PROBLEM-DUAL AUGMENTATION AND TRAINING FOR NEURAL COMBINATORIAL OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Much recent progress in neural combinatorial optimization comes from generating more varied candidate solutions for each instance. For Euclidean routing, instance augmentation—solving rotated and reflected copies of an instance—does this reliably, but job scheduling problems have no geometry to transform and lack a widely used counterpart that changes the model's input while preserving the objective exactly; MatNet's random machine embeddings, the closest mechanism, act like resampling. We show that classical problem duality provides a non-geometric one. An exact dual instance $T(I)$ has feasible solutions in an objective-preserving bijection with those of I, so any solution's cost is read directly off $T(I)$. The transformations are largely classical in operations research; what is new is their use as model-agnostic test-time augmentation for learned solvers, which we study on the asymmetric traveling salesman problem (ATSP) and five scheduling problems: permutation and flexible flow shop, job shop, flexible job shop, and single-machine scheduling with total weighted tardiness. The dual admits two uses. Problem-dual augmentation (PDA) solves I and $T(I)$ with one trained model, typically at the same candidate budget as solving I alone. It needs no retraining and improved the forward-only solver in 30 of 32 problem–size settings with a static dual, tying in one; the gain comes neither from the dual being easier nor from drawing more samples, but from the two candidate sets being complementary. Problem-dual training (PDT) shares one model across both directions, and its effect is problem-dependent: it narrows the remaining ATSP gap by about a tenth, from 3.62% to 3.27%, yet on job shop scheduling it offsets part of augmentation's gain, and on flexible job shop the sign of its effect changes between training runs. Combined, the two improve the forward-only solver in 28–30 of 35 settings, depending on the training run. Neither use changes the network architecture or the learning algorithm, and we apply both to four published solvers trained by supervised, reinforcement, self-imitation and preference learning.

1 INTRODUCTION

Neural combinatorial optimization (NCO) (Vinyals et al., 2015; Kool et al., 2019; Kwon et al., 2020) learns construction heuristics end-to-end. Much recent progress has come from generating *more varied candidates* for the same instance and keeping the best: POMO (Kwon et al., 2020) starts construction from every node and solves eight rotated and reflected copies of each instance, Sym-NCO (Kim et al., 2022) trains for invariance to such transformations, and MP-ASIL (Choi et al., 2025) conditions one network on many latent vectors.

Job scheduling lacks a counterpart Geometric augmentation needs coordinates. Rotating an instance of the traveling salesman problem (TSP) leaves every tour length unchanged, and the rotated instance is a genuinely different input the same model solves differently, which reliably improves inference. Job scheduling problems have no coordinates to rotate. Relabeling jobs or machines is a symmetry of these problems, but it is exactly the kind of change that permutation-equivariant encoders are designed to ignore. Scheduling solvers have therefore relied mainly on drawing more samples from the same construction process.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

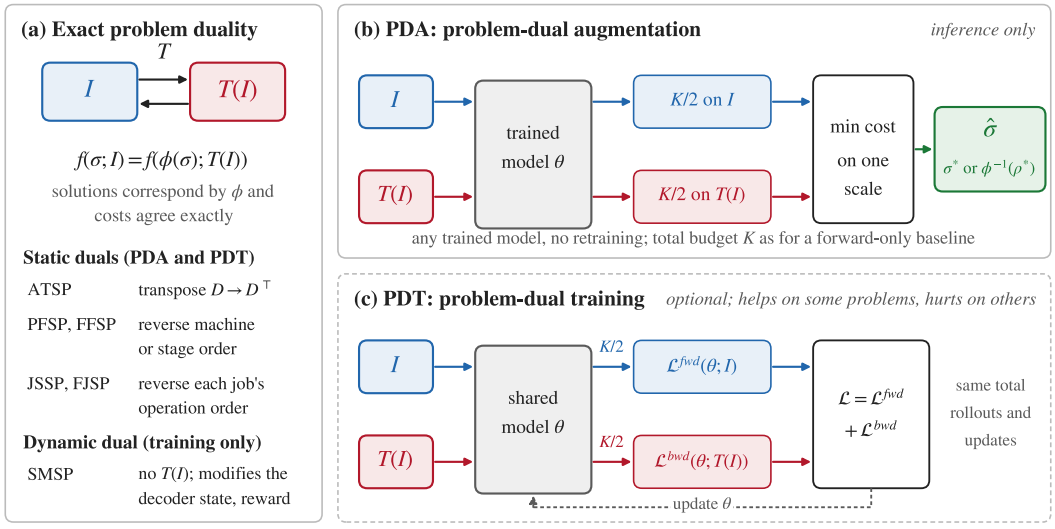


Figure 1: **Exact duals and their two uses. (a)** A static exact dual maps an instance I to $T(I)$, and a bijection ϕ between their solutions preserves cost exactly; five of our six problems admit one, while the SMSP dual exists only inside the decoder. **(b)** PDA gives I and $T(I)$ to any trained model with half of the candidate budget each, compares costs on one scale, and returns $\hat{\sigma}$, applying ϕ^{-1} only if the winner came from $T(I)$. **(c)** PDT optionally trains one shared model on both directions with the two losses summed, at the same total rollouts and updates as single-direction training.

Classical duality supplies one Operations research has long established that many combinatorial optimization problems admit *exact dual instances*: a transformed problem $T(I)$ whose feasible solutions are in an objective-preserving bijection with those of I. Reversed flow shops and job shops preserve the makespan and are used in heuristic search (Muth, 1979; Yamada & Nakano, 1997), and reversing every tour of an asymmetric TSP simply transposes its distance matrix. A dual instance is to a scheduling problem what a rotation is to a Euclidean one. The duals are therefore not our contribution. What has been missing is their use in learned solvers: an account of how to obtain them across problem classes, of the two ways a neural solver can exploit them, and of what each way buys. We study six problems: the asymmetric TSP (ATSP), permutation and flexible flow shop scheduling (PFSP, FFSP), job shop and flexible job shop scheduling (JSSP, FJSP), and single-machine scheduling with total weighted tardiness (SMSP).

Two components Because exact duality gives $f(\rho; T(I)) = f(\phi^{-1}(\rho); I)$ for *every* feasible ρ and not just the optimum, a backward solution's cost is read directly off $T(I)$. This licenses two uses, which we evaluate separately. **Problem-dual augmentation (PDA)** solves both I and $T(I)$ with one trained model and returns the better solution, reallocating a fixed candidate budget; it takes no gradient step and applies to any released checkpoint. **Problem-dual training (PDT)** shares one model across both directions with the two losses summed (Figure 1). We write **PDAT** for both together. Neither changes the network architecture or the learning algorithm: PDA wraps inference around an existing model, and PDT adds a second loss term to an existing training loop. We apply both unchanged to four published solvers trained by supervised learning, reinforcement learning, self-imitation learning and preference optimization. This makes the method complementary to progress in architectures and training algorithms rather than a competitor to it: a future solver inherits the gain without re-engineering, and PDA applies to deployed checkpoints.

Separating them is the paper's main empirical contribution The obvious way to evaluate a dual-trained model—against a forward-trained one, each evaluated in its own way—would move two variables at once, since only the dual-trained model would see the dual at inference. We instead hold the inference procedure fixed and vary only the checkpoint, and the two components turn out to behave differently.

Augmentation is reliable: on the five problems and six backbone configurations with a static dual, it improved the forward-only solver in 30 of 32 problem–size settings without a gradient step; of the other two, one is a tie and one improves in both repeat training runs (Section 5.3). Section 5.3 shows the gain is not from the dual being easier to solve, nor from drawing more candidates, but from the two candidate sets being complementary. Training is problem-dependent—it helps modestly on ATSP, while on JSSP it offsets part of augmentation's gain and on both FJSP data sets its sign depends on the training run. We test one plausible explanation for why it *can* cost (Section 5.4).

2 Related Work

Neural CO and diversity Pointer Networks (Vinyals et al., 2015) and the Attention Model (Kool et al., 2019) established the encoder–decoder paradigm; POMO (Kwon et al., 2020) added multi-start rollouts and MatNet (Kwon et al., 2021) extended it to matrix-encoded problems. Diversity has since been the main axis of progress: PolyNet (Hottung et al., 2025) learns complementary strategies, MEMENTO (Chalumeau et al., 2024) diversifies inference-time distributions, Sym-NCO (Kim et al., 2022) trains on symmetric transformations, and MP-ASIL (Choi et al., 2025) conditions one network on many latent vectors.

Relation to geometric augmentation Sym-NCO and MatNet are the closest relatives of PDA. A dual is no more exact than a rotation, which preserves every tour's cost and so satisfies Definition 1 of Section 3, and no better at reallocating a fixed budget, since K can be split across eight rotations as easily as across two directions. The one real difference is *availability*—rotation needs a metric space to act on, and job scheduling problems have none.

Test-time search and augmentation Efficient active search (Hottung et al., 2022) adapts a model's parameters to each instance, and simulation-guided beam search (Choo et al., 2022), also applied to flow shops, expands candidates with policy-guided rollouts; both search the instance as given, so either can also be run on $T(I)$. MatNet's random one-hot machine embeddings amount to relabeling the machines and on FFSP are indistinguishable from resampling (Appendix J). PDA is the combinatorial counterpart of test-time augmentation in vision (Shanmugam et al., 2021), without its aggregation problem: exactness puts both directions on one objective.

Duality elsewhere Dual learning (He et al., 2016) exploits duality between two *tasks* with two models and a round-trip reward; we exploit duality between two *instances* of one task with one model. Reversed-problem constructions are classical in OR, for flow shops (Muth, 1979) and for job shops (Yamada & Nakano, 1997), where arc reversal preserves critical-path length, and iterating forward and backward passes improves project schedules (Li & Willis, 1992). They refine one schedule with a hand-designed heuristic; PDA draws independent candidates from one learned policy on I and $T(I)$. A bidirectional graph attention network (Zhang et al., 2024) encodes the disjunctive graph in both directions inside the architecture; we transform the instance instead.

3 Exact Duals and How to Use Them

Duals come in two kinds, and the difference decides which of the two components is available. A *static* dual transforms the instance itself, so $T(I)$ is another instance of the same problem that any trained model can solve: both augmentation and training are available. A *dynamic* dual has no such instance—the reversal lives in the decoder's state and reward—so it can only be used by retraining. Five of our six problems admit a static dual; SMSP admits only a dynamic one.

Definition 1 (Static Exact Dual) For an instance I with objective $f(\sigma; I)$ over feasible set $\mathcal{S}(I)$, a *static exact dual* is a transformed instance $T(I)$ with a bijection $\phi_I : \mathcal{S}(I) \to \mathcal{S}(T(I))$ such that $f(\sigma; I) = f(\phi_I(\sigma); T(I))$ for all σ. Since ϕ_I is a bijection the identity runs both ways: $f(\rho; T(I)) = f(\phi_I^{-1}(\rho); I)$, the form used to score a backward rollout against the original objective. All five static duals below satisfy $T(T(I)) = I$ and $\phi_{T(I)} = \phi_I^{-1}$; since each ϕ_I reverses a sequence and so does not depend on I, we drop the subscript. This is duality between two instances of one problem, not the linear-programming or Lagrangian dual: the correspondence is a bijection

between feasible solutions with equal objective values, available to any trained model as another instance to solve, rather than a bound on the optimum in a different space of variables.

The five static duals ATSP transposes the distance matrix, with ϕ reversing the tour. PFSP and FFSP reverse the machine or stage order, with ϕ reflecting the time axis, which preserves precedence, non-overlap and makespan (Muth, 1979). JSSP reverses each job's operation sequence, with ϕ reversing every arc of the disjunctive graph, whose longest-path length is arc-reversal invariant (Yamada & Nakano, 1997); FJSP extends this, since machine assignments are invariant under time reflection. A solution is a machine assignment with a feasible order on each machine, its objective the longest-path length of the induced graph. Figure 1a shows them; Appendix A derives them. In each case the backward cost is read off $T(I)$ directly: ϕ is applied only to report the chosen solution.

SMSP: an exact dynamic dual Here there is no $T(I)$ to hand a trained model, so augmentation is unavailable and the dual can be used only by retraining. SMSP minimizes total weighted tardiness on one machine, and the static recipe fails: due dates are anchored to absolute time, so reversing the sequence changes every completion time. We construct the dual in the decoder instead. With $P = \sum_j p_j$ and q_t the total processing time of the jobs placed before step t of a backward rollout, which are those that follow them in the forward order, assigning job j at step t incurs $\Delta_t(j) = w_j \max(P - q_t - d_j, 0)$.

Proposition 1 For every permutation ρ, $\sum_t \Delta_t(\rho_t) = \mathrm{TWT}(\mathrm{rev}(\rho); I)$, where TWT denotes total weighted tardiness. *Proof.* In an idle-free schedule the job placed at backward step t completes at $P - q_t$ in the forward order, so Δ_t is its weighted tardiness there. □

The backward rollout thus optimizes the original objective through a modified state and reward rather than a transformed instance, which is why SMSP appears in Figure 1a under training only. This *static* versus *dynamic* distinction determines whether augmentation is available at all (Section 5.5).

4 The Two Components

PDA: problem-dual augmentation Given a trained model and instance I (Figure 1b), draw $\mathcal{C}_F \subset \mathcal{S}(I)$ of size $K/2$ and $\mathcal{C}_B \subset \mathcal{S}(T(I))$ of size $K/2$. The two live in different solution spaces, but Definition 1 puts their costs on one scale:

$$c^* = \min\Big\{ \min_{\sigma \in \mathcal{C}_F} f(\sigma; I),\ \ \min_{\rho \in \mathcal{C}_B} f(\rho; T(I)) \Big\}.$$

The returned solution $\hat{\sigma}$ is the forward minimizer σ^* if the first term attains c^*, and $\phi^{-1}(\rho^*)$ for the backward minimizer ρ^* otherwise—ϕ^{-1} is applied once, to the winner, only to report it. The candidate count matches a baseline drawing K from I alone, and this works on any released checkpoint.

This reallocation applies to every backbone with a splittable candidate axis: PFSP, JSSP, FJSP, FFSP. The two ATSP backbones need separate accounting. GOAL decodes greedily, so $K = 1$ and there is nothing to split; PDA uses two invocations there and we compare against an equal-invocation forward-only control (Section 5.3). MatPOENet splits its eight instance augmentations, four to each direction, so its 400 rollouts per instance match forward-only. Appendix F gives its full budget grid and traces why our forward-only reproduction differs from the published figure.

PDT: problem-dual training One shared model receives rollouts from both directions with the losses summed (Figure 1c), $\mathcal{L} = \mathcal{L}^{fwd}(\theta; I) + \mathcal{L}^{bwd}(\theta; T(I))$; PDT is equally independent of the backbone: the backward loss is the backbone's own loss evaluated on $T(I)$, so the same line of code serves supervised, reinforcement-learning, self-imitation and preference objectives alike. Summing is a deliberate choice: on PFSP it beats backpropagating only the better direction (1.68 against 1.74 average gap) and taking a separate optimizer step per direction (1.73 at matched update count, 1.82 with twice the updates; Appendix G). We report **PDT** alone only where it isolates the training effect. The two compose but are separate interventions: augmentation is reversible and, on the rollout-split backbones, candidate-budget matched; training is neither, so our experiments treat the *augmented* forward-only model as the baseline against which training is judged.

Budget matching For the five rollout-split backbones, PDT divides the K candidates (latent vectors; environments for PGMPO) into $K/2 + K/2$, so total rollouts, updates and instances processed match the baselines exactly; wall-clock does not, costing $1.02\times$–$2.16\times$ in our sequential implementation (Appendix E). GOAL matches all three via per-epoch subsampling; MatPOENet is matched only per step and we report both conventions. Appendix B gives the budget accounting.

5 EXPERIMENTS

5.1 SETUP

Variants All four variants run at an equal candidate budget K on the five scheduling backbones; Section 4 covers the two ATSP backbones separately. **FWD** (forward-only) trains and evaluates on I; **BWD** (backward-only) on $T(I)$; **FWD + PDA** takes the FWD checkpoint and splits K into $K/2$ on each instance; **PDAT** trains one shared model on both directions and evaluates it the same way.

Backbones For each problem we use the strongest published solver we could reproduce: GOAL (Drakulić et al., 2025) and MatPOENet (Pan et al., 2025) for ATSP, MP-ASIL (Choi et al., 2025) for PFSP, JSSP, FFSP and SMSP, and PGMPO (Choi et al., 2026) for FJSP—four solvers from 2025–2026, on seven backbone configurations, spanning supervised learning, reinforcement learning, self-imitation learning and preference optimization. On four of the six problems the forward-only variant is the published method at its published configuration. The ATSP (GOAL) and JSSP models were trained below the original budget. Appendix K gives the setting for each problem, Appendix D the backbones and hyperparameters, and Appendix C the provenance of every training and evaluation set together with the reference values behind each gap.

Metrics Following MP-ASIL, the reference-relative gap of instance s is $100 \times (f_o^s - f_b^s)/f_b^s$, where f_o^s is the objective value of the solution a method returns and f_b^s the reference value for that instance, averaged over instances. For PFSP and JSSP the reference is the best-known solution of each Taillard instance (Taillard, 1993), so a 0% gap is near-optimality. SMSP is reported as mean normalized total weighted tardiness, with gaps against MP-ASIL (Large) given in Appendix I, where a 0% gap means matching that solver rather than optimality. For FJSP and FFSP we report mean makespan instead, because the available references would make a gap measure the reference more than the method: the time-limited OR-Tools solutions used for FJSP vary in quality with instance size—our forward-only model is 38% above them at 15×10 and 6% below them at 40×10—and the FFSP reference, MP-ASIL (Large), is another learned solver published to one decimal place. Every variant is evaluated on the same instances, so this choice does not affect any comparison between them; Table C1 gives the references and Tables I6–I7 their values. MatPOENet has no accepted reference, so we report raw tour lengths. GOAL's ATSP evaluation scores the final arc in the wrong direction, so we rescore every GOAL tour by its closed length, under which the released checkpoint's gap is 3.46% rather than the 0.16% its decoder reports (Appendix C). Two references, MP-ASIL (Large) for FFSP and SMSP, are published only as means, so percentages against them are ratios of means, not averaged per-instance gaps; the two agree only for homogeneous instances.

5.2 MAIN RESULTS

Each row separates the two components Moving from FWD to FWD + PDA changes only the inference procedure, since the checkpoint is the same; moving from FWD + PDA to PDAT changes only the checkpoint, since the inference is the same.

Augmentation helps wherever the dual is static It improves the forward-only checkpoint in every row whose dual is static. On JSSP it does so at all eight Taillard sizes, on PFSP at six of seven and ties at the seventh, and on FFSP at all three, where it removes 10.4%, 9.4% and 18.3% of the gap to the MP-ASIL (Large) reference; on a greedy decoder with no candidate dimension of its own it removes 22.3%, or 17.5% against an equal-invocation forward-only control (Section 5.3). Only SMSP degrades, because its dual is dynamic.

Which component wins depends on the problem FWD + PDA improves on forward-only in 30 of the 32 static-dual settings of Appendices F and I, and PDAT in 28–30 of all 35 depending on

Table 1: Main results per problem (one size; PFSP and JSSP averaged over all sizes)

Problem	Backbone	Metric	FWD	BWD	FWD + PDA	PDAT
ATSP	GOAL	gap %, N=100	4.667	4.501	3.624*	**3.272**
ATSP	MatPOENet	tour, N=50, 400 rollouts	1.6145	1.6129	1.6121	**1.5929**
PFSP	MP-ASIL	gap %, 7 sizes	1.785	1.794	1.712	**1.708**
JSSP	MP-ASIL	gap %, 8 sizes	8.50	8.82	**7.84**	8.13
FJSP-SD2	PGMPO	makespan, 10×5	356.43	357.82	354.78	**354.07**
FJSP-SD1	PGMPO	makespan, 10×5	99.19	99.02	**98.91**	99.04
FFSP	MP-ASIL	makespan, 20×12	26.880	26.874	26.747	**26.705**
SMSP	MP-ASIL	TWT, n=100	0.2422	**0.2367**	0.2433†	0.2368

*Lower is better; bold marks the best of the four in each row. FWD + PDA and PDAT use bidirectional inference at the same candidate budget as FWD, except as marked. *GOAL decodes greedily, so PDA uses two invocations rather than splitting one; Section 5.3 compares it against an equal-invocation forward-only control. MatPOENet splits its eight instance augmentations four to each direction, so every entry uses 400 rollouts. PFSP values come from a single evaluation script that produces all four columns in one pass; Table I2 reports the published five-run protocol. †SMSP has no static dual; this control feeds the forward model the backward environment (Section 5.3). Full per-size results for every problem are in Appendix I, and for MatPOENet in Appendix F.*

the FJSP training run (Tables J6–J7). PDAT wins outright on both ATSP backbones, on FFSP, and on FJSP-SD2 at the 10×5 training size shown here—Section 5.4 shows that this last advantage does not reproduce at the zero-shot sizes; FWD + PDA wins on JSSP and, in the run shown, on FJSP-SD1; on PFSP the two are within noise; and on SMSP a single-direction backward model is best. That the winner depends on the problem is the paper's central empirical finding. Section 5.3 isolates why augmentation works, and Section 5.4 when training helps.

5.3 Augmentation: Where the Gain Comes From

Table 2: Decomposing the candidate budget, forward-only models

Candidate set	Total	JSSP, K=512	JSSP, K=64	FFSP, K=24
Original instance only	K	8.476 ± 0.062	9.758 ± 0.120	26.874 ± 0.030
Dual instance only	K	8.416 ± 0.093	9.658 ± 0.107	26.856 ± 0.026
Original only, as $K/2 + K/2$	K	8.504 ± 0.089	9.691 ± 0.089	26.867 ± 0.028
Original $K/2$ + dual $K/2$	K	**7.791 ± 0.026**	**9.096 ± 0.062**	**26.722 ± 0.019**
Original only, $2K$	$2K$	8.110 ± 0.081	9.293 ± 0.113	26.471 ± 0.025
Original K + dual K	$2K$	**7.500 ± 0.073**	**8.641 ± 0.048**	**26.320 ± 0.029**

JSSP columns are reference gaps (%) over the same 80 Taillard instances; FFSP is makespan over 1,000 instances. Every entry is the mean ± sd over five sampling runs, with the two halves of every split drawn independently. All come from a script that samples the candidate sets directly, so they differ from Table 1 by the usual run-to-run margin. Bold marks the best at each total budget.

Table 2 fixes one forward-only checkpoint and varies only where its candidates come from. Three readings follow. *The dual is not an easier problem*: solving only $T(I)$ is at most 0.10 better than solving only I on average, with the sign changing between runs, while drawing from both beats either alone by 0.56 to 0.69 on JSSP. *Nor is the gain from drawing more candidates*: two independent draws of $K/2$ from I come within 0.07 of one draw of K in every column. *Nor is it any change of representation*: on JSSP at K=64, spending the second half on a job-relabeled, machine-relabeled or doubly relabeled copy of I—each of which also preserves the objective exactly—gains 0.06–0.13 gap points averaged over two runs, six to thirteen times less than the 0.73 the dual gains in the same runs (Table J4); on FFSP the same relabelings change the makespan by -0.02 to +0.05, against -0.14 for the dual (Appendix J). *The gain comes from complementarity*: replacing $K/2$ forward candidates with backward ones cuts the JSSP gap by 0.69 points at K=512 and the FFSP makespan by 0.15 at K=24. At $2K$ the dual still beats more forward samples, by 0.61 and 0.15, and on JSSP splitting K across the two directions beats drawing $2K$ from I alone, at both budgets and in all five runs. On FFSP it does not (26.722 against 26.471), so at FFSP's 1.6–1.7× time overhead a

wall-clock-matched forward baseline can beat PDA; on JSSP PDA wins even against $2K$ forward candidates, a baseline we did not time. Candidates from one instance under one policy cover a narrower region than the same number drawn from I and $T(I)$ together. The effect is uneven—on JSSP the dual half improves the best candidate in 69 of 160 instance–run pairs and worsens it in 39 (Table J4)—so the average reflects gains on a minority.

The dual reaches regions re-sampling does not We measure the candidates directly, as normalized Kendall-τ distance between machine orders after mapping every candidate back to I. On JSSP, the backward set's mean distance from the forward set is 44% larger than that of a second forward draw (0.1262 against 0.0878), and its nearest-neighbor novelty is 65% higher (0.1108 against 0.0670) and exceeds that of the second draw on all 80 instances; job- and machine-relabeled sets differ from a second forward draw by at most 1% on both measures (Appendix J). Relabeling changes the input but not where the policy lands; the dual changes both. Distance alone does not predict the per-instance gain (Spearman 0.05): the dual reliably reaches new regions, and the gain depends on what they contain.

Augmentation survives retraining The gain is largely robust to the training initialization. On both FJSP data sets we repeated the forward-only training from three initializations and evaluated all six sizes: augmentation improves 35 of the 36 checkpoint-and-size combinations, while the training effect measured on the same checkpoints changes sign between runs (Section 5.4). The exception is the first SD2 run at 20×10, worse by 1.40 makespan, where the two repeat runs both improve.

A greedy decoder gains most GOAL decodes one tour per instance, with no multi-start, augmentation or latent dimension, so it has nothing to reallocate: PDA there uses a second decoder invocation. The right comparison is therefore a second *forward* invocation. Decoding forward twice and keeping the better tour is worth 0.27 gap points (4.67 $\rightarrow$ 4.39); spending the second invocation on the dual is worth 1.04, 3.8 times as much, with non-overlapping five-seed ranges (Table I1). On the authors' full-budget released checkpoint the ratio grows to 6.3 (0.79 against 0.13 gap points), taking it from 3.46% to 2.66%, though it solves $T(I)$ alone no better than I (3.48% against 3.46%).

Augmentation needs a static dual SMSP's dual reformulates the decoder's state and inputs rather than transforming the instance, so there is no $T(I)$ to hand an existing model. A forward-trained SMSP model fed the backward environment receives $(p_j, P - d_j, w_j)$ where it was trained on (p_j, d_j, w_j), and degrades: 0.2433 against 0.2422 at $n = 100$, 0.3797 against 0.3788 at $n = 500$.

Cost Candidate-matched is not wall-clock-matched: our sequential implementation costs at most $2.1\times$ at inference (Appendix E), while MatPOENet, which batches the two directions, decodes at parity. Appendix J examines how the augmentation margin varies with the candidate budget.

5.4 Training: Where It Helps and Where It Does Not

Table 1 shows the training effect directly as the step from FWD + PDA to PDAT, with the inference procedure held fixed. It changes sign by problem and, where we repeated training, by run; on FJSP, a single run would have supported either conclusion.

Where training helps On ATSP it helps modestly and consistently. At one candidate per instance it narrows the GOAL gap from 4.67% to 4.20% (-9.9%), and at two from 3.62% to 3.27% (-9.7%, 0.35 points); both comparisons are budget-matched, with non-overlapping five-seed ranges. Matching the budget means that in each epoch the dual-trained model sees half as many original instances, the other half replaced by their duals (Appendix B), and since a dual's label is the original LKH tour mapped through ϕ (Appendix A.1), the gain costs no additional labeling. The two components add rather than compound: a second candidate narrows the gap by the same factor, about 1.3, with or without dual training. MatPOENet, trained by reinforcement learning, shows the same direction: 1.2% of tour length at a matched 400 rollouts (Table F1).

Where training costs On JSSP, training helps at only two of eight sizes and costs at six, a mean of +0.29 gap points, with the largest penalties at zero-shot shapes (Table I3). PDAT still has the lowest average against the single-direction baselines, but only because it alone uses augmentation; neither

variant reaches the published 7.4%, which uses four times our training policies. This rests on one training run; the repeat runs on FJSP below show how far one run can mislead.

Where a single run misleads FJSP is where we can see through it, because we repeated both data sets from three initializations (Appendix J). Within a run the effect is coherent at the larger sizes: from 20×10 to 40×10 it keeps one sign in all six runs and mostly grows with the distance from the 10×5 training size, so it reflects the trained model rather than noise in the evaluation. Across runs that sign flips. On SD2 the first run gains 7.87 and 7.74 makespan at 30×10 and 40×10, and the two repeat runs lose 4.58 and 5.07, and 7.35 and 8.63. On SD1 the first run loses at five of six sizes, and the two repeat runs gain at five and at all six, by up to 1.72 and 1.74 at 40×10. A single run would have told us that training helps on SD2 and costs on SD1; three runs show neither. The only size where all three runs agree is SD2 at 10×5, where training helps by 0.41 to 0.71 makespan. What grows with the size gap is the spread between training runs, not a benefit of training.

Where it is within noise On FFSP training lowers makespan by 0.042, 0.029 and 0.002 at the three sizes (Table I6), and on PFSP it lowers the gap by 0.004 points (Table 1). We repeated FFSP 20×12 from three independent initializations: the combined PDAT margin over forward-only holds in every run, at 0.126–0.168 at the final checkpoint (Table J2; 0.175 in Table 1's evaluation), but the training component alone is the size of the 0.03 run-to-run spread (Appendix J), so we claim no sign.

SMSP With augmentation unavailable, using the dual requires retraining, but not necessarily in both directions. Backward-only reaches 0.2367 at $n = 100$, tying PDAT, and at $n = 500$ reaches 0.3324 against PDAT's 0.3704, a mean 9.8% below MP-ASIL (Large)'s 0.3685, the largest margin over a published reference in our experiments (Table I7). A second run reproduces backward-only almost exactly (0.2367, 0.3321) while forward-only moves to 0.5498 at n=500; PDAT was trained once. On this evidence a practitioner facing SMSP should train backward-only. The dual is still worth deriving: at n=500 it gains 12% and 40% over forward-only, from reformulation, not tuning.

Why training can cost A model trained on both directions at a fixed budget sees half as many rollouts per direction, which can reduce its competence in each. This predicts that where training costs, the dual-trained model should be worse than the forward-only one *in the forward direction alone*—and on JSSP it is, 8.76 against 8.40 gap points in a separate evaluation of both checkpoints, a penalty of 0.36 close to the 0.29 it loses overall. On FFSP, where training is neutral, the difference is within noise. This is consistent with, but does not prove, capacity dilution.

A hypothesis the data do not settle One might expect training to help where the two directions produce similar candidate sets. The across-instance correlation between each instance's mean forward and mean backward objective value does not separate the cases—ATSP, where training helps, and both FJSP data sets, where its sign changes between runs, all sit above 0.9—and it largely reflects shared instance difficulty (Appendix J).

On four public FJSP benchmarks evaluated zero-shot, augmentation helps on three and training adds to it on two (Appendix H). Measured against the augmented forward-only model, training fares worst away from the size it was trained on: on JSSP it costs at all five zero-shot sizes, by 0.35 to 0.80 gap points, against a mean of $+0.29$ over all eight; on PFSP it is within 0.08 at every zero-shot size; and on both FJSP data sets the zero-shot sizes are exactly where the training runs disagree. Distribution shift is where the training component is least dependable.

5.5 Which Component to Use

One of the two questions a practitioner faces is answered by the derivation; the other is not.

The static/dynamic question needs no experiment If reversing the instance yields another instance of the same problem—as for all five static duals of Section 3—then $T(I)$ can be handed to any trained model and augmentation is available. If the reversal must instead be expressed as a change to the decoder's state and reward, as in SMSP, using the dual requires retraining, and which scheme to use is a separate question.

Whether to train is an empirical choice The effect is positive on both ATSP backbones, negative on JSSP, seed-dependent on both FJSP data sets, and within noise on PFSP and FFSP. A short pilot is the practical test: train both variants for a fraction of the schedule and compare on a held-out set—and, since the sign can change between initializations, on more than one seed at the sizes that matter. The conservative default is to apply PDA wherever the dual is static—it needs no retraining, is reversible, and improved 30 of 32 such settings—and to treat training as a separate, empirical choice.

6 CONCLUSION

We presented problem-dual augmentation and training for neural combinatorial optimization. We used exact duals for six problem classes—most of them classical in operations research, together with a backward construction for SMSP where sequence reversal does not preserve the objective—as a non-geometric augmentation for learned solvers. Both components are plug-ins that leave the network architecture and the learning algorithm unchanged, and we applied them to four published solvers spanning four learning paradigms.

Augmentation alone improves the forward-only solver in 30 of 32 settings with a static dual, the combined method in 28–30 of 35, but the two components behave differently, which a combined evaluation would hide.

Augmentation is reliable and applies to models that already exist On the five problems with a static dual it improves on solving I alone: on JSSP from 8.50% to 7.84% gap, on FFSP by 0.12 to 0.22 makespan, on FJSP by 0.03–0.47% of the objective at all but one of the reported sizes, and on ATSP-GOAL from 4.67% to 3.62%, or from 4.39% against an equal-invocation forward-only control. It also narrows the gap of the authors' released GOAL checkpoint from 3.46% to 2.66%. The gain comes from the two candidate sets being complementary (Section 5.3): this is what geometric augmentation does for Euclidean routing, and duality does it without geometry.

Training is a problem-dependent complement Judged against a model already using augmentation, it helps modestly on ATSP-GOAL (3.62% to 3.27%), costs on JSSP, and on both FJSP data sets changes sign between training runs. On SMSP, where augmentation is unavailable, retraining is the only route, and backward-only did best in each training run. We identify capacity dilution as one plausible mechanism for the cost, but the natural predictor of which problems pay it—the instance-level correlation between forward and backward costs—does not separate the cases.

Limitations The instance generators we use draw processing times identically across machines, stages and operation positions, so reversing an instance leaves it inside the training distribution, as a rotation does for a Euclidean instance. Where a real shop has position-dependent structure—a systematically slower final stage, say—the dual stays exact, so correctness is never at risk, but $T(I)$ may leave the training distribution, and weak backward candidates would then spend half the budget for no gain. We have not observed this: of the two static-dual settings where augmentation did not improve, one is a tie and one improves in both repeat runs (Section 5.3); its one miss on public FJSP benchmarks is within evaluation spread (Appendix H). Each problem requires identifying a valid dual, and SMSP's modifies the decoder's state update. Our training loops decode the two directions sequentially for six of our seven backbone configurations, costing 2–116% extra wall-clock. The ATSP (GOAL) and JSSP models were trained below the original budget; MatPOENet is matched per optimizer step but not on total trajectories. PFSP's ten instances per size leave its training effect below the noise floor. We repeated training three times on FFSP and both FJSP data sets: on FFSP the training effect (0.042 at 20×12) is comparable to the 0.03 run-to-run spread while the combined margin holds in every run, and on FJSP its sign changes between runs. SMSP's single-direction variants were trained twice. Elsewhere each configuration was trained once: both ATSP backbones; PFSP; JSSP, whose training cost therefore rests on a single run; and SMSP's PDAT.

Future work Three questions remain open: how much geometric and dual augmentation overlap on Euclidean routing, where both apply to one model; whether the sign of the training effect can be predicted before training; and whether our duals combine with the other kind—reduced costs or bounds from a relaxation, used as decoder features or baselines.

AI USE STATEMENT

We used generative AI tools to implement methods (evaluation and analysis scripts), to provide feedback on experimental design, to interpret results, and to draft and edit the manuscript. We did not use them to develop the theoretical framework or to prove Proposition 1. All AI-assisted code was executed by the authors and its outputs verified against independent implementations where feasible; all numerical claims were regenerated from our own experimental logs. We take responsibility for the final content of this work, including text, claims and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work uses only publicly available benchmark instances and synthetically generated instances, and involves no human subjects or personal data. It concerns routing and scheduling optimization and has no foreseeable direct negative societal impact beyond those general to optimization methods. All backbones are used under their released code and licenses.

REPRODUCIBILITY STATEMENT

All six problems use publicly released backbones and benchmark instances, cited in Section 5.1 and detailed in Appendix K. Appendix A derives each dual transformation and Section 3 states the one theoretical result with its proof. Appendix D lists hyperparameters for every configuration, including the two where we trained below the original budget; Appendix C records the reference values against which gaps are computed, and Appendix B the budget accounting per backbone. The supplementary material contains, for every problem, our modifications to the backbone's training and evaluation loops, the standalone scripts behind the budget decomposition, relabeling, diversity and correlation measurements, and the test instances where their size permits, each with a README giving the settings used. It also contains the trained checkpoints behind Table 1 for ATSP (GOAL), JSSP, FJSP and SMSP; those for PFSP, FFSP and ATSP (MatPOENet) exceed the size limit and will be released with the code upon publication.

REFERENCES

Félix Chalumeau, Refiloe Shabe, Noah De Nicola, Arnu Pretorius, Thomas D. Barrett, and Nathan Grinsztajn. Memory-enhanced neural solvers for efficient adaptation in combinatorial optimization. arXiv:2406.16424, 2024.

Inguk Choi, Woo-Jin Shin, Sang-Hyun Cho, and Hyun-Jung Kim. Towards generalizable multi-policy optimization with self-evolution for job scheduling. In *Advances in Neural Information Processing Systems*, volume 38, pp. 24569–24616, 2025. doi: 10.52202/085713-0828.

Inguk Choi, Woo-Jin Shin, Sang-Hyun Cho, and Hyun-Jung Kim. Preference-guided multi-policy optimization for flexible job shop scheduling. In *Proceedings of the Thirty-Fifth International Joint Conference on Artificial Intelligence, IJCAI-26*, pp. 6156–6165. International Joint Conferences on Artificial Intelligence Organization, 2026. doi: 10.24963/ijcai.2026/685.

Jinho Choo, Yeong-Dae Kwon, Jihoon Kim, Jeongwoo Jae, André Hottung, Kevin Tierney, and Youngjune Gwon. Simulation-guided beam search for neural combinatorial optimization. In *Advances in Neural Information Processing Systems*, volume 35, pp. 8760–8772, 2022.

Andrea Corsini, Angelo Porrello, Simone Calderara, and Mauro Dell'Amico. Self-labeling the job shop scheduling problem. In *Advances in Neural Information Processing Systems*, volume 37, pp. 105528–105551, 2024.

Darko Drakulić, Sofia Michel, and Jean-Marc Andreoli. GOAL: A generalist combinatorial optimization agent learner. In *International Conference on Learning Representations*, pp. 52465–52488, 2025. Code: https://github.com/naver/goal-co/.

Di He, Yingce Xia, Tao Qin, Liwei Wang, Nenghai Yu, Tie-Yan Liu, and Wei-Ying Ma. Dual learning for machine translation. In *Advances in Neural Information Processing Systems*, volume 29, 2016.

Keld Helsgaun. An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems. Technical report, Roskilde University, 2017.

André Hottung, Yeong-Dae Kwon, and Kevin Tierney. Efficient active search for combinatorial optimization problems. In *International Conference on Learning Representations*, 2022.

André Hottung, Mridul Mahajan, and Kevin Tierney. PolyNet: Learning diverse solution strategies for neural combinatorial optimization. In *International Conference on Learning Representations*, pp. 80417–80435, 2025.

Minsu Kim, Junyoung Park, and Jinkyoo Park. Sym-NCO: Leveraging symmetricity for neural combinatorial optimization. In *Advances in Neural Information Processing Systems*, volume 35, pp. 1936–1949, 2022.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. POMO: Policy optimization with multiple optima for reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 33, pp. 21188–21198, 2020.

Yeong-Dae Kwon, Jinho Choo, Iljoo Yoon, Minah Park, Duwon Park, and Youngjune Gwon. Matrix encoding networks for neural combinatorial optimization. In *Advances in Neural Information Processing Systems*, volume 34, pp. 5138–5149, 2021.

K. Y. Li and R. J. Willis. An iterative scheduling technique for resource-constrained project scheduling. *European Journal of Operational Research*, 56(3):370–379, 1992. doi: 10.1016/0377-2217(92)90320-9.

Eginhard J. Muth. The reversibility property of production lines. *Management Science*, 25(2): 152–158, 1979. doi: 10.1287/mnsc.25.2.152.

Wenzheng Pan, Hao Xiong, Jiale Ma, Wentao Zhao, Yang Li, and Junchi Yan. UniCO: On unified combinatorial optimization via problem reduction to matrix-encoded general TSP. In *International Conference on Learning Representations*, pp. 23403–23443, 2025.

Divya Shanmugam, Davis Blalock, Guha Balakrishnan, and John Guttag. Better aggregation in test-time augmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 1194–1203, 2021.

Eric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, volume 28, 2015.

Takeshi Yamada and Ryohei Nakano. Job-shop scheduling. In A. M. S. Zalzala and P. J. Fleming (eds.), *Genetic Algorithms in Engineering Systems*, volume 55 of *IEE Control Engineering Series*, pp. 134–160. The Institution of Electrical Engineers, 1997.

Cong Zhang, Zhiguang Cao, Yaoxin Wu, Wen Song, and Jing Sun. Learning topological representations with bidirectional graph attention network for solving job shop scheduling problem. In *Proceedings of the Fortieth Conference on Uncertainty in Artificial Intelligence*, volume 244 of *Proceedings of Machine Learning Research*, pp. 4192–4208, 2024.

A Derivations of the Duals

A.1 ATSP

Backward transformation $T(D) = D^\top$. Training reward $r^{bwd} = -\text{cost}(\rho; D^\top) = -\text{cost}(\text{rev}(\rho); D)$. Inference: evaluate backward candidates by $\text{cost}(\rho; D^\top)$, equal to $\text{cost}(\text{rev}(\rho); D)$ by exactness.

Implementation note (GOAL). GOAL stores each training instance with its nodes renumbered so that the labeled (LKH) tour visits them in index order, and its supervised targets rely on that order. A plain transpose breaks it, since the labeled tour, read on $D^\top$, visits the nodes in reverse. The backward instance is therefore $F = \text{flip}(\text{flip}(D, 0), 1)^\top$, which transposes *and* reverses the node order, so that the labeled tour is again in index order. We verified on subproblems of varying size and offset that the identity path length is preserved.

Implementation note (evaluation data). GOAL's evaluation instances use a different terminal-node duplication convention from its training instances. The backward transform for evaluation must reconstruct the terminal row and column under the evaluation convention; applying the training-side transform instead makes the start node artificially symmetric and yields optimistically biased results. We validated the corrected transform by checking that $\text{cost}(\tau, D) = \text{cost}(\text{rev}(\tau), F)$ for random permutations, with F the evaluation-side backward instance, and that the released checkpoint scores comparably in both directions. Tours are scored by their closed length, for the reason given in Appendix C.

A.2 PFSP AND FFSP

Backward transformation $p^{rev}_{j,k} = p_{j,m+1-k}$. Training reward $r^{bwd} = -C_{\max}(\rho; T(I))$, equal to $-C_{\max}(\text{rev}(\rho); I)$ by exactness. The environment computes makespan on the original instance in both directions, which is redundant for PFSP (exactness guarantees equality) but harmless.

FFSP is the same construction with stages in place of machines: reversing the stage order and reflecting the time axis preserves stage precedence, machine capacity within each stage and the makespan.

A.3 JSSP AND FJSP

Backward transformation $(\mu'_{j,k}, p'_{j,k}) = (\mu_{j,m_j+1-k}, p_{j,m_j+1-k})$. Training reward $r^{bwd} = -C_{\max}(S^{bwd}; T(I))$. By exactness this equals $-C_{\max}(\phi(S^{bwd}); I)$ for every feasible schedule, so no machine-wise reversal is needed during training. For solution reporting, apply ϕ: extract machine-wise operation orders from S^{bwd}, reverse each, convert backward operation indices to forward indices via $k^{fwd} = m_j + 1 - k^{bwd}$, and recompute the earliest-start schedule on I.

FJSP extends this. Time reflection acts on the operation order and on each machine's sequence, and leaves the machine assignment of every operation unchanged, so the assignment part of a solution maps to itself and the ordering part reverses.

A.4 SMSP (DYNAMIC DUAL)

Backward encoder input $x^{bwd}_j = (p_j, P - d_j, w_j)$, against the forward input (p_j, d_j, w_j): only the due date is reflected, and neither direction is rescaled by P; backward state $q_t = \sum_{k<t} p_{\rho_k}$; backward step cost $\Delta_t(j) = w_j \max(P - q_t - d_j, 0)$; solution mapping $\phi(\rho) = \text{rev}(\rho)$; training reward $r^{bwd} = -\sum_t \Delta_t(\rho_t) = -\text{TWT}(\text{rev}(\rho); I)$ (exact by Proposition 1). The forward stream's reward must likewise be computed on the original (p, d, w), not on whatever transformed values the encoder receives.

B BUDGET ACCOUNTING

Table B1: Budget quantities per training step

Problem	Backbone	Split axis	FWD rollouts	PDAT rollouts	Optimizer steps	Instances/step
PFSP	MatNet (MP-ASIL)	latent vectors	128	64 + 64	equal	equal
SMSP	MP-ASIL	latent vectors	128	64 + 64	equal	equal
JSSP	MP-ASIL on LEHD (SLIM)	latent vectors	64	32 + 32	equal	equal
FFSP	MP-ASIL on MatNet	latent vectors	24	12 + 12	equal	equal
FJSP	PGMPO	preference-conditioned environments	1024	512 + 512	equal	equal
ATSP	GOAL	batch + epoch subset	256	128 × 2	equal	half
ATSP	MatPOENet-A	batch	200	100 × 2	2×	half
ATSP	MatPOENet-B	batch	200	200 × 2	equal	equal

For GOAL, halving the batch alone would double the optimizer steps; we additionally sample a fresh random half of the instance pool each epoch, restoring step parity. For MatPOENet the backbone concatenates along the batch axis with no separable candidate dimension, so we report both conventions. Appendix F reports results under both conventions.

Loss-combination ablation Appendix G compares three ways of combining the two directions' losses: summing them into one step (the rule used throughout), backpropagating only the better direction, and taking a separate optimizer step for each direction. The first two perform one optimizer step per batch and run for the full 1000 epochs. The separate-step rule performs two per batch and is therefore run for 500 epochs so that total parameter updates match; Table G1 also reports it at 1000 epochs, to show what the extra updates do. All three see the same number of rollouts per batch ($64 + 64$).

C Data Provenance

Table C1: Gap reference values

Problem	Reference f_b^s
ATSP (GOAL)	Benchmark reference tour lengths shipped with the test set
ATSP (MatPOENet)	None — we report raw tour length
PFSP	Taillard best-known makespans
JSSP	Taillard best-known makespans, as stored in the benchmark files
FJSP	OR-Tools with a 1,800-second limit, as reported for PGMPO (Choi et al., 2026)
FFSP	MP-ASIL (Large) (Choi et al., 2025), the authors' reference
SMSP	MP-ASIL (Large) (Choi et al., 2025), the authors' reference

The following provenance details are worth stating explicitly, because each affects how absolute numbers should be read.

The GOAL repository ships two files with the same name. The authors' 128-instance evaluation set sits at *data/atsp100_test.npz*; a separate 10,000-instance file generated by their data tools sits at *atsp/atsp100_test.npz*. The two are not interchangeable, so results on one should not be compared with results on the other. All ATSP numbers in this paper use the former, matching the protocol of the original work.

Scoring ATSP tours. GOAL's decoder builds each tour as an open path from node 0 to a duplicated terminal node and scores it as the sum of the arcs along that path. In the evaluation file the terminal node's column stores the distances *out of* node 0 rather than *into* it, so the path's final arc is counted in the wrong direction. The two coincide for a symmetric instance, which is presumably why the convention is harmless for TSP, but for ATSP the scored length differs from the closed tour's length by up to 0.8 on tours of length about 10, enough for a single greedy tour to appear 2% shorter than the reference on individual instances. We therefore rescore every tour, forward and backward, by its closed ATSP length on the instance's distance matrix, mapping backward tours through ϕ^{-1} first. On one test instance, for example, the decoder scores a greedy tour at 12.31, its closed length is 13.00, and the reference is 12.55. The evaluation file gives reference lengths but not reference

Table C2: Training and evaluation instances per problem

Problem	Training instances	Evaluation instances
ATSP (GOAL)	Authors' released 1M-instance ATSP-100 set (tmat, LKH-labeled); we use a 200k subset	Authors' released 128-instance ATSP-100 test set (*data/atsp100_test.npz* in the released repository)
ATSP (MatPOENet)	Generated on the fly by the authors' *get_random_problems*: integers $\sim U[0, 10^6]$, zero diagonal, Floyd–Warshall closure, rescaled by 10^6	Authors' released *test_set/50_10000* (10,000 instances; we evaluate the first 1,000)
PFSP	Generated on the fly, $p_{jk} \sim U(1, 99)$, at 20×5 and 20×10	Taillard benchmark, 10 instances per size
JSSP	5,000 instances for each of six sizes (10×10 through 20×20), $p_{ji} \sim U(\{1, \ldots, 99\})$, 30,000 per epoch	Taillard benchmark, 10 instances per size
FFSP	Generated on the fly, $p_{jk} \sim U(2, 10)$, 3 stages $\times$ 4 machines, 1,000 per epoch, at 20 / 50 / 100 jobs	Authors' released 10,000-instance files at each size; first 1,000 evaluated, per the reference config
FJSP (SD1)	Authors' SD1 generator, 10×5; per-operation mean processing time with narrow per-machine deviation	Authors' released SD1 sets at 10×5, 20×5, 15×10, 20×10, 30×10, 40×10 (100 each)
FJSP (SD2)	Authors' SD2 generator, 10×5; each operation–machine time drawn independently from $U[1, 99]$	Authors' released SD2 sets at the same six sizes, plus Brandimarte and Hurink edata/vdata/rdata
SMSP	Generated on the fly at $n = 50$, 1,000 per epoch	Fixed test sets at $n \in \{50, 100, 500\}$

tours, so we solved all 128 instances ourselves with LKH-3 (Helsgaun, 2017) as closed ATSP tours. LKH reproduces the shipped references to within 0.034% on average and 0.38% at most, and finds a shorter closed tour on none of them, so the references are closed tour lengths. Because the optimal closed length is unchanged when the matrix is transposed, this holds whichever direction the matrix is read in. Consistently, no tour of the released checkpoint, with or without augmentation, beats the reference under closed scoring, whereas under the decoder's scoring a single greedy tour appears up to about 2% shorter than the reference on individual instances. The released checkpoint's gap is 3.46% under closed scoring and 0.16% under the decoder's. All GOAL numbers in this paper use the closed length. The issue is specific to GOAL's duplicated terminal node: MatPOENet scores tours in its own environment, and its forward and backward results agree (Table F1), as the distributional symmetry of its instances predicts.

MatPOENet's training and test distributions differ. The generator enforces the triangle inequality by construction; the released test instances violate it on 21% of entry triples. The UniCO paper

describes its ATSP training matrices as drawn from Uniform$(0, 1)$ with no metric closure, whereas the released generator enforces the triangle inequality; we train on the generator's output, as the released checkpoints were. Neither difference affects the comparison, since all four variants share the same setup, and we verified that the released checkpoint scores identically on both distributions (1.6468 vs 1.6464).

FJSP's two data sets differ in how processing times depend on the machine. SD1 draws a mean processing time $\bar{p} \sim U(1, 20)$ for each operation and then each compatible machine's time from $U(0.8\bar{p}, 1.2\bar{p})$, so every compatible machine's time lies within ±20% of the operation's mean. SD2 draws every operation–machine time independently from $U(1, 99)$, so the choice of machine can change it several-fold. Machine assignment therefore matters far more on SD2 than on SD1.

Zero-shot sizes. The Taillard benchmarks come in several sizes. For PFSP and JSSP, every size other than the training size contains no instance the model has seen, and we label those sizes zero-shot throughout.

D Backbones and Hyperparameter Settings

Table D1: Backbones

Problem	Backbone	Venue	Learning paradigm
ATSP	GOAL (Drakulić et al., 2025)	ICLR 2025	Supervised, on LKH solutions
ATSP	MatPOENet (Pan et al., 2025)	ICLR 2025	REINFORCE with POMO baselines
PFSP	MatNet-based MP-ASIL (Choi et al., 2025)	NeurIPS 2025	Self-imitation
JSSP	SLIM-based MP-ASIL (Choi et al., 2025; Corsini et al., 2024)	NeurIPS 2025	Self-imitation
FJSP	PGMPO (Choi et al., 2026)	IJCAI-ECAI 2026	Preference optimization (Bradley–Terry)
FFSP	MatNet-based MP-ASIL (Choi et al., 2025)	NeurIPS 2025	Self-imitation
SMSP	MP-ASIL (Choi et al., 2025)	NeurIPS 2025	Self-imitation

Table D2: Hyperparameters per configuration

Hyperparameter	ATSP (GOAL)	ATSP (MatPOENet)	PFSP	JSSP	FJSP	FFSP	SMSP
Backbone	GOAL	MatPOENet	MatNet (MP-ASIL)	LEHD (MP-ASIL)	PGMPO	MatNet (MP-ASIL)	MP-ASIL
Training algorithm	Supervised	REINFORCE	SIL	SIL	PGMPO loss	SIL	SIL
Training size(s)	ATSP-100	$N = 50$	20×5, 20×10	6 sizes, 10×10–20×20	10×5 (SD1, SD2)	$20/50/100 \times 12$	$n = 50$
Learning rate	5e-4	4e-4	1e-4	1e-4	2e-4	1e-5	1e-4
LR schedule	×0.9 / 5 ep	constant	MultiStep [0.9, 0.95]	constant	constant	MultiStep [80, 90]	MultiStep [0.9, 0.95]
Embedding dim	128	512	128	128	128	256	128
Encoder layers	9	8	6	2 (GATv2)	2 (OMB/MMB)	3	1
Decoder layers	—	—	1	2	3 (MLP actor)	1	5
Heads	8	16	8	3	4	16	8
Batch size (FWD/BWD)	256	200	200 (100×2 accum.)	16 (1×16 accum.)	16 (8×2 accum.)	50	50 (2×25 accum.)
Batch size (PDAT)	128	100 / 200	200 (100×2 accum.)	16 (1×16 accum.)	16 (8×2 accum.)	50	50 (2×25 accum.)
K (FWD/BWD)	—	N	128	64	128 envs	24	128
K (PDAT, per direction)	—	N	64	32	64 envs	12	64
Epochs	150	2000	1000	15	8000 updates	100 / 150 / 200	100
Instances/epoch	200k (PDAT 100k)	10k	100k	30k	16/update	1k	1k
Inference k	greedy / ×2	N	128	512	100 samples	24	128
GPU	H200 80GB	H200 80GB	RTX 4090	H200 80GB	H200 80GB	H200 80GB	RTX 4090 (run 2: H200)

Batch sizes are instances per optimizer update; where gradients are accumulated, the per-call batch and the number of accumulated calls are given in parentheses, and the per-call batch is the one Table E1 reports. OMB/MMB are PGMPO's operation and machine message attention blocks.

Deviations from the original papers, applied identically to every variant within a problem: GOAL uses 200k instead of 1M instances and 150 instead of 400 epochs on a single GPU rather than eight; JSSP uses 64 instead of 256 training policies and 15 instead of 20 epochs. All other settings follow the respective original papers, including MatPOENet's nearest-neighbor rough solver for POE initialization. FFSP trains from the released MatNet checkpoint, as the reference implementation does; the latent-conditioning module has no counterpart there and is initialized randomly.

E Wall-Clock

All timings below are measured on one H200, re-running a single epoch of each configuration on the same machine so that the ratios are comparable across problems. Absolute values are not comparable to the original papers, whose hardware differs.

The overhead ranges from negligible to 2.16× and is a property of how we implemented the two streams, not of the method. Our training loops run the forward and backward rollouts sequentially, so PDAT issues twice as many decoder calls, each operating on half the work. Where a single call

Table E1: Training wall-clock per epoch (per optimizer update for FJSP), PDAT relative to FWD

Problem	Backbone	Per-direction work	FWD	PDAT	Ratio
FFSP	MatNet-based MP-ASIL	50×12	43 s	44 s	**1.02**×
ATSP	GOAL	128	209 s	218 s	**1.04**×
PFSP	MatNet-based MP-ASIL	100×64	1.94 min	2.67 min	**1.38**×
FJSP	PGMPO	8×64	2.76 s/update	4.13 s/update	**1.50**×
ATSP	MatPOENet	100–200	0.26 min	0.44 min	**1.67**×
SMSP	MP-ASIL	2×64	2.20 min	4.18 min	**1.90**×
JSSP	SLIM-based MP-ASIL	1×32	3.7 h	8.0 h	**2.16**×

Per-direction work is batch size × candidates per instance, the shape of a single decoder call under PDAT. FJSP is reported per optimizer update rather than per epoch, following its training loop.

still saturates the GPU the second loop costs almost nothing (FFSP, and ATSP with GOAL); where it does not, launch overhead and underutilization dominate. SMSP is the clearest case: with a batch of 2 and 64 candidates, each call is far too small to fill an H200, and the doubled call count shows up almost undiminished in the wall-clock.

A likely fix is to concatenate both directions along the batch axis and decode them in one loop, which is what MatPOENet does natively and what keeps its ratio below what its small batch would otherwise imply. The JSSP codebase already contains such an implementation for evaluation (*sample_training_parallel*); we did not port it to training. We report measured numbers rather than optimizing after the fact, but a practitioner adopting PDAT should expect the batched implementation rather than these figures.

One consequence is worth flagging: the overhead grew when we moved these measurements from an RTX 4090 to an H200. SMSP went from 1.23× to 1.90× on the same code. Faster hardware shortens the compute inside each call without changing the number of calls, so the sequential implementation penalty becomes more visible, not less.

Inference time points the same way Table E2 reports inference wall-clock at matched rollout budgets.

Table E2: Inference wall-clock at matched rollout budget (H200)

Problem	Implementation	FWD	BWD	PDAT	PDAT / FWD
ATSP (MatPOENet), N=50	batch-concatenated	6.4 ms	6.4 ms	6.8 ms	1.06×
SMSP, $n = 500$	sequential	3.37 s	3.42 s	3.43 s	1.02×
JSSP, 100×20	sequential	3.42 s	3.40 s	4.07 s	1.19×
FJSP (SD2), 40×10	sequential	10.33 s	10.58 s	13.67 s	1.32×
FFSP, 100×12	sequential	8.5 ms	7.9 ms	13.5 ms	1.59×
PFSP, 200×10	sequential	16.2 ms	16.1 ms	18.6 ms	1.15×
JSSP, 50×20	sequential	0.99 s	1.05 s	2.05 s	2.07×

For SMSP the PDAT column is the dual-trained model run forward, since SMSP admits no $T(I)$ to augment with. Per-instance times, from a single run, measured over the rollout loop only. Constructing the dual instance is preprocessing—it depends only on the instance, not on the rollout budget—and is excluded; on JSSP it costs roughly as much as one rollout pass, once per instance regardless of k. Table E3 gives all 38 problem–size combinations.

Three things follow.

Backward-only costs the same as forward-only. Once the dual instance is built, decoding it is the same computation on the same-shaped tensors. Across all 38 problem–size combinations in Table E3 the ratio averages 1.00 and never leaves 0.74–1.18. The dual is not a more expensive problem to solve; it is the same problem relabeled.

Where the backbone decodes both directions in one batched call, PDAT decoding is essentially free. MatPOENet concatenates along the batch axis, and at a matched budget of 400 rollouts per instance its rollout loop runs at parity with forward-only—6.8 ms against 6.4 ms per instance, within

the spread of repeated runs at this scale. At that budget, 200 rollouts per direction, PDAT reaches 1.5929 against forward-only's 1.6145 (Table F1), so both directions run for the price of one.

Where we decode sequentially, the cost falls as each call gets more work to do. PDAT splits the candidate dimension, so at small sizes each half-sized call is launch-bound and the second one costs nearly a full call, while at large sizes the call is compute-bound and halving it genuinely halves the work. PFSP traces this out cleanly: at each machine count the ratio falls with the number of jobs, from 1.52× at 20×5 to 1.15× at 200×10. JSSP and FJSP also end lower at their largest sizes, at 1.19× and about 1.3×, though not monotonically (Table E3). FFSP is the exception at 1.56–1.71×, because its candidate count is fixed at 24 regardless of job count, so larger instances lengthen the decode without widening the batch. The largest inference overhead we measured is 2.1×; training, in Table E1, reaches 2.16×.

Taken together, this suggests an answer to the question the training table raises. The same paper contains a backbone where bidirectional decoding is essentially free, and others where our implementation makes it expensive in exactly the regime one would predict—small per-call work. This suggests the gap is an implementation choice, concatenating the two directions along the batch axis, rather than a property of the method; we have not implemented that for these backbones.

All measurements on a single H200, over the rollout loop only; constructing the dual instance is preprocessing and is excluded. Times are per instance, converted where the evaluation script reports a total.

Sizes whose per-instance time falls below roughly 20 ms are reported as the median of three runs, because at that scale the measurement is dominated by kernel-launch overhead rather than by the decode itself and a single run can be off by a factor of two. Larger sizes are single runs; we verified stability there (JSSP 100×20 and SMSP $n = 500$ reproduce to within 1% across runs).

Per size, JSSP sits near 2.0× across its mid sizes and drops to 1.19× only at 100×20, FJSP peaks at 1.56× at 20×5 before settling near 1.3×, and backward-only matches forward-only to two decimal places on PFSP.

Table E3: Per-instance inference time (ms), all problems and sizes

Problem	Size	FWD	BWD	PDAT	BWD/FWD	PDAT/FWD
ATSP (MatPOENet)	N=50	6.4	6.4	6.8	1.00	1.06
PFSP	20×5	0.52	0.52	0.79	1.00	1.52
PFSP	50×5	1.51	1.51	2.05	1.00	1.36
PFSP	100×5	4.39	4.38	5.23	1.00	1.19
PFSP	20×10	0.59	0.59	0.93	1.00	1.58
PFSP	50×10	1.68	1.68	2.39	1.00	1.42
PFSP	100×10	4.72	4.72	5.93	1.00	1.26
PFSP	200×10	16.15	16.12	18.59	1.00	1.15
JSSP	15×15	350	260	490	0.74	1.40
JSSP	20×15	300	320	630	1.07	2.10
JSSP	20×20	400	420	820	1.05	2.05
JSSP	30×15	470	480	930	1.02	1.98
JSSP	30×20	610	640	1,230	1.05	2.02
JSSP	50×15	760	780	1,540	1.03	2.03
JSSP	50×20	990	1,050	2,050	1.06	2.07
JSSP	100×20	3,420	3,400	4,070	0.99	1.19
FJSP (SD1)	10×5	458	411	582	0.90	1.27
FJSP (SD1)	20×5	978	982	1,526	1.00	1.56
FJSP (SD1)	15×10	2,063	2,061	2,792	1.00	1.35
FJSP (SD1)	20×10	3,321	3,411	4,354	1.03	1.31
FJSP (SD1)	30×10	6,392	6,512	8,402	1.02	1.31
FJSP (SD1)	40×10	10,723	10,519	13,377	0.98	1.25
FJSP (SD2)	10×5	441	410	550	0.93	1.25
FJSP (SD2)	20×5	1,009	1,017	1,571	1.01	1.56
FJSP (SD2)	15×10	2,039	2,182	2,991	1.07	1.47
FJSP (SD2)	20×10	3,359	3,474	4,505	1.03	1.34
FJSP (SD2)	30×10	6,357	6,565	8,316	1.03	1.31
FJSP (SD2)	40×10	10,334	10,578	13,672	1.02	1.32
FFSP	20×12	3.6	3.2	5.6	0.89	1.56
FFSP	50×12	4.9	5.0	8.4	1.02	1.71
FFSP	100×12	8.5	7.9	13.5	0.93	1.59
FJSP (bench)	Brandimarte	2,250	2,464	3,280	1.10	1.46
FJSP (bench)	Hurink edata	2,405	2,505	3,387	1.04	1.41
FJSP (bench)	Hurink vdata	2,565	2,548	3,499	0.99	1.36
FJSP (bench)	Hurink rdata	2,475	2,435	3,384	0.98	1.37
SMSP	n=50	22	26	24	1.18	1.09
SMSP	n=100	78	80	84	1.03	1.08
SMSP	n=500	3,366	3,424	3,430	1.02	1.02

F ATSP MatPOENet — Full Budget Grid and Provenance

Setup MatPOENet is a POMO-style REINFORCE solver with a pseudo one-hot positional embedding derived from a nearest-neighbor rough tour. We follow the authors' settings exactly: $N = 50$, tmat-class instances generated on the fly, 512-dimensional embeddings, 8 encoder layers, 16 heads, lr 4e-4, 10,000 instances per epoch, 2,000 epochs, and the first 1,000 instances of their released test set. Forward-only and backward-only use batch 200; PDAT concatenates the two directions along the batch axis, so we report both budget conventions—**step-matched** at batch 100 (identical rollouts per step, twice the optimizer updates) and **update-matched** at batch 200 (identical updates, twice the rollouts).

On the choice of scale. The two ATSP experiments run at different N. GOAL is evaluated at $N = 100$ because that is the only scale for which the authors release both training data and a test set. MatPOENet is evaluated at $N = 50$ to keep four complete 2,000-epoch training runs within our compute budget; at $N = 100$ a single run takes 36 hours on our hardware. The two experiments are therefore not comparable to each other in absolute terms. Each is internally controlled—same

data, same hyperparameters, same evaluation across its own variants—and we read them as two independent tests of whether the ATSP dual helps, not as a scale study.

We evaluate every model under two inference budgets and, orthogonally, with and without dual augmentation. Since PDA applies to any checkpoint, we run it for all three so that the training effect is separated from the augmentation effect. All numbers are the mean and standard deviation over five seeds on 1,000 test instances.

Table F1: ATSP, MatPOENet backbone ($N = 50$, mean tour length over 5 evaluation seeds; one trained model per variant)

Method	Inference ×1		Inference ×8		
	no PDA	with PDA	no PDA	with PDA, ×4 each	with PDA, ×8 each
Rollouts / instance	50	100	400	400	800
FWD (MatPOENet)†	1.6551 ± 0.0005	1.6365 ± 0.0007	1.6145 ± 0.0007	1.6121 ± 0.0006	1.6027 ± 0.0003
BWD	1.6540 ± 0.0009	1.6333 ± 0.0005	1.6129 ± 0.0005	1.6105 ± 0.0004	1.6010 ± 0.0003
PDT / PDAT (step-matched)	**1.6247 ± 0.0004**	**1.6110 ± 0.0004**	**1.5949 ± 0.0002**	**1.5929 ± 0.0003**	**1.5858 ± 0.0002**
PDT / PDAT (update-matched)	1.6308 ± 0.0003	1.6159 ± 0.0003	1.5986 ± 0.0004	1.5960 ± 0.0002	1.5889 ± 0.0002
training effect (step-matched)	*−1.83%*	*−1.56%*	*−1.21%*	*−1.19%*	*−1.05%*
training effect (update-matched)	*−1.47%*	*−1.26%*	*−0.98%*	*−1.00%*	*−0.86%*

†Forward-only is MatPOENet with the same code, hyperparameters and 2,000 epochs, run by us; the data-provenance note below explains why our absolute values sit above the published 1.5643. The "with PDA" columns apply dual augmentation; the "×4 each" column splits the eight augmentations between the two directions and so matches the 400 rollouts of forward-only, and is the one reported in Table 1. The "×8 each" column doubles the rollouts. Without PDA the dual-trained rows are PDT. The training effect rows compare the dual-trained model against FWD (MatPOENet) within each column, so the inference procedure is held fixed.

Both dual-trained variants beat both baselines in all five inference settings, and the five-seed ranges never overlap—the worst PDAT seed beats the best baseline seed everywhere. Four observations follow.

The result does not depend on which budget axis is fixed. The step-matched variant holds rollouts per step equal to the baselines and takes twice as many optimizer updates; the update-matched one holds updates equal and takes twice the rollouts. Both improve on forward-only by 0.9–1.8%, the former ahead by 0.003–0.006 throughout. Whichever convention a reader prefers, the conclusion is the same, which is why we report both rather than choosing one.

Both components contribute, and they partly overlap. The leftmost column holds inference fixed at 50 forward rollouts and still shows a 1.83% training effect. PDA helps every checkpoint, including the forward-only one (1.6551 → 1.6365), confirming that the two directions produce genuinely different tours. A dual-trained model gains less from it, 0.84% against 1.12% at ×1 and 0.57% against 0.73% at ×8, as expected if training already captures part of what the backward direction adds.

Forward and backward are nearly equivalent directions. BWD is ahead of FWD in all five settings, but only by 0.07–0.20%, close to what a distributionally symmetric dual predicts, and each variant is a single trained model. PDAT beats the better of the two by 0.95–1.77%, seven to twenty-five times that margin, which rules out the reading that PDAT merely benefits from one direction happening to be easier.

Augmentation and directional diversity partially overlap. The margin narrows from 1.83% at ×1 to 1.21% at ×8. Instance augmentation is itself an exploration mechanism, and once 400 candidates per instance are drawn, part of what the backward stream contributes is already covered. The ordering is unchanged, but the effect size is budget-dependent, and reporting only the largest number would misrepresent it.

Data provenance Training instances are generated on the fly by the authors' *get_random_problems*, which draws integer entries uniformly from $[0, 10^6]$, zeroes the diagonal, applies a Floyd–Warshall closure until the triangle inequality holds, and rescales by 10^6. Note that this differs from the description in the UniCO paper, which specifies ATSP matrices drawn from Uniform$(0, 1)$ without a metric closure; we follow the released code. Evaluation uses the authors' released *test_set/*

50_10000, whose instances violate the triangle inequality on 21% of entry triples. We apply this combination identically to every variant.

Our absolute numbers do not match the 1.5643 reported for MatPOENet-8× at $N = 50$. Evaluating the authors' released checkpoint through our pipeline gives 1.6443, and through their own *test.py* gives 1.6456—consistent with each other, and *worse* than every one of our trained models. We ruled out four explanations:

- *Evaluation code.* Two independent implementations—ours and the authors' *test.py*—agree to within 0.0013.
- *Positional-embedding initialization.* The checkpoint was trained with the nearest-neighbor rough solver enabled; disabling it at inference degrades it further, to 1.6575.
- *Test-set composition.* The released 10,000-instance set contains only ATSP instances, uniform in scale and value distribution across the index range; it is not a mixture of the four task types used elsewhere in UniCO.
- *Training distribution.* Evaluated in a single run on freshly generated triangle-closed instances and on the released test set, the released checkpoint scores within 0.0004 across the two, so the metric/non-metric distinction does not account for the gap.

We did not resolve where the published number comes from and do not speculate further. What matters for the comparison in Table F1 is that all methods share one data generator, one test set, one evaluation script, and one set of hyperparameters; the absolute values are internally consistent but should not be read against the published table.

This second ATSP backbone tests whether the GOAL result transfers across learning paradigms: GOAL is trained by supervised imitation of solutions from the LKH heuristic solver, MatPOENet by REINFORCE with POMO baselines. It shows the same direction: 1.2% of tour length at a matched 400 rollouts.

G Ablation on the Loss-Combination Rule

Table G1: PFSP, loss combination strategies (Taillard gap %, 5 evaluation runs)

Update rule	Epochs	20×5	20×10	50×5	50×10	100×5	100×10	200×10	Avg
Summed loss	1000	**0.32 ± 0.07**	**3.23 ± 0.08**	**0.15 ± 0.02**	**4.11 ± 0.07**	**0.15 ± 0.02**	**2.10 ± 0.04**	**1.70 ± 0.05**	**1.68**
Better direction only	1000	0.33 ± 0.03	3.50 ± 0.13	0.16 ± 0.02	4.18 ± 0.19	0.16 ± 0.03	2.14 ± 0.13	1.74 ± 0.06	1.74
Separate steps	500	0.37 ± 0.05	3.36 ± 0.10	0.17 ± 0.01	4.15 ± 0.08	0.16 ± 0.03	2.17 ± 0.12	1.75 ± 0.07	1.73
Separate steps	1000	0.35 ± 0.06	3.49 ± 0.23	0.18 ± 0.03	4.20 ± 0.09	0.19 ± 0.02	2.25 ± 0.11	2.06 ± 0.15	1.82
(FWD)	*1000*	*0.40 ± 0.03*	*3.38 ± 0.06*	*0.21 ± 0.03*	*4.20 ± 0.16*	*0.17 ± 0.02*	*2.26 ± 0.13*	*1.72 ± 0.09*	*1.76*

Summed loss adds both directions' losses and takes one optimizer step; *better direction only* backpropagates only the direction with higher reward; *separate steps* takes one optimizer step per direction, so at 500 epochs it performs the same number of parameter updates as the other two do at 1000.

Summing has the lowest average gap and the lowest mean at every individual size. The gap to the better-direction rule (1.68 vs 1.74) is consistent with that rule discarding one direction's gradient entirely on each batch: half the computed signal is thrown away.

The separate-step rows isolate the update-count confound Run at 500 epochs, matching total parameter updates, separate steps reach 1.73. Run at 1000 epochs—twice the updates of every other row—it *degrades* to 1.82, worse than forward-only. The regression is concentrated at 200 × 10 (2.06 vs 1.75), the largest zero-shot size. Applying the two directions' gradients as separate sequential steps therefore does not simply learn faster; past a point it overfits, and the extra updates hurt generalization to unseen sizes. This is the ablation's most useful outcome: it shows that running the separate-step rule for half the epochs is not bookkeeping for a fair comparison but what that rule needs in order to work.

Standard deviations of 0.01–0.23 mean individual columns are not always separated between the three rules. The average column and the consistency of the per-size ordering carry the conclusion: summing both directions' losses into a single step is both the best-performing and the simplest of the three, and we use it everywhere.

H FJSP PUBLIC BENCHMARKS

Models trained on SD2 10×5, evaluated zero-shot on Brandimarte and the three Hurink families.

Table H1: FJSP public benchmarks, SD2-trained models (gap % vs OR-Tools reference, 5 evaluation runs)

Method	Brandimarte (10)	Hurink edata (40)	Hurink vdata (40)	Hurink rdata (40)
FWD (PGMPO)†	4.70 ± 0.58	6.50 ± 0.22	1.07 ± 0.03	4.85 ± 0.13
BWD	5.48 ± 0.12	6.60 ± 0.15	1.12 ± 0.05	5.27 ± 0.04
FWD + PDA	**4.55 ± 0.39**	**5.72 ± 0.08**	0.97 ± 0.04	4.92 ± 0.08
BWD + PDA	4.86 ± 0.19	5.79 ± 0.13	1.07 ± 0.01	5.02 ± 0.17
PDAT	4.73 ± 0.37	6.01 ± 0.16	**0.86 ± 0.04**	**4.70 ± 0.09**

Instance counts in parentheses. †Forward-only is PGMPO at its published configuration. Gaps are per-instance against the OR-Tools solutions released with the backbone, averaged. The + PDA rows apply dual augmentation to the single-direction checkpoints, so the PDAT row can be compared at a fixed inference procedure as in Table 1.

The decomposition splits the four benchmarks in half. **Augmentation alone helps on three of four**, by 0.10 to 0.78 points, and is neutral on Hurink rdata (+0.07, within the spread). **Adding training helps on two** (Hurink vdata -0.11, rdata -0.22) and costs on the other two (Brandimarte +0.18, edata +0.29). Against the original forward-only baseline PDAT improves the mean on three of four and is within the evaluation spread on the fourth, but that aggregate hides the split: on Brandimarte and edata, augmentation alone is the better configuration.

Backward-only without augmentation is the weakest row on all four, trailing forward-only by 0.05–0.78 points. This is the one part of our study where a single direction is consistently worse rather than merely different, and augmenting it recovers most of the gap (5.48 $\rightarrow$ 4.86 on Brandimarte), so half a budget spent on the weaker direction is not wasted.

I FULL RESULT TABLES

One table per problem, at every evaluated size, in the order of Table 1. MatPOENet's grid spans two budget conventions and is reported separately in Table F1. Lower is better throughout; except in Table I1, bold marks the lowest value in each column, all tied values are marked, and a column where every variant ties is left unmarked. Published reference rows are excluded from the comparison.

Table I1: ATSP, GOAL backbone (128 instances, gap %, 5 evaluation seeds; one trained model per variant)

Method	Training	Inference	Cand.	Gap
Released, full budget	forward	greedy on I	1	**3.457 ± 0.059**
Released, full budget	forward	greedy on $T(I)$	1	3.483 ± 0.050
Released $\times 2$ (control)	forward	greedy twice on I	2	3.330 ± 0.042
Released + PDA	forward	greedy on I, $T(I)$	2	**2.663 ± 0.060**
FWD (GOAL)§	forward	greedy on I	1	4.667 ± 0.027
BWD	backward	greedy on $T(I)$	1	4.501 ± 0.169
PDT	bidirectional	greedy on I	1	**4.203 ± 0.046**
FWD $\times 2$ (control)	forward	greedy twice on I	2	4.394 ± 0.045
FWD + PDA	forward	greedy on I, $T(I)$	2	3.624 ± 0.072
PDAT	bidirectional	greedy on I, $T(I)$	2	**3.272 ± 0.029**

§Reduced training budget. The first four rows are the authors' released checkpoint, trained at full budget and evaluated in our pipeline. Every tour is scored by its closed ATSP length; GOAL's decoder reports 0.16% for the released checkpoint because it counts the final arc in the wrong direction (Appendix C). Bold marks the best at each candidate count within each block. BWD + PDA reaches 3.365 ± 0.021.

Table I2: PFSP, Taillard benchmark (gap % vs best-known, 5 evaluation runs)

Method	20×5	50×5	100×5	20×10	50×10	100×10	200×10	Avg
FWD (MP-ASIL)†	0.40	0.21	0.17	3.38	4.20	2.26	1.72	1.76
BWD	0.39	**0.15**	0.20	3.52	4.20	2.13	1.86	1.78
FWD + PDA‡	0.37	**0.15**	0.15	3.30	4.19	2.12	1.72	1.71
BWD + PDA‡	0.35	0.17	**0.14**	3.34	**4.10**	2.16	**1.69**	1.71
PDAT	**0.32**	**0.15**	0.15	**3.23**	4.11	**2.10**	1.70	**1.68**

†Forward-only is MP-ASIL: same code, hyperparameters and 1000 epochs, run by us. ‡From a separate script that draws both candidate sets in one pass; within that run forward-only averages 1.79, so the augmentation effects quoted below are computed inside it.

Table I3: JSSP, Taillard benchmark (gap % vs best-known, 5 evaluation runs)

Method	15×15	20×15	20×20	30×15	30×20	50×15	50×20	100×20	Avg
MP-ASIL (published, K=256)†	6.8	8.5	8.7	10.4	12.8	4.2	7.0	1.0	7.4
FWD (MP-ASIL, K=64)§	7.07	8.91	9.38	11.04	13.23	7.33	8.21	2.82	8.50
BWD	7.12	9.18	8.80	12.03	13.59	6.38	8.89	4.58	8.82
FWD + PDA	6.71	**8.70**	8.92	**10.39**	**12.43**	**5.64**	**7.40**	**2.54**	**7.84**
BWD + PDA	6.60	8.97	**8.70**	10.72	13.05	6.25	7.84	3.61	8.22
PDAT	**6.29**	8.74	8.71	11.08	12.89	6.20	7.75	3.34	8.13

†Published result, at the original training and inference budget. §Reduced training budget (Appendix K).

Table I4: FJSP SD2, trained on 10×5 (makespan, 5 evaluation runs)

Method	10×5	20×5	15×10	20×10	30×10	40×10
FWD (PGMPO)†	356.43	620.14	520.86	552.75	726.13	915.13
BWD	357.82	619.92	521.23	553.68	725.88	914.87
FWD + PDA	354.78	617.52	**518.63**	554.15	724.40	911.30
BWD + PDA	355.43	618.08	519.79	554.32	725.55	912.11
PDAT	**354.07**	**615.61**	518.86	**549.24**	**716.53**	**903.56**

†Forward-only is PGMPO at the authors' settings. Gaps against the OR-Tools reference, across the six sizes: FWD 9.37 / 3.01 / 38.44 / 19.19 / 4.92 / −6.46, PDAT 8.67 / 2.27 / 37.94 / 18.43 / 3.54 / −7.64. These values come from one training run; Table J6 reports the same comparison for two further runs, where the training effect at the zero-shot sizes does not reproduce.

Table I5: FJSP SD1, same protocol (makespan, 5 evaluation runs)

Method	10×5	20×5	15×10	20×10	30×10	40×10
FWD (PGMPO)†	99.19	190.30	150.23	189.73	274.03	363.84
BWD	99.02	190.27	150.42	190.05	274.40	364.18
FWD + PDA	98.91	189.40	150.01	**189.60**	**273.94**	**363.62**
BWD + PDA	**98.90**	**189.37**	150.36	189.93	274.31	364.09
PDAT	99.04	190.57	**149.79**	190.29	274.93	364.71

$K = 100$. *†Forward-only is PGMPO at the authors' settings. These values come from one training run; Table J6 reports two further runs, in which the training effect has the opposite sign at all six and at five of the six sizes respectively.*

Table I6: FFSP, three sizes (makespan, 5 evaluation runs)

Method	20×12	50×12	100×12
FWD (MP-ASIL)‡	26.880 ± 0.014	51.128 ± 0.025	90.926 ± 0.025
BWD	26.874 ± 0.029	51.166 ± 0.017	90.874 ± 0.022
FWD + PDA	26.747 ± 0.022	51.012 ± 0.016	90.702 ± 0.026
BWD + PDA	26.750 ± 0.017	50.987 ± 0.023	**90.675 ± 0.020**
PDAT	**26.705 ± 0.014**	**50.983 ± 0.023**	90.700 ± 0.017
MP-ASIL (Large)*	25.6	49.9	89.7

*‡Forward-only is MP-ASIL at its published configuration. *As reported by MP-ASIL (Choi et al., 2025); spends roughly three times our inference budget and is the reference for the gaps quoted below.*

Table I7: SMSP, mean normalized total weighted tardiness (trained on $n = 50$)

Method	$n = 50$	Gap†	$n = 100$	Gap†	$n = 500$	Gap†
FWD (MP-ASIL)‡	0.2181	0.00%	0.2422	+1.94%	0.3788	+2.80%
FWD + PDA	0.2181	0.00%	0.2433	+2.40%	0.3797	+3.04%
BWD	0.2181	0.00%	**0.2367**	**−0.38%**	**0.3324**	**−9.80%**
FWD, run 2	0.2186	+0.23%	0.2390	+0.59%	0.5498	+49.20%
BWD, run 2	0.2181	0.00%	0.2367	−0.38%	0.3321	−9.88%
PDAT	0.2181	0.00%	0.2368	−0.34%	0.3704	+0.52%
MP-ASIL (Large)*	0.2181	0.00%	0.2376	0.00%	0.3685	0.00%

†Computed from the mean values in this table, not as an average of per-instance gaps, because the reference is published only as a mean. The reference uses 3.7× more inference time. ‡Forward-only is MP-ASIL at $k = 128$. Each size is evaluated on 100 test instances; re-evaluating a checkpoint changes entries only in the fourth decimal (0.2422 against 0.2423 for forward-only at $n = 100$). The run-2 rows are a second training run of each single-direction variant, on an H200; all other rows come from the first run, on an RTX 4090, and PDAT was trained only once.

J Supporting Measurements

J.1 Forward–backward candidate correlation

Table J1 tests whether the similarity of the two directions' candidate sets predicts the sign of the training effect. For each instance we take the mean objective value of its forward candidates and of its backward candidates, both drawn from the forward-trained checkpoint and in the units of the corresponding result table, and correlate the two across instances. The ATSP value uses the closed-length scoring of Appendix C, like every GOAL number. It does not separate the cases: ATSP, where training helps, and both FJSP data sets, where its sign changes between runs, all sit above 0.9, and the statistic is instance-level, so part of each correlation reflects difficulty rather than candidate similarity (Section 5.4).

J.2 FFSP across three training runs

Table J2 repeats FFSP 20×12 from three initializations. The combined PDAT margin over forward-only holds in every run, while the training component alone is the size of the run-to-run spread, which is why we do not claim its sign there.

J.3 Training curves

Figure J1 evaluates checkpoints saved every ten epochs across three independent training runs, so the comparison holds at every point of training rather than only at convergence: every PDAT run is below every forward-only and backward-only run at every checkpoint, by 0.13 to 0.20 makespan over the forward-only mean. Table J3 lists the values.

Table J1: Across-instance correlation of mean candidate cost, forward against backward

Problem	Correlation	Training effect
ATSP (GOAL)	0.97	$-$**0.352** (helps)
FFSP 20×12	0.52	-0.042 (comparable to run-to-run spread)
FFSP 100×12	0.62	-0.002 (within run-to-run spread)
JSSP	0.79	**+0.29** (costs)
FJSP-SD2	0.93	sign changes between training runs (Table J6)
FJSP-SD1	0.96	sign changes between training runs (Table J6)

Correlations are single measurements on a forward-only checkpoint. The training effect is the step from FWD + PDA to PDAT at fixed inference; negative favors training.

Table J2: FFSP 20×12 **across three independent training runs (makespan)**

Training run	FWD, forward inference	PDAT, $K/2{+}K/2$
1	26.874 ± 0.015	26.706 ± 0.004
2	26.895 ± 0.041	26.735 ± 0.020
3	26.861 ± 0.021	26.735 ± 0.028
spread across runs	**0.034**	**0.029**

Each cell is the mean over five evaluation runs of one trained model. Run 1 is the run reported in Table I6; the values here come from a separate evaluation of the same checkpoints, which is why its forward-only figure differs from Table I6 by 0.006. Backward-only is omitted because the claim this table supports concerns the margin over forward-only; its three runs appear in Table J3 and Figure J1.

Table J3: Data behind Figure J1 (makespan; each variant over three independent training runs)

Epoch	10	20	30	40	50	60	70	80	90	100
FWD (mean ± sd)	27.055 ± 0.015	26.990 ± 0.053	26.969 ± 0.052	26.959 ± 0.043	26.923 ± 0.019	26.902 ± 0.012	26.891 ± 0.018	26.889 ± 0.014	26.877 ± 0.037	26.853 ± 0.052
BWD (mean ± sd)	27.044 ± 0.021	26.961 ± 0.012	26.953 ± 0.037	26.903 ± 0.048	26.903 ± 0.005	26.865 ± 0.052	26.899 ± 0.019	26.871 ± 0.024	26.842 ± 0.029	26.882 ± 0.036
PDAT (mean ± sd)	**26.877** ± 0.034	**26.843** ± 0.027	**26.766** ± 0.013	**26.794** ± 0.009	**26.741** ± 0.032	**26.752** ± 0.011	**26.737** ± 0.017	**26.729** ± 0.024	**26.749** ± 0.018	**26.722** ± 0.012

The FWD and PDAT runs are the three of Table J2, and the three BWD runs are trained the same way. The curve values come from a separate evaluation pass over every saved checkpoint. Because evaluation samples its K latent vectors at random, re-evaluating a checkpoint moves its score by about the ± shown in Table J2, which is why the epoch-100 values differ by up to 0.024 from the average of Table J2's three per-run values (26.877 for FWD against 26.853 here).

J.4 Relabeling control

One might ask whether any objective-preserving change of representation would do as well as the dual. We test this on JSSP and on FFSP. For JSSP we fix the forward-only checkpoint and $K = 64$, and draw the second $K/2$ candidates from a relabeled copy of each Taillard instance instead of from $T(I)$: a random permutation of the jobs, of the machine labels, or of both. All five candidate sets share the same first $K/2$ forward draw, so each is compared with the split control instance by instance. Job relabeling behaves like a fresh forward draw, as expected of an encoder that is equivariant to job order. Every relabeling does come out slightly ahead of the split control in both runs, but by only 0.06–0.13 on average, with per-instance wins and losses nearly balanced; the dual gains 0.73, winning in 69 and losing in 39 of the 160 instance–run pairs (80 instances, two runs). For FFSP we fix the forward-only checkpoint, $K = 24$ and the 1,000 test instances of 20×12, and relabel the jobs with one order shared by all stages, the machines independently within each stage, or both. Machine relabeling is indistinguishable from a fresh forward draw, as expected: MatNet already gives machines random one-hot initial embeddings, so renaming them adds no new input. Job relabeling is slightly worse in both runs. The dual again gains in both.

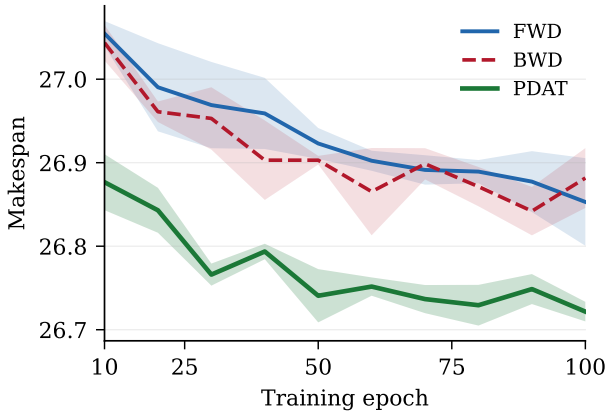


Figure J1: **FFSP** 20×12 **makespan against training epoch** (1,000 test instances). Lines are means over three independent training runs per variant and shaded bands one standard deviation. PDAT is below both single-direction baselines at every checkpoint, while the forward-only and backward-only bands overlap throughout. The PDAT curve uses bidirectional inference and the other two do not, so the gap combines both components; at 100 epochs augmentation accounts for 0.133 of it and training for 0.042 (Table I6).

Table J4: Relabeling versus the dual, two runs each (JSSP: $K = 64$, 80 Taillard instances, gap %; FFSP: $K = 24$, 1,000 instances of 20×12, makespan)

Problem	Second $K/2$ drawn from	Run 1	Run 2	Change vs split (mean)	Better / worse / tie (both runs)
JSSP	Original I (split control)	9.762	9.706	—	—
JSSP	Job-relabeled I	9.760	9.593	−0.058	47 / 45 / 68
JSSP	Machine-relabeled I	9.552	9.664	−0.126	52 / 41 / 67
JSSP	Job- and machine-relabeled I	9.658	9.643	−0.084	48 / 41 / 71
JSSP	Dual $T(I)$	**9.088**	**8.917**	**−0.732**	69 / 39 / 52
FFSP	Original I (split control)	26.864	26.910	—	—
FFSP	Job-relabeled I	26.893	26.977	+0.048	373 / 433 / 1194
FFSP	Machine-relabeled I	26.868	26.907	+0.001	406 / 393 / 1201
FFSP	Job- and machine-relabeled I	26.853	26.885	−0.018	408 / 395 / 1197
FFSP	Dual $T(I)$	**26.708**	**26.787**	**−0.140**	521 / 381 / 1098

JSSP entries are reference gaps (%) and FFSP entries makespan. The last column counts instance–run pairs pooled over the two runs, 160 for JSSP (80 instances) and 2,000 for FFSP (1,000 instances). Runs differ in the sampling seed and in the random relabelings. Every relabeling preserves the makespan of every schedule exactly, as the dual does. Bold marks the lowest value in each column within each problem.

J.5 Solution-space diversity

With the same forward-only JSSP checkpoint, for each of the 80 Taillard instances we draw a reference set C_F of 32 forward candidates and four further sets of 32: a second forward draw, and draws on $T(I)$, on a job-relabeled copy and on a machine-relabeled copy of I. Every candidate is mapped back to a schedule of I—backward candidates through ϕ^{-1}, which reverses each machine's job order, and relabeled ones through the inverse permutation—and represented by its machine orders. The distance between two schedules is the normalized Kendall-τ distance between their job orders, averaged over machines. We report the mean distance between a set and C_F, and its novelty, the mean distance from each of its candidates to the nearest candidate in C_F. Recomputing the makespan of 320 extracted schedules on I from their machine orders reproduces the model's value in every case, including backward schedules mapped through ϕ^{-1}; this checks both the extraction and the exactness of the dual.

Table J5: Distance of each candidate set from a forward reference set (JSSP, 80 Taillard instances)

Second set drawn from	Mean distance to C_F	Novelty	Novelty relative to a second forward draw
I (a second forward draw)	0.0878	0.0670	1.00
Job-relabeled I	0.0879	0.0675	1.01
Machine-relabeled I	0.0880	0.0673	1.00
Dual $T(I)$	**0.1262**	**0.1108**	**1.65**

Novelty of the backward set exceeds that of the second forward draw on all 80 instances. Across instances, the excess novelty and the PDA gain over the split control are essentially uncorrelated (Spearman 0.05, Pearson 0.10). Bold marks the largest value in each column.

J.6 FJSP ACROSS THREE TRAINING RUNS

We repeated both the forward-only and the dual-trained training on both FJSP data sets from two further initializations, at the same configuration and budget, and evaluated every size with the five-run protocol of Tables I4 and I5. The code and hyperparameters are identical across the three runs; only the training seed differs.

Table J6: FJSP training effect over three independent training runs (makespan, mean of 5 evaluation runs)

Family	Size	Run 1	Run 2	Run 3	Mean	Agree in sign
SD2	10×5	−0.71	−0.65	−0.41	−0.59	yes
SD2	20×5	−1.91	+1.19	+0.95	+0.08	no
SD2	15×10	+0.23	−0.67	−1.42	−0.62	no
SD2	20×10	−4.91	+0.52	+0.67	−1.24	no
SD2	30×10	−7.87	+4.58	+5.07	+0.59	no
SD2	40×10	−7.74	+7.35	+8.63	+2.75	no
SD1	10×5	+0.13	−0.25	−0.07	−0.06	no
SD1	20×5	+1.17	−0.36	−0.43	+0.13	no
SD1	15×10	−0.22	+0.26	−0.33	−0.10	no
SD1	20×10	+0.69	−0.97	−0.91	−0.40	no
SD1	30×10	+0.99	−1.39	−1.56	−0.65	no
SD1	40×10	+1.09	−1.72	−1.74	−0.79	no

Training effect is PDAT minus FWD + PDA at fixed inference; negative favors training. Run 1 is the run reported in Tables I4 and I5. Table J7 gives the augmentation step on the same checkpoints.

J.7 BUDGET DEPENDENCE OF THE AUGMENTATION MARGIN

On JSSP the margin does not depend on the sample count On the same 80 Taillard instances, bidirectional sampling improves on original-only sampling by 0.66 gap points at K=64 and 0.69 at K=512 (Table 2, five runs each), even though the eightfold larger forward budget itself lowers the original-only gap by 1.28 points. Drawing more samples of the same kind therefore does not erode what the dual adds.

It may shrink when diversity comes from another source For the forward-trained MatPOENet checkpoint, adding PDA improves the tour length by 1.12% at ×1 instance augmentation and by 0.73% at ×8 (Table F1), where the extra candidates come from a second augmentation rather than from more samples; in both comparisons PDA doubles the rollout count. At a matched 400 rollouts, where the dual replaces four of the eight augmentations instead of adding to them, the gain is only 0.15%. This is consistent with a coverage account—sources of diversity that reach overlapping regions of the solution space substitute for one another—but it is a single backbone and not a controlled sweep, so we treat it as a hypothesis.

Table J7: FJSP augmentation effect over the same three training runs (makespan, mean of 5 evaluation runs)

Family	Size	Run 1	Run 2	Run 3	Improves in
SD2	10×5	−1.65	−1.61	−1.85	3 of 3
SD2	20×5	−2.62	−2.91	−2.57	3 of 3
SD2	15×10	−2.23	−2.32	−2.18	3 of 3
SD2	20×10	+1.40	−0.11	−1.10	2 of 3
SD2	30×10	−1.73	−2.99	−2.89	3 of 3
SD2	40×10	−3.83	−3.31	−2.89	3 of 3
SD1	10×5	−0.28	−0.44	−0.37	3 of 3
SD1	20×5	−0.90	−0.18	−0.22	3 of 3
SD1	15×10	−0.22	−0.12	−0.07	3 of 3
SD1	20×10	−0.13	−0.09	−0.05	3 of 3
SD1	30×10	−0.09	−0.05	−0.04	3 of 3
SD1	40×10	−0.22	−0.22	−0.23	3 of 3

Augmentation effect is FWD + PDA minus FWD on the same forward-only checkpoint; negative favors augmentation. Run 1 is the run reported in Tables I4 and I5. It improves in 35 of 36 checkpoint-and-size combinations; the exception is SD2 run 1 at 20×10.

K Code Sources and Per-Problem Settings

Code sources All backbones are run from their authors' released code: PGMPO from https://github.com/Ingukchoi/PGMPO; MP-ASIL from its supplementary material (https://papers.nips.cc/paper_files/paper/2025/file/2330c4be5e12c504705ba1db8a507966-Supplemental-Conference.zip); MatPOENet from the UniCO repository (https://github.com/Thinklab-SJTU/UniCO); and GOAL from https://github.com/naver/goal-co/ (Drakulić et al., 2025).

Per-problem settings *ATSP.* Two backbones with different learning paradigms. **GOAL** (Drakulić et al., 2025) is a generalist transformer trained by supervised imitation of solutions from the LKH heuristic solver; we train on N=100 tmat instances at a reduced budget (a fifth of the authors' instance pool for a shorter schedule, about a thirteenth of their total training work) and evaluate on the released 128-instance test set. **MatPOENet** (Pan et al., 2025) is a POMO-style REINFORCE solver at N=50, at the authors' settings. Appendix D gives hyperparameters.

PFSP. MP-ASIL (Choi et al., 2025) at its published configuration—no budget reduction—trained on 20×5 and 20×10 with $K = 128$ latent-conditioned policies, evaluated on Taillard.

JSSP. MP-ASIL on the LEHD-based pointer network of SLIM (Corsini et al., 2024), following the reference protocol: 30,000 instances per epoch across six shapes, Taillard evaluation with $k = 512$. We reduce the training policies from 256 to 64 and epochs from 20 to 15 for compute reasons; all variants share the reduced budget.

FJSP. PGMPO (Choi et al., 2026), trained with a Bradley–Terry preference objective that pairs the best solution among its $C = 128$ preference-conditioned policies against the others, rather than with the objective value as a reward, at the authors' settings, trained on 10×5 with 100 sampled solutions at evaluation. The environment dimension provides the split axis. We train on both data sets the authors provide: **SD2** samples each operation–machine time independently from $U[1, 99]$, so machine choice substantially changes duration; **SD1** draws a per-operation mean and perturbs it narrowly, so assignment affects duration little. The dual is the FJSP extension of Appendix A; we verified exactness numerically during training, the two ways of computing the backward makespan agreeing to machine precision on every batch.

FFSP. MatNet-based MP-ASIL on the standard 3-stage, 4-machines-per-stage FFSP, at the authors' hyperparameters, with 1,000 test instances per size and $K = 24$ latent-conditioned policies. The dual reverses the stage order. Each size is trained separately.

SMSP. MP-ASIL with the dynamic dual of Section 3, at the original hyperparameters, trained on $n = 50$ with $K = 128$ and evaluated on 100 test instances at each $n \in \{50, 100, 500\}$ (Table I7).

Row labels On four of the six problems the forward-only variant is the published method at its published configuration—same code, hyperparameters and epochs—so we name the backbone in those rows and omit a separate published row, noting any reproduction discrepancy in a footnote. The BWD rows carry no backbone name because no published method trains in that direction: they are our own variant, included so the two directions can be compared. For ATSP (GOAL) and JSSP we trained below the original budget, marked §, and the published result appears as its own row.

Candidates For the four MP-ASIL backbones (PFSP, JSSP, FFSP, SMSP) the K candidates come from a *single* policy conditioned on K latent vectors, and for PGMPO (FJSP) from a single policy run in K preference-conditioned environments—in neither case from K starting points as in POMO. PDA splits exactly this dimension. Several implementations name it *pomo_size* for historical reasons; we call it K and reserve "POMO" for multi-start.

ATSP evaluation. GOAL draws a fresh random scalar per node at every forward pass, so greedy decoding is stochastic across runs: different seeds shift the gap by up to 0.5 points. We therefore report mean and standard deviation over five seeds for ATSP.