Balanced Label Propagation for Partitioning Massive Graphs

Johan Ugander
Center for Applied Mathematics
Cornell University
Ithaca, NY
jhu5@cornell.edu

Lars Backstrom
Facebook
Menlo Park, CA
lars@fb.com

ABSTRACT

Partitioning graphs at scale is a key challenge for any application that involves distributing a graph across disks, machines, or data centers. Graph partitioning is a very well studied problem with a rich literature, but existing algorithms typically can not scale to billions of edges, or can not provide guarantees about partition sizes.

In this work we introduce an efficient algorithm, *balanced label propagation*, for precisely partitioning massive graphs while greedily maximizing edge locality, the number of edges that are assigned to the same shard of a partition. By combining the computational efficiency of label propagation — where nodes are iteratively relabeled to the same 'label' as the plurality of their graph neighbors — with the guarantees of constrained optimization — guiding the propagation by a linear program constraining the partition sizes — our algorithm makes it practically possible to partition graphs with billions of edges.

Our algorithm is motivated by the challenge of performing graph predictions in a distributed system. Because this requires assigning each node in a graph to a physical machine with memory limitations, it is critically necessary to ensure the resulting partition shards do not overload any single machine.

We evaluate our algorithm for its partitioning performance on the Facebook social graph, and also study its performance when partitioning Facebook's 'People You May Know' service (PYMK), the distributed system responsible for the feature extraction and ranking of the friends-of-friends of all active Facebook users. In a live deployment, we observed average query times and average network traffic levels that were 50.5% and 37.1% (respectively) when compared to the previous naive random sharding.

Categories and Subject Descriptors

G.2.2 [**Mathematics of Computing**]: Discrete Mathematics—*Graph Theory, Graph Algorithms*

WSDM'13, February 4–8, 2013, Rome, Italy.

General Terms

Algorithms, Measurement

Keywords

graph clustering, graph partitioning, label propagation, social networks

1. INTRODUCTION

In Plato's *Phaedrus*, Socrates tells us that a key principle of rhetoric is the ability to divide ideas "where the natural joints are, and not trying to break any part, after the manner of a bad carver" [15]. This, too, is the goal of graph partitioning: breaking a graph at its natural joints.

In this work, we present an algorithm for finding 'joints' in graphs of particularly massive proportions, with an emphasis on the Facebook social graph consisting of over 800 million nodes and over 68 billion edges [19]. The algorithm we present uses label propagation to relocate inefficiently assigned nodes while respecting strict shard balancing constraints. We show how this *balanced label propagation algorithm* can be formulated as a convex optimization problem that reduces to a manageable linear programming problem. The algorithm is fundamentally iterative, where each iteration entails solving a linear program.

While we find that our algorithm performs well under random initialization, by initializing the algorithm with a greedy geographic assignment, we find that it is possible to effectively achieve convergence within a single step of the update algorithm, while random initialization requires many iterations to slowly converges to a less performative solution.

Our algorithm has the ability to follow arbitrary partition size specifications, a generalization of the more common goal of symmetric partitioning [10]. This functionality makes it possible to use the greedy efficiency of label propagation when considering partitions that are not symmetric by creation, but might still benefit from label propagation. Specifically, the geographic initialization we consider is not symmetric, yet balanced label propagation can still be used to considerably improve the partitioning.

Label propagation unfortunately offers no formal performance guarantees with respect to the graph partitioning objective, and neither does our modification. It is worth remarking that the basic problem of constrained graph partitioning, bisecting a graph into two equal parts with as few crossing edges as possible, is the well known NP-hard *minimum bisection problem* [8], with the best known approximation algorithm being a $O(\sqrt{n}\log n)$-factor approximation [6]. Knowing this, our goal has been to develop a highly scalable

algorithm that can deliver precise partition size guarantees while performing well at maximizing edge locality in practice.

People You May Know. After presenting our algorithm and analyzing its performance on both the Facebook social graph and a LiveJournal graph, we then present the results of a full deployment of the partitioning scheme for the Facebook friend recommendation system, 'People You May Know' (PYMK). This system computes, for a given user u, and each friend-of-friend (FoF) w of u a feature vector $x_{u,w}$ of graph metrics based on the local structure between u and w. The system then uses machine learning to rank all the suggestions w for u, based on the feature vector $x_{u,w}$, as well as demographic features of u and w.

Due to the size of the graph and high query volume, Facebook has built a customized system for performing this task. Each user u is assigned to a specific machine m_u (this assignment is what we are optimizing). When a FoF query is issued for a user u, it must be sent to machine m_u. Then, for each $v \in N(u)$, a query must be issued to the machine m_v, to retrieve the nodes two hops from u. The results of these queries are then aggregated to compute the features that are input to the machine learning phase, which outputs the final ranked list of FoFs.

Our goal is to perform our graph sharding such that, as often as possible, $m_v = m_u$ for $v \in N(u)$. By doing this well, we can reduce the number of other machines that we need to query, and also reduce the total amount of data that needs to be transferred over the network, increasing overall system throughput and latency. We are constrained by the fact that machines have a limited amount of memory, which puts a hard cap on the size of each shard.

At the outset, it is important to note that it is not actually clear that a sophisticated sharding will be a performance win in this application. When naively sharding a graph, one typical approach is to assign nodes to machines by simply taking the modulus of the user ID; see Figure 1. The advantage of this approach is that the machine location of a node is directly encoded in the user ID. Using a non-trivial sharding requires extensive additional lookups in a *shard map*, as well as an additional network operation at the start of the query where the initial request is forwarded to the appropriate machine hosting the subject of the query. The shard map for all 800 million nodes must also be mirrored in memory across all machines.

As we will see in our ultimate demonstration, the cost of this additional complexity is greatly overshadowed by the improvements in locality that can be had. The novel graph sharding algorithm we introduce, combined with the intelligent initialization based on geographic metadata, is able to produce a sharding across 78 machines where 75.2% of edges are local to individual machines. In the conclusion to this work, we deploy the resulting sharding on Facebook's 'People You May Know' realtime service, and observe dramatic performance gains in a realtime environment.

The main distinction when designing an algorithm applicable at Facebook's scale is that the full graph can not be easily stored in memory. Realistically, this means that the only admissible algorithms are those that examine the graph in streaming iterations. One iteration of our balanced label propagation algorithm takes a single aggregating pass over the edge list, executable in MapReduce, and then solves a linear program with complexity dependent on the number of machines, not the size of the graph. A final assignment step takes a single streaming pass over the nodes, again in MapReduce, to redefine their assignments. Recent work on streaming graph partitioning [17] produced notable performance using a single greedy iteration, but the results were obtained within a framework that did not naturally lend itself to additional iteration.

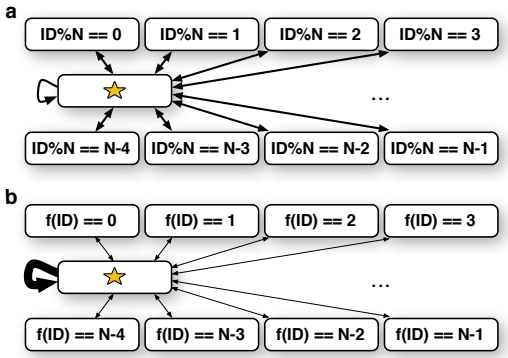

Figure 1: Sharding the neighbors of a node across N machines. (a) Aggregating properties of the neighbors when the edge list is sharded according to node ID mod N. (b) Aggregating when the edge list is sharded according to a shardmap f. The goal of an efficient shard map is to greatly increase the likelihood that a node is located on the same shard as its neighbors.

Organization of the paper. Section 2 presents the details of the balanced label propagation algorithm. Section 3 discusses a geographic initialization that, when possible, greatly improves performance. In Section 4, we evaluate the algorithm and its ability to shard the Facebook social graph and a publicly availably LiveJournal graph dataset. In Section 5, we study a deployment of the sharding to load-balance Facebook's PYMK service. We conclude with Section 6 by discussing future directions for work on this problem.

2. BALANCED LABEL PROPAGATION

Label propagation was first proposed as an efficient method for learning missing labels for graph data in a semi-supervised setting [20]. In such a setting, unlabeled nodes iteratively adopt the label of the plurality of their neighbors until convergence, making it possible to infer a broad range of traits that are fundamentally assortative along the edges of a graph.

In a context very similar to graph partitioning, label propagation has been found to be a very efficient technique for network community detection [16], the challenge of finding naturally dense network clusters in an unlabeled graph [7]. In this context, each node of a graph is initialized with an individual label, and label propagation is iterated where nodes again update their labels to the plurality of their neighbor's labels.

Network community detection and graph partitioning are very similar challenges, with two key differences. First, community detection algorithms need not and should not require *a priori* specification of the number or size of graph communities to find. Second, community detection algorithms ought to support overlapping communities, while graph par-

titions seek explicitly disjoint structure. While some partitioning applications may benefit from assigning nodes to multiple partitions, the Facebook social graph we aim to partition has a modest maximum degree of $5,000$, and so we restrict our investigation to creating true partitions.

Label propagation fails to detect overlapping communities [7], though an adaptation does exist [9]. But this failure of the ordinary label propagation algorithm in fact makes it a strong candidate for graph partitioning. The remaining difficulty is therefore that label propagation provides no way of constraining the sizes of any of the resulting community partitions. Our contribution address precisely this difficulty.

A previous attempt to 'constrain' label propagation utilizes an optimization framework with a cost penalty to encourage balanced partitions [3], but this approach does not offer constraints in a formal sense. The constraint-based algorithm we introduce in this work offers the possibility of precisely constraining the size of all the resulting shards – it is in fact not limited to constraints that produce balanced partitions. It is also worth noting that the previous cost penalty approach lacks the computational efficiency of label propagation, and the largest graph that framework was originally tested on contained just 120,000 edges.

2.1 Partition constraints

The goal of our balanced label propagation algorithm is to take a graph $G = (V, E)$ and produce a partition $\{V_1, \ldots, V_n\}$ of V, subject to explicitly defined size constraints. The algorithm is capable of enforcing arbitrary size constraints in the form of lower bounds S_i and upper bounds T_i, such that $S_i \leq |V_i| \leq T_i$, $\forall i$. These constraints can easily take the form of balanced constraints, targeting exact balance $S_i = \lfloor |V|/n \rfloor$ and $T_i = \lceil |V|/n \rceil$, $\forall i$, or operating with leniency, $S_i = \lfloor (1-f)|V|/n \rfloor$ and $T_i = \lceil (1+f)|V|/n \rceil$, for some fraction $f > 0$.

To initialize the algorithm, we begin by randomly assigning nodes to shards, in proportions that are feasible with respect to these sizing constraints.

2.2 The constrained relocation problem

Given an initial feasible sharding, we wish to maintain the specified balance of nodes across shards between iterations. The key challenge is however that some shards will be more popular than others. In fact, under ordinary label propagation without any balance constraints, labelling all nodes with the same single label is a trivial equilibrium. Because we won't be able to move all nodes, our greedy approach is to sychronously move those nodes that stand to increase their colocation count (the number of graph neighbors they are co-located with) the most.

Given a constraint specification, we now formalize our greedy relocation strategy as a maximization problem subject to the above constraints. Consider therefore the nodes that are assigned to shard i but would prefer to be on shard j. Order these nodes according to the number of additional neighbors they would be co-located with if they moved, from greatest increase to least increase, labeling them $k = 1, \ldots, K$. Let $u_{ij}(k)$ be the change in utility (co-location count) from moving the kth node from shard i to j.

Let $f_{ij}(x) = \sum_{k=1}^{x} u_{ij}(k)$ be the *relocation utility function* between shard i and j, the total utility gained from moving the leading x nodes from i to j. Observe that because $u_{ij}(k) \geq 0$ and $u_{ij}(k) \geq u_{ij}(k+1)$ for all k (since they are ordered), all $f_{ij}(x)$ are increasing and concave.

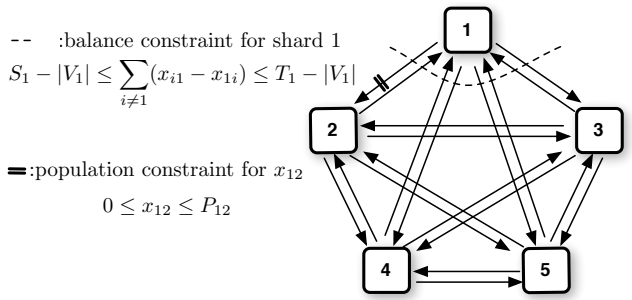

Figure 2: Illustration of the constraints for balanced label propagation applied to five shards. Each shard has a two-sided balance constraint, while each pair of shards has a population constraint.

Our goal can then be formulated as a concave utility maximization problem with linear constraints.

PROBLEM 1 (CONSTRAINED RELOCATION). *Given a graph $G = (V, E)$ with the node set partitioned into n shards $V_1, \ldots, V_n$, and size constraints $S_i \leq |V_i| \leq T_i$, $\forall i$, the constrained relocation problem is to maximize:*

$$\max_X \textstyle\sum_{i,j} f_{ij}(x_{ij}) \qquad s.t. \tag{1}$$

$$\begin{array}{rcccll} S_i - |V_i| & \leq & \sum_{j \neq i}(x_{ij} - x_{ji}) & \leq & T_i - |V_i|, & \forall i \\ 0 & \leq & x_{ij} & \leq & P_{ij}, & \forall i, j. \end{array} \tag{2}$$

Here P_{ij} is the number of nodes that desire to move from shard i to j, and $f_{ij}(x)$ is the relocation utility function between shard i and j, both derivable from the graph and the partition.

For an illustration of the constraints, see Figure 2.

When the above problem is considered under continuous values of x_{ij}, we will now show that it reduces to a fully tractable optimization problem. Note that relaxing these integrality constraints on all x_{ij} is a fully reasonable approximation as long as the number of nodes seeking to be moved between each pair of shards is large.

We aim to rewrite the above optimization problem as a linear program. Notice that all f_{ij} are piecewise-linear concave functions. To see this, notice that the slope of f_{ij} is constant across all intervals where the ordered users have the same derived utility. As a consequence of this, we obtain the following straight-forward lemma.

LEMMA 1. *Assuming a bounded degree graph G, the objective function in Problem 1, $f(x) = \sum_{i,j} f_{ij}(x_{ij})$, is a piecewise-linear concave function, seperable in x_{ij}.*

PROOF. The separability is clear from the fact that the f_{ij} depend on different variables. Since users are atomic and the graph is of bounded degree, there are a finite number of utilities, and the sum is therefore also piecewise linear. Since the nodes are sorted in order of decreasing utility, the function is concave. □

Recall that any piecewise linear concave function $f(x) : \mathbb{R}^n \to \mathbb{R}$ can be written as $f(x) = \min_{k=1,\ldots,\ell}(a_k^T x + b_k)$, for some choices of a_k's and b_k's. For our problem, all the a_k's and b_k's are scalar. Now, also recall the following [5].

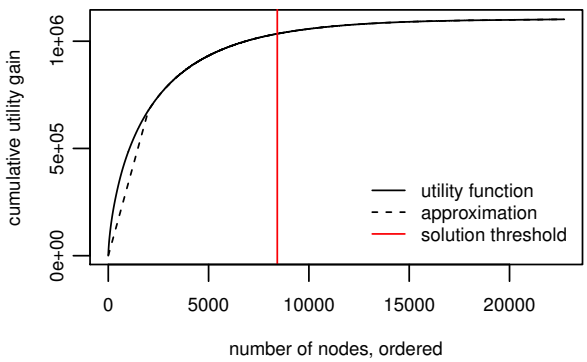

Figure 3: The piecewise-linear utility function for moving nodes from one shard to another, from an example problem iteration. The discontinuous derivates are imperceivable. The utilty approximation shown allows for a significant reduction in the number of constraints in the LP. The red line indicates the threshold found in the optimal solution for balanced propagation: here 8,424 of the 22,728 nodes that wanted to move were moved.

LEMMA 2. *Let $x \in \mathbb{R}^n$ and $f(x) = \min_{k=1,\ldots,\ell}(a_k^T x + b_k)$ be a piecewise linear concave function. Maximizing $f(x)$ subject to $Ax \leq b$ is then equivalent to:*

$$\max z \qquad s.t. \tag{3}$$

$$\begin{cases} Ax & \leq & b \\ a_k^T x + b_k & \geq & z, \quad \forall k. \end{cases} \tag{4}$$

Utilizing these two lemmas, we can thus solve the concave maximization problem in Problem 1 using a linear program.

THEOREM 2. *Consider a bounded degree graph $G = (V, E)$. Under continuous x_{ij}, the constrained relocation problem can be written as*

$$\max_{X,Z} \textstyle\sum_{i,j} z_{ij} \qquad s.t. \tag{5}$$

$$\begin{array}{rcccl} S_i - |V_i| & \leq & \sum_{j \neq i}(x_{ij} - x_{ji}) & \leq & T_i - |V_i|, \quad \forall i \\ 0 & \leq & x_{ij} & \leq & P_{ij}, \quad \forall i,j \\ & & -a_{ijk}x_{ij} + z_{ij} & \leq & b_{ijk}, \quad \forall i,j,k, \end{array} \tag{6}$$

where all a_{ijk} and b_{ijk} derive directly from the relocation utility functions f_{ij}. Assuming n shards and at most K unique utility gains achieved by nodes that would like to move, this constitutes a linear program with $2n(n-1)$ variables and at most $2n^2 + Kn(n-1)$ sparse constraints.

Since the most utility a node can gain is its degree, and furthermore only a small set of nodes in real world graphs have high degree, it becomes unlikely that all pairs of shards will observe nodes seeking large utility gains. Thus, in practice the number of constraints is typically small. For $m = 78$ and $K = 100$, this would imply a linear program with 12,012 variables and 612,768 constraints, which is fully manageable by a basic LP solver owing to the extensive sparsity of the matrix of constraints.

2.3 Iteration

Procedurally, an iteration of this algorithm differs very little from an iteration of ordinary label propagation. First, determine where every node would prefer to move, and how much each node would gain from its preferred relocation. Second, sort the node gains for each shard pair and construct the Constrained Relocation linear program. Third, solve the linear program, which determines how many nodes should be moved, in order, between each shard pair. Fourth, move these nodes. This constitutes one iteration.

When compared to ordinary label propagation, the only difference is that rather than moving every node that asks to move, our algorithm pauses, solves a linear program, and then proceeds to move as many nodes as possible without breaking the balance. As with ordinary label propagation, the bulk of the work lies in determining where every node would prefer to move (which requires examining every edge in the graph). The entire balancing procedure has a complexity that depends principally on the number of shards and is nearly independent of the graph size (the number of constraints per shard pair, K, can depend weakly on the graph size in practice).

2.4 Approximating utility gain

If the number of constraints becomes limiting, we note that it is possible to greatly reduce the number of constraints by a very slight approximation of the objective function. We emphasize that constraint satisfaction is still guaranteed under this approximation, it is merely the utility gain of the iteration that is approximated.

Observe that for each shard pair, the handful of nodes that stand to gain the most are likely to be contribute relatively unique utility levels, and so contribute many of the constraints in the problem. This is rather unnecessary, since those nodes are highly likely to move. Thus, by disregarding the unique utility levels of the first C nodes, all very likely to move, and approximating them by the mean gain of this population, we can greatly reduce the number of constraints. To exemplify this approximation, in Figure 3 we show one of the piecewise linear concave utility functions from a problem instance, corresponding to movement between two shards, and the threshold on the number of users that were allowed to be moved in the optimal solution.

3. GEOGRAPHIC INITIALIZATION

When applying greedy algorithms to intricate objective functions, initialization can dramatically impact performance. For the balanced label propagation algorithm presented in the previous section, random initialization — in proportions that are feasible with respect to the sizing constraints — might be considered an adequate initialization. However, by using node data to initialize our assignment, we find that we can improve on both the number of iterations and the final sharding quality. How well one can do in this initial assignment depends on what auxiliary node information is available. For example, on the web one might use domain, or in a computer network one might use IP-address. In the case of the Facebook social network, we find that geographical information gives us good initial conditions. Thus, in this section we will examine an initial graph partitioning based on a geographic partitioning in the absence of any graph structure.

Facebook's geolocation services assign all 800 million users to one of approximately 750,000 cities wordwide. The idea behind our geographic partitioning is to harness the intuitive and well-studied properties of geography as a strong basis for graph assortativity in social networks: individuals have an

elevated tendency to be friends with people geographically close to them [2].

A key challenge, however, when partitioning a realtime service based on geographic information, is the heterogeneity of traffic between geographies. When balanced propagation is run under random initialization, it is our experience that the local improvements made by the algorithm tend to not discover any large-scale geographic structure. Yet some parts of the world are much more active on Facebook than others, and as a result, they have more friends and their friend-of-friend calculations are much more expensive. As a result, it is desirable to configure the geographic initialization so that shards with higher than average degree contain fewer nodes, and shards with a lower than average degree contain more.

To achieve this, instead of cutting up the geographic space to form shards of exactly equal population, we consider a more general cost model, based on the number of users in each city and also on the average degree of users in that city. Note that in this particular sharding challenge we are most concerned with sharding node attributes, and the distribution of the graph (as an edge list) is not as central a concern, but instead we are considering average degree as a means of distributing computational load. For each city c with population n_c and average degree d_c, the cost of the city is given by $\text{Cost}(c) = n_c(1 + \lambda d_c)$, where λ is a weight parameter determined by the proportion of node attribute data to edge attribute data to be sharded. For our applications we used $\lambda = 1/d_{avg}$, the reciprocal of the average degree across the graph.

3.1 Balloon partitioning algorithm

The partitioning algorithm we will present here constructs shards centered at the most populated cities in the data set, beginning with the most populated city not yet assigned to a shard, and grows circular 'balloons' around these cities.

This balloon algorithm, illustrated in Figure 4, consists of a single iterated loop. For each of the $i = 1, \ldots, n$ shards, the goal is to obtain shards each with $(\sum_c \text{Cost}(c))/n$ in cost assigned to them. The algorithm first finds the city C with the largest unassigned cost. The city C is selected as the center-point of a new shard, and all cities with a non-zero remaining cost are sorted according to their geodesic distance from C. Since edges in the social network are overwhelmingly internal to countries, a negative distance reward is introduced to all cities in the same country as C. This ensures that the algorithm finishes assigning each country completely before moving on to another country. Beginning with C, the algorithm progresses down the sorted list of cities, and while there is capacity, it assigns each city to the shard currently being constructed. When the algorithm does change countries, a new negative distance reward is given to cities in that country. Eventually, when there is not enough capacity in the current shard to accommodate an entire city, a fractional assignment is noted, and the cost is subtracted from the remaining cost of that city.

The result is n shards, centered over population centers, each containing all nodes within a certain radius of the central city, adaptively configured such that each shard is of equal burden under the cost model. A handful of cities are assigned to multiple shards according to a fractional division. The map of cities to shards is used to assign nodes to shards, and for those cities with distributed fractional assignments (at most n of the 750,000 cities), simple randomization is used, distributing nodes in such cities proportional to the fraction of the cost mapped to each shard. A map of the Facebook social graph partitioned into 234 partitions is shown in Figure 6.

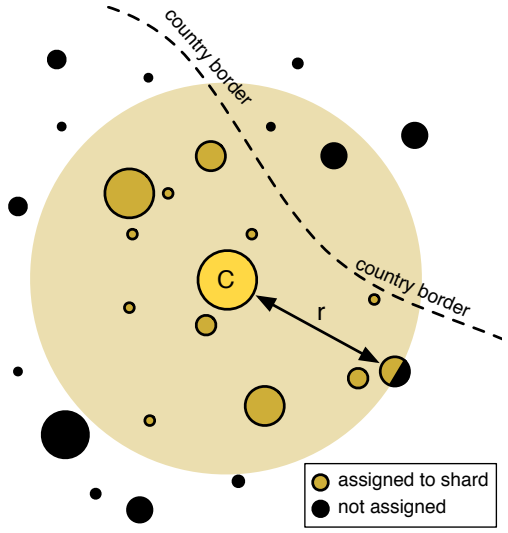

Figure 4: Geometric illustration of the greedy geographic initialization of a shard, with cities as circles with radii indicating cost. The algorithm centers itself at the most costly remaining city, C, and then fills a shard with the cities closest to that city, with a penalty for crossing national borders. When the shard is full, fractional assignments are made.

After performing an initial aggregation of the social graph, this assignment algorithm operates only on the set of cities and not on the social graph itself, making it possible to quickly run this complete assignment on a single machine in a single processor thread in seconds, with no actual graph analysis being necessary.

It should be noted that the geographic sharding performed here is ultimately static, and as new users register to join the site, these users can be assigned in accordance with the map from cities to shards. We note that as the geographic distribution of Facebook users slowly changes over time, re-sharding may be useful.

3.2 Oversharding

Selecting shards for a real-time graph computation service from a geographic initialization has another serious practical challenge that we have not yet discussed. For the purposes of load-balancing a service that handles real-time requests, it is important to mitigate potentially problematic peak loads that result from assigning geographically concentrated regions to the same shard. A three day window of user traffic is shown in Figure 5, where we see that local geographic regions experience much more volatile peak loads than the full site on average. Our solution to this problem was to create 3 times more shards than there are machines, and then sort shards by the longitude of their most populated city. The shards are then distributed cyclically across the n machines so that, e.g., shard 1, $(n + 1)$, and $(2n + 1)$ in longitudinal order are assigned to the same machine.

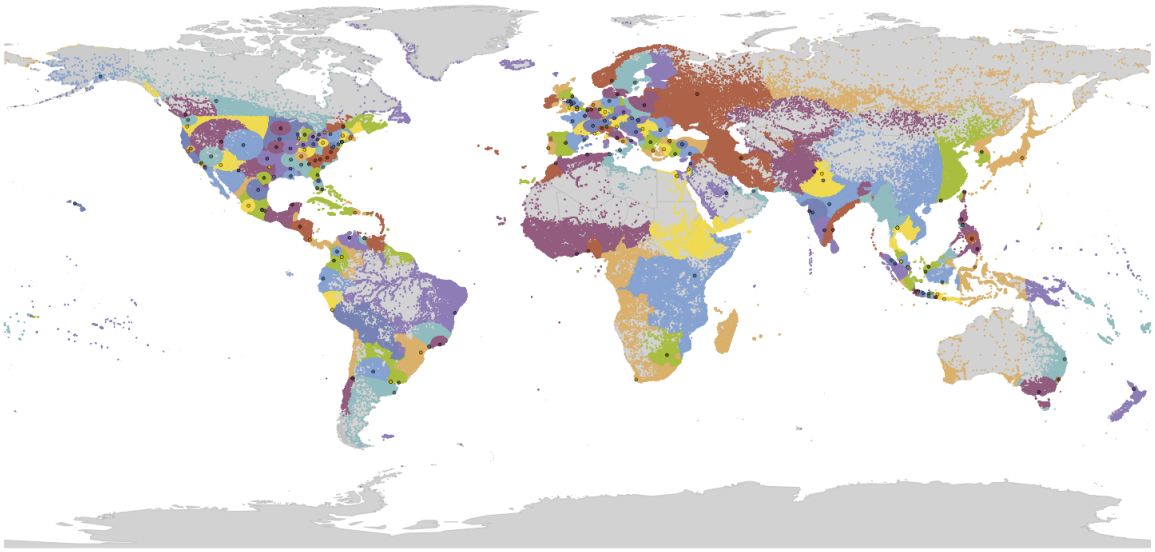

Figure 6: Output of the greedy geographic initialization algorithm for the $\sim 750,000$ known cities, obtaining 234 shards of equal cost, with each shard's most costly city marked. As described in the text, the algorithm is aware of national borders.

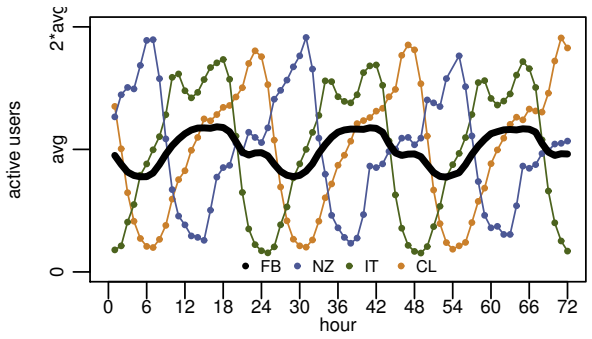

Figure 5: User traffic differences between countries. Comparing traffic for New Zealand, Italy, and Chile to that of Facebook as a whole, the intraday variability in users accessing the site from a single country far exceeds the variability of the site as whole.

4. EVALUATING PERFORMANCE

In this section we discuss the performance of our balanced label propagation algorithm as a graph cutting procedure applied to the Facebook social graph, under both random and geographic initialization. In Section 5 we evaluate shardings of this graph when deployed for a realtime graph computation service. To align the discussion between the two sections, we evaluate the algorithm by sharding the graph into 78 shards, where the service we evaluate later will consist of 78 machines, evenly split across two server racks. We also provide a comparison to performance on a publicly available social graph collected from LiveJournal [1].

4.1 Sharding the Facebook social graph

For the random initialization, all shards of the partition were constrained to symmetrically balanced node counts. Meanwhile, for the geographic initialization, partition size constraints were inherited from the output of the geographic balloon algorithm initialization, where partition node counts were tuned to mitigate the differences in average degree between different parts of the globe, as discussed in the previous section. All iterations were allowed a five percent leniency, $f = 0.05$, and utility approximation was used within the constrained relocation problem for the leading $5,000$ nodes between each shard pair.

The matrix of shard-shard edge counts resulting from both geographic and random initialization are shown in Figure 7, while the convergence properties observed when iterating the algorithm are shown in Figure 8. We observe that initializing the algorithm with a greedy geographic assignment greatly accelerates the convergence of the algorithm. Using geographic initialization, before even beginning the propagation we see that 52.7% of edges are locally confined. After a single propagation, fully 71.5% of edges are local, and after four iterations this rises to 75.2%.

For the geographic initialization instance, we overshard by a factor of 3, meaning that 234 virtual shards were distributed to 78 actual shards. When oversharding for distributed computation, it is useful to distribute closely related shards across machines located on the same physical rack. While this does not effect the fraction of edges that are local to individual machines, the order in which the shards are assigned does effect the cross-rack traffic. Machines within the same rack have relatively fast high-bandwidth connections compared with machine in different racks, where all traffic must pass through a switch. Since the service we study later is split across two racks, we distribute our 234 geographic shards across the 78 machine shards by splitting them into two block groups. The first block was set to contain all shards centered in countries in North America, Africa and Oceania, and the second block contains all shards centered in South America, Europe and Asia, with shards centered in the Middle East balancing between the two sets. The two-block structure is clearly visible in Figure 7, but we reiterate that it does not effect the local edge fraction.

For comparison, we investigate the performance of the algorithm when initialized with a random distribution, also

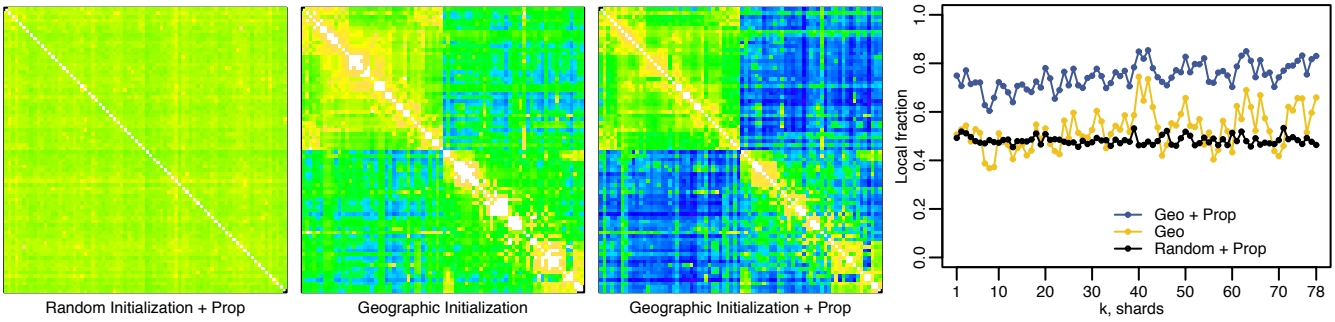

Figure 7: Matrices of edges between 78 shards under three different shardings: Balanced label propagation with random initialization, the sharding produced by the geographic initialization, and balanced label propagation after geographic initialization. All matrices share the same logarithmic color scale, saturated to make the structure of the random initialization visible. The fraction of local edges for each shard is also shown.

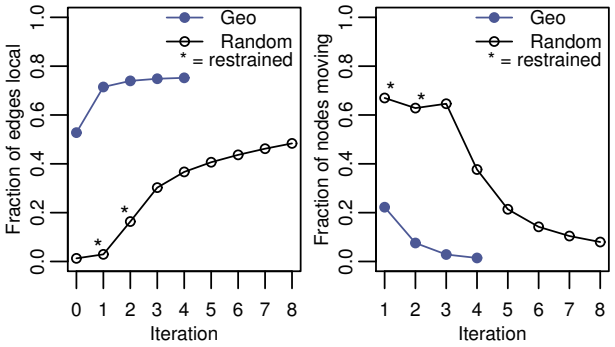

Figure 8: Iterating the balanced label propagation algorithm with 78 shards, from a geographic and random initialization. Left: the fraction of edges that are local as the balanced propagation is iterated. Right: the fraction of nodes that are moved in each iteration.

shown in Figure 8. After 8 iterations, this random initialization achieved a local fraction of 44.8%.

When nodes are initially distributed at random, it implies that a node's neighbors are initially distributed uniformly across all shards. For graphs where nodes possess many neighbors, as in the Facebook social graph, this implies that it can take many iterations until the initial symmetry of the random initialization is broken. In an important demonstration of how label propagation functions, we observed that applying our balanced label propagation algorithm from a random initial condition meant that 96.7% of nodes were relocated during the first iteration, a chaotic shuffling that slows the algorithm's ability to converge. To address this, in the random initialization shown here, the linear program constraints were modified to only move nodes that claimed to gain 2 or more additional neighbors post-propagation. This 'restrained' modification meant that only 67.0% of nodes were relocated during the first iteration. This 'restraint' was used during the first two steps of the random initialization algorithm, as denoted in the Figure 8, after which it was removed and ordinary balanced label propagation was performed.

Why should one bother to move nodes that only stand to gain one additional neighbor co-location? As another instructive highlight of how balanced label propagation operates, nodes that are mostly indifferent to moving offer very useful 'slack' for the linear program: by moving them, it is possible to balance the constraints while still moving many nodes who stand to make larger gains.

The Facebook social graph is enormous, and these computations do not come cheaply. The graph aggregations required for a single iteration utilizes approximately 100 CPU days (2395 CPU hours) on Facebook's Hadoop cluster. For comparison, a simple two-sided join aggregation of the graph edge list (such as computing a shard-shard matrix in Figure 7) uses 72 CPU days. Once the aggregations have been performed, the linear program is solved in a matter of minutes on a single machine using lpsolve [4].

While the randomly initialized algorithm only achieves 44.8% locality compared to 75.2% locality for the geographic initialization, the random initialization produces an impressively homogenous sharding free of 'hot' shard-shard connections. It would be interesting to iterate the random initial algorithm further, but running the random initialization for 8 iteration utilized approximately 800 CPU days. Examining the properties of balanced label propagation more throughly on modest graphs would be important future work.

4.2 LiveJournal comparison

Because the Facebook social graph is not publicly available, we also report the performance of our algorithm when attempting to partition a large publicly available social network dataset, LiveJournal [1]. The LiveJournal graph, collected in 2006, is a directed graph consisting of of 4.8 million nodes and 69.0 million arcs. Because we are principally interested in partitioning undirected graphs, we consider the undirected graph consisting of 4.8 million nodes and the 42.9 million unique edges that remain and disregarding directionality.

The resulting graph is more than 20 times smaller than the Facebook graph by node count and more than 1000 times smaller by edge count. The average degree is only 8.8, making partitioning much less challenging. Because the public LiveJournal graph lacks complete geographic information, we consider only random initialization. We consider the performance of cutting the graph into 20, 40, 60, 80, and 100 symmetric shards, with five percent leniency ($f = 0.05$) and no approximation of the utility gain. Splitting into 20 shards took less than 3 minutes using a single threaded C++ im-

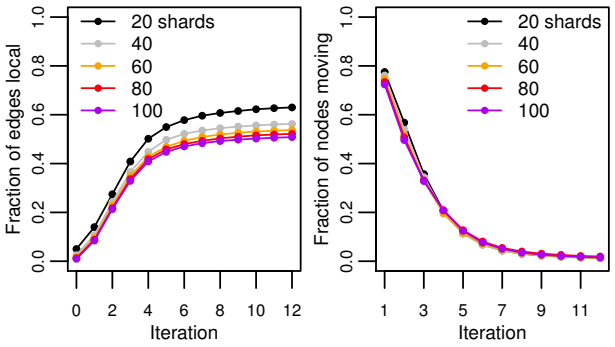

Figure 9: Balanced label propagation applied the LiveJournal social graph, partitioning the graph into 20, 40, 60, 80, and 100 shards. Left: the fraction of edges that are local as the balanced propagation is iterated. Right: the fraction of nodes that are moved in each iteration.

plementation, while splitting into 40 shards took 8 minutes, and splitting into 100 shards took 88 minutes.

The results from partitioning the LiveJournal graph are shown in Figure 9, where we see that when partitioning the LiveJournal graph into 20 parts, 63% of edges are local to a single partition. Impressively, when the algorithm is scaled up to 100 shards, fully 51% of edges are local to a single partition. Overall, we observe that the fraction of edges that are local is nearly unchanged when increasing the number of shards from 40 to 100. We interpret this performance to be a consequence of the greedy nature of the algorithm, as the algorithm principally exploits local graph relocations that are significantly below the scale of any of the shard sizes, while global improvements are less possible. A further investigation of this scaling behavior in relation to studies of natural social network cluster sizes [11] would make for interesting future work.

5. REALTIME DEPLOYMENT

In this section, we present the results of a large-scale experiment where the sharding algorithm we develop is evaluated in a realtime distributed graph computation service: Facebook's 'People You May Know' (PYMK) service for suggesting friend recommendations.

Many pages on Facebook occasionally feature a small module presenting users with 'People You May Know'. The PYMK service has contributed significantly to the growth of Facebook, accounting for around 40% of all friending on Facebook. The friend suggestions that populate this module are mostly (but not exclusively) generated by the system described in this work.

5.1 People You May Know

The PYMK system computes, for a given user u, and each friend-of-friend (FoF) of u, w, a feature vector $x_{u,w}$ of graph metrics based on the local structure between u and w. The system then uses machine learning to rank all the suggestions w for u, based on the graph-based feature vector $x_{u,w}$, as well as demographic features of u and w. The system described here returns the top 100 suggestions for each user (out of a potential of many thousands of FoFs). Regeneration of suggestions is performed when a user logs in to Facebook and no recent suggestions for the user are found in the cache.

The PYMK service consists of 78 machines split across two racks (there are 40 machines to a rack, but one machine per rack is reserved as a backup). All 78 machines feature 72 GB of memory, which is used to store two in-memory indexes. First, a mirrored data structure containing basic demographic data of all 800 million users (19 GB). Second, a sharded index containing the friendlist data of those users assigned to the individual machine (∼40 GB). All in-memory indexes are stored as open-addressed hash tables. To handle the full query volume, a number of identical copies of this 78 machine system run in parallel.

Prior to the optimizations presented in this paper, users were assigned to machines based on their user ID mod 78, see Figure 1. This naive sharding has a direct advantage over any sophisticated sharding in that the shard ID is encoded directly in the user ID. Introducing the more sophisticated shardings used in this paper requires adding an additional data structure serving as a *shard map*, mirrored on all machines, to map the 800 million user IDs to shard IDs.

The evaluation we perform examined three separate instances of the PYMK service operating in parallel, receiving identical and evenly balanced shares of the service load, differing only with regard to their sharding configuration. The three systems were:

- **Baseline sharding**: assigning users by the modulus sharding, 'user ID % 78'.
- **Geographic sharding**: assigning users using 234 geographic shards, with no label propagation.
- **Propagated sharding**: one step of balanced label propagation after geographic initialization.

Because of resource constraints we were only able to test three parallel systems, and did not deploy a propagated sharding featuring random initialization, which required much more iteration and achieved worse edge localization than the unpropogated geographic initialization.

First, we characterize the impact of our sharding by evaluating our algorithm's ability to concentrate requests to few machines, reporting on the number of machines queried across requests. Next, because intelligent sharding disadvantages the PYMK service by requiring an additional round of requests (as described in the introduction), we evaluate the algorithm's ability to reduce to total query time of FoF requests, with particular attention to the slowest machine, as well as the ability to reduce the total cross-machine network traffic within the full system.

The evaluation was performed during the three day period September 20-22, 2011. Because the evaluation required the dedication of considerable hardware resources (6 racks of machines, in total 240 machines), testing on exactly identical hardware configurations was not possible (it was important to test the three systems in parallel so that no external events could impact the results). All machines featured 72GB of memory, while the machines in the baseline system and the geographic system featured 12 CPUs and the machines in the propagated system featured 24 CPUs, all 2.67 GHz Intel Xeon processors. This difference of CPU resources is one of the main reasons we focus our analysis on hardware invariant performance evaluations such as request concentration and network traffic measurements.

5.2 Request concentration

Here we consider the concentration of requests in a real-time setting, recording the number of machines accessed per query. Because the PYMK system has to wait for the slowest query response before performing ranking, the number of machines queried is an important performance characteristic. By waiting for fewer machines, the expected time needed to wait until the slowest machine has returned data can be significantly decreased.

Baseline performance. In the baseline system, users are sharded by their Facebook user ID modulus 78. Prior to the sharding optimizations in this paper, the distribution of requests on each machine was not instrumented, and because there were significant architectural changes to the service when the sharding optimization was introduced, the baseline system was not upgraded to include instrumentation.

Fortunately, the trailing digits of a user's ID are uncorrelated from the trailing digits of the user IDs of their friends, and thus the question of how many machines are queried during an average aggregation can be computed directly given only the degree distribution of the graph.

Consider a graph sharded across m machines. Let X_i, $i = 1, \ldots, m$ be the Bernoulli random variables indicating whether a given user has a friend on machine i. Let $Y = \sum_{i=1}^{m} X_i$ be the total number of machines that are queried when this user's friends data is aggregated. By the law of total probability,

$$Pr(Y = k) = \sum_{d} Pr(Y = k | \deg(u) = d) \cdot Pr(\deg(u) = d),$$

where $\deg(u)$ is the degree of the user, and $Pr(\deg(u) = d)$ is the empirical Facebook degree distribution. Focusing on the term $Pr(Y = k | \deg(u) = d)$,

$$Pr(Y = k | \deg(u) = d) = Pr\left(\sum_{i=1}^{m} X_i = k | \deg(u) = d\right),$$

where X_i are Bernoulli distributed random variables. To derive the distribution of X_i, notice that, $\forall i$,

$$\begin{aligned} Pr(X_i = 1 | \deg(u) = d) &= \\ &= 1 - Pr(X_i = 0 | \deg(u) = d) \\ &= 1 - (1 - 1/m)^d . \end{aligned}$$

Thus, we see that $X_i \sim \text{Bernoulli}(1-(1-1/m)^d)$, $\forall i$. While these m variables are identically distributed, they are unfortunately not independent.

Since the variables X_i are not independent, we resort to a simple Monte Carlo simulation of the distribution $Pr(Y = k | \deg(u) = d)$ for a uniformly sharded system. Combing this distribution with the empirical degree distribution from PYMK queries, $Pr(\deg(u) = d)$, gives us the theoretical request concentration distribution for the baseline system.

Results. In Figure 10, we compare the distribution of requests as measured for the two algorithms and computed for the baseline system. Note that our measurement of request concentration is not affected by differences in hardware or traffic volumes between the three systems.

The median number of non-local machines queried for the baseline, geographic, and once-propagated shardings were 59, 12, and 9 machines, respectively. Notice that under the old naive system, the most common occurrence was that friend lists had to be aggregated from all 77 non-local machines, while the modes for the new shardings are 6 non-local machines for the geographic sharding and 3 non-local machines for the once-propagated sharding.

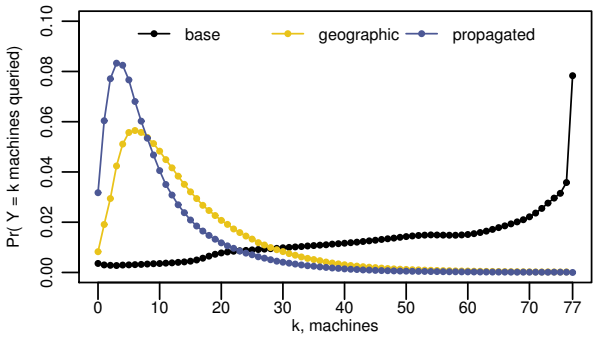

Figure 10: Number of non-local machines queried per request during friend-of-friend calculations in PYMK. The median number of machines queried for the baseline, geographic, and once-propagated shardings were 59, 12, and 9 machines, respectively.

5.3 Query time and network traffic

Here we report on the relative performance of the three systems with regard to query time and network traffic. While the hardware specifications of the three PYMK services were not identical, with the propagated sharding operating with twice as many CPUs per machine, we still report the query times, noting that each query was run in a single thread, and that for the most part, the number of cores per machine did not come into play. The baseline and geographic systems were run on identical hardware, and in any event the bandwidth comparisons are independent of the machine specifications and can thus be taken at face value.

The time-averages of the average machine query times for the baseline, geographic, and once-propagated shardings were 109ms, 68ms, and 55ms, respectively. The time-averages of the maximum machine query times were 122ms, 106ms, and 100ms, respectively (not plotted). Notice that the geographic system, which operated on hardware identical to the baseline system, featured an average query time only 62.3% of the baseline system. The total improvement when comparing to the propagated system was an average query time only 50.5% of the baseline system.

Recall that the two optimized systems being tested are disadvantaged compared to the baseline system because they must perform an additional query redirection, since the web tier does not possess a copy of the shard map and doesn't know which of the 78 machines the user lives on.

Meanwhile, when we turn to network traffic, the time-averages of the average machine traffic for the baseline, geographic, and once-propagated shardings were 35.3 MB/s, 21.8 MB/s, and 13.1 MB/s, respectively. The time-averages of the maximum machine traffic (not plotted) were 103.6 MB/s, 68.0 MB/s, and 51.0 MB/s, respectively. The systems were configured to be equally load-balanced, each handling equal thirds of the total traffic. Thus, the machines in the propagated system saw network traffic levels only 37.1% of the baseline system machines.

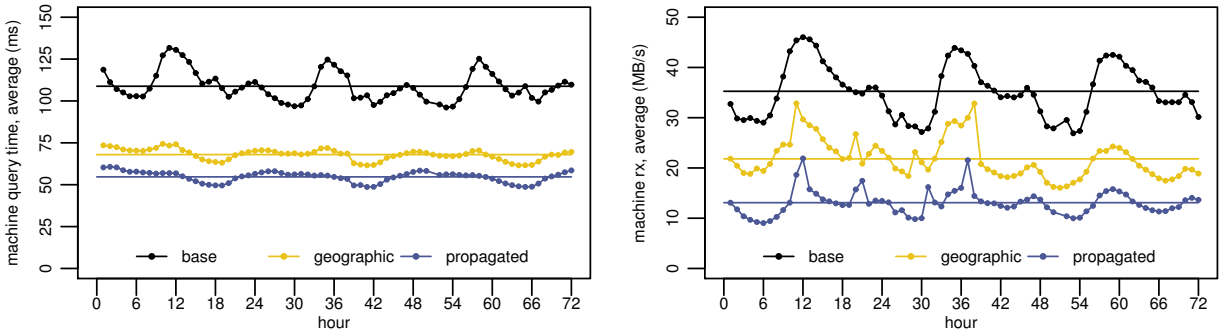

Figure 11: Query time and network traffic for the three different shardings applied to the PYMK service. Because network traffic is instrumented on a machine level, the data also captures daily traffic bursts which correspond to loading data into the service. For this reason, momentary outliers should be considered benign.

6. DISCUSSION

The problem of clustering a graph for community detection is a widely studied active area of research within computer science and physics [13, 14]. In this work, we approach the rather different challenge of graph partitioning. We develop and evaluate a novel algorithm, balanced label propagation, for partitioning a graph while managing these challenges.

We show that by using intelligent partitioning in the context of load-balancing a realtime graph computation service, we are able to dramatically outperform a baseline configuration. While a random initialization of our balanced label propagation algorithm produces an impressive sharding, we show that by using user metadata we can derive a sharding that is greatly superior to a random initialization while still maintaining uniformity in shard sizes. These techniques were applied to a single system in this work, but we believe that they are broadly applicable to any graph computation system that distributes graphs across many machines.

7. ACKNOWLEDGMENTS

We thank Jon Kleinberg for helpful discussions. This work was supported in part by NSF grants IIS-0910664 and IIS-1016099.

8. REFERENCES

[1] L. Backstrom, D. Huttenlocher, J. Kleinberg, X. Lan. Group formation in large social networks: membership, growth, and evolution. In *KDD*, 44–54, 2006.

[2] L. Backstrom, E. Sun, C. Marlow. Find me if you can: improving geographical prediction with social and spatial proximity. In *WWW*, 61–70, 2010.

[3] M.J. Barber, J.W. Clark. Detecting network communities by propagating labels under constraints. *Physical Review E*, 80:026129, 2009.

[4] M. Berkelaar. The lpsolve package. *http://lpsolve.sourceforge.net*, 2011.

[5] S.P. Boyd, L. Vandenberghe. *Convex optimization*. Cambridge University Press, 2004.

[6] U. Feige, R. Krauthgamer. A polylogarithmic approximation of the minimum bisection. In *FOCS*, 105–115, 2000.

[7] S. Fortunato. Community detection in graphs. *Physics Reports*, 486:75–174, 2010.

[8] M.R. Garey, D.S. Johnson, L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical computer science*, 1(3):237–267, 1976.

[9] S. Gregory. Finding overlapping communities in networks by label propagation *New Journal of Physics*, 12:103018, 2010.

[10] G. Karypis, V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Computing* 48(1):96–129, 1998.

[11] J. Leskovec, K. Lang, A. Dasgupta, M. Mahoney. Community structure in large networks: natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6(1):29–123, 2009.

[12] D. Liben-Nowell, J. Novak, R. Kumar, P. Raghavan, A. Tomkins. Geographic routing in social networks. *PNAS*, 102:11623, 2005.

[13] P.J. Mucha, T. Richardson, K. Macon, M.A. Porter, and J.P. Onnela. Community structure in time-dependent, multiscale, and multiplex networks. *Science*, 328:876, 2010.

[14] M.E.J. Newman, M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69:026113, 2004.

[15] H.N. Fowler. *Plato: Phaedrus*, Loeb Classical Library, 1971.

[16] U.N. Raghavan, R. Albert, S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76:036106, 2007.

[17] I. Stanton, G. Kilot. Streaming graph partitioning for large distributed graphs. *KDD*, 1222–1230, 2012.

[18] A. Thusoo, JS Sarma, N Jain, Z Shao, P Chakka, N Zhang, S Antony, H Liu, R Murthy. Hive – a petabyte scale data warehouse using hadoop. In *ICDE*, 996–1005, 2010.

[19] J. Ugander, B. Karrer, L. Backstorm, C. Marlow. The anatomy of the Facebook social graph. Arxiv preprint arXiv:1111.4503, 2011.

[20] X. Zhu, Z. Ghahramani. Learning from labeled and unlabeled data with label propagation. CMU CALD Tech Report CMU-CALD-02-107, 2002.

APPENDIX

This appendix is an addendum to the published version of this paper.

A. ASSEMBLING THE LP

Here we briefly review how to assemble the problem instance from the input data, in particular the task of determining the proper values of a_{ijk} and b_{ijk} for specifying the program. We obtain the following result.

LEMMA 3. *Let $c_{i,j,k}$ be the number of users who gain the kth utility level (in decreasing order) when switching from i to j, and $S_k = \sum_{\ell=1}^{k} c_{i,j,\ell}$ be the partial sum of users gaining more than the kth utility level. For the program specified above, the values of $a_{i,j,k}$ and $b_{i,j,k}$ can be calculated recursively as:*

$$\begin{array}{lcll} a_{i,j,k} & = & u_{ij}(k), & \forall i,j,k \\ b_{i,j,1} & = & 0, & \forall i,j \\ b_{i,j,k} & = & b_{i,j,k-1} + c_{i,j,k-1}a_{i,j,k-1} & \\ & & +S_{k-2}a_{i,j,k-1} - S_{k-1}a_{i,j,k}, & \forall i,j,\forall k > 1. \end{array}$$

PROOF. Recall that each linear segment of f_{ij} corresponds to a unique gain in utility by some non-empty set of users. The $k = 1, \ldots, K$ unique gains in utility per user are then precisely the slopes of the corresponding line segments, so $a_{ijk} = u_{ij}(k)$, for all i, j, k.

Next, observe that for each $f_{ij}(0) = 0$ for all i, j, which means that $b_{ij0} = 0$ for all i, j.

Lastly, the recursive formula is an exercise in straightforward geometry, to compute the correct y-intercept of each line segment. □

By sorting the users by which machine they would like to move to and their co-location gain, it is therefore possible to build the entire LP instance via a single streaming pass of the input data, recording the necessary partial sums.