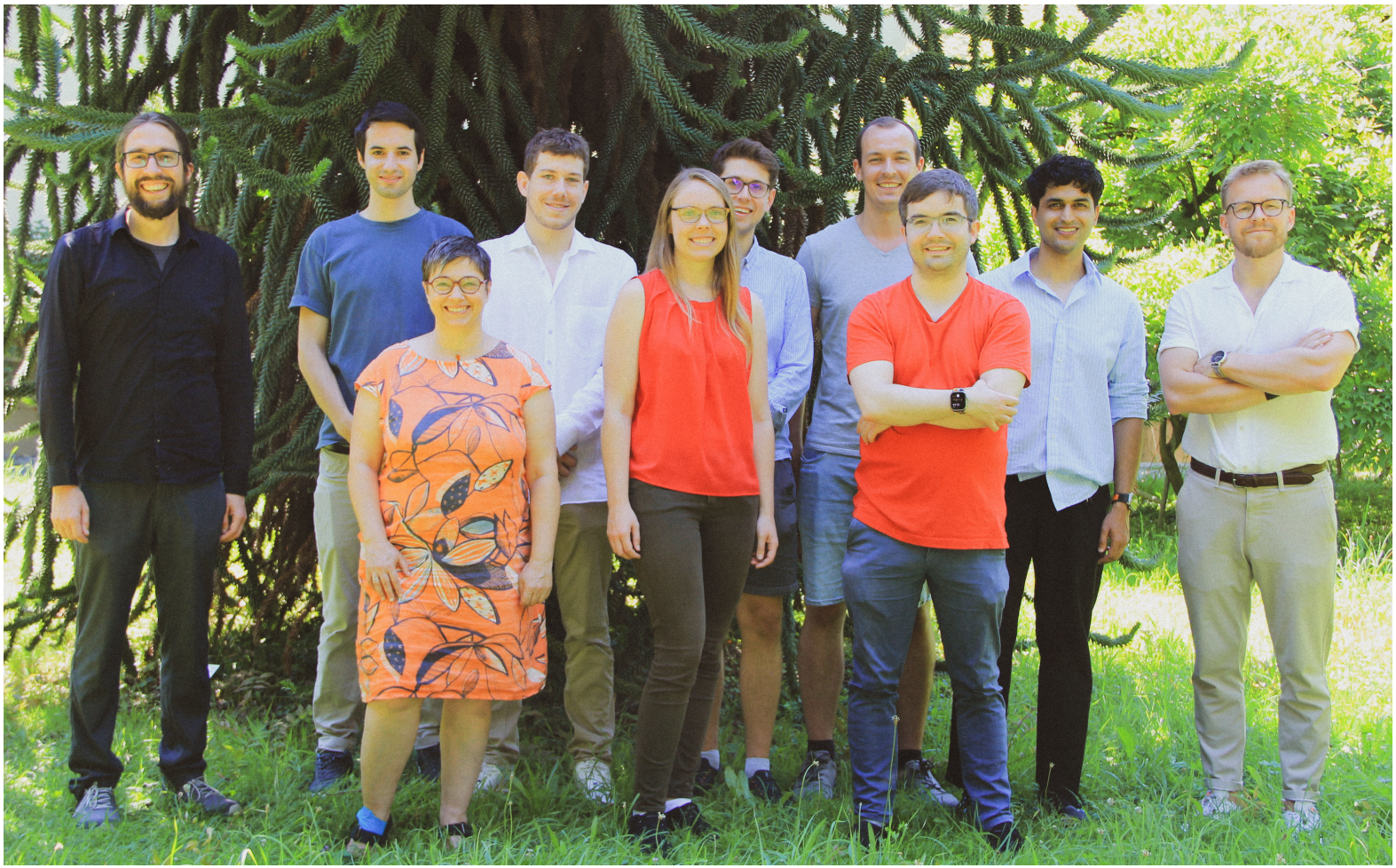
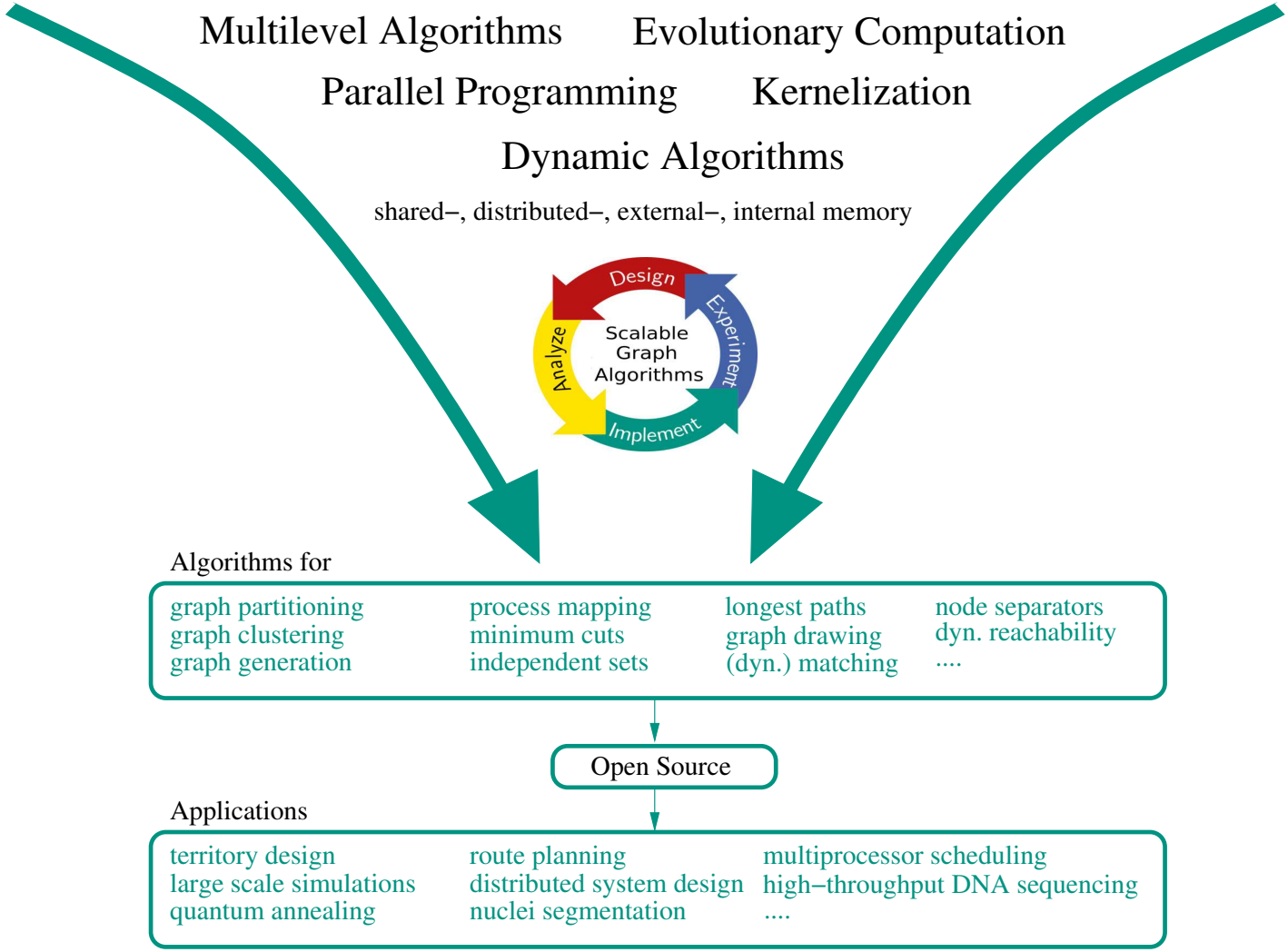
Graph Algorithms and Graph Learning:
A Two-Way Street

Christian Schulz



Algorithm Engineering Group
Heidelberg University



(a) algorithms that run on multi-/many-core systems

(b) algorithms for multi-/many-core system support
→ load balancing, communication efficiency

Algorithm Engineering Group
Heidelberg University

Open Source

Graph & Hypergraph Decomposition [KaHIP org]

- **KaHIP** -- High Quality Partitioning
- **KaMinPar** -- Parallel Graph Partitioning
- **KaHyPar** -- Hypergraph Partitioning
- **Mt-KaHyPar** -- Multi-Threaded Hypergraph Partitioning
- **HeiStream** -- Streaming Graph Partitioning
- **StreamCPI** -- Compressed Streaming Partitioning
- **FREIGHT** -- Streaming Hypergraph Partitioning

Cuts & Clustering [KaHIP org]

- **VieCut** -- Vienna Minimum Cuts
- **HeiCut** -- Hypergraph Minimum Cuts
- **HeiConnect** -- Connectivity Augmentation
- **VieClus** -- Vienna Graph Clustering
- **CluStRE** -- Streaming Graph Clustering
- **SCC** -- Correlation Clustering
- **HeidelbergMotifClustering** -- Motif Clustering
- **fpt-max-cut** -- FPT Kernelization for Max Cut

Independence & Packing [KaMIS org]

- **KaMIS** -- Maximum Independent Sets
- **CHILS** -- Concurrent MWIS Heuristic
- **LearnAndReduce** -- GNN-Guided MWIS Reductions
- **HyperMIS** -- Hypergraph Independent Sets
- **HeiHGM/Bmatching** -- Hypergraph b-Matching
- **HeiHGM/Streaming** -- Streaming Hypergraph Matching
- **red2pack** -- 2-Packing Set Solver

Dynamic Graph Algorithms [DynGraphLab org]

- **DynDeltaOrientation** -- Dynamic Edge Orientation
- **DynDeltaApprox** -- Edge Orientation Approx.
- **DynMatch** -- Dynamic Maximal Matching
- **DynWMIS** -- Dynamic Max Weight Ind. Set
- **DynReach** -- Dynamic Reachability

Process Mapping & HPC [KaHIP org]

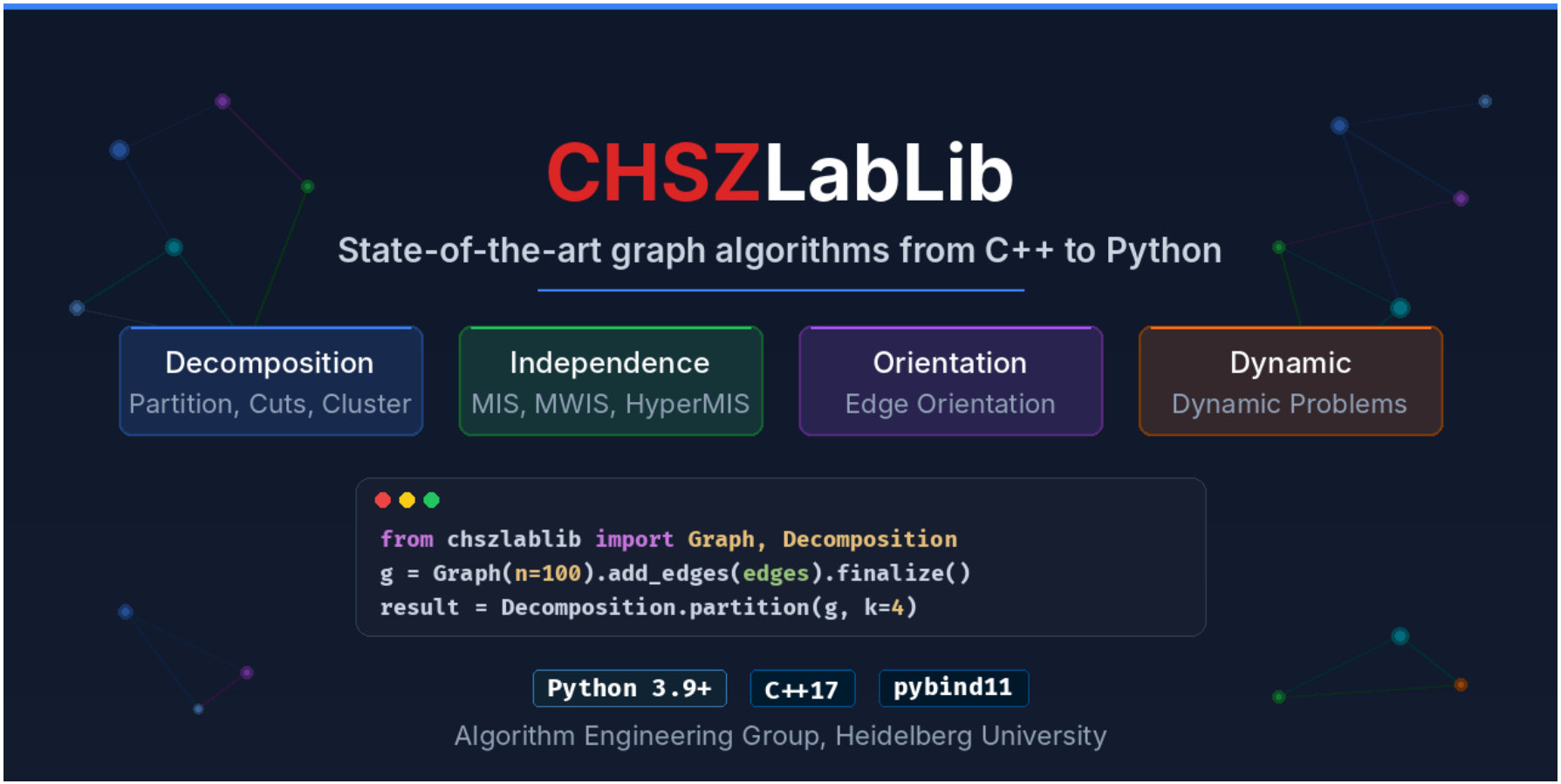
- **VieM** -- Mapping & Sparse QAP
- **SharedMap** -- Shared-Memory Process Mapping
- **IntegratedProcessMapping** -- Multi-Level Mapping
- **OnlineMultiSection** -- Streaming Process Mapping

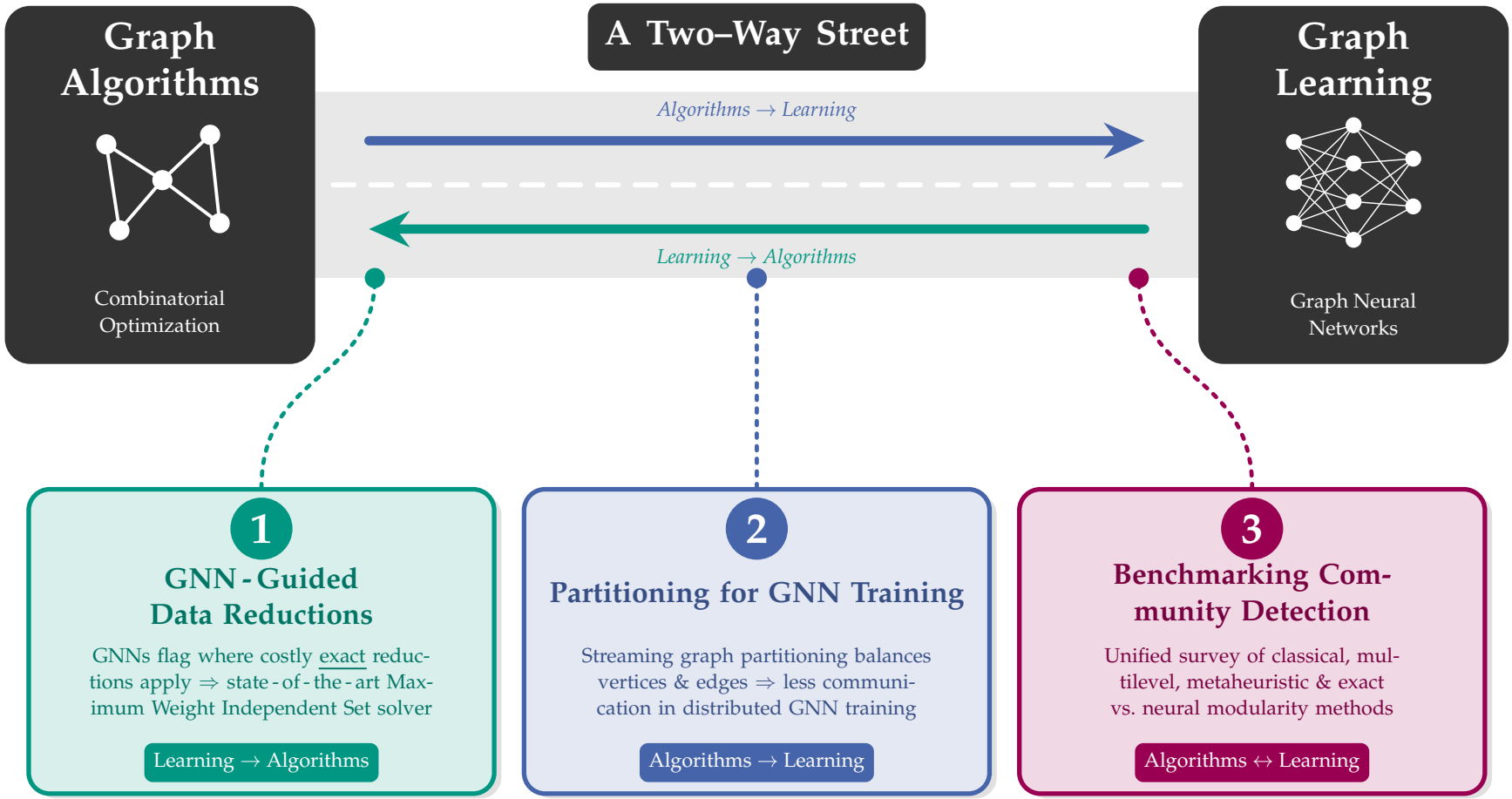
Other

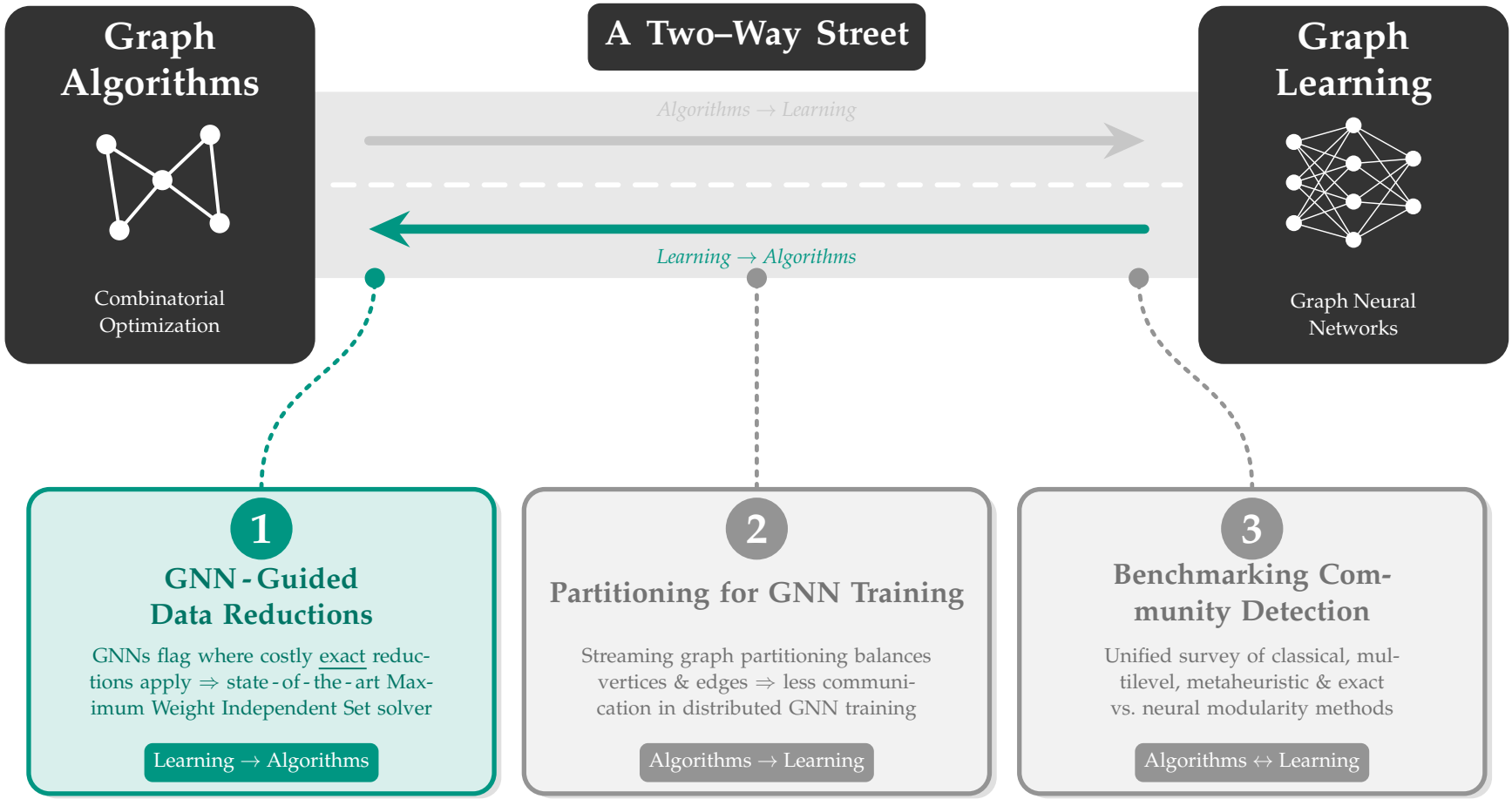
- **HeiOrient** -- Edge Orientation
- **KaGen** -- Graph Generation
- **KaDraw** -- Graph Drawing
- **KaLP** -- Longest Paths
- **KaSVM** -- Support Vector Machine
- **Arc-FlagTB** -- Public Transit Routing

+ AI Skills like https://github.com/CHSZLab/AgenticAlgorithmEngineering
⇝ auto optimize your algorithms using Algorithm Engineering

Open Source


CHSZLabLib
State-of-the-art graph algorithms from C++ to Python
Decomposition
Partition, Cuts, Cluster
Independence
MIS, MWIS, HyperMIS
Orientation
Edge Orientation
Dynamic
Dynamic Problems
from chszlablib import Graph, Decomposition
g = Graph(n=100).add_edges(edges).finalize()
result = Decomposition.partition(g, k=4)
Python 3.9+
C++17
pybind11
Algorithm Engineering Group, Heidelberg University

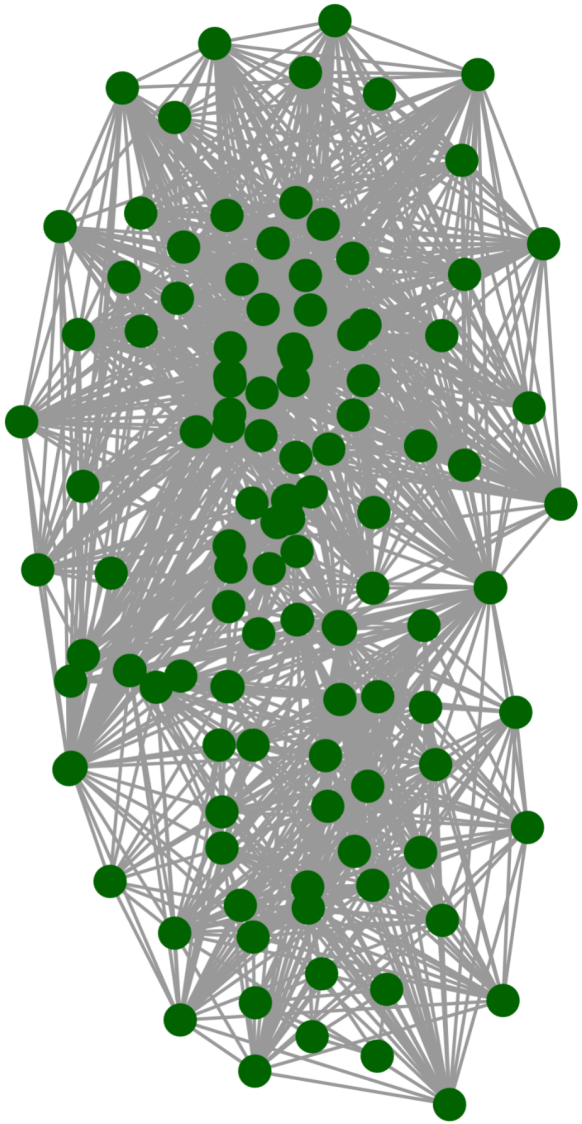

Graph Algorithms
Combinatorial Optimization
A Two–Way Street
Algorithms → Learning
Learning → Algorithms
Graph Learning
Graph Neural Networks
1
GNN-Guided Data Reductions
GNNs flag where costly exact reductions apply ⇒ state-of-the-art Maximum Weight Independent Set solver
Learning → Algorithms
2
Partitioning for GNN Training
Streaming graph partitioning balances vertices & edges ⇒ less communication in distributed GNN training
Algorithms → Learning
3
Benchmarking Community Detection
Unified survey of classical, multilevel, metaheuristic & exact vs. neural modularity methods
Algorithms ↔ Learning



Joint work w/ E. Großmann and K. Langedal

Why Data Reductions?

input **graph**

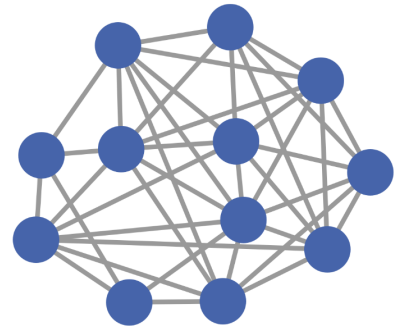


Data Reductions:

rules to
decrease graph size

while maintaining
optimality

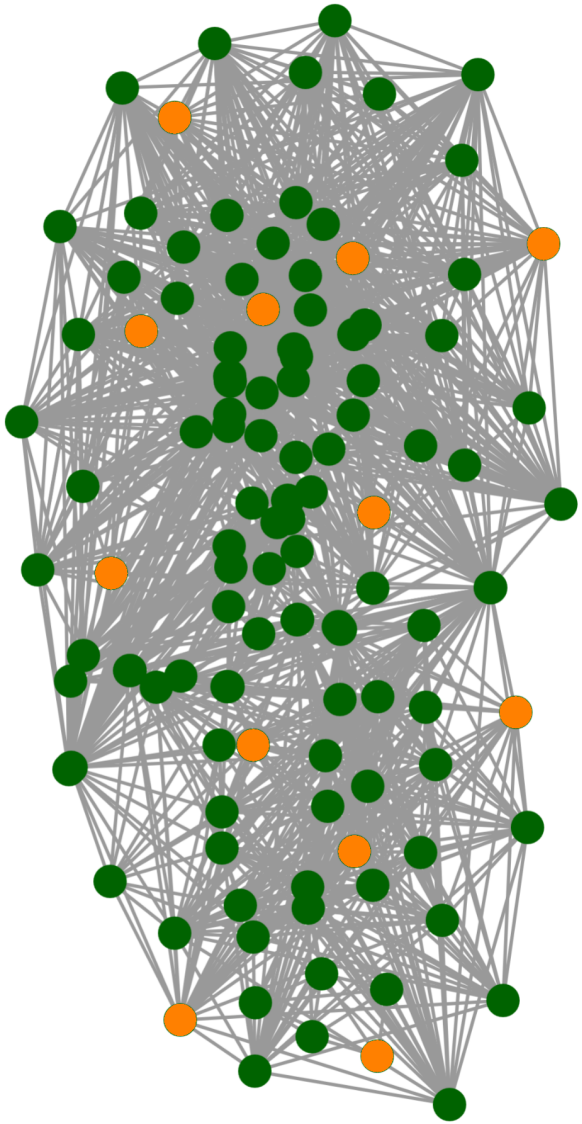
kernel



Algorithm Engineering Group
Heidelberg University

Why Data Reductions?

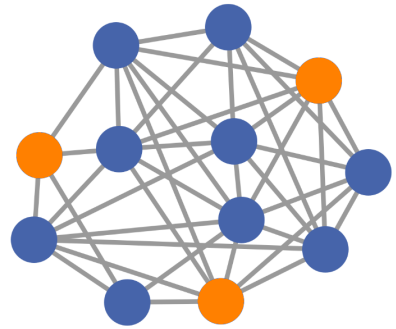
input **graph**



reconstruct

solution
(exact or heuristic)

kernel
solution



apply exact or
heuristic solver

Algorithm Engineering Group
Heidelberg University

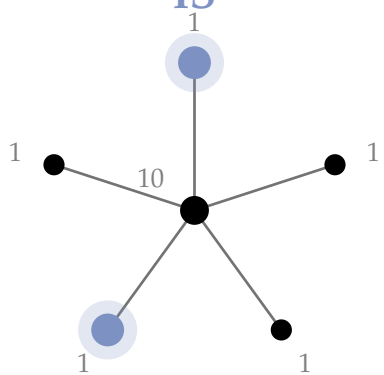
Independent Set

Independent Set:
a set of vertices that are pairwise non-adjacent

Maximum Independent Set:
an independent set of maximum cardinality

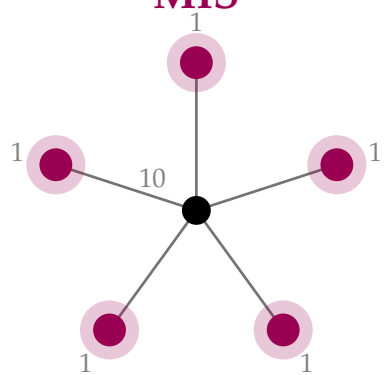
Maximum Weight Independent Set:
an independent set of maximum weight

IS



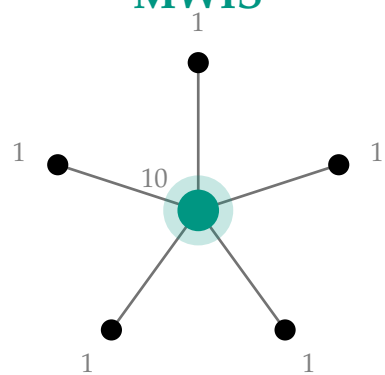
weight = 2

MIS



cardinality = 5

MWIS



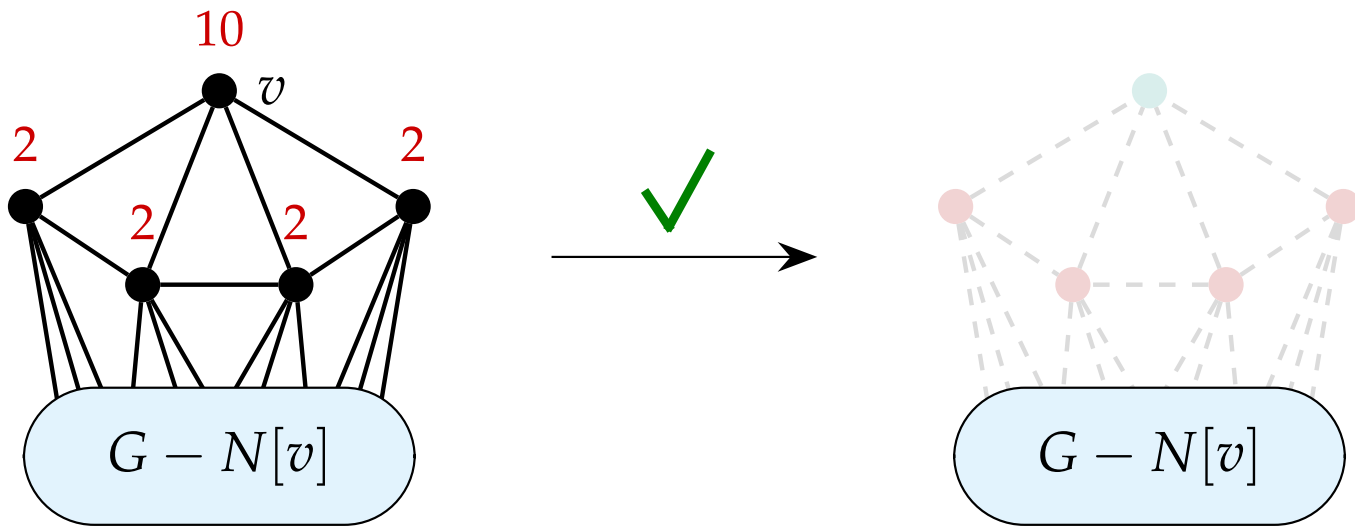
weight = 10

Algorithm Engineering Group
Heidelberg University

MWIS Reductions

Reduction

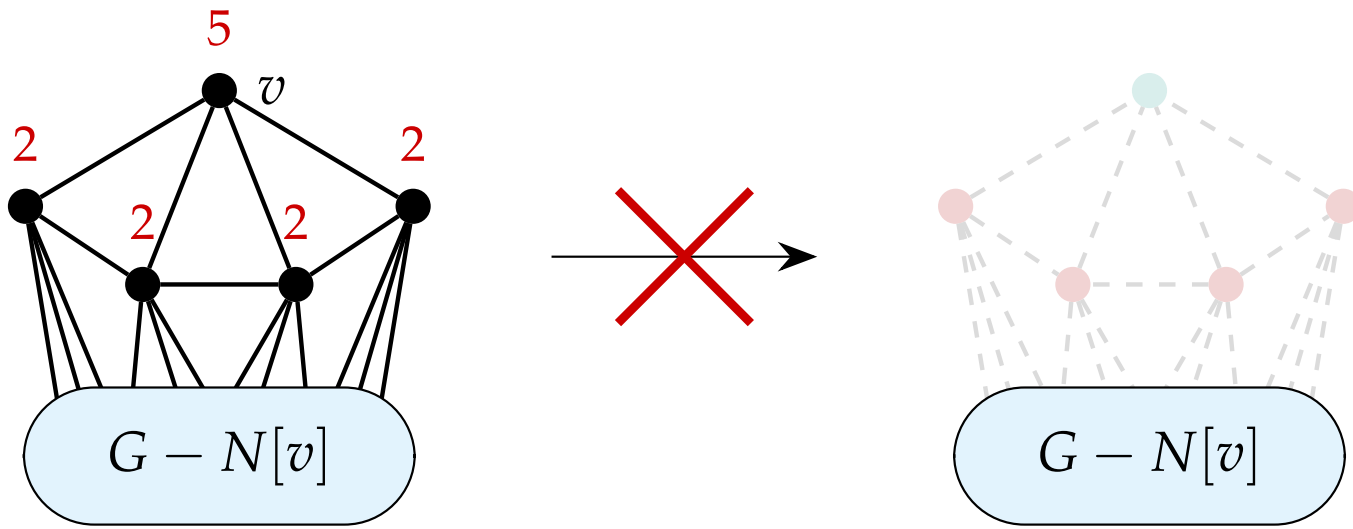
Include a vertex v if $\omega(v) \geq \omega(N(v))$



MWIS Reductions

Reduction

Include a vertex v if $\omega(v) \geq \omega(N(v))$

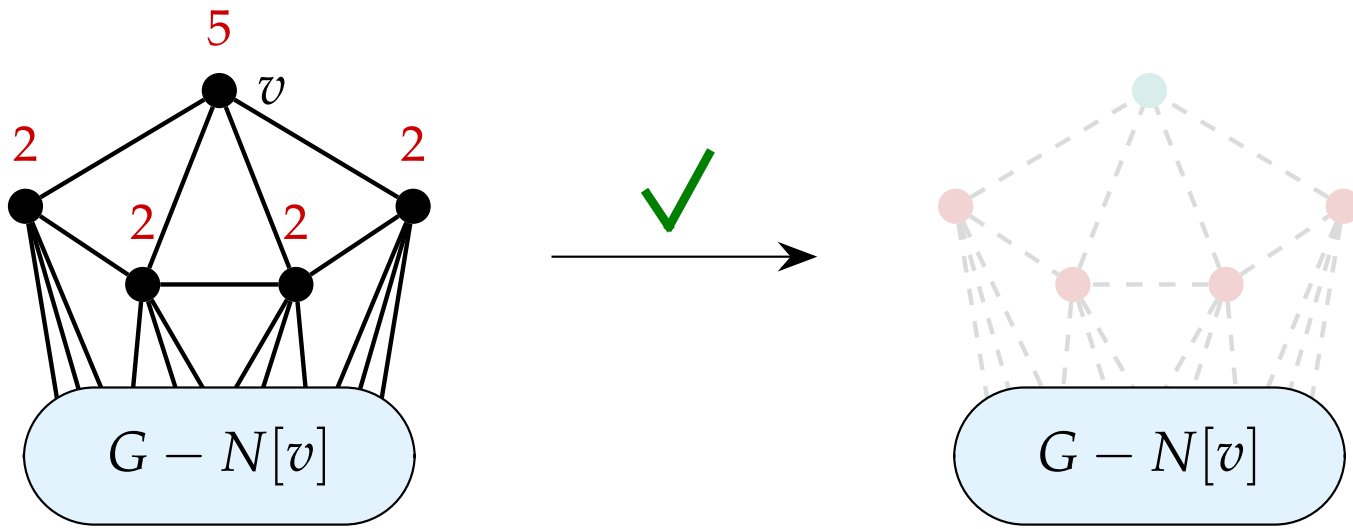


Tighten the bound $\rightarrow$ solve MWIS in $G[N(v)]$

MWIS Reductions

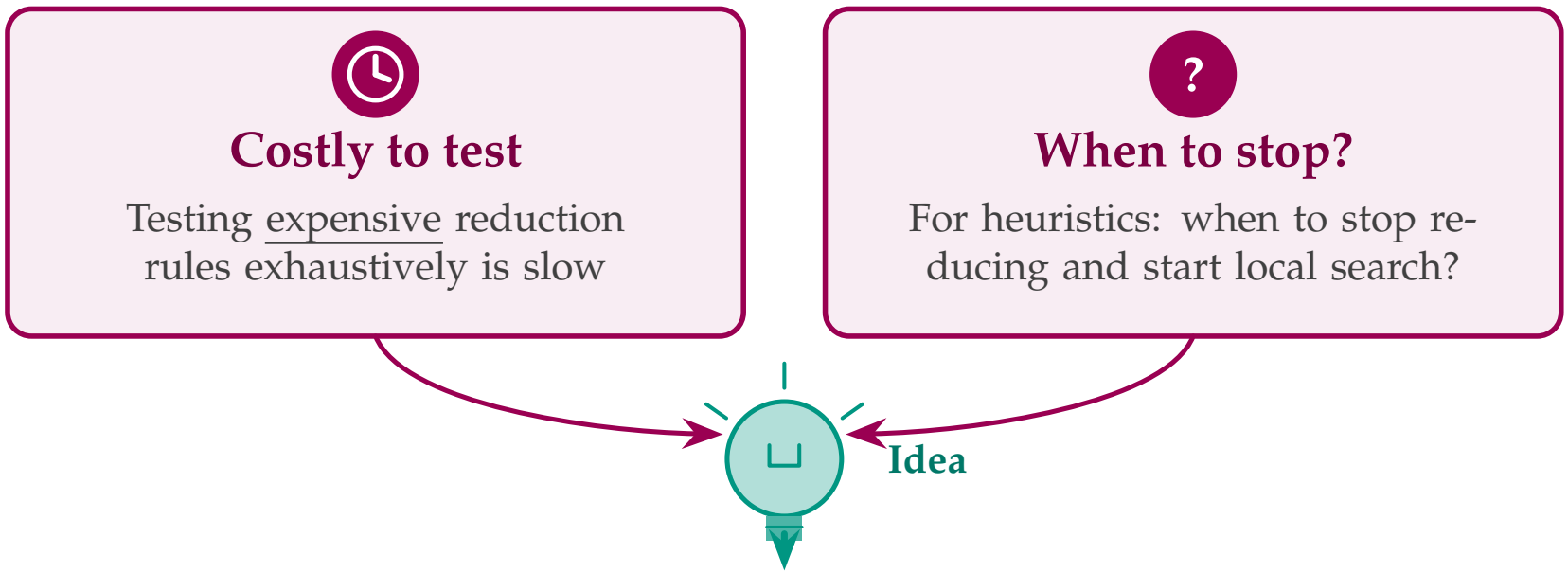
Reduction

Include a vertex v if $\omega(v) \geq \alpha_\omega(G[N(v)])$



Tighten the bound $\rightarrow$ solve MWIS in $G[N(v)]$

Motivation

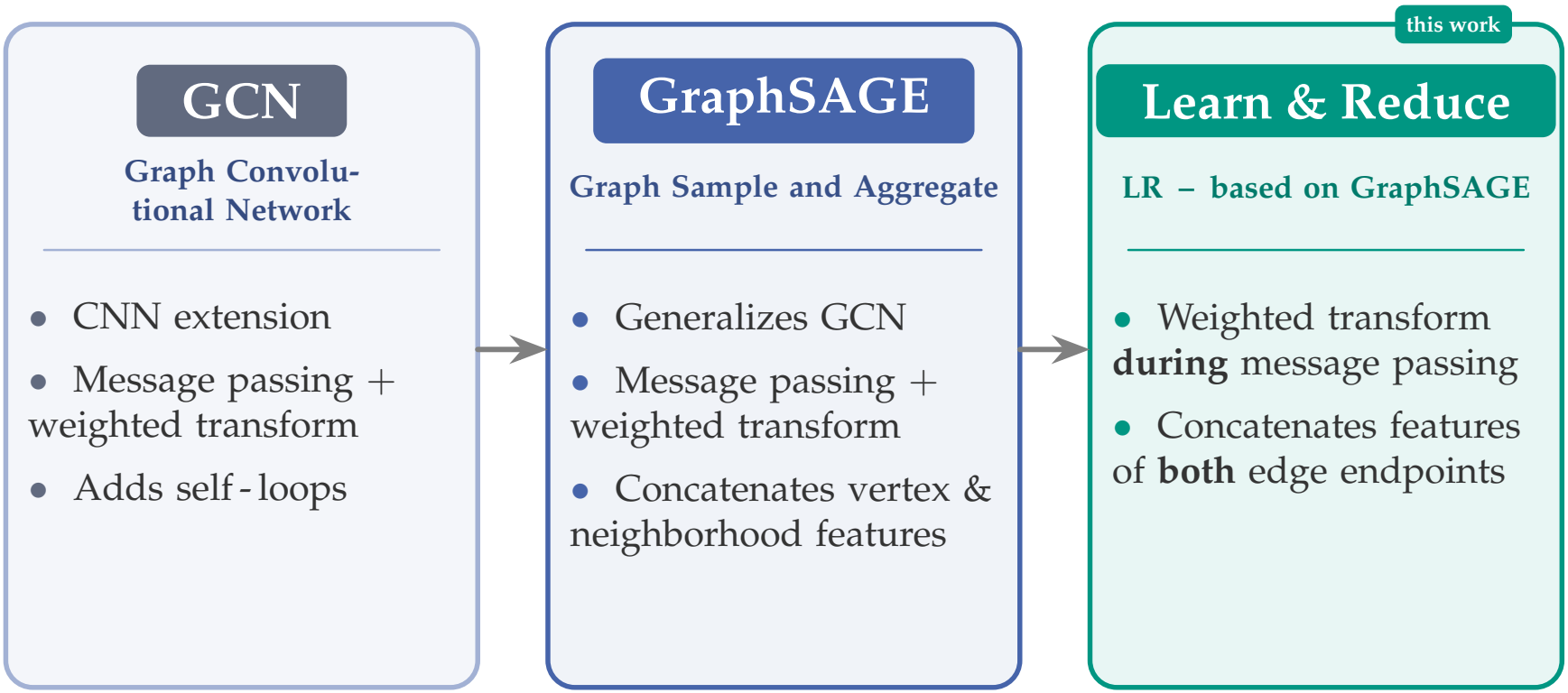


Selective, GNN-guided reductions

- ✓ Always apply the **cheap, simple** reductions
- ✓ Run **expensive** reductions only where **GNN** predicts success

Algorithm Engineering Group
Heidelberg University

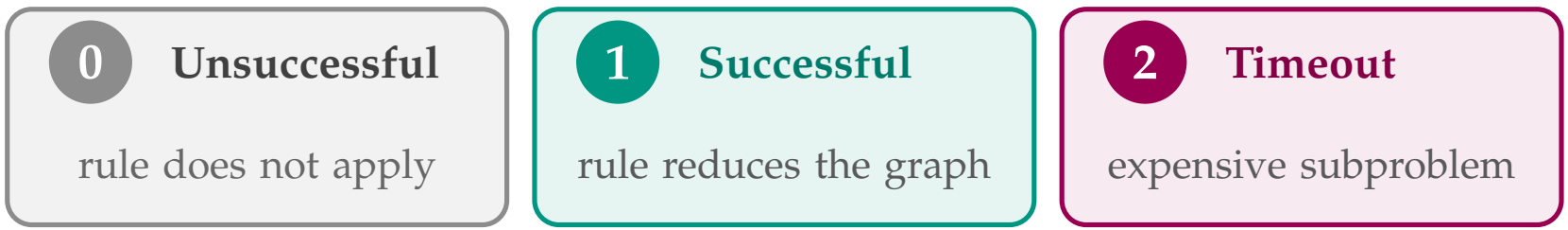
GNN Architectures



Supervised Learning Dataset

Each instance is stored twice:

Label every (vertex, red rule) pair by testing rule (without reducing):



Screening uses the **reduced** graphs: **71** graphs with $>$**1 million** labeled vertices (timeouts excluded from training)

Open dataset: zenodo.org/records/15210077

Training Setup

Inference — for every vertex, in three steps:

Test expensive reductions only where the score is high $\Rightarrow$ skip the rest and save time

Under the hood: 8 cheap features per vertex · 2 message-passing layers (16 hidden) + small MLP · 71 graphs (>1M vertices), Adam, 300 epochs

$$H_u^{(l+1)} = \text{MEAN}\left\{ \sigma\left(W^{(l)} \cdot \text{CONCAT}(H_u^{(l)}, H_v^{(l)}) + b^{(l)}\right) : v \in N(u) \right\}$$

Training learns the weights $W^{(l)}, b^{(l)}$;

Inference runs steps 1–3 and keeps vertices whose score exceeds a threshold

Training Results

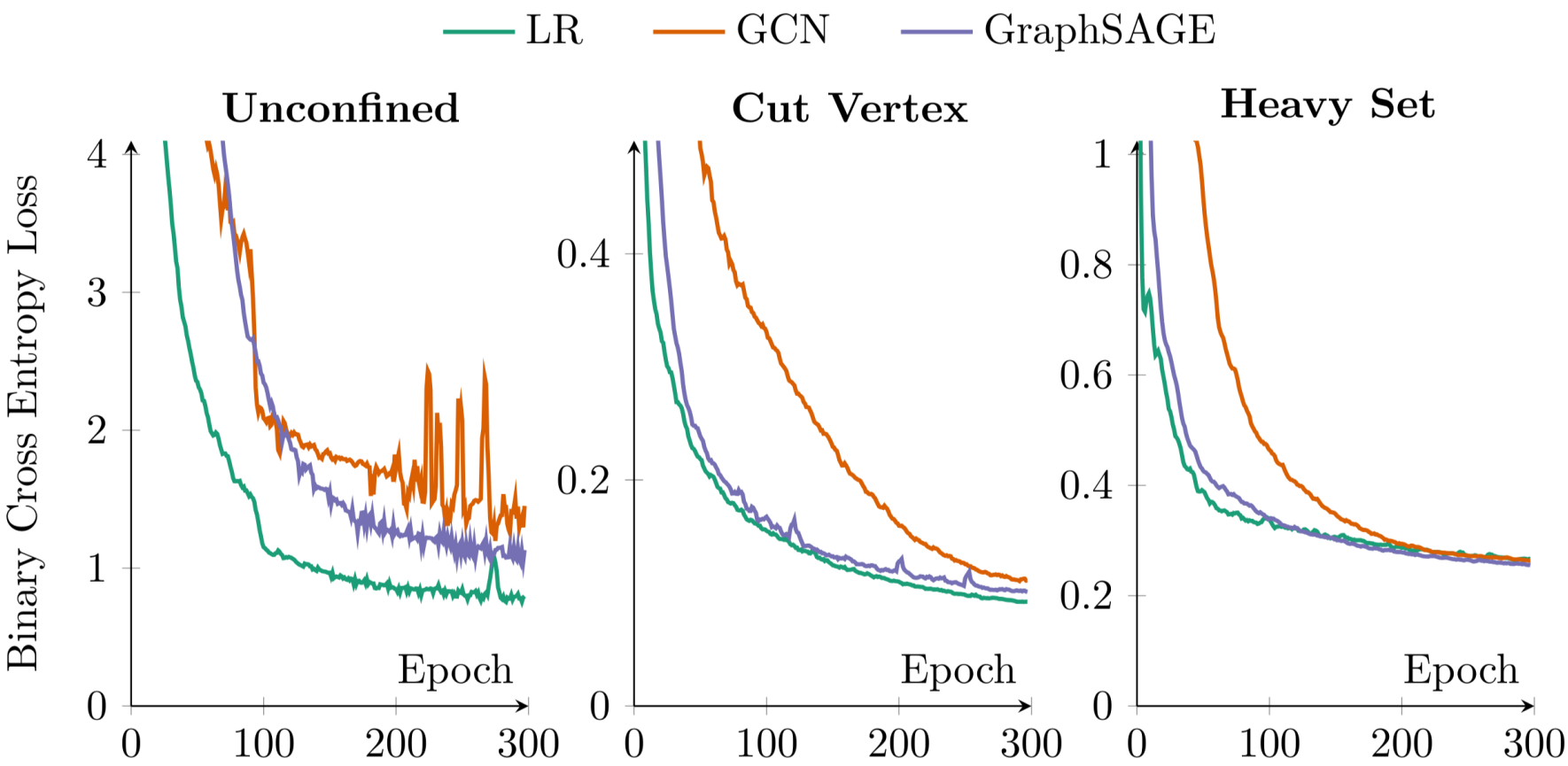


Figure: Training loss for each architecture and different rules

Training Results

Further metrics:

- **Accuracy:**
  Probability that a suggested vertex can be reduced
- **Coverage:**
  Fraction of reducible vertices suggested
- **Screening:**
  Fraction of total vertices suggested

Training Results:

Per-Rule Screening Quality

Reduction	GCN			GraphSAGE			LR (ours)		
	Scr ↓	Cov ↑	Acc ↑	Scr ↓	Cov ↑	Acc ↑	Scr ↓	Cov ↑	Acc ↑
Critical	7.03	**44.06**	38.81	**4.72**	31.73	41.63	6.52	43.99	**42.23**
Cut	0.03	2.89	55.91	**0.00**	0.16	24.25	0.06	**8.07**	**77.80**
Gen. fold	**0.14**	2.34	**42.17**	0.83	12.68	38.28	0.98	**14.33**	36.73
Heavy set	**2.25**	31.39	**50.65**	3.77	44.84	43.39	5.02	**51.03**	37.13
Heavy set 3	**0.47**	7.88	**45.45**	0.51	5.74	30.60	0.90	**9.35**	28.00
Unconfined	10.15	25.95	25.37	**5.65**	17.20	**27.07**	30.97	**53.36**	14.10

Screening: fraction of all vertices suggested

Coverage: fraction of reducible vertices suggested

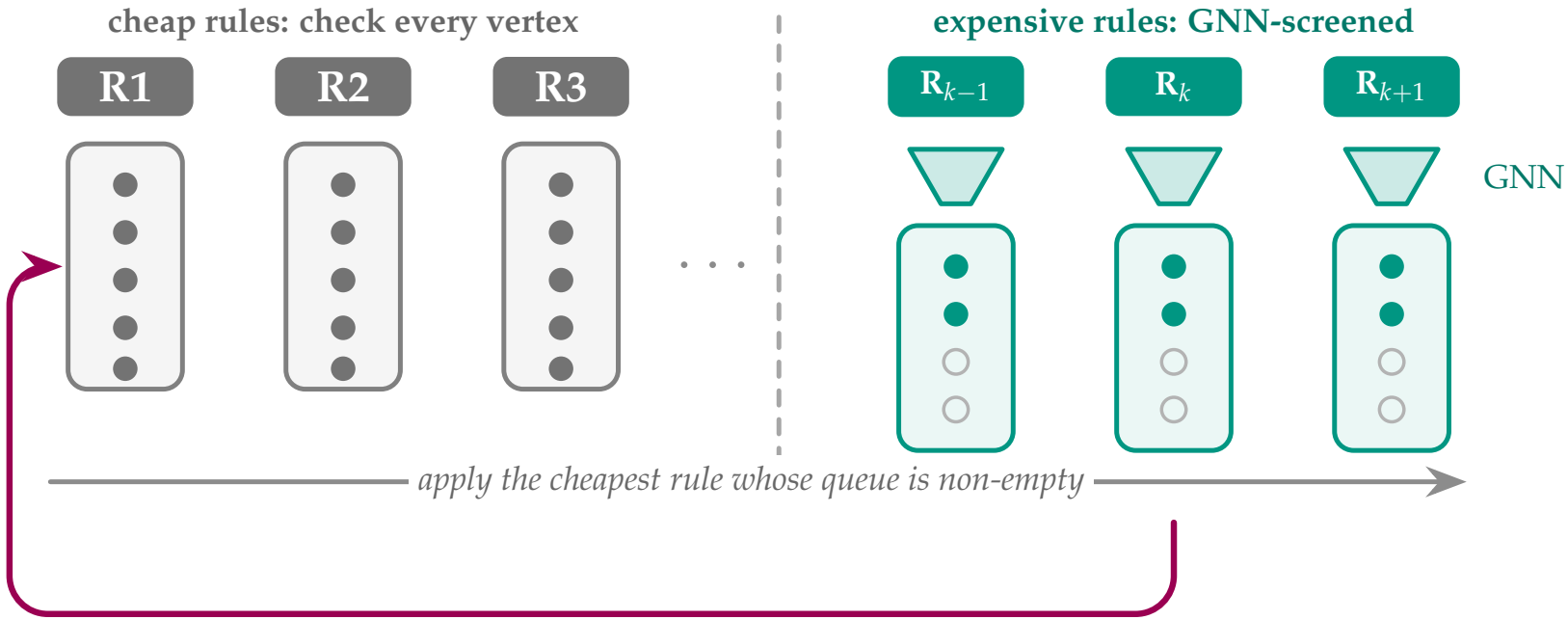
Accuracy: chance a suggested vertex reduces

LR reaches the highest **coverage** on almost every rule

Algorithm Engineering Group
Heidelberg University

The Reduction Process

Ordered list of reduction rules, cheap → expensive – each keeps a FIFO queue of vertices



a successful reduction ⇒ restart at the cheapest rule and re-queue the changed vertex's neighbors in every rule

Algorithm Engineering Group
Heidelberg University

Different Configurations

more screening ⇒ faster

Configuration	Initially checks …	When re-applied …
no gnn red	no expensive reductions at all	— (baseline)
never	all vertices	changed vertices (exhaustive)
always	GNN-suggested vertices	re-run the GNN, check new suggestions
initial	GNN-suggested vertices	changed vertices (exhaustive)
initial tight	GNN-suggested vertices	— (only the remaining initial suggestions)

All configurations still apply the **cheap** reductions exhaustively – they differ only in how the **expensive** rules are screened.

Algorithm Engineering Group
Heidelberg University

Experiments: Reduction Configurations

Config	GNN	kernel $n_K / \tilde{n}$	m_K / m_G	offset	time [s]
no gnn red	–	100.00 %	51.13 %	13 179 456	36.55
never	–	**86.78 %**	50.85 %	13 270 743	74.74
always	✓	87.11 %	50.90 %	13 269 029	65.51
initial	✓	87.13 %	50.88 %	13 268 824	58.59
initial tight	✓	87.41 %	50.91 %	13 267 958	**45.12**

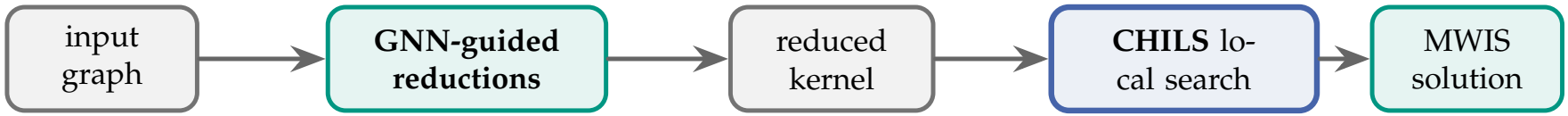
GNN screening (**initial tight**) keeps almost the same kernel as exhaustive reductions (**86.78 %** → 87.41 %) while cutting reduction **time by** ≈40 % (75 s → 45 s)

$\tilde{n}$: vertices of the largest kernel (*no gnn red*); G: input graph.

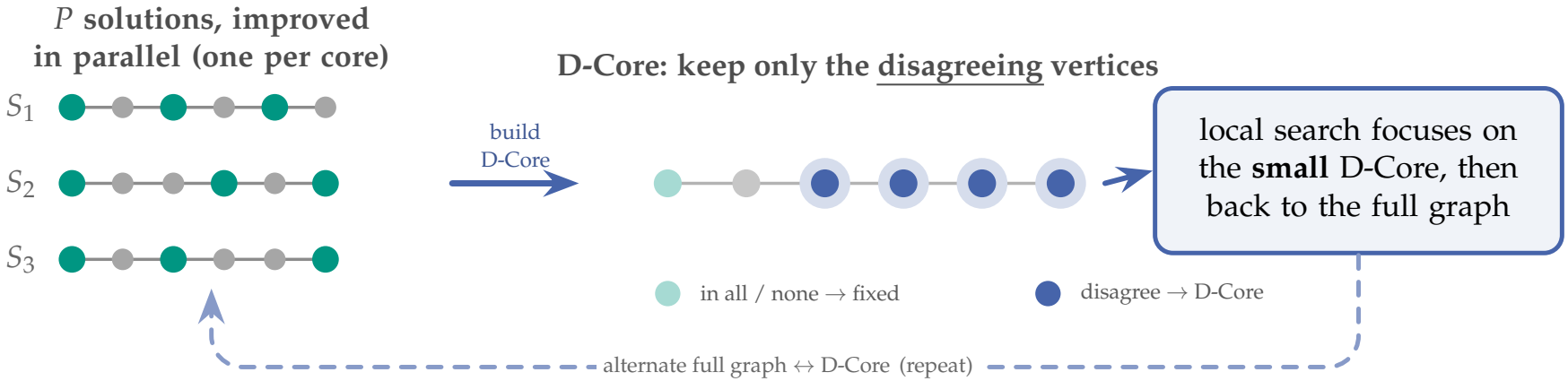
Algorithm Engineering Group
Heidelberg University

Conc. Hybrid Iterated Local Search

Full solver pipeline:



Inside CHILS — the **Difference-Core (D-Core)**



Embarrassingly parallel: one solution per core
$\Rightarrow$ up to **104**$\times$ speedup on a 128-core machine

Main Results

best on ALL

reduced (non-VR) instances: CHILS finds the best solution of all tested heuristics

31 / 37

vehicle-routing instances won (35 / 37 with parallel CHILS)

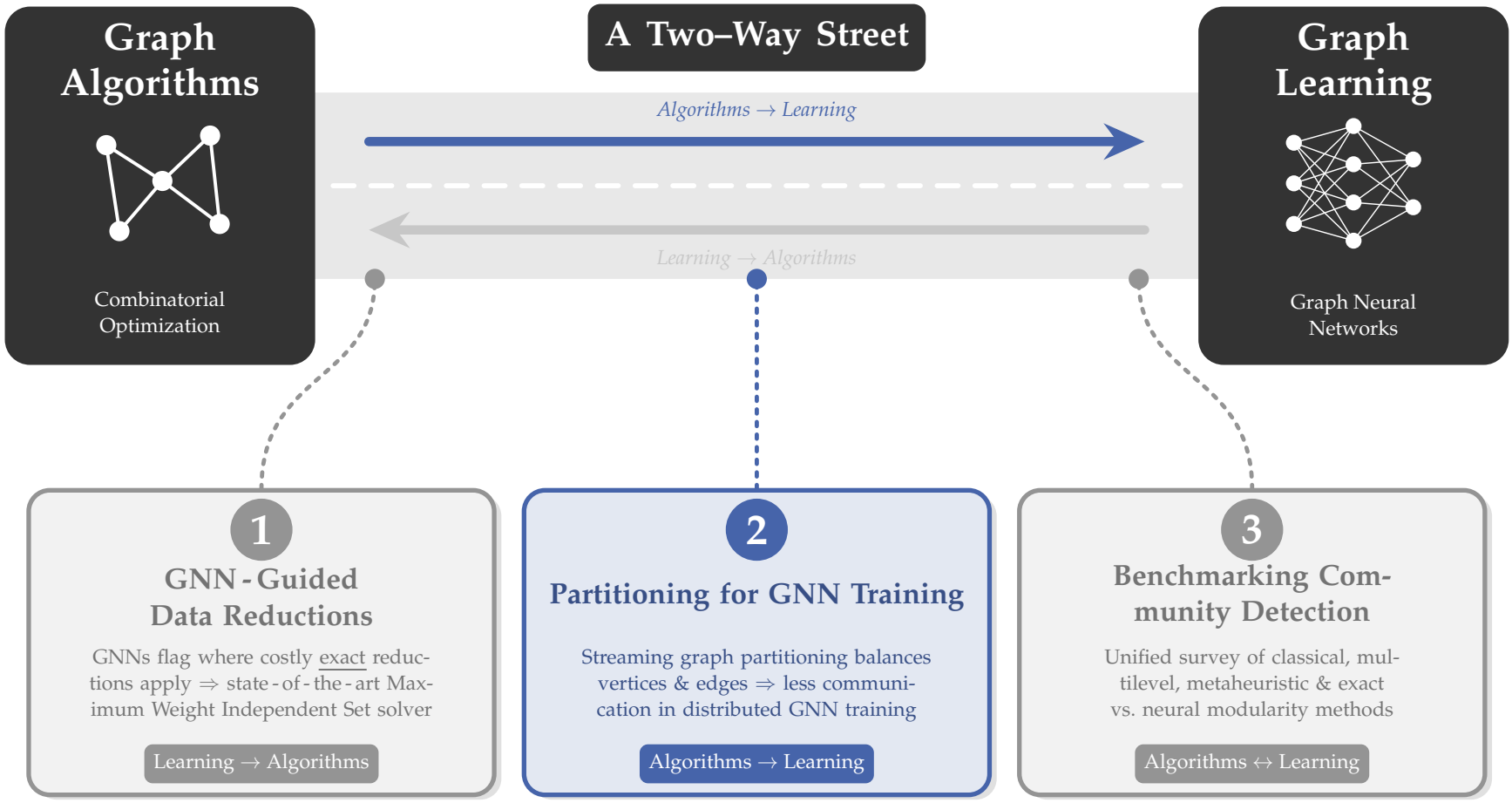
up to 104×

parallel speedup on a 128-core machine (slowest instance still 28×)

open source

`pip install chszlablib` · standalone tool & C/C++ library

GNN-guided reductions + CHILS ⇒ state-of-the-art **MWIS** solver



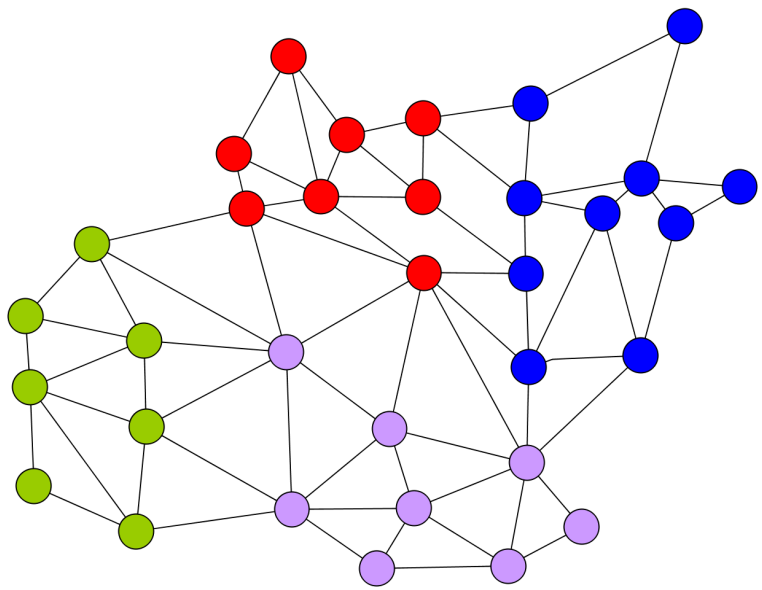
Joint work w/ B. Hoffmann, S. Perez, A. Chhabra, R. Mayer

ϵ-Balanced Graph Partitioning

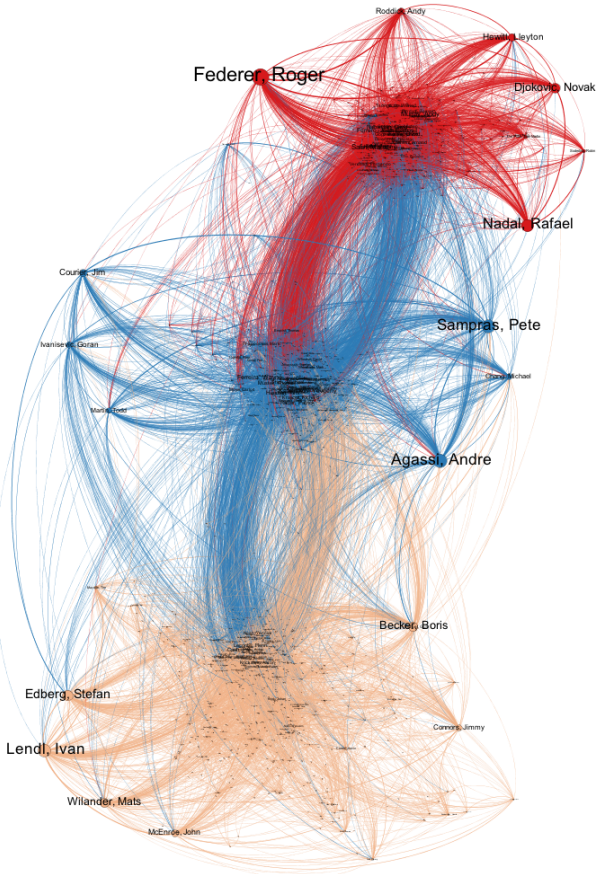
Partition $G = (V, E)$ with node weights $c : V \to \mathbf{R}_{>0}$ and edge weights $\omega : E \to \mathbf{R}_{>0}$ into k disjoint blocks:

Balance: each block's total node weight $\leq \frac{1+\epsilon}{k} \sum_{v \in V} c(v)$

Minimum cut: minimize the total weight of edges running between different blocks



The Common Parallel Approach



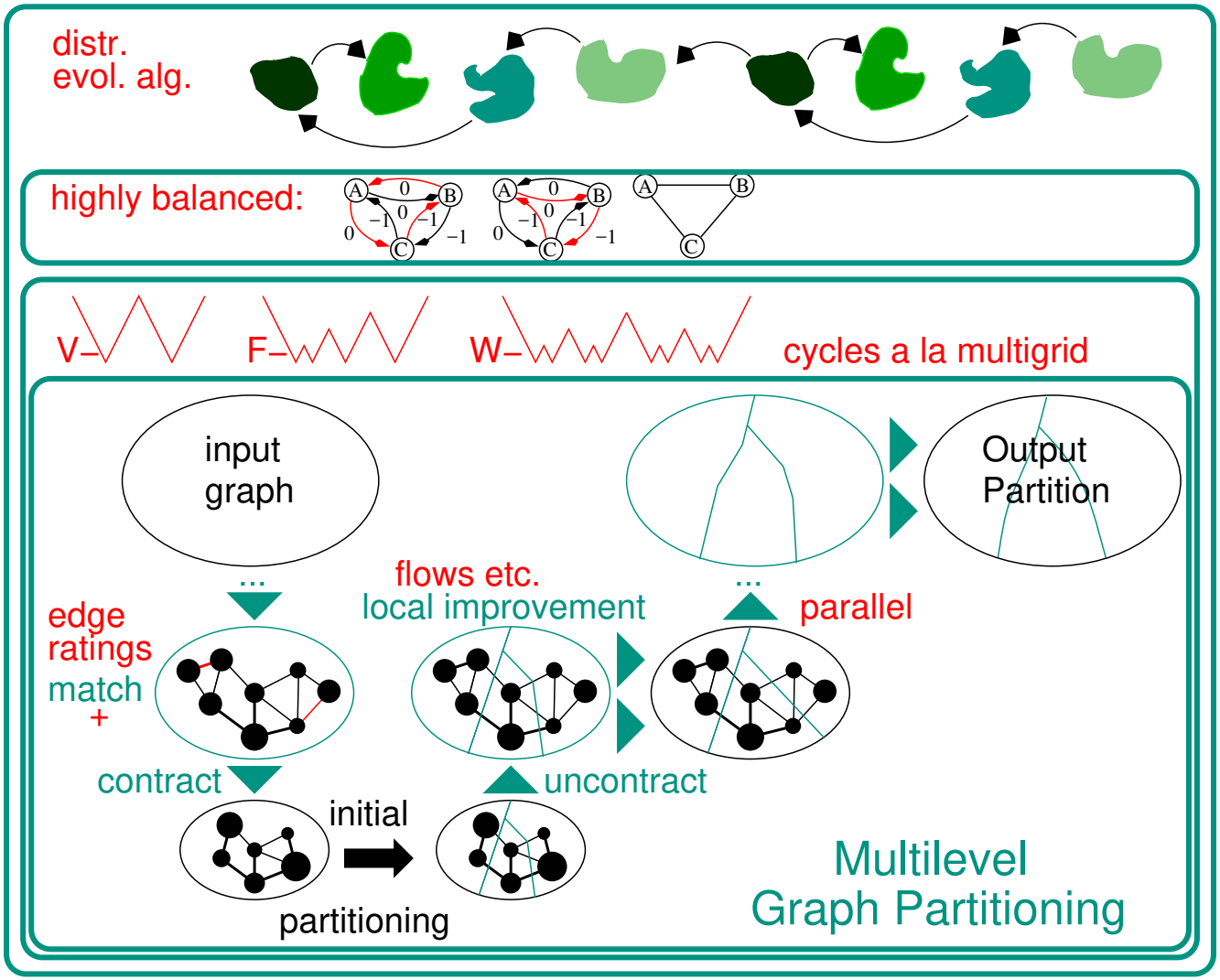
Mesh → dual graph: volumes (data & compute) become vertices, interdependencies become edges

Balanced work: every processing element (PE) gets the same amount of work

Communication: edges between blocks = costly cross-PE communication

Graph Partitioning Problem: split the graph into (almost) equally sized blocks while **minimizing the edges between blocks**

KaHIP

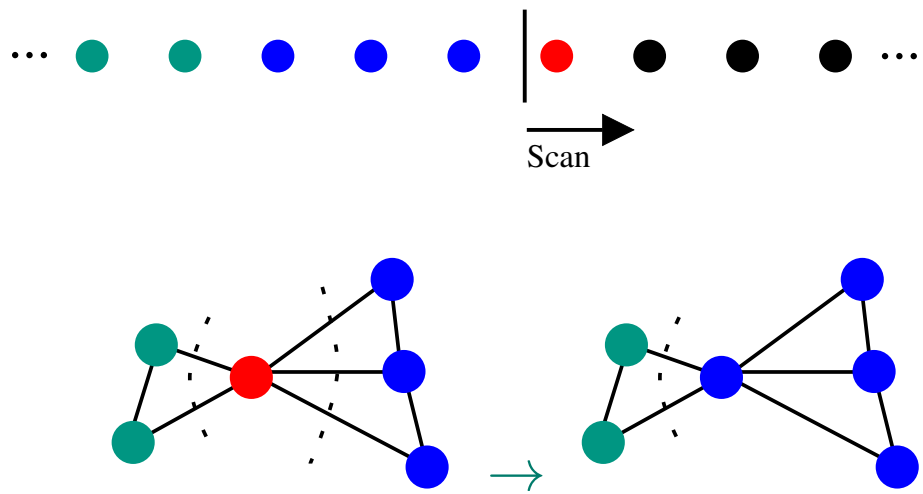

distr.
evol. alg.
highly balanced:
A
B
C
0
−1
V–
F–
W–
cycles a la multigrid
input
graph
Output
Partition
...
flows etc.
local improvement
parallel
edge
ratings
match
+
contract
uncontract
initial
partitioning
Multilevel
Graph Partitioning

Streaming Model

Online: nodes arrive one at a time, each with its neighborhood

Memory: only $\Theta(n)$ space, just the block IDs

Irrevocable: each assignment is final, decisions cannot be undone

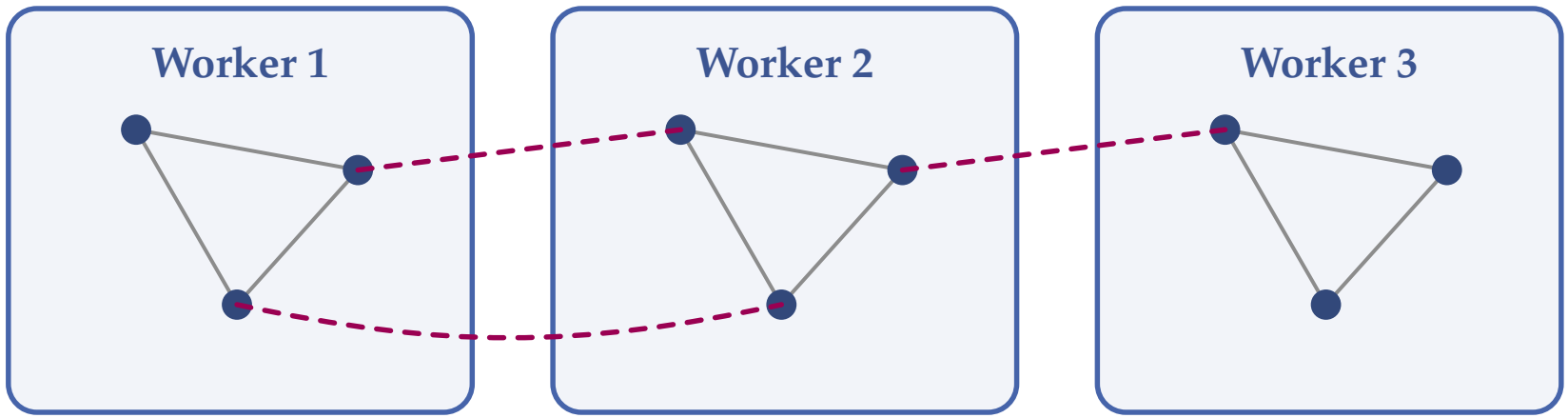


place each arriving node in the block with the **strongest connection**

used under tight memory or when a partition is needed very fast.

Why Partition for GNN Training?

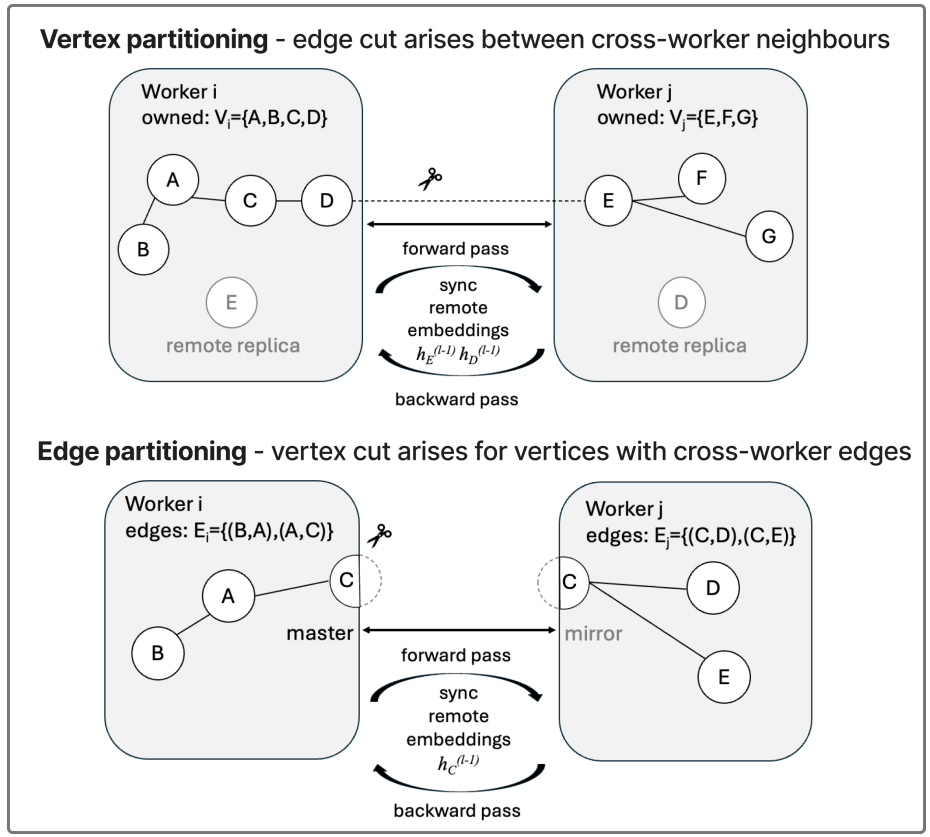
Distributed GNN training: one huge graph is split across k workers



cross-worker edges = network communication every layer

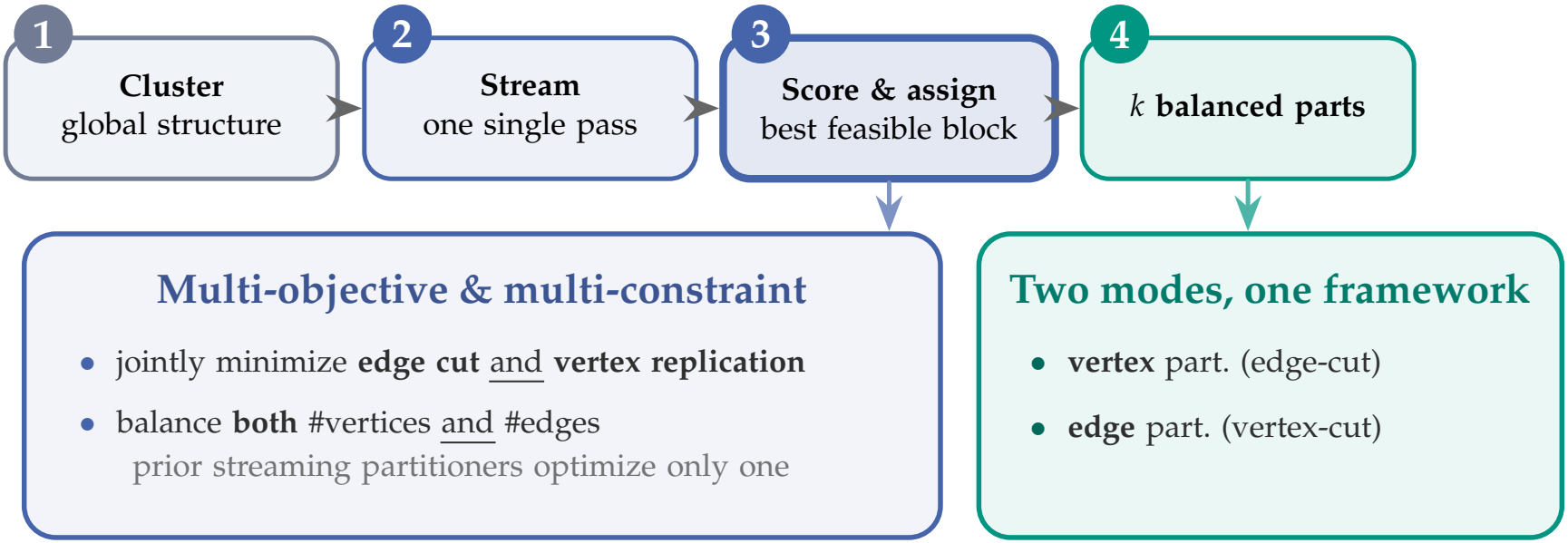
Every layer aggregates neighbors ⇒ **minimize cross-edges**
and **balance the load** across workers for **faster training**

Two Ways to Cut a Graph



Each cut creates communication – **cut edges** (vertex partitioning) or **replicated vertices** (edge partitioning) must be synchronized every layer

Streaming, Multi-Objective Partitioning



One streaming pass with clustering preprocessing $\Rightarrow$
scales to graphs that do not fit in memory $(O(m + nk))$

Under the hood: vertex mode = normalized Fennel score, edge mode = HDRF-style · dynamic capacity relaxes over the stream · jointly minimizes edge cut and vertex replication · defaults γ=2.5, τ=0.5

Multi-Objective Streaming Framework

Greedy assignment
each streamed element x (a vertex or an edge) goes to the best feasible block

$$p^* = \arg\max_{p \in \mathcal{F}(x)} S(x,p), \qquad S(x,p) = \text{affinity}(x,p) - \text{penalty}(x,p)$$

Multi-constraint feasibility
load vector L_p and capacity U_p per block; p is feasible if, for every hard-constrained dimension i,

$$L_{p,i} + \Delta_{x,i}(p) \;\le\; U_{p,i}\,\sigma(t)$$

Dynamic capacity
tighten early, relax as the stream progresses (t = fraction streamed)

$$\sigma(t) = \sigma_{\min} + (1 - \sigma_{\min})\sqrt{t}, \qquad \sigma_{\min} = \max\Big\{0.9,\ \max_{p,i} \tfrac{L_{p,i}}{U_{p,i}}\Big\}$$

If no block is feasible, x is assigned to the block minimizing the maximum relative load.

Vertex Partitioning

Example

Load & contribution

(assigned vertices, edge volume)

$$L_p = \left(L_p^{\text{vertex}},\, L_p^{\text{vol}}\right), \qquad \Delta_v(p) = \left(1,\, d(v)+1\right)$$

Capacities

$$U_p^{\text{vertex}} = \left\lceil (1+\varepsilon)\,\frac{|V|}{k} \right\rceil, \qquad U_p^{\text{vol}} = \left\lceil (1+\varepsilon_E)\,\frac{2|E|+|V|}{k} \right\rceil$$

Normalized Fennel score

locality $e(v,p) = |\{u \in N(v) : \pi(u) = p\}|$ minus imbalance

$$S_{\text{vertex}}(v,p) = \frac{e(v,p)}{d(v)} - \rho_p^{\gamma-1.1}, \qquad \rho_p = \max\left\{ \frac{L_p^{\text{vertex}}}{U_p^{\text{vertex}}},\, \frac{L_p^{\text{vol}}}{U_p^{\text{vol}}} \right\}$$

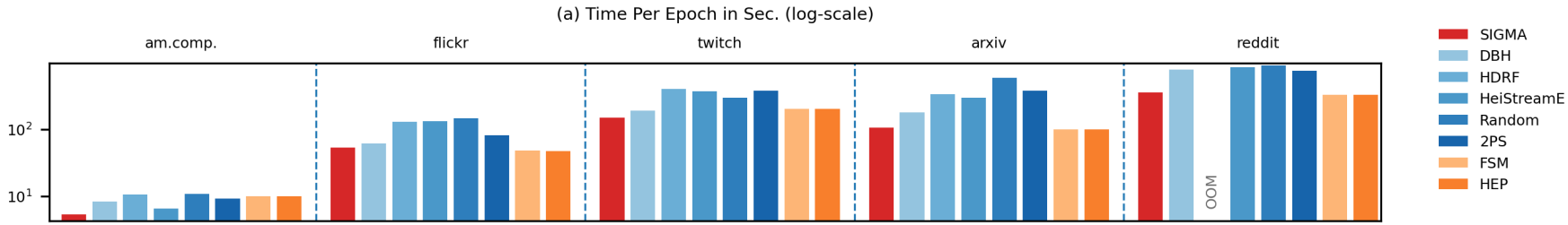
Multi-objective: add a replication penalty

($R(v,p)$ estimates new ghost replicas)

$$S_{\text{MO}}(v,p) = \frac{e(v,p)}{d(v)} - \rho_p^{\gamma-1.1} - \tau\,\frac{R(v,p)}{d(v)+k}$$

$R = R_1 + R_2$: new neighbor replicas + extra neighbor blocks ; defaults $\gamma = 2.5,\ \tau = 0.5$.

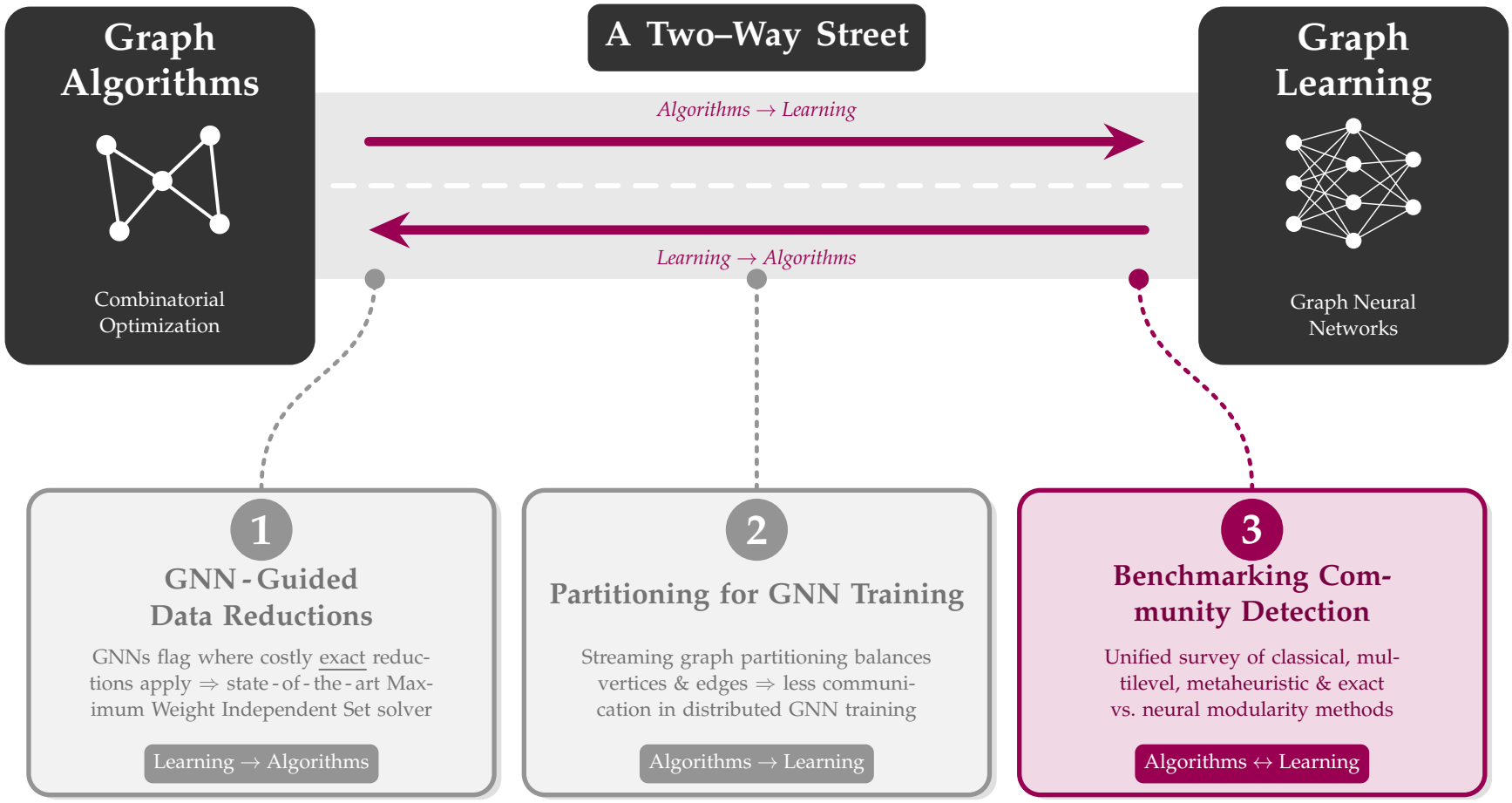
Faster Distributed GNN Training



Mean time per epoch (edge partitioning, DistGNN); red = SIGMA, blue = streaming, orange = in-memory.

up to −82%	25–62%	≈ 1.0	**1 pass**
vertex replicas vs. HDRF / Random	faster epoch time vs. streaming baselines	near-ideal vertex & edge balance	streaming ⇒ low memory, scales

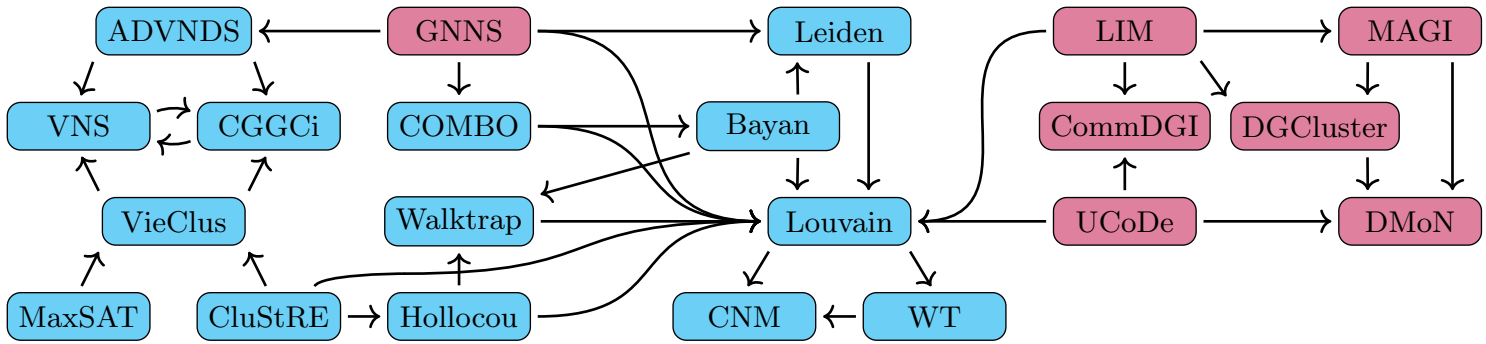
Better partitions ⇒ less communication ⇒ **faster distributed GNN training**
– competitive with in-memory METIS / KaHIP / HEP at **streaming speed**



Joint work w/ A. Chhabra, K. Langedal

Community Detection: Two Camps

Cluster a graph so that **modularity** is maximized
two camps grew up **side by side**, on different graphs with different metrics



Classical

algorithm engineering: LEIDEN, LOUVAIN, VIECLUS, CLUSTRE

Learning

GNN-based: DMON, MAGI, UCODE, DGCLUSTER

We build the **first unified benchmark**: **17 algorithms**, same graphs, same metrics (**modularity**, NMI, F1)

Three Quality Metrics

Modularity Q

are communities **denser** than pure chance?

$$\frac{1}{2m}\sum_{ij}\left(A_{ij}-\frac{k_i k_j}{2m}\right)\delta(c_i,c_j)$$

range $(-\frac{1}{2},1]$; higher = stronger structure

internal – no labels

NMI

how much does the clustering **agree** with ground-truth labels?

external – vs. truth

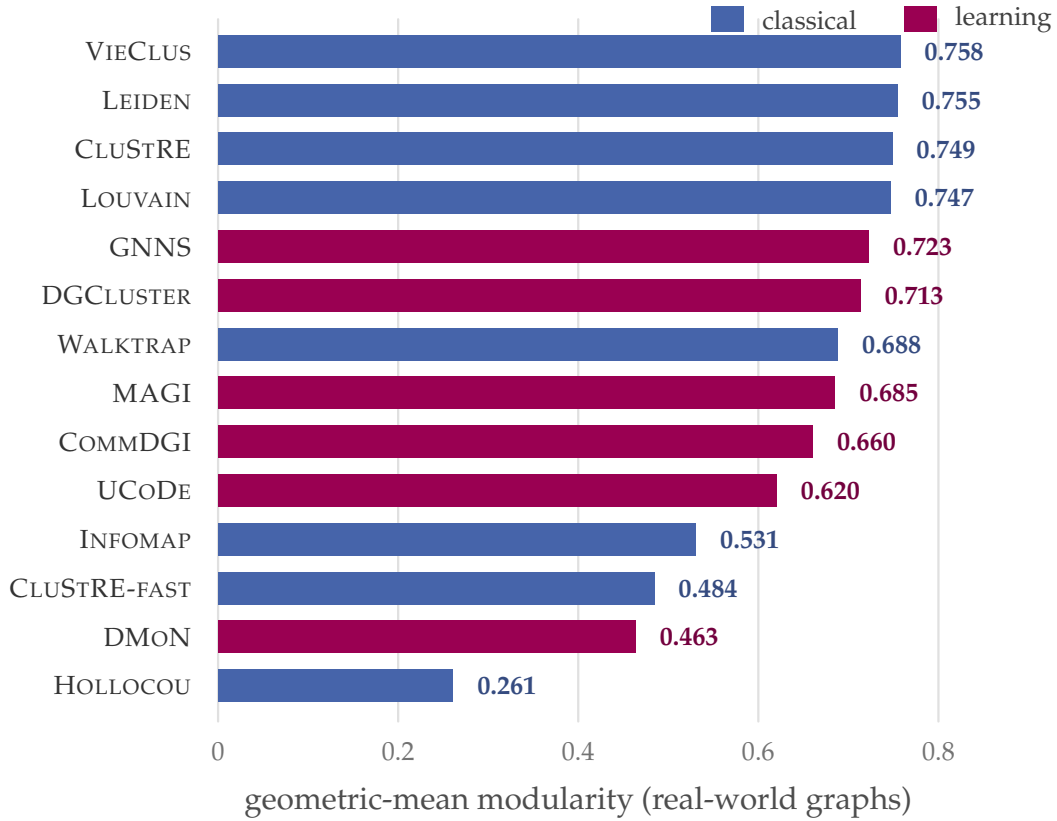
F_1

F1-Score measures a model's true accuracy on imbalanced data by calculating the harmonic mean of how many predictions it got right versus how many actual targets it missed.

external – vs. truth

Modularity is the objective methods optimize;
NMI/F_1 judge them against **known** communities

Modularity Results



Top 4 are classical

VieClus, Leiden, CluStRE, Louvain

Best GNN: GNNS

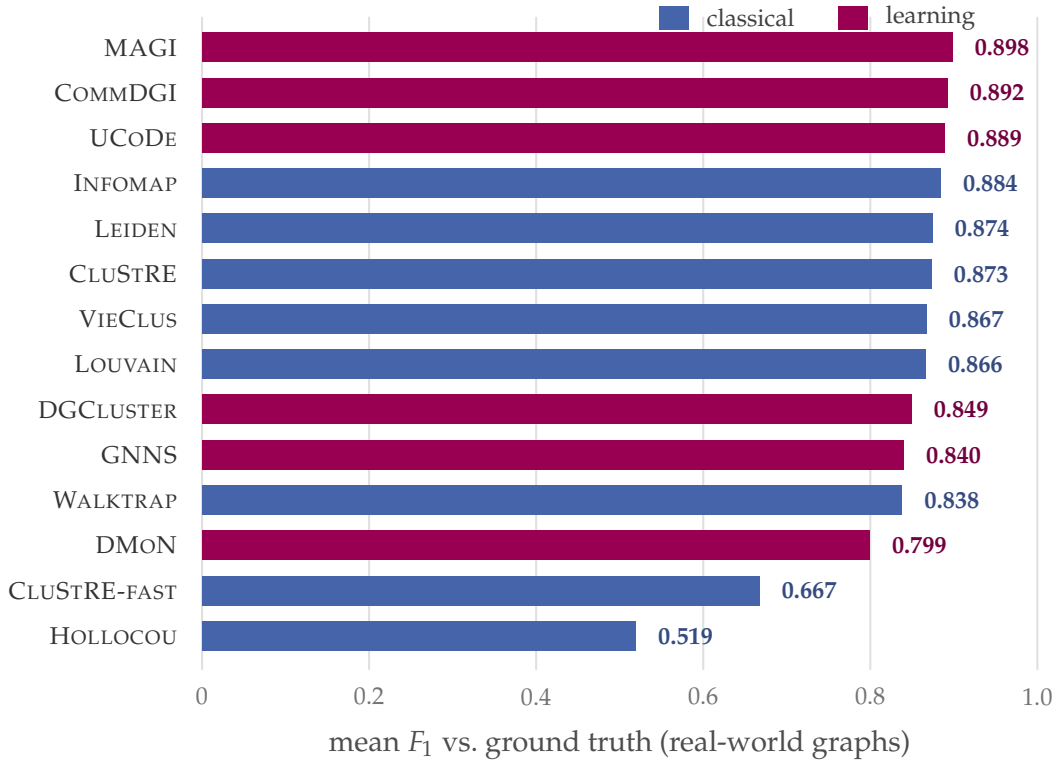
close (0.72), but far slower

Spread is large

neural quality ranges 0.46–0.72

For **pure modularity**, classical local search is still the method to beat

F1 vs. Ground Truth



Top 3 are learning
MAGI, COMMDGI, UCODE

Scores bunch up
most methods
within 0.84–0.90

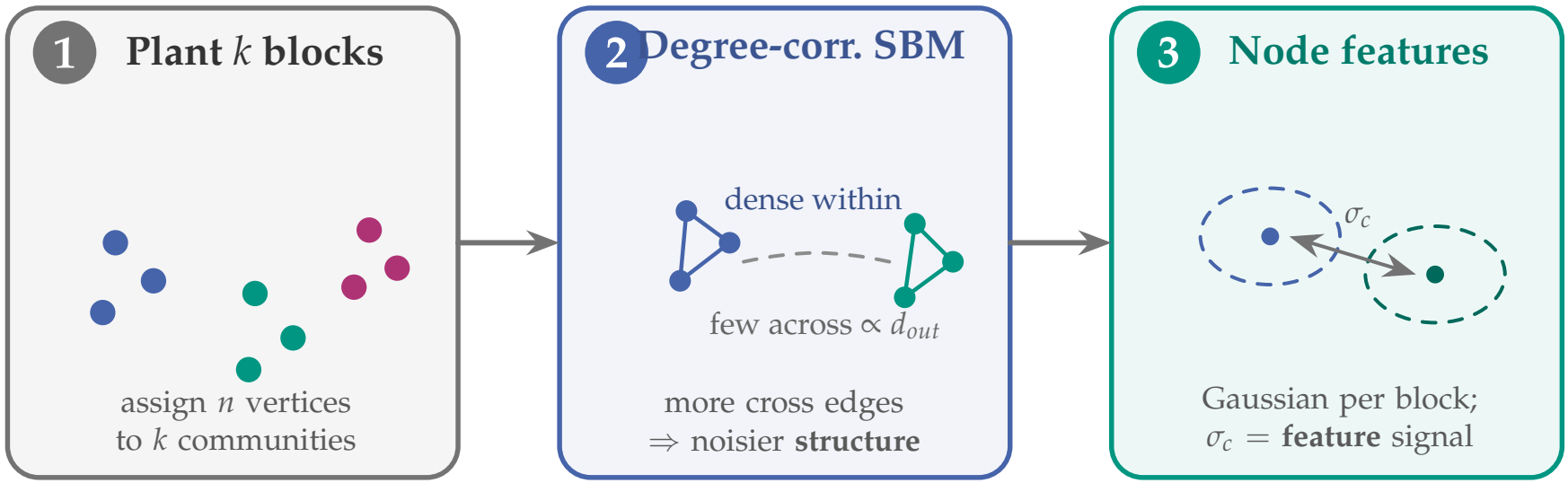
Ranking flips
vs. the modularity ordering

Change the **metric**, change the **winner**: against ground-truth labels, feature-aware GNNs lead; Note: graph algorithms do not see features

Algorithm Engineering Group
Heidelberg University

Generating Graphs: ADC-SBM

Attributed **D**egree-**C**orrected **S**tochastic **B**lock **M**odel – two knobs we can turn independently.

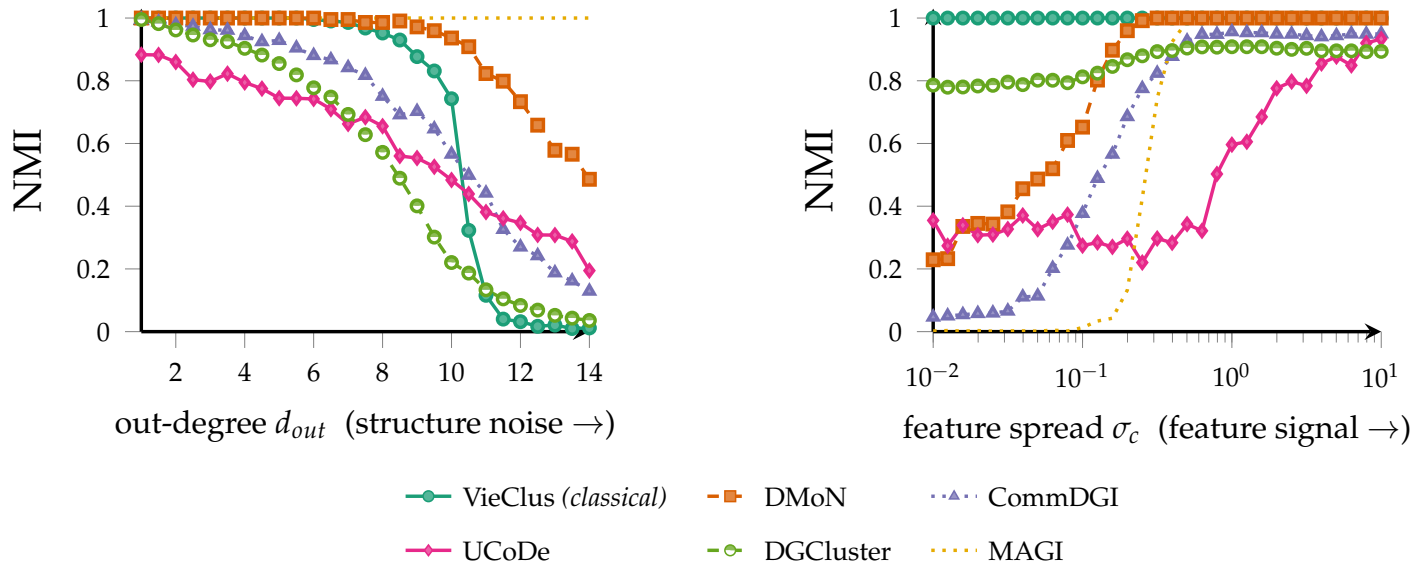


Turn d_{out} (structure noise) and σ_c (feature signal) **independently** $\Rightarrow$ isolate when features help

Algorithm Engineering Group
Heidelberg University

When Do Node Features Help?

NMI vs. ground truth as we sweep the two ADC-SBM knobs.



Structure noisy?

feature-aware MAGI stays at NMI = 1 while structure-only methods collapse

Features noisy?

neural methods drop to 0; VIECLUS (structure only) is unaffected

Features help **only** when structure is ambiguous **and** features are informative – otherwise classical methods win

Community Detection: Takeaways

What the benchmark shows

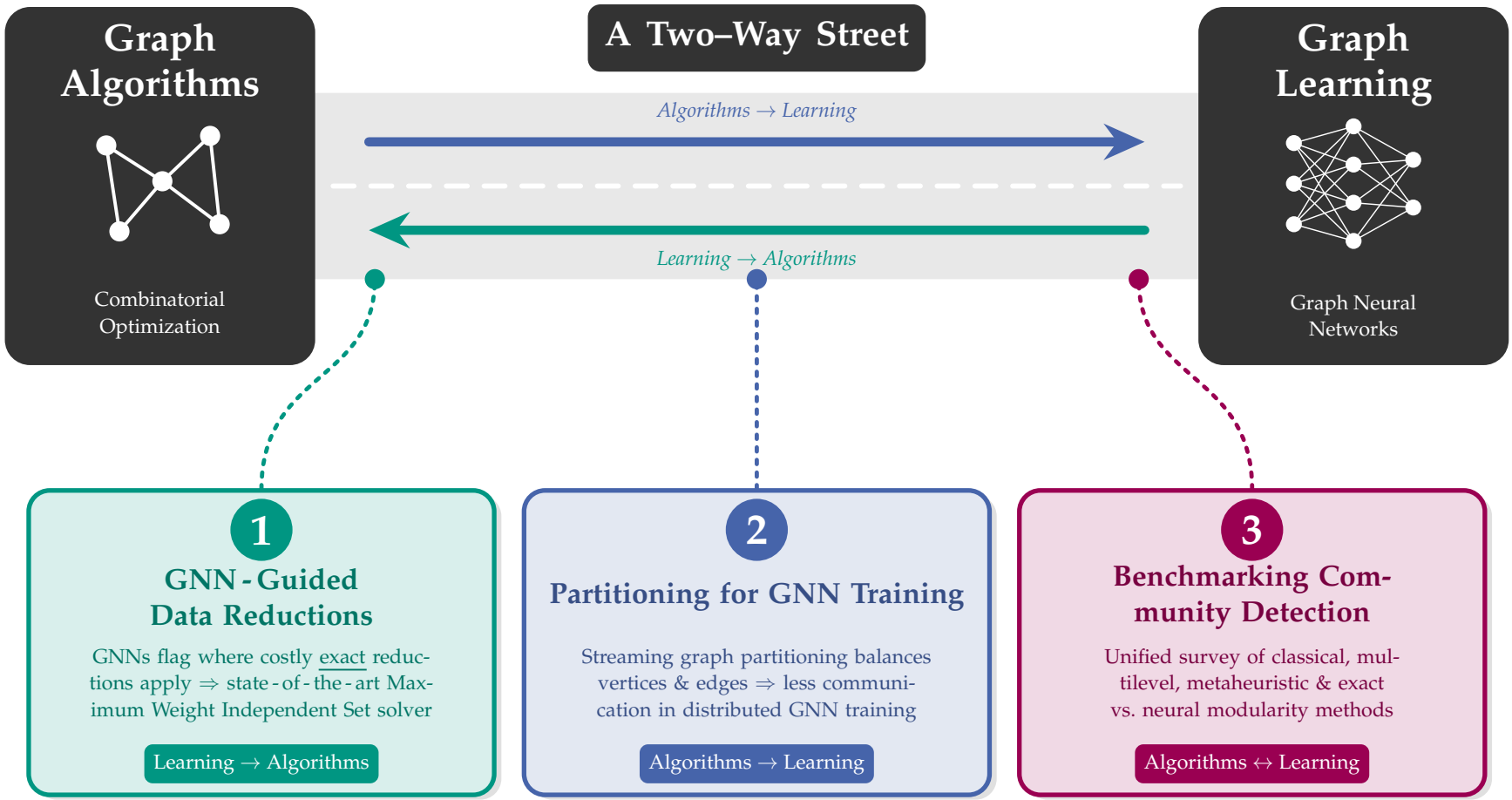
- For **pure modularity**, **classical** algorithm engineering is fast and top quality.
- **Neural** methods shine only in a narrow regime: **feature-rich, structure-poor** graphs.
- Fair comparison needs **one benchmark**: same graphs, same metrics.

The two-way street

- Classical solvers are strong **baselines** for learning to beat.
- They are also **building blocks**: preprocessing, refinement, and ground truth for training.
- Learning, in turn, **guides** classical algorithms (cf. GNN-screened reductions).

The bigger picture: the two camps still develop largely **in parallel**. Community detection is just **one example** – new ML methods are often **not benchmarked against state-of-the-art** classical baselines.

Bridging them is a genuine **two-way street**


Graph Algorithms
Combinatorial Optimization
A Two–Way Street
Algorithms → Learning
Learning → Algorithms
Graph Learning
Graph Neural Networks
1
GNN-Guided Data Reductions
GNNs flag where costly exact reductions apply ⇒ state-of-the-art Maximum Weight Independent Set solver
Learning → Algorithms
2
Partitioning for GNN Training
Streaming graph partitioning balances vertices & edges ⇒ less communication in distributed GNN training
Algorithms → Learning
3
Benchmarking Community Detection
Unified survey of classical, multilevel, metaheuristic & exact vs. neural modularity methods
Algorithms ↔ Learning