Algorithm 2 Poppy training with starting points

Input: problem distribution $\mathcal{D}$, number of starting points per instance P, number of agents K, batch size B, number of training steps H, a pretrained encoder h_ψ and decoder q_ϕ
$\phi_1, \phi_2, \dots, \phi_K \leftarrow \texttt{CLONE}(\phi)$ {Clone the pre-trained decoder parameters K times.}
for step 1 to H **do**
  $\rho_i \leftarrow \texttt{Sample}(\mathcal{D})\ \forall i \in 1, \dots, B$
  $\alpha_{i,1}, \dots, \alpha_{i,P} \leftarrow \texttt{SelectStartPoints}(\rho_i, P)\ \forall i \in 1, \dots, B$
  $\tau^k_{i,p} \leftarrow \texttt{Rollout}(\rho_i, \alpha_{i,p}, h_\psi, q_{\phi_k})\ \forall i \in 1, \dots, B, \forall p \in 1, \dots, P, \forall k \in 1, \dots, K$
  $b^k_i \leftarrow \frac{1}{P} \sum_p R(\tau^k_{i,p})$
  $k^*_{i,p} \leftarrow \arg\max_{k \leq K} R(\tau^k_{i,p})\ \forall i \in 1, \dots, B, \forall p \in 1, \dots, P$ {Select the best agent per (instance, starting point).}
  $\nabla L(h_\psi, q_{\phi_1}, q_{\phi_2}, \dots, q_{\phi_K}) \leftarrow -\frac{1}{BP} \sum_{i,p} (R(\tau^{k^*_{i,p}}_{i,p}) - b^{k^*_{i,p}}_i) \nabla \log p_{\psi, \phi_{k^*_{i,p}}}(\tau^{k^*_{i,p}}_{i,p})$ {Propagate gradients through these only.}
  $(h_\psi, q_{\phi_1}, q_{\phi_2}, \dots, q_{\phi_K}) \leftarrow (h_\psi, q_{\phi_1}, q_{\phi_2}, \dots, q_{\phi_K}) - \alpha \nabla L(h_\psi, q_{\phi_1}, q_{\phi_2}, \dots, q_{\phi_K})$

A Additional Details on Poppy

A.1 Number of Parameters (TSP)

Table 4 shows the total number of parameters of our models as a function of the population size when experimenting on TSP. Since the decoder represents less than 10% of the parameters, scaling the population size can be done efficiently. For instance, a population of 16 agents roughly doubles the model size. This observation transfers to CVRP and KP whose encoder-decoder architectures are similar to TSP. The architecture for JSSP is slightly different but the observation remains that the decoder represents a smaller part of the model (18%) and thus the population can be efficiently scaled.

Table 4: Number of parameters for different population sizes.

	Encoder	Decoder	Population size 1	4	8	16	32
Parameters	1,190,016	98,816	1,288,832	1,585,280	1,980,544	2,771,072	4,352,128
Extra parameters	-	-	0%	23%	54%	115%	238%

A.2 Training Details

In Section 3.2 (see "Training Procedure"), we described that Poppy consists of two phases. In a nutshell, the first phase consists of training our model in a single-agent setting (i.e., an encoder-decoder model with a single decoder head), whereas the second phase consists of keeping the encoder and cloning the previously trained decoder K times (where K is the number of agents) and specialize them using the population objective. Algorithm 2 shows the low-level implementation details of the training of the population (i.e., Phase 2) for environments where POMO [Kwon et al., 2020] uses several starting points; namely, given K agents and P starting points, $P \times K$ trajectories are rolled out for each instance, among which only P are effectively used for training.

B Mathematical Elements

B.1 Gradient derivation

We recall that the population objective for K is defined as:

$$J_{\text{pop}}(\theta_1, \theta_2, \dots, \theta_n) \doteq \mathbb{E}_{\rho \sim \mathcal{D}} \mathbb{E}_{\tau_1 \sim \pi_{\theta_1}, \dots, \tau_K \sim \pi_{\theta_K}} \max\left[R(\tau_1), \dots, R(\tau_K)\right].$$

Theorem (Policy gradient for the population objective)**.** *The gradient of the population objective is given by:*

$$\nabla J_{\text{pop}}(\theta_1, \theta_2, \dots, \theta_n) = \mathbb{E}_{\rho \sim \mathcal{D}} \mathbb{E}_{\tau_1 \sim \pi_{\theta_1}, \dots, \tau_K \sim \pi_{\theta_K}} \big(R(\tau_{i^*}) - R(\tau_{i^{**}})\big) \nabla \log p_{\theta_{i^*}}(\tau_{i^*}),$$

where:

$$i^* = \arg\max\big[R(\tau_1), \dots, R(\tau_K)\big], \qquad \textit{(index of the best trajectory)}$$
$$i^{**} = \arg\text{second}\max\big[R(\tau_1), \dots, R(\tau_K)\big], \qquad \textit{(index of the second best trajectory)}$$

Proof. We first derive the gradient with respect to θ_1 for convenience. As all the agents play a symmetrical role in the objective, the same procedure can be applied to any index.

$$\begin{aligned}
\nabla_{\theta_1} J_{\text{pop}}(\theta_1, \theta_2, \dots, \theta_K) &= \nabla_{\theta_1} \mathbb{E}_{\rho\sim\mathcal{D}} \mathbb{E}_{\tau_1\sim\pi_{\theta_1},\dots,\tau_K\sim\pi_{\theta_K}} \max_{i\in\{1,2,\dots,K\}} \big[R(\tau_i)\big] \\
&= \mathbb{E}_{\rho\sim\mathcal{D}} \mathbb{E}_{\tau_1,\dots,\tau_K} \max_{i\in\{1,2,\dots,K\}} \big[R(\tau_i)\big] \nabla_{\theta_1} \log p(\tau_1, \dots, \tau_K) \\
&= \mathbb{E}_{\rho\sim\mathcal{D}} \mathbb{E}_{\tau_1,\dots,\tau_K} \max_{i\in\{1,2,\dots,K\}} \big[R(\tau_i)\big] \nabla_{\theta_1} \log(\pi_{\theta_1}(\tau_1) \dots \pi_{\theta_K}(\tau_K)) \\
&= \mathbb{E}_{\rho\sim\mathcal{D}} \mathbb{E}_{\tau_1,\dots,\tau_K} \max_{i\in\{1,2,\dots,K\}} \big[R(\tau_i)\big] \nabla_{\theta_1} (\log \pi_{\theta_1}(\tau_1) + \dots + \log \pi_{\theta_K}(\tau_K)) \\
&= \mathbb{E}_{\rho\sim\mathcal{D}} \mathbb{E}_{\tau_1,\dots,\tau_K} \max_{i\in\{1,2,\dots,K\}} \big[R(\tau_i)\big] \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1),
\end{aligned}$$

We also have for any problem instance ρ and any trajectories $\tau_2, \dots, \tau_K$:

$$\begin{aligned}
\mathbb{E}_{\tau_1\sim\pi_{\theta_1}} \max_{i\in\{2,\dots,K\}} \big[R(\tau_i)\big] \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1) &= \max_{i\in\{2,\dots,K\}} \big[R(\tau_i)\big] \mathbb{E}_{\tau_1\sim\pi_{\theta_1}} \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1) \\
&= \max_{i\in\{2,\dots,K\}} \big[R(\tau_i)\big] \mathbb{E}_{\tau_1\sim\pi_{\theta_1}} \frac{\nabla_{\theta_1} \pi_{\theta_1}(\tau_1)}{\pi_{\theta_1}(\tau_1)} \\
&= \max_{i\in\{2,\dots,K\}} \big[R(\tau_i)\big] \sum_{\tau_1} \nabla_{\theta_1} \pi_{\theta_1}(\tau_1) \\
&= \max_{i\in\{2,\dots,K\}} \big[R(\tau_i)\big] \nabla_{\theta_1} \sum_{\tau_1} \pi_{\theta_1}(\tau_1) \\
&= \max_{i\in\{2,\dots,K\}} \big[R(\tau_i)\big] \nabla_{\theta_1} 1 = 0
\end{aligned}$$

Intuitively, $\max_{i\in\{2,\dots,K\}} \big[R(\tau_i)\big]$ does not depend on the first agent, so this derivation simply shows that $\max_{i\in\{2,\dots,K\}} \big[R(\tau_i)\big]$ can be used as a baseline for training θ_1.

Subtracting this to the quantity obtained in Equation B.1, we have:

$$\begin{aligned}
\nabla_{\theta_1} J_{\text{pop}}(\theta_1, \theta_2, \dots, \theta_K) &= \mathbb{E}_{\rho\sim\mathcal{D}} \mathbb{E}_{\tau_1,\dots,\tau_K} \max_{i\in\{1,2,\dots,K\}} \big[R(\tau_i)\big] \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1), \\
&= \mathbb{E}_{\rho\sim\mathcal{D}} \mathbb{E}_{\tau_2,\dots,\tau_K} \mathbb{E}_{\tau_1} \max_{i\in\{1,2,\dots,K\}} \big[R(\tau_i)\big] \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1), \\
&= \mathbb{E}_{\rho\sim\mathcal{D}} \mathbb{E}_{\tau_2,\dots,\tau_K} \mathbb{E}_{\tau_1} \left(\max_{i\in\{1,2,\dots,K\}} \big[R(\tau_i)\big] - \max_{i\in\{2,\dots,K\}} \big[R(\tau_i)\big] \right) \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1), \\
&= \mathbb{E}_{\rho\sim\mathcal{D}} \mathbb{E}_{\tau_2,\dots,\tau_K} \mathbb{E}_{\tau_1} \mathbb{1}_{i^*=1} \big(R(\tau_1) - R(\tau_{i^{**}})\big) \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1), \qquad (1) \\
&= \mathbb{E}_{\rho\sim\mathcal{D}} \mathbb{E}_{\tau_1,\dots,\tau_K} \mathbb{1}_{i^*=1} \big(R(\tau_{i^*}) - R(\tau_{i^{**}})\big) \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1).
\end{aligned}$$

Equation (1) comes from the fact that $\big(\max_{i\in\{1,2,\dots,K\}} \big[R(\tau_i)\big] - \max_{i\in\{2,\dots,K\}} \big[R(\tau_i)\big]\big)$ is 0 if the best trajectory is not τ_1, and $R(\tau_1) - \max_{i\in\{2,\dots,K\}} \big[R(\tau_i)\big] = R(\tau_1) - R(\tau_{i^{**}})$ otherwise.

Finally, for any $j \in \{1, \dots, K\}$, the same derivation gives:

$$\nabla_{\theta_j} J_{\text{pop}}(\theta_1, \theta_2, \dots, \theta_K) = \mathbb{E}_{\rho\sim\mathcal{D}} \mathbb{E}_{\tau_1,\dots,\tau_K} \mathbb{1}_{i^*=j} \big(R(\tau_{i^*}) - R(\tau_{i^{**}})\big) \nabla_{\theta_j} \log \pi_{\theta_j}(\tau_j).$$

Therefore, we have:

$$\nabla_\theta = \sum_{j=1}^{n} \mathbb{E}_{\rho\sim\mathcal{D}} \mathbb{E}_{\tau_1,\dots,\tau_K} \mathbb{1}_{i^*=j} \big(R(\tau_{i^*}) - R(\tau_{i^{**}})\big) \nabla_{\theta_j} \log \pi_{\theta_j}(\tau_j),$$

$$\nabla_\theta = \mathbb{E}_{\rho\sim\mathcal{D}} \mathbb{E}_{\tau_1,\dots,\tau_K} \big(R(\tau_{i^*}) - R(\tau_{i^{**}})\big) \nabla_{\theta_{i^*}} \log \pi_{\theta_{i^*}}(\tau_{i^*}),$$

which concludes the proof. □

C Comparison to Active Search

We implement a simple sampling strategy to give a sense of the performance of Poppy with a larger time budget. Given a population of K agents, we first greedily rollout each of them on every starting point, and evenly distribute any remaining sampling budget across the most promising K (agent, starting point) pairs for each instance with stochastic rollouts. This two-step process is motivated by the idea that is not useful to sample several times an agent on an instance where it is outperformed by another one. For environments without starting points like JSSP, we stick to the simplest approach of evenly distributing the rollouts across the population, although better performance could likely be obtained by selectively assigning more budget to the best agents.

Setup For TSP, CVRP and JSSP, we use the same test instances as in Tables 1, 2 and 3b. For TSP and CVRP, we generate a total of $200 \times 8 \times N$ candidate solutions per instance (where 8 corresponds to the augmentation strategy by Kwon et al. [2020] and N is the number of starting points), accounting for both the first and second phases. We evaluate our approach against POMO [Kwon et al., 2020] and EAS [Hottung et al., 2022] with the same budget, as well as against the supervised methods GCN-BS [Joshi et al., 2019], CVAE-Opt [Hottung et al., 2021], and DPDP [Kool et al., 2021]. As EAS has three different variants, we compare against EAS-Tab since it is the only one that does not backpropagate gradients through the network, similarly to our approach; thus, it should match Poppy's compute time on the same hardware. For JSSP, we use the same setting as EAS [Hottung et al., 2022], and sample a total of 8,000 solutions per problem instance for each approach. For a proper comparison, we reimplemented EAS with the same model architecture as Poppy.

Results Tables 5, 6 and 7 show the results for TSP, CVRP and JSSP respectively. With extra sampling, Poppy reaches a performance gap of 0.002% on TSP100, and establishes a state-of-the-art for general ML-based approaches, even when compared to supervised methods. For CVRP, adding sampling to Poppy makes it on par with DPDP and EAS, depending on the problem size, and it is only outperformed by the active search approach EAS, which gives large improvements on CVRP. As the two-step sampling process used for Poppy is very rudimentary compared to the active search method described in Hottung et al. [2022], it is likely that combining the two approaches could further boost performance, which we leave for future work.

Table 5: TSP results (active search)

		Inference (10k instances)			**0-shot** (1k instances)					
		$n = 100$			$n = 125$			$n = 150$		
	Method	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time
	Concorde	7.765	0.000%	82M	8.583	0.000%	12M	9.346	0.000%	17M
	LKH3	7.765	0.000%	8H	8.583	0.000%	73M	9.346	0.000%	99M
SL	GCN-BS	7.87	1.39%	40M*	-	-	-	-	-	-
SL	CVAE-Opt	-	0.343%	6D*	8.646	0.736%	21H*	9.482	1.45%	30H*
SL	DPDP	7.765	0.004%	2H*	8.589	0.070%	31M*	9.434	0.94%	44M*
RL	POMO (200 samples)	7.769	0.056%	2H	8.594	0.13%	20M	9.376	0.31%	32M
RL	EAS	7.768	0.048%	5H*	8.591	0.091%	49M*	9.365	0.20%	1H*
RL	**Poppy 16 (200 samples)**	**7.765**	**0.002%**	2H	**8.584**	**0.009%**	20M	**9.351**	**0.05%**	32M

Table 6: CVRP results (active search)

	Method	**Inference** (10k instances) $n = 100$ Obj.	Gap	Time	**0-shot** (1k instances) $n = 125$ Obj.	Gap	Time	**0-shot** (1k instances) $n = 150$ Obj.	Gap	Time
	LKH3	15.65	0.000%	6D	17.50	0.000%	19H	19.22	0.000%	20H
SL	CVAE-Opt	-	1.36%	11D*	17.87	2.08%	36H*	19.84	3.24%	46H*
SL	DPDP	15.63	−0.13%	23H*	17.51	0.07%	3H*	19.31	0.48%	5H*
RL	POMO (200 samples)	15.67	0.18%	4H	17.56	0.33%	43M	19.43	1.08%	1H
RL	EAS	**15.62**	**-0.14%**	8H*	17.49	0.00%	80M*	19.36	0.72%	2H*
RL	**Poppy 32 (200 samples)**	**15.62**	**-0.14%**	4H	**17.49**	**-0.10%**	42M	**19.32**	**0.50%**	1H

Table 7: JSSP

Method	**Inference** (100 instances) 10×10 Obj.	Gap	Time
OR-Tools (optimal)	807.6	0.0%	37S
L2D (Greedy)	988.6	22.3%	20S*
L2D (Sampling)	871.7	8.0%	8H*
EAS-L2D	860.2	6.5%	8H*
Sampling	862.1	6.7%	3H
EAS	858.4	6.3%	3H
Poppy 16	**849.7**	**5.2%**	3H

D Problems

We here describe the details of the four CO problems we have used to evaluate Poppy, namely TSP, CVRP, KP and JSSP. We use the corresponding implementations from Jumanji [Bonnet et al., 2023]: `TSP`, `CVRP`, `Knapsack` and `JobShop`. For each problem, we describe below the training (e.g. architecture, hyperparameters) and the process of instance generation. In addition, we show some example solutions obtained by a population of agents on TSP and CVRP. Finally, we thoroughly analyze the performance of the populations in TSP.

D.1 Traveling Salesman Problem (TSP)

Instance Generation The n cities that constitute each problem instance have their coordinates uniformly sampled from $[0, 1]^2$.

Architecture We use the same model as Kool et al. [2019] and Kwon et al. [2020] except for the batch-normalization layers, which are replaced by layer-normalization to ease parallel batch processing. We invert the mask used in the decoder computations (i.e., masking the available cities instead of the unavailable ones) after experimentally observing faster convergence rates. The results reported for POMO were obtained with the same implementation changes to keep the comparison fair. These results are on par with those reported in POMO [Kwon et al., 2020].

Hyperparameters To match the setting used by Kwon et al. [2020], we use the Adam optimizer [Kingma and Ba, 2015] with a learning rate $\mu = 10^{-4}$, and an L_2 penalization coefficient of 10^{-6}. The encoder is composed of 6 multi-head self-attention layers with 8 heads each. The dimension of the keys, queries and values is 16. Each attention layer is composed of a feed-forward layer of size 512, and the final node embeddings have a dimension of 128. The decoders are composed of 1 multi-head attention layer with 8 heads and 16-dimensional key, query and value.

The number of starting points P is 50 for each instance. We determined this value after performing a grid-search based on the first training steps with $P \in \{20, 50, 100\}$.

Example Solutions Figure 5 shows some trajectories obtained from a 16-agent population on TSP100. Even though they look similar, small decisions differ between agents, thus frequently leading to different solutions. Interestingly, some agents (especially 6 and 11) give very poor

trajectories. We hypothesize that it is a consequence of specializing since agents have no incentive to provide a good solution if another agent is already better on this instance.

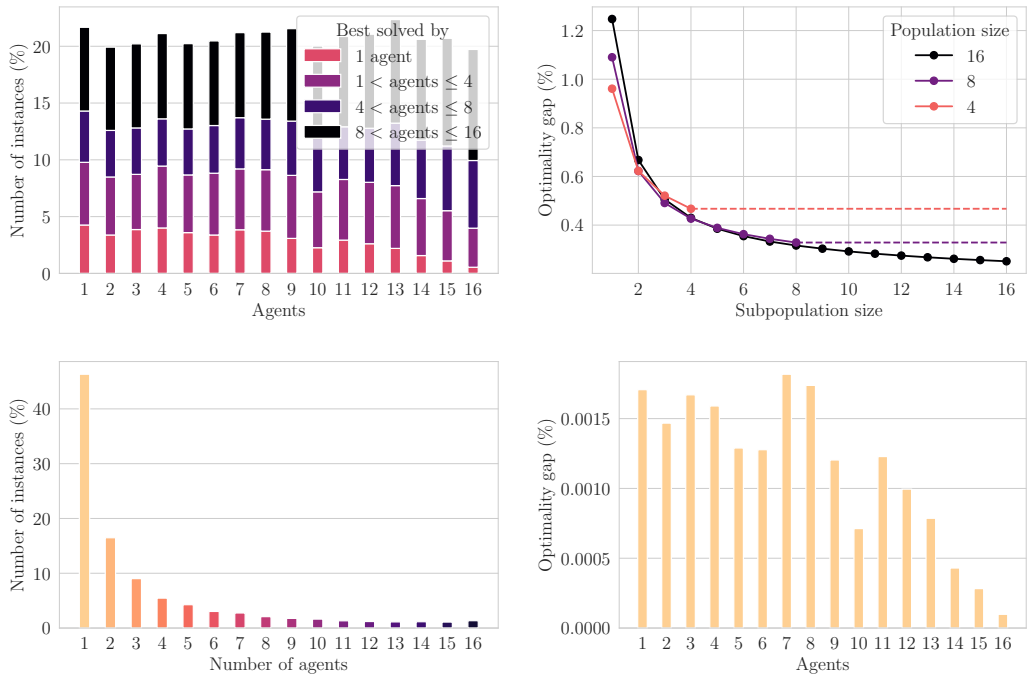


Figure 4: **Upper left**: Proportion of instances that each agent solves best among the population for Poppy 16 on TSP100. Colors indicate the number of agents in the population giving the same solution for these sets of instances. **Upper right**: The mean performance of 1,000 randomly drawn sub-populations for Poppy 1, 4, 8 and 16. **Bottom left**: Proportion of test instances where any number of Poppy 16 agents reaches the exact same best solution. The best performance is reached by only a single agent in 47% of the cases. **Bottom Right**: Optimality gap loss suffered when removing any agent from the population using Poppy 16. Although some agents contribute more (e.g. 7, 8) and some less (e.g. 15, 16), the distribution is relatively even, even though no explicit mechanism enforces this behavior

Population Analysis Figure 4 shows some additional information about individual agent performances. In the left figure, we observe that each agent gives on average the best solution for 20% of the instances, and that for around 4% it gives the unique best solution across the population. These numbers are generally evenly distributed, which shows that every agent contributes to the whole population performance. Furthermore, we observe the performance is quite evenly distributed across the population of Poppy 16; hence, showing that the population has not collapsed to a few high-performing agents, and that Poppy benefits from the population size, as shown in the bottom figure. On the right is displayed the performance of several sub-populations of agents for Poppy 4, 8 and 16. Unsurprisingly, fixed size sub-populations are generally better when sampled from smaller populations: Poppy 16 needs 4 agents to recover the performance of Poppy 4, and 8 agents to recover the performance of Poppy 8 for example. This highlights the fact that agents have learned complementary behaviors which might be sub-optimal if part of the total population is missing.

D.2 Capacitated Vehicle Routing Problem (CVRP)

Instance Generation The coordinates of the n customer nodes and the depot are uniformly sampled in $[0, 1]^2$. The demands are uniformly sampled from the discrete set $\{\frac{1}{D}, \frac{2}{D}, \dots, \frac{9}{D}\}$ where $D = 50$ for CVRP100, $D = 55$ for CVRP125, and $D = 60$ for CVRP150. The maximum vehicle capacity is 1. The deliveries cannot be split: each customer node is visited once, and its whole demand is taken off the vehicle's remaining capacity.

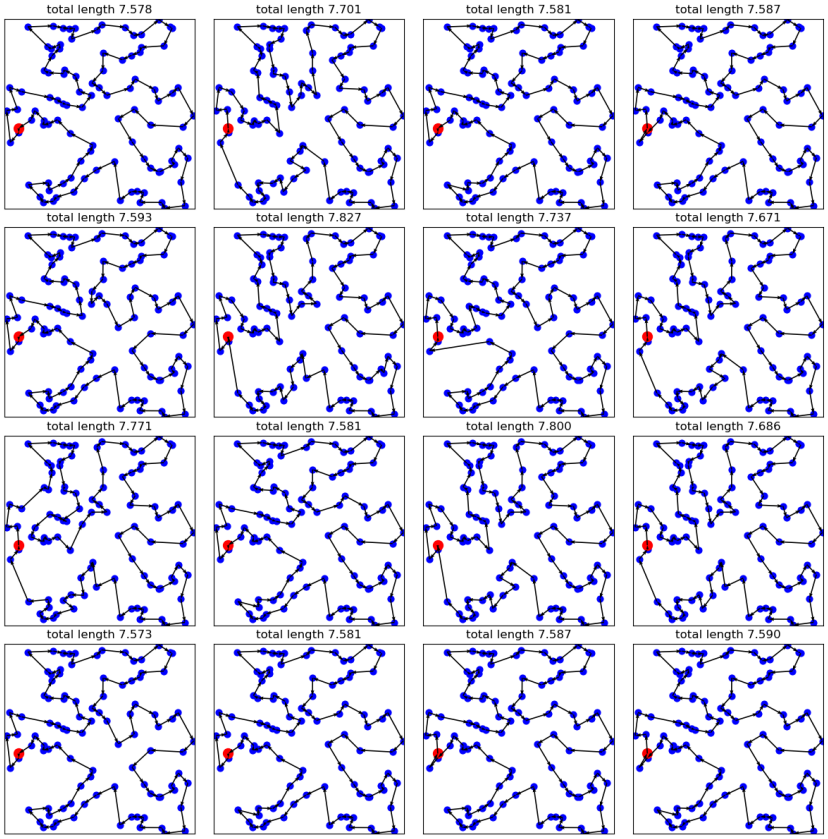


Figure 5: Example TSP trajectories given by Poppy for a 16-agent population from one starting point (red).

Architecture We use the same model as in TSP. However, unlike TSP, the mask is not inverted; besides, it does not only prevent the agent from revisiting previous customer nodes, but also from visiting the depot if it is the current location, and any customer node whose demand is higher than the current capacity.

Hyperparameters We use the same hyperparameters as in TSP except for the number of starting points P per instance used during training, which we set to 100 after performing a grid-search with $P \in \{20, 50, 100\}$.

Example Solutions Figure 6 shows some trajectories obtained by 16 agents from a 32-agent population on CVRP100. Unlike TSP, the agent/vehicle performs several tours starting and finishing in the depot.

D.3 0-1 Knapsack (KP)

Problem Description Given a set of items, each with a weight and a value, the goal is to determine which items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

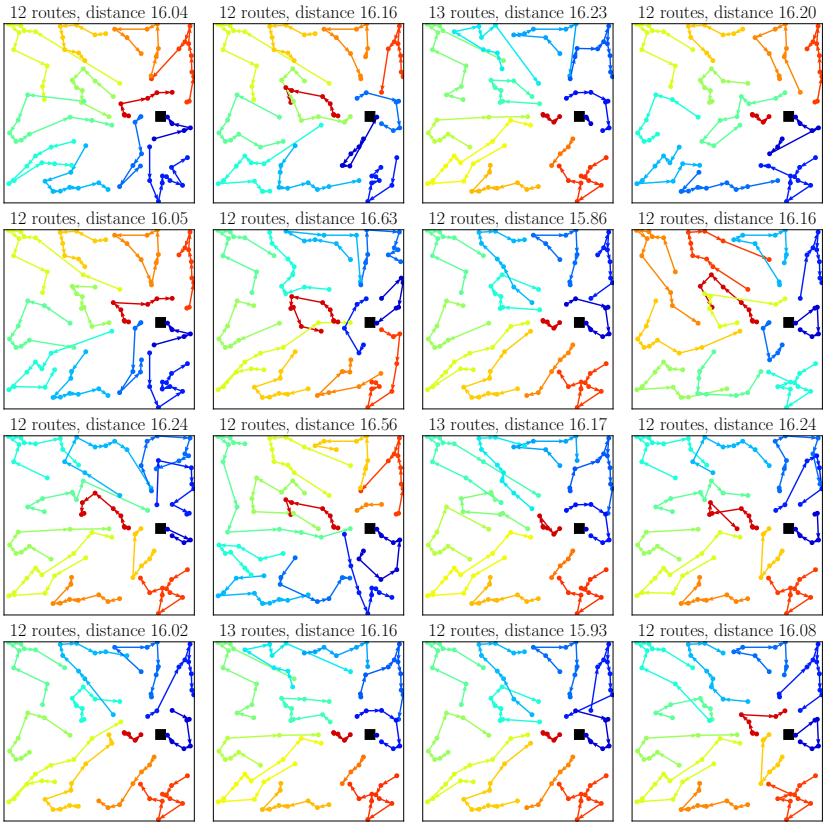


Figure 6: Example CVRP trajectories given by Poppy for 16 agents from a 32-agent population. The depot is displayed as a black square. The edges from/to the depot are omitted for clarity.

Instance Generation Item values and weights are both uniformly sampled in $[0, 1]$. The bag capacity is fixed at 25.

Training For KP, and contrary to the other three environments, training an agent is lightning-fast as it only takes a few minutes. In this specific case, we noticed it was not necessary to train a single decoder first. Instead, (i) we directly train a population in parallel from scratch, and (ii) specialize the population exactly as done in the other environments.

Architecture We use the same model as in TSP. However, the mask used when decoding is not inverted, and the items that do not fit in the bag are masked together with the items taken so far.

Hyperparameters We use the same hyperparameters as in TSP except for the number of starting points P used during training, which we set to 100 after performing a grid-search with $P \in \{20, 50, 100\}$.

D.4 Job-Shop Scheduling Problem (JSSP)

Problem Description We consider the problem formulation described by Zhang et al. [2020] and also used in Bonnet et al. [2023], in the setting of an equal number of machines, jobs and operations per job. A job-shop scheduling problem consists in N jobs that all have N operations that have to be scheduled on N machines. Each operation has to run on a specific machine for a given time. The solution to a problem is a schedule that respects a few constraints:

- for each job, its operations have to be processed/scheduled in order and without overlap between two operations of the same job,
- a machine can only work on one operation at a time,
- once started, an operation must run until completion.

The goal of the problem is to determine a schedule that minimizes the time needed to process all the jobs. The length of the schedule is also known as its makespan.

Instance Generation We use the same generation process as Zhang et al. [2020]. For each of the N jobs, we sample N operation durations uniformly in $[1, 99)$. Each operation is given a random machine to run on by sampling a random permutation of the machine IDs.

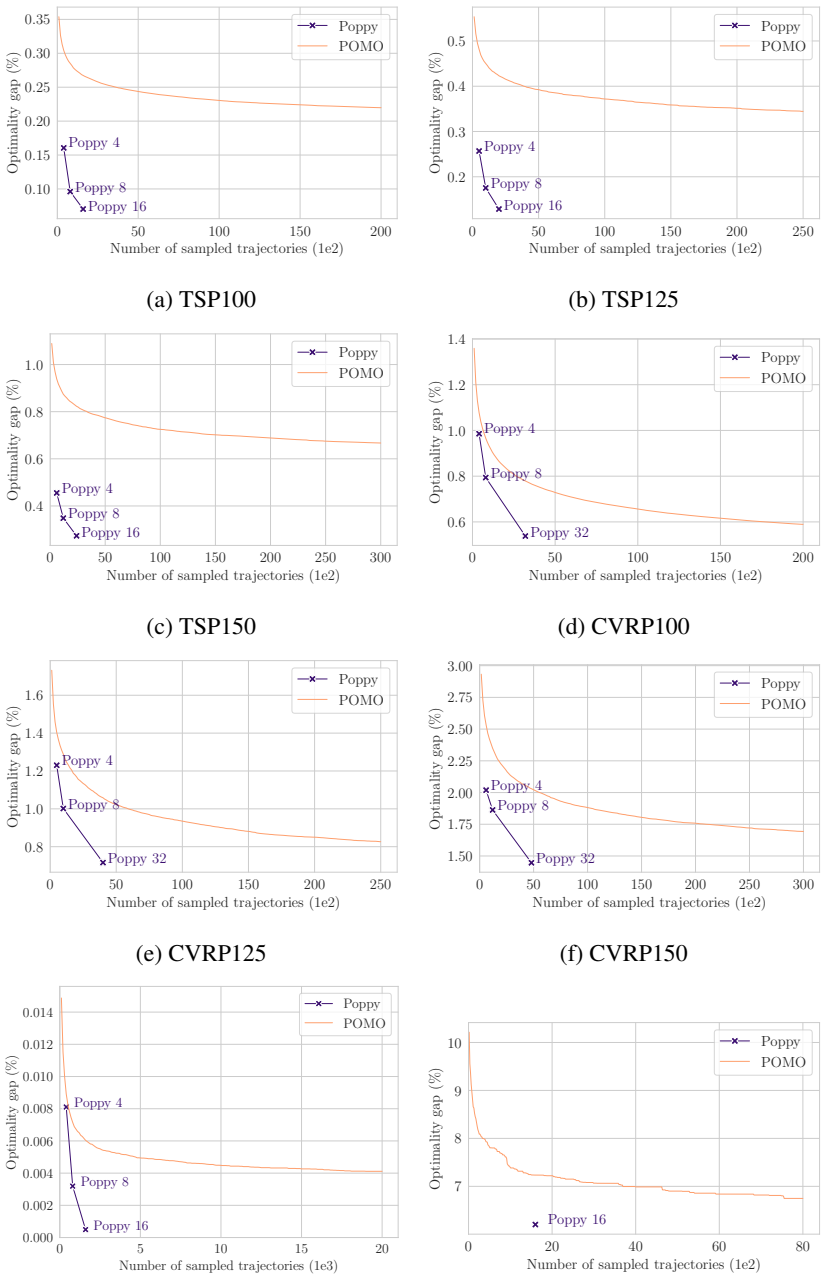
Transition Function To leverage JAX, we use the environment dynamics implemented in Jumanji [Bonnet et al., 2023] which differs from framing proposed by L2D [Zhang et al., 2020]. However, the two formulations are equivalent, therefore our results on the former would transfer to the latter. Our implementation choice was purely motivated by environment speed.

Architecture We use the actor-critic transformer architecture implemented in Jumanji which is composed of an attention layer on the machines' status, an attention layer on the operation durations (with positional encoding) and then two attention layers on the joint sequence of jobs and machines. The network outputs N marginal categorical distributions for all machines, as well as a value for the critic. The actor and critic networks do not share any weights.

Hyperparameters Like in Zhang et al. [2020], we evaluate our algorithm with $N = 10$ jobs, operations and machines, i.e. 10×10 instances. We use REINFORCE with the critic as a baseline (state-value function). Since episodes may take a long time (for an arbitrary policy, the lowest upper bound on the horizon is $98 \times N^3$), we use an episode horizon of 1250 and give a reward penalty of two times the episode horizon when producing a makespan of more than this limit.

E Time-performance Tradeoff

We present in figure 7 a comparison of the time-performance Pareto front between Poppy and POMO as we vary respectively the population size and the amount of stochastic sampling. Poppy consistently provides better performance for a fixed number of trajectories. Strikingly, in almost every setting, matching Poppy's performance by increasing the number of stochastic samples does not appear tractable.



(a) TSP100 (b) TSP125

(c) TSP150 (d) CVRP100

(e) CVRP125 (f) CVRP150

(g) KP100 (h) JSSP100

Figure 7: Comparison of the time-performance Pareto front of Poppy and POMO, for each problem used in the paper. The x-axis is the number of trajectories sampled per test instance, while the y-axis is the gap with the optimal solution for TSP, KP and JSSP, and the gap with LKH3 for CVRP.