GraphBrew: Adaptive Graph Reordering with Machine Learning

Per-Community Algorithm Selection using Perceptron-based ML

GraphBrew Team

January 2026

Contents

Problem Statement: Why Graph Reordering Matters

The Memory Wall Challenge

Graph algorithms (BFS, PageRank, SSSP, etc.) exhibit **random memory access patterns** that result in **poor cache utilization**.

Impact: Up to 70% of execution time is spent waiting for memory!

The Key Challenge

- Different graphs benefit from different reordering algorithms
- **Social networks** → community-based reordering (RabbitOrder, Leiden)
- **Road networks** → bandwidth reduction (RCM)
- **Web graphs** → hub-based reordering (HubSort, DBG)

No Single Algorithm Wins for All Graphs!

GraphBrew Overview

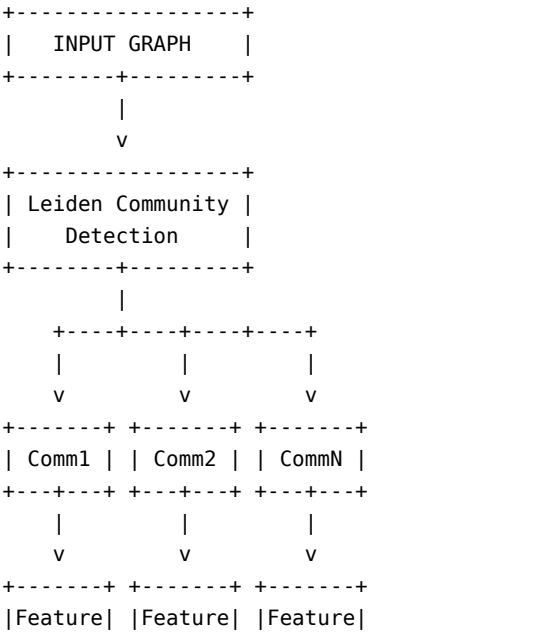
Architecture Flow

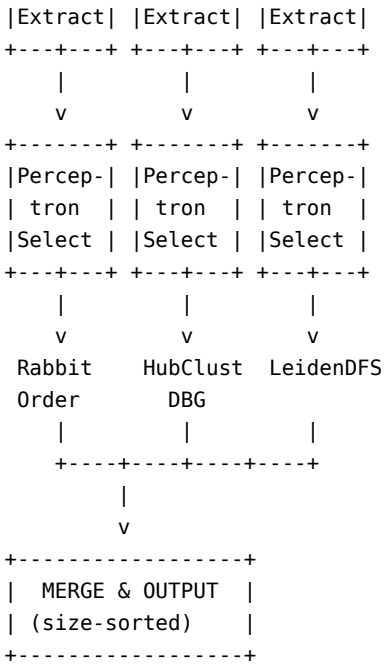
1. **Input Graph** –> Leiden Community Detection
2. **Per Community**: Extract features (density, hub concentration, etc.)
3. **Perceptron ML Selector**: Score each algorithm
4. **Apply Best Algorithm** to each community
5. **Merge** communities in size-sorted order

Key Innovation

Instead of applying one algorithm globally, GraphBrew **selects the best algorithm for each community** based on its structural characteristics.

GraphBrew Architecture Diagram





Feature Extraction: 12 Graph Features

Feature	Description	Normalization	Impact
modularity	Community structure strength	[0, 1]	High –> Leiden
density	Edge density	edges/(n*(n-1)/2)	Dense –> simple
degree_variance	CV of degree distribution	std/mean	High –> hub-based
hub_concentration	Top 10% edge share	[0, 1]	High –> HubSort
clustering_coeff	Triangle density	[0, 1]	High –> local
avg_path_length	Average shortest path	/10	High –> BFS-based
diameter	Graph diameter estimate	/50	Wide –> RCM
community_count	Sub-community count	log10	Many –> recursive
log_nodes	Graph size (nodes)	log10	Scaling
log_edges	Graph size (edges)	log10	Scaling
avg_degree	Average degree	/100	Connectivity
reorder_time	Algorithm runtime	negative	Time penalty

Perceptron Model: ML-Based Algorithm Selection

Score Computation (Per Algorithm)

```
score = bias
      + w_modularity * modularity
      + w_log_nodes * log10(nodes) * 0.1
      + w_log_edges * log10(edges) * 0.1
      + w_density * density
      + w_avg_degree * (avg_degree / 100)
      + w_degree_variance * degree_variance
      + w_hub_concentration * hub_concentration
      + w_clustering_coeff * clustering_coeff
```

```
      + w_avg_path_length * (avg_path_length / 10)
      + w_diameter * (diameter / 50)
      + w_community_count * log10(community_count)
      + w_reorder_time * reorder_time

final_score = score * benchmark_weights[benchmark]
```

Algorithm with highest score is selected!

Perceptron Scoring Diagram

```
INPUTS (Features)         WEIGHTS              OUTPUT
=================         =======              ======

modularity: 0.72  --*---> w_mod: 0.28 ---+
                                         |
density: 0.001    --*---> w_den: -0.15 --+
                                         |
degree_var: 2.1   --*---> w_dv: 0.18 ----+
                                         |
hub_conc: 0.45    --*---> w_hc: 0.22 ----+----> SUM ---> SCORE
                                         |      (+bias)
cluster_coef: 0.3 --*---> w_cc: 0.12 ----+
                                         |
avg_path: 5.2     --*---> w_ap: 0.08 ----+
                                         |
diameter: 16      --*---> w_di: 0.05 ----+
                                         |
...               --*---> ...       -----+


ALGORITHM SELECTION:
====================
RABBITORDER:    score = 2.31  <-- WINNER
LeidenDFS:      score = 2.18
HubClusterDBG:  score = 1.95
GORDER:         score = 1.82
ORIGINAL:       score = 0.50
```

What is a Perceptron?

Single-Layer Neural Network for Classification

A **perceptron** is the simplest form of a neural network - a linear classifier that computes a weighted sum of inputs.

Mathematical Formula

```
output = activation(sum(w_i * x_i) + bias)

Where:
  x_i = input features (modularity, density, etc.)
  w_i = learned weights (how important each feature is)
  bias = base score (algorithm's inherent quality)
```

Why Perceptron for GraphBrew?

1. **Interpretable**: Each weight tells us feature importance
2. **Fast**: O(n) computation where n = number of features
3. **Online Learning**: Can update weights incrementally
4. **No Overfitting**: Simple model generalizes well

Multi-Class Selection

We use **one perceptron per algorithm**. Each computes a score, and we pick the algorithm with the **highest score**.

Type System: Graph Clustering with Centroids

The Problem

Different graph families (social, road, web) need different weight configurations. Training one set of weights for all graphs leads to suboptimal results.

Solution: Cluster Graphs into Types

We use **k-means-like clustering** on graph features to group similar graphs.

What is a Centroid?

A **centroid** is the "center point" of a cluster - the average feature values of all graphs in that type.

```
Centroid for type_0 (social networks):
  modularity: 0.72      (high - strong communities)
  degree_variance: 2.1  (high - power-law degrees)
  hub_concentration: 0.45
  avg_degree: 15.3

Centroid for type_1 (road networks):
  modularity: 0.15      (low - mesh-like)
  degree_variance: 0.3  (low - uniform degrees)
  hub_concentration: 0.12
  avg_degree: 2.8
```

Centroid Matching Diagram

```
FEATURE SPACE (2D simplified view)
==================================

     degree_variance
          ^
          |
     3.0  |        * type_2 (power-law)
          |
     2.5  |
          |
     2.0  |  [X] <-- NEW GRAPH        * type_0 (social)
          |          features:             centroid
     1.5  |          mod=0.68
          |          dv=1.9
     1.0  |          hub=0.42
```

```
     |
0.5  |                    * type_1 (road)
     |                      centroid
0.0  +----------------------------------> modularity
     0.0  0.2  0.4  0.6  0.8  1.0
```

```
DISTANCE CALCULATION:
====================
dist(X, type_0) = sqrt((0.68-0.72)^2 + (1.9-2.1)^2 + ...) = 0.24
dist(X, type_1) = sqrt((0.68-0.15)^2 + (1.9-0.3)^2 + ...) = 1.82
dist(X, type_2) = sqrt((0.68-0.45)^2 + (1.9-2.8)^2 + ...) = 0.98

RESULT: type_0 (social) is closest --> Load type_0.json
```

How Type Matching Works

At Runtime: Find Nearest Centroid

When we see a new graph, we compute its features and find the **closest type** using Euclidean distance:

```python
def find_best_type(graph_features, type_registry):
    best_type = None
    min_distance = infinity

    for type_name, type_info in type_registry.items():
        centroid = type_info['centroid']

        # Euclidean distance in feature space
        distance = sqrt(
            (graph.modularity - centroid.modularity)^2 +
            (graph.degree_variance - centroid.degree_variance)^2 +
            (graph.hub_concentration - centroid.hub_concentration)^2 +
            (graph.avg_degree - centroid.avg_degree)^2
        )

        if distance < min_distance:
            min_distance = distance
            best_type = type_name

    return best_type  # e.g., "type_3"
```

Result

Load weights from `type_3.json` instead of generic weights!

Type Registry Structure

type_registry.json

```
{
  "type_0": {
    "description": "Dense social networks",
    "centroid": {
      "modularity": 0.72,
```

```
      "degree_variance": 2.1,
      "hub_concentration": 0.45,
      "avg_degree": 15.3,
      "density": 0.0012
    },
    "graphs": ["soc-LiveJournal1", "com-Youtube", "twitter"]
  },
  "type_1": {
    "description": "Road networks",
    "centroid": {
      "modularity": 0.15,
      "degree_variance": 0.3,
      "hub_concentration": 0.12,
      "avg_degree": 2.8,
      "density": 0.00001
    },
    "graphs": ["roadNet-CA", "roadNet-PA", "germany_osm"]
  },
  "type_2": {
    "description": "Web crawl graphs",
    "centroid": { ... }
  }
}
```

Generating Type Weights: The Training Loop

Phase 1: Initialize Types

```
# Create initial type clusters from known graphs
python3 graphbrew_experiment.py --init-weights
```

This creates: - `type_registry.json` with initial centroids - `type_0.json` through `type_N.json` with default weights

Phase 2: Fill with Heuristic Defaults

```
python3 graphbrew_experiment.py --fill-weights --graphs all
```

For each graph: 1. Run all 18 algorithms 2. Record speedups vs baseline 3. Update weights based on which algorithms work best

Phase 3: Incremental Training

```
python3 graphbrew_experiment.py --train-adaptive --graphs all
```

Online learning: after each benchmark, immediately update weights.

Weight Update: The Learning Process

When Algorithm Performs Well (speedup $>$ 1.0)

```
error = speedup - 1.0 - current_score  # positive

# Increase weights for features that predicted this success
w_modularity += learning_rate * error * modularity
w_hub_concentration += learning_rate * error * hub_concentration
# ... etc for all features
```

Effect: Algorithm gets higher scores for similar graphs in future.

When Algorithm Performs Poorly (speedup < 1.0)

```
error = speedup - 1.0 - current_score  # negative

# Decrease weights (error is negative)
w_modularity += learning_rate * error * modularity
```

Effect: Algorithm gets lower scores for similar graphs in future.

Key Insight

Weights encode: "When this feature is high, how well does this algorithm perform?"

From Training to Runtime Selection

Training Time (Python)

```
For each (graph, algorithm, benchmark):
    1. Run benchmark, get speedup
    2. Compute features
    3. Find graph's type (nearest centroid)
    4. Update weights in type_N.json
    5. Save immediately (incremental)
```

Runtime (C++ AdaptiveOrder)

```
// In builder.h - SelectBestReorderingForCommunity()

// 1. Compute community features
CommunityFeatures feat = ComputeCommunityFeatures(community);

// 2. Load weights for this graph type
auto weights = LoadPerceptronWeightsForFeatures(
    modularity, degree_variance, hub_concentration, ...);

// 3. Score each algorithm
for (algo : all_algorithms) {
    scores[algo] = weights[algo].bias
        + weights[algo].w_modularity * feat.modularity
        + weights[algo].w_density * feat.density
        + ...;
}

// 4. Select algorithm with highest score
return argmax(scores);
```

Complete Flow: Training to Inference

```
TRAINING PHASE (Python - graphbrew_experiment.py)
================================================

Graph Dataset
     |
```

```
     v
[Feature Extraction] --> modularity, density, hub_conc, ...
     |
     v
[Type Assignment] --> Find nearest centroid --> type_3
     |
     v
[Run Benchmarks] --> For each algorithm, get speedup
     |
     v
[Weight Update] --> Update type_3.json weights
     |
     v
[Save Weights] --> scripts/weights/type_3.json


INFERENCE PHASE (C++ - builder.h AdaptiveOrder)
================================================

Input Graph
     |
     v
[Leiden Community Detection] --> 847 communities
     |
     v
For each community:
     |
     v
[Feature Extraction] --> density=0.08, hub_conc=0.38, ...
     |
     v
[Load Type Weights] --> type_3.json (based on global features)
     |
     v
[Perceptron Scoring] --> RABBITORDER=2.1, LeidenDFS=1.9, ...
     |
     v
[Select Best] --> RABBITORDER
     |
     v
[Apply Reordering] --> Reorder this community with RABBITORDER
     |
     v
[Merge Communities] --> Final reordered graph
```

Supported Algorithms (18 Total)

ID	Algorithm	Best For
0	ORIGINAL	No reordering (baseline)
1	RANDOM	Random permutation
2	SORT	Degree-sorted ordering
3	HUBSORT	Hub vertices first
4	HUBCLUSTER	Hub clustering
5	DBG	Degree-based grouping

ID	Algorithm	Best For
6	HUBSORTDBG	HubSort + DBG hybrid
7	HUBCLUSTERDBG	HubCluster + DBG
8	RABBITORDER	Community-aware (social)
9	GORDER	Graph reordering
10	CORDER	Cache-optimized
11	RCM	Reverse Cuthill-McKee (road)
15	LeidenOrder	Modularity-based
16	LeidenDFS	Leiden + DFS traversal
17	LeidenDFSHub	Leiden + DFS + Hub sort
18	LeidenDFSSize	Leiden + size ordering
19	LeidenBFS	Leiden + BFS traversal
20	LeidenHybrid	Adaptive Leiden variant

Weight Training Process

Phase 1: Generate Reorderings

For each graph x algorithm:

1. Run converter to generate reordered graph
2. Record reorder_time
3. Save .lo mapping file

Phase 2: Run Benchmarks

For each graph x algorithm x benchmark (pr, bfs, cc, sssp, bc):

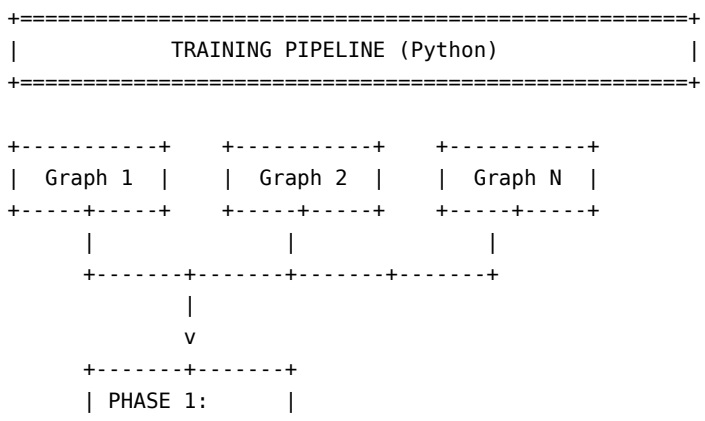
1. Run benchmark, record trial_time
2. Compute speedup = baseline_time / trial_time
3. Extract topology features from C++ output
4. **IMMEDIATELY update weights** (incremental learning)

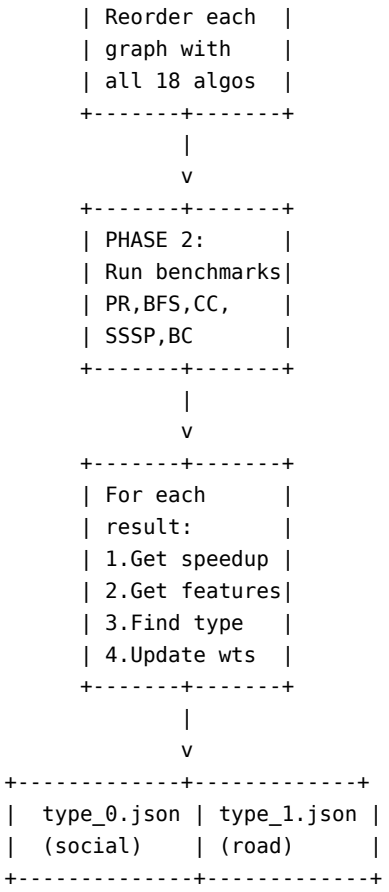
Phase 3: Cache Simulations (Optional)

- Simulate L1/L2/L3 cache behavior
- Update cache_l1_impact, cache_l2_impact, cache_dram_penalty

---

Training Pipeline Diagram





Gradient Update Rule: Online Learning

Error Signal Computation

```
error = (speedup - 1.0) - current_score
# Positive error: algorithm performed better than expected
# Negative error: algorithm performed worse than expected
```

Weight Update (Stochastic Gradient Descent)

```
learning_rate = 0.01

# Core feature weights
w_modularity      += lr * error * modularity
w_density         += lr * error * density
w_degree_variance += lr * error * degree_variance
w_hub_concentration += lr * error * hub_concentration

# Extended topology weights
w_avg_path_length += lr * error * (avg_path_length / 10)
w_diameter        += lr * error * (diameter / 50)
w_clustering_coeff += lr * error * clustering_coeff

# Cache impact weights (when simulation data available)
w_cache_l1_impact += lr * error * l1_hit_rate
w_cache_dram_penalty += lr * error * dram_penalty
```

Type-Based Weight System

Graph Type Clustering

The system uses auto-generated type files for specialized tuning:

- `scripts/weights/type_0.json` (Cluster 0 weights)
- `scripts/weights/type_1.json` (Cluster 1 weights)
- `scripts/weights/type_N.json` (Additional clusters)
- `scripts/weights/type_registry.json` (maps graph names → type + centroids)

Runtime Type Selection

1. Compute graph features (modularity, density, etc.)
2. Find nearest centroid using Euclidean distance
3. Load weights from corresponding type_N.json file
4. Fall back to defaults if no type file exists

Benefits

- **Specialized weights** for different graph families
- **Auto-clustering** discovers natural graph categories
- **Incremental updates** per type file

AdaptiveOrder Algorithm (C++)

Runtime Algorithm Selection

```
void GenerateAdaptiveMappingRecursive(...) {
    // Step 1: Leiden community detection
    auto communities = runLeiden(graph, resolution=0.75);

    // Step 2: For each community, compute features
    for (community : communities) {
        CommunityFeatures feat = ComputeCommunityFeatures(community);
        // Features: density, degree_variance, hub_concentration,
        //           clustering_coeff, avg_path_length, diameter

        // Step 3: Select best algorithm using perceptron
        ReorderingAlgo best = SelectBestReorderingForCommunity(feat);

        // Step 4: Apply selected algorithm to this community
        ApplyLocalReordering(community, best);
    }

    // Step 5: Combine communities in size-sorted order
    AssignGlobalIds(communities_sorted_by_size);
}
```

Feature Computation (C++)

ComputeCommunityFeatures Implementation

```
CommunityFeatures ComputeCommunityFeatures(
    const vector<NodeID>& comm_nodes, const Graph& g) {

    CommunityFeatures feat;

    // Basic features
    feat.num_nodes = comm_nodes.size();
    feat.num_edges = count_internal_edges(comm_nodes, g);
    feat.internal_density = 2.0 * feat.num_edges /
                            (feat.num_nodes * (feat.num_nodes - 1));

    // Degree statistics
    vector<int64_t> degrees = compute_internal_degrees(comm_nodes, g);
    feat.avg_degree = mean(degrees);
    feat.degree_variance = cv(degrees);  // coefficient of variation

    // Hub concentration: top 10% edge share
    sort(degrees, descending);
    feat.hub_concentration = sum(top_10_percent) / sum(all);

    // Clustering coefficient (sampled for efficiency)
    feat.clustering_coeff = sampled_triangle_ratio(samples=50);

    // Diameter & avg path (single BFS from hub node)
    auto [diameter, avg_path] = bfs_from_hub(comm_nodes, g);

    return feat;
}
```

Benchmark Workloads

Supported Graph Algorithms

Benchmark	Algorithm	Access Pattern
PR	PageRank	Pull-based iteration
BFS	Breadth-First Search	Level-synchronous
CC	Connected Components	Label propagation
SSSP	Single-Source Shortest Path	Delta-stepping
BC	Betweenness Centrality	Multiple BFS
TC	Triangle Counting	Intersection-based

Benchmark-Specific Weight Adjustments

```
benchmark_weights = {
    'pr': 1.2,    # PageRank benefits most from reordering
    'bfs': 0.9,   # BFS has more sequential access
    'cc': 1.0,    # Neutral
    'sssp': 1.1,  # Similar to BFS but iterative
    'bc': 0.8     # Multiple traversals, less locality gain
}
```

Example Weight File: type_0.json


```
{
  "RABBITORDER": {
    "bias": 0.35,
    "w_modularity": 0.28,
    "w_log_nodes": 0.05,
    "w_log_edges": 0.03,
    "w_density": -0.15,
    "w_avg_degree": 0.02,
    "w_degree_variance": 0.18,
    "w_hub_concentration": 0.22,
    "w_clustering_coeff": 0.12,
    "w_avg_path_length": 0.08,
    "w_diameter": 0.05,
    "w_community_count": 0.10,
    "w_reorder_time": -0.02,
    "benchmark_weights": {
      "pr": 1.15, "bfs": 1.05, "cc": 1.0
    },
    "_metadata": {
      "sample_count": 1247,
      "avg_speedup": 1.32,
      "win_rate": 0.68
    }
  }
}
```


Training Pipeline Commands

Full Training Workflow

```
# 1. Initialize fresh weights
python3 scripts/graphbrew_experiment.py --init-weights

# 2. Fill weights with heuristic defaults
python3 scripts/graphbrew_experiment.py --fill-weights \
    --graphs medium --benchmarks pr bfs cc

# 3. Run comprehensive training
python3 scripts/graphbrew_experiment.py --train-adaptive \
    --graphs all \
    --benchmarks pr bfs cc sssp bc \
    --trials 5 \
    --learning-rate 0.01

# 4. Validate trained weights
python3 scripts/graphbrew_experiment.py --validate-adaptive \
    --graphs all --benchmarks pr bfs
```

End-to-End Example

Input: soc-LiveJournal1 (4.8M nodes, 68M edges)

Step 1: Extract Global Features

- modularity = 0.71, degree_variance = 2.34
- hub_concentration = 0.45, avg_path_length = 5.2

Step 2: Find Best Type Match

- Matches type_3 (social network cluster)
- Load weights from type_3.json

Step 3: Leiden Detects 847 Communities

- Community 1 (450K nodes): dense, high clustering → **RABBITORDER**
- Community 2 (120K nodes): sparse, hub-dominated → **HubClusterDBG**
- Community 3 (50K nodes): small → **ORIGINAL**

Step 4: Apply & Merge

Result: PageRank speedup = **1.47x** vs ORIGINAL (Best fixed: 1.35x)

Key Innovations

1. Per-Community Algorithm Selection

Unlike traditional approaches that apply one algorithm globally, GraphBrew selects the best algorithm for each community.

2. Online Learning with Incremental Updates

Weights are updated immediately after each benchmark run.

3. Type-Based Specialization

Auto-clustering groups similar graphs into types with specialized weights.

4. Extended Feature Set

12 features capture both global graph structure and local community characteristics.

5. Cache-Aware Training

Optional cache simulation data helps tune weights for specific memory hierarchies.

Performance Summary

Expected Speedups

Graph Type	Typical Speedup	Best Algorithm
Social Networks	1.3-1.5x	LeidenDFS, RabbitOrder
Road Networks	1.2-1.4x	RCM, CORDER
Web Graphs	1.4-1.6x	HubClusterDBG, DBG
Power-Law	1.3-1.5x	RabbitOrder, HubCluster
Uniform Random	1.0-1.1x	ORIGINAL, DBG

Adaptive Advantage

- **5-15% improvement** over best single algorithm
- **Handles mixed graphs** where no single algorithm wins
- **Robust across benchmarks** (PR, BFS, CC, SSSP, BC)

Conclusion

GraphBrew Capabilities

- Adaptive per-community algorithm selection
- ML-based perceptron with 12 graph features
- Online learning with incremental weight updates
- Type-based weight specialization
- 18 reordering algorithms supported
- 5 graph benchmarks (PR, BFS, CC, SSSP, BC)
- Cache simulation support

Key Results

- **5-15% improvement** over best fixed algorithm
- **Automatic adaptation** to graph characteristics
- **Continuous learning** from benchmark results

Future Work

- GPU-accelerated community detection
- Larger training corpus
- Neural network score function
- Multi-level recursive adaptive ordering

GraphBrew - Making Graph Processing Cache-Efficient Through Intelligent Reordering