2025 Fall CSE5025

Combinatorial Optimization

组合优化

Instructor: 刘晟材

Lecture 7-2: Advanced Topics:

Learning to Optimize

Agenda for Today's Lecture

In this lecture, we will focus on

- The basic idea of learning to optimize (L2O)
- Learning to branch
- Neural combinatorial optimization (NCO)

Learning objectives for this lecture

- Understanding the strengths and weakness of L2O
- Understanding the basic ideas of learning-to-branch and NCO

References used in This Lecture

[1] Bengio et al., Machine learning for combinatorial optimization: A methodological tour d'horizon. EJOR 2021.

[2] Khalil et al., Learning to Branch in Mixed Integer Programming. AAAI 2016.

[3] Vinyals et al., Pointer Networks. NeurIPS 2015.

[4] Kool et al., Attention, Learn to Solve Routing Problems! ICLR 2019.

[5] Liu et al., How Good Is Neural Combinatorial Optimization? A Systematic Evaluation on the Traveling Salesman Problem. IEEE CIM 2023.

The Classical CO Worldview

Design Philosophy: Algorithms are designed by human experts (e.g., Karp).

Performance Metric:

- Worst-Case Runtime Complexity, e.g., $O(2^n)$, to find optimal solutions
- Worst-Case Runtime Complexity, e.g., $O(n^2)$, to achieve ρ-approximation

Requirement: The algorithm must perform acceptably well on **any** valid input problem instance, including extremely rare corner cases and "adversarial" ones.

Example:

- Double-Tree Algorithm with $O(n^2)$ runtime achieves 2-approximation for metric TSP.
- Branch & Bound with $O(2^n)$ runtime can solve any BIP instance to optimality.

The Gap: Theory vs. Reality

The Reality: In real-world applications (logistics, chip design, routing), we rarely face "arbitrary" or "adversarial" instances.

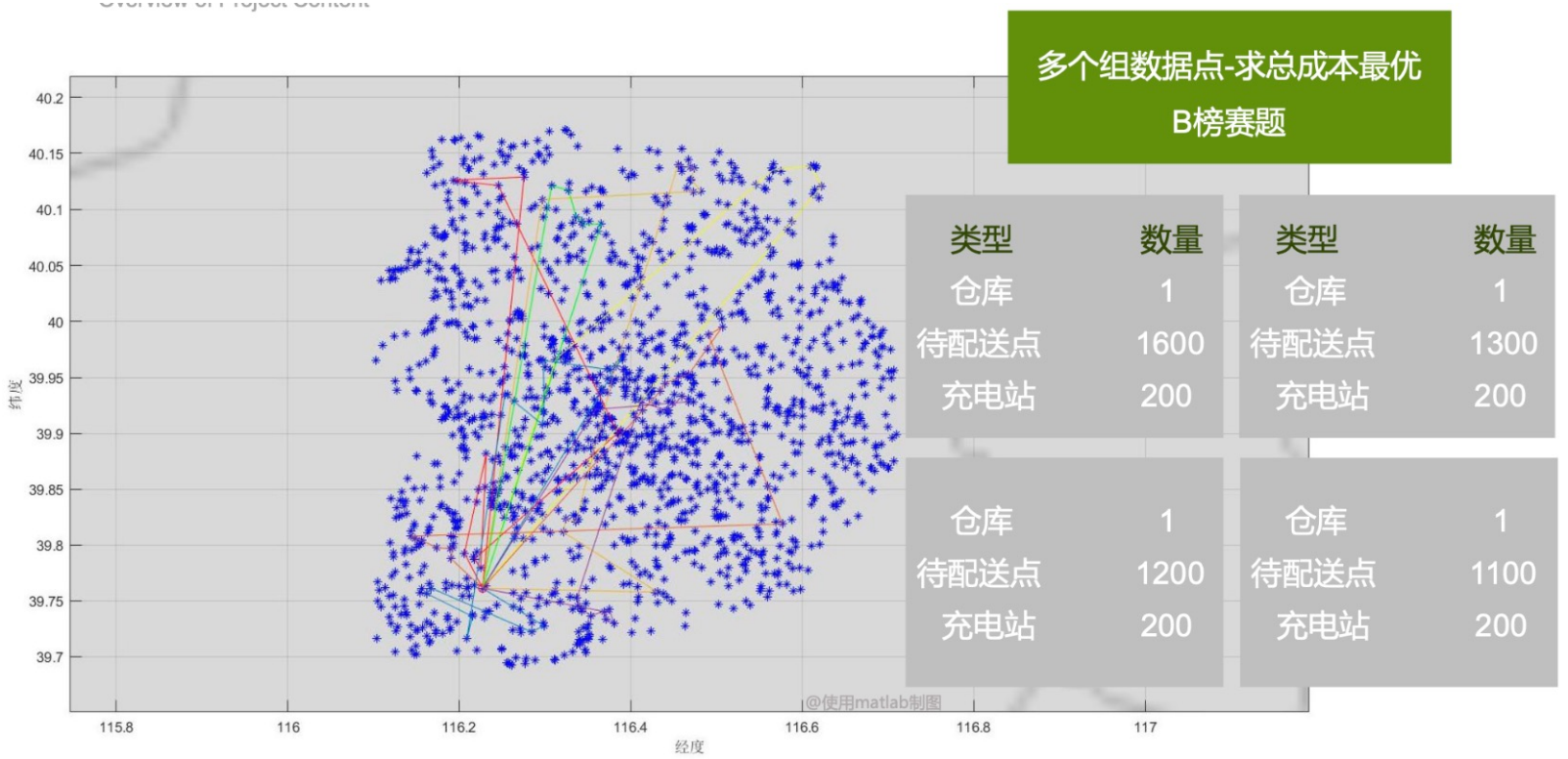
The Distribution Hypothesis:

- Problem instances come from a specific, often unknown, probability distribution $\mathcal{D}$.

The Inefficiency:

- Classical algorithms could be "over-engineered" to handle worst-cases that might never happen in $\mathcal{D}$.
- They do not exploit the structure or patterns inherent in $\mathcal{D}$.

An Example of Reality



VRP instances for JD Logistics in Beijing always look similar (same road network, similar customer density) day after day.

Motivation Behind L2O: The Average Case

Design Goal Shift: Instead of considering the worst-case performance, we want to minimize the **expected cost** (or maximize **expected value**) achieved by the algorithm over distribution $\mathcal{D}$.

Analogy:

- Classical: A suit of armor designed to survive a nuclear blast (Heavy, slow, overkill for daily life).
- L2O: A raincoat designed for Shenzhen weather (Light, specialized, efficient for the actual environment).

Key Insight: If we can "learn/exploit" the structure of $\mathcal{D}$, we can specialize our algorithm to be better or faster on those specific types of instances.

Motivation Behind L2O: Computational Bottleneck

Many classical algorithms have components that are mathematically strong but computationally expensive.

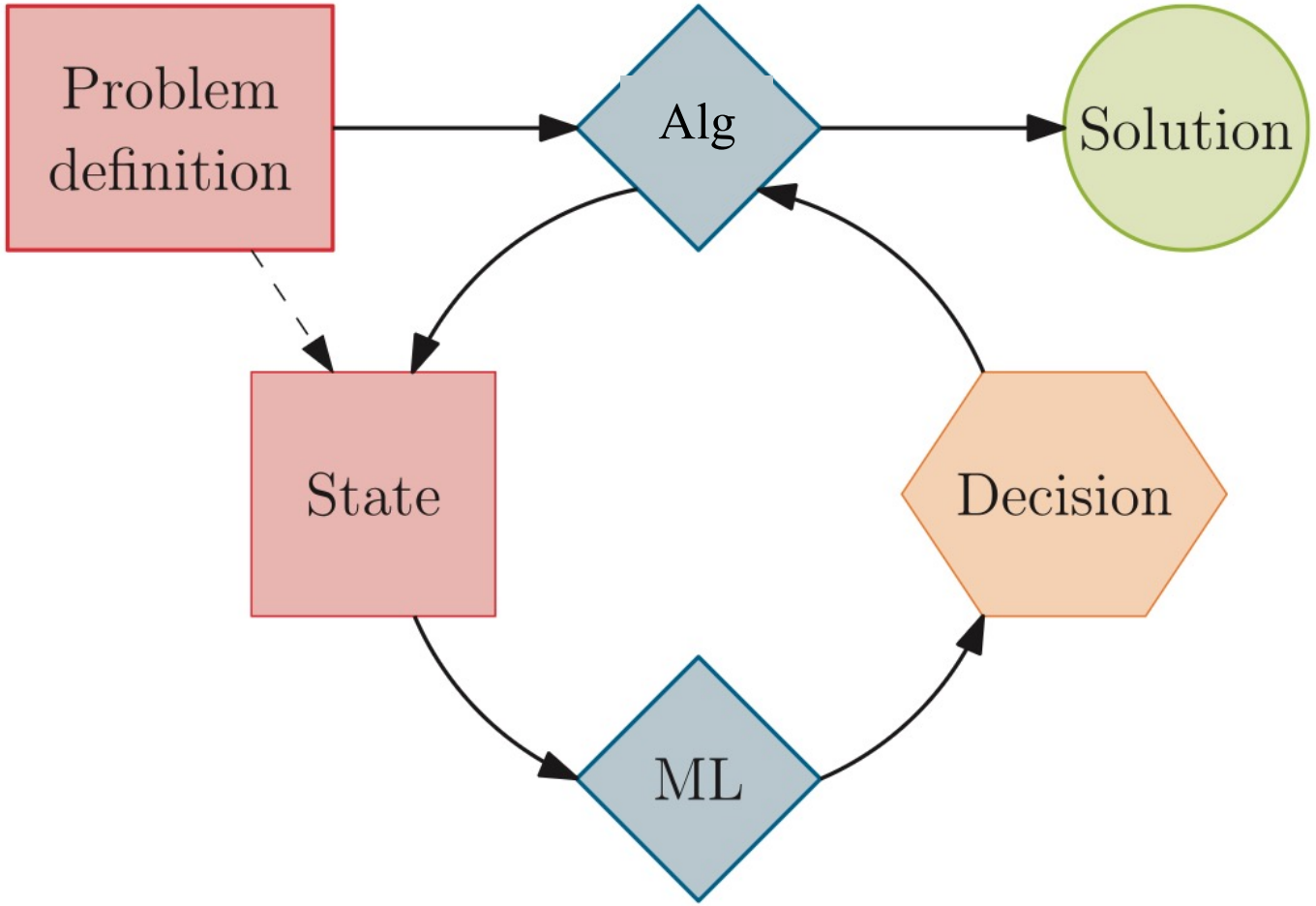
The Dilemma:

- Option 1: A fast, simple heuristic (e.g., random branching in B&B) → Poor decisions.
- Option 2: A complex, expert-designed heuristic (e.g., strong branching in B&B) → excellent decisions, but extremely slow.

The Learning Opportunity:

- Can we learn a model (e.g., neural network) to **approximate** the expert heuristic?
  - **Input**: Current state.
  - **Output**: The decision the expert would have made.
  - **Benefit**: The model inference is much faster than the heavy computation.

One Possible Implementation



Learn to Approximate: The machine learning (ML) model is used to augment an optimization algorithm (Alg) with valuable pieces of information.

Motivation Behind L2O: Replacing Human Intuition

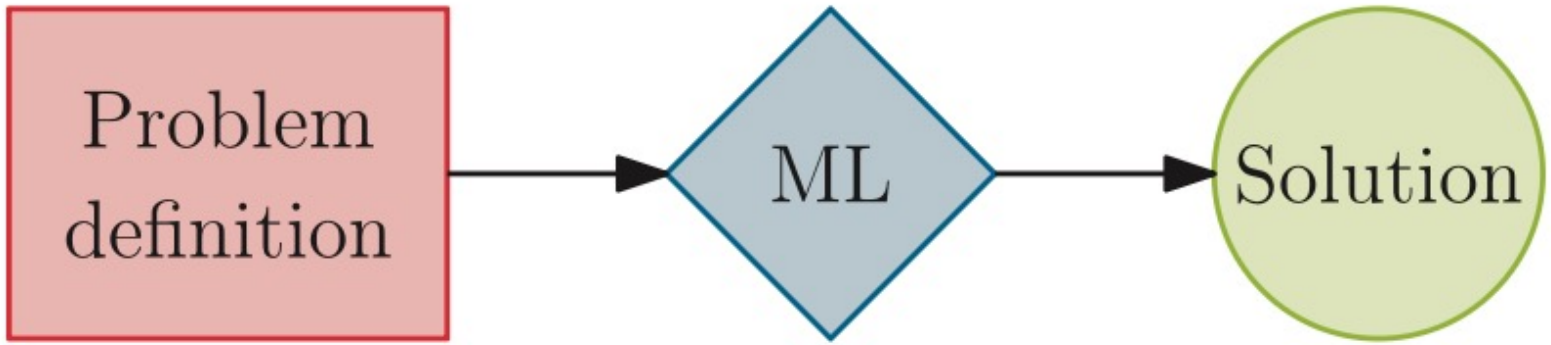
The Limit of Expert Knowledge:

- Designing heuristics (e.g., insertion heuristics for TSP) relies on human intuition.
- Humans are biased and cannot explore the infinite space of algorithmic decisions.
- Existing heuristics (like pseudo-cost branching) are “hand-crafted”.

The Learning Goal:

- Explore the space of algorithmic decisions automatically.
- Learn a policy (behavior) from experience.
- Hope: The machine might discover a strategy that is non-intuitive to humans but performs better on the distribution $\mathcal{D}$.

One Possible Implementation



Learn to Solve: Machine learning acts alone to provide a solution to the problem.

Formal Definition of L2O

The General Mathematical Framework:

- Problem Instance: $x \in \mathcal{X}$, e.g., a graph, a matrix.
- Distribution: Problem instances are sampled from $\mathcal{D}$, i.e., $x \sim \mathcal{D}$.
- Objective function specified by x: C_x, e.g., cost to minimize.
- Solution to instance x: $y \in \mathcal{Y}$, e.g., a TSP tour, a selection of items.
- Algorithm (Optimizer): A mapping $A: \mathcal{X} \to \mathcal{Y}$, parameterized by θ, e.g., weights of a NN, parameters of a heuristic, and so on.

The Learning Objective

Goal: Find parameters θ^* that minimizes the expected cost over the distribution.

$$\theta^* = \arg\min_\theta \mathbb{E}_{x\sim\mathcal{D}}\left[\mathcal{L}\big(A(x;\theta)\big)\right]$$

The Loss Function $\mathcal{L}$:

- **Direct Cost**: $\mathcal{L} = C_x(A(x;\theta))$
- **Optimality Gap**: $\mathcal{L} = C_x(A(x;\theta)) - C_x^*$, where C_x^* is the known optimal solution to x
- **Imitation Loss**: Cross-entropy between the algorithm's decision and an expert's decision.

Generalization in L2O

The **generalization** of the learned optimizers is critical.

Unlike classical algorithms, the learned optimizers must be **Trained** and **Tested**.

- **Training Set:** $\{x_1, x_2, \dots, x_N\} \sim \mathcal{D}$: used to optimize θ.
- **Test Set**: $\{x'_1, x'_2, \dots, x'_M\} \sim \mathcal{D}$: used to evaluate performance.

Research Targets:

- **In-Distribution Generalization**: Does it work on unseen graphs of the same size/type? (Standard L2O goal).
- **Out-of-Distribution (OOD):** Does a model trained on metric TSP work on general TSP? (Very hard, open research problem).

Representative Approaches in L2O

In this lecture, we will introduce two representative approaches in L2O.

Learning to Branch:

- Keep the exact solver backbone (Branch & Bound).
- Use learning models to replace specific components (e.g., branching variable selection).
- Goal: Fast approximation.

Neural combinatorial optimization:

- Replace the algorithm entirely with a deep neural network.
- Input Instance(Graph) → NN → Output Solution.
- Goal: Extremely fast heuristics, discovering new policies.

A Recap on B&B

1) Initialize a "global incumbent" $Z_{INC} = -\infty$.

2) Add the root LP relaxation to a "**set**" of problems.

3) While **set** is not empty:

a. Select a problem P from the set.

b. Solve its LP relaxation, get Z_{LP}^{P} and $\boldsymbol{x}^{*}$.

c. Pruning 1 (by Bound): If $Z_{LP}^{P} \leq Z_{INC}$, prune this branch, and goto step 3).

d. Pruning 2 (Infeasible): If $Z_{LP}^{P} = -\infty$, prune this branch and goto step 3).

e. If $\boldsymbol{x}^{*}$ are all integers, we have found a feasible integer solution. Check if $Z_{LP}^{P} > Z_{INC}$ and update the incumbent accordingly. Goto Step 3).

f. Branching: If $\boldsymbol{x}^{*}$ is fractional (e.g., $x_j =$ fractional value a). Create two new subproblems with added constraints: $x_j \leq \lfloor a \rfloor$ and $x_j \geq \lceil a \rceil$, respectively. Add both to the set.

4) Return Z_{INC}.

The Branching Bottleneck in B&B

1) Initialize a "global incumbent" $Z_{INC} = -\infty$.

2) Add the root LP relaxation to a "**set**" of problems.

3) While **set** is not empty:

a. Select a problem P from the set.

b. Solve its LP relaxation, get Z_{LP}^{P} and $\boldsymbol{x}^{*}$.

c. Pruning 1 (by Bound): If $Z_{LP}^{P} \leq Z_{INC}$, prune this branch, and goto step 3).

d. Pruning 2 (Infeasible): If $Z_{LP}^{P} = -\infty$, prune this branch and goto step 3).

e. If $\boldsymbol{x}^{*}$ are all integers, we have found a feasible integer solution. Check if $Z_{LP}^{P} > Z_{INC}$ and update the incumbent accordingly. Goto Step 3).

f. Branching: If $\boldsymbol{x}^{*}$ is fractional (e.g., x_j = fractional value a). Create two new subproblems with added constraints: $x_j \leq \lfloor a \rfloor$ and $x_j \geq \lceil a \rceil$, respectively. Add both to the set.

4) Return Z_{INC}.

What if we have multiple factional values here

The Branching Bottleneck

The efficiency of the B&B algorithm relies heavily on selecting the best integer variable to branch on at each node.

- **Problem**: Choosing a good variable drastically reduces the search tree size. Conversely, a naive strategy degrades performance significantly.
- The **gold standard is strong branching**: it exhaustively tests candidate variables at each node, calculating the best one based on how much it improves the bound.
- The **Dilemma**: While strong branching yields significantly smaller search trees , it is computationally expensive due to solving many LP problems per node. The time spent often overshadows the time saved.

Goal: Achieve the node-efficiency of strong branching while maintaining the time-efficiency of faster, simpler heuristics.

Mimicking the Expert (Strong Branching)

The proposed framework aims to replace the time-consuming strong branching with a fast, learned surrogate function.

Data Collection Phase:

- The B&B algorithm initially runs with strong branching for a fixed number of nodes (e.g., 500 nodes). At **each node during this phase**, the actual strong branching scores are computed for candidate variables.
- **Features of variables** (structural role, historical performance, etc.) are extracted.

Labeling:

- The scores are used to generate a binary label ($y_j^i \in \{0, 1\}$) for each variable x_j. A variable is labeled "1" if its score is close to the maximum score. This captures the fact that multiple variables may be good branching candidates.

The ML Model: Learning a Ranking Function

The problem of variable selection is framed as a **learning-to-rank problem**.

- **Objective**: Learn a function that assigns a score s_j to each variable.
- **Loss Function (Pairwise Ranking)**: The model is trained to minimize violations of pairwise ordering constraints:
  - *Constraint*: If strong branching ranks x_j better than x_k (i.e., $y_j = 1$, $y_k = 0$), the learned score must satisfy $s_j > s_k$
  - *Rationale*: **The score itself is not important; the ranking order is**.
- Model: The framework often utilizes support vector machines (SVM) or other ranking models.

Empirical Results

After the training phase, the learned function f replaces strong heuristic in the same B&B tree and other B&B trees when **solving instances belong to the same distribution.**

Inference Complexity: dominated by feature computation (polynomial) and *is significantly faster than* solving multiple LPs.

Key Results (Empirical):

- Significantly smaller search trees than other heuristics, e.g., pseudo-cost heuristic.
- Competitive with commercial solver's default strategy.
- The method is adaptive and instance-specific, as the feature weights are learned based on the characteristics encountered during the search.

Neural Combinatorial Optimization (NCO)

Goal: Solve the problem end-to-end using NNs.

- Input: problem instance (typically a graph).
- Output: A solution (e.g., sequence of cities).

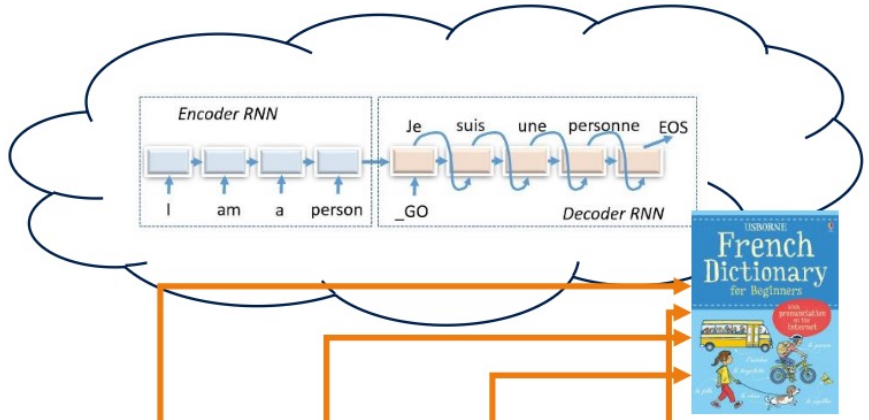
No Solver Backbone: we are not using B&B. We are constructing the solution step-by-step using a learned policy.

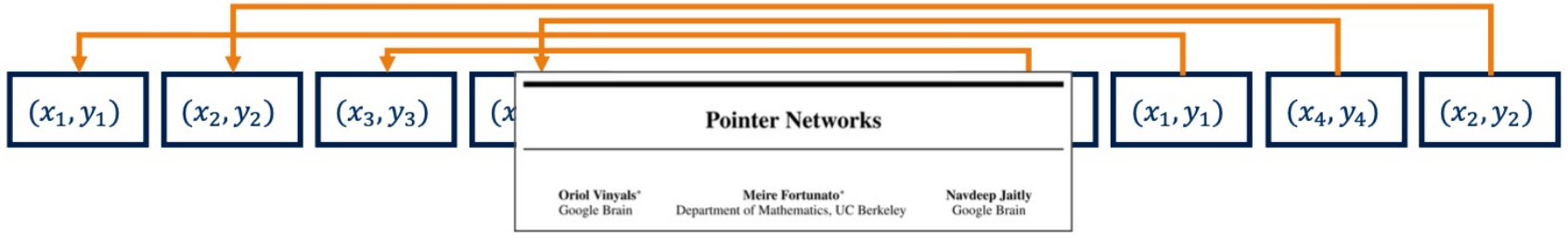
Constructive approach, like the constructive heuristic:

- Start empty.
- Loop: NN predicts the next item to add or the next city to visit

Similarity Between Optimization and Translation

‘Translate’ problem into solution





How does that work?

Instance $s =$
$((x_1, y_1), (x_2, y_2), \dots, (x_n, y_n))$

Model $p_\theta(\pi_t | s, \pi_{<t})=$
p_θ(next node | partial tour)

Solution $\boldsymbol{\pi} = (\pi_1, \pi_2, \dots)$
with length $L(\boldsymbol{\pi})$

Sample $\pi_1 \sim p_\theta(\pi_1 | s)$

Sample $\pi_2 \sim p_\theta(\pi_2 | s, \pi_1)$

Sample $\pi_t \sim p_\theta(\pi_t | s, \pi_{<t})$

Randomized algorithm
with expected cost:

$$E_{p_\theta(\boldsymbol{\pi}|s)}[L(\boldsymbol{\pi})]$$

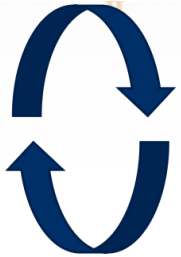
How to optimize $\boldsymbol{\theta}$?

NEURAL COMBINATORIAL OPTIMIZATION WITH REINFORCEMENT LEARNING

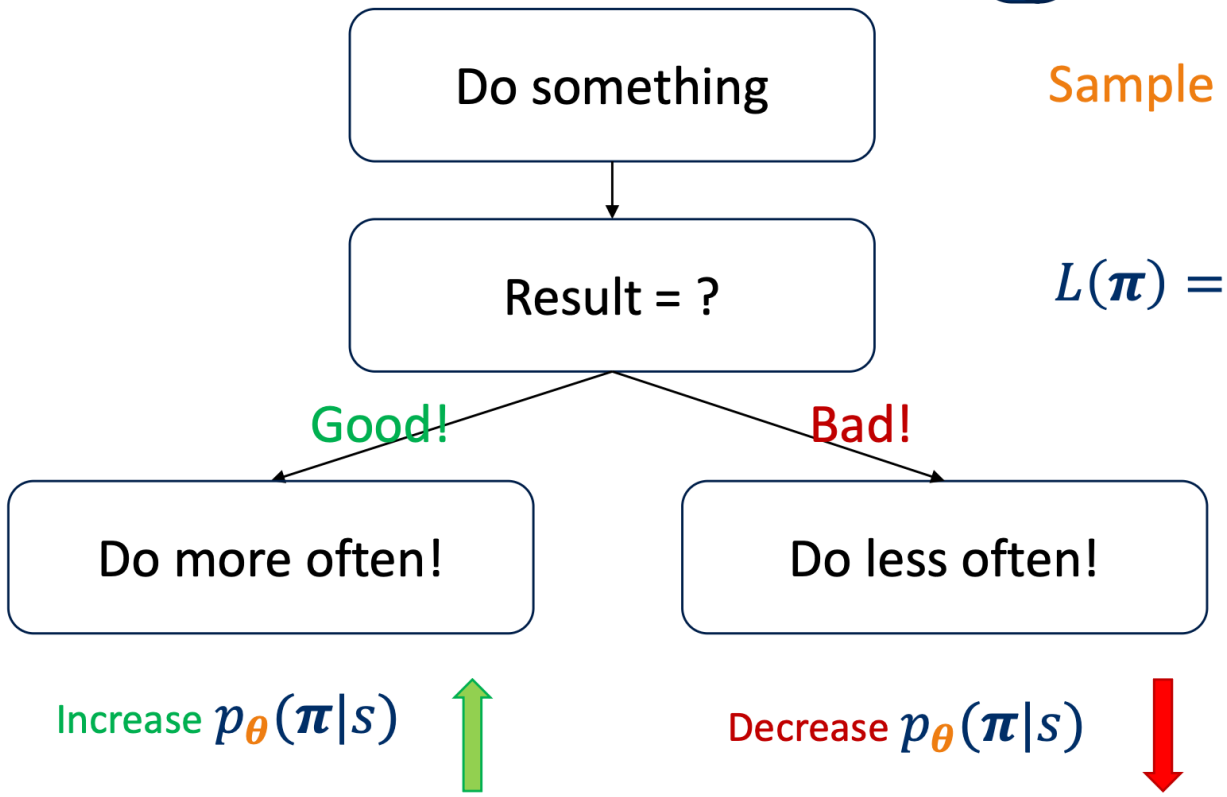
Irwan Bello*, Hieu Pham*, Quoc V. Le, Mohammad Norouzi, Samy Bengio
Google Brain
{ibello,hyhieu,qvl,mnorouzi,bengio}@google.com

How to optimize θ: Reinforcement Learning

REINFORCE (for dummies)



Repeat



Sample $\pi \sim p_\theta(\cdot|s)$

$L(\pi) = 7.43$

We need a *baseline* to compare against: *rollout* earlier model

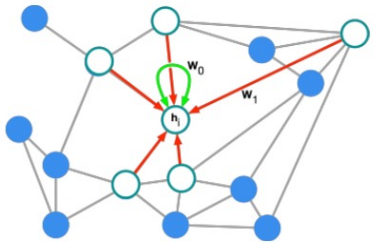
$\pi^{bl} \sim p_{\theta^{bl}}(\cdot|s)$ **(greedy!)**

$L(\pi^{bl}) = 6.89$

Model Architecture

$$p_{\boldsymbol{\theta}}(\pi_t | s, \pi_{<t})$$

Graph convolutions



+

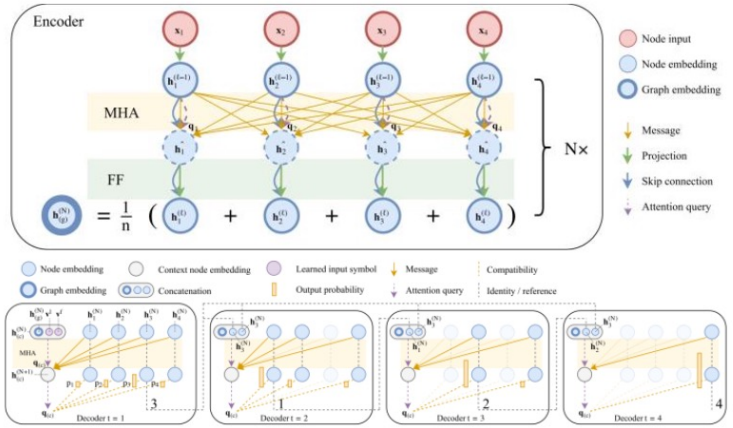
Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Read the paper…

ATTENTION, LEARN TO SOLVE ROUTING PROBLEMS!

Wouter Kool University of Amsterdam w.w.m.kool@uva.nl
Herke van Hoof University of Amsterdam h.c.vanhoof@uva.nl
Max Welling University of Amsterdam m.welling@uva.nl



Is NCO Magic?

Question: If NCO (trained NNs) solves TSP in polynomial time, so $P = NP$?

The Answer is NO:

- **Definition of *P***: An algorithm must find the **exact optimal** solution for **ALL** instances in polynomial time.
- The NCO Reality:
  - **Time**: Inference is indeed polynomial.
  - **Quality**: The output is probabilistic, not guaranteed.
    - The network outputs a distribution: $P(\pi|\ \text{Graph})$
    - It acts like a high-quality Heuristic.
- **Conclusion**: NCO replaces "Worst-case Guarantee" with "Average-case Performance". It bypasses NP-hardness by relaxing the requirement of Optimality.

Difficulty: Combinatorial Chaos

It is hard for NNs to learn the exact optimizer perfectly.

- NNs excel at learning statistical correlations and smooth functions. E.g., “Closer cities are more likely to be connected.” (A smooth, continuous mapping).
- Combinatorial Optimization (e.g., TSP):
  - The mapping $x \to y^*$ (optimal solution) is **Highly Non-Linear** and **Discontinuous**.
  - The **Butterfly Effect**: Moving a single city coordinate slightly can trigger a global restructuring of the optimal tour.
- The **Conflict**:
  - A fixed-parameter NN tries to fit a “smooth curve” over a “chaotic, rugged landscape”.

Difficulty: Capacity Bottleneck (1)

Could a NN theoretically learn the exact solver?

- Universal Approximation Theorem: A NN can approximate any function.
  - **The Catch:** It requires an arbitrarily large number of neurons (width).
- The **Capacity Argument**:
  - The solution space of TSP grows as $O(n!)$.
  - A standard NCO model typically has **polynomial or fixed** parameters
  - Information theory: A polynomial-sized model has limited "memory" (VC Dimension). It cannot "memorize" or encode the complex decision boundaries for the exponentially growing number of graph permutations.

Difficulty: Capacity Bottleneck (2)

- The **Paradox**:
  - To map every instance to its exact optimum, the network size would likely need to grow **Exponentially**.
  - If the network is exponential in size, forward inference becomes **Exponential Time**.
  - We are back to: No Polynomial Exact Optimizer.

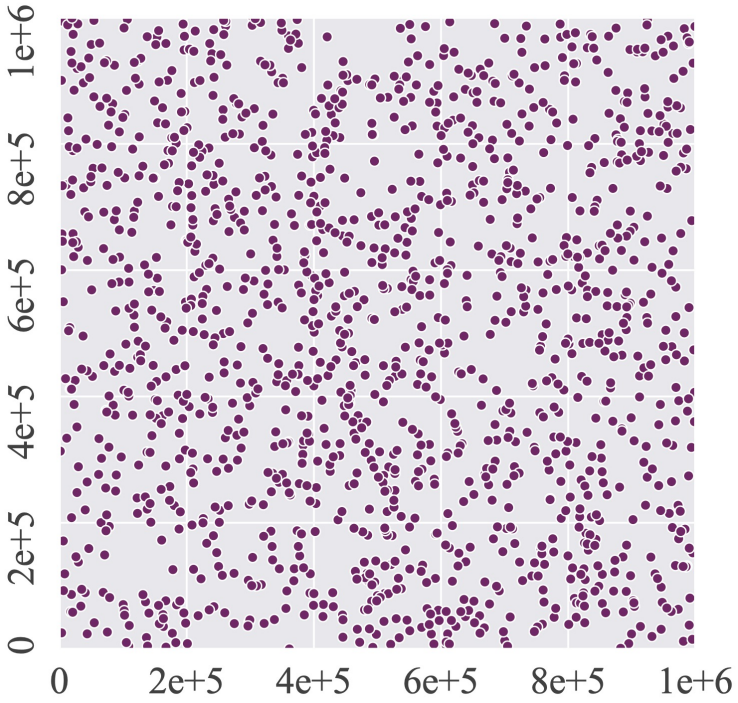
Current Performance of NCO on TSP

Classical algorithms/solvers still significantly **outperform** NCO solvers in obtaining **high-quality solutions, regardless of problem types and sizes.**

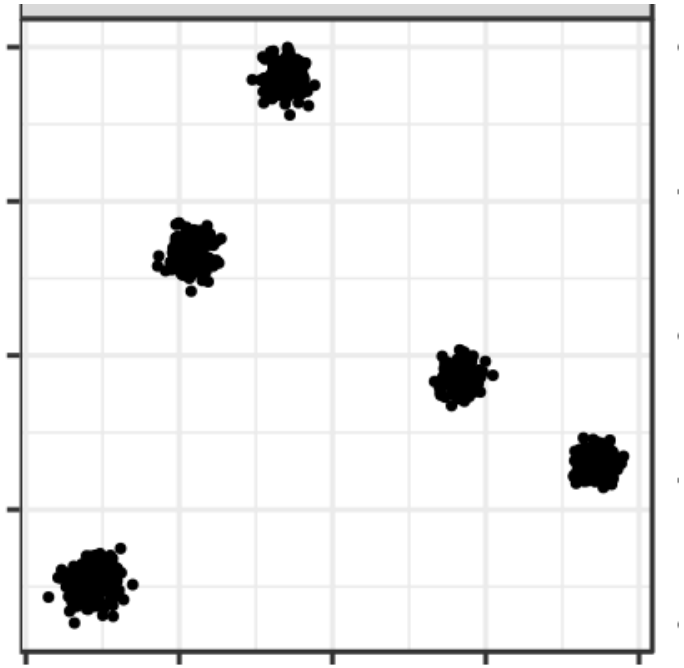
NCO solvers exhibit superior time and energy efficiency (with GPUs) for small-scale problem instances.

NCO solvers exhibit severe performance degradation when testing instances are significantly different from training instances.

Two TSP Distributions



rue instances



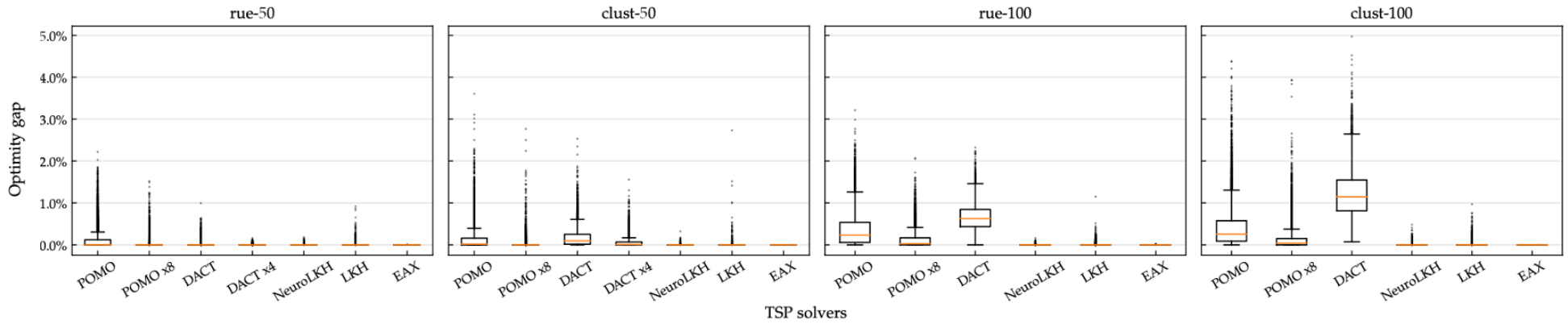
clu instances

Results on Small-Scale TSP

TABLE 1

Testing results of experiment 1 which is designed to assess the effectiveness, efficiency and stability of the solvers on small-size problem instances. The results are presented in terms of the average optimum gap, the total computation time and energy consumed by the solver. For each metric, the best performance is indicated in grey. Note for DACT with instance augmentation mechanism, its results on *rue*-100 and *clust*-100 are missing because it runs prohibitively long to solve the testing instances.

Method	rue-50			clust-50			rue-100			clust-100		
	Gap (%) ± std (%)	Time (s)	Energy (J)	Gap (%) ± std (%)	Time (s)	Energy (J)	Gap (%) ± std (%)	Time (s)	Energy (J)	Gap (%) ± std (%)	Time (s)	Energy (J)
POMO, no aug.	0.1185 ± 0.0000	2.57	290.83	0.1353 ± 0.0000	2.58	292.14	0.3646 ± 0.0000	12.59	2588.90	0.4318 ± 0.0000	12.83	2675.18
POMO, ×8 aug.	0.0228 ± 0.0000	16.98	3361.87	0.0213 ± 0.0000	17.04	4193.39	0.1278 ± 0.0000	87.73	25873.38	0.1405 ± 0.0000	93.08	27299.18
DACT	0.0167 ± 0.0291	1991.49	402635.40	0.1770 ± 0.1117	1921.99	393994.52	0.6596 ± 0.5216	6141.72	1269009.13	1.2220 ± 0.4773	6517.2110	1390740.24
DACT, ×4 aug.	0.0006 ± 0.0013	8534.42	1742933.29	0.0576 ± 0.0390	8735.50	1676140.80		-			-	
NeuroLKH	0.0003 ± 0.0003	34.22	4006.11	0.0004 ± 0.0007	131.92	12698.78	0.0004 ± 0.0005	74.90	9819.77	0.0021 ± 0.0031	309.75	29397.05
LKH	0.0035 ± 0.0035	291.22	10458.70	0.0022 ± 0.0018	257.95	9512.24	0.0044 ± 0.0048	313.73	12868.38	0.0048 ± 0.0040	340.58	14264.64
EAX	0.0000 ± 0.0000	343.79	15089.39	0.0000 ± 0.0000	321.06	12541.47	0.0000 ± 0.0000	598.34	35145.37	0.0000 ± 0.0000	561.41	27893.49



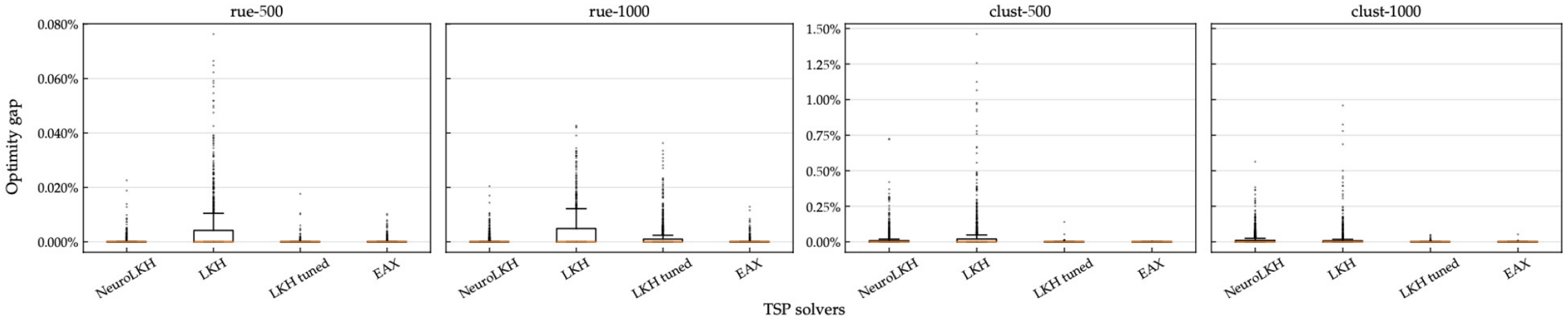
- Traditional solvers consistently obtain better solutions than NCO solvers
- POMO exhibits excellent efficiency in terms of both runtime and energy

Results on Medium-Scale TSP

TABLE 2

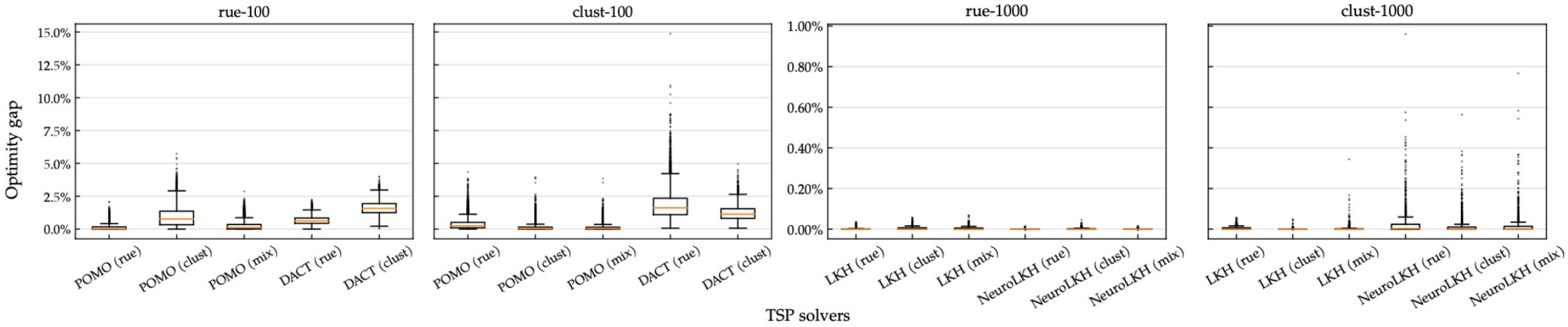
Testing results of experiment 2 which is designed to assess the effectiveness, efficiency and stability of the solvers on large-size problem instances. The results are presented in terms of the average optimum gap, the total computation time and energy consumed by the solver. For each metric, the best performance is indicated in grey. Note in this experiment POMO and DACT are not tested due to their poor scalability.

Method	rue-500			clust-500			rue-1000			clust-1000		
	Gap (‰) ± std (‰)	Time (s)	Energy (J)	Gap (‰) ± std (‰)	Time (s)	Energy (J)	Gap (‰) ± std (‰)	Time (s)	Energy (J)	Gap (‰) ± std (‰)	Time (s)	Energy (J)
NeuroLKH	0.0273 ± 0.0441	242.71	21455.69	1.8080 ± 2.5469	2695.06	171826.14	0.0417 ± 0.0510	728.41	58569.96	1.8599 ± 2.2236	5094.77	334771.95
LKH	0.4356 ± 0.5084	143.36	17077.63	4.2779 ± 3.6307	313.80	35213.13	0.3620 ± 0.3624	413.43	48867.49	1.8761 ± 1.2798	859.39	100514.45
LKH (tuned)	0.0086 ± 0.0165	162.63	19087.78	0.0345 ± 0.0398	279.80	32685.72	0.1732 ± 0.1770	406.16	52252.97	0.0460 ± 0.0537	522.41	66511.40
EAX	0.0140 ± 0.0291	269.41	32370.64	0.0006 ± 0.0012	209.09	24456.69	0.0182 ± 0.0242	620.30	75168.05	0.0086 ± 0.0122	631.68	75222.30



- Traditional solvers consistently obtain better solutions than NCO solvers
- Parameter tuning can improve LKH in solution quality, runtime and energy

Results on OOD Generalizations



- The performance of a NCO solver would be degraded over OOD
- Even if a mixed training set is used, the learned/trained solver still cannot achieve the best possible performance

Results on Problem Sizes

TABLE 3
Testing results of experiment 4 which is designed to assess the learned solvers' generalization ability over different problem sizes. The results are presented in terms of the average optimum gap.

Method (training set)	rue-100 Gap (%) ± std (%)	Method (training set)	clust-100 Gap (%) ± std (%)
POMO (rue-50)	0.6703 ± 0.0000	POMO (clust-50)	0.6829 ± 0.0000
POMO (rue-100)	0.1278 ± 0.0000	POMO (clust-100)	0.1405 ± 0.0000
DACT (rue-50)	27.5437 ± 31.4449	DACT (clust-50)	21.4630 ± 5.3292
DACT (rue-100)	0.6596 ± 0.5216	DACT (clust-100)	1.2220 ± 0.4773

- when applying NCO solvers on the testing instances having larger sizes than the training instances, the performance would be significantly degraded

NCO as a Grandmaster's Intuition, not a Calculator

The Neural Network (Intuition):

- Glances at the board (Graph).
- Instantly suggests a “very strong move” based on pattern recognition.
- Effective for: Average cases, fast responses.

The Search (Calculation):

- To find real-high quality solutions, one must calculate variations.
- Necessary for: Corner cases, proofs of optimality.

One possible future direction:

$$\text{NCO} + \text{Search} \rightarrow \text{State of the art}$$

A Revisit On L2O

Let's step back from NNs. L2O is fundamentally a **Meta-Optimization** problem over a distribution of instances.

$$\theta^* = \arg\min_\theta \mathbb{E}_{x\sim\mathcal{D}} \left[\mathcal{L}\left(A(x;\theta)\right)\right]$$

- The optimizer A can be any algorithm parameterized by θ
  - Can be neural (weights), heuristic (mutation rate, population size) and even an ensemble (5 individual algorithms with respective parameters).
- The nature of the optimizer (differentiable? White-box? Black-box?) determines the training method (supervised learning, reinforcement learning, gradient-free methods).

Key Insight: L2O is not just "Learning for optimization". It is about **automating the design of algorithms** using data.

Beyond Beating Classical Algorithms

If we just want to solve static TSP, Concorde (Exact) or LKH (Heuristic) are incredibly hard to beat. L2O is often not the winner there yet.

Where is L2O **naturally well-suited** ? Still an Open research problem.

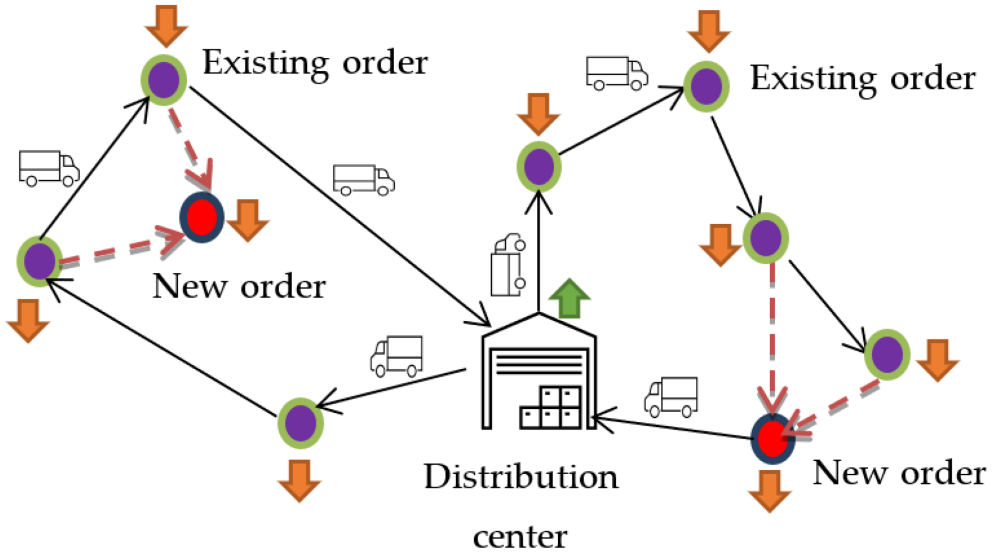
One example here:

- **Dynamics**: When the problem changes in real-time.

Dynamic Combinatorial Optimization

Context: The problem changes while we are solving it.

- Dynamic VRP: New customer requests arrive while trucks are moving.
- Dynamic Job Shop: Machines break down, new rush orders arrive.



Requirement: Instant Response.

- When a new request comes, we must decide immediately (assign to Truck A or B?). We cannot pause for 10 minutes to run a genetic algorithm.

Traditional vs. L2O in Dynamic Settings

Traditional approaches: Re-optimization.

- Every time the state changes, run a heuristic/meta-heuristic (e.g., local search) on the snapshot.
- **Pros**: Good quality.
- **Cons**: Computationally expensive, latency.

L2O: Learning a Policy.

- Train an optimizer to map state → action.
- State: Current truck locations, current demands.
- Action: next destination.
- Pros: instant inference.
- Cons: Possibly not as good as re-optimization.

The Road Ahead: Challenges

While promising, L2O faces significant hurdles.

Generalization:

- Models trained on 100 nodes often fail on 200/500 nodes.
- Models fail on OOD generalization.

Scalability:

- Training NNs on graphs with 10,000 nodes is extremely memory/time intensive.

Feasibility:

- Hard constraints (e.g., Time Windows) are difficult for NNs to satisfy perfectly.
- Often need a “Masking” step or a post-processing repair step.