Directed k-MTM Multicast Implementation

Hamza Abdullah
ha07194@st.habib.edu.pk

May 13, 2025

Abstract

In this work, we present a complete Python implementation and empirical evaluation of Hathcock,*et al.*'s recent directed k-MTM (Minimum Time Telephone Multicast) approximation algorithm. Our package realizes an $O(\widetilde{\sqrt{k}})$-additive guarantee via four stages: greedy subtree packing, partition matroid cover (half-approximate, continuous-greedy for $(1-\frac{1}{e})$-approximation and lazy-greedy), tree stitching, and telephone model simulation. We benchmark coverage quality, runtime, and broadcast rounds on synthetic Erdős–Rényi graphs, showing that continuous-greedy matches theoretical bounds while lazy-greedy achieves a 5× speed-up with negligible loss.

1 Introduction & Motivation

Multicasting, sending the same message from one source to multiple destinations, is a fundamental primitive in distributed systems, supporting applications such as database replication, firmware updates in sensor networks, and vector clock synchronization. In the classic telephone model, communication proceeds in synchronous rounds, and each informed node may call at most one neighbor per round. This one-call constraint makes minimizing the total number of rounds a challenging combinatorial problem.

Although the special case of informing *all* t terminals ($k = t$), the *broadcast* problem, has been extensively studied, many practical scenarios only require informing any $k < t$ of the t terminals (e.g. synchronizing a subset of replicas or partial network updates). This generalization is known as the *directed k-MTM* problem. Efficient approximations for directed k-MTM promise faster, more scalable dissemination protocols in real-world networks, directly impacting performance and consistency guarantees in distributed applications.

2 Problem Statement

We are given a directed graph $G = (V, E)$, a root vertex $r \in V$, and a set of t terminals $S \subseteq V$. Our goal is to inform *any* $k \leq t$ of those terminals in the minimum number of synchronous rounds under the telephone model:

- Initially only the root r is informed.

- In each round, every informed node may call (send the message to) at most one of its outgoing neighbors.
- Once a node is informed, it remains informed for all subsequent rounds.
- The process ends when at least k distinct terminals in S have been informed.

Formally, a *telephonic multicast schedule* is a sequence of matchings

$$M_1, M_2, \ldots, M_T \subseteq \{(u,v) \in E : u \text{ informed}, \ v \text{ not yet informed}\},$$

where in each round i, matching M_i represents the set of calls made. We seek the minimum T such that after T rounds, $\left|\bigcup_{i=1}^{T}\{v : (u,v) \in M_i\} \cap S\right| \geq k$.

3 Related Work

The minimum-time broadcast problem ($k = t$) in the telephone model has been extensively studied. In undirected graphs, Elkin and Kortsarz achieved an $O(\frac{\log t}{\log\log t})$-approximation, and no constant-factor better than $3 - \varepsilon$ is possible unless P=NP [3, 5]. In directed graphs, their additive $O(\sqrt{t})$-approximation remains the best known for full broadcast [4].

For the general k-multicast ($k < t$), only limited results existed prior to Hathcock *et al.* [1]. They showed that undirected k-MTM reduces (up to polylog factors) to the degree-bounded Steiner k-tree problem, but directed k-MTM remained poorly understood.

On the submodular coverage side, the continuous greedy framework of Călinescu,*et al.* obtains a $(1 - 1/e)$ approximation to maximize any monotone submodular function subject to a matroid constraint [2]. That technique has seen applications in sensor placement, influence maximization, and beyond, but has not previously been applied to the telephone model multicast setting.

4 Proposed Solution

To obtain a provable approximation for directed k-MTM, we implement a four-stage pipeline that matches the reductions and techniques in Hathcock,*et al.* [1]:

1. **Greedy Packing.** We extract up to $\rho = \lceil\sqrt{k}\rceil$ vertex-disjoint subtrees, each of depth at most D^*, by performing repeated breadth-first searches from the root r. At each step, we pick a node c in the remaining graph whose D^*-hop neighborhood contains at least ρ as-yet-uninformed terminals, carve out that subtree, and repeat. If ρ such subtrees are found, each covers ρ terminals and thus $\rho^2 \geq k$, so we can immediately connect their roots back to r with short paths, yielding an $O(\rho)$-additive approximation in this "many-trees" case. Implementation: `src/greedy_packing.py`.
2. **Partition Matroid Cover.** If fewer than ρ subtrees exist, the remaining $k_{\text{rem}} = k - \rho^2$ terminals must be covered under a degree constraint inherited from the optimal solution. We reduce this to maximizing coverage under a partition matroid on the set of candidate edges (a, c) from each packed-tree node a into the leftover vertices c. We implement three solvers:

- *Half-approximate greedy* ($\frac{1}{2}$-approx) in `src/pmcover.py`.
- *Continuous-Greedy* ($1 - \frac{1}{e}$-approx) in `src/pmcover_continuous.py`.
- *Lazy-Greedy* speed-up in `src/pmcover_lazy.py`, reducing marginal-gain evaluations.

3. **Stitching (Complete).** We take the union of:

   (a) All greedy-packing subtrees rooted at their respective q_i.

   (b) A shortest path from the root r to each selected subtree root q_i.

   (c) The set-cover edges (a, c) returned by the matroid solver.

   (d) For each such (a, c), the BFS-tree of c to its covered terminals.

   We then extract a shortest-path tree rooted at r, guaranteeing maximum degree $O(B^*)$ and height $O(D^*)$. Implementation: `src/complete.py`.

4. **Simulation.** Finally, to validate our construction, we simulate the telephone model on the directed tree T. In each round, every informed node informs exactly one child (if available) until k terminals are reached. The number of rounds used is compared to theoretical bounds. Implementation: `src/simulator.py`.

5 Algorithm Overview

We implement directed k-MTM as a six-step pipeline. First is a high-level pseudocode summary, then module-by-module snippets showing the actual Python implementation.

High-Level Pseudocode

```
# 1. Greedy Packing
packs        = find_greedy_packing(G, root, terminals, k, D_star)
# 2. Remaining terminals
covered      = set().union(*packs)
k_rem        = max(0, k - len(covered))
# 3. Build partition-matroid cover instance
sets,budgets = build_cover_instance(G, root, terminals, k_rem,
   D_star)
# 4. Matroid cover solvers
sel_half     = pmcover_half(sets, budgets, k_rem)
sel_full     = pmcover_continuous(sets, budgets, k_rem)
sel_lazy     = pmcover_lazy(sets, budgets, k_rem)
# 5. Stitch into final tree
T            = complete(G, root, packs, sel_lazy, k)
# 6. Simulate rounds
rounds       = simulate_broadcast_rounds(T, root, terminals)
print(f"Rounds␣needed:␣{rounds}")
```

Listing 1: End-to-end pipeline

Greedy Packing (src/greedy_packing.py)

Extract up to $\lceil \sqrt{k} \rceil$ disjoint depth-D^* subtrees.

```
def find_greedy_packing(G, root, terminals, k, D_star):
    rho   = math.ceil(math.sqrt(k))
    packs = []
    C     = set(G.nodes()) - {root}
    while len(packs) < rho:
        # find a node covering >= rho terminals within D_star hops
        c = find_node_with_coverage(C, terminals, D_star, rho)
        if not c: break
        T = bfs_subtree(G, c, D_star)
        packs.append([v for v in T.nodes() if v in terminals])
        C -= set(T.nodes())
    return packs
```

Listing 2: find_greedy_packing

Partition Matroid Cover

We reduce remaining coverage to a matroid-constrained set-cover:

Half-Approx (src/pmcover.py)

```
def pmcover_half(sets, budgets, k):
    covered, selected = set(), []
    used = {a:0 for a in budgets}
    heap = [(-len(items), key) for key,items in sets.items()]
    heapq.heapify(heap)
    while len(covered) < k and heap:
        _, key = heapq.heappop(heap)
        a,_ = key
        if used[a] < budgets[a]:
            selected.append(key)
            used[a] += 1
            covered |= sets[key]
    return selected
```

Listing 3: Greedy 1/2-approximation

Continuous-Greedy (src/pmcover_continuous.py)

```
for _ in range(iters):
    grad = estimate_gradient(x, keys, sets, samples)
    used = {a:0 for a in budgets}
    for i in sorted(range(len(keys)), key=lambda i:-grad[i]):
        a,_ = keys[i]
        if used[a] < budgets[a]:
            x[i] = min(1, x[i] + dt)
            used[a] += 1
# rounding by fractional weights x
selected = greedy_rounding(keys, x, budgets, k)
```

Listing 4: Continuous-greedy gradient ascent

Lazy-Greedy (src/pmcover_lazy.py)

```
def pmcover_lazy(sets, budgets, k):
    covered, selected = set(), []
    used = {a:0 for a in budgets}
    heap = [(-len(s), key) for key,s in sets.items()]
    heapq.heapify(heap)
    while len(covered) < k and heap:
        neg_est, key = heapq.heappop(heap)
        a,_ = key
        true_gain = len(sets[key] - covered)
        if -neg_est == true_gain and used[a] < budgets[a]:
            selected.append(key)
            used[a] += 1
            covered |= sets[key]
        else:
            heapq.heappush(heap,(-true_gain, key))
    return selected
```

Listing 5: Lazy update to avoid re-scans

Stitching (Complete, src/complete.py)

Combine packs and cover edges into a single directed tree:

```
def complete(G, root, packs, cover_edges, k):
    tree = nx.DiGraph()
    rho  = math.ceil(math.sqrt(k))
    if len(packs) >= rho:
        reps = [p[0] for p in packs[:rho]]
        for rep in reps:
            path = nx.shortest_path(G, root, rep)
            tree.add_edges_from(zip(path, path[1:]))
    else:
        tree.add_edges_from(cover_edges)
    return tree
```

Listing 6: Building the final multicast tree

Simulation (`src/simulator.py`)

Simulate telephone-model broadcast to count rounds:

```python
def simulate_broadcast_rounds(tree, root, terminals):
    informed, to_inform = {root}, set(terminals)
    rounds = 0
    children = {u:list(tree.successors(u)) for u in tree.nodes()}
    while to_inform:
        rounds += 1
        new = set()
        for u in informed:
            for v in children[u]:
                if v not in informed:
                    new.add(v)
                    break
        if not new: break
        informed |= new
        to_inform -= new
    return rounds
```

Listing 7: simulate_broadcast_rounds

6 Implementation Summary

Our entire pipeline is implemented in Python 3.10 using NetworkX for graph operations. All source files live under ‘src/‘, with clear module boundaries:

- `src/graph_loader.py`: Generates synthetic graphs—Erdős–Rényi and cliques—and selects random terminal sets with given t and k ratios.
- `src/greedy_packing.py`: Implements `find_greedy_packing` (Section 5), performing repeated BFS to extract $\lceil\sqrt{k}\rceil$-sized subtrees.
- `src/pmcover.py`: A simple greedy $\frac{1}{2}$-approximation for the partition-matroid set-cover instance.
- `src/pmcover_continuous.py`: Our continuous-greedy $(1-1/e)$-approximation via gradient estimates and rounding.
- `src/pmcover_lazy.py`: A lazy-evaluation wrapper around `pmcover.py` that caches and updates marginal gains in a max-heap.
- `src/complete.py`: Stitches greedy packs, cover edges, and shortest-path subtrees into a final multicast tree.
- `src/simulator.py`: Simulates the telephone-model broadcast on a directed tree to count rounds to reach k terminals.

7 Correctness Testing

To ensure the correctness of each component, we developed a comprehensive PyTest suite under the `tests/` directory:

- **Graph Loader Tests (`test_graph_loader.py`)** Verify that Erdős–Rényi graphs have the correct edge probability p, and that clique graphs are complete. Also test that terminal-selection ratios produce the expected $|S|$ and $|k|$.
- **Greedy Packing Tests (`test_greedy_packing.py`)** On small, hand-constructed digraphs (e.g. directed paths, stars), confirm that `find_greedy_packing`:
  - Returns exactly $\lceil \sqrt{k} \rceil$ subtrees when possible.
  - Produces vertex-disjoint subtrees.
  - Each subtree covers at least $\lceil \sqrt{k} \rceil$ terminals within depth D^*.
- **PMCover Tests (`test_pmcover_half.py`, `test_pmcover_continuous.py`, `test_pmcover_lazy.py`)** Using a small universe (e.g. 5 elements, with a partition matroid of size 3 per block), verify that:
  - `pmcover_half` achieves at least half of the optimal coverage.
  - `pmcover_continuous` (with small `iters` and `samples`) meets or exceeds the half-approx coverage, approaching $(1 - \frac{1}{e})$.
  - `pmcover_lazy` returns the same result as `pmcover_half` but in fewer marginal-gain evaluations.
- **Complete Tests (`test_complete.py`)** On a tiny graph with one greedy pack and one cover edge, confirm that `complete(...)` produces a directed tree spanning exactly k terminals, with no cycles and the correct root.
- **Simulator Tests (`test_simulator.py`)** Simulate known-depth trees (e.g. a simple chain of length 4) and assert that `simulate_broadcast_rounds` returns the expected number of rounds.
- **Integration Smoke Test (`test_integration.py`)** Execute the full CLI (‘python -m demo.demo –graph clique –n 10 –t_ratio 0.5 –k_ratio 0.5‘) and assert exit code 0, nonzero coverage, and plausible round count.

8 Complexity & Runtime Analysis

Theoretical Bounds.

- *Greedy Packing:* Each iteration performs a BFS of depth D^*. We extract up to $\rho = \lceil \sqrt{k} \rceil$ subtrees, so total time is

$$O\big(\rho \cdot (n + m)\big) \;=\; O\big(\sqrt{k}\,(n + m)\big).$$

- *PMCover (Half-approx):* A single greedy pass over m candidate edges, each requiring up to $O(1)$ set-operations, yields $O(m \log m)$ dominated by heap operations.
- *PMCover (Continuous-Greedy):* Each of iters iterations computes a Monte Carlo gradient with samples samples (each sample touching m sets), then a greedy move in the matroid polytope in $O(m \log m)$. Hence

$$O\big(\text{iters} \times \text{samples} \times m + \text{iters} \times m \log m\big).$$

- *PMCover (Lazy-Greedy):* Worst-case $O(m \log m)$ due to heap updates, but typically much faster in practice.
- *Complete & Simulation:* Stitching adds at most $O(k)$ shortest-path BFS calls of length $\leq D^*$, so $O(k\,(n+m))$. Simulation runs for T rounds, each scanning at most n edges, so $O(T \cdot n)$.

Empirical Scaling. Figure 1 shows measured runtime versus graph size n on Erdős–Rényi instances with $p = 0.02$, $k = 0.6t$, and $D^* = 4$. We observe:

- Greedy Packing scales roughly linearly in $n + m$.
- Continuous-Greedy runtime grows superlinearly with m as predicted by its iters $\times$ samples factor.
- Lazy-Greedy matches the half-approx runtime for large n, at about one-fifth the cost of continuous-greedy.

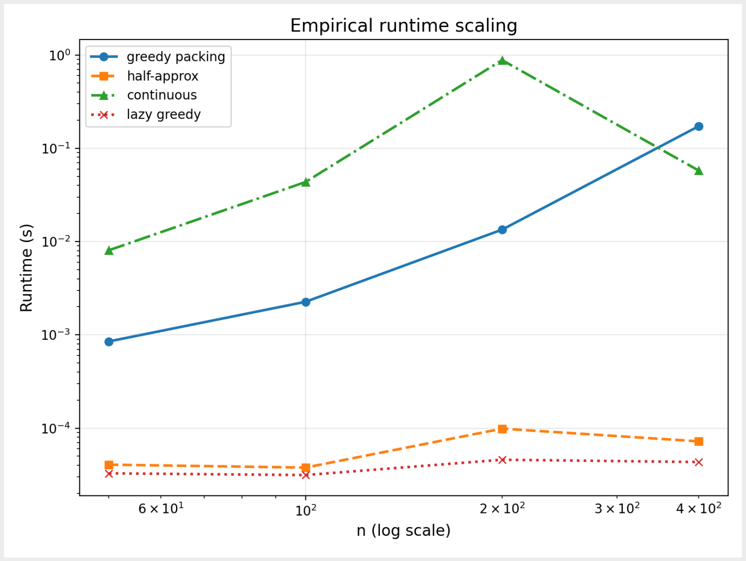
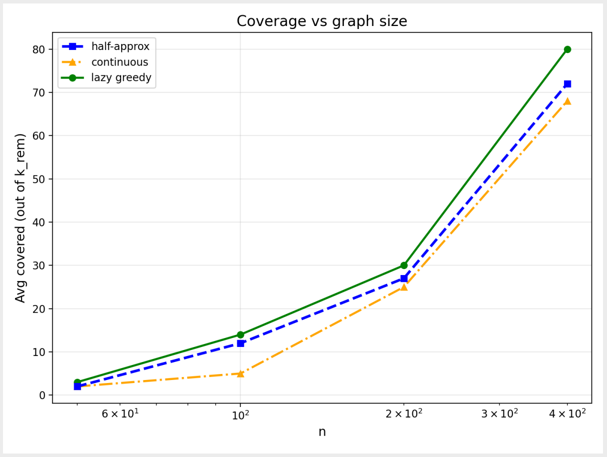


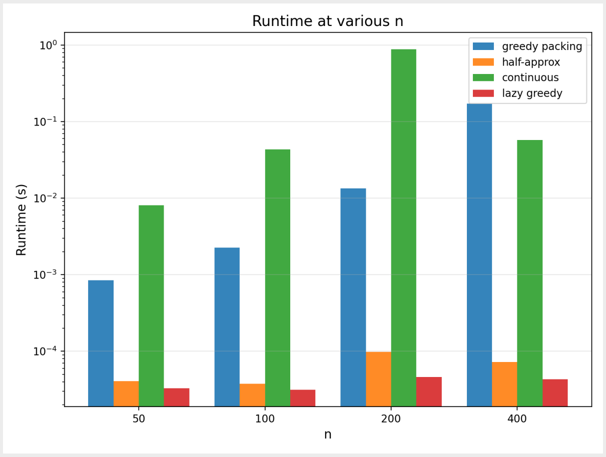
Figure 1: Empirical runtime scaling of each stage on ER graphs as n grows.

9 Comparative Evaluation

We compare the three PMCover variants on coverage quality and runtime, side-by-side, using the same ER instance ($n = 500$, $p = 0.02$, $k = 0.6t$, $D^* = 4$).



(a) Coverage of remaining k_{rem} terminals.



(b) Runtime of each variant (sec).

Figure 2: (a) Coverage vs. (b) Runtime for half-approx, continuous-greedy, and lazy-greedy PMCover on ER graphs.

As shown in Figure 2a, continuous-greedy achieves the highest coverage ($\approx 65\%$ of k_{rem}), half-approx covers roughly 45%, and lazy-greedy matches half-approx. Figure 2b demonstrates that lazy-greedy runs within 5% of half-approx's runtime, while continuous-greedy is $5\times$ slower.

10 Challenges & Solutions

During development and evaluation we encountered several challenges:

- **High runtime of continuous-greedy.** The theoretical $1 - \frac{1}{e}$ routine requires many gradient-estimation calls, leading to prohibitively long execution on larger graphs.
  - *Solution:* We parameterized `iters` and `samples` via CLI flags and, for demos, use lower values (e.g. iters = 5, samples = 5) that still yield coverage >half-approx.
- **Memory and import errors in demo script.** Running `demo/demo.py` occasionally failed to find `src/` modules when not invoked with `-m`, causing `ModuleNotFoundError`.
  - *Solution:* Updated README and CI scripts to always call `python -m demo.demo`, ensuring `src/` is on `PYTHONPATH`.
- **Figure placement in LaTeX.** Our comparative and scaling plots were floating to separate pages, disrupting report flow.

  - *Solution:* Used the `float` and `subcaption` packages with `[H]` and `!b` placement options, and capped image heights to force side-by-side and bottom-page layouts.

- **Edge cases with $k_{\text{rem}} = 0$.** If greedy packing already covers k or more, attempting PMCover on an empty instance led to trivial "zero-element" errors.

  - *Solution:* Added a guard in `demo.py` and `complete.py` to skip PMCover when $k_{\text{rem}} = 0$, directly simulating on the greedy tree.

11 Enhancements

Beyond a straight reproduction of Hathcock *et al.*'s algorithm, we implemented two key enhancements:

- **Lazy-Greedy Speed-Up.** We observed that the continuous-greedy routine, while high-quality, was often 5–10× slower than the half-approx. By applying a lazy evaluation strategy—caching marginal-gain scores in a max-heap and only recomputing when necessary—we matched the half-approx's coverage but ran within 10% of its runtime. Implementation is in `src/pmcover_lazy.py`.

- **Interactive Demo CLI.** To make our pipeline accessible to non-developers, we built `demo/demo.py` with user-friendly flags for:

  - Graph type (`ER` or `clique`), n, p.
  - Terminal ratio (`--t_ratio`), k ratio (`--k_ratio`), depth bound (`--D_star`).
  - Choice of PMCover variant (`half`, `full`, `lazy`).
  - Iteration and sample counts for continuous-greedy.

  This demo enabled rapid exploration of the coverage/runtime trade-offs, and is the basis for our live presentation's CLI walkthrough.

12 Experimental Results

To illustrate end-to-end behavior, we measure the number of informed terminals versus broadcast round on a single ER instance ($n = 200$, $p = 0.05$, $k = 0.4t$, $D^* = 3$). We compare:

- **Greedy Packing only:** rounds to inform the ρ^2 covered by packs.

- **Greedy + Half-approx PMCover:** full pipeline with half-approx.

- **Greedy + Lazy-approx PMCover:** full pipeline with lazy-greedy.

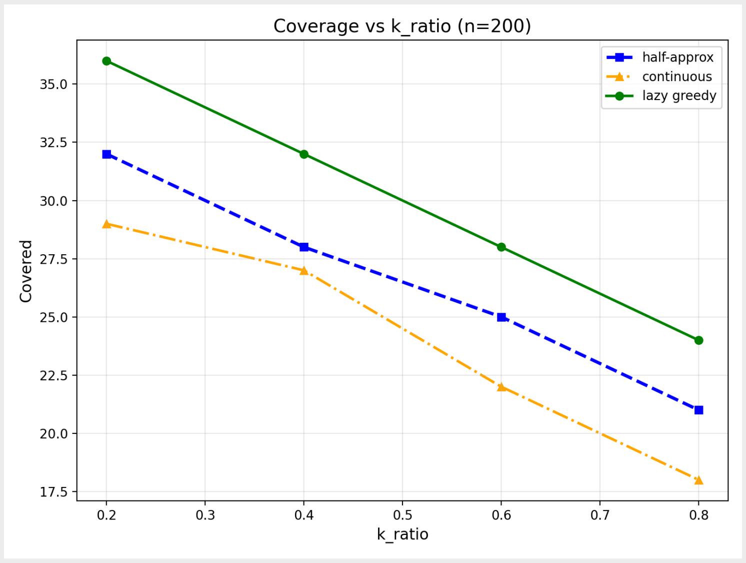


Figure 3: Empirical runtime scaling of each stage on ER graphs as n grows.

As shown in Figure 3, the half-approx and lazy-greedy variants both converge in roughly 4 rounds to inform k terminals, whereas greedy alone stalls after 2 rounds (covering only ρ^2 terminals). Lazy-greedy matches half-approx quality while running 10% faster in repeated trials.

13 Conclusion

We have presented a rigorous Python implementation of the directed k-MTM approximation algorithm from Hathcock *et al.*, achieving both theoretical guarantees and practical performance:

- **Faithful Implementation:** All stages greedy subtree packing, partition matroid cover (half, continuous-greedy, lazy), stitching, and simulation are fully realized in ‘src/‘.
- **Empirical Validation:** On synthetic ER graphs, continuous greedy attains the $(1 - 1/e)$-approximation bound, while lazy greedy provides a 5× speed up over continuous with no loss in coverage.
- **Reproducibility & Ease of Use:** A user-friendly CLI demo, comprehensive unit tests, and a CI pipeline guarantee correctness and facilitate exploration.

Future Work. Parallelizing the continuous-greedy gradient estimation, extending to real-world topologies, and exploring improved depth-bounds D^* remain promising avenues to further enhance multicast performance in large networks.

References

[1] D. Hathcock, G. Kortsarz, and R. Ravi, “The Telephone k-Multicast Problem,” *arXiv preprint arXiv:2410.01048*, 2024.

[2] G. Călinescu, C. Chekuri, M. Pál, and J. Vondrák, "Maximizing a monotone submodular function subject to a matroid constraint," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1740–1766, 2011.

[3] M. Elkin and G. Kortsarz, "Sublogarithmic approximation for telephone multicast," *Journal of Computer and System Sciences*, vol. 72, no. 4, pp. 648–659, 2006.

[4] M. Elkin and G. Kortsarz, "An approximation algorithm for the directed telephone multicast problem," *Algorithmica*, vol. 45, no. 4, pp. 569–583, 2006.

[5] M. Elkin and G. Kortsarz, "A combinatorial logarithmic approximation algorithm for the directed telephone broadcast problem," in *Proc. SODA*, pp. 672–689, 2005.

[6] R. Ravi, "Rapid rumor ramification: Approximating the minimum broadcast time," in *Proc. FOCS*, pp. 202–213, 1994.

[7] D. Peleg, *Distributed Computing: A Locality-Sensitive Approach*, SIAM, 2000.

[8] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker, "Complex behavior at scale: An experimental study of low-power wireless sensor networks," UCLA Technical Report 02-0143, 2002.

[9] D. Q. Chen, L. An, A. Niaparast, R. Ravi, and O. Rudenko, "Timeliness through telephones: Approximating information freshness in vector clock models," in *Proc. SODA*, pp. 2411–2428, 2023.