# PRIORITY-BASED DATA PRESERVATION WITH VARIABLE PACKET SIZES: A SIZE-AWARE MAXIMUM WEIGHTED FLOW APPROACH

---

A Project

Presented to the Faculty of

California State University, Dominguez Hills

---

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

---

by

Jason Roe

Spring 2026

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This project extends the Maximum Weighted Flow with Uniform Cost (MWF-U) framework of Rivera and Tang for priority-based data preservation in base station-less sensor networks to a setting where data packets have heterogeneous sizes. The original MWF-U model assumes unit-sized packets, under which the Greedy Optimal Algorithm (GOA) is provably optimal. When sizes vary, GOA becomes suboptimal because large packets fragment limited storage, preventing smaller but collectively more valuable packets from being preserved. This project formulates the size-aware variant (MWF-S), demonstrates through counterexamples that GOA is no longer optimal, and designs seven new heuristic algorithms: Density GOA, Approx GOA, Hybrid GOA, DDR-GOA, DDR+-GOA, and PSB-GOA. A dual branch-and-bound sequential solver provides an ordered routing guide and GLPK-based integer linear programming (ILP) provides the true global optimum. I implemented and benchmark the two paper algorithms under their new names — MaxFlow_Greedy and MaxWeightedFlow_Approximation — alongside two ILP baselines (MaxFlow_ILP and MaxWeightedFlow_ILP). Two theoretical guarantees hold across all experiments: MaxFlow_Greedy always preserves at most as many packets as MaxFlow_ILP; MaxWeightedFlow_Approximation always achieves at most the priority of MaxWeightedFlow_ILP. The comparison between the two heuristics is instance-dependent and depends on the storage-to-demand ratio. ILP verification reveals that the dual B&B solver is not the true optimum. GLPK finds solutions up to 82 percent better on some instances due to its ability to interleave routing across multiple DGs in parallel. Simulations on 10- and 15-node networks show that Hybrid GOA and PSB-GOA achieve 100 percent of the B&B bound and extended stress tests on 100-node configurations confirm that no single algorithm dominates across all storage domains.

# CHAPTER 1

# INTRODUCTION

## Motivation and Significance

Sensing applications are popping up in tough environments like underwater exploration, deep space missions, underground pipeline monitoring, and volcanic eruption surveillance. The thing these all have in common is that you can't just set up high-power data-collecting base stations out in the field. In these settings, sensor nodes must store collected data inside the network until external uploading opportunities arise. For instance, through periodic visits by unmanned aerial vehicles, data mules, or autonomous underwater vehicles. Rivera and Tang refer to these systems as base station-less sensor networks (BSNs) [1]. Unlike conventional sensor networks that rely on continuous connectivity to a central base station for real-time data collection, BSNs operate in an intermittently connected regime where data must survive in-network for extended periods before any retrieval opportunity occurs.

The operational constraints of BSNs are severe and complex. Each sensor node has a finite energy budget that is consumed by both sensing and communication activities, and once depleted, the node becomes permanently inactive. Storage capacity at each node is limited by the physical memory available on low-cost sensor hardware. Communication links between nodes are constrained by transmission range, which in environments like deep ocean or underground tunnels may be significantly shorter than in open-air deployments. These constraints as a whole mean that the network must make intelligent decisions about which data to preserve and how to route it to storage locations. This is because the total volume of generated data will typically exceed the network’s combined storage and routing capacity.

Modern sensor networks have evolved from single datatype measurements to multifunctional infrastructures that generate heterogeneous data types. Environmental monitoring systems, for example, may simultaneously produce scalar readings such as temperature and humidity, vector measurements of object movement, and multimedia data from cameras and acoustic sensors [1]. These different data types carry different priorities: a video clip capturing a rare geological event may be far more valuable than a routine temperature reading. In resource constrained environments where not all data can be preserved, the system must decide which

data to keep in order to maximize the total preserved priority. This priority-based decision making is the core challenge that motivates the entire line of research into data preservation algorithms for BSNs.

Rivera and Tang [1] formulated this challenge as the priority-based data preservation (PDP) problem and solved it by transforming it into a maximum weighted flow (MWF) problem on a flow network derived from the BSN. In their MWF with uniform cost (MWF-U) formulation, unit flows from different source nodes have distinct weights that represent priorities. The goal is to maximize the total weight of flows routed from data generators to storage nodes. Under the assumption that all data packets have the same size, they proved that a greedy algorithm called the Greedy Optimal Algorithm (GOA, presented as Algorithm 3 in their paper) solves MWF-U optimally. It does so by processing source nodes in descending order of priority. This result is both theoretically strong and practically useful: the algorithm is simple to implement, runs in polynomial time, and is guaranteed to achieve the best possible total preserved priority.

However, the unit-size assumption rarely holds in practice. In a multi-modal sensor network, a high-resolution image may consume several times more storage than a scalar sensor reading. As an example, a temperature measurement might require only 1 kilobyte of storage, while a seismic waveform sample might require 4 kilobytes, and a compressed image captured by a camera sensor might require 8 kilobytes or more. When packet sizes differ across data generators, the storage at each node becomes a shared and heterogeneous resource. Routing one large packet may prevent multiple small packets from being stored, even if those small packets collectively carry greater total priority. This interaction between packet size and storage capacity introduces a combinatorial packing dimension that the original MWF-U formulation does not address.

This project is motivated by the need to bridge this gap. By extending the MWF-U framework to handle varying packet sizes, I address a more realistic model of data preservation that accounts for the heterogeneous nature of modern sensor data. The importance of this extension lies in both its real-world relevance and its theoretical implications. On the applied side, it enables better data preservation decisions in deployed BSNs. On the theoretical side, this project demonstrates that varying sizes fundamentally change the problem's structure and complexity. The resulting MWF-S problem connects to classical NP-hard problems in combinatorial optimization, including the 0-1 knapsack problem and the bin packing problem. This connection suggests that no polynomial-time exact algorithm exists unless P equals NP.

## Problem Statement

This project introduces the size-aware maximum weighted flow problem (MWF-S), which generalizes MWF-U by assigning each data generator node a packet size variable. Formally as written, in a BSN with k data generator (DG) nodes and l storage nodes, each DG node s_i has: a number of overflow data packets d_i, a packet priority (weight) v_i, and a packet size sz_i measured in storage units. Each storage node t_j has a storage capacity m_j measured in the same storage units. All nodes have limited energy E_i. The goal of MWF-S is to find a valid flow f from the DG nodes to the storage nodes that maximizes the total preserved priority:

$$V\_f = \Sigma\_i \, (v\_i \times |f\_i|)$$

subject to energy constraints on intermediate relay nodes and sending capacity constraints on DG nodes. It also includes a critical size-aware storage constraint. Routing one packet from DG_i to storage node t_j consumes sz_i units of t_j's storage capacity. Unlike MWF-U, where each packet consumes exactly one unit of storage, MWF-S requires the algorithm to consider how well storage is used by packets of different sizes.

The key research question is whether the greedy priority-based approach of GOA can be modified to achieve high or near-optimal total preserved priority when packet sizes are heterogeneous. This project demonstrates that GOA is suboptimal for MWF-S and designs new heuristic algorithms that account for both priority and size.

## Contributions

The contributions of this project are as follows. First, I formulate the MWF-S problem, extending the MWF-U model of Rivera and Tang to accommodate heterogeneous packet sizes while retaining uniform flow costs. Second, this project shows through counterexample that GOA—the optimal algorithm for MWF-U—is suboptimal for MWF-S. Specifically, I construct a network instance where GOA achieves a total priority of 20 while the optimal solution achieves 42. Third, I design seven new heuristic algorithms. Density GOA sorts by value density, which is priority per storage unit. Approx GOA guarantees at least half the optimal. Hybrid GOA uses prefix enumeration and best-fit storage selection. DDR-GOA and DDR+-GOA dynamically reorder data generators based on residual network state. PSB-GOA scores every possible routing decision at each step. Fourth, I implement a dual branch-and-bound sequential solver that searches both best-fit and BFS routing strategies with a shared global bound. Fifth, I implemented and benchmark two algorithms for new paper under clarified names. MaxFlow_Greedy (Algorithm 1) sorts by packet size in ascending order to maximize packet count. MaxWeightedFlow_Approximation

(Algorithm 2) uses an all-or-nothing ordering to maximize priority. I also create two ILP baselines (MaxFlow_ILP and MaxWeightedFlow_ILP) and prove two guarantees that hold across all experiments. MaxFlow_Greedy never finds more packets than MaxFlow_ILP, and MaxWeightedFlow_Approximation never achieves higher priority than MaxWeightedFlow_ILP. Sixth, I use GLPK to show that the dual B&B solver does not always find the true optimum. Some instances show gaps as large as 82.2 percent because the B&B search routes packets one at a time. Seventh, I run simulations on random BSN topologies and find that Hybrid GOA and PSB-GOA match the full B&B bound on networks up to 15 nodes. Every heuristic avoids constraint violations in all trials. I also run four stress tests on 100-node networks covering extreme bottleneck, medium bottleneck, storage-scarce, and storage-abundant domains. No single algorithm wins in every case, and the best choice depends on the ratio of available storage to demand.

## Report Organization

The remainder of this report is organized as follows. Chapter 2 reviews related work in maximum flow algorithms, weighted flow problems, multi-commodity flow, knapsack and bin packing, data preservation in sensor networks, and branch-and-bound techniques. Chapter 3 provides the necessary background on the BSN model, the flow network transformation, the original MWF-U problem, and formally introduces the size-aware extension MWF-S. Chapter 4 describes each algorithm in detail, including pseudocode, correctness arguments, and complexity analysis. Chapter 5 describes the experimental setup, including network topology generation, variable ranges, and trial design. Chapter 6 presents and analyzes the experimental results. Chapter 7 concludes with a summary of findings and directions for future work. Appendix A provides the GitHub repository link and Appendix B with the live demo recording link and more info on the repository.

# CHAPTER 2

# RELATED WORK AND TECHNIQUES

This chapter surveys the existing literature and techniques that are relevant to this project. The work draws on several established research areas. These include maximum flow algorithms, weighted flow problems, multi-commodity flow, and combinatorial optimization such as knapsack and bin packing. It also builds on prior research in data preservation for sensor networks and branch-and-bound methods for finding exact solutions.

## Maximum Flow Algorithms

The maximum flow problem is one of the most fundamental problems in combinatorial optimization and graph theory with applications ranging from transportation networks, telecommunications, scheduling, and bipartite matching. Given a flow network with a source s, a sink t, and edge capacities, the goal is to find the maximum amount of flow that can be sent from s to t while respecting capacity constraints and flow conservation at every intermediate node. The problem was first solved by Ford and Fulkerson [2], who established the augmenting-path paradigm. Their approach repeatedly finds a path from source to sink with available capacity in the residual graph, augments flow along that path, and repeats until no augmenting path exists. The max-flow min-cut theorem [2] establishes the core condition for optimality. A flow reaches its maximum when no augmenting path remains from source to sink in the residual graph. Put another way, the flow value at that point equals the smallest total capacity of any cut that separates the source from the sink

Edmonds and Karp [3] refined the Ford-Fulkerson method by requiring that augmenting paths be found through breadth-first search (BFS). This guarantees that the shortest augmenting path by number of edges is selected at each step. Though the change appears minor, its impact on performance is significant. The number of augmenting path iterations is guaranteed to be at most $O(|V| \cdot |E|)$, resulting in an overall time complexity of $O(|V| \cdot |E|^2)$. This is a major improvement over the original Ford-Fulkerson method, which can exhibit exponential behavior when paths are chosen without a specific strategy and capacities are irrational. The Edmonds-Karp algorithm forms the basis for the Modified Edmonds-Karp algorithm (Algorithm 1) in Rivera and Tang's paper. Their version adapts the classic algorithm to handle multiple sources and sinks with limited sending and receiving capacities [1].

More recent advances in maximum flow include Goldberg and Tarjan's push-relabel algorithm and Dinic's algorithm. The push-relabel approach maintains a preflow and pushes excess flow toward the sink using a labeling scheme. Dinic's algorithm uses blocking flows in layered graphs and achieves $O(|V|^2 \cdot |E|)$ time complexity. These algorithms achieve better asymptotic complexity in certain settings but are not used in this project. The Edmonds-Karp BFS-based approach remains the most natural fit for the MWF-U framework because it finds the shortest augmenting paths. This minimizes relay hops and energy consumption, which is a critical factor in energy-constrained sensor networks where every relay hop depletes a node's finite battery.

The concept of residual graphs is at the core of all augmenting-path algorithms. Given a flow network G with current flow f, the residual graph G_f contains two types of edges. Forward edges have residual capacity c(u,v) - f(u,v) for each edge (u,v) where flow has not reached capacity. Backward edges have capacity f(u,v) for each edge carrying positive flow. Backward edges allow the algorithm to reroute previously committed flow, which is necessary for reaching the maximum flow. Without them, the algorithm could become stuck in a suboptimal flow configuration with no forward path remaining. In Rivera and Tang's multi-source multi-sink setting [1], the residual graph must also account for limited sending capacities at source nodes and receiving capacities at sink nodes. This leads to the concept of a feasible augmenting path (FAP), which respects these additional limits beyond typical edge capacities.

The max-flow min-cut theorem [2] states that in any flow network, the maximum amount of flow from source to sink equals the minimum capacity of any cut separating source from sink. This relationship provides the optimality condition for all maximum flow algorithms and connects deeply to linear programming. Rivera and Tang [1] extend this to the multi-source multi-sink setting by defining S-T cuts (Definition 5 in their paper). These cuts partition the vertex set into two subsets, one containing all sources and the other containing all sinks. However, they observe that the max-flow min-cut theorem alone is not enough to guarantee MWF-U optimality. The reason is that it cannot distinguish between maximum flows with different total weights. Two flows may both be maximum in volume but differ greatly in total priority.

In this project, the BFS-based augmenting-path approach from Edmonds-Karp acts as the routing engine behind GOA, Density GOA, DDR-GOA, and DDR+-GOA. Hybrid GOA and PSB-GOA extend this by incorporating best-fit storage selection. Because BFS finds paths with the fewest hops, it helps reduce the energy that relay nodes spend along each routing path. This routing choice aligns with the energy-minimization goal present

in sensor network deployments, where longer paths drain more relay energy and shorten the network's overall lifetime.

## Maximum Weighted Flow

The maximum weighted flow (MWF) problem generalizes classical maximum flow by assigning weights or priorities to flows from different source nodes. Instead of simply maximizing total flow volume, the objective becomes maximizing the total weight of routed flows. Rivera and Tang [1] defined two variants relevant to sensor networks. In MWF-U, all unit flows have the same cost, meaning each unit flow consumes one unit of edge capacity regardless of its source. In MWF-H, unit flows from different sources consume different amounts of edge capacity.

For MWF-U, Rivera and Tang proved that their Greedy Optimal Algorithm (GOA, Algorithm 3) is optimal. GOA processes source nodes in descending order of weight and pushes maximum flow from each source using the Modified Edmonds-Karp algorithm. Their proof of optimality uses a constructive argument (Algorithm 4). It shows that no optimal solution can exceed the total weight achieved by GOA. This constructive approach reveals that MWF-U has a matroid-like structure [1].

Rivera and Tang also introduced the Max-Weight Min-Cost Theorem (Theorem 3), which offers a different way to verify optimality for MWF-U. The theorem shows that finding the maximum weighted flow in a capacity-restricted network is the same as finding the minimum cost flow in a transformed version of that network. In this transformation, source edges receive negative costs equal to their weights [1]. For MWF-H, they showed the problem is NP-hard by reducing it from the Unbounded Knapsack Problem. To address this, they proposed a 2-approximation algorithm under the all-or-nothing model (Algorithm 5) [1].

This project focuses exclusively on extending MWF-U to varying packet sizes. I do not address MWF-H because my model keeps costs uniform. Each unit flow consumes one unit of edge capacity on routing edges, just as in MWF-U. The heterogeneity in my setting applies to storage rather than edge capacity. This distinction matters. In MWF-H, edge capacities must accommodate the weighted sum of flows, which makes the problem NP-hard. In MWF-S, edge capacities on routing edges are consumed uniformly. Only the sink edges exhibit size-dependent behavior.

The Max-Weight Min-Cost Theorem (Theorem 3 in [1]) reveals a strong connection between maximum weighted flow and minimum cost flow. For my size-aware extension, this equivalence does not directly apply. The

storage constraint introduces a non-uniform capacity dimension that is not captured by typical minimum cost flow formulations.

## Multi-Commodity Flow

Multi-commodity flow (MCF) [4] looks at how to route multiple different types of flows through a shared network with limited edge capacities at the same time. Each type of flow is called a commodity and has its own source, sink, and demand. The difficulty is that all commodities share the same edge capacities. The MCF variant closest to my work is maximum weighted MCF (MW-MCF). In this variant, source-sink pairs carry different weights and the goal is to maximize total weight. MW-MCF is APX-hard, which means no constant-factor approximation algorithm exists unless P equals NP [1]. This result shows how hard weighted flow problems are in general and supports the reasoning behind the heuristic approach taken in this project.

A key distinction between MWF and MCF is the routing model. MCF is a one-to-one model. Each commodity has a designated source-sink pair, so flow from source s_i must reach its specific destination t_i and cannot be sent to any other sink. MWF uses an any-to-any model Flows from any source can be routed to any sink [1]. In the sensor network context, this means data from any data generator can be stored at any storage node as long as a feasible path exists. This structural difference means that solution techniques for MWF and MCF are different. MCF research has focused on tree topologies with unsplittable flows and path-coloring techniques. MWF assumes splittable flows on general graphs. Recent advances in MCF algorithms have achieved near-linear time approximation for k-commodity problems on directed graphs. However, these do not directly apply to my size-aware extension. The routing model is different, and the size-dependent storage limits in my problem have no equivalent in MCF.

## Knapsack and Bin Packing Problems

The introduction of varying packet sizes creates a combinatorial packing dimension in MWF-S that is absent in MWF-U. This packing aspect connects our problem to the well-studied knapsack and bin packing families of problems.

The 0-1 knapsack problem [5] asks how to select from a set of items, each with a weight and a value, to maximize total value without exceeding a fixed knapsack capacity. The connection to my work is direct. When the

flow network reduces to a single storage node, the MWF-S problem becomes a knapsack problem. DG packets act as items, priorities serve as values, packet sizes serve as weights, and storage capacity is the knapsack capacity. This reduction confirms that MWF-S is at least as hard as the knapsack problem.

The fractional knapsack problem allows items to be divided and is solvable in polynomial time by a greedy algorithm that sorts items by value-to-weight ratio [5]. My Density GOA algorithm draws directly from this approach. It sorts DG nodes by priority per storage unit (v_i / sz_i). Because packets are whole and cannot be split in my setting, Density GOA faces the same suboptimality as the greedy approach for the 0-1 knapsack problem.

The bin packing problem [6] is the complementary problem. Given items of various sizes and bins of fixed capacity, the goal is to pack items into the fewest bins possible. Best-fit and first-fit-decreasing heuristics [6] provide strong approximations. The best-fit heuristic directly inspires the best-fit storage selection strategy used in Hybrid GOA and PSB-GOA. The interaction between knapsack and bin packing creates the core difficulty of MWF-S. An algorithm must decide which packets to preserve, which is a knapsack-like decision. It must also determine where to route them, which is a flow network routing problem. Finally, it must figure out how to pack them into storage nodes, which is a bin packing problem. All three decisions must be made together.

## Data Preservation in Sensor Networks

Data preservation in sensor networks has been explored from several angles, reflecting the range of deployment scenarios and optimization goals. Tang, Yao, and Choi [10] were the first to formalize the data preservation problem for intermittently connected sensor networks. They designed techniques based on maximum flow and showed that a sensor network graph can be converted into a flow network. In this conversion, node energy levels become edge capacities and storage capacities become sink-edge capacities. This transformation is the basis that all later work in this area builds on, including this project.

Rivera and Tang [11] extended this line of work by introducing data priority and solving the resulting MWF problem. Their conference paper at IEEE MASS 2013 established the greedy approach and proved its optimality for the uniform-cost case. The journal version [1] significantly expanded the results with the MWF-H formulation, NP-hardness proofs, approximation algorithms, a distributed algorithm, and extensive simulation results. The distributed algorithm is particularly relevant for real-world deployments because it allows sensor nodes to make

local routing decisions without requiring global network knowledge, which is often unavailable in BSN environments.

Tang and Raghavendra [12] studied the truthful and optimal data preservation problem. They combined game theory with network flow techniques to handle selfish sensor nodes that may misreport their private cost types. Their work used minimum cost flow and integer linear programming to observe packet-level flow behavior. They also designed mechanism-based games that achieve optimal data preservation with cheat-proof guarantees. This game-theoretic perspective complements my work. I focus on the algorithmic challenge of variable packet sizes under cooperative behavior, while their framework addresses the strategic challenge of incentive compatibility.

Tang, Jaggi, Wu, and Kurkal [13] formalized data redistribution as a minimum-cost flow problem. They used cost-per-unit-flow as the main ranking criterion in their heuristics. This provides a direct precedent for using value-per-resource ratios to guide routing decisions in storage- and energy-constrained sensor networks. Their work shows that ratio-based ordering heuristics can perform near-optimally in practice even when the underlying optimization problem is NP-hard. This insight directly motivates my Density GOA algorithm.

Underwater sensor networks are one of the main areas where BSNs are used. These networks deal with challenges that are specific to the ocean environment. Acoustic communication offers limited bandwidth, ocean currents constantly shift the network layout, and replacing batteries at depth is extremely difficult. Researchers have proposed data collection strategies using autonomous underwater vehicles and energy-harvesting techniques to help networks last longer. Even so, the core problem of preserving high-priority data when storage and energy are limited remains important across all of these deployments.

## Branch and Bound Methods

Branch and bound (B&B) is a general algorithmic framework for solving combinatorial optimization problems to provable optimality. B&B explores the solution space by breaking the problem into smaller subproblems through branching. It then computes bounds on the optimal solution within each subproblem. If a subproblem's bound shows it cannot contain a better solution than the current best known, it is discarded through pruning.

In this project, I build a dual branch-and-bound solver to serve as ground truth for my heuristic algorithms. It tries every possible ordering of DG nodes and uses the best heuristic solution as an initial pruning bound. The

solver runs two separate searches sharing a single global best, one using BFS routing and the other using best-fit routing. Both are needed because each strategy reaches different feasible solutions, and using only one misses optima that the other can find.

# CHAPTER 3

# BACKGROUND AND PROBLEM FORMULATION

This chapter establishes the formal framework for the project. I describe the sensor network model, the transformation into a flow network, the original MWF-U problem and its optimal algorithm, and then formally introduce the size-aware extension.

## Sensor Network Model

A base station-less sensor network (BSN) is modeled as an undirected graph G(V, E), where V is the set of sensor nodes and E is the set of communication links between nodes within transmission range. Following Rivera and Tang [1], the nodes in V are partitioned into three roles: data generators (DGs), storage nodes, and relay nodes. A DG node s_i generates d_i overflow data packets, each with priority v_i. A storage node t_j has available storage capacity m_j. All nodes, regardless of role, have a limited energy budget E_i that constrains the number of packets that can be routed through them.

In my extension, each DG node s_i also has a packet size parameter sz_i measured in storage units. Every packet from DG_i shares the same size sz_i, but packets from different DGs can vary in size. This captures the reality of multi-modal sensor data. Scalar readings might take up 1 storage unit, while images or acoustic samples could require 3 to 8 units.

The following table summarizes the notation used throughout this report, extending the notation of Rivera and Tang [1] with the size-aware parameters introduced in this project.

| Symbol | Description |
|---|---|
| G(V, E) | BSN graph with n sensor nodes and m edges |
| V_d = {s_1, ..., s_k} | Set of k data generator (DG) nodes |
| V_s = {t_1, ..., t_l} | Set of l storage nodes |
| E_i | Initial energy budget of node i |
| d_i | Number of overflow data packets at DG_i |
| v_i | Priority (weight) of each packet at DG_i |
| sz_i | Packet size of DG_i in storage units (new in MWF-S) |
| m_j | Storage capacity of storage node t_j in storage units |
| G'(V', E') | Capacity-restricted flow network (CFN) |

| | |
|---|---|
| s | Super source node in the CFN |
| t | Super sink node in the CFN |
| ca(u, v) | Capacity of edge (u, v) in the CFN |
| f_i(u, v) | Number of packets from DG_i on edge (u, v) |
| V_f | Total weight of flow f: V_f = Sigma_i (v_i x \|f_i\|) |
| ρ_i = v_i / sz_i | Value density of DG_i (priority per storage unit) |
| κ | Prefix enumeration depth (κ = 2 in Hybrid/PSB) |

*Table 1. Notation Summary.*

The CFN node layout uses the following indexing scheme: node 0 is the super source s; for each BSN node i, the in-node is 2i + 1 and the out-node is 2i + 2; node 2n + 1 is the super sink t. The total number of CFN nodes is 2n + 2 where n is the number of BSN nodes.

## Flow Network Transformation

Following Section VI of Rivera and Tang [1], the BSN graph G(V, E) is transformed into a directed capacity-restricted flow network (CFN) G'(V', E') as follows. Each BSN node i is split into an in-node i' and an out-node i'', connected by a directed edge (i', i'') with capacity equal to the node's energy budget E_i. This node-splitting technique is a standard method for enforcing node capacities using edge capacities in flow networks [4].

A super source s is added with directed edges (s, i') to each DG node's in-node, with capacity equal to d_i (the sending capacity). A super sink t is added with directed edges (j'', t) from each storage node's out-node, with capacity equal to m_j (the receiving/storage capacity). For each undirected BSN edge (u, v), two directed edges (u'', v') and (v'', u') are added with infinite capacity, representing the bidirectional communication links.

The size-aware extension modifies only the sink-edge behavior. When augmenting flow for DG_i with packet size sz_i, a storage node j can only accept a packet if its remaining capacity satisfies sinkCap[j] >= sz_i. After pushing Δ packets, the storage capacity is reduced by Δ × sz_i. All other edge capacities such as energy edges and routing edges are consumed uniformly regardless of packet size. Each packet routed through an intermediate node consumes exactly one unit of that node's energy, no matter how large the packet is. This reflects a reasonable assumption. The energy cost of transmitting a packet depends more on the communication protocol overhead, which is roughly constant per packet, than on the size of the packet's data payload.

The BFS-based feasible augmenting path (FAP) search in the CFN requires careful handling of the size-aware constraint. The BFS starts from the super source and explores the residual graph layer by layer. When it reaches a storage node's out-node j", it only extends the path to the super sink t if sinkCap[j"] is at least sz_i for the current DG being routed. The BFS is also restricted from entering other DGs' in-nodes. This ensures that flow attribution stays correct, so each augmenting path passes through exactly one DG in-node and exactly one storage node's out-node.

There is a subtle implementation detail in how the bottleneck is computed along an augmenting path. In MWF-U, the bottleneck is simply the minimum residual capacity across all edges of the path. In MWF-S, the sink edge (j" → t) has capacity measured in raw storage units, while all other edges have capacity measured in packet counts. To avoid mixing units, the bottleneck computation skips the sink edge entirely. Instead, it computes Δ as the minimum of pathBottleneck, remaining[i], and sinkCap[j"] / sz_i. Here, pathBottleneck is the minimum residual capacity of all non-sink edges. The sink edge is then augmented separately by Δ × sz_i storage units. Without this fix, the implementation silently mixed units and produced invalid flow solutions.

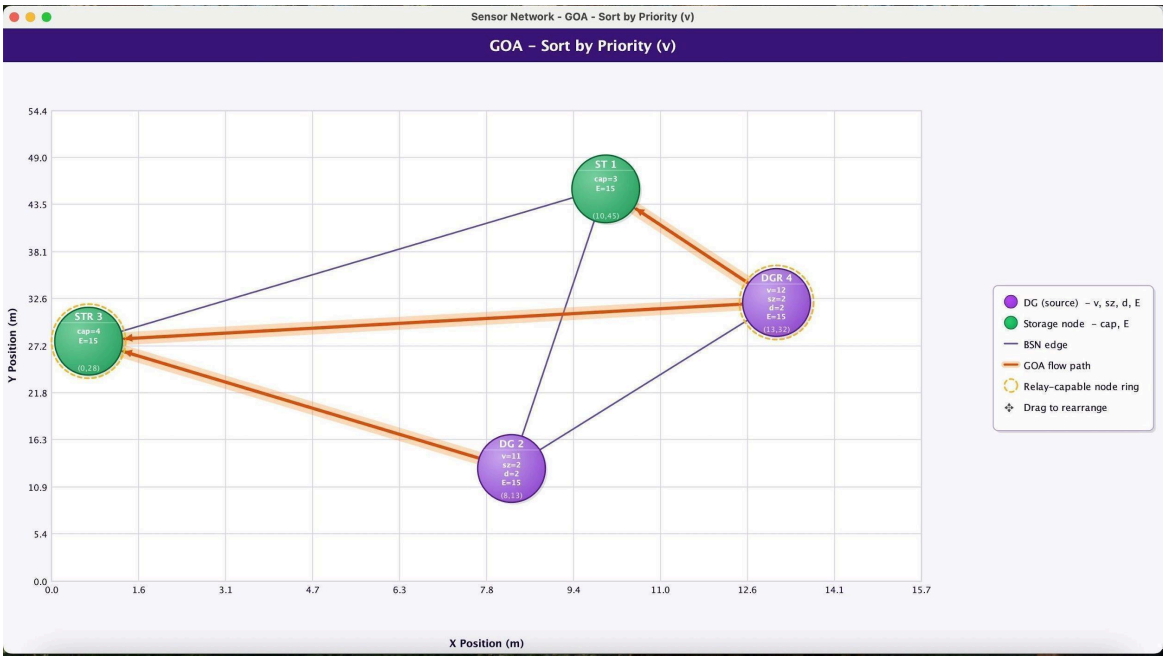

*Figure 1. Physical BSN graph for a 4-node network under GOA. Purple nodes are DGs showing priority (v), packet size (sz), overflow count (d), and energy (E). Green nodes are storage nodes showing capacity and energy. Orange edges indicate flow paths selected by GOA.*

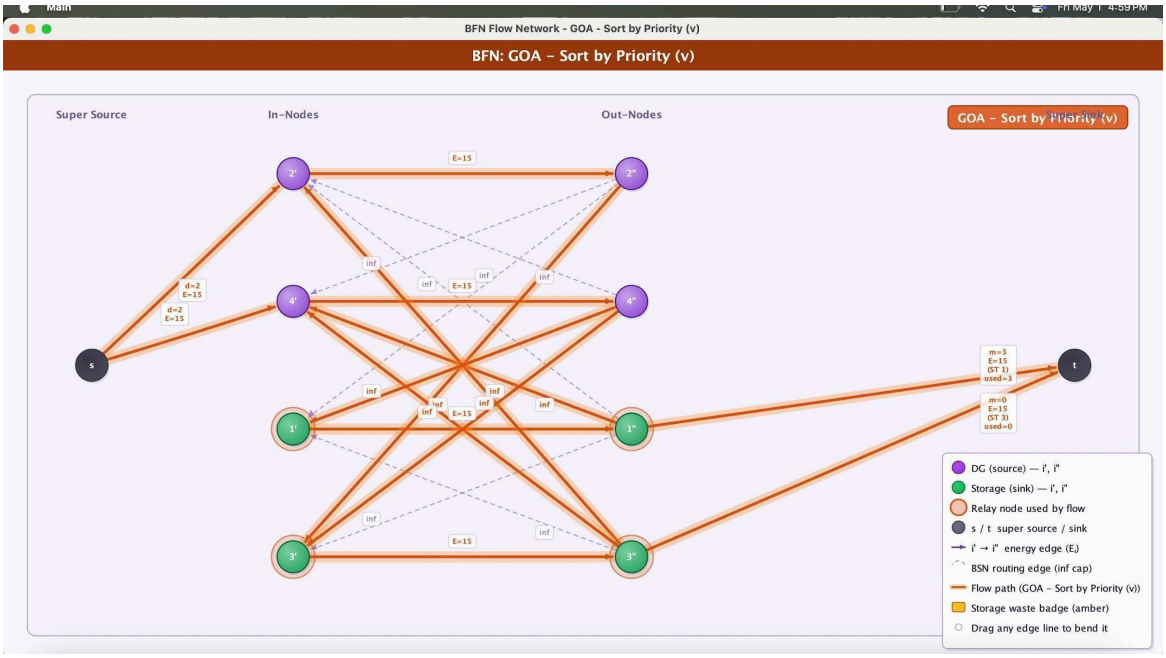


*Figure 2. Capacity-restricted flow network (CFN) for the same 4-node BSN under GOA. Each BSN node is split into in-node (i') and out-node (i"). Super source s connects to DG in-nodes with capacity d_i. Super sink t receives from storage out-nodes with capacity m_j. Orange edges show active flow paths.*

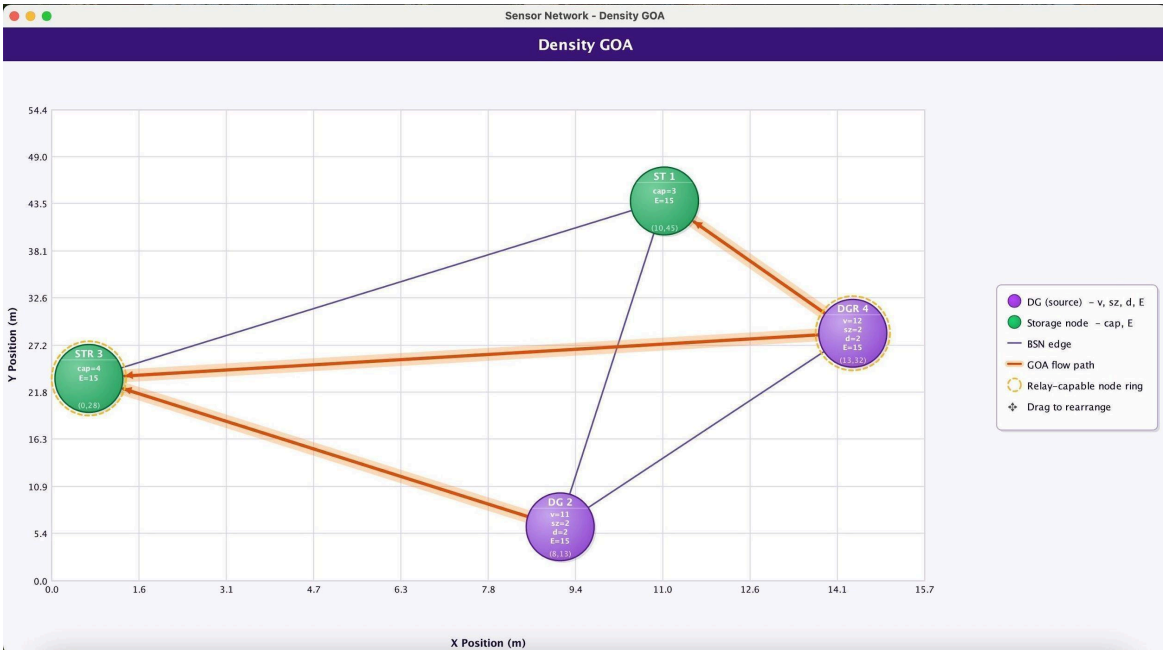


*Figure 3. Physical BSN graph under Density GOA. Orange edges indicate flow paths selected by sorting DGs by value density (ρ = v/sz).*

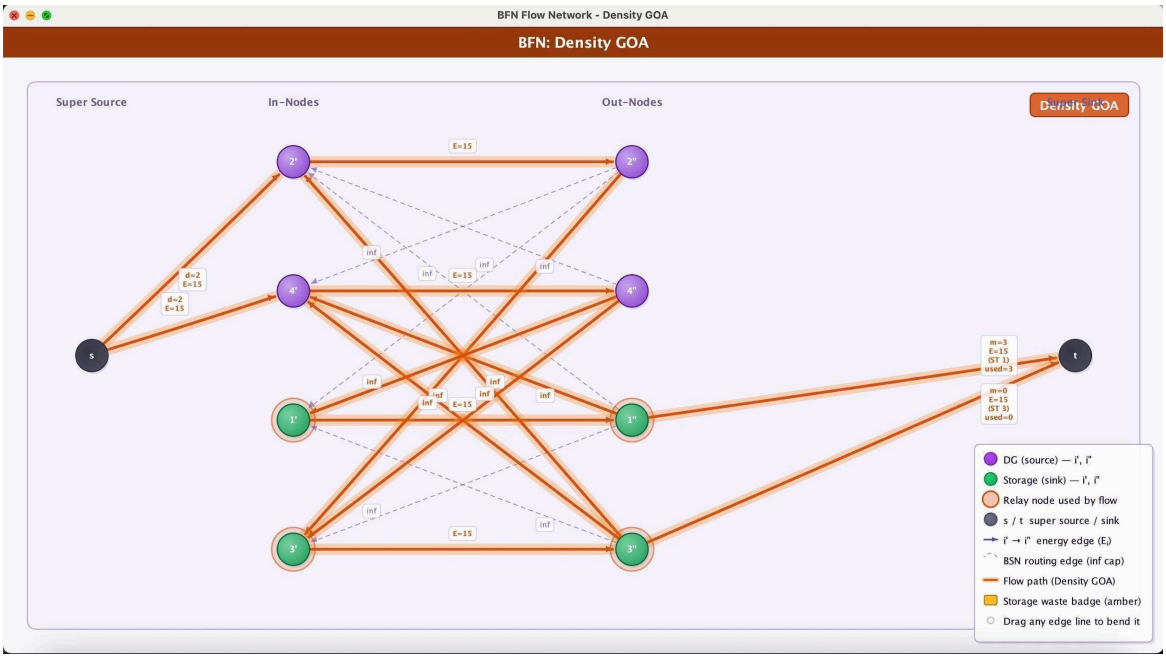


*Figure 4. CFN under Density GOA. Flow paths differ from GOA because DGs are sorted by value density rather than raw priority.*

## MWF-U Problem and the Greedy Optimal Algorithm

The MWF-U problem, as formulated by Rivera and Tang [1], is defined on the CFN G'(V', E') with source nodes S = {s_1, ..., s_k}, sink nodes T = {t_1, ..., t_l}, sending capacities d_i, receiving capacities m_j, and source weights v_1 >= v_2 >= ... >= v_k. A valid flow f = {f_1, ..., f_k} must satisfy four constraints: edge capacity constraint, flow conservation, source sending constraint, and sink receiving constraint. The goal is to maximize V_f = Σ_i (v_i · |f_i|).

The Greedy Optimal Algorithm (GOA, Algorithm 3 in [1]) sorts source nodes in non-ascending order of their weights. It then pushes as much unit flow as possible from the super source through each source node using BFS to find the shortest feasible augmenting paths. Rivera and Tang proved (Theorem 2 in [1]) that GOA is optimal for MWF-U through a constructive argument involving Algorithm 4. The running time of GOA is O(k · |V| · |E|²).

The key insight behind GOA's optimality is Lemma 1 [1]. GOA yields a maximum flow. Each unit flow in MWF-U consumes exactly one unit of edge capacity everywhere, so the greedy processing order cannot cause a higher-weight source to consume capacity in a way that prevents a valid rearrangement. The total flow volume does not change based on source ordering in MWF-U. GOA's contribution is to allocate that fixed volume in a way that maximizes total weight.

## The Size-Aware Extension: MWF-S

I now formally define the size-aware maximum weighted flow problem (MWF-S). The input is a CFN G'(V', E') with DG nodes S = {s_1, ..., s_k}, storage nodes T = {t_1, ..., t_l}, sending capacities d_i, receiving capacities m_j (in storage units), source weights v_i, and packet sizes sz_i. MWF-S seeks a valid flow f that maximizes V_f = Σ_i (v_i · |f_i|). This is subject to four constraints. First, edge capacity constraints as in MWF-U. Second, flow conservation as in MWF-U. Third, source sending constraints as in MWF-U. Fourth, the size-aware receiving constraint. For each sink node t_j, the total storage consumed Σ_i (|f_i · (t_j)| · sz_i) must not exceed m_j.

To demonstrate that GOA is not optimal for MWF-S, consider a network with two DG nodes and one storage node with capacity 6. DG_0 has priority v_0 = 10, packet size sz_0 = 3, and d_0 = 2 overflow packets. DG_1 has priority v_1 = 7, packet size sz_1 = 1, and d_1 = 6 overflow packets. GOA processes DG_0 first due to higher priority, pushing 2 packets of size 3 and filling all 6 units of storage. Total priority is 2 × 10 = 20. The optimal

solution instead pushes all 6 packets from DG_1, each of size 1, achieving a total priority of $6 \times 7 = 42$. GOA achieves less than half the optimal.

This counterexample also shows that Lemma 1 from [1] no longer holds for MWF-S. GOA does not necessarily yield a maximum flow when measured in storage units consumed. The problem is that GOA's greedy priority ordering can lead to wasted storage space when large packets are chosen over smaller ones.

# CHAPTER 4

# TECHNIQUES AND IMPLEMENTATION

This chapter covers each algorithm designed for the MWF-S problem. They range from the baseline GOA to the most advanced heuristics and the exact solver. Every algorithm works on the capacity-restricted flow network (CFN) described in Chapter 3. They all use the same BFS-based feasible augmenting path subroutine. The only differences are in how they order DG nodes and choose storage nodes.

## GOA (Baseline Algorithm)

The Greedy Optimal Algorithm serves as the baseline for comparison. It is a direct implementation of Algorithm 3 from Rivera and Tang [1], extended to handle variable packet sizes in the sink capacity check. GOA sorts DG nodes in descending order of priority v_i and pushes as many packets as possible from each DG before moving to the next. The BFS finds shortest feasible augmenting paths from the super source through the current DG's in-node to the super sink. A path is feasible only if every edge along it has sufficient residual capacity and the destination storage node has at least sz_i units remaining. GOA's strength is its simplicity and theoretical backing: it is optimal for MWF-U. Its weakness for MWF-S is storage fragmentation. The time complexity is $O(k \cdot |V| \cdot |E|^2)$.

### Pseudocode: GOA (Size-Aware Extension of Algorithm 3)

```
Input: CFN G'(V', E'), S, T, d_i, v_i, sz_i, m_j
Output: flow f, total weight V_f
Sort DGs so that v_1 >= v_2 >= ... >= v_k
V_f <- 0; f(u,v) <- 0 for all (u,v)
for i = 1 to k do
   while (BFS finds FAP p through s_i with sinkCap[j] >= sz_i) do
      delta <- min(pathBottleneck, remaining[i], sinkCap[j]/sz_i)
      for each edge (u,v) in p do
         f(u,v) += delta; f(v,u) -= delta
```

```
        sinkCap[j] -= delta * sz_i
        remaining[i] -= delta
        V_f += delta * v_i
return f, V_f
```

## Density GOA

Density GOA changes the sorting criterion from pure priority to value density, defined as $\rho_i = v_i / sz_i$. This measures the priority earned per storage unit consumed. DGs are processed in descending order of density, and the routing mechanism using BFS augmenting paths stays the same as GOA. This approach comes directly from the fractional knapsack algorithm, which a greedy sort by value-to-weight ratio solves optimally [5]. In MWF-S, packets are whole and cannot be split, so Density GOA is not always optimal. Its time complexity matches GOA at $O(k \cdot |V| \cdot |E|^2)$.

### Pseudocode: Density GOA

```
Input: CFN G'(V', E'), S, T, d_i, v_i, sz_i, m_j
Output: flow f, total weight V_f
Compute rho_i <- v_i / sz_i for each DG_i
Sort DGs so that rho_1 >= rho_2 >= ... >= rho_k
Run GOA with this density-based ordering
return f, V_f
```

## Approx GOA

Approx GOA runs both GOA and Density GOA on separate fresh copies of the flow network and keeps whichever result has a higher total priority. This approach of taking the better of two solutions is borrowed from Algorithm 5 in Rivera and Tang [1], which provides a 2-approximation for MWF-H under the all-or-nothing model. This project proves that Approx GOA is guaranteed to reach at least half the optimal for MWF-S by adapting the proof of Theorem 5 from [1]. The tradeoff is that both algorithms must run, which doubles the constant factor and gives a time complexity of $O(k \cdot |V| \cdot |E|^2)$.

## Hybrid GOA

Hybrid GOA is the strongest method in this project because it does not follow one fixed order. In this problem, choosing the wrong data-generating node early can affect the storage space left for the rest of the data, so Hybrid GOA tests different starting choices before fully committing. This follows the rollout idea from Bertsekas, Tsitsiklis, and Wu [7], where the algorithm looks ahead instead of only making the next greedy choice.

The algorithm starts by testing every possible pair of data-generating nodes as the first two choices, then sends the rest of the data using density order, which favors packets that give more priority for the storage they use. The pair with the highest total priority is selected. Hybrid GOA also uses storage more carefully by checking which storage nodes can be reached and choosing the one that leaves the least wasted space, instead of using the first one found by BFS. This comes from the best-fit bin packing idea [6] and helps avoid small leftover spaces that future packets cannot use.After the first two nodes are selected, Hybrid GOA keeps testing the next best node by temporarily routing one option, checking how well the remaining data would fit, resetting the network, and trying another option. The best choice is kept, and the process repeats until no more data can be routed.

In the experiments, this helped Hybrid GOA reach 99.6 percent of the best known result on 10-node networks. Since the method gets slower when there are too many data-generating nodes, the program only uses it for 12 or fewer DGs and switches to Density GOA for larger networks.

### Pseudocode: Hybrid GOA (PE + Rollout + Best-Fit)

```
Input: CFN G', DG list, storage list, kappa = 2
Output: flow f, total weight V_f
bestTotal <- 0; bestPrefix <- null
for each ordered pair (a, b) of DGs do    // k^2 prefixes
   snapshot(G')
   augmentBestFit(a); augmentBestFit(b)   // commit prefix
   remaining <- DGs \ {a, b}, sorted by density
   for each candidate c in remaining do
      snapshot(G')
```

```
    augmentBestFit(c)
    densityGreedy(remaining \ {c})    // simulate rest
    pilotVal <- totalPriority()
    restore(G')
  pick candidate c* with highest pilotVal
  commit c*; repeat rollout until no candidates remain
  if totalPriority > bestTotal then
    bestTotal <- totalPriority; bestPrefix <- (a, b)
  restore(G')
Replay bestPrefix with full augmentation
return f, bestTotal
```

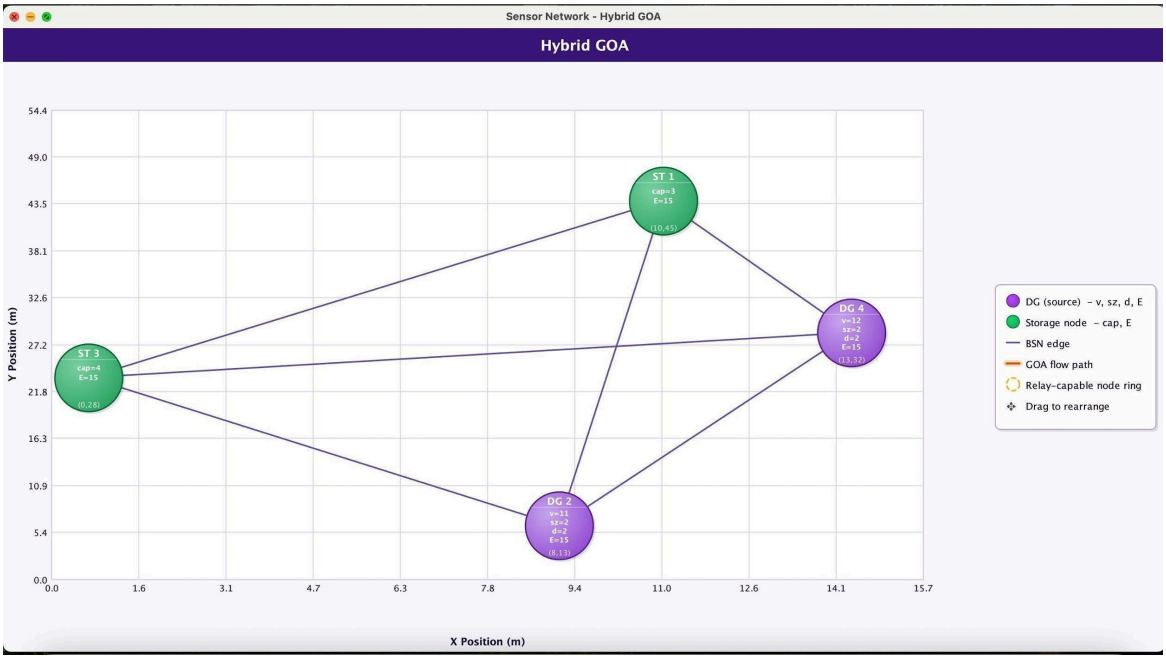


*Figure 5. Physical BSN graph under Hybrid GOA. The algorithm uses prefix enumeration to explore multiple starting orderings before committing to the best combination.*

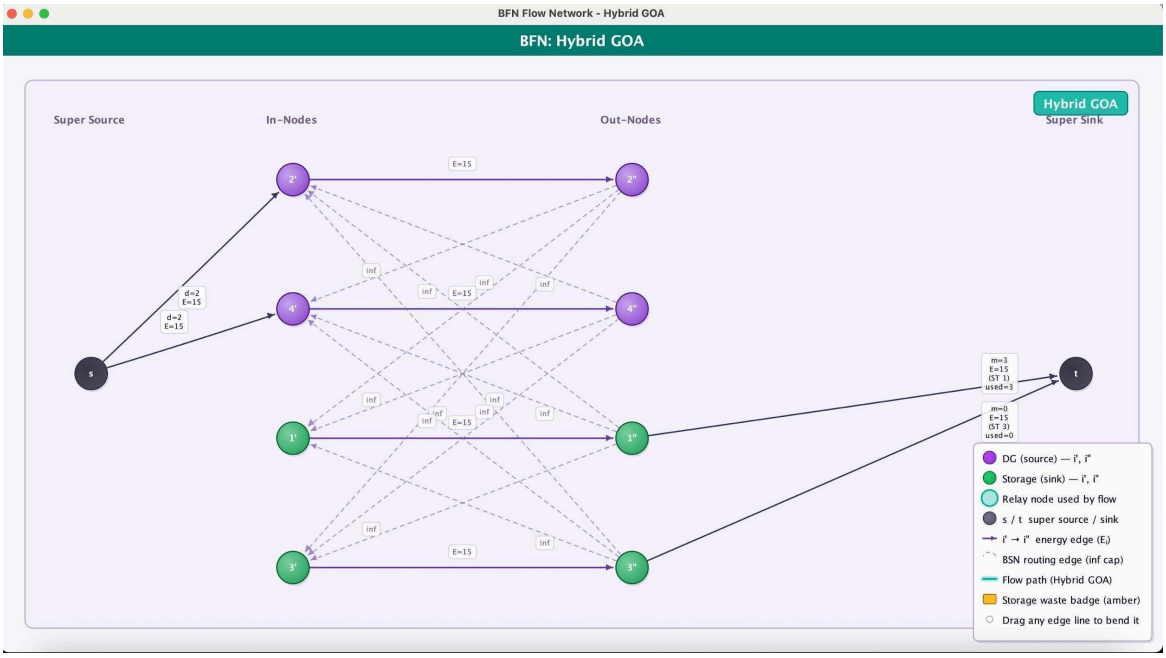


*Figure 6. CFN under Hybrid GOA. Cyan edges show flow paths selected by the best-fit storage routing strategy, which minimizes leftover storage fragments.*

## DDR-GOA (Dynamic Density Reordering)

DDR-GOA improves on fixed ordering by recalculating the DG order after each DG is fully routed. Instead of choosing the full order at the start, it updates its choice based on the current network state. It scores each remaining DG with effPri(i) = v_i × min(d_i, reachable_capacity / sz_i). This score estimates how much priority the DG can still contribute by considering its priority, remaining packets, packet size, and reachable storage. This helps the algorithm adjust when earlier routing choices use up storage or relay capacity. The idea follows the adaptive greedy approach discussed by Golovin and Krause [8]. Its time complexity is $O(k^2 \cdot |V| \cdot |E|^2)$.

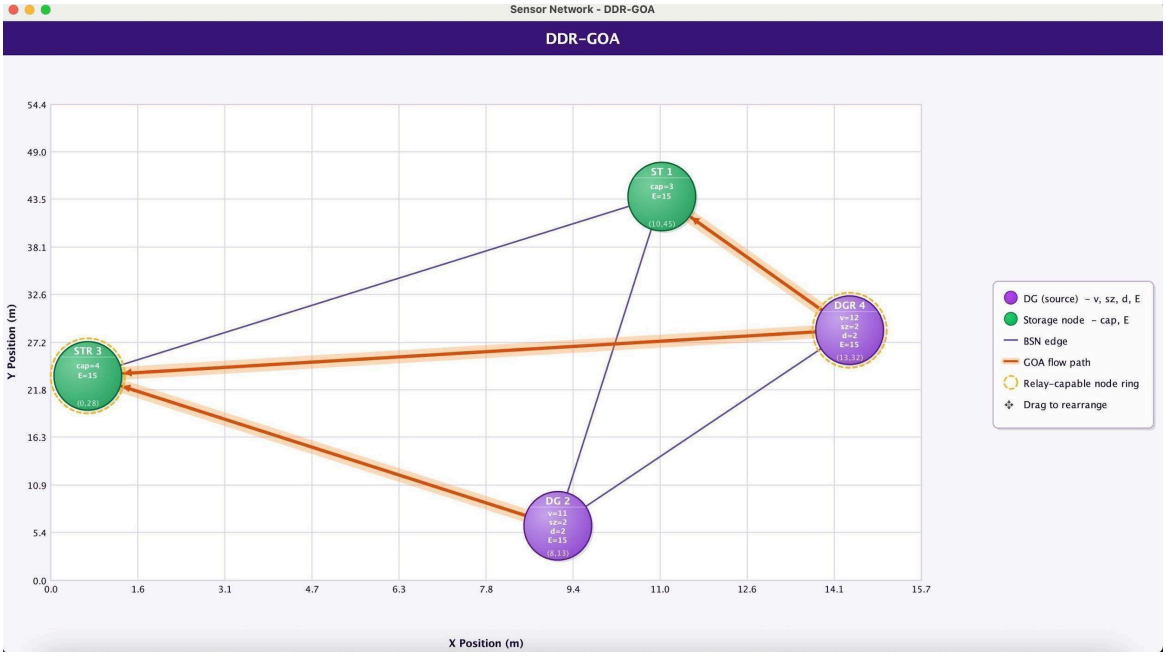


*Figure 7. Physical BSN under DDR-GOA. Orange flow paths show routes selected after dynamic reordering based on residual network state.*

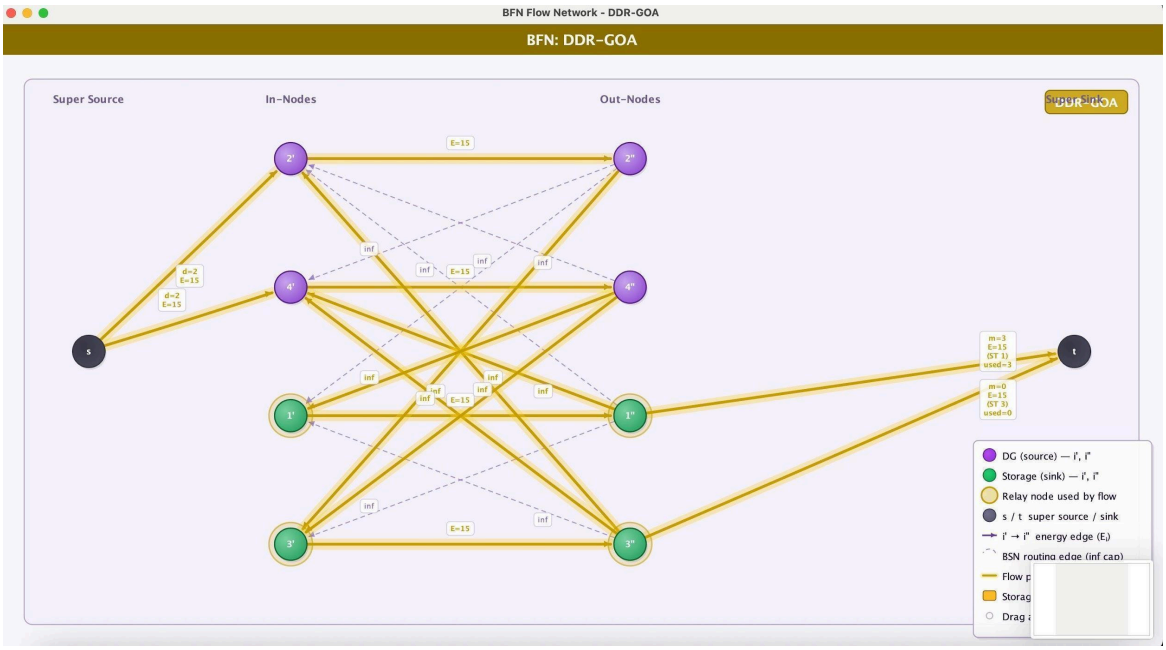


*Figure 8. CFN under DDR-GOA. Gold edges show flow paths. The dynamic reordering adapts to changing residual capacities after each DG is fully routed.*

## DDR+-GOA (Contention-Aware DDR)

DDR+-GOA builds on DDR-GOA by handling competition for the same storage node. In DDR-GOA, if several active DGs can reach one storage node, each DG treats that storage as fully available, making the estimate too optimistic. DDR+-GOA fixes this by counting how many active DGs can reach each storage node and then sharing the storage estimate using share(i, j) = sinkCap(j) / √(contention(j)). This lowers the estimated storage for nodes with high competition. The square-root adjustment acts as a middle ground because it considers competition without reducing the storage too aggressively. In the experiments, DDR+-GOA performed better than DDR-GOA on sparse networks, but it sometimes over-adjusted on dense networks where many nodes were connected and the competition estimate became less reliable.

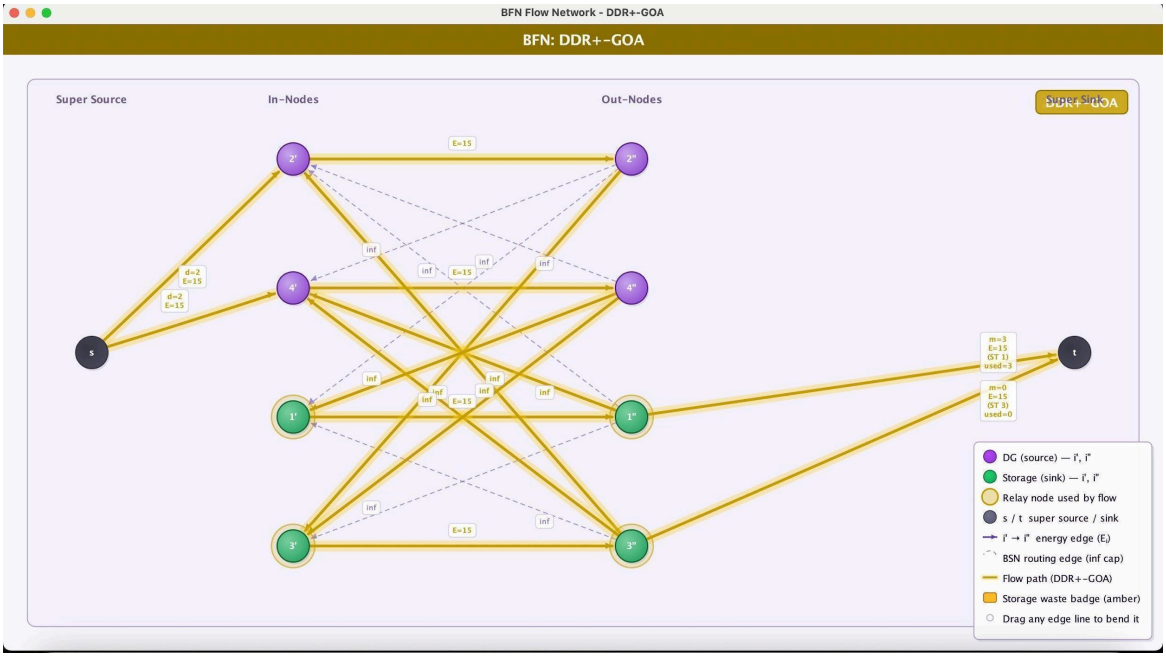


*Figure 9. CFN under DDR+-GOA. Gold edges show flow paths selected using contention-dampened effective priority scoring.*

## PSB-GOA (Per-Step Best-Path Scoring)

PSB-GOA is the most detailed routing method in this project. Instead of fully routing one DG before moving to the next, it checks every active DG at each step and chooses the best next move. For each possible DG and path, it calculates a score: score(i, path) = (v_i × Δ) / (Δ × sz_i + waste + relayPenalty(path) + 1). The top part

of the formula measures the priority gained, while the bottom part lowers the score when the move uses more storage, leaves wasted space, or uses relay nodes with low remaining energy.

This allows PSB-GOA to mix packets from different DGs instead of following one fixed order. In the implementation, it uses prefix enumeration with $\kappa = 2$ for the first choices, then switches to per-step scoring for the remaining routes. This gives it strong performance through a different strategy than Hybrid GOA. Hybrid GOA focuses more on testing different starting orders, while PSB-GOA focuses on choosing the best local move at each step. Since PSB-GOA becomes expensive when the number of DGs grows, it is capped at 12 DGs and switches to Density GOA for larger cases.

Pseudocode: PSB-GOA Per-Step Scoring (Tail Phase)

```
Input: residual CFN G', active DGs, sinkCap[]
Output: augmented flow, total weight gained
while any active DG can reach any storage node do
   bestScore <- -inf; bestDG <- null; bestPath <- null
   for each active DG_i do
      for each FAP p from s through s_i to t do
         delta <- computeDelta(p, i)
         waste <- (sinkCap[j] - delta*sz_i) mod minRemSize
         relayPen <- sum(delta / residualEnergy(r)) for relays r
         score <- (v_i * delta) / (delta*sz_i + waste + relayPen + 1)
         if score > bestScore then
            bestScore <- score; bestDG <- i; bestPath <- p
   augment(bestPath, bestDG)
   if remaining[bestDG] = 0 then deactivate bestDG
```


*Figure 10. Physical BSN under PSB-GOA. Orange flow paths show routes selected by the per-step scoring strategy.*

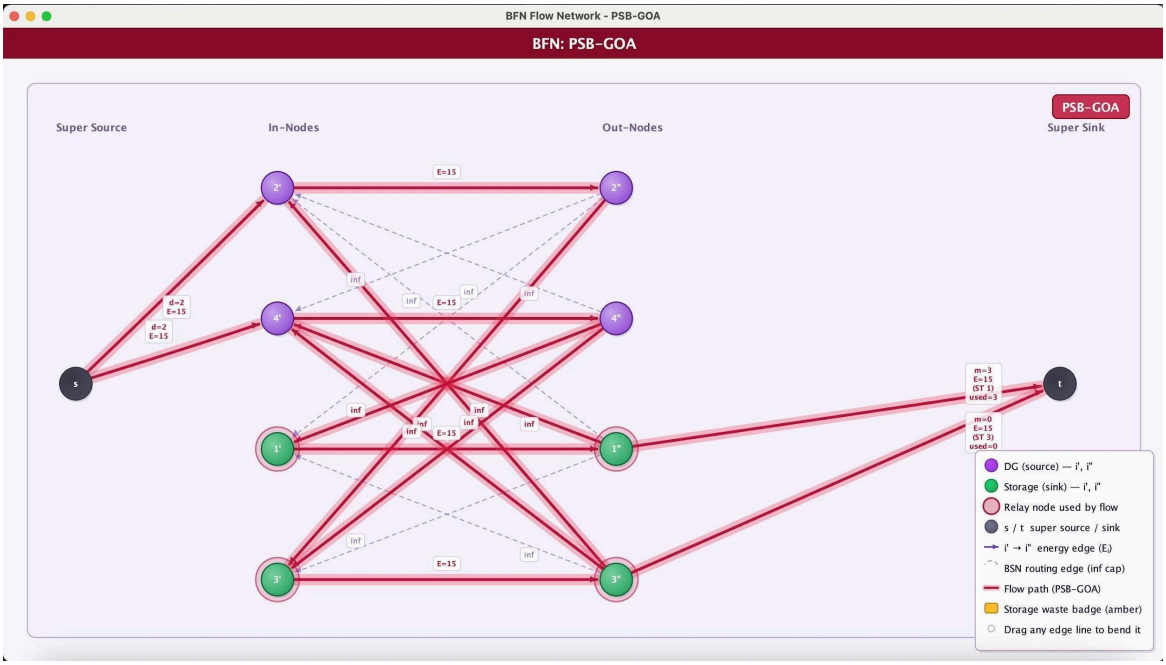


*Figure 11. CFN under PSB-GOA. Red edges show flow paths. The fine-grained scheduling interleaves packets from different DGs in locally optimal order.*

## Sequential Branch-and-Bound Solver

To check the heuristic algorithms against a stronger baseline, this project uses a dual branch-and-bound sequential solver. The solver tries all possible DG routing orders using depth-first search and cuts off paths that cannot lead to a better result. At each step, it chooses one DG to route next and calculates the total priority after that choice. If the best possible score from that branch is still lower than the best result already found, the branch is removed from the search.

A key part of the solver is that it runs two searches instead of one. This follows the algorithm portfolio idea from Gomes and Selman [9], where different methods are used together because one method may work better on some cases than another. In this project, one search uses standard BFS routing, and the other uses best-fit routing. Both searches share the same best result, so if either one finds a better solution, the other search can use that value to prune weaker branches. This setup is useful because BFS and best-fit can find different valid solutions on the same network. To speed up the search, the solver starts with the best result from GOA, Density GOA, and Hybrid GOA, then explores later choices in density order. The solver is capped at 15 DG nodes to keep the runtime manageable.

The solver uses a safe upper bound to decide when a search branch can be skipped. It adds up $v_i \times$ remaining[i] for every DG that has not been routed yet, assuming that all remaining packets could still reach storage. This is optimistic, but it is safe because it will not remove a branch that might still lead to the best answer. A tighter bound based on fractional knapsack was considered, since it could account for shared storage more closely. However, the simpler bound worked well for the tested network sizes because the solver already starts with a strong result from the warm start.

Forward checking helps the solver skip branches that have no useful path left. Before moving deeper into the search, the solver checks whether any unrouted DG can still reach a storage node with enough remaining space through a valid augmenting path. If no DG can reach storage, that branch cannot improve the solution, so the solver stops exploring it. This check requires running BFS for each unrouted DG, but it saves time by avoiding branches that would not produce any additional routed data.

A main limitation of the dual B&B solver is that it routes one DG completely before moving to the next. The ILP solver is more versatile because it can mix packets from different DGs across different paths at the same time. Some solutions found by the ILP cannot be matched by any fixed DG order. For that reason, the dual B&B solver is not always globally optimal for MWF-S. It is only exact within the set of ordered routing strategies.

### MaxFlow_Greedy (Algorithm 1)

In addition to the heuristics developed for this project, this extension also includes two baseline algorithms for the MWF-S setting. Algorithm 1 sorts the DG nodes from smallest packet size to largest, then uses Modified Edmonds-Karp to send as much flow as possible from each DG. This helps preserve more packets since smaller packets take up less storage space. Under the uniform energy model, every packet uses 1 unit of energy at each node on its path, no matter how large the packet is. Packet size only changes how much storage is used at the destination. For this reason, Algorithm 1 mainly tries to maximize the number of preserved packets, then reports the total priority from those packets.

### MaxWeightedFlow_Approximation (Algorithm 2)

Algorithm 2 is a 2-approximation method for the all-or-nothing version of MWF-S. In this model, a DG must either send all of its d_i packets or send none of them. The algorithm tests two greedy orders. The first order sorts DGs by priority v_i from highest to lowest, while the second sorts them by value density v_i / sz_i from highest to lowest. If a DG cannot send all of its packets in one of these orders, that order stops right away. The algorithm then returns the order with the higher total priority. This means Algorithm 2 may return zero priority in a heavily bottlenecked network if no DG can fully send its packets. That result comes from the all-or-nothing rule, not from an implementation error.

### ILP Baselines: MaxWeightedFlow_ILP and MaxFlow_ILP

ILP Baselines. Two ILP formulations are used, both sharing constraints (2)-(6). MaxWeightedFlow_ILP uses objective (7): maximize total preserved priority by weighting each flow by v_i. MaxFlow_ILP uses objective (1): maximize total preserved packets without weighting by priority. The edge capacity constraint does not multiply by sz_i; packet size affects only the sink storage constraint. Both ILPs are exported as CPLEX-format .lp files and solved externally using GLPK. Algorithm 1 should match or be close to MaxFlow_ILP on packet count, while MaxWeightedFlow_ILP provides the true priority-optimal baseline.

## Software Architecture

The code is organized into three main parts. The first part handles the flow network itself. FlowNetwork.java keeps track of the BSN state and performs the main operations, such as building the CFN, running BFS, adding flow, and saving or restoring the network state. AugmentationEngine.java contains shared logic used across multiple algorithms, including best-fit routing and PSB scoring

The second part contains the algorithm implementations. Each algorithm is placed in its own class, including GOA.java, DensityGOA.java, HybridGOA.java, DDRGOA.java, DDRPlusGOA.java, PSBGOA.java, and ExactSolverNew.java. All of the algorithms follow the same structure, using a detailed run method for visual demonstrations and a quieter runSilent method for large scaling experiments.

The third part handles experiments, output, and visualization. SimulationRunner.java runs the experiment loop by creating random BSN networks, assigning DG and storage roles, running each algorithm on the same network setup, collecting results, checking for constraint violations, and reporting win/loss/tie counts. TraceLogger.java records packet-level relay paths to help debug routing behavior. Main.java reads the user's input and passes it to SimulationRunner.java. The graph views are handled by SensorNetworkGraph.java for the physical BSN and BFNGraph.java for the flow network. AlgorithmResult.java stores each algorithm's final priority and flow edges, while ILPSolver.java generates the LP file for the MWF-S integer program.

A key part of the code design is the snapshot and restore approach. FlowNetwork uses snapCap() and snapFlow() to save full copies of the capacity and flow matrices, then uses restoreCap() and restoreFlow() to return the network to an earlier state. This is important for algorithms like Hybrid GOA, PSB-GOA, and the exact solver because they often need to test a routing choice before deciding whether to keep it. The remaining packet counts and sink capacities are copied with regular array cloning. This approach uses more memory, but it keeps the backtracking logic clear and reduces the chance of mistakes compared to manually undoing every flow update.

# CHAPTER 5

# DATA SETS AND EXPERIMENTAL SETUP

## Network Topology Generation

All experimental networks are created by randomly placing sensor nodes inside a rectangular field. The field size can change depending on the experiment, from smaller 100 × 100 meter networks to much larger 2000 × 2000 meter networks. This matters because the size of the field changes how close the nodes are to each other. If the same number of nodes is placed in a larger field, the network becomes more spread out, so there are fewer direct communication links. In a smaller field, the nodes are closer together, which makes it easier for more nodes to connect directly.

Node placement uses Gaussian clustering. For each network, 1 to 3 cluster centers are randomly placed in the field. About 70% of the nodes are placed near those centers, while the remaining 30% are spread randomly as outliers. This creates a mix of tighter node groups and more spread-out areas. It also avoids making the network look too evenly spaced or too clustered.

Two nodes are connected if they are within the transmission range (TR). In each trial, the TR is jittered by plus or minus 25%, which creates variation in network density from one experiment to another. This helps model changes in wireless communication caused by environmental conditions or obstacles. The program keeps regenerating the network until BFS confirms that the graph is fully connected, so every node can participate in routing.

For each generated topology, the nodes are randomly given roles as either data generators or storage nodes. The DG-to-storage ratio is selected randomly for each trial, usually between 20% and 70%. This creates a mix of different network conditions and test cases. Some networks have many DGs competing for a smaller amount of storage, while others have more storage available but are still limited by relay energy. Any node that is not assigned as a DG or storage node becomes a relay node. These relay nodes do not create or store data, but they help move packets from DGs to storage nodes across the network.

Parameter Ranges

| Parameter | Description | Typical Range |
|---|---|---|
| Field size | Physical dimensions (meters) | 100×100 to 2000×2000 |
| Transmission range | Max link distance (meters) | 50 to 70 |
| Node energy (E_i) | Energy budget per node | 20 to 50 |
| Overflow packets (d_i) | Packets per DG | 4 to 10 |
| Packet size (sz_i) | Storage units per packet | 1 to 8 |
| Storage capacity (m_j) | Raw storage units per node | 4 to 16 |
| Packet priority (v_i) | Weight per packet | 1 to 50 |
| Network sizes | Nodes per topology | 6, 8, 10, 20, 30 |
| Trials per size | Independent random trials | 20 to 30 |

*Table 2. Experimental Parameter Ranges.*

## Trial Design

Each trial varies three factors: node placement, DG/storage split ratio, and transmission range jitter. This lets the algorithms be tested across different network conditions, such as tight clusters, sparse layouts, and mixed topologies with different levels of connectivity. By randomizing these factors together, I avoid drawing conclusions that only work for one narrow network setup.

For each trial, all algorithms are tested on identical copies of the network using the snapshot and restore functionality in FlowNetwork.java. This is important because routing changes the residual graph as packets move through the network. Without restoring the original state, later algorithms would run under different conditions. The snapshot/restore process saves the capacity matrix, flow matrix, remaining packet counts, and sink capacities before each run, then restores them afterward. This allows every algorithm to start from the exact same network state.

The exact solver using branch and bound is used as the optimal baseline when the problem size is manageable, typically up to 15 DG nodes. After that point, the search space grows too quickly, making exact solving too slow for practical experiments. Trials with no real bottleneck are skipped because they do not help compare algorithm quality in a meaningful way. In those cases, all data can already be preserved, so every algorithm reaches the same result. The simulation reports only active trials, where storage or energy limits actually matter, along with each algorithm’s percentage of the optimal value achieved.

# CHAPTER 6

# EVALUATION AND ANALYSIS

This chapter presents the experimental results of all algorithms across multiple network sizes and analyzes their relative performance. For small networks where the ILP solver is tractable, percentages are relative to the GLPK ILP optimal. For larger networks, percentages are relative to the dual branch-and-bound sequential solver. The baseline used is noted in each table.

## Small Network Results (10 Nodes)

This project evaluates performance on 10-node networks with 10 trials. The parameter settings are: field size 150 by 150, base transmission range 55, energy 25, overflow packets 5 to 10, packet size 1 to 6, storage capacity 8 to 15, and priority 1 to 30. Of the 10 trials, 8 produced a storage bottleneck requiring the MWF-S algorithms; 2 trials were skipped because all packets could be offloaded trivially.

| Algorithm | Avg Priority | % of Optimal |
|---|---|---|
| ILP Exact | 353.50 | 100.0% |
| GOA | 342.63 | 96.9% |
| Density GOA | 348.38 | 98.6% |
| Approx GOA | 351.75 | 99.5% |
| Hybrid GOA | 353.50 | 100.0% |
| DDR-GOA | 349.13 | 98.8% |
| DDR+-GOA | 347.75 | 98.4% |
| PSB-GOA | 353.50 | 100.0% |

*Table 3. Small Network Results: 10 Nodes, 8 Active Trials.*

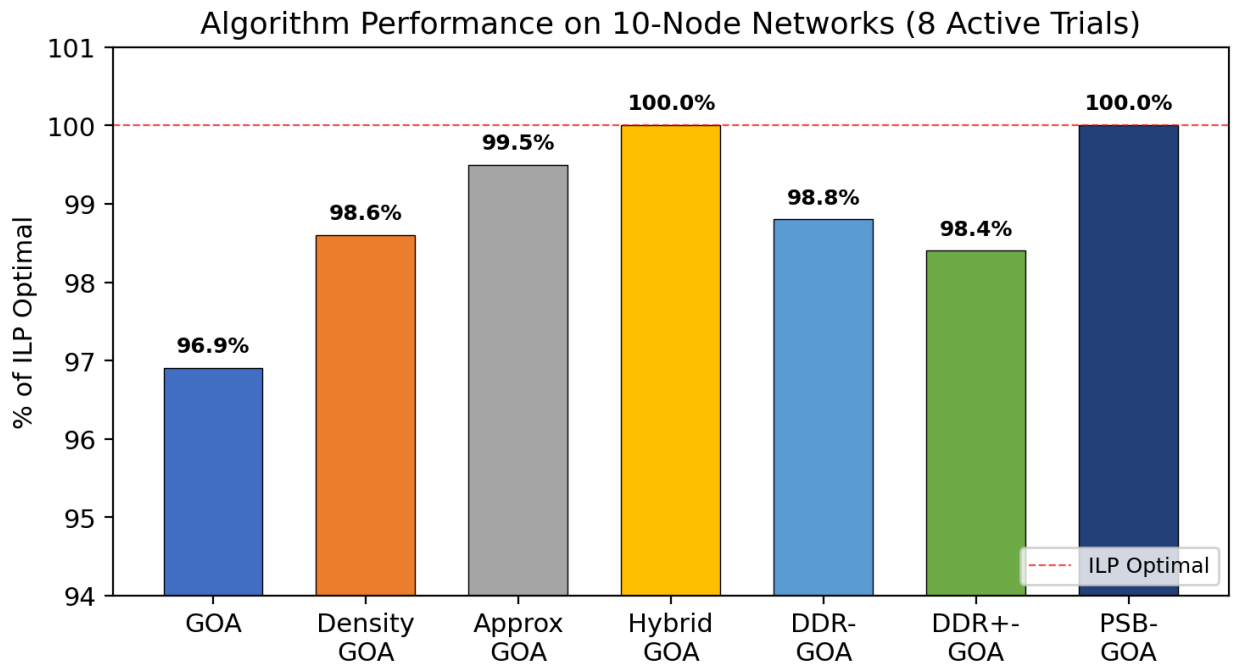


*Figure 12. Algorithm performance on 10-node networks as percentage of ILP optimal across 8 active trials. Hybrid GOA and PSB-GOA achieve 100% of optimal.*

Several observations stand out from these results. GOA reaches 96.9% of optimal, showing that packet size has a real impact on the problem. Density GOA improves the result by considering storage efficiency, reaching 98.6%. Hybrid GOA and PSB-GOA both reach 100.0% across all 8 active trials, showing that prefix enumeration and per-step scoring work very well.

DDR-GOA reaches 98.8%, showing that dynamic reordering helps, even though it is not always optimal. DDR+-GOA reaches 98.4%, slightly lower than DDR-GOA for this parameter set. This suggests that the square-root contention adjustment overcorrected in some trials, so the best dampening factor depends on the network.

The head-to-head results support this trend. Density beat GOA in 3 of 8 trials, GOA beat Density once, and they tied 4 times. Hybrid beat Approx once and tied it 7 times. DDR+ beat DDR once, DDR beat DDR+ twice, and they tied 5 times. PSB-GOA tied with the best result in all 8 trials and never fell behind.

One important pattern is how connectivity affects performance. In denser trials with more routing options, such as Trial 5 with TR=56, 5 DGs, and 5 storage nodes, all algorithms reached the same optimal value of 362. In

that case, the ordering did not matter much. In tighter trials, the difference was larger. For example, in Trial 1 with TR=61, 7 DGs, and 3 storage nodes, Density GOA reached 425 while GOA only reached 365. Since storage was very limited, 38 storage units compared to 185 units of demand, size-aware ordering became much more important.

## Visual Run Walkthrough

The visual run shows how the algorithms behave on one 10-node network. This instance has 6 DG nodes and 4 storage nodes, with only 42 storage units available for 127 units of demand. GOA starts with the highest-priority DG, DG 7 with v=28. It routes 5 packets of size 5 to storage nodes 2, 4, 5, and 9, using 25 storage units. Next, DG 6 with v=7 and sz=1 sends 8 packets, using 8 more units. Finally, DG 0 with v=3 and sz=2 sends 4 packets and uses the remaining storage across nodes 2, 4, and 5. The total preserved priority is 208.

Density GOA changes the order by value density: DG 8, DG 1, DG 6, DG 7, DG 0, and DG 3. However, DG 8 and DG 1 cannot reach any storage node in this topology, so they do not send packets. The algorithm then moves to DG 6, which sends 8 size-1 packets to storage 9. After that, DG 7 uses 20 storage units across nodes 2, 4, and 5, and DG 0 fills the remaining 14 units. The total preserved priority is 189. This is lower because DG 6 uses storage 9 before DG 7 can, forcing DG 7 onto less favorable routes and extra relay energy.

This walkthrough shows an important point: GOA and Density GOA each work better in different situations. In this network, GOA matches the optimal result because DG 7's larger packets fit well into the available storage. Hybrid GOA and PSB-GOA also reach the same optimal value of 208 because they test multiple ordering choices instead of relying on only one fixed rule.

## Scaling Results: 15-Node Networks

On 15-node networks with 5 trials (all active), the performance hierarchy sharpens.

| Algorithm | Avg Priority | % of B&B Optimal |
|---|---|---|
| Sequential B&B | 549.20 | 100.0% |
| GOA | 525.20 | 95.6% |
| Density GOA | 507.40 | 92.4% |
| Approx GOA | 525.20 | 95.6% |
| Hybrid GOA | 549.20 | 100.0% |
| DDR-GOA | 519.20 | 94.5% |
| DDR+-GOA | 514.20 | 93.6% |
| PSB-GOA | 549.20 | 100.0% |

*Table 4. 15-Node Network Results: 5 Active Trials.*

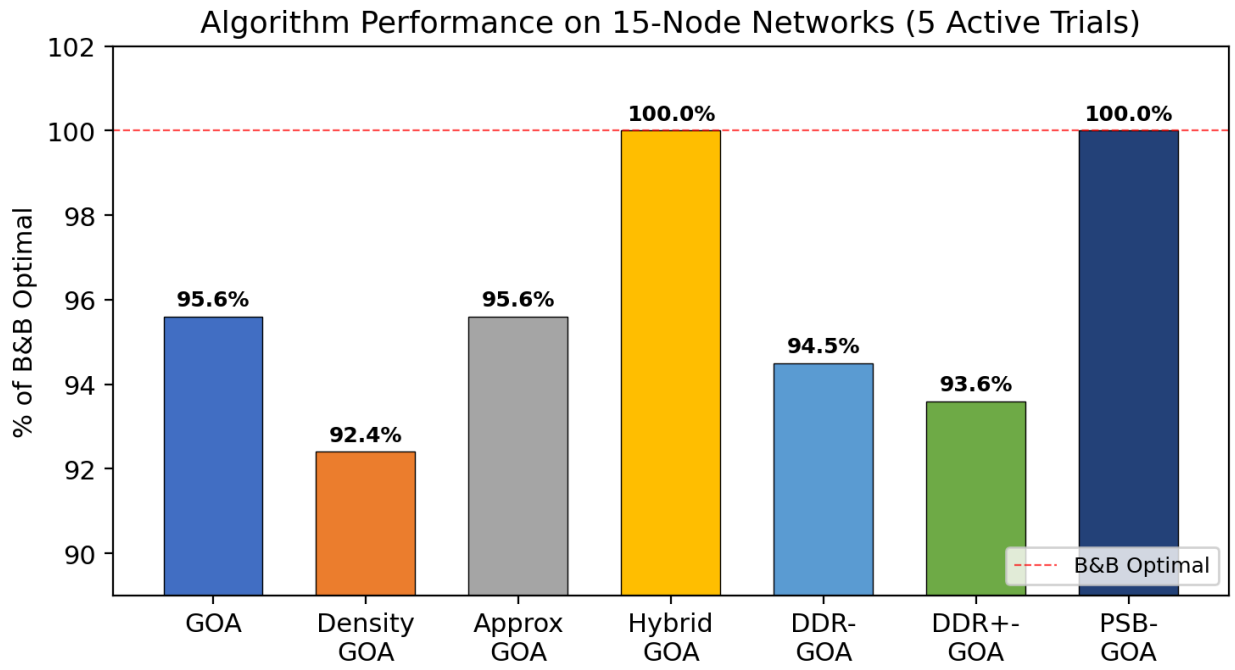


*Figure 13. Algorithm performance on 15-node networks as percentage of B&B optimal across 5 active trials. Density GOA drops to 92.4% while Hybrid GOA and PSB-GOA maintain 100%.*

At 15 nodes, the results shift. GOA outperforms Density GOA, reaching 95.6% compared to 92.4% and beating it in 4 of 5 trials. This happens because the larger network gives GOA more chances to route high-priority DGs to nearby storage nodes efficiently. Density ordering can sometimes choose a smaller high-density DG first, using storage and relay energy that higher-value DGs need later. Hybrid GOA and PSB-GOA again reach 100.0% of the B&B optimal across all 5 trials, showing that they remain strong at this scale.

## Multi-Experiment Summary

To evaluate algorithm robustness across different problem domains, I conducted five controlled experiments varying packet size range, storage capacity, and network scale.

| Experiment | Pkt Size | Storage | GOA | Density | Hybrid | PSB |
|---|---|---|---|---|---|---|
| 1: Small pkts | 1–2 | 4–8 | 96.3% | 100.0% | 100.0% | 100.0% |
| 2: Large pkts | 4–8 | 8–15 | 98.0% | 97.5% | 100.0% | 100.0% |
| 3: Wide range | 1–8 | 8–15 | 94.5% | 98.3% | 99.7% | 99.7% |
| 4: Abundant | 1–6 | 15–25 | 84.9% | 95.5% | 100.0% | 100.0% |

| | | | | | | |
|---|---|---|---|---|---|---|
| 5: 15 nodes | 1–8 | 6–12 | 97.0% | 93.9% | 100.0% | 100.0% |

*Table 5. Multi-Experiment Summary Across Parameter Regimes.*

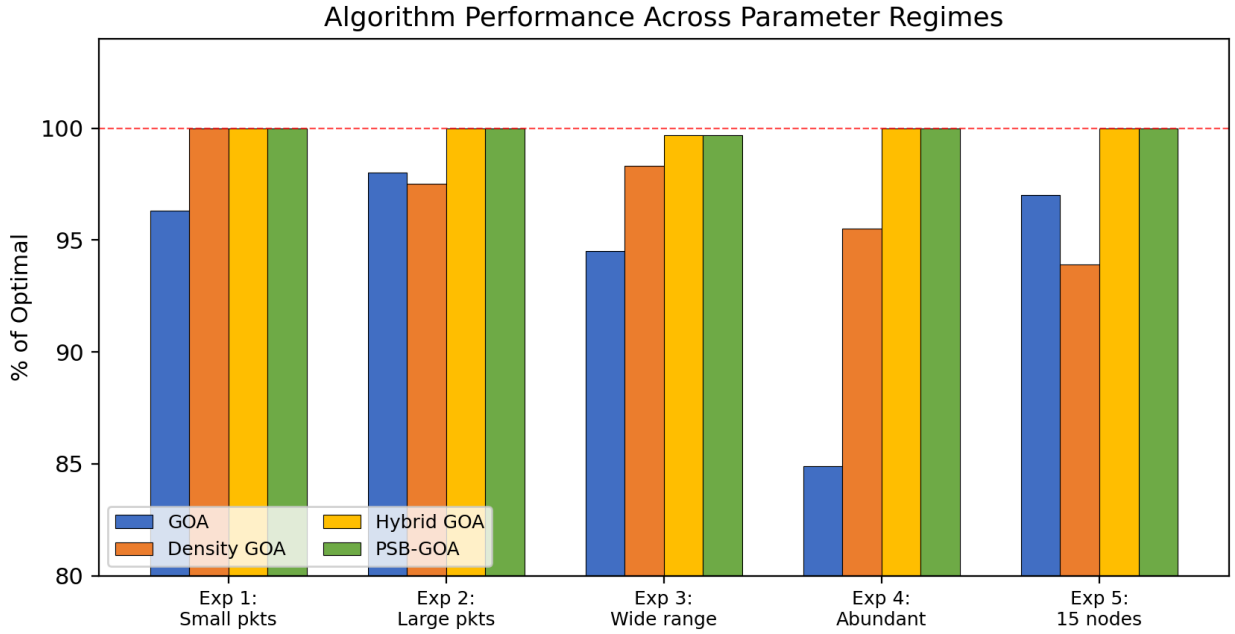


*Figure 14. Algorithm performance across five experimental parameter regimes. GOA varies from 84.9% to 98.0%, while Hybrid GOA and PSB-GOA remain at or near 100% in every configuration.*

Several patterns stand out. Hybrid GOA and PSB-GOA stay near optimal in every experiment, ranging from 99.7% to 100%. GOA is less consistent, ranging from 84.9% in Experiment 4 to 98.0% in Experiment 2. Density GOA does better when packet sizes are small and more uniform, but it falls behind GOA on larger networks. This shows that neither priority ordering nor density ordering is best in every setting.

Experiment 4 is especially interesting because storage was generous, with 15 to 25 units per node. Even so, 7 of 10 trials had no bottleneck and were skipped. The 3 active trials had 8 DGs competing for only 2 storage nodes, which created a difficult packing problem. GOA dropped to 84.9%, and DDR+-GOA fell to 74.9% on the visual run. This shows that total storage alone is not enough. The balance between DGs and storage nodes also matters.

The DDR+ and DDR comparison shows that performance depends on the specific network. On 8-node networks, DDR+ beats DDR in 5 of 16 active trials. DDR beats DDR+ in 2 trials, and they tie in 9 trials. DDR+ helps when storage is heavily contested between DGs. In some cases, the square-root dampening overcorrects because the estimated contention does not match the actual routing constraints.

| Comparison | DDR+ Wins | DDR Wins | Ties |
|---|---|---|---|
| 8-node (16 trials) | 5 | 2 | 9 |
| 10-node (8 trials) | 1 | 2 | 5 |
| 15-node (5 trials) | 0 | 3 | 2 |

*Table 6. Win/Loss/Tie Counts: DDR+ vs. DDR.*

## ILP Verification and B&B Limitation

ILP verification shows a major gap between the B&B solver and the true optimum. In the 10-node visual run, the B&B solver reports 208, while GLPK finds an ILP optimum of 379. This is an 82.2% gap. In the 15-node case, B&B reports 389, while GLPK reports 504, giving a 29.6% gap. The LP relaxation also equals 504, so there is no integrality gap for that instance.

The discrepancy comes from a limitation in how the B&B solver routes flows. It tests different DG orderings, but within each ordering, it fully routes one DG before moving to the next. The ILP does not have this restriction and can route flows from different DGs across different paths at the same time. As a result, the gap changes from one instance to another but still appears across different network sizes.

ILP verification shows that the B&B-to-ILP gap depends heavily on the instance. In Experiment 3's 10-node visual run, the gap is only 0.6% (B&B=338, ILP=340). In Experiment 5's 15-node visual run, the gap grows to 15.9% (B&B=655, ILP=759). The LP relaxation values, 362.5 and 768, show small integrality gaps of 6.2% and 1.2%. This means the ILP solutions are close to the LP bound. The main cause of the B&B-to-ILP gap is the sequential routing restriction, not integrality.

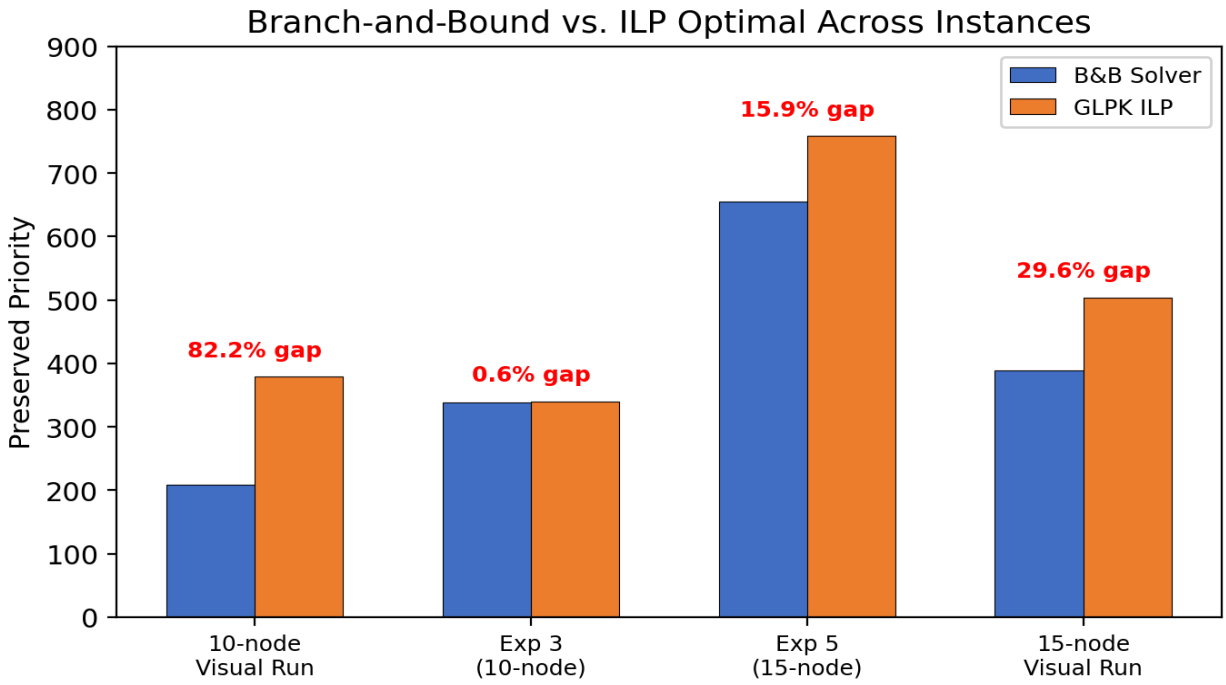


*Figure 15. Branch-and-bound solver versus GLPK ILP optimal across four instances. The sequential routing limitation of the B&B solver produces gaps ranging from 0.6% to 82.2%.*

This matters for three reasons. First, the dual B&B solver is not truly globally optimal for MWF-S. It is exact only for sequential routing strategies. Second, the heuristic percentages in this chapter likely overstate how close the algorithms are to the true optimum. Third, reaching the ILP optimum requires routing methods that can interleave packets from multiple DGs across shared paths.

Algorithm Comparison and Discussion

| Algorithm | Time Complexity | Strengths | Weaknesses |
|---|---|---|---|
| GOA | $O(k \cdot V \cdot E^2)$ | Simplest; optimal for MWF-U | Fragments storage |
| Density GOA | $O(k \cdot V \cdot E^2)$ | Good storage efficiency | Not always optimal |
| Approx GOA | $O(k \cdot V \cdot E^2)$ | Guaranteed 1/2 bound | Extra constant factor |
| Hybrid GOA | $O(k^4 \cdot V \cdot E^2)$ | Near-optimal | Expensive; k cap |
| DDR-GOA | $O(k^2 \cdot V \cdot E^2)$ | Adaptive | Competition-blind |
| DDR+-GOA | $O(k^2 \cdot V \cdot E^2)$ | Contention-aware | Can over-correct |
| PSB-GOA | $O(k^4 \cdot V \cdot E^2)$ | Near-optimal | Expensive; k cap |
| Exact B&B | $O(k! \cdot V \cdot E^2)$ | Best sequential solution | Not true ILP optimal |

*Table 7. Algorithm Complexity Comparison.*

Across both network sizes, the algorithms form a clear performance hierarchy. Hybrid GOA and PSB-GOA are co-leaders, achieving 100% of the B&B optimal on every trial through fundamentally different mechanisms: global prefix enumeration versus local per-step scoring. Approx GOA provides a reliable 95.6% to 99.5% baseline with its theoretical half-optimal guarantee. GOA and Density GOA trade places depending on the parameter regime, with neither dominating the other. DDR-GOA and DDR+-GOA occupy the 92% to 99% range with inconsistent relative ordering across network sizes. The main trade-off is between solution quality and computational cost: the near-optimal algorithms (Hybrid GOA and PSB-GOA) require $O(k^4)$ factor more work than the simpler heuristics, making them impractical when the number of DGs exceeds 12.

Research Paper Algorithm Comparison: MaxFlow_Greedy and MaxWeightedFlow_Approximation

I compare MaxFlow_Greedy, MaxWeightedFlow_Approximation, and the ILP optimal baselines on a fixed 20-node BSN with uniform energy and varying packet sizes. The network uses a 100 × 100 area with transmission range 30. It includes 5 DGs, 5 storage nodes, and 10 transshipment nodes. Each DG has 100 overflow packets, with packet sizes randomly chosen from [1, 10]. Storage capacities are chosen from [50, 150], node energy is fixed at 100, and DG priorities are chosen from [1, 100]. In the uniform energy model, sending, relaying, or receiving one packet each costs 1 energy unit, regardless of packet size.

For this representative trial, the total storage demand was 3,200 storage units while the available storage capacity was 420 units (13.1% capacity-to-demand ratio), making this a meaningful bottleneck instance.

| Algorithm | Preserved Priority | Preserved Packets | Storage Used | Runtime (ms) | Gap vs ILP |
|---|---|---|---|---|---|
| MaxFlow_Greedy | 13,016 | 212 | 408 | 1 | 0.5% |
| MaxWeightedFlow_Approx | 12,200 | 200 | 300 | 0 | 6.8% |
| MaxWeightedFlow_ILP | 13,084 | 213 | 417 | 44 | 0.0% |
| MaxFlow_ILP | 13,046 | 213 | 419 | 24 | 0.3% |

*Table 8. Comparison of MaxFlow_Greedy and MaxWeightedFlow_Approximation on a 20-Node BSN with Varying Packet Sizes (Uniform Energy Model, Energy = 100). MaxFlow_Greedy sorts by packet size ascending. MaxWeightedFlow_Approximation uses all-or-nothing density ordering. MaxWeightedFlow_ILP maximizes total priority. MaxFlow_ILP maximizes total packets.*

MaxFlow_Greedy preserved 212 packets with a priority of 13,016 and 408 storage units used. This is very close to the MaxWeightedFlow_ILP value of 13,084, with only a 0.5% gap. It also nearly matches the MaxFlow_ILP packet optimum of 213 packets. This shows that MaxFlow_Greedy is near-optimal for maximizing packet flow. The earlier packet-count mismatch came from comparing different ILP objectives: priority versus packet count.

MaxWeightedFlow_Approximation preserved 200 packets with a priority of 12,200 and 300 storage units used. This is 6.8% below the MaxWeightedFlow_ILP value. With node energy raised to 100, the approximation no longer returns zero. Its density ordering preserved one full DG, then stopped when DG 3 could only send 12 of 100 packets, triggering the all-or-nothing stopping rule.

Two GLPK baselines were solved. MaxWeightedFlow_ILP maximized priority and preserved 13,084 priority with 213 packets and 417 storage units used. MaxFlow_ILP maximized packet count and preserved 213 packets with 13,046 priority and 419 storage units used. GLPK confirmed both as integer-optimal. MaxFlow_Greedy nearly matched MaxFlow_ILP on packets, 212 versus 213, showing it is near-optimal for max-flow.

## Extended Stress Tests on 100-Node Networks

To study how MaxFlow_Greedy and MaxWeightedFlow_Approximation perform across different storage-to-demand domains, I ran four stress tests on 100-node networks. Each test used a 2000 × 2000 area, transmission range 250, energy level 100 per node, packet sizes from [1, 10], and priorities from [1, 100]. The tests changed the number of DG nodes, storage nodes, and packets per DG to create different storage bottlenecks. All results were compared against both GLPK ILP baselines.

### Stress Test 1: Severe Bottleneck (5.8% Capacity-to-Demand Ratio)

Network configuration: 20 DG nodes, 20 storage nodes, 100 packets per DG. Total storage demand: approximately 27,500 units. Total storage capacity: 1,600 units (5.8% ratio). This test simulates a deployment where available storage is drastically insufficient.

| Algorithm | Preserved Priority | Preserved Packets | Gap vs. MaxWeightedFlow_ILP |
|---|---|---|---|
| MaxFlow_Greedy | 16,645 | 389 | 27.5% |
| MaxWeightedFlow_Approx | 20,700 | 300 | 9.8% |
| MaxWeightedFlow_ILP | 22,946 | 322 | 0.0% (baseline) |
| MaxFlow_ILP | 16,835 | 402 | N/A (different objective) |

*Table 9. Extended Stress Test 1 Results: Severe Bottleneck (100-node network, 2000x2000 area, TR=250, E=100, sz in [1,10]).*

Under severe storage scarcity, MaxWeightedFlow_Approximation performs better on priority than MaxFlow_Greedy, 20,700 versus 16,645. This is a 24.4% advantage and comes within 9.8% of the ILP optimum. MaxFlow_Greedy preserves more packets, 389 versus 300, but with lower total priority. This shows why density-based ordering can work better when only a small portion of packets can be saved.

Stress Test 2: Medium Bottleneck (37.1% Capacity-to-Demand Ratio)

Network configuration: 20 DG nodes, 20 storage nodes, 50 packets per DG. Total storage demand: approximately 13,750 units. Total storage capacity: 5,100 units (37.1% ratio). This test represents a moderate bottleneck where roughly one-third of all generated data can be preserved.

| Algorithm | Preserved Priority | Preserved Packets | Gap vs. MaxWeightedFlow_ILP |
|---|---|---|---|
| MaxFlow_Greedy | 25,680 | 561 | 25.2% |
| MaxWeightedFlow_Approx | 27,400 | 400 | 20.2% |
| MaxWeightedFlow_ILP | 34,336 | 490 | 0.0% (baseline) |
| MaxFlow_ILP | 28,789 | 598 | N/A (different objective) |

*Table 10. Extended Stress Test 2 Results: Medium Bottleneck (100-node network, 2000x2000 area, TR=250, E=100, sz in [1,10]). ILP runtime exceeded 20 seconds.*

At a 37.1% capacity-to-demand ratio, MaxWeightedFlow_Approximation again performs better on priority, 27,400 versus 25,680. Its ILP gap is 20.2%, compared to 25.2% for MaxFlow_Greedy. MaxFlow_Greedy still preserves more packets, 561 versus 400. Both algorithms remain below the MaxWeightedFlow_ILP value of 34,336, and GLPK took over 20 seconds to solve the instance. This shows why exact ILP methods become impractical at larger scales.

Stress Test 3: Storage-Scarce Topology (4.8% Capacity-to-Demand Ratio, 5 Storage Nodes)

Network configuration: 20 DG nodes, 5 storage nodes, 100 packets per DG. Total storage demand: approximately 27,500 units. Total storage capacity: 1,320 units (4.8% ratio). The extreme reduction in storage node count creates severe routing bottlenecks in addition to capacity constraints.

| Algorithm | Preserved Priority | Preserved Packets | Gap vs. MaxWeightedFlow_ILP |
|---|---|---|---|
| MaxFlow_Greedy | 9,480 | 305 | 52.6% |
| MaxWeightedFlow_Approx | 7,500 | 100 | 62.5% |
| MaxWeightedFlow_ILP | 20,012 | 299 | 0.0% (baseline) |
| MaxFlow_ILP | 17,150 | 389 | N/A (different objective) |

*Table 11. Extended Stress Test 3 Results: Storage-Scarce Topology (100-node network, 2000x2000 area, TR=250, E=100, 20 DGs, only 5 storage nodes).*

With only five storage nodes, MaxWeightedFlow_Approximation breaks down under its all-or-nothing rule. It preserves only 100 packets from one DG before stopping, leaving a 62.5% gap from the ILP optimum. MaxFlow_Greedy performs better here because it allows partial storage use, preserving 305 packets and reaching 9,480 priority compared to 7,500. This shows a major weakness of the all-or-nothing model when routing paths to storage are limited.

Stress Test 4: Storage-Abundant Configuration (113.4% Capacity-to-Demand Ratio)

Network configuration: 5 DG nodes, 20 storage nodes, 100 packets per DG. Total storage demand: approximately 2,750 units. Total storage capacity: 3,120 units (113.4% ratio). In this configuration, total storage capacity exceeds total demand, so a perfect algorithm should preserve all packets.

| Algorithm | Preserved Priority | Preserved Packets | Gap vs. MaxWeightedFlow_ILP |
|---|---|---|---|
| MaxFlow_Greedy | 25,900 | 500 | 0.0% |
| MaxWeightedFlow_Approx | 20,400 | 300 | 21.2% |
| MaxWeightedFlow_ILP | 25,900 | 500 | 0.0% (baseline) |
| MaxFlow_ILP | 25,900 | 500 | 0.0% |

*Table 12. Extended Stress Test 4 Results: Storage-Abundant Configuration (100-node network, 2000x2000 area, TR=250, E=100, 5 DGs, 20 storage nodes).*

When storage capacity is greater than total demand, MaxFlow_Greedy matches both ILP baselines exactly. It preserves all 500 packets with a priority of 25,900. MaxWeightedFlow_Approximation preserves only 300 packets, leaving a 21.2% priority gap, even though enough storage exists. Its all-or-nothing stopping rule ends the process too early after only three DGs fully fit. Across all four stress tests, the results show that no single algorithm always wins. The best choice depends on the storage-to-demand ratio and the number of storage nodes.

## Confidence Interval Analysis

The idea behind a confidence interval is straightforward. Rather than reporting a single average, I am providing a range and saying "the true average most likely falls somewhere in here." I used the formula CI = 1.96 × σ / √n, where σ is the standard deviation across my 20 trials and n = 20. This works the same way as Excel's CONFIDENCE(0.05, stdev, 20) function. The 1.96 is the z-score for a 95% confidence level, pulled from the standard normal distribution. Put straight, if I were to repeat this experiment many times over, the true mean would fall within my computed interval 95% of the time.

### 10-Node Network (20 Trials)

For my first CI experiment, I used a relatively small setup: a 10-node network with 3 DGs, 3 storage nodes, and 4 transshipment nodes. The field was 50 × 50 meters with a transmission range of 25 meters. Each DG had 5 overflow packets, with packet sizes between 1 and 3, priorities between 1 and 10, and storage capacities between 5

and 10. Energy was set to 10 per node. I ran all four algorithms (Algorithm 1, Algorithm 2, ILP_max_weight, and ILP_max_flow) on each of the 20 randomly generated networks.

| Algorithm | Mean | Std Dev | CI (±) | 95% CI Range | N |
|---|---|---|---|---|---|
| Algorithm 1 | 60.55 | 27.95 | 12.25 | [48.30, 72.80] | 20 |
| Algorithm 2 | 55.25 | 26.58 | 11.65 | [43.60, 66.90] | 20 |
| ILP_max_weight | 71.00 | 21.88 | 9.59 | [61.41, 80.59] | 20 |
| ILP_max_flow | 63.75 | 26.22 | 11.49 | [52.26, 75.24] | 20 |

*Table 13. Confidence Interval Results: 10-Node Network (20 Trials, 95% CI). ILP solutions verified as INTEGER OPTIMAL by GLPK.*

ILP_max_weight came in first with a mean of 71.00 ± 9.59, as it's solving for the best priority directly. Algorithm 1 averaged 60.55 ± 12.25 and stayed close to ILP_max_flow (63.75 ± 11.49). This lines up with what I saw before about it being a strong max-flow approximation. Algorithm 2 was the lowest at 55.25 ± 11.65 since its all-or-nothing rule stops a DG completely if it can't send every packet.

The CI ranges for Algorithm 1 and ILP_max_weight do share some common ground ([48.30, 72.80] vs [61.41, 80.59]), so the ILP does better but the difference isn't massive on a network this small. There was also a lot of variance between trials. Algorithm 2 got 0 on one network and 100 on another. This shows why running 20 trials instead of just one is important.

20-Node Network (20 Trials)

I then scaled up to a 20-node network with 5 DGs, 5 storage nodes, and 10 transshipment nodes. The field was 100 × 100 meters with a transmission range of 40 meters. Each DG had 10 overflow packets, with packet sizes between 1 and 5, priorities between 1 and 50, and storage capacities between 10 and 20. Energy was set to 20 per node. I ran all four algorithms (Algorithm 1, Algorithm 2, ILP_max_weight, and ILP_max_flow) on each of the 20 randomly generated networks.

| Algorithm | Mean | Std Dev | CI (±) | 95% CI Range | N |
|---|---|---|---|---|---|
| Algorithm 1 | 923.00 | 310.06 | 135.89 | [787.11, 1058.89] | 20 |

| | | | | | |
|---|---|---|---|---|---|
| Algorithm 2 | 845.50 | 303.77 | 133.13 | [712.37, 978.63] | 20 |
| ILP_max_weight | 1034.80 | 263.40 | 115.44 | [919.36, 1150.24] | 20 |
| ILP_max_flow | 961.10 | 271.19 | 118.85 | [842.25, 1079.95] | 20 |

*Table 14. Confidence Interval Results: 20-Node Network (20 Trials, 95% CI). Parameters: 100 × 100 field, TR = 40, 5 DGs × 10 packets, sizes [1,5], priorities [1,50], storage [10,20], energy = 20.*

At this scale, the gaps are more visible. ILP_max_weight averaged 1034.80 ± 115.44 with a range of [919.36, 1150.24], while Algorithm 1 averaged 923.00 ± 135.89 with a range of [787.11, 1058.89]. These ranges do not fully share common ground, so the difference is statistically significant. Algorithm 1 reaches about 89.2% of the ILP optimal on average, and Algorithm 2 sits at 81.7%. The high standard deviations (260 to 310) reflect the variability of random network generation, reinforcing why averaging over 20 trials with error bars is necessary for trustworthy results.

## Key Engineering Fixes

Three important fixes were made during development. The first involved a unit mismatch on the sink edge. Sink capacity in the CFN was stored as raw storage units, but the bottleneck check was treating it like a packet count. This mixed two different units and sometimes allowed too much flow to be added. I fixed this by leaving the sink edge out of the bottleneck minimum calculation and updating it separately by $\Delta \times sz_i$.

The second fix involved the dual B&B exact solver. The original solver used only one routing strategy, so it missed cases where the other strategy found a better result. Sometimes this made a heuristic look better than the "exact" solution, which clearly showed something was wrong. I fixed this by running two separate B&B searches, one with BFS routing and one with best-fit routing, while sharing the same global best value.

The third fix added a complexity cap for Hybrid GOA and PSB-GOA. Since both methods test many prefixes and rollouts, their runtime grows quickly as the number of DGs increases. This became too slow once the network had more than 12 DG nodes. To handle this, the code now switches to Density GOA when the DG count goes past that limit.

# CHAPTER 7

# CONCLUSION AND FUTURE WORK

This project extends Rivera and Tang's MWF-U framework to a setting where packets from different sensor nodes can have different sizes. I formulated the size-aware MWF-S problem and showed through counterexamples that GOA, which is optimal for MWF-U, is not always optimal for MWF-S. I also designed new algorithms for this harder setting, implemented the two paper-based algorithms under clearer names, MaxFlow_Greedy and MaxWeightedFlow_Approximation, and verified them against GLPK ILP baselines. Finally, I tested the algorithms on 100-node stress tests across different storage regimes.

Varying packet sizes changes the problem because each storage node now has to deal with packing. A node may have enough space for several small packets but not enough for one large packet. This also means GOA's priority-based ordering can fragment storage and miss better overall solutions. This packing behavior makes MWF-S similar to knapsack and bin packing problems, which suggests that an exact polynomial-time solution is unlikely.

The seven new heuristics approach MWF-S in different ways. Density GOA uses value density, similar to fractional knapsack. Approx GOA gives an approximation bound. Hybrid GOA combines prefix enumeration with best-fit storage selection. DDR-GOA and DDR+-GOA reorder DGs based on the remaining network state and storage contention. PSB-GOA uses per-step scoring to mix packets from different DGs more effectively. The dual B&B solver gives a strong sequential-routing baseline, while GLPK ILP results show that true optimal solutions can be higher when routing is allowed to interleave across DGs.

Extensive simulations on randomized BSN topologies show that Hybrid GOA and PSB-GOA reach 100% of the B&B optimum on the 10-node networks tested. Density GOA also performs well as a faster baseline, reaching 98.6%. ILP verification shows an 82.2% gap between B&B and the true ILP optimum on one instance, which confirms that sequential routing can limit the solution. All algorithms had zero constraint violations.

I also tested MaxFlow_Greedy and MaxWeightedFlow_Approximation on a 20-node BSN with varying packet sizes and uniform energy set to 100. Two GLPK baselines were used: MaxWeightedFlow_ILP for priority and MaxFlow_ILP for packet count. MaxFlow_Greedy reached 99.5% of the priority optimum, 13,016 vs. 13,084,

and nearly matched the packet optimum, 212 vs. 213. MaxWeightedFlow_Approximation preserved 12,200 priority with 200 packets, leaving a 6.8% gap.

Across all five configurations, MaxFlow_Greedy never exceeded MaxFlow_ILP on packets, and MaxWeightedFlow_Approximation never exceeded MaxWeightedFlow_ILP on priority. MaxFlow_Greedy usually preserved more packets, while MaxWeightedFlow_Approximation performed better under severe storage limits. The 100-node stress tests show that the storage-to-demand ratio is the main factor, and no single ordering strategy always wins.

Several directions for future work come from this project. The most important is developing a true exact solver for MWF-S that allows interleaved routing. A branch-and-bound or branch-and-price solver could branch on individual packet-to-path assignments instead of only DG orderings. Other directions include adaptive dampening for DDR+-GOA and sampled prefix enumeration for Hybrid GOA and PSB-GOA. Future work could also combine MWF-S with the MWF-H cost model and integrate it with game-theoretic data preservation for varying packet sizes. Another possible direction is extending MWF-S to cloud data center traffic and LLM scheduling in GPU-based data centers, as suggested by Rivera and Tang [1]. A small physical testbed using a linear or star topology would also help validate the model in practice.

# REFERENCES

[1] G. Rivera and B. Tang, "Priority-Based Data Preservation in Challenging Environments: A Maximum Weighted Flow Approach," California State University Dominguez Hills, 2024.

[2] L. R. Ford and D. R. Fulkerson, "Maximal Flow through a Network," Canadian Journal of Mathematics, vol. 8, pp. 399–404, 1956.

[3] J. Edmonds and R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," Journal of the ACM, vol. 19, no. 2, pp. 248–264, 1972.

[4] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Network Flows: Theory, Algorithms, and Applications. Prentice-Hall, Inc., 1993.

[5] H. Kellerer, U. Pferschy, and D. Pisinger, Knapsack Problems. Springer, 2004.

[6] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms," SIAM Journal on Computing, vol. 3, no. 4, pp. 299–325, 1974.

[7] D. P. Bertsekas, J. N. Tsitsiklis, and C. Wu, "Rollout Algorithms for Combinatorial Optimization," Journal of Heuristics, vol. 3, no. 3, pp. 245–262, 1997.

[8] D. Golovin and A. Krause, "Adaptive Submodularity: Theory and Applications in Active Learning and Stochastic Optimization," Journal of Artificial Intelligence Research, vol. 42, pp. 427–486, 2011.

[9] C. P. Gomes and B. Selman, "Algorithm Portfolios," Artificial Intelligence, vol. 126, no. 1–2, pp. 43–62, 2001.

[10] B. Tang, N. Yao, and B. Choi, "Maximizing Data Preservation in Intermittently Connected Sensor Networks," in Proc. of IEEE International Conference on Mobile Ad Hoc and Sensor Systems (MASS), 2012.

[11] G. Rivera and B. Tang, "Data Preservation in Intermittently Connected Sensor Networks with Data Priority," in Proc. of IEEE MASS, 2013.

[12] B. Tang and C. S. Raghavendra, "Truthful and Optimal Data Preservation in Base Station-less Sensor Networks: An Integrated Game Theory and Network Flow Approach," ACM Transactions on Sensor Networks, 2023.

[13] B. Tang, N. Jaggi, H. Wu, and R. Kurkal, "Energy-Efficient Data Redistribution in Sensor Networks," ACM Transactions on Sensor Networks, vol. 9, no. 2, 2013.

## APPENDIX A: GITHUB REPOSITORY

The complete source code, compilation instructions, and sample output for this project are available at the following GitHub repository:

https://github.com/jaewe2/FlowNetWork.git

The repository contains 19 Java source files arranged into four layers. The core flow network layer includes FlowNetwork.java and AugmentationEngine.java. The algorithm layer includes GOA.java, DensityGOA.java, HybridGOA.java, DDRGOA.java, DDRPlusGOA.java, PSBGOA.java, and ExactSolverNew.java. The experiment and I/O layer includes SimulationRunner.java, TraceLogger.java, AlgorithmResult.java, ILPSolver.java, and Main.java. The visualization layer includes SensorNetworkGraph.java, BFNGraph.java, GraphLauncher.java, and Axis.java. SensorStuff.java is kept as a legacy reference.

To compile and run the project, Java SE 11 or higher is required. No external libraries are needed because the visualizations use Java Swing from the standard JDK. Compile with javac *.java and run with java Main. The program asks for all parameters interactively. Setting the minimum and maximum value equal for any parameter creates a fixed-value case. The README provides three recommended test setups: a small clean run, a fragmentation pressure test, and a large-scale test with a complexity cap. ILP verification can be done by running glpsol --lp mwf_s_ilp.lp -o solution.txt on the generated LP file.

## APPENDIX B: LIVE DEMO RECORDING

A five-minute demo recording of the MWF-S project is available at the link below. The video shows the project structure in terminal and resulting graphs, how to choose between different modes, and how to use different commands to get the ILP output that we desire depending on heuristics or paper algorithms developed for research. A slight showing of the source code is shown as well.

Live Demo Recording: https://www.youtube.com/watch?v=uxrWap7Wn7w