replicas as necessary to achieve k-redundancy.
Downtime()	Do nothing.

Replica Dummy (RDUMMY)

Table 10 – RDUMMY's Actions

Operation	Action
Add_User(u)	Bare Minimum
Remove_User(u)	Bare Minimum
Befriend(u, v)	Bare Minimum
Unfriend(u, v)	Bare Minimum
Add_Partition(p)	Bare Minimum
Remove_Partition(p)	Bare Minimum
Downtime()	No Action

Replica METIS (RMETIS)

Table 11 – RMETIS's Actions

Operation	Action
Add_User(u)	Bare Minimum
Remove_User(u)	Bare Minimum
Befriend(u, v)	Run the RMETIS partitioning algorithm 10% of the time. See below for a description of the algorithm.
Unfriend(u, v)	Bare Minimum
Add_Partition(p)	Bare Minimum
Remove_Partition(p)	Runs SPAR's Partition Removal algorithm
Downtime()	Run RMETIS partitioning algorithm

The RMETIS Repartitioning Algorithm has four steps:

- Drop all replicas to save memory.
- Run METIS to generate a new partitioning from scratch
- Add replicas as necessary to fulfill neighbor collocation constraint
- Add replicas as necessary to fulfill k-replication constraint

Note that the first step technically violates both k-replication and neighbor collocation; as a result, RMETIS does not strictly meet the requirements of MIN_REPLICA. However, given that RMETIS is not a proposed solution, but rather a high-performing centralized solution against which the others can be compared, this tradeoff does not affect the correctness of the hybrid results.

SPAR

Table 12 – SPAR's Actions

Operation	Action
Add_User(u)	Bare Minimum
Remove_User(u)	Bare Minimum
Befriend(u, v)	Runs SPAR's Befriending algorithm
Unfriend(u, v)	Runs SPAR's Unfriending algorithm
Add_Partition(p)	Bare Minimum
Remove_Partition(p)	Runs SPAR's Partition Removal algorithm
Downtime()	No Action

For details, see the Background section.

Spaja

Table 13 – SPAJA's Actions

Operation	Action
Add_User(u)	Bare Minimum
Remove_User(u)	Bare Minimum
Befriend(u, v)	Runs SPAR's Befriending algorithm
Unfriend(u, v)	Bare Minimum
Add_Partition(p)	Bare Minimum
Remove_Partition(p)	Runs SPAR's Partition Removal algorithm
Downtime()	Run the Spaja repartitioning algorithm. See the description below.

The Spaja repartitioner differs from Ja-be-Ja's partitioning algorithm as follows. In addition to the incremental-partitioning and single-start changes mentioned in Jabar, Spaja replaces the edge-cut scoring method with a replication scoring method, which is a nontrivial change. For an explicit algorithm, see Appendix A.

Sparmes

Table 14 – SPARMES's Actions

Operation	Action
Add_User(u)	Bare Minimum
Remove_User(u)	Bare Minimum
Befriend(u, v)	Runs SPAR's Befriending algorithm
Unfriend(u, v)	Bare Minimum
Add_Partition(p)	Bare Minimum
Remove_Partition(p)	Runs SPAR's Partition Removal algorithm
Downtime()	Run the Sparmes repartitioning algorithm. See the description below.

The Sparmes repartitioner differs from Hermes's Lightweight Repartitioner as follows. Specifically, it replaces the gain calculation with the number of replicas that it can assume would be deleted. Note that it has to make pessimistic assumptions, because it only has a partial view of the system, and other vertices may have made changes that would further reduce the number of replicas from this move. For an explicit algorithm, see Appendix A.

Experimental Setup

The experiments performed compare the performance and cost of the solutions, as well as a few additional examinations of their behavior as appropriate. Each experiment simulates a typical week of operation in an OSN, and is averaged over four runs to reduce variability. Each one starts with an initial graph, and either a hard, or random initial partitioning, or a soft one, which has been improved beforehand with METIS.

MIN_REPLICA experiments also include an initial set of replica partitions meeting k-replication and neighbor-collocation constraints, with $k = 2$ unless otherwise specified.

Each experiment involves a sequence of 10,081 operations, consisting of the aforementioned user addition, user deletion, etc., to represent one operation per minute for a week, plus one more to make the endpoints inclusive. Two of these operations are partition removal, occurring at randomly-generated times; two are partition addition, triggered when the average partition size exceeds a preset number representing the ideal capacity of the partition; seven are regularly-scheduled downtime; and the rest are selected at random. See appendix E for more details.

These experiments use five initial graphs:

- The Mislove Facebook graph [22]
- The ASU Friendster graph [21]
- The Vibrobox graph [23], a Vibroacoustic problem
- The Stiffness Matrix graph [23], a stiffness matrix for an automobile chassis
- The Synthetic 70k graph, generated from a model

The experiments include Stiffness Matrix and Vibrobox to test the algorithms against general graphs, and not just those representing social networks.

The system measures the following after each operation in each trace:

- The edge cut
- The cumulative number of moves, omitting moves forced by partition failure
- The expected all-neighbors query delay for minimum cut solutions
- The number of replicas for MIN_REPLICA solutions

For minimum cut solutions, DUMMY provides a baseline result, and METIS a state-of-the-art centralized partitioning algorithm. The first experiments show the resulting edge cut and the number of physical moves. Later experiments show the expected all-neighbors query delay of a single query of the system, as defined in the background section, using a lognormal distribution with a mean of 12.0844 and a standard deviation of 0.446314 to model expected server latency [17]. No experiments measure replication overhead, as these solutions do not replicate.

For MIN_REPLICA solutions, RDUMMY provides the baseline, and RMETIS as a quasi-state-of-the art solution; METIS was not designed to minimize replication overhead, but as the plot of edge cut vs. replicas shows, it is closely related. The first experiments show the replication and the number of physical moves. Later experiments compare the edge cut to the minimum number of replicas. Further experiments examine minReps vs. Replication Overhead. No experiments measure query delay, because friend-collocation eliminates it by design.

In both sections, some experiments show logical movement in a separate set of plots. Given that logical movement is less expensive than physical movement, these experiments weight it at 10% of the physical cost. These plots exclude DUMMY, METIS, RDUMMY, RMETIS, and SPAR, as they involve no logical movement.

5. EXPERIMENTAL RESULTS

Minimum Cut Solutions

Synthetic 70k

Figure 1 shows the results of the minimum cut solutions applied to the Synthetic 70k traces, both soft start and hard start. The top-left plot shows the edge cut of the six solutions over the weeklong simulation with a hard start. The top-right plot shows the same, but with a soft start. The bottom row of plots shows the cumulative number of physical moves for the soft start and hard start runs, respectively. Note that sometimes Hermar and Hermes overlap, or Ja-be-Ja and Jabar, both in this figure and others.

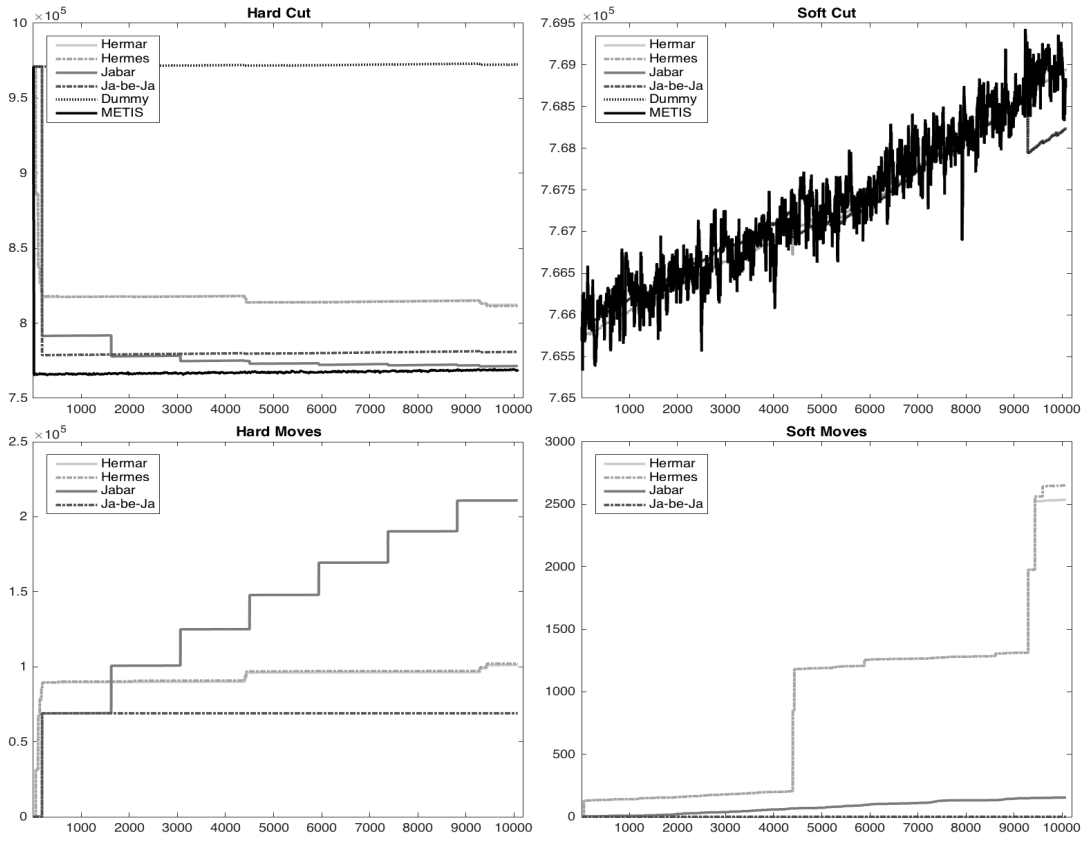

Figure 1 – Synthetic 70k with Minimum Cut Solutions

The figure shows that, with a hard start, Jabar and Ja-be-Ja approach the quality of METIS at moderate expense in terms of moves, while Hermes and Hermar underperform, albeit at lower expense. DUMMY, on the other hand, makes almost no improvements over the run. With a soft start, nearly everything paces METIS except DUMMY, which has an unexpectedly good result from its partition removal steps.

This suggests that the simulated annealing used in Ja-be-Ja and Jabar might be well suited to making large improvements quickly, while Hermes's and Hermar's Lightweight Partitioner might make improvements more slowly. It also suggests that the performance in the hard start might not be directly related to that of the soft start. It is also worth noting that the stair-step changes align with the periodic DOWNTIME.

Vibrobox

Figure 2 shows the results of the minimum cut solutions applied to the Synthetic 70k traces, both soft start and hard start, in the same manner as Figure 1.

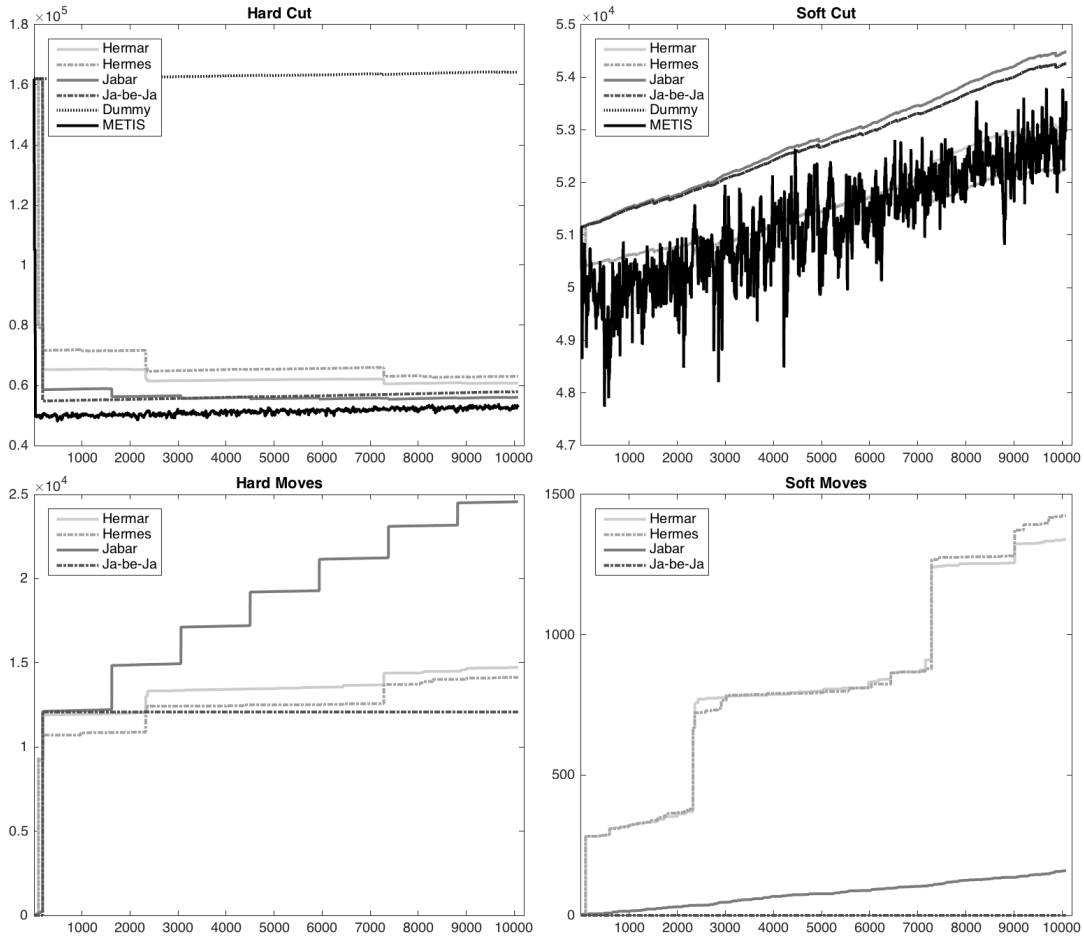

Figure 2 – Vibrobox with Minimum Cut Solutions

This figure shows that, with a hard start, Jabar approaches the quality of METIS at moderate expense, while Ja-be-Ja comes fairly close at somewhat lower expense, and Hermes and Hermar also come fairly close at similar expense. With a soft start, Ja-be-Ja and Jabar fail to beat DUMMY, Hermar matches METIS, and Hermes actually

outperforms METIS at low expense. Oddly, Ja-be-Ja and Jabar fall behind DUMMY at a slow and steady pace.

The difference in the pace at which Hermes and Hermar approach METIS versus the pace at which Jabar and Ja-be-Ja approach it in the hard start lends further credence to the observation that the latter may be better optimized to making large improvements quickly. Conversely, Jabar and Ja-be-Ja seem to have difficulty making the small improvements necessary to keep up with METIS in the soft start, further suggesting that the performance of a solution may be closely tied to the quality of the initial partitioning. Indeed, Nicoara et al. even state that Hermes's "initial partitioning needs to be optimized" and that in tests "[They] use Metis to obtain the initial data partitioning" [12, p4].

Stiffness Matrix

Figure 3 shows the results of the minimum cut solutions applied to the Synthetic 70k traces, both soft start and hard start, in the same manner as Figure 1.

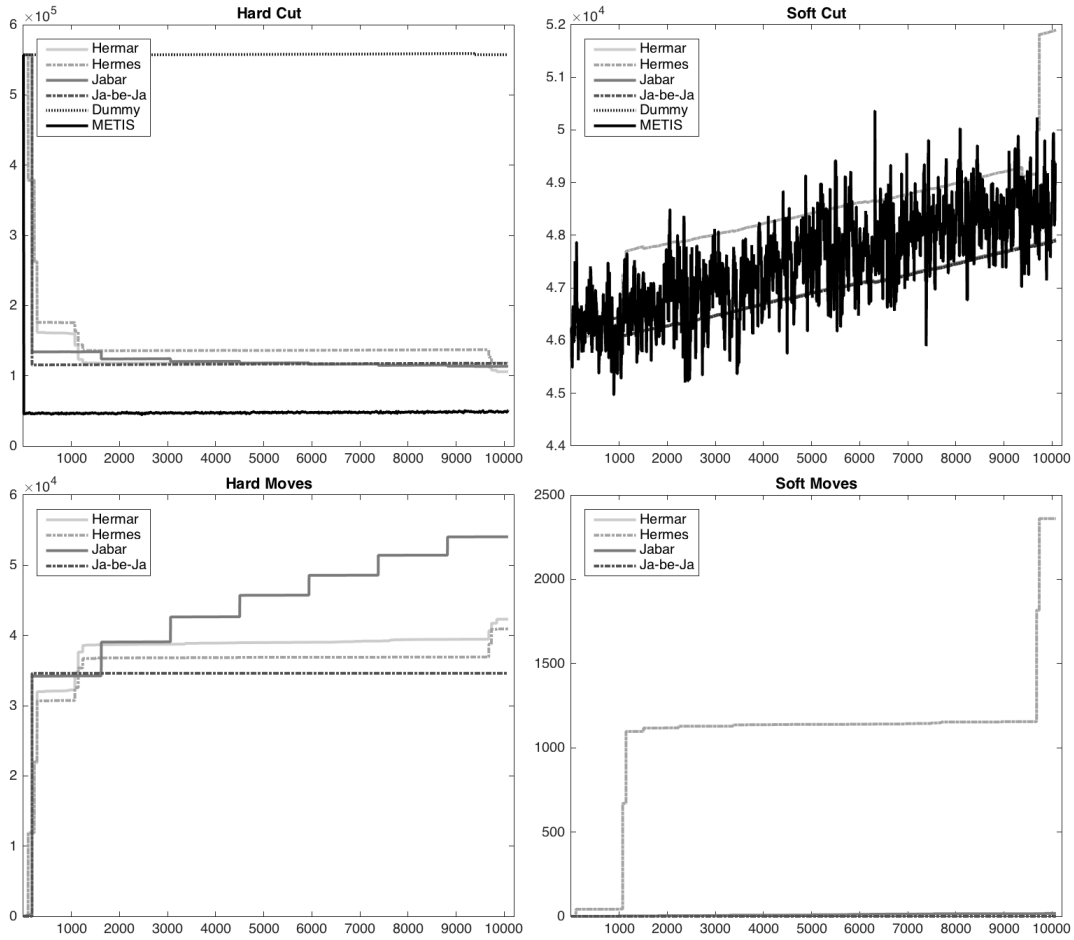

Figure 3 – Stiffness Matrix with Minimum Cut Solutions

This figure shows that, with a hard start, every active solution clusters at about twice the cut of METIS, at similar expenses, while Jabar outperforms the others slightly at a greater expense. This suggests that the social-network optimized algorithms may not perform as well on other types of graphs. With a soft start, Hermes and Hermar have relatively poor showings, while DUMMY, Jabar and Ja-be-Ja actually beat METIS by a noticeable margin.

This differs from the experiments in Figures 1 and 2 in multiple ways. For one, Hermes and Hermar catch up with METIS nearly as quickly as Jabar and Ja-be-Ja. For another, Jabar and Ja-be-Ja are able to make the small improvements necessary to improve upon partitionings generated by METIS, while Hermes and Hermar struggle to do so. As noted earlier, this suggests that a solution's performance may depend heavily on the type of graph data it partitions.

Facebook

Figure 4 shows the results of the minimum cut solutions applied to the Synthetic 70k traces, both soft start and hard start, in the same manner as Figure 1.

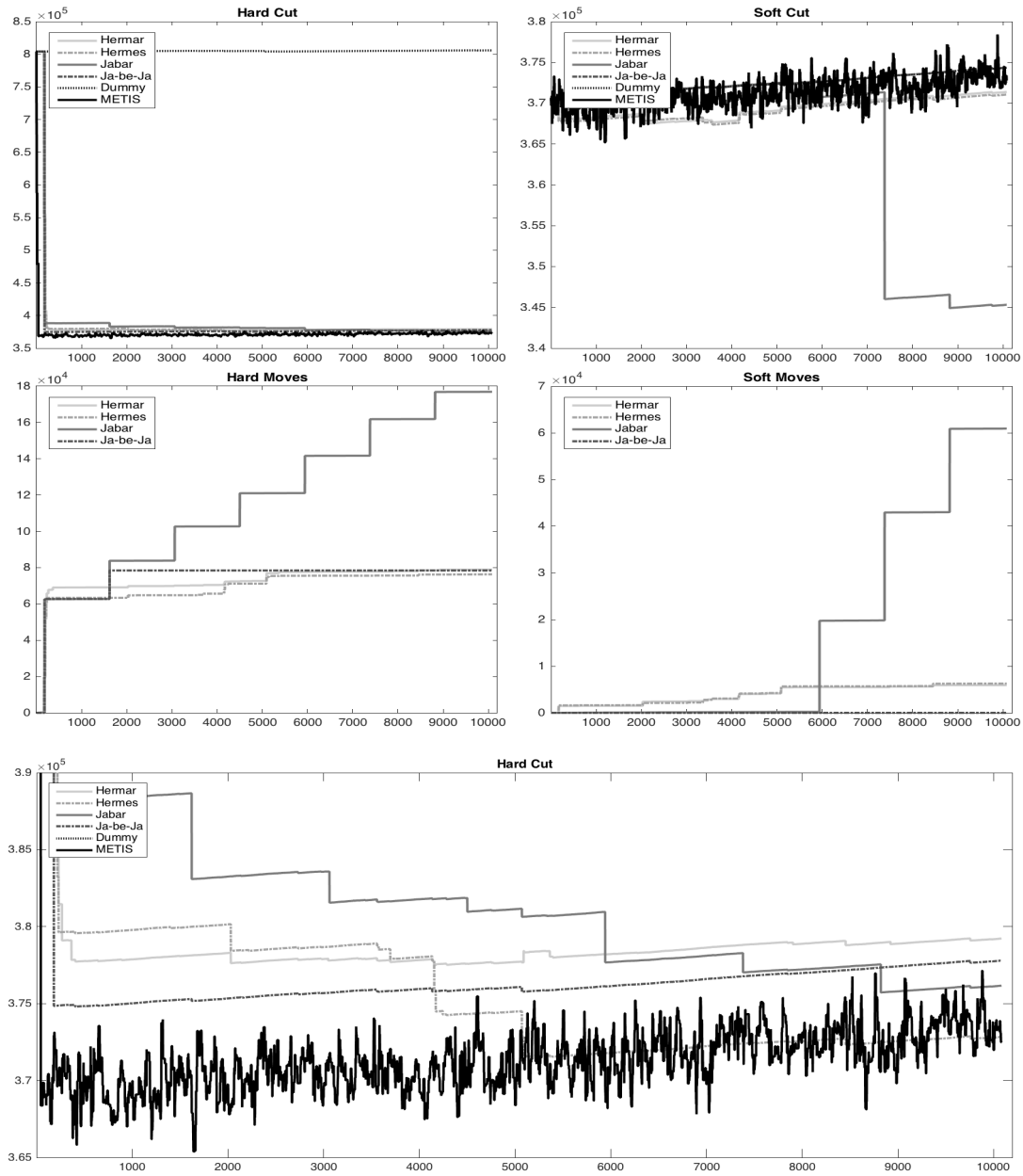

Figure 4 – Facebook with Hard-Cut Detail with Minimum Cut Solutions

This figure shows that, with a hard start, Hermes outperforms the others slightly, at lower expense, while Hermar eventually beats Jabar. With a soft start, Ja-be-Ja again fails to beat DUMMY, Hermes and Hermar edge out METIS, and Jabar puts in an uncommonly good showing.

This further clouds the observations that Jabar and Ja-be-Ja are better tuned to making improvements quickly in the hard-start case; the detail may show that they are quicker to improve, but the full view shows just how small that advantage is. On the other hand, the soft start results provide further evidence that Hermes and Hermar are good at making small improvements to good partitionings, and that Jabar and Ja-be-Ja seek large improvements that they rarely find; however, in this case, Jabar does manage to find such an improvement over METIS.

Friendster

Figure 5 shows the results of the minimum cut solutions applied to the Synthetic 70k traces, both soft start and hard start, in the same manner as Figure 1.

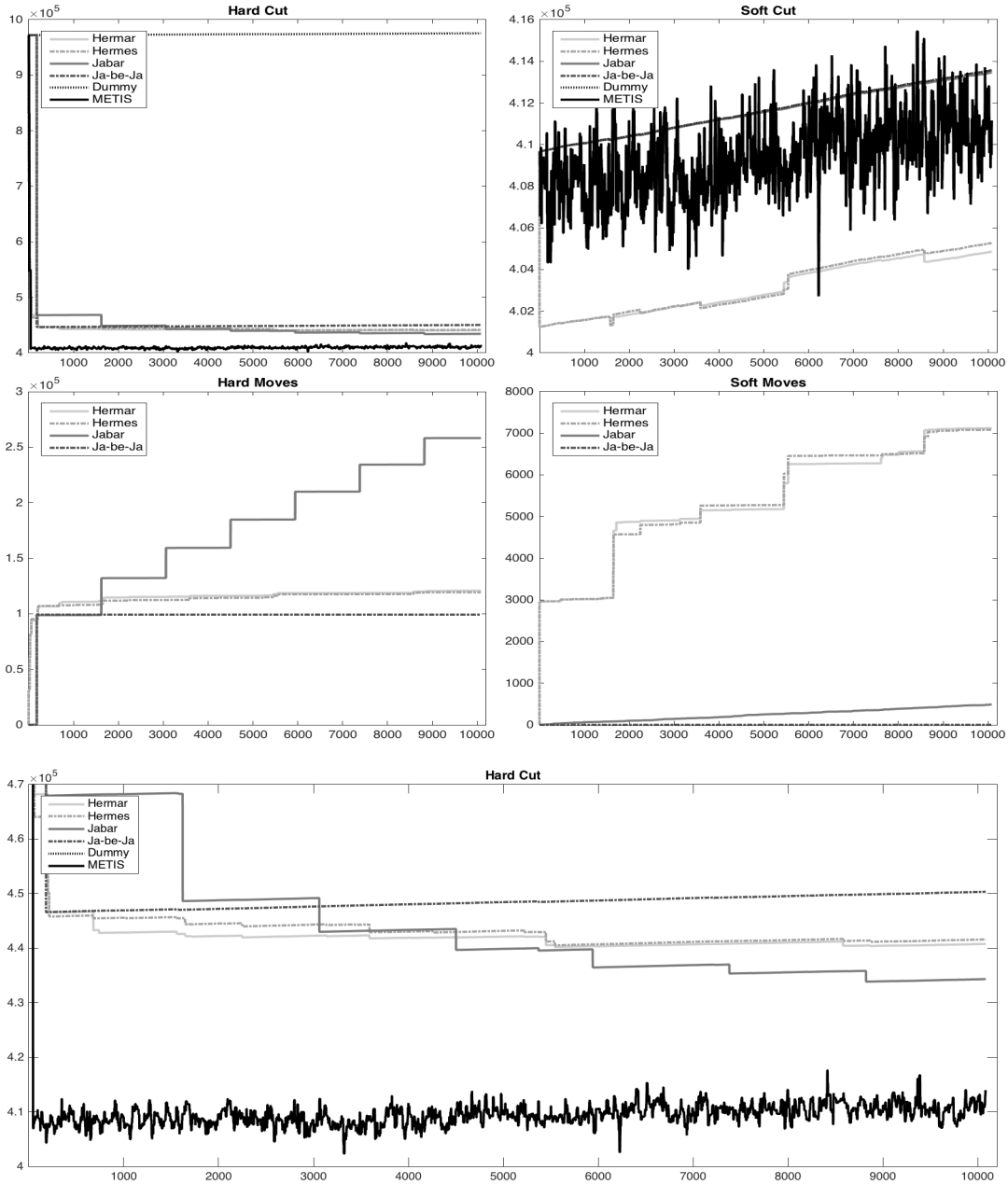

Figure 5 – Friendster with Hard-Cut Detail with Minimum Cut Solutions

This figure shows that, with a hard start, all of the active solutions get within 10% of METIS, with Jabar coming in slightly better and at moderately higher expense. In the soft start, Hermes and Hermar quickly pull ahead of METIS by a significant margin.

The results of this experiment, together with the results of the Facebook experiment shown in Figure 4, suggest that Jabar and Ja-be-Ja don't significantly outperform Hermes and Hermar on actual social network data in the hard-start case. It also makes a strong case for Hermar and Hermes's ability to improve upon SPAR in the soft start.

Expected All-Neighbors Query Delay on Social Media Graphs

Figure 6 shows the expected all-neighbors query delay for the experimental runs detailed in Figures 4 and 5. The top-left plot shows the expected delay for the Facebook run with a hard start, while the top-right plot shows the same, but with a soft start, both of which are shown in Figure 4. The bottom row of plots shows the equivalent for the Friendster runs, both of which are shown in Figure 5.

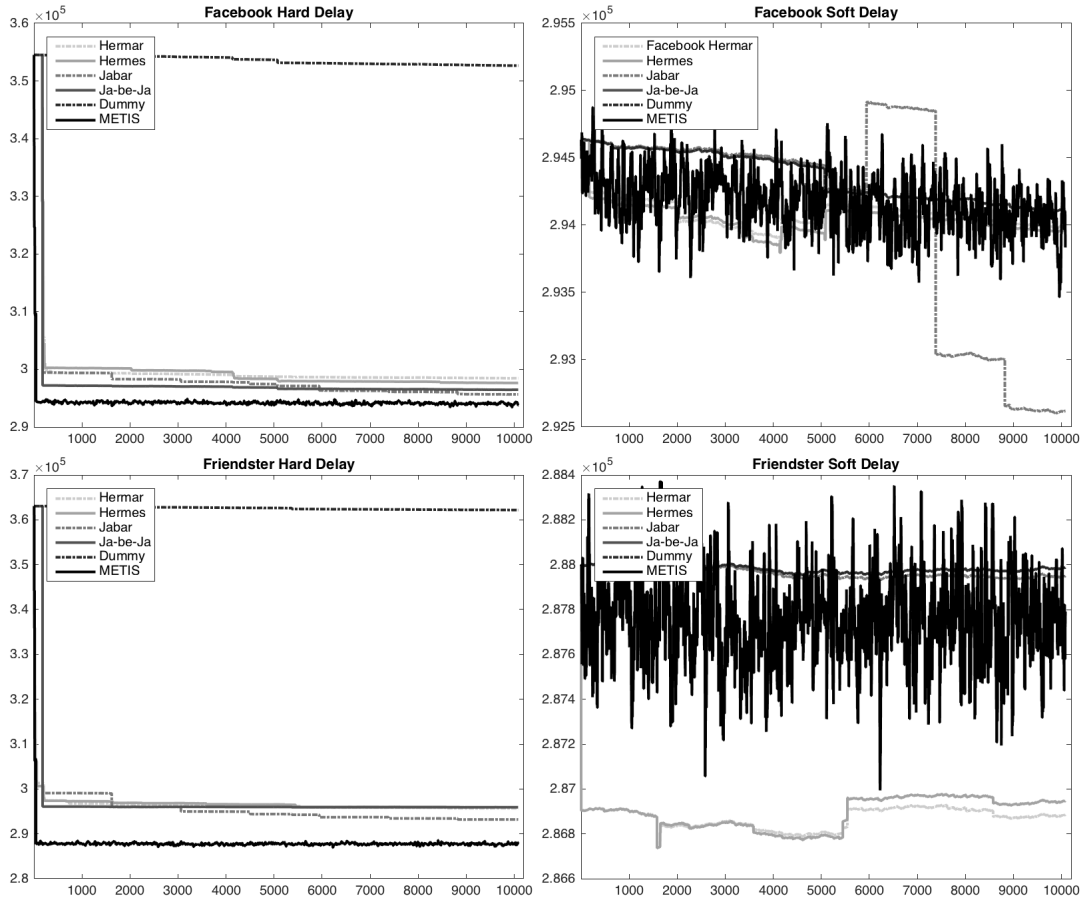

Figure 6 – Expected All-Neighbors Query Delay

This figure shows that, as expected, all-neighbors query delay tracks edge cut very closely, suggesting that edge cut acts as an acceptable proxy therefor.

Impact of Logical Movement Ratio on Movement Cost

Figure 7 shows the weighted logical-and-physical moves for the experimental runs detailed in Figure 4. The top plot shows the weighted moves for the Facebook run with a hard start, while the bottom plot shows the same, but with a soft start. Note that the weighted logical-and-physical moves are equal to the cumulative physical moves plus 10% of the cumulative logical moves; the logical moves are weighted lower to reflect their lower operational cost. Note that Figure 7 excludes DUMMY and METIS because they do not make use of logical movement.

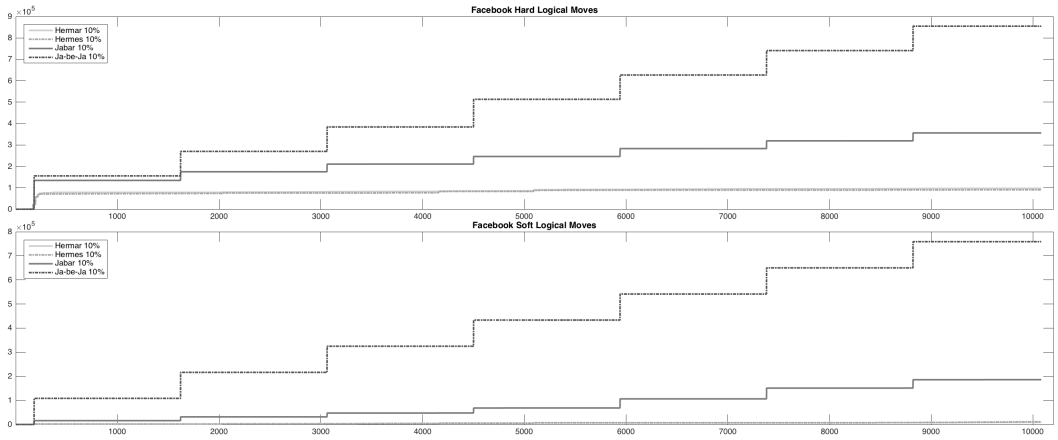

Figure 7 – Logical Moves vs. Physical Moves with Minimum Cut Solutions

This figure shows that, as with the non-logically-weighted plots, Ja-be-Ja and Jabar are significantly more expensive than Hermes and Hermar. Interestingly, Jabar and Ja-be-Ja switch places; this is not unexpected, since Ja-be-Ja runs through three times as many repartitioning iterations. In practice, the gap between Jabar on one side, and Hermes and Hermar on the other may be smaller than shown, as their latter pair's logical moves involve far more data than Jabar's.

MIN_REPLICA Solutions

Synthetic 70k

Figure 8 shows the MIN_REPLICA solutions applied to the Synthetic 70k traces, both soft and hard start, with minReps = 2. The top-left plot shows the number of replicas of the five solutions over the weeklong hard-start simulation. The top-right plot shows the same with a soft start. The bottom row of plots shows the cumulative number of physical moves for the soft- and hard-start runs, respectively. Note that sometimes Spaja overlaps with SPAR in the soft-start rep plots, both in this figure and others.

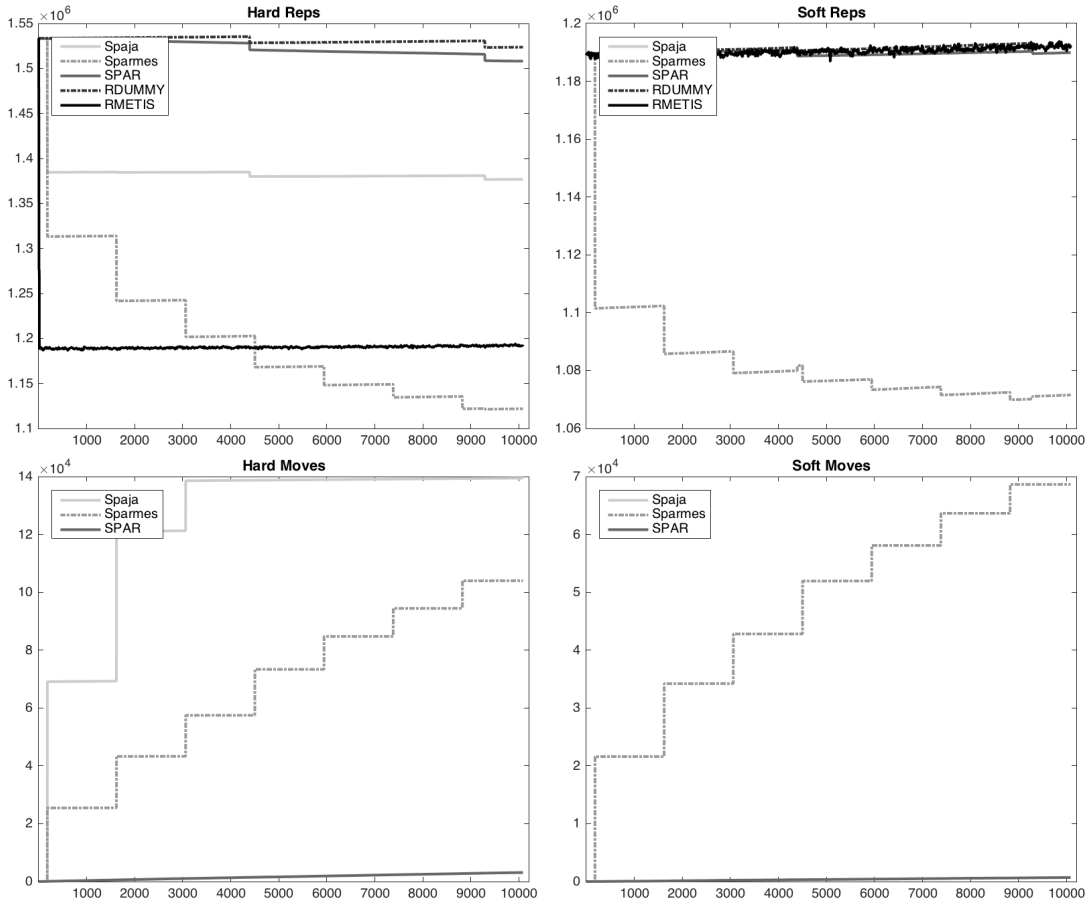

Figure 8 – Synthetic 70k with MIN_REPLICA Solutions

This figure shows that, in the hard start, Sparmes beats the others at moderate expense, while Spaja is a reasonable improvement over SPAR, and SPAR steadily pulls away from RDUMMY. In the soft start, Sparmes pulls away quickly while everything else clusters together. It is worth noting that Sparmes is far more expensive than the others in the soft start, but that may be because the others seem to be unable to make many improvements.

One thing that clearly differs from previous experiments is the way SPAR makes improvements gradually, rather than the stair-step pattern seen in the Minimum Cut section. This is because SPAR makes improvements during BEFRIEND operations, which make up over a quarter of all operations, rather than the periodic DOWNTIME operations.

Vibrobox

Figure 9 shows the results of the minimum cut solutions applied to the Synthetic 70k traces, both soft start and hard start, in the same manner as Figure 8.

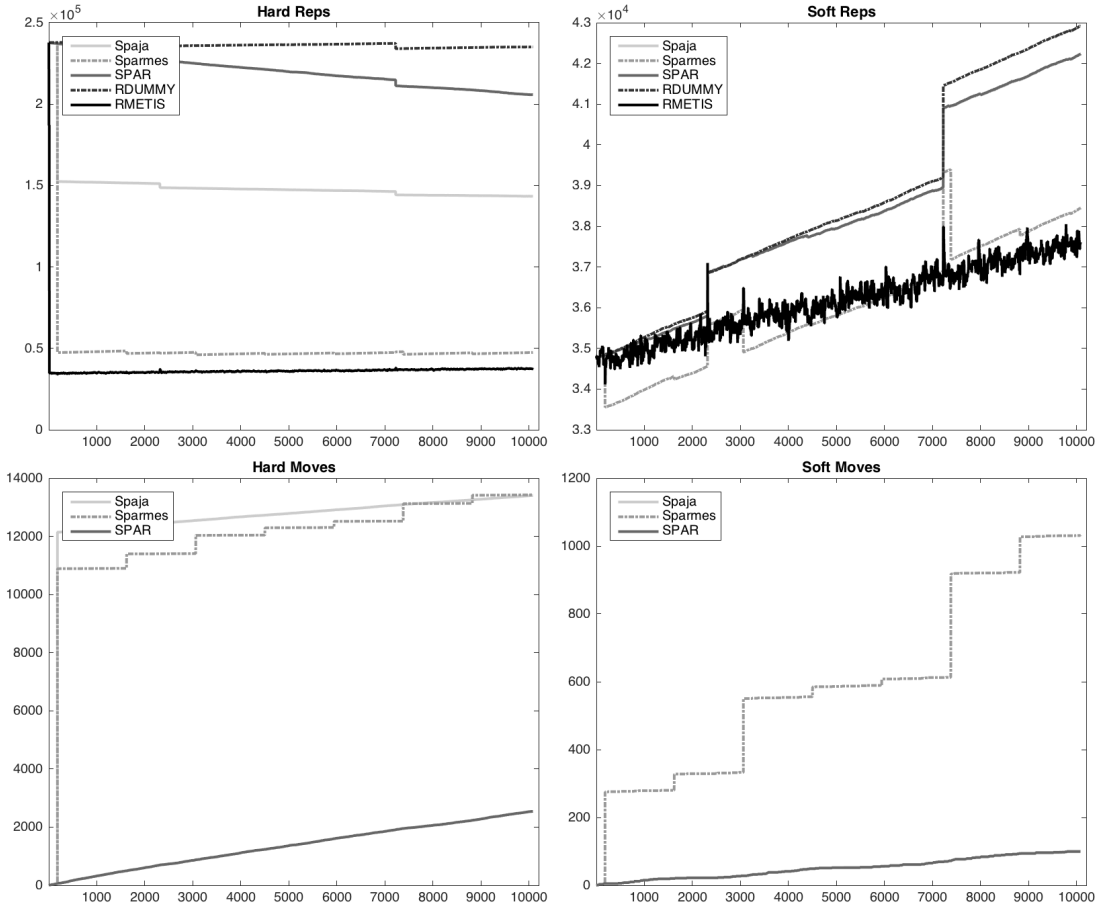

Figure 9 – Vibrobox with MIN_REPLICA Solutions

This figure shows that Sparmes beats everything but RMETIS at moderate expense, while SPAR consistently pulls away from RDUMMY in the hard start, and Spaja is a reasonable improvement over SPAR in the hard start. Further, Sparmes is roughly tied with RMETIS in the average in the soft start, though it degrades over time. This suggests that Sparmes might not be as well suited to non-social network data.

Stiffness Matrix

Figure 10 shows the results of the minimum cut solutions applied to the Synthetic 70k traces, both soft start and hard start, in the same manner as Figure 8.

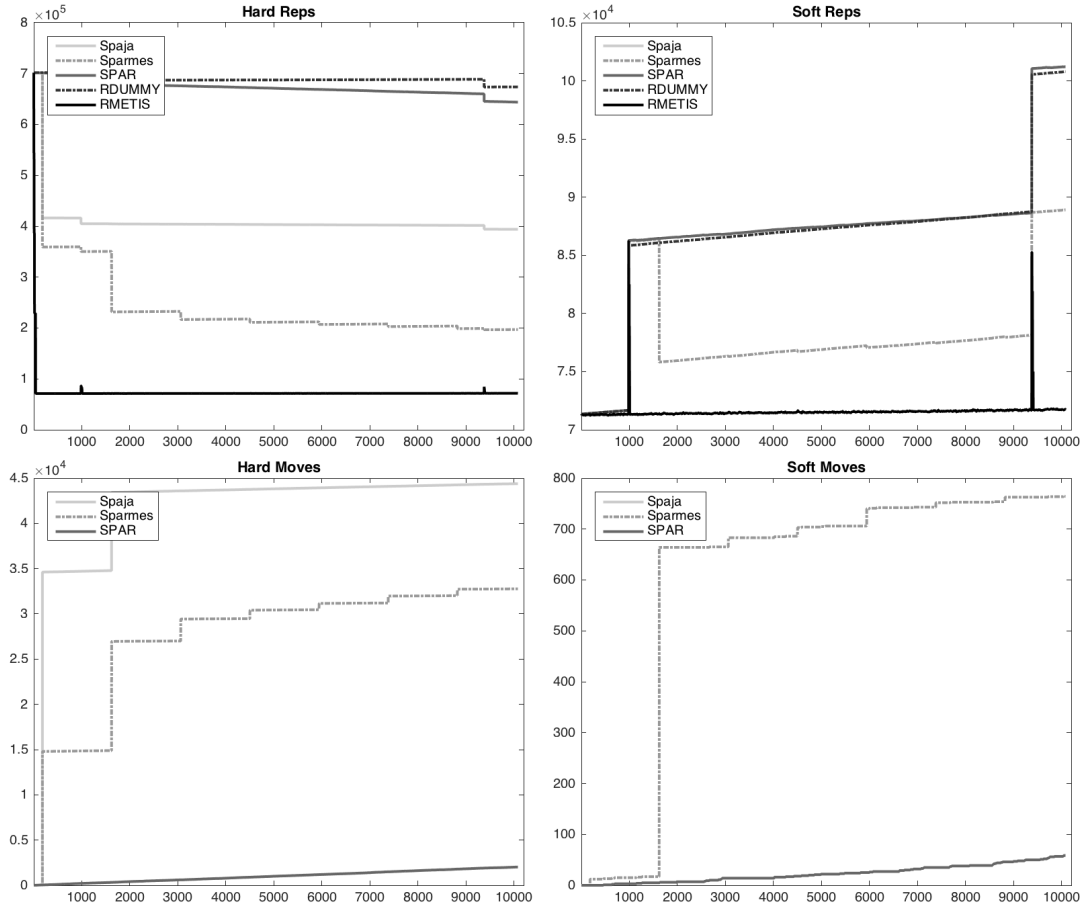

Figure 10 – Stiffness Matrix with MIN_REPLICA Solutions

This figure shows that, similarly to the Vibrobox experiment shown in Figure 9, Sparmes beats everything but RMETIS at moderate expense, albeit to a lesser extent, while SPAR slowly pulls away from RDUMMY, and Spaja is a reasonable improvement over SPAR in the hard start. In the soft start, Sparmes is relatively close to RMETIS, while the others all cluster together at a lower level of performance. This reinforces the hypothesis that Sparmes does not perform as well on non-social network data.

Facebook

Figure 11 shows the results of the minimum cut solutions applied to the Synthetic 70k traces, both soft start and hard start, in the same manner as Figure 8.

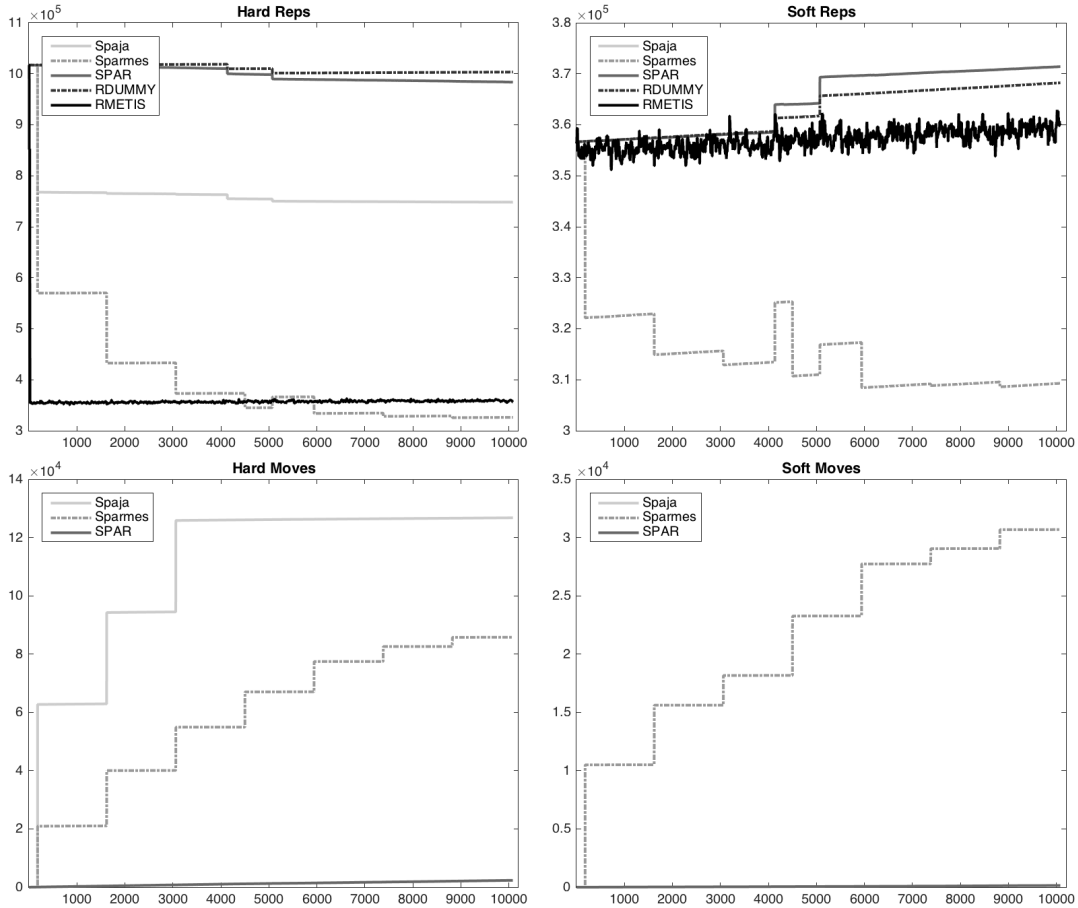

Figure 11 – Facebook with MIN_REPLICA Solutions

This figure shows that Sparmes beats all other solutions at moderate expense, while SPAR slowly pulls away from RDUMMY in the hard start, and Spaja is a reasonable improvement over SPAR in the hard start. Notably, RDUMMY outperforms SPAR in the soft start, pulling away at the partition removals.

This again suggests that Sparmes delivers better performance on social network data, both compared to its performance on non-social network data, and compared to RMETIS's performance on OSN data.

Friendster

Figure 12 shows the results of the minimum cut solutions applied to the Synthetic 70k traces, both soft start and hard start, in the same manner as Figure 8.

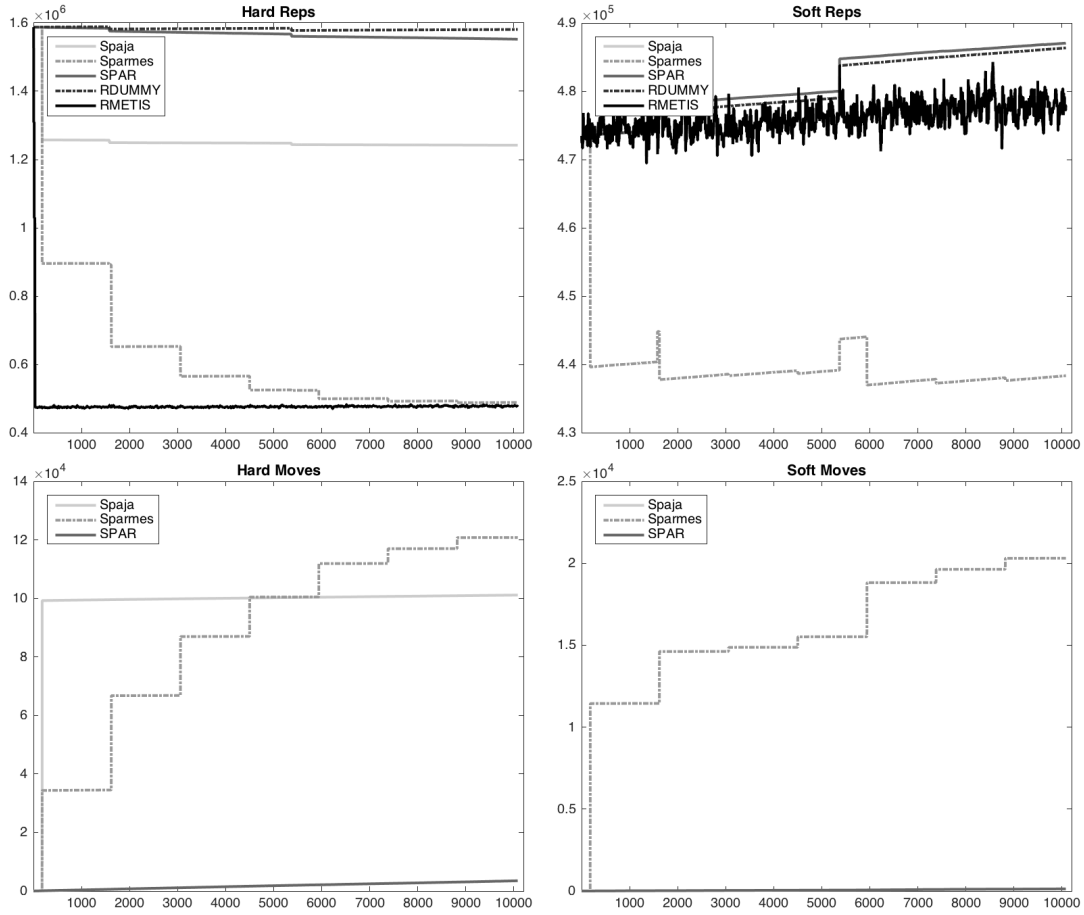

Figure 12 – Friendster with MIN_REPLICA Solutions

This figure shows that, yet again, Sparmes beats the others at moderate expense, with the exception of an eventual tie with RMETIS in the hard start. SPAR slowly pulls away from RDUMMY in the hard start, and Spaja is a reasonable improvement over SPAR in the hard start. Interestingly, RDUMMY beats SPAR in the soft start, showing improvement at the partition removals.

This solidifies the observation that Sparmes is well suited to social network data. It further suggests, in conjunction with the results from Facebook in Figure 11, that SPAR may be at a disadvantage during partition removal; one possibility is that the small-scale improvements that SPAR makes during befriending can interfere with the optimizations of the water-filling strategy used in migration after partition removal. Another possibility is that SPAR's partition removal algorithm's modification of the water-filling strategy might be counter-productive in some circumstances. As noted above, for SPAR "[t]he algorithm decides the server in which a slave replica is promoted to master, based on the ratio of its neighbors that already exist on that server", [3, p379]. This restriction might hamper minimizing the number of replicas in the case of real social network data.

Cut vs. Replication Overhead on Social Media Graphs

Figure 13 compares the total number of replicas to the edge cut for the experimental runs detailed in Figures 11 and 12. The top-left plot compares replicas and edge cut for the Facebook run with a hard start, while the top-right plot shows the same, but with a soft start, both of which are come from Figure 11. The bottom row of plots shows the equivalent for the Friendster runs, both of which come from Figure 12.

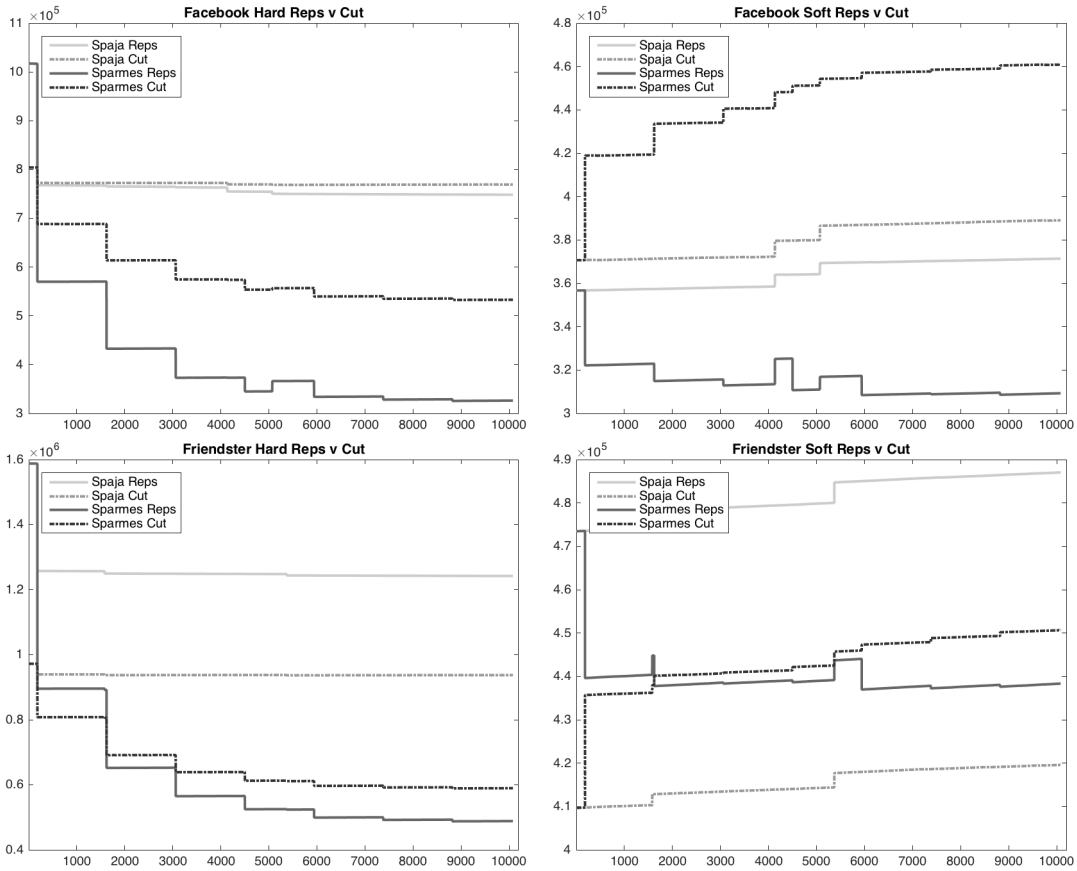

Figure 13 – Cut vs. Number of Replicas

This figure shows that, as predicted, there is a relationship between replication overhead and underlying edge cut. However, in the soft starts, Sparmes somehow

reduces its replication overhead despite significant increases in edge cut for both Friendster and Facebook, suggesting that the connection is not ironclad.

This has implications for RMETIS as a proxy for a state-of-the-art solution. METIS is a true state-of-the-art solution for reducing the edge cut, but given the considerable divergence between the edge cut and the replication overhead in the soft start, it could prove useful to seek out other solutions.

Impact of minReps on Total Replication Overhead

Figure 14 shows the impact of the value of minReps on the total number of replicas for the experimental runs detailed in Figure 11. The top plot compares minReps at values 0 and 2 for the Facebook run with a hard start, while the bottom plot shows the same, but with a soft start.

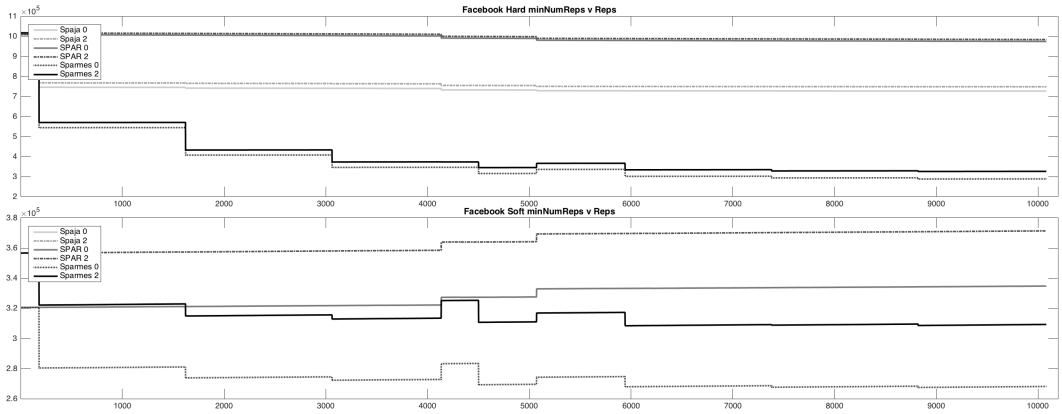

Figure 14 – minNumReps vs. Number of Replicas

This figure shows that SPAR, Spaja and Sparmes can all ratchet up the minReps without much penalty, as going from 0 to 2 costs SPAR and Sparmes less than 15%. This lends credence to Pujol et al.'s claim that additional redundancy can be achieved with minor marginal costs [3]. Note that Spaja is tied with SPAR in the soft run.

This suggests that there might not be a strong tradeoff between redundancy and performance, which could be a boon to OSN operators who want high-availability at a moderate cost.

Impact of Logical Movement Ratio on Movement Cost

Figure 15 shows the weighted logical-and-physical moves for the experimental runs detailed in Figure 11. The top plot shows the weighted moves for the Facebook run with a hard start, while the bottom plot shows the same, but with a soft start. Note that the weighted logical-and-physical moves are equal to the cumulative physical moves plus 10% of the cumulative logical moves; the logical moves are weighted lower to reflect their lower operational cost. Note that Figure 15 excludes RDUMMY, RMETIS, and SPAR because they do not make use of logical movement.

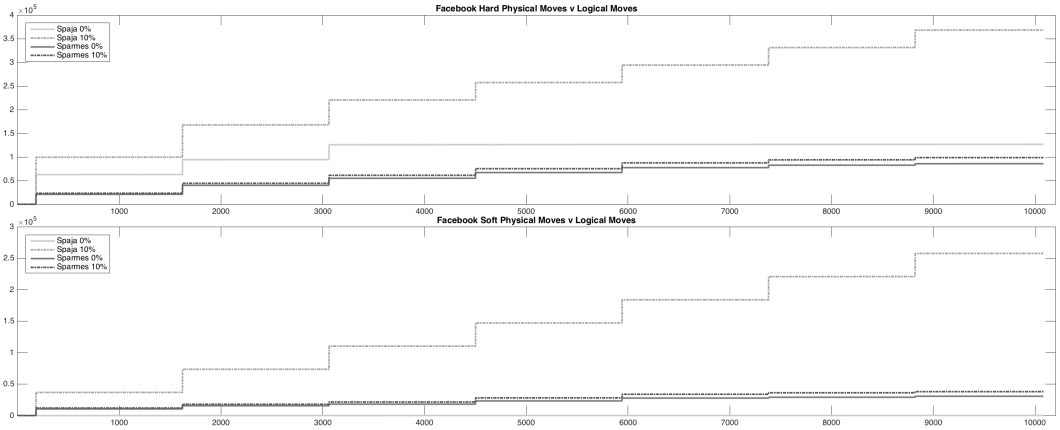

Figure 15 – Logical Moves vs. Physical Moves with MIN_REPLICA Solutions

This figure shows that the relative shape of the cost of Sparmes and Spaja does not change when considering the cost of logical moves in the hard start. In the soft start, however, Spaja costs much more. This suggests that Sparmes has an even greater advantage over Spaja than suggested in Figures 8-12.

6. RESULT EVALUATION

Minimum Cut Solutions

Hermes and Hermar are the right choices for soft-start partitions, with their low cost, and their ability to maintain and improve upon already great partitionings. Hermes seems to have a slight edge, suggesting that, while Hermar's befriending strategy seems to work at times, it is not a runaway success.

Jabar and Ja-be-Ja are somewhat better with hard-start partitions. They come at a higher movement cost, but have the benefit of improving upon bad partitions very quickly. Ja-be-Ja starts out the gate very well, but Jabar catches up and surpasses it in nearly all cases, and while Jabar makes more physical migrations, its total cost is much lower when taking logical migrations into account.

Additionally, while simulated annealing involves many more moves, the Lightweight Repartitioner requires a significant imbalance, which is 15% in this case, so this may not be feasible in production systems. It seems likely that, the addition of a convergence detection mechanism to the implementation of the Simulated Annealing algorithm could reduce costs significantly.

Note that all algorithms outperform the DUMMY implementation in hard starts, though in the soft starts it seems to do well in a few cases.

Finally, the similarity between expected all-neighbors query delay and edge cut supports the theory that edge cut is a good proxy for the performance of a system.

MIN_REPLICA Solutions

For MIN_REPLICA solutions, Sparmes is far and away the best, both in soft and hard starts. Furthermore, it frequently had fewer moves than Spaja in the hard-start experiments, and while it had more moves in the soft-start experiments, that seems to be because Spaja had a hard time finding improvements to make. Of particular note is that Sparmes beats RMETIS in at least part of every soft start, and in part of all but two of the hard starts, still coming close in Vibrobox and Friendster.

With a logical movement ratio of 10%, it seems that Sparmes is even stingier in comparison with Spaja than without taking logical movement into account. This may be because Sparmes's Lightweight Repartitioner is capable of early termination if convergence is detected, unlike the extension of the Simulated Annealing algorithm into the MIN_REPLICA problem.

In some soft-start experiments, namely Figures 11-12, it seems that SPAR underperforms RDUMMY, and that it seems to fall behind during the partition removal operations. It is worth noting that the Figures 8-10 showed no such problem, nor did any of the hard-start experiments.

It appears that increasing the k in k-redundancy comes at a very low cost, suggesting that high-availability and fault-tolerance are quite achievable.

Finally, it seems that, while edge cut and replication overhead track well in some cases, they are not quite the same thing.

7. CONCLUSIONS AND FUTURE WORK

Future Work

It appears that Ja-be-Ja, Jabar, and Spaja seem to have a hard time improving already-high-quality partitions. This could be because of a flaw in the implementation, a suboptimal choice of parameters, or even a fundamental characteristic of KL-type partitioners. Examining this would be worthwhile, especially if it could improve Jabar's already-high performance.

Comparisons of the Lightweight Repartitioner-derived solutions and the Simulated Annealing-derived ones are not truly apples-to-apples, given that the former can use γ times as much partition space. Thus, it would be interesting to divide their maximum partition size by γ to make a more direct comparison. See the $|\mathbb{V}|/P$ column in Table 15 in Appendix E for the chosen, implicit maximum partition sizes.

As noted earlier, the Simulated Annealing-derived logical moves are not really as expensive as the Lightweight Repartitioner ones, so coming up with a more direct comparison could shed light on their relative cost.

Similarly, there was no examination of the cost of movement of replicas, either physical or logical. While this seems like a minor concern next to the movement cost of masters, the result might be different from the expectation.

MIN_REPLICAS treats k as a constant, ensuring that each vertex has at least k replicas at any given time. However, some vertices receive significantly more requests than others, such as celebrities, meaning that a failure in availability for such vertices

would be more costly to the OSN operator. Thus, it would be worthwhile to consider increasing k as a function of popularity, or some other weighting function.

Similarly, some of the algorithms from the literature have the option to seek balanced partitions based not on the number of vertices, but the total weight of the vertices, typically representing popularity; the motivation is to balance computational load, as a partition that contains 500 celebrities is likely to have higher CPU utilization than one with 500 typical users. Thus, it would be worthwhile to incorporate this into the simulations to see what effect it has.

Given that Simulated Annealing handles poor-quality partitions well, and Lightweight Repartitioner handles high-quality ones better, it would be interesting to hybridize these somehow, most likely by running a few iterations of both, comparing the results, and continuing with whichever one shows more promise.

Further, given that edge cut and replication overhead differ at times, it would be good to seek out a state-of-the-art solution to see if RMETIS is underperforming.

It is worth noting that the all-neighbors query delay calculation is a function of the number of partitions on which each vertex has neighbors, rather than the edge cut itself. Thus, it would be interesting to examine minimizing the number of such partitions as a third problem in addition to MIN_REPLICAS and Minimum Cut.

Finally, given that recent updates to the Google Compute Engine allow for low-cost, persistent local storage on VMs, it would be neat to implement the solutions for real, and not in a simulation. This would enable actual performance metrics rather than cost estimations.

Conclusions

The results show that hybridizing can indeed yield significant benefits, with Jabar greatly reducing the logical cost of Ja-be-Ja, and Sparmes dramatically improving the performance of SPAR. Further, the movement cost of Sparmes is quite reasonable, generally lower than Spaja. At their best, the hybridized solutions can approach, or even slightly outperform METIS and RMETIS. This suggests that hybridizing is a field with significant potential.

APPENDIX SECTION

APPENDIX A – ALGORITHMS FOR HYBRID SOLUTIONS

Jabar Befriending

```
//Let k be the number of swap candidates considered at once
//Let α be the power to which Jabar raises the comparison function
REBALACE(u, v)
    if u.pid = v.pid
        return NULL
    partner_u ← FIND_PARTNER(u, v.pid)
    partner_v ← FIND_PARTNER(v, u.pid)
    if (partner_u ≠ NULL) ∧ (partner_v ≠ NULL)
        (u_old, u_new) ← CALCULATE_SCORE(u, partner_u, 1)
        (v_old, v_new) ← CALCULATE_SCORE(v, partner_u, 1)
        if (u_new - u_old) ≥ (v_new - v_old)
            return (u, partner_u)
        else
            return (v, partner_v)
    else if (partner_u ≠ NULL) ∧ (partner_v = NULL)
        return (u, partner_u)
    else if (partner_u = NULL) ∧ (partner_v ≠ NULL)
        return (v, partner_v)
    else
        return NULL

FIND_PARTNER(v, p)
    score_best ← 0
    candidate_best ← NULL
    candidates ← p.drawRandomly(min(k, p.size))
    for u ∈ candidates
        (score_current, score_proposed) ← CALCULATE_SCORE(u, v, α)
        if (score_proposed > score_best) ∧ (score_proposed > score_current)
            score_best ← score_proposed
            candidate_best ← u
    return candidate_best

CALCULATE_SCORE(u, v, α)
    u_nbhd_current ← | u.nbhd ∩ u.pid |
    v_nbhd_current ← | v.nbhd ∩ v.pid |
    u_nbhd_proposed ← | u.nbhd ∩ v.pid |
    v_nbhd_proposed ← | v.nbhd ∩ u.pid |
    if u ∈ v.nbhd
        v_nbhd_proposed ← v_nbhd_proposed - 1
        u_nbhd_proposed ← u_nbhd_proposed - 1
    score_old ← u_nbhd_current^α + v_nbhd_current^α
    score_new ← u_nbhd_proposed^α + v_nbhd_proposed^α
    return (score_old, score_new)
```

The Jabar befriending algorithm is essentially a truncated, focused version of the simulated annealing algorithm. FIND_PARTNER finds which of k candidates from $v.pid$ would most reduce the edge cut if swapped with u, called $partner_u$, and similarly find $partner_v$. Whichever of $partner_u$ and $partner_v$ has greater gain is chosen to swap with u or v, respectively, if the gain is positive; otherwise it returns NULL. Note that, because Ja-be-Ja does not allow partition imbalance, it cannot migrate an individual vertex, which is why it seeks candidates in the other partition.

Hermar Befriending

```
//Let γ be the maximum ratio of any partition's weight to the average
partition's weight
REBALANCE(u, v)
    gain1_to_2 ← CALCULATE_GAIN(u, v.pid)
    gain2_to_1 ← CALCULATE_GAIN(v, u.pid)
    if (gain1_to_2 > gain2_to_1) ∧ (gain1_to_2 > 0)
        return u
    else if (gain2_to_1 > gain1_to_2) ∧ (gain2_to_1 > 0)
        return v
    else
        return NULL

CALCULATE_GAIN(u, p)
    if (p.size+1) < γ * |U| / |P|
        nbhdcurrent ← | u.nbhd ∩ u.pid |
        nbhdp ← | u.nbhd ∩ p |
        return nbhdp - nbhdcurrent
    else
        return -∞
```

The Hermar befriending algorithm considers the gain in the number of neighbors that a vertex may experience by moving to the other vertex's partition. The vertex with the largest gain to a non-overweight partition is selected to migrate, as long as that gain is positive.

Spaja Repartitioning

```
//Let T_initial be the initial temperature of the system
//Let δT be the change in decrease in temperature each iteration
//Let k be the number of swap candidates Spaja considers at once
//Let α be the power to which Spaja raises the comparison function
REPARTITION()
    T ← T_initial
    while T ≥ 1
        for v ∈ V
            partner ← FIND_PARTNER(v, V.getRandomSample(k), T)
            if partner =/= NULL
                SWAP_AND_REREPLICATE(v, partner)//details omitted
        T ← T-δT

FIND_PARTNER(v, candidates, T)
    lowest ← ∞
    bestPartner ← NULL
    for u ∈ candidates
        v_cur ← | v.nbhd ∩ v.pid |
        u_cur ← | u.nbhd ∩ u.pid |
        δv ← Δ_REPLICAS(v, u.pid)
        δu ← Δ_REPLICAS(u, v.pid)
        score_current ← v_cur^α + u_cur^α
        score_proposed ← (v_cur + δv)^α + (u_cur + δu)^α
        if (score_proposed < lowest) ∧ ((score_proposed/T) < score_current)
            lowest ← score_proposed
            bestPartner ← u
    return bestPartner

Δ_REPLICAS(v, P_dest)
    add_on_src ← ( (v.nbhd ∩ v.pid) =/ ∅ ) ? 1 : 0
    remove_from_dest ← v ∈ REPLICAS_ON(P_dest)
    add_on_dest ← |(v.nbhd \ P_dest) \ REPLICAS_ON(P_dest)|

    remove_from_src ← 0
    for u ∈ (REPLICAS_ON(v.pid) ∩ v.nbhd )
        if u.nbhd ∩ v.pid = ∅
            to_add_on_dest ← to_add_on_dest + 1

    if num_replicas(v) + add_on_src - remove_from_dest < k
        remove_v_from_dest = 0;

    return add_on_src + add_on_dest – remove_from_dest - remove_from_src
```

The Spaja repartitioning algorithm is based on Ja-be-Ja's simulated annealing algorithm. In each iteration within REPARTITION, for each vertex, it attempts to find a swap partner from a set of random candidates using Ja-be-Ja's random partner selection

policy. If one is found, it performs the swap and adds or removes replicas in the two partitions as required to maintain k-replication and neighbor collocation.

FIND_PARTNER rates partners based on the number of neighbor replicas that each vertex would have in each of the two partitions; in this case, it determines the δ from the number of replicas it would need to add to maintain k-replication and local semantics, and how many it could remove without violating constraints. Since it tries to minimize replicas instead of maximizing gain, it adjusts Ja-be-Ja's findPartner algorithm to divide by T, instead of multiplying by it.

Sparmes repartitioning

```
//Let γ be the maximum ratio of any partition’s weight to the average
partition’s weight
//Let k be the maximum number of vertices moved from one partition in
one iteration
REPARTITION
    iteration ← 0
    converged ← false
    while ¬converged ∧ (iteration < ITERATION_LIMIT)
        changed_1 ← PERFORM_STAGE(1)
        changed_2 ← PERFORM_STAGE(2)
        converged ← ¬(changed_1 ∨ changed_2)
        iteration ← iteration + 1
    PHYSICALLY_MIGRATE() //details omitted

PERFORM_STAGE(stage)
    changes ← {}
    for p ∈ P
        changes ← changes U GET_CANDIDATES(p)
    LOGICALLY_MIGRATE(changes) //details omitted
    return |changes| > 0

GET_CANDIDATES(p, stage)
    candidates ← {}
    for v ∈ p.logical
        target ← GET_TARGET_PART(v, stage)
        if target ≠ NULL
            candidates ← candidates U {target}
    return TOP_K(candidates, k, gain) //retrieve top k by target.gain
```

```
GET_TARGET_PART(v, stage)
    if imbalance(v.P \ {v}) > 2 – γ
        return NULL
    target ← NULL
    gainmax ← 0
    if imbalance(v.P) > γ
        gainmax ← –∞
    for P' ∈ ALL_P
        if((stage=1) ∧ (P'.ID > v.P.ID)) ∨
         ((stage=2) ∧ (P'.ID < v.P.ID))
            if imbalance(P' U {v}) < γ
                gain ← GAIN(v, v.P, P')
                if gain > gainmax
                    target ← P'
                    gainmax ← gain
    return (target, gainmax)
```

The two differences between Sparmes's Lightweight Repartitioner and Hermes's are that Sparmes's logical and physical movement steps involve replica data, and the gain function considers replication instead of cut.

The gain of moving u from P to P' is defined as the number of replicas that it can assume would be deleted minus the number that could be added. Note that it has to make pessimistic assumptions, because each partition only has a partial view of the system, and other vertices may have made changes that would further reduce the number of replicas from this move.

APPENDIX B - DIFFERENCES BETWEEN PAPERS AND IMPLEMENTATIONS

Hermes

First and foremost, the implementation does not take vertex weighting into account [12]. Because the other solutions do not make use of it by default, it would not be terribly helpful. Additionally, ignoring it does not harm the correctness of the algorithm - it merely creates a special case in which all weights are 0 or 1. As a result, there is only one operation that can trigger repartitioning, and that is user addition.

Second, most of the middleware actions are unspecified in the paper. Specifically, removing a vertex, befriending and unfriending, adding and removing partitions, and downtime are not mentioned, so the implementation uses reasonable interpretations.

Third, the algorithm does not always converge, despite the authors' proof that it does. In correspondence, Dr. Nicoara agrees that in certain adversarial cases the convergence is not guaranteed. Thus, the implementation limits the iterations to a fixed maximum, which is 100 by default.

Fourth, in lines 11-13 of the algorithm GET_TARGET_PART, the implementation does not use >, but instead fills a set with all tied candidates and breaks the tie based on whichever one is lightest. This is a minor tweak of which the authors might approve.

Fifth, the paper describes a two-phase physical movement process. The implementation skips this, since it is not running an actual distributed system and it does not impact the simulation. Instead, it migrates each vertex in sequence in one step.

Finally, it does not actually maintain the logical vertices, it just recreate them when it is time to repartition. This should not have any impact on the simulation.

Ja-be-Ja

Ja-be-Ja allows for something called a one-host-many-partitions model, which refers to virtualized servers, where multiple logical servers exist on one physical server [11]. It notes that in such cases, swaps between different partitions on the same server are much faster than those across servers. However, the implementation does not make use of this optional enhancement, and instead uses the one-host-one-partition model.

The convergence behavior and early termination of Ja-be-Ja is not explicit in the paper, and also does not appear to be guaranteed. This behavior may be a subtle bug in the implementation or a misinterpretation. Regardless, the implementation stops the simulated annealing process upon reaching $t = 1$.

Ja-be-Ja mentions that it uses multi-restart, but provides no details. The parameter tuning, covered in Appendix C, found that 3 restarts delivers decent results.

Finally, much like Hermes, Ja-be-Ja does not specify what to do in each of the seven middleware operations that this thesis specifies. All it provides is a partitioner, together with the statement that, "We assume the nodes of the graph are processed periodically and independently," which suggests a fit with the DOWNTIME operation.

As with Hermes, it implements all other operations in the most unembroidered fashion, including vertex migration after partition.

SPAR

In its algorithm for edge addition, SPAR does not explicitly state that if u moves to V, and u had a replica in V, that it should delete that replica [3]. However, it seems obvious, given that a replica on the same partition would not improve reliability nor performance.

SPAR also says that "[M]ovement either happens to a partition with fewer masters, or to a partition for which the savings in terms of number of replicas of the best configuration to the second best one is greater than the current ratio of load imbalance between partitions." The precise definition of "number of replicas" is unclear. It could refer to the total number of replicas in the system, or the number of replicas across the two partitions in question. Having implemented it both ways, the latter produces better results, so the implementation under test uses that interpretation.

APPENDIX C - PARAMETER TRAINING

The parameter training process ran the solutions against the ~4k Leskovec Facebook trace set with various combinations of parameters, typically at least eight times per combination to stamp out random noise [24]. For Ja-be-Ja and Jabar, it trained each parameter separately, except for δT vs. $numRestarts$ in Ja-be-Ja. For the others, it tried all combinations of the various parameters. Note that there are no trainable parameters in DUMMY, METIS, RDUMMY, RMETIS, or SPAR.

Ja-be-Ja

$T_{initial}$: It tried 4, 3.5, 3, 2.5 and 2, and found that 2 performed the best on hard starts, while there was no significant difference for soft starts. Thus makes sense, as the authors suggest 2 [11].

δT vs. $numRestarts$: Unlike other tunings, here it performed two phases. In the first, it used (δT, $numRestarts$) = (0.01, 1), (0.03, 3), (0.05, 5), (0.1, 10), (0.2, 20), and (0.5, 50), chosen because holding $numRestarts/\delta T$ constant should yield approximately equal runtimes. It found that improvements to δT uniformly outweighed increasing the number of restarts, and also found that the runtimes were indeed quite similar. It then performed a second phase with (0.0025, 1), (0.0075, 3), and (0.0125, 5). It again found that lower δT beat out more restarts, though 1 restart and 3 restarts were close. Three restarts seemed to best preserve the spirit of the paper, which mentions multi-restart. It also set δT=0.0025 to put Ja-be-Ja on equal footing with Jabar. This

required renting VMs from Google's Compute Engine to handle the extra runtime for running the experiments.

α: It tried 1, 1.5, 2, 2.5 and 3, and found that 3 produced the best results. In an informal set of tests, it found that values above 3 could produce better results, but when applied to some traces performed very poorly.

k (neighborhood): It tried 3, 7, 11, 15 and 25, and found that values of 15 performed well, and that anything above that took a long time.

Hermes

γ: It tried 1.01, 1.05, 1.1 and 1.15, and found that higher values uniformly yielded better results. 1.1 seemed best, because the marginal gains from increasing it were small and it seemed impractical to have imbalance greater than that.

k (max moves per partition per stage): It tried values from 1 to 5, and found that larger values generally yielded better results, but that above 3 there was a higher likelihood of getting caught in a multi-sequence loop where vertices would shuffle between two or more partitions in successive iterations.

$maxIterations$: It tried 10, 50, 100, 175 and 250, and found that values above 100 did not seem to improve things, but did take longer to run.

Jabar

$T_{initial}$: 2 seemed best, because Ja-be-Ja showed no improvement from higher values, and the authors of Ja-be-Ja suggested it [11].

δT: It tried 0.00125, 0.0025, 0.0075, and 0.0125. Unlike Ja-be-Ja, Jabar has only one parameter that yields clear benefits at the cost of run time. Values below 0.0025 produced unacceptably long runtimes, so 0.0025 was best.

α: It tried 1, 1.5, 2, 2.5 and 3, with results identical to Ja-be-Ja, yielding 3.

k (neighborhood): It tried 3, 7, 11, 15 and 25, and like Ja-be-Ja found that values of 15 performed well, and that anything above that took a long time.

Hermar

All parameters had essentially the same results as Hermes.

Spaja

$T_{initial}$: 2 seemed best, because Ja-be-Ja showed no improvement from higher values, and the authors of Ja-be-Ja suggested it [11].

δT: It tried 0.0025, 0.0075, 0.0125, 0.025, 0.05, 0.125, 0.25 and 0.5. In a result that defies explanation, Spaja performed best when it had higher values of δT, so 0.5 was best. It would make sense for more iterations to allow greater refinement of the partitions, but this did not prove to be the case in the training runs. It confirmed this with runs on other trace sets, so it does not appear to be a fluke.

α: It tried 1, 1.5, 2, 2.5 and 3, and found that 3 produced the best results.

k (neighborhood): It tried 3, 7, 11, 15 and 25, and much like Ja-be-Ja, it found that values of 15 performed well, and that anything above that took a long time.

Sparmes

γ: It tried 1.01, 1.05, 1.1 and 1.15, and it found that higher values uniformly yielded better results. Unlike Hermes and Hermar, where 1.1 was the best tradeoff between runtime and quality, 1.05 appeared to be the sweet spot.

k (max moves per partition per stage): It tried values from 1 to 5, and similarly to Hermes, larger numbers yielded better results, but there was a significant increase in vertex movement above k=3.

$maxIterations$: It tried 10, 50, 100, 175 and 250, and found that values above 100 did not seem to improve things, nor did they take longer to run, so 100 was best.

APPENDIX D - INSTALLING AND RUNNING VNTR

Installing and running VNTR requires a Java JDK, version 1.7 or later, Apache Maven, version 3 or later, and Git, version 2 or later. It has been tested with versions 1.8.0_131, 3.3.9, and 2.7.2, respectively. The project can be retrieved from github with the command, "git clone https://github.com/vntr-admin/combiner". From that point, following the instructions in README.txt should suffice.

Please note that the state-of-the-art solutions METIS and RMETIS require installation of the METIS software package, which may require compilation of the C source. Alternately, Debian-based Linux distributions can use the command, "apt-get install METIS".

The distribution only includes one trace set in the data directory. The rest of the trace sets are available as follows:

- Synthetic 70k: https://github.com/vntr-admin/combiner-dataset-synth_70k
- ASU Friendster: (distribution permission has not yet been granted by ASU)
- Facebook 63k: https://github.com/vntr-admin/combiner-dataset-mislove-facebook
- Vibrobox: https://github.com/vntr-admin/combiner-dataset-vibrobox
- Stiffness Matrix: https://github.com/vntr-admin/combiner-dataset-stiffness-matrix

For more information on these trace sets, see Appendix E.

APPENDIX E - TRACE SETS

This thesis uses of thirty-six traces, collected as six sets of six. Each trace consists of an initial graph, an initial partitioning of the vertices, a set of replicas cross the partitions that obey k-replication and neighbor-collocation constraints, and a series of operations of the seven types mentioned in the Social Network Middleware subsection of the Background section, e.g. user addition.

This thesis uses the following datasets for the initial graphs:

- The Leskovec-McAuley Facebook graph [24], used for parameter training
- The Mislove Facebook graph [22]
- The ASU Friendster graph [21], which had edges to vertices not present in the dataset, so they were omitted
- The Vibrobox graph [23], a Vibroacoustic problem
- The Stiffness Matrix graph [23], a stiffness matrix for an automobile chassis
- The Synthetic 70k graph, generated from a model as explained below

The synthetic graph comes from a group-based model from Newman [18] using 70,000 vertices, 1500 groups, an intra-group friendship probability of 5.5%, and a group membership probability of 0.22%. The paper provides an explanation of this model in the background section for the relation between the two probabilities and Newman's model. These numbers result in a group size of about 150, which has been experimentally found to be approximately the size of most true communities, known as the Dunbar Number [25, 26].

For each trace set it sets the number of vertices per partition to keep the total number of partitions between 32 and 128, similar to what appears in the literature [12].

Each trace set has two distinct initial partitionings, one generated randomly, called a hard start, and one generated with METIS 5.1.0's gpmetis command, called a soft start.

Each partitioning for each graph has three distinct replica partitions obeying 0-, 1- and 2-replication, respectively.

Finally, each trace set has a single series of operations that it uses across all six traces. These consist of 10,081 operations, which represent one per minute for a week starting at midnight, plus one for the endpoints to be inclusive. There are seven hard-coded DOWNTIMEs, one per day at 3:00 AM, plus two randomly-selected REMOVE_PARTITIONs to simulate server failures. ADD_PARTITIONs are triggered only when the average number of vertices per partition exceeded the maximum allowed amount shown in the $|\mathbb{V}|/|P|$ column of Table 15 at the end of Appendix E, either because of user addition or partition removal.

The remaining ~10,070 operations consist of adding and removing vertices and edges according to a simple random model. From the literature, it seems that in a mature social network, befriending is about three times as common as unfriending, as seen in Table 4 of Armstrong et al. [27]. This, together with $\Sigma = 1$ provides two of the four constraints necessary to uniquely determine the appropriate probabilities. The third constraint sets user addition to 3x the probability of user removal, similar to befriending/unfriending. Finally, each trace set sets the ratio of user operations to friending operations to achieve a slow growth in the average number of edges/vertex, in keeping with Leskovec's results on graph densification over time [28].

In retrospect, selecting vertices to remove randomly yielded few add/delete user operations, because when a vertex with many neighbors leaves, the number of edges plummets; intuitively, users with more friends seem less likely to leave.

Table 15 – Numerical Summary of Trace Sets

Initial Graph	$\lvert\mathbb{V}\rvert$	$\lvert\mathbb{E}\rvert$	Assortivity	Partition Drop 1	Partition Drop 2	Max $\lvert\mathbb{V}\rvert/\lvert P\rvert$	FriendOps / UserOps
Leskovec FB	4039	88234	0.06358	3278	6893	125	61.5
Mislove FB	63731	817090	0.1769	4134	5071	1000	30.25
Friendster	100199	981920	0.007669	1580	5370	1000	30.25
Vibrobox	12328	165250	0.4613	2315	7223	250	30.25
Stiffness Matrix	35588	572914	0.4682	984	9378	1000	40.67
Synthetic 70k	70000	984874	0.005437	4392	9287	1000	30.25

APPENDIX F – PSEUDOCODE FROM THE LITERATURE

Kernighan-Lin Partition Improvement [8]

```
//Let G be a graph consisting of a set V of vertices and E of Edges
//Let D[v] be the sum of edge weights to v's neighbors in the other
partition minus the sum of edge weights to v's neighbors in its
partition
//Let c(u, v) be a function that resolves to the weight of the edge
between u and v, or 0 if none exists
function Kernighan-Lin(G(V,E)):
  determine a balanced initial partition of the nodes into sets A, B
    do
      compute D values for all a in A and b in B
      let g_v, a_v, and b_v be empty lists
      for (n := 1 to |V|/2)
          find a∈A , b∈B, s.t. g = D[a] + D[b] - 2*c(a, b) is maximal
            remove a and b from further consideration in this pass
          add g to g_v, a to a_v, and b to b_v
          update D values for the elements of A = A\a and B = B\b
        end for
        find k which maximizes g_max, the sum of g_v[1],...,g_v[k]
        if (g_max > 0) then
          Exchange a_v[1],a_v[2],...,a_v[k] with b_v[1],b_v[2],...,b_v[k]
    until (g_max ≤ 0)
  return G(V,E)
```

Fiduccia-Mattheyes Partition Improvement

```
//Let V be the set of vertices, with initial partitions P1 and P2
//Let γ be the maximum ratio of any partition's weight to the average
partition's weight
FM()
    MOVE ← []
    GAIN ← []
    i ← 0
    while | MOVE | < |V|
        (u, u_gain) ← FIND_CANDIDATE (P1, MOVE)
        (v, v_gain) ← FIND_CANDIDATE (P2, MOVE)
        if( IMBALANCE(P1) > γ )
            MOVE[i] ← v
            GAIN[i] ← v_gain
        else if( IMBALANCE(P2) > γ )
            MOVE[i] ← u
            GAIN[i] ← u_gain
        else if(u_gain ≥ v_gain)
            MOVE[i] ← u
```

```
            GAIN[i] ← u_gain
        else
            MOVE[i] ← v
            GAIN[i] ← v_gain
        // would update here, that is handled elsewhere
        i ← i + 1

    gsum ← 0
    gsum_max ← 0
    i_max ← NULL
    for i=1:|MOVE|
        gsum ← gsum + GAIN[i]
        if(gsum > gsum_max)
            gsum_max ← gsum
            i_max ← i

    if(gsum_max > 0)
        move MOVE[1..i_max] to other partition

IMBALANCE(P)
    size_p ← | (P U (MOVE \ P)) \ (MOVE ∩ P) |
    size_avg ← |V|/2
    return size_p / size_avg

FIND_CANDIDATE(P, MOVE)
    CUR_P ← (P U (MOVE \ P)) \ (MOVE ∩ P)
    CUR_P' ← V \ CUR_P
    gain_max ← -∞
    v_max ← NULL
    for v : P \ MOVE
        neighbors_on_partition ← nbhd(v) ∩ CUR_P
        neighbors_off_partition ← nbhd(v) ∩ CUR_P'
        gain ← neighbors_off_partition - neighbors_on_partition
        if gain > gain_max
            gain_max ← gain
            v_max ← v
    return (v_max, gain_max)
```

Ja-be-Ja's Simulated Annealing

```
//Let T_initial be the initial temperature of the system
//Let δT be the change in decrease in temperature each iteration
//Let k be the number of swap candidates Ja-be-Ja considers at once
//Let α be the power to which Ja-be-Ja raises the comparison function
//Let T_r ← T_initial, initially
procedure SampleAndSwap
    partner ← findPartner(p.getRandomSampleOnPhysicalPartition(),T_r)
    if partner = null then
        partner ← findPartner(p.getRandomSampleAcrossAllPartitions(),T_r)
    end if
```

```
    if partner =/= null then
        handshake for color exchange between p and partner
    end if
    Tr ← Tr – δT
    if Tr < 1 then
        Tr ← 1
    end if
end procedure

function findPartner(Vertex[] vertices, float Tr)
    highest ← 0
    partnerbest ← null
    for q ∈ vertices do
        xpp ← p.getDegree(p.color)
        xqq ← q.getDegree(q.color)
        old ← xpp^α + xqq^α
        xpq ← p.getDegree(q.color)
        xqp ← q.getDegree(p.color)
        new ← xpq^α + xqp^α
        if (new × Tr > old) ∧ (new > highest) then
            partnerbest ← q
            highest ← new
        end if
    end for
    return partnerbest
end function
```

Hermes's Lightweight Repartitioner

```
//Let γ be the maximum ratio of any partition's weight to the average
partition's weight
//Let k be the maximum number of vertices moved from one partition in
one iteration
procedure REPARTITIONING_ITERATION(partition Ps)
    for stage ∈ {1, 2} do
        candidates ← {}
        for Vertex v ∈ VertexSet(Ps) do
            target(v) ← GET_TARGET_PART(v,stage)//set target(v), gain(v)
            if target(v) =/ null then
                candidates.add(v)
        top-k ← k candidates with maximum gains
        for Vertex v ∈ top-k do
            migrate(v, Ps , target(v))
        Ps.update_auxiliary_data
procedure GET_TARGET_PART(v in Ps, current stage of the iteration.)
    if imbalance factor(Ps \ {v}) < 2-γ then
        return (NULL, 0)
    target ← NULL
    gainmax ← 0
    if imbalance factor(Ps) > γ then
        gainmax ← -∞
```

```
for partition P_t ∈ partitionSet do
    if (stage=1 ∧ P_t.ID > P_s.ID) ∨ (stage=2 ∧ P_t.ID < P_s.ID) then
        gain ← Gain(v, P_s, P_t)
        if imbalance factor(P_t U {v}) < γ ∧ gain > gain_max then
            target ← P_t
            gain_max ← gain
return (target, gain_max)
```

REFERENCES

1. Yang, Shengqi, Xifeng Yan, Bo Zong, and Arijit Khan. "Towards effective partition management for large graphs." In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 517-528. ACM, 2012.

2. Friendster Lost Lead Because of a Failure to Scale," CNN.com, last modified November 13, 2007, http://highscalability.com/blog/2007/11/13/friendster-lost-lead-because-of-a-failure-to-scale.html.

3. Pujol, Josep M., Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. "The little engine (s) that could: scaling online social networks." ACM SIGCOMM Computer Communication Review 40, no. 4 (2010): 375-386.

4. Monien, Burkhard, and Ivan Hal Sudborough. "Min cut is NP-complete for edge weighted trees." In International Colloquium on Automata, Languages, and Programming, pp. 265-274. Springer, Berlin, Heidelberg, 1986.

5. Schloegel, Kirk, George Karypis, and Vipin Kumar. Graph partitioning for high performance scientific simulations. Army High Performance Computing Research Center, 2000.

6. Leskovec, Jure, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters." Internet Mathematics 6, no. 1 (2009): 29-123.

7. Kernighan, Brian W., and Shen Lin. "An efficient heuristic procedure for partitioning graphs." The Bell system technical journal 49, no. 2 (1970): 291-307.

8. Ravikumār, Si Pi. Parallel methods for VLSI layout design. Greenwood Publishing Group, 1996.

9. Fiduccia, Charles M., and Robert M. Mattheyses. "A linear-time heuristic for improving network partitions." In Papers on Twenty-five years of electronic design automation, pp. 241-247. ACM, 1988.

10. Karypis, G., and V. Kumar. "Parallel multilevel k-way partitioning scheme for irregular graphs, Department of Computer Science Tech. Rep. 96-036." University of Minnesota, Minneapolis, MN (1996).

11. Rahimian, Fatemeh, Amir H. Payberah, Sarunas Girdzijauskas, Mark Jelasity, and Seif Haridi. "Ja-be-ja: A distributed algorithm for balanced graph partitioning." In Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on, pp. 51-60. IEEE, 2013.

12. Nicoara, Daniel, Shahin Kamali, Khuzaima Daudjee, and Lei Chen. "Hermes: Dynamic Partitioning for Distributed Social Network Graph Databases." In EDBT, pp. 25-36. 2015.

13. Newman, Mark EJ, and Juyong Park. "Why social networks are different from other types of networks." Physical Review E 68, no. 3 (2003): 036122.

14. Gonzalez, Joseph E., Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs." In OSDI, vol. 12, no. 1, p. 2. 2012.

15. Travers, Jeffrey, and Stanley Milgram. "An experimental study of the small world problem." Sociometry (1969): 425-443.

16. Benevenuto, Fabrício, Tiago Rodrigues, Meeyoung Cha, and Virgílio Almeida. "Characterizing user behavior in online social networks." In Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference, pp. 49-62. ACM, 2009.

17. Andrew Charneski, “Modeling Network Latency”, Simia Cryptus Blog, last modified October 25, 2015, http://blog.simiacryptus.com/2015/10/modeling-network-latency.html.

18. Newman, Mark EJ. "Properties of highly clustered networks." Physical Review E 68, no. 2 (2003): 026121.

19. Mondal, Jayanta, and Amol Deshpande. "Managing large dynamic graphs efficiently." In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 145-156. ACM, 2012.

20. Vaquero, Luis M., Felix Cuadrado, Dionysios Logothetis, and Claudio Martella. "Adaptive partitioning for large-scale dynamic graphs." In Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on, pp. 144-153. IEEE, 2014.

21. R. Zafrani and H. Liu, Social Computing Data Repository at ASU, http://socialcomputing.asu.edu.

22. Viswanath, Bimal, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. "On the evolution of user interaction in facebook." In Proceedings of the 2nd ACM workshop on Online social networks, pp. 37-42. ACM, 2009.

23. Tim Davis and Yifan Hu, SuiteSparse Matrix Collection, https://www.cise.ufl.edu/research/sparse/matrices/

24. Leskovec, Jure, and Julian J. Mcauley. "Learning to discover social circles in ego networks." In Advances in neural information processing systems, pp. 539-547. 2012.

25. Hernando, A., D. Villuendas, C. Vesperinas, M. Abad, and A. Plastino. "Unravelling the size distribution of social groups with information theory in complex networks." The European Physical Journal B-Condensed Matter and Complex Systems 76, no. 1 (2010): 87-97.

26. Gjoka, Minas, Maciej Kurant, Carter T. Butts, and Athina Markopoulou. "Walking in facebook: A case study of unbiased sampling of osns." In Infocom, 2010 Proceedings IEEE, pp. 1-9. IEEE, 2010.

27. Armstrong, Timothy G., Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. "LinkBench: a database benchmark based on the Facebook social graph." In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 1185-1196. ACM, 2013.

28. Leskovec, Jure, Jon Kleinberg, and Christos Faloutsos. "Graph evolution: Densification and shrinking diameters." ACM Transactions on Knowledge Discovery from Data (TKDD) 1, no. 1 (2007): 2.