# A Local Algorithm for Ad Hoc Majority Voting via Charge Fusion

Yitzhak Birk, Liran Liss, Assaf Schuster, and Ran Wolff*

Technion – Israel Institute of Technology, Haifa 32000, Isreal
{birk@ee,liranl@tx,assaf@cs,ranw@cs}.technion.ac.il

**Abstract.** We present a local distributed algorithm for a general Majority Vote problem: different and time-variable voting powers and vote splits, arbitrary and dynamic interconnection topologies and link delays, and any fixed majority threshold. The algorithm combines a novel, efficient anytime spanning forest algorithm, which may also have applications elsewhere, with a "charge fusion" algorithm that roots trees at nodes with excess "charge" (derived from a node's voting power and vote split), and subsequently transfers charges along tree links to oppositely charged roots for fusion. At any instant, every node has an ad hoc belief regarding the outcome. Once all changes have ceased, the correct majority decision is reached by all nodes, within a time that in many cases is independent of the graph size. The algorithm's correctness and salient properties have been proved, and experiments with up to a million nodes provide further validation and actual numbers. To our knowledge, this is the first locality-sensitive solution to the Majority Vote problem for arbitrary, dynamically changing communication graphs.

## 1 Introduction

### 1.1 Background

Emerging large-scale distributed systems, such as the Internet-based peer-to-peer systems, grid systems, ad hoc networks and sensor networks, impose uncompromising scalability requirements on (distributed) algorithms used for performing various functions. Clearly, for an algorithm to be perfectly scalable, i.e., O(1) complexity in problem size, it must be "local" in the sense that a node only exchanges information with nodes in its vicinity. Also, information must not need to flow across the graph. For some problems, there are local algorithms whose execution time is effectively independent of the graph size. Examples include Ring Coloring [1] and Maximal Independent Set [2].

Unfortunately, there are important problems for which there cannot be such perfectly-scalable solutions. Yet, locality is a highly desirable characteristic: locality decouples computation from the system size, thus enhancing scalability; also, handling the effects of input changes or failures of individual nodes locally

* The work of Liran Liss, Assaf Schuster, and Ran Wolff was supported by the Intel Corporation and the Mafat Institute for Research and Development.

R. Guerraoui (Ed.): DISC 2004, LNCS 3274, pp. 275–289, 2004.

cuts down resource usage and prevents hot spots; lastly, a node is usually able to communicate reliably and economically with nearby nodes, whereas communication with distant nodes, let alone global communication, is often costly and prone to failures.

With these motivations in mind, efficient local (or "locality sensitive") algorithms have also been developed for problems that do not lend themselves to solutions whose complexity is completely independent of the problem instance. One example is an efficient Minimum Spanning Tree algorithm [2]. Another example is fault-local mending algorithms [3, 4]. There, a problem is considered fault-locally mendable if the time it takes to mend a batch of transient faults depends only on the number of failed nodes, regardless of the size of the network. However, the time may still be proportional to the size of the network for a large number of faults.

The notion of locality that was proposed in [3, 4] for mending algorithms can be generalized as follows: an algorithm is local if its execution time does not depend directly on the system size, but rather on some other measure of the problem instance. The existence of such a measure for non-trivial instances of a problem suggests (but may not guarantee) the possibility of a solution with unbounded scalability (in graph size) for these instances. This observation encourages the search for local algorithms even for problem classes that are clearly global for some instances. In this paper, we apply this idea to the Majority Vote problem, which is a fundamental primitive in distributed algorithms for many common functions such as leader election, consensus and synchronization.

### 1.2 The Majority Vote Problem

Consider a system comprising an unbounded number of nodes, organized in a communication graph. Each node has a certain (possibly different) voting power on a proposed resolution, and may split its votes arbitrarily between "Yes" and "No". Nodes may change their connectivity (topology changes) at any moment, and both the voting power and the votes themselves may change over time[1]. In this dynamic setting, we want every node to decide whether the fraction of Yes votes is greater than a given threshold. Since the outcome is inherently ad hoc, it makes no sense to require that a node be aware of its having learned the "final" outcome, and we indeed do not impose this requirement. However, we do require eventual convergence in each connected component.

The time to determine the correct majority decision in a distributed vote may depend on the significance of the majority rather than on system size. In certain cases such as a tie, computing the majority would require collecting at least half of the the votes, which would indeed take time proportional to the size of the system. Yet, it appears possible that whenever the majority is evident throughout the graph, computation can be extremely fast by determining the correct majority decision based on local information alone.

[1] Nodes are assumed to trust one another. We do not address Byzantine faults in this paper.

Constantly adapting to the input in a local manner can also lead to efficient anytime algorithms: when the global majority changes slowly, every node can track the majority decision in a timely manner, without spending vast network resources; when a landslide majority decision flips abruptly due to an instant change in the majority of the votes, most of the nodes should be able to reach the new decision extremely fast as discussed above; and, after the algorithm has converged, it should be possible to react to a subsequent vote change that increases the majority with very little, local activity. A less obvious situation occurs when a vote change reduces the majority (but does not alter the outcome), because the change may create a local false perception that the outcome has changed as well. The challenge to the algorithm is to squelch the wave of erroneous perceived outcome fast, limiting both the number of affected nodes and the duration of this effect.

The Majority Vote problem thus has instances that require global communication, instances that appear to lend themselves trivially to efficient, local solutions, and challenging instances that lie in between.

The main contribution of this paper is a local algorithm for the Majority Vote problem. Our algorithm comprises two collaborating components: an efficient anytime spanning forest algorithm and a charge-fusion mechanism. A node's initial charge is derived from its voting power and vote split such that the majority decision is determined by the sign of the net charge in the system. Every node bases its ad-hoc belief of the majority decision on the sign of its charge or that of a charged node in its vicinity. The algorithm roots trees at charged nodes, and subsequently fuses opposite charges using these trees until only charges of one (the majority) sign are left, thus disseminating the correct decision to all nodes.

We provide proof sketches for key properties (for full proofs, which are omitted for brevity, see [5]) as well as simulation results that demonstrate actual performance and scalability. Offering a preview of our results, our experiments show that for a wide range of input instances, the majority decision can be computed "from scratch" in constant time. Even for a tight vote of 52% vs. 48%, each node usually communicates with only tens of nearby nodes, regardless of system size. In [6], similar behavior was demonstrated using an (unrelated) algorithm that was suited only for tree topologies. To our knowledge, the current paper offers, for the first time, a locality-sensitive solution to the Majority Vote problem for arbitrary, dynamically changing topologies.

The remainder of the paper is organized as follows: In Section 2 we provide an overview of our approach. In sections 3 and 4 we present our Spanning Forest (SF) and Majority Vote (MV) algorithms, respectively, along with formal statements of their properties. In section 5, we provide some empirical results to confirm our assumptions and demonstrate the performance of our algorithm. Section 6 describes at some length related work. We conclude the paper in Section 7.

## 2 Overview of Our Approach

Consider a vote on a proposition. The voting takes place at a set of *polls*, which are interconnected by communication links. We propose the following simple scheme for determining the global majority decision. For each unbalanced poll, transfer its excess votes to a nearby poll with an opposite majority, leaving the former one balanced. Every balanced poll bases its current belief regarding the majority decision on some unbalanced poll in its vicinity. We continue this poll consolidation process until all remaining unbalanced polls posses excess votes of the same type, thus determining the global majority decision. We next state the problem formally, and elaborate on our implementation of the foregoing approach, extending it to an arbitrary majority threshold.

Let *G(V,E)* be a graph, and let $\lambda = \lambda_n/\lambda_d$ be a rational threshold between 0 and 1. Every node $i$ is entitled to $V_i$ votes; we denote the number of node $i$'s Yes votes by $Y_i$. For each connected component $X$ in $G$, the desired majority vote decision is Yes if and only if the fraction of Yes votes in $X$ is greater than the threshold: $\frac{\sum_{i\in X} Y_i}{\sum_{i\in X} V_i} > \lambda$ .

A node can change its current vote in any time. Therefore, we need to distinguish between a node's current vote and the votes or "tokens" that we transfer between nodes during the consolidation process. In order to prevent confusion, we introduce the notion of the *("electrical") charge* of a node, and base the majority decision on the sign of the net charge in the system. The following equivalent criterion for determining a Yes majority vote decision allows us to work with integers and only deal with linear operations (addition and subtraction) for an arbitrary majority threshold: $\lambda_d \sum_{i\in X} Y_i - \lambda_n \sum_{i\in X} V_i > 0$ .

A node $i$'s charge, $C_i$, is initially set to $\lambda_d Y_i - \lambda_n V_i$. Subsequent single-vote changes at a node from No to Yes (Yes to No) increase (decrease) its charge by $\lambda_d$. An addition of one vote to the voting power of a node reduces its charge by $\lambda_n$ if the new vote is No, and increases it by $\lambda_d - \lambda_n$ if the vote is Yes. A reduction in a node's voting power has an opposite effect. Charge may also be transferred among nodes, affecting their charges accordingly but leaving the total charge in the system unchanged. Therefore, the desired majority vote decision is Yes if and only if the net charge in the system is non-negative: $\sum_{i\in X} C_i \geq 0$ .

Our Majority Vote algorithm (MV) entails transferring charge among neighboring nodes, so as to "fuse" and thereby eliminate equal amounts of opposite-sign charges. So doing also relays ad hoc majority decision information. Eventually, all remaining charged nodes have an identical sign, which is the correct global majority decision. Therefore, if we could transfer charge such that *nearby* charged nodes with opposite signs canceled one another without introducing a livelock, and subsequently disseminate the resulting majority decision to neutral nodes locally, we would have a *local* algorithm for the Majority Vote problem.

We prevent livelock with the aid of a local spanning forest algorithm (SF) that we will introduce shortly. The interplay between SF and MV is as follows. The roots of SF's trees are set by MV at charged nodes. SF gradually constructs distinct trees over neutral nodes. MV then deterministically routes charges of one

sign over directed edges of the forest constructed by SF towards roots containing opposite charge. The charges are fused, leaving only their combined net charge. Finally, MV unroots nodes that turned neutral, so SF guarantees that all neutral nodes will join trees rooted at remaining charged ones in their vicinity. Each node bases its (perceived global) majority decision on the sign of the charge of its tree's root. Therefore, dissemination of a majority decision to all nodes is inherently built into the algorithm.

We note that although the system is dynamic, we ensure that the total charge in any connected component of the graph always reflects the voting power and votes of its nodes. By so doing, we guarantee that the correct majority decision is eventually reached by every node in any given connected component, within finite time following the cessation of changes.

## 3 Spanning Forest Algorithm (SF)

In this section, we describe SF, an efficient algorithm for maintaining a spanning forest in dynamic graphs, and state its loop-freedom and convergence properties. In the next section, we will adapt this algorithm and utilize it as part of MV.

### 3.1 SF Algorithm Description

**Overview.** Given a (positive) weighted undirected graph and a set of nodes marked as *active* roots, the algorithm gradually builds trees from these nodes. At any instant, edges and nodes can be added or removed, edge weights can change, and nodes can be marked/unmarked as active roots on the fly. However, the graph is always loop-free and partitioned into distinct trees. Some of these trees have active roots, while others are either inactive singletons (the initial state of every node) or rooted at nodes that used to be active. We denote a tree as *active* or *inactive* based on the activity state of its root.

Whenever the system is static, each connected component converges to a forest in which every tree is active (if active roots exist). Loop freedom ensures that any node whose path to its root was cut off, or whose root became inactive, will be able to join an active tree in time proportional to the size of its previous tree. Unlike shortest path routing algorithms that create a single permanent tree that spans the entire graph (for each destination), SF is intended to create multiple trees that are data-dependent, short-lived, and local. Therefore, in order to reduce control traffic, an edge-weight change does not by itself trigger any action. Nevertheless, expanding trees do take into account the most recent edge weight information. So, although we do not always build a shortest path forest, our paths are short.

**The Algorithm.** Algorithm 1 presents SF. In addition to topological changes, the algorithm supports two operations to specify whether a node should be treated as an active root ($Root_i$ and $UnRoot_i$), and one query ($NextHop_i$) that returns the identity of a node's downtree neighbor, or $\perp$ if the node is a root. (We denote by downtree the direction from a node towards its root.)

**Algorithm 1** Spanning Forest (SF).

**Variables for node i**:

- $R_i, T_i, W_i, A_i, P_i$ - Root and tree activity states ({0,1}), path weight and Ack number (positive Int), and a next-hop pointer (a node identifier), respectively.
- $\forall j \in N^i : \lambda_i(T_j), \lambda_i(W_j), \lambda_i(A_j)$ - A neighbor $j$'s tree state, weight and Ack number as known to i.

**Macros:**

$Inactive(i) \equiv (T_i = 0) \vee (P_i \neq \perp \wedge \lambda_i(T_{P_i}) = 0)$

$IsAck(i)$ - Evaluates to *true* iff $i$'s neighbors have all acknowledged $i$'s most recent (highest) Ack number. Nodes that become neighbors are considered to have sent and received all Acks that could have been pending to or from each other. (The details of Ack management are omitted for brevity, but are included in the running code.)

**Events:** /* trigger + event-specific action */

- $Init_i() : R_i = 0; T_i = 0; W_i = \infty; P_i = \perp; A_i = 0; \forall j \in N^i : LinkDown_i(j)$
- $LinkUp_i(j)$: send $Update(T_i, W_i, A_i)$ to $j$
- $LinkDown_i(j) : \lambda_i(T_j) = 0; \lambda_i(W_j) = \infty; \lambda_i(A_j) = \perp$; if $(P_i = j)$ $P_i = \perp$
- $Root_i$ operation: $R_i = 1$
- $UnRoot_i$ operation: $R_i = 0$
- receive $Update(T, W, A)$ from $j$: update $\lambda_i(T_j)$, $\lambda_i(W_j)$, and $\lambda_i(A_j)$
- receive $Ack(A)$ from $j$: record the value of $i$'s Ack num as acknowledged by $j$

After every event also do: /* common actions*/

1. if $(R_i = 1)$ /* set $i$ as an active root */
   (a) $T_i = 1, W_i = 0, P_i = \perp$
2. else /* $R_i = 0$ */
   (a) /* if $i$ is inactive and all uptree nodes have acknowledged, update $i$'s weight according to its next hop */
   if $((T_i = 0) \wedge (IsAck(i) = true))$ $W_i = \begin{cases} \infty, & P_i = \perp \vee \lambda_i(W_{P_i}) = \infty \\ \lambda_i(W_{P_i}) + d(i, P_i), & \text{otherwise} \end{cases}$
   (b) /* improve $i$'s path or join an active tree with the same weight if $i$ is inactive or about to become inactive */
   let $j \in N^i$ s.t. $W(j)$ is minimal, where
   $W(j) = \begin{cases} \lambda_i(W_j) + d(i, j), & \lambda_i(T_j) \neq 0 \\ \infty, & \text{otherwise} \end{cases}$
   if $((W(j) < W_i) \vee (W(j) = W_i \wedge W(j) < \infty \wedge Inactive(i)))$
   $P_i = j, W_i = W(j), T_i = \lambda_i(T_j)$
   (c) /* if $i$ is turning inactive, increment $i$'s Ack */
   if $((T_i \neq 0) \wedge (P_i = \perp \vee \lambda_i(T_{P_i}) = 0))$ $T_i = 0, A_i = A_i + 1$
3. send $Update(T_i, W_i, A_i)$ to all neighbors if anything changed
4. send $Ack(\lambda_i(A_j))$ to each unacknowledged neighbor $j$, with the exception of $P_i$ if $IsAck(i) = false$

The answer to the $NextHop_i$ query is $P_i$'s current value

To its neighbors, a node $i$'s state is represented by its perceived tree's activity state $T_i$, its current path weight $W_i$ to some root, and an acknowledgement number $A_i$. The algorithm converges in a similar manner to Bellman-Ford algorithms [7]: after each event, node $i$ considers changing its next hop pointer ($P_i$) to a neighbor that minimizes the weight of its path to an active root (step 2b).

More formally, to a neighbor $j$ that is believed by $i$ to be active ($\lambda_i(T_j) = 1$) and for which $\lambda_i(W_j) + d(i,j)$ is minimal.

Loops are prevented by ensuring that whenever a portion of a tree is inactivated due to an *UnRoot* operation or a link failure, a node will not point to a (still active) node that is uptree from it [8]. (Edge weight increases can also cause loops. However, we do not face this problem because such increases do not affect a node's current weight in our algorithm.) This is achieved both by limiting a node $i$'s choice of its downtree node (next hop) to neighbors that reduce $i$'s current weight, and by allowing $i$ to increase its current weight only when $i$ and all its uptree nodes are inactive (step 2a).

In order to relay such inactivity information, we use an acknowledgement mechanism as follows: a node $i$ will not acknowledge the fact that the tree state of its downtree neighbor has become inactive (step 4), before $i$ is itself inactivated ($T_i$ is set to 0 and $A_i$ is incremented in step 2c) and receives acknowledgements for its own inactivation from all its neighbors ($IsAck(i)$ becomes *true*). Note that $i$ will acknowledge immediately an inactivation of a neighbor that is not its downtree node. Therefore, if a node $i$ is inactive and has received the last corresponding acknowledgement, all of $i$'s uptree nodes must be inactive and their own neighbors are aware of this fact.

An active root expands and shrinks its tree at the fastest possible speed according to shortest path considerations. However, once a root is marked inactive, a three-phase process takes place to mark all nodes in the corresponding tree as inactive and reset their weight to $\infty$. First, the fact that the tree is inactive ($T_i = 0$) propagates to all the leaves. Next, Acks are aggregated from the leaves and returned to the root. (Note that node weights remain unchanged.) Finally, the root increases its weight to $\infty$. This weight increase propagates towards the leaves, resetting the weight of all nodes in the tree to $\infty$ on its way. It may seem that increasing the weight of the leaves only in the third phase is wasteful. However, this extra phase actually speeds up the process by ensuring that nodes in "shorter" branches do not choose as their next hop nodes in "longer" branches that haven't yet been notified that the tree is being inactivated. (This phase corresponds to the *wait* state in [8].)

### 3.2 Loop Freedom

For facility of exposition, given a node $i$ we define $\widehat{W_i}$ to equal $W_i$ if $T_i = 1$ and $\infty$ otherwise. In [5], we prove the following technical lemma by induction on all network events:

**Lemma 1.** *For every node $i$*

1. *if $IsAck(i) = true$ then, for every $j$ uptree from $i$ or $j = i$ and for every neighbor $m$ of $j$: (a) $\lambda_m(\widehat{W_j}) \geq \widehat{W_i}$, and (b) for every in-transit update message $u$ sent by $j$ with weight $\widehat{W_u}$: $\widehat{W_u} \geq \widehat{W_i}$.*
2. *if $IsAck(i) = false$ then the same claims hold when replacing $\widehat{W_i}$ with $W_i$.*

**Theorem 1.** *There are no cycles in the graph at any instance.*

*Proof.* Let $i$ be a node that closes a cycle at time $t_0$. Therefore, at $t_o^+$ we have $\lambda_i(\widehat{W_{P_i}}) = \lambda_i(W_{P_i}) < W_i = \widehat{W_i}$. According to 1) or 2) of Lemma 1, $\lambda_i(\widehat{W_{P_i}}) \geq \widehat{W_i}$, since $P_i$ is also uptree from $i$. A contradiction. □

### 3.3 Convergence

We assume that the algorithm was converged at time 0, after which a finite number of topological and root changes occurred. Let $t_0$ be the time of the last change.

**Theorem 2.** *After all topological and root changes have stopped, SF converges in finite time.*

*Proof.* Based on the fact that the number of possible weights that a node can have at $t_0$ is finite, we show in [5] that there exists a time $t_1 > t_0$ such that for every $t > t_1$ $IsAck = true$ for all nodes. After this time, every node that joins some active tree will remain in one. Since the graph is finite and loop-free, all nodes will either join some active tree (if there are any) or increase their weight to $\infty$. From this point onward, the algorithm behaves exactly like the standard Bellman-Ford algorithm, in which remaining active roots are simulated by zero weighted edges connected to a single destination node. Therefore, proofs for Bellman-Ford algorithms apply here [9]. □

## 4 Majority Vote Algorithm (MV)

In this section, we first describe the required adaptations to SF for use in our Majority Vote algorithm (MV). Next, we provide a detailed description of MV, discuss its correctness, and state its locality properties.

### 4.1 SF Adaptation

We augment SF as follows:

1. To enable each neutral node to determine its majority decision according to its tree's root, we expand the SF root and tree state binary variables ($R_i$ and $T_i$) to include the value of $-1$ as well. While inactive nodes will still bear the value of $T = 0$, the tree state of an active node $i$ will always equal the sign of its next hop (downtree neighbor) as known to $i$: $T_i = \lambda_i(T_{P_i})$ or the sign of $R_i$ if $i$ itself is an active root.
2. We attach a "Tree ID" variable to each node for symmetry breaking as explained next. It is assigned a value in every active root, and this value is propagated throughout its tree.
3. To enable controlled routing of charge from the root of one tree to that of an opposite-sign tree that collided with it, each node also maintains an *inverse hop*, which designates a weighted path to the other tree's root.

Node $i$ considers a neighbor $j$ as a candidate for its inverse hop in two cases: (a) $i$ and $j$ belong to different trees and have opposite signs ($T_i = -T_j$); (b) $i$ is $j$'s next hop, both nodes have the same sign ($T_i = T_j$), and $j$ has an inverse hop. We further restrict $i$'s candidates to those designating a path towards a root with a higher Tree ID. (Different IDs ensure that only one of the colliding trees will develop inverse hops.) If there are remaining candidates, $i$ selects one that offers a path with minimal weight.

4. To guarantee that paths do not break while routing charges, we prevent an active node from changing its next hop[2]. However, as will be explained shortly, there are cases wherein new active roots should be able to take over nodes of neighboring active trees. Therefore, we extend the *Root* operation to include an *expansion flag*. Setting this flag creates a bounded one-shot expansion wave, by repeatedly allowing any neighboring nodes to join the tree. The wave will die down when it stops improving the shortest path of neighboring nodes or when the bound is reached.

The adaptations do not invalidate the correctness or the convergence of the SF algorithm [5]. The interface of the augmented SF algorithm exposed to MV is summarized in the following table:

| Procedure | Function |
|---|---|
| $Root_i\,(sign, ID, expand)$ | Mark $i$ as an active root |
| $UnRoot_i$ | Unmark $i$ as an active root |
| $TreeSign_i$ | Return $i$'s tree state |
| $TreeID_i$ | Return $i$'s tree ID |
| $NextHop_i$ | Return $i$'s next hop, or $\perp$ if $i$ is a root |
| $InvHop_i$ | Returns $i$'s preferred inverse hop, or $\perp$ if there is none |

### 4.2 MV Algorithm Description

**Overview.** MV is an asynchronous reactive algorithm. It operates by expressing local vote changes as charge, relaying charge sign information among neighboring nodes using SF, and fusing opposite charges to determine the majority decision based on this information. Therefore, events that directly affect the current charge of a node, as well as ones that relay information on neighboring charges (via SF), cause an algorithm action.

Every distinct charge in the system is assigned an ID. The ID need not be unique, but positive and negative charges must have different IDs (e.g., by using the sign of a charge as the least significant bit of its ID). Whenever a node remains charged following an event, it will be marked as an active root (using the SF *Root* operation), with the corresponding sign and a charge ID. If the event was a vote change, we also set the root's expansion flag in order to balance tree sizes among the new tree and its neighbors. This improves overall tree locality, since a vote change has the potential of introducing a new distinct charge (and hence, a new tree) into the system.

[2] We apply similar restrictions to an active node that is marked as a new root [5].

When trees of opposite signs collide, one of them (the one with the lower ID) will develop inverse hops as explained above. Note that inverse hops are not created arbitrarily: they expand along a path leading directly to the root. Without loss of generality, assume that the negative tree develops inverse hops. Once the negative root identifies an inverse hop, it sends all its charge (along with its ID) to its inverse hop neighbor and subsequently unmarks itself as an active root (using the SF *UnRoot* operation). The algorithm will attempt to pass the charge along inverse hops of (still active) neutral nodes that belonged to the negative tree (using the SF *InvHop* query), and then along next hops of nodes that are part of the positive tree (using the SF *NextHop* query).

As long as the charge is in transit, no new roots are created. If it reaches the positive root, fusion takes place. The algorithm will either inactivate the root or update the root's sign and charge ID, according to the residual charge. In case the propagation was interrupted (due to topological changes, vote changes, expanding trees, etc.), the charge will be added to that of its current node, possibly creating a new active root.

**The Algorithm.** Algorithm 2 states MV formally. $C_i(j)$ keeps track of every charge transferred between a node and each of its neighbors. It is used to ensure that charges remain within the connected component in which they were generated. $GenID(charge)$ can be any function that returns a positive integer, as long as different IDs are generated for positive and a negative charges. However, we have found it beneficial to give higher IDs to charges with greater absolute values, as this causes them to "sit in place" as roots. This scheme results in faster fusion since charges with opposite signs and lower absolute values will be routed towards larger charges in parallel. It also discourages fusion of large same-sign charges. This situation could arise when multiple same-sign charges are sent concurrently to a common destination node that held an opposite-sign charge.

After updating a node $i$'s charge information following an event, the algorithm performs two simple steps. In step 1, if $i$ is charged, the algorithm attempts to transfer the charge according to $i$'s tree sign and current next/inverse hop information obtained from SF. In step 2, $i$'s root state is adjusted according to its remaining charge. The output of the algorithm, i.e., the estimated majority decision at every node, is simply the sign of the node's tree state (using SF's *TreeSign* query). For inactive nodes, we arbitrarily return *true*.

### 4.3 Correctness

Assume that all external events (link state changes, bit changes, etc.) stop at some time $t_0$. Because no new charges are introduced to the system and existing charges can only be fused together, the number of distinct charges after $t_0$ becomes constant after some finite time $t_1 > t_0$. By induction on charge IDs, we show in [5] that all remaining charges after $t_1$ must be of the same sign.

**Theorem 3.** *After all external events have ceased, MV stops in finite time with the correct output in every node.*

**Algorithm 2** Majority Vote.

**Variables for node i**:
- $Y_i, V_i, C_i, ID_i$ - "Yes" votes, total votes,charge and charge ID, respectively.
- $\forall j \in N^i : C_i(j)$ - total charge transferred **between** $i$ and a neighbor $j$, from $i$'s perspective.

**Macros:**
$GenID(charge)$ generates a new charge ID
$Charge(V,Y) \equiv \lambda_d \cdot Y - \lambda_n \cdot V$

**Events:** /* trigger + event specific action */
- $Init_i$: $V_i; Y_i; C_i = Charge(V_i, Y_i); ID_i = GenID(C_i); \forall j \in N(i) : C_i(j) = 0$
- $LinkUp_i(j)$: do nothing
- $LinkDown_i(j)$: $C_i = C_i + C_i(j); C_i(j) = 0$
- $ChangeVote_i(V,Y)$: $C_i = C_i + (Charge(V,Y) - Charge(V_i, Y_i))$;
  $V_i = V; Y_i = Y; ID_i = GenID(C_i)$
- Receive $Transfer(C, ID)$ from $j$:
  if $(C_i = 0)$ $ID_i = ID$ else $ID_i = GenID(C_i + C)$;
  $C_i = C_i + C; C_i(j) = C_i(j) - C$

After each of the events above or a change in $SF$'s state do: /* common actions */
1. /* if $i$ is charged, try to transfer the charge */
   if $((C_i \neq 0) \wedge (TreeID_i \geq ID_i))$
   if $(Sign(C_i) = -TreeSign_i)$ $temp = NextHop_i$ else $temp = InvHop_i$;
   if $(temp \neq \perp)$ send $Tansfer(C_i, ID_i)$ to $temp, C_i(j) = C_i(j) + C_i, C_i = 0$
2. /* Mark $i$ as an active root if it remained charged. Otherwise, unmark it */
   if $(C_i = 0)$ $UnRoot_i$ else $Root_i(Sign(C_i), ID_i, f)$ where $f = true$ if invoked by a $ChangeVote_i$ operation

**Output**: $true$ if $TreeSign_i \geq 0$, and $false$ otherwise

*Proof.* Once all charges are identical, termination is guaranteed [5]. Let $X$ be a connected component after the algorithm stopped. Assume that the majority decision for all nodes in $X$ should be $true$, i.e., $\lambda_d \sum_{i \in X} Y_i - \lambda_n \sum_{i \in X} V_i \geq 0$. Hence, $\sum_{i \in X} C_i \geq 0$ [5]. Since all remaining charges have the same sign, it follows that $\forall i \in X : C_i \geq 0$. Therefore, all nodes in $X$ decide $true$, as there are no negative trees in the graph. The situation when the majority decision should be $false$ is shown similarly. □

### 4.4 Locality Properties

The locality of an execution depends on the input instance. In all cases in which the majority is evident throughout the graph, the algorithm takes advantage of this by locally fusing minority and majority charges in parallel. Many input instances follow this pattern, especially when the majority is significant.

The algorithm operates in a way that preserves the charge distribution because: 1) further vote changes create new roots uniformly, and 2) our charge ID scheme discourages fusion of charges of the same sign. Therefore, we conjecture that for many input instances, the size of remaining trees after the algorithm has converged will be determined by the majority percentile, rather than by the graph size. For example, consider a fully connected graph of size $N$ for which

each node has a single vote, a threshold of 1/2, and a tight vote of 48% vs. 52%. After the algorithm converges, the absolute net charge is $4\% \cdot N$. Assuming that the remaining charge is spread uniformly so that every charge unit establishes an active root of its own, the number of nodes in each tree is about $\frac{N}{4\% \cdot N} = 25$, regardless of whether the graph contains a hundred or a million nodes.

From this conjecture it follows that, for these instances, there exists a non-trivial upper bound $R$ on the radius of any tree in the graph. We initially prove that the algorithm is local for single vote changes and when several changes occur far from one another. We then show that the algorithm is local for any fixed number of changes. In the next section, we will use simulations to verify our conjecture empirically, and to demonstrate the local characteristics of our algorithm for arbitrary vote changes.

**Theorem 4.** *Assume that all vote and topological changes have stopped, and MV has converged. Let $R$ be an upper bound on the radius of any tree in the graph. If some node changes a single vote, then the algorithm converges in $O(R)$ time, independent of the overall graph size.*

*Proof (sketch).* An increase in the majority can only result in establishing an additional local tree. A decrease in the majority can introduce a single opposite-sign charge of bounded size, and therefore will be covered by charges from a fixed-sized neighborhood. Note that an opposite-sign charge may still generate an opposite-sign tree. However, the growth rate of this tree is at most half the one-way propagation speed, because other trees have to be inactivated before their nodes can join it [5]. Therefore, an opposite-sign tree cannot expand too far before it is itself inactivated, and its nodes rejoin remaining trees. All these operations take $O(R)$ time. □

**Corollary 1.** *Theorem 4 also holds for multiple vote changes at different nodes, such that the resulting protocol actions do not coincide with one another.*

For an arbitrary number of vote changes, we do not give a bound on convergence time. However, we show that the algorithm remains local when the majority decision does not change, by proving finite convergence time even for infinite graphs.

**Theorem 5.** *Let $G$ be an infinite graph, for which MV has converged. Assume $G$ has infinitely many charged roots (of the same sign) such that there exists an upper bound $R$ on the radius of any tree in the graph. If $m < \infty$ vote changes occur, then the algorithm converges in finite time.*

*Proof (sketch).* We show that the number of opposite-sign charges (with respect to charges before the vote changes) drops to zero in finite time, thereby reducing the problem to a finite region of the graph, which converges in finite time. □

## 5 Empirical Study

We simulated the algorithm's execution on large graphs. For simplicity, we only considered a 50% majority threshold and one vote per node. However, simulations were run for several Yes/No voting ratios, thereby checking the sensitivity

of the results to the proximity of the vote to the decision threshold. Two representative graph topologies were used: a mesh for computing centers and sensor networks, and de Bruijn graphs for structured peer-to-peer systems [10]. For each, graph sizes varied from 256 to one million nodes. Finally, both bulk ("from scratch") voting and ongoing voting were simulated.

In bulk voting, all nodes voted simultaneously at $t = 0$ with the desired Yes/No ratio, and we measured the time until various fractions (90%, 95%, 100%, etc.) of the nodes decided on the correct outcome without subsequently retracting. Multiple experiments were carried out for a <graph type, size, Yes/No ratio> combination, with i.i.d drawings of the votes in the different experiments, and the results were averaged.

We found that the mean time to achieve any convergence percentile under 100% only depends on the Yes/No ratio and is independent of graph size. The algorithm's communication costs follow a similar pattern [5]. This is evidence of the algorithm's local behavior. Figure 1 (a) depicts the results for a convergence percentile of 100%, i.e, the time until the last node reaches the correct outcome. We observe that for de Bruijn graphs, the time to 100% convergence is nearly constant regardless of graph size. For mesh graphs, the time appears proportional to the logarithm of graph size as the Yes/No ratio approaches the threshold. Nevertheless, this time is sub-linear in the graph diameter.

Figure 1 (b) focuses on the convergence percentile, providing the probability distribution of converged nodes over time. Two things are readily evident from the figure: 1) beyond the mean time to convergence, the number of unconverged nodes declines exponentially with time; 2) this distribution is independent of graph size. In fact, the distributions for different graph sizes are barely distinguishable. This strongly suggests that locality and scalability hold for virtually every convergence percentile except 100%.

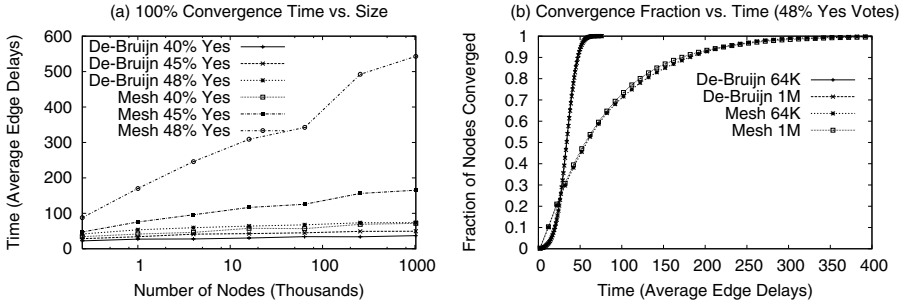


**Fig. 1.** Bulk voting convergence and locality.

In ongoing voting, a given fraction (0.1%) of the nodes changes its vote once every mean edge delay, while the overall Yes/No ratio remains constant. We view this operation mode as the closest to real-life. In this setting we wish to evaluate the time it takes for the effects of dispersed vote changes to subside and to validate that our algorithm does not converge to some pathological situation

(e.g., all remaining charge converges to a single node, whose tree spans the entire graph). Therefore, we ran the system for some time reaching a steady state. Subsequently, we stopped all changes and measured the convergence time and the number of nodes in each tree upon convergence.

As expected, convergence time in on-going voting only depended on the Yes/No ratio. Furthermore, it was substantially shorter compared to bulk voting. For example, the time for 95% convergence in on-going voting took half that of bulk voting for 45% Yes votes, and a tenth for 40%. In addition, tree sizes where tightly distributed about their mean, which was less than twice the ideal size according to a given Yes/No ratio. These experiments thus confirm our conjecture that tree sizes are small, and demonstrate that locality is maintained in on-going voting as well.

## 6 Related Work

Our work bears some resemblance to Directed Diffusion [11], a technique to collect aggregate information in sensor networks. As in their work, our routing is data-centric and based on local decisions. However, our induced routing tables are relatively short-lived, and do not require refreshments or enforcements. The SF algorithm we present, builds upon previous research in distributed Bellman-Ford routing algorithms which avoid loops such as [8] and [9].

Several alternative approaches can be used to conduct majority voting such as sampling, pseudo-static computation, and flooding. With sampling, the idea is to collect data from a small number of nodes selected with uniform probability from the system, and compute the majority based on that sample. One such algorithm is the gossip based work of Kempe et al. [12]. Unfortunately, sampling cannot guarantee correctness and is sensitive to biased input distributions. Moreover, gossip based algorithms make assumptions on the mixing properties of the graph which do not hold for any graph. Pseudo-static computation suggests to perform a straightforward algorithm that would have computed the correct result had the system been static, and then bound the error due to possible changes. Such is the work by Bawa et. al. [13]. In flooding, input changes of each node are flooded over the whole graph, so every node can compute the majority decision directly. While flooding guarantees convergence, its communication costs are immense and it requires memory proportional to the system size in each node.

One related problem that has been addressed by local algorithms is the problem of local mending or persistent bit. In this problem all nodes have a state bit that is initially the same. A fault changes a minority of the bits and the task of the algorithm is to restore the bits to their initial value. Local solutions for this problem were given in [3, 4]. However, these solutions assume a static topology and synchronous communication.

Finally, [6] also conducts majority votes in dynamic settings. However, their algorithm assumes the underlying topology is a spanning tree. Although this algorithm can be layered on top of another distributed algorithm that provides a tree abstraction, a tree overlay does not make use of all available links as we

do, and its costs must be taken into account. Even when assuming that once a tree is constructed its links do not break, simulations have shown that while [6] is faster in cases of a large majority, our algorithm is much faster as the majority is closer to the threshold.

## 7 Conclusions

We presented a local Majority Vote algorithm intended for dynamic, large-scale asynchronous systems. It uses an efficient, anytime spanning forest algorithm as a subroutine, which may also have other applications. The Majority Vote algorithm closely tracks the ad hoc solution, and rapidly converges to the correct solution upon cessation of changes. Detailed analysis revealed that if the occurrences of voting changes are uniformly spread across the system, the performance of the algorithm depends only on the number of changed votes and the current majority size, rather than the system size. A thorough empirical study demonstrated the excellent scalability of the algorithm for up to millions of nodes – the kind of scalability that is required by contemporary distributed systems.

## References

1. Linial, N.: Locality in distributed graph algorithms. SIAM J. Comp. **21** (1992) 193–201
2. Awerbuch, B., Bar-Noy, A., Linial, N., Peleg, D.: Compact distributed data structures for adaptive network routing. Proc. 21st ACM STOC (1989)
3. Kutten, S., Peleg, D.: Fault-local distributed mending. Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (1995)
4. Kutten, S., Patt-Shamir, B.: Time-adaptive self-stabilization. Proc. of the 16th Annual ACM Symp. on Principles of Distributed Computing (1997) 149–158
5. Birk, Y., Liss, L., Schuster, A., Wolff, R.: A local algorithm for ad hoc majority voting via charge fusion. Technical Report EE1445 (CCIT497), Technion (2004)
6. Wolff, R., Schuster, A.: Association rule mining in peer-to-peer systems. In Proc. of the IEEE Conference on Data Mining (ICDM) (2003)
7. Ford, L., Fulkerson, D. In: Flows in Networks. Princton University Press (1962)
8. Jaffe, J., Moss, F.: A responsive routing algorithm for computer networks. IEEE Transactions on Communications (1982) 1758–1762
9. Garcia-Luna-Aceves, J.: A distributed, loop-free, shortest-path routing algorithm. Proceedings of IEEE INFOCOM (1988) 1125–1137
10. Kaashoek, F., Karger, D.: Koorde: A simple degree-optimal distributed hash table. In Proc. of the Second Intl. Workshop on Peer-to-Peer Systems (IPTPS) (2003)
11. C. Intanagonwiwat, R.G., Estrin, D.: Directed diffusion: A scalable and robust communication paradigm for sensor networks. In Proceedings of the Sixth Annual Intl. Conf. on Mobile Computing and Networking (2000)
12. Kempe, D., Dobra, A., Gehrke, J.: Computing aggregate information using gossip. Proceedings of Foundations of Computer Science (FOCS) (2003)
13. Bawa, M., Garcia-Molina, H., Gionis, A., Motwani, R.: Estimating aggregates on a peer-to-peer network. Technical report, Stanford University, DB group (2003)