CSE565 Final project

Randomized Competitive Greedy Algorithm for Flow Decomposition

Wzc5440 Wei-An, Chen

1. Algorithm Overview

This work proposes a Randomized Competitive Greedy Algorithm for the Minimum Flow Decomposition (MFD) problem. Instead of deterministically extracting paths or cycles in a fixed sequence (which often leads to premature structural fragmentation) the method introduces a competitive bidding mechanism in which candidate paths and cycles compete based on their bottleneck capacities, adjusted by stochastic perturbations.
The algorithm also employs Monte Carlo search: multiple randomized decomposition sequences are explored within a time budget, and the best solution is retained. All core logic is implemented in the decompose_flow_competitive function, which coordinates path extraction, cycle discovery, and competitive decision-making.

2. Detailed Method

2.1 Noisy Widest-Path Extraction (Path Candidate)

Func: find_widest_path
Classical widest-path algorithms rely on fully deterministic Dijkstra relaxation. However, selecting the absolute strongest path at every iteration often results in overly dominant, elongated paths that unintentionally destroy alternative flow routes. This behavior can severely reduce the quality of the final decomposition by causing local improvements at the expense of global structure.
To address this issue, a multiplicative noise factor is injected into each relaxation step:

```
# 2. add rnad noise
noise = rng.uniform(0.8, 1.2)
perceived_flow = true_flow * noise

new_cap = min(d, perceived_flow)
```

Introducing noise weakens the determinism of the relaxation process, preventing the algorithm from repeatedly committing to the same “greedy-maximal” path. As a result, the search is encouraged to explore nearby alternatives that may have slightly smaller capacities but lead to more structurally coherent decompositions. This “Noisy Dijkstra” mechanism therefore provides controlled randomness that improves global solution quality by allowing beneficial deviations from the dominant path.

```
# 1. randomly mixup the order
neighbors = [v for v in adj[u] if flow.get((u, v), 0) > 0]
rng.shuffle(neighbors)
```

In addition, the ordering of outgoing neighbors is randomized before each relaxation step, preventing the search from following a fixed topological pattern and further reducing fragmentation. After a candidate path is obtained, the true bottleneck is recomputed to ensure correctness.

2.2 Global Widest-Cycle Search (Cycle Candidate)

func: get_best_cycle

```python
def get_best_cycle(adj, flow, V, rng):
    """
    Scans for the cycle with the LARGEST bottleneck.
    """
    candidates = []

    # Gather potential start nodes (nodes with high output flow)
    node_flow_out = []
    for u in range(1, V + 1):
        f_out = sum(flow.get((u, v), 0) for v in adj[u])
        if f_out > 0:
            node_flow_out.append((f_out, u))

    if not node_flow_out:
        return None, 0

    node_flow_out.sort(key=lambda x: x[0], reverse=True)

    # Check top K active nodes
    check_limit = min(len(node_flow_out), 25)

    for _, start_node in node_flow_out[:check_limit]:
        cyc = find_cycle_dfs(adj, flow, start_node, rng)
        if cyc:
            bn = min(flow[(cyc[i], cyc[(i+1) % len(cyc)])] for i in range(len(cyc)))
            # Prioritize Cycle with larger bottleneck
            candidates.append((bn, cyc))

            # Early exit if we found a massive cycle
            if bn > node_flow_out[0][0] * 0.9:
                return cyc, bn

    if not candidates:
        return None, 0

    candidates.sort(key=lambda x: x[0], reverse=True)
    return candidates[0][1], candidates[0][0]
```

Rather than taking the first cycle encountered in a local DFS, this method performs a global search to identify the cycle with the maximum bottleneck capacity:

1. Nodes with positive outgoing flow are ranked by total outflow
2. Only the top K=25 nodes are used as DFS seeds.
   This balances computational efficiency with good cycle coverage.
3. Each DFS uses a flow-guided randomized expansion, where outgoing edges are sorted by a flow-weighted score with small injected noise. This prioritizes promising structural directions while preserving exploration diversity.
4. All discovered cycles are evaluated, and the one with the largest bottleneck is selected.

Furthermore, if a discovered cycle achieves a bottleneck close to the maximum outgoing flow of the entire graph, the search terminates early. This heuristic prevents unnecessary DFS exploration and ensures that highly valuable cycles are extracted promptly.

```python
nbs = [v for v in adj[start_node] if flow.get((start_node, v), 0) > 0]
# Add noise to randomize search
nbs.sort(key=lambda x: flow.get((start_node, x), 0) * rng.uniform(0.9, 1.1), reverse=True)
```

2.3 Competitive Selection with Dynamic Bias

Corresponding code: main loop in decompose_flow_competitive

After generating a path candidate P and a cycle candidate C, the algorithm determines which one to extract using a dynamic bias factor:

In code:

```
# 3. COMPETITION LOGIC

# Expand Bias Range: 0.5 (Path Favor) to 3.0 (Cycle Favor)
bias = rng.uniform(0.5, 2.0)

take_cycle = False

if cyc_cand and cyc_bn > 0:
    if not path_cand or path_bn == 0:
        take_cycle = True
    elif cyc_bn * bias >= path_bn:
        take_cycle = True
```

During empirical evaluation, I observed that cycles tend to introduce more severe fragmentation in the residual flow graph compared to paths. Once a path is extracted, any cycles that share its edges may be partially destroyed, often resulting in multiple low-capacity fragments. In contrast, extracting cycles earlier preserves the structural integrity of the remaining flow and prevents the creation of scattered, difficult-to-merge components.
To address this, I modified the competitive selection rule to favor cycles more strongly by increasing the expected value of the bias parameter

3. Refinements and Optimization

Several refinements improve the decomposition quality:

1. Monte Carlo Randomized Search
   Multiple decomposition sequences are generated from different seeds; the minimal solution is kept.
2. Structural Anti-Fragmentation
   Replacing naive DFS cycles with get_best_cycle ensures high-capacity cycles are extracted before they can be destroyed by path removal.
3. Validity Checks and Pruning
   Extracted structures are sanitized, ensuring that:
   - extracted paths remain simple,
   - cycles close properly,
   - bottleneck deductions maintain flow correctness.
4. Canonical Merging
   Duplicate or rotated cycles are canonicalized and merged to minimize decomposition size.

4. Novel Contributions

This algorithm introduces two primary innovations:

1.Competitive Bidding Mechanism
Paths and cycles compete dynamically at every iteration, making the algorithm sensitive to the evolving structure of the residual graph rather than following a predetermined order.

2. Stochastic Perception of Capacities
By injecting noise into edge capacities, the deterministic widest-path heuristic is transformed into a local search method capable of escaping poor local minima.