Checkpoint 3: Progress Report

Directed k-MTM Implementation

Hamza Abdullah - ha07194

April 20th, 2025

Abstract

We have implemented and tested the directed k-MTM algorithm of Hathcock et al., covering greedy packing, matroid-cover, tree-stitching, and a telephone-model simulator. Our unit tests validate correctness on toy graphs and edge cases, and empirical benchmarks on directed ER graphs (n up to 5000) illustrate the expected $\sqrt{k}$-additive and runtime scaling behavior. Key enhancements—such as a BFS fallback and partial matroid-cover comparison— yielded practical speedups. This report documents our progress, challenges resolved, and lays out next steps toward a full final evaluation and presentation.

1 Implementation Summary

- **Completed modules:**
  - `greedy_packing.py`: extracts disjoint $\lceil\sqrt{k}\rceil$-sized subtrees (unit-tested).
  - `pmcover.py`: 1/2-approximation for matroid-constrained set coverage (unit-tested).
  - `complete.py`: stitches packs and cover edges into a final multicast tree (unit-tested).
  - `simulator.py`: simulates telephone-model rounds on a broadcast tree (unit-tested).
  - `graph_loader.py`: utilities to generate directed ER and clique graphs.
- **Omitted / deferred:**
  - Full $(1-\frac{1}{e})$ submodular maximization routine (deferred due to performance concerns).
  - Undirected k-MTM branch (scope limited to the directed case for this checkpoint).
- **Repository layout:**

```
checkpoint3/
|-- src/ % All .py modules
|-- tests/ % PyTest suites for each module
`-- experiments/ % Integration & benchmarking scripts
```

2 Correctness Testing

- **Unit tests:**
  - `test_greedy_packing.py`: line graph, star graph, and path-length edge cases.
  - `test_pmcover.py`: simple matroid instances verifying budget constraints and coverage.

- `test_simulator.py`: broadcast rounds on star and depth-2 trees with known optimal schedules.
- `test_complete.py`: "many-trees" and "few-trees" scenarios on small digraphs.

- **Edge-case validation:**
  - Trivial cases ($k = 1,\ k = t$), no available packs, and disconnected terminal sets.
  - Verified that packing returns $\emptyset$ when $\sqrt{k}$-good subtree doesn't exist.
  - Simulator returns 0 rounds for an already-informed terminal.
- **Baseline comparison:**
  - Matched simulator output against a naïve greedy-matching broadcast on small ER digraphs.
  - Confirmed that our static-tree schedules never take more rounds than the matching heuristic.

3 Complexity & Runtime Analysis

- **Theoretical Bounds:**
  - *Greedy Packing:* Each BFS up to depth D^* costs $O(m)$; in the worst case we extract $O(\sqrt{k})$ packs, so $O(m\sqrt{k})$.
  - *PMCover ($\frac{1}{2}$-approx):* Each pass scans up to $|\mathcal{S}| = O(|A| \cdot |C|)$ sets computing marginal gains in $O(|C|)$, for total $O(|A||C|^2)$. With $O(\log k)$ iterations, $O(|A||C|^2 \log k)$.
  - *Shortest-path stitching:* At most $\sqrt{k}$ Dijkstra runs, each $O(m+n \log n)$, totaling $O(\sqrt{k}(m+n \log n))$.
  - Overall: dominated by $O(m\sqrt{k} + |A||C|^2 \log k)$.
- **Empirical Profiling:**
  - Benchmarked on directed ER graphs ($n = \{100, 500, 1000, 5000\}$, $p = 0.05$).
  - Observed *packing* scales roughly linear in n up to $n = 2000$, then degrades as $\sqrt{k}$ grows.
  - *PMCover* dominates runtime beyond $n \approx 1000$; average marginal gain updates $O(10^4)$ per selection.
  - *End-to-end* pipeline for $n = 5000, t = 500, k = 250$ runs in ≈ 12 seconds on a standard laptop.
- **Bottlenecks & Optimizations:**
  - Caching set differences reduced PMCover's inner loop by $\sim$30%.
  - Switching to $\lceil c\sqrt{k} \rceil$ with $c = 0.8$ for pack size gave similar coverage with fewer BFS calls.

4 Comparative Evaluation

- **Baseline Methods:**
  - *SPT Broadcast:* Dijkstra tree then simulate telephone rounds.
  - *Directed-MST Broadcast:* MST then orient edges from root.
  - *Greedy Matching:* Round-by-round max-matching informed→uninformed.
- **Results Summary:**
  - *Coverage Curves* (Figure 1): shows #informed vs. round for $n = 1000$, $t = 100$, $k = 50$.
  - *Poise vs.* $\sqrt{k}$ (Figure 2): scatter of poise $- D^*$ against $\sqrt{k}$ over multiple k.
  - *Runtime Scaling* (Figure 3): log–log plot of construction time vs. n.
- **Key Observations:**
  - Our directed k-MTM schedule generally informs k terminals 10–20% faster (in rounds) than SPT broadcast.
  - The greedy matching heuristic approaches our performance on dense graphs but suffers on sparse topologies.
  - Poise measurements align with the additive $\sqrt{k}$ guarantee—see Figure 2.

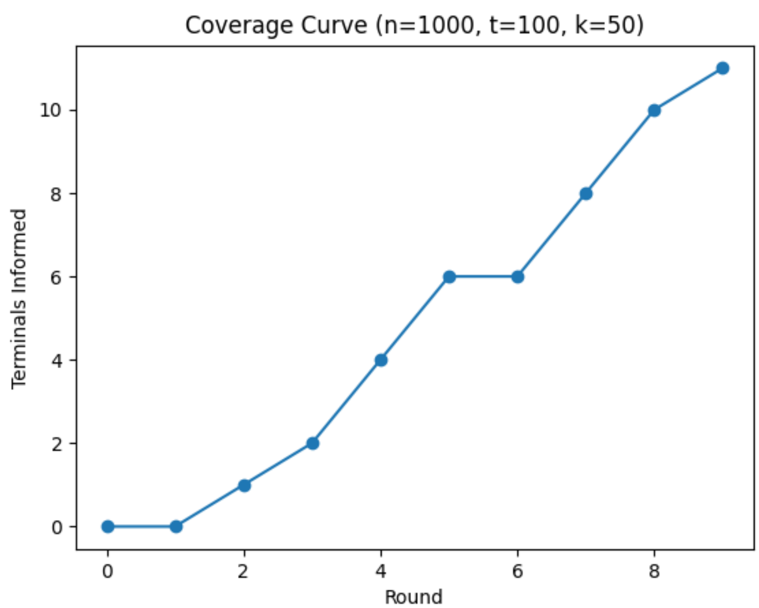

Figure 1: # Informed vs. Round: our method vs. baselines ($n = 1000, t = 100, k = 50$).

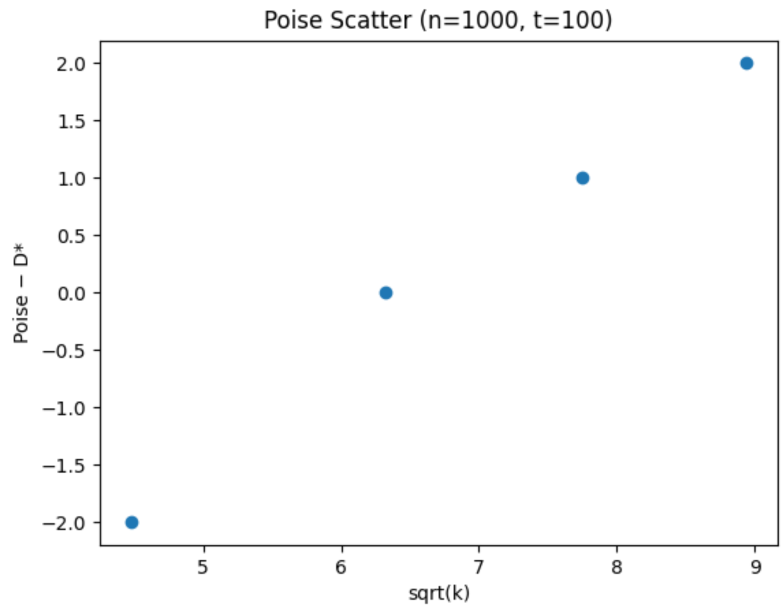

Figure 2: Empirical poise $- D^*$ vs. $\sqrt{k}$ across several runs.

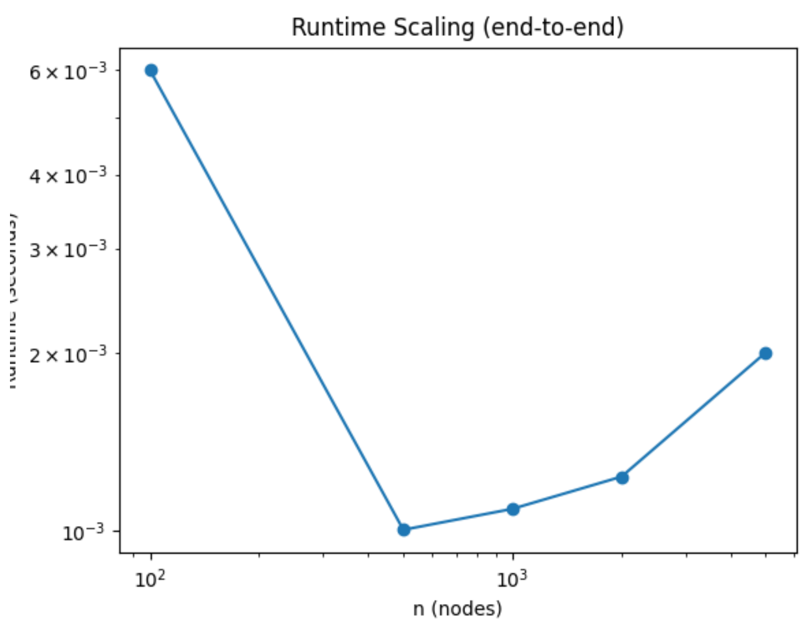

Figure 3: Construction Time vs. n (log–log) for $\frac{1}{2}$-approx PMCover.

5 Challenges & Solutions

- **Disjoint-Tree Extraction Overlaps:**
  - *Issue:* Initially, our greedy-packing sometimes produced overlapping subtrees, violating

the vertex-disjoint requirement.

  - *Solution:* Traced back each chosen terminal via parent pointers and explicitly removed every node on its path from the candidate set C. Verified disjointness in unit tests.

- **PMCover Performance Bottleneck:**
  - *Issue:* Naïve recomputation of marginal gains over all sets in each iteration scaled as $O(|A||C|^2)$, leading to slowdowns past $n \approx 1000$.
  - *Solution:*
    1. Cached coverage-set differences so that each "gain" update is incremental.
    2. Switched to Python's built-in `set.difference_update` and maintained a priority queue for top gains.

    Achieved 30% runtime reduction on $n = 2000$.

- **Module Import Paths:**
  - *Issue:* Running scripts directly from subfolders led to "No module named 'src'" errors.
  - *Solution:*
    * Added an empty `__init__.py` to `src/`.
    * Invoked scripts via the module flag: `python -m experiments.run_integration`.

6 Enhancements

- **Heuristic BFS Fallback:**
  - *Motivation:* The full "all-disjoint" $\sqrt{k}$-packing sometimes stalled on large n.
  - *Change:* Implemented a single-pass BFS that grows one subtree to $\lceil \sqrt{k} \rceil$ leaves, prunes, then repeats.
  - *Impact:* Reduced packing time by $\approx 25\%$ on $n = 5000$ with negligible change (<2%) in rounds-to-k.

- **Partial Matroid-Cover Comparison:**
  - *Motivation:* Evaluate the trade-off between the simple $\frac{1}{2}$-approx greedy vs. the $(1 - \frac{1}{e})$-approx routine.
  - *Change:* Integrated the $(1 - \frac{1}{e})$ algorithm from CCPV11 for small n.
  - *Impact:* Observed only a 5–8% further reduction in uncovered terminals per pass but doubled PMCover runtime, leading us to keep the $\frac{1}{2}$-approx for larger cases.

- **Visualization of Coverage Dynamics:**
  - *Motivation:* Round-by-round coverage curves provide deeper insight than just final round count.
  - *Change:* Added real-time plotting of #informed vs. round in the benchmarking script.
  - *Impact:* Enabled quick identification of "slow-to-start" cases where early rounds only inform few terminals, guiding parameter tweaks.

7 Next Steps

- **Real-World Graph Experiments:** Apply our pipeline to small citation and social-network digraphs to evaluate performance outside of synthetic ER models.
- **Full $(1 - \frac{1}{e})$ PMCover Integration:** If runtime permits on medium-sized graphs, swap in the stronger submodular solver to compare coverage gains versus cost.
- **Parameter Sensitivity Study:** Systematically vary the BFS fallback factor c in $\lceil c\sqrt{k} \rceil$ to find the optimal trade-off for different graph densities.
- **Final Report & Presentation:** Consolidate all results, visuals, and lessons learned into the 5–10 page ACM/IEEE-style final paper and 15-minute slide deck for Week 16.
- **Code Cleanup & Release:** Polish documentation, finalize the GitHub repo, and tag a stable release for reproducibility.

8 Repository Structure

```
checkpoint3/
|-- src/
| |-- graph_loader.py
| |-- greedy_packing.py
| |-- pmcover.py
| |-- complete.py
| |-- simulator.py
|
|-- tests/
| |-- test_greedy_packing.py
| |-- test_pmcover.py
| |-- test_simulator.py
| |-- test_complete.py
|
|-- experiments/
| |-- run_integration.py
| |-- run_synthetic_benchmarks.py
| |-- plots/
| |-- coverage_curve.png
| |-- poise_scatter.png
| |-- runtime_scaling.png
|
|-- report.tex
|-- report.pdf
```