A Extended Results

In this section, we compare the results of our method to an extensive list of baselines for the TSP, CVRP, and JSSP problems. Subsection A.1 focuses on the results of the standard benchmark, corresponding to instances from the training distribution as well as larger instances. Subsection A.2 presents the results for procedurally transformed (and hence out-of-distribution) instances.

A.1 Extended results on standard benchmark

In Section 4 of the main paper, we report the performances of the main competitors on a standard benchmark for TSP, CVRP, and JSSP. We report extended results, with other competitors (2-Opt-DL (Wu et al., 2022) and LIH (de O. da Costa et al., 2020)) and inference time. We also report the results of an alternative architecture (L2D) for JSSP. Tables 3a, 3b, and 3c show the results of all methods for TSP, CVRP, and JSSP problems, respectively. The first column in the tables shows the average performance across the validation set, which is the mean tour length for TSP and CVRP, and the mean schedule duration for JSSP, and the second and third columns show the optimality gap and total run-time, respectively.

Examining the inference time provides valuable insights for comparison, especially considering that time constraints are often a crucial factor in industrial applications. On the whole benchmark, our method is as fast as the baselines (POMO, Poppy) and almost three times faster than EAS.

Results from Table 6c confirm our choice to use the attention-based model from Jumanji (Bonnet et al., 2023): the attention-based model is performing better on all sets, while being 5 times faster. We can also compare EAS used with both architectures (L2D or the attention-based model). EAS is performing better with the attention-based architecture in 2 datasets out of 3. Interestingly, EAS (w/ L2D) is performing very well on the training distribution, but generalizes less than EAS (w/ attention-based model).

A.2 Results on the procedurally transformed instances

In Section 4.2, we present the performance of the methods under study on procedurally transformed instances. In particular, Figure 3 reports the relative performance of the baselines compared to our method COMPASS. In this section, we report the numerical results corresponding to this plot. Tables 4a show the results of all methods on TSP and 4b on CVRP. Each line corresponds to a mutation power used to procedurally transform instances. Mutations are used to create 1800 new instances. The columns report the average tour length and the optimality gap. Details about the way those datasets are created can be found in Appendix C.

Our method outperforms the baselines on the entire benchmark. Nevertheless, the industrial solver LKH3 is still performing better than COMPASS on all mutated datasets, showing room for improvement of the RL-based approaches.

B Analysis of the Performance during the Search Process

In this section, we examine how the performance of COMPASS, along with the baseline methods, evolves during the search process. The evaluation procedure consists of 160,000 rollouts per problem for TSP and CVRP, and 8,000 rollouts per problem for JSSP, distributed over the population for all problems. Figures 6, 7, 8 show the performances – for TSP, CVRP, and JSSP, respectively, – for our method COMPASS and the three main baselines, POMO (single-agent for JSSP), Poppy, and EAS. Each figure showcases the overall performance and the latest performance achieved by the methods for various instance sizes, including the training distribution size (left column), medium size (middle column), and large size (right column). The first row of plots illustrates the performance of the best solution discovered thus far in the search process, while the second row of plots presents the best performance of the last batch of solutions found at the current timestep of the search process.

Note that the 'Latest' performance metric reported in these plots is different from the one reported in Section 4.3 of the paper, as the latter reports the mean and standard deviation.

We can draw three main conclusions from those plots. (i) On all instance size of the TSP, COMPASS clearly outperforms baselines, with a search that constantly find better solutions on average. (ii) We

Table 3: Results of COMPASS against the baseline algorithms for (a) TSP, (b) CVRP, and (c) JSSP problems. The methods are evaluated on instances from training distribution as well as on larger instance sizes to test generalization. Tables report the best solutions found, gaps to best industrial solvers, and inference times.

(a) TSP

	Training distr.			**Generalization**								
	$n = 100$			$n = 125$			$n = 150$			$n = 200$		
Method	Obj.	Gap	Time	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	10.687	0.000%	31M
LKH3	7.765	0.000%	8H	8.583	0.000%	73M	9.346	0.000%	99M	10.687	0.000%	3H
2-Opt-DL	7.83	0.87%	-	-	-	-	-	-	-	-	-	-
LIH	7.87	1.42%	-	-	-	-	-	-	-	-	-	-
POMO (greedy)	7.796	0.404%	41M	8.635	0.607%	6S	9.440	1.001%	10S	10.933	2.300%	21S
POMO	7.779	0.185%	2H	8.609	0.299%	20M	9.401	0.585%	32M	10.956	2.513%	70M
Poppy 16	7.766	0.013%	2H	8.587	0.050%	20M	9.359	0.141%	32M	10.795	1.007%	70M
EAS	7.779	0.176%	5H	8.601	0.252%	57M	9.382	0.381%	2H	10.758	0.660%	4H
COMPASS (ours)	**7.765**	**0.002%**	2H	**8.586**	**0.036%**	20M	**9.354**	**0.083%**	32M	**10.724**	**0.348%**	70M

(b) CVRP

	Training distr.			**Generalization**								
	$n = 100$			$n = 125$			$n = 150$			$n = 200$		
Method	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time
LKH3	15.65	0.000%	-	17.50	0.000%	-	19.22	0.000%	-	22.00	0.000%	-
LIH	16.03	2.47%	5H	-	-	-	-	-	-	-	-	-
POMO (greedy)	15.874	1.430%	2M	17.818	1.818%	<1M	19.750	2.757%	1M	23.318	5.992%	2M
POMO	15.713	0.399%	4H	17.612	0.642%	43M	19.488	1.393%	1H	23.378	6.264%	100M
Poppy 32	15.663	0.084%	4H	17.548	0.276%	42M	19.421	1.044%	1H	23.352	6.144%	100M
EAS	15.661	0.068%	9H	17.517	0.094%	93M	**19.285**	**0.341%**	3H	**22.264**	**1.120%**	6H
COMPASS (ours)	**15.594**	**-0.361%**	4H	**17.511**	**0.064%**	42M	19.313	0.485%	1H	22.462	2.098%	100M

(c) JSSP

	Training distr.			**Generalization**					
	10×10			15×15			20×15		
Method	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time
OR-Tools	807.6	0.0%	37S	1188.0	0.0%	3H	1345.5	0.0%	80H
L2D (Greedy)	988.6	22.3%	20S	1528.3	28.6%	44S	1738.0	29.2%	60S
L2D (Sampling)	871.7	8.0%	8H	1378.3	16.0%	25H	1624.6	20.8%	40H
EAS (w/ L2D)	**837.0**	**3.7%**	7H	1326.4	11.7%	22H	1570.8	16.8%	37H
Single	862.1	6.7%	3H	1302.6	9.6%	5H	1503.0	11.7%	8H
Poppy 16	849.7	5.2%	3H	1290.4	8.6%	5H	1495.7	11.2%	8H
EAS	858.4	6.3%	5H	1295.2	9.0%	9H	1498.0	11.3%	11H
COMPASS (ours)	845.5	4.7%	3H	**1282.8**	**8.0%**	5H	**1485.6**	**10.4%**	8H

can clearly see the difference between principled search (COMPASS, EAS) and stochastic sampling (POMO, Poppy 16), as the former have an improving 'latest batch performance', whereas the latter do not (iii) Interestingly, on several tasks, EAS has a higher maximum value on its latest batch (averaged on the 1000 problem instances); but the wider search of COMPASS enables to find better solutions for each problem instance in average. This can be observed on all JSSP sets and on the two first CVRP sets. For larger instances of CVRP, we can see that EAS is able to outperform our method: EAS is updating more parameters than COMPASS and this difference of modified parameters increases as the instance size increases (because its the product of the embedding size and the number of nodes in the instance). This larger number of updated parameters enables a better adaptation but also comes with a computational cost.

C Mutation Operators Used in the Generalization Tasks

In this section, we outline the mutation operators employed to generate the out-of-distribution (OOD) instances. The training distribution comprises random uniform Euclidean (RUE) instances, which are created by uniformly sampling the city coordinates from a unit square. To diversify the dataset and assess the methods' generalization abilities, these RUE instances can be mutated to exhibit different underlying distributions. We utilize nine mutation operators (taken from Bossek et al. (2019)) to construct the OOD dataset for TSP and CVRP by mutating the RUE instances. These mutated instances not only closely resemble real-world geographical data but also serve as valuable

Table 4: Results of the methods on procedurally transformed instances, obtained by applying mutations with increasing mutation powers (referred to as 'Mu'). Our method COMPASS outperforms other baselines on all mutated instances.

(a) TSP

	LKH3		POMO		Poppy 16		EAS		**COMPASS (ours)**	
Mu	Obj.	Gap	Obj.	Gap	Obj.	Gap	Obj.	Gap	Obj.	Gap
0	7.768	0.000%	7.782	0.188%	7.769	0.02%	7.781	0.177%	**7.769**	**0.013%**
1	7.609	0.000%	7.624	0.195%	7.611	0.022%	7.623	0.175%	**7.611**	**0.019%**
2	7.562	0.000%	7.577	0.199%	7.563	0.022%	7.575	0.179%	**7.563**	**0.014%**
3	7.485	0.000%	7.502	0.217%	7.488	0.029%	7.499	0.183%	**7.487**	**0.019%**
4	7.386	0.000%	7.402	0.219%	7.389	0.034%	7.399	0.168%	**7.388**	**0.021%**
5	7.308	0.000%	7.326	0.251%	7.311	0.042%	7.322	0.197%	**7.310**	**0.026%**
6	7.182	0.000%	7.201	0.272%	7.186	0.06%	7.196	0.198%	**7.184**	**0.036%**
7	7.063	0.000%	7.085	0.311%	7.069	0.078%	7.079	0.224%	**7.066**	**0.044%**
8	6.910	0.000%	6.938	0.402%	6.918	0.118%	6.927	0.249%	**6.914**	**0.067%**
9	6.732	0.000%	6.770	0.557%	6.744	0.176%	6.751	0.277%	**6.738**	**0.091%**

(b) CVRP

	LKH3		POMO		Poppy 16		EAS		**COMPASS (ours)**	
Mu	Obj.	Gap	Obj.	Gap	Obj.	Gap	Obj.	Gap	Obj.	Gap
0	15.595	0.000%	15.661	0.426%	15.613	0.119%	15.611	0.103%	**15.594**	**-0.009%**
1	15.337	0.000%	15.41	0.471%	15.36	0.148%	15.356	0.121%	**15.339**	**0.012%**
2	15.271	0.000%	15.345	0.482%	15.294	0.15%	15.29	0.121%	**15.274**	**0.018%**
3	15.156	0.000%	15.23	0.488%	15.181	0.164%	15.174	0.123%	**15.16**	**0.027%**
4	15.032	0.000%	15.112	0.531%	15.061	0.188%	15.058	0.169%	**15.039**	**0.043%**
5	14.805	0.000%	14.885	0.545%	14.835	0.206%	14.828	0.154%	**14.813**	**0.055%**
6	14.561	0.000%	14.646	0.581%	14.594	0.229%	14.587	0.182%	**14.568**	**0.048%**
7	14.306	0.000%	14.398	0.641%	14.348	0.292%	14.339	0.227%	**14.322**	**0.11%**
8	13.886	0.000%	13.989	0.741%	13.936	0.363%	13.922	0.263%	**13.907**	**0.15%**
9	13.507	0.000%	13.612	0.773%	13.566	0.431%	13.548	0.298%	**13.534**	**0.196%**

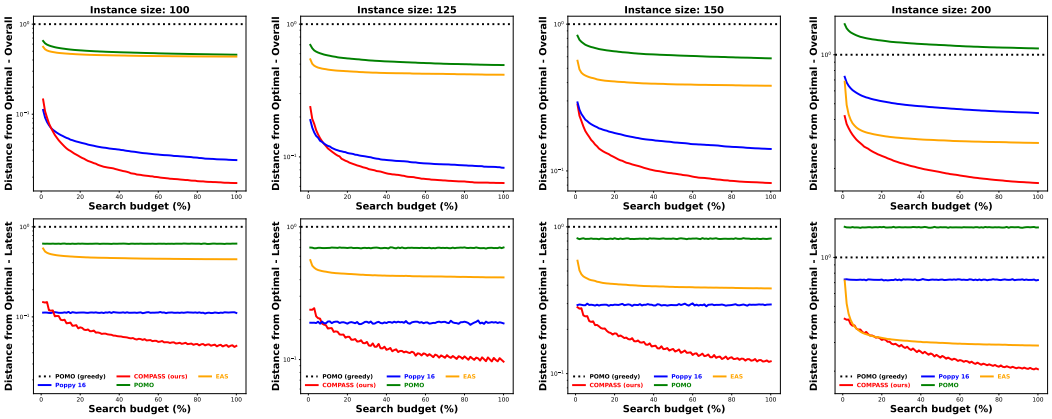

Figure 6: Evolution of the performance along the budget on **TSP**. This plots shows - for each method and each instance size - the evolution of (i) the performance of the method overall, hence the best solution ever found, on the top panel, and (ii) the performance of the latest sampled batch for each method, hence the best solution found in the recent attempts.

benchmarks for evaluating the methods' generalization capabilities. Figure 9 presents a visual representation of a RUE instance as well as the instance mutated by mutation operators and the list below defines each operator.

- Explosion: this operator simulates a random explosion that creates a gap or hole in the point cloud. It randomly selects a center of explosion, and then displaces all cities within a specified radius of this center to locations outside the radius.

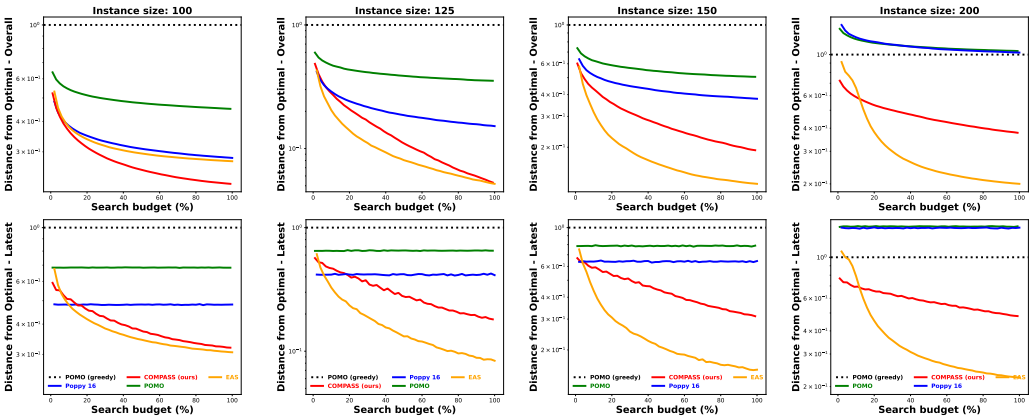

Figure 7: Evolution of the performance along the budget on **CVRP**. This plots shows - for each method and each instance size - the evolution of (i) the performance of the method overall, hence the best solution ever found, on the top panel, and (ii) the performance of the latest sampled batch for each method, hence the best solution found in the recent attempts.

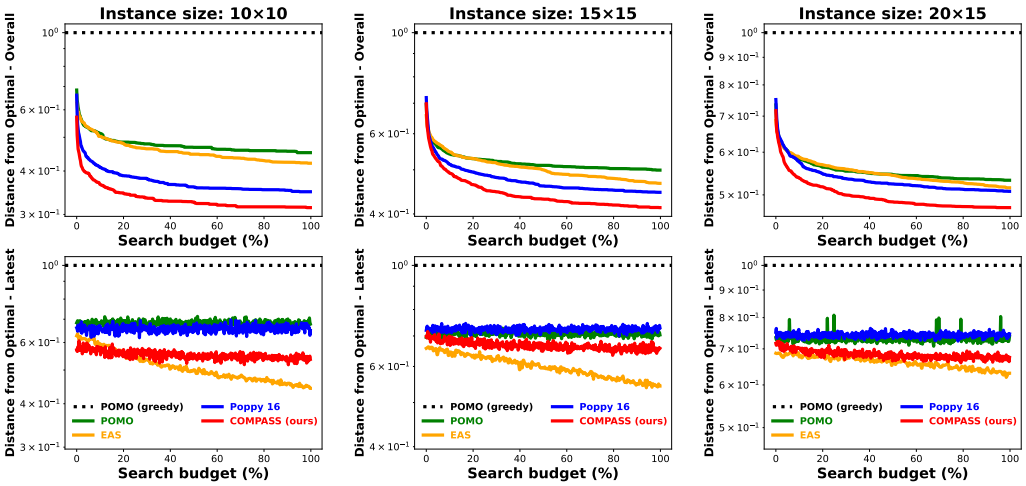

Figure 8: Evolution of the performance along the budget on **JSSP**. This plots shows - for each method and each instance size - the evolution of (i) the performance of the method overall, hence the best solution ever found, on the top panel, and (ii) the performance of the latest sampled batch for each method, hence the best solution found in the recent attempts.

- Implosion: this operator serves as the inverse of the explosion operator by bringing cities closer together towards a central point. It involves randomly selecting an implosion center and radius, and subsequently shifting all cities located within the implosion radius towards the center.
- Cluster: this operator generates a concentrated cluster of cities by randomly selecting a cluster center and mutating cities within a specified radius around the center. Specifically, cities are randomly chosen and their locations are modified to be within the selected radius of the cluster center.
- Rotation: this operator applies a rotation transformation to the cities around a specified pivot point. This mutation operator introduces angular displacement and rearranges the spatial arrangement of the cities in the TSP instance.
- Linear Projection: this operator performs a linear projection of the cities onto a randomly generated line. A random subset of cities is selected and repositioned along the line according to their original distances.

- Expansion: this operator merges the concepts of the explosion and linear projection mutations by displacing cities farther away from a randomly generated line in an orthogonal direction.
- Compression: this operator, conversely to the expansion operator, is a combination of the implosion and linear projection operators by displacing the cities closer to the randomly generated line in an orthogonal direction.
- Axis Projection: this operator is a special case of the linear project operator as the randomly generated line can either be parallel to the x- or y-axis.
- Grid: this operator maps randomly chosen cities onto a grid-like structure. Specifically, the width, height, and proximity of the cities within the grid are randomly selected. Next, several cities are chosen from the instance and displaced into one of the grid locations.

It is worth noting that all operators are parameterized by a probability argument which denotes the likelihood of mutating each city within the instance, referred to as mutation power. As a result, the mutation power is directly proportional to the number of cities mutated, and hence positively correlated with the *shift* between the underlying distribution and the distribution of the RUE instance. Consequently, we use this mutation power as a reference to define the scale when studying the robustness of the baselines to out-of-distribution instances.

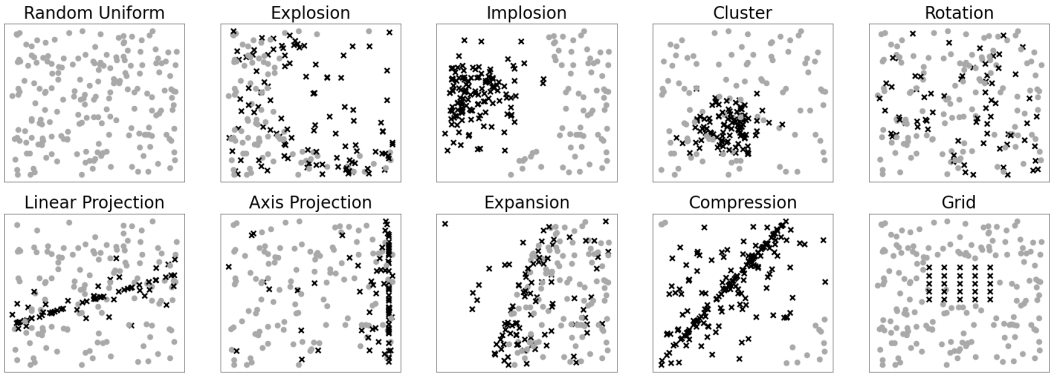

Figure 9: Visual representation of the RUE instance and mutated version of the instance through the 9 mutation operators. The gray dots represent the original (unmutated) cities, whereas, the black crosses represent the cities that have been mutated by the operator.

D Architecture details

D.1 TSP and CVRP Networks

The architecture of our model for the TSP and CVRP problems consists of several key components. First, we have a single encoder responsible for encoding problem instances into a matrix of embeddings. This encoder follows a similar approach to other reinforcement learning methods for combinatorial optimization, such as POMO (Kwon et al., 2020) and Poppy (Grinsztajn et al., 2022).

Next, we have a single conditioned decoder that takes in the embeddings and the current state of the environment and outputs the next action. In contrast to the encoder being called at the beginning of the episode, the decoder is called at each step of the episode conditioned to the same latent vector throughout a given episode. The conditioned decoder architecture is quite similar to the one utilized in POMO and Poppy. It incorporates a multi-head attention mechanism to compute cross-attention between the embeddings and a local context. This local context includes the embedding of the starting point, the embedding of the current node, and the mean embedding of all nodes. The notable difference between our method COMPASS and prior works (POMO and Poppy) is that our decoder is conditioned on a vector sampled from a 16-dimension latent space. This latent vector is concatenated with the key, query, and value inputs of the multi-head attention decoder module. This conditioning allows us to create distinct policies while processing the same observation from the environment.

Each latent vector corresponds to a unique policy, and thus, sampling the latent space to obtain vectors that our model can condition upon gives us an infinite set of policies.

D.2 JSSP Network

The model architecture used for the JobShop Scheduling Problem (JSSP) is different from the networks used for TSP and CVRP (where the latter were taken from POMO (Kwon et al., 2020)). Prior work, which achieved state-of-the-art for JSSP (Hottung et al., 2022), use the L2D model (Zhang et al., 2020) which is a Graph Neural Network, on a different, yet equivalent environment. We implemented an attention-based model and observed our model to outperform L2D and thus, decided to use our transformer-based architecture (similar to the TSP and CVRP models) to tackle JSSP.

We utilize the actor-critic transformer architecture, as implemented in Jumanji (Bonnet et al., 2023). This architecture consists of an encoder and decoder network for both the actor and critic components. The encoder network incorporates attention layers for the machines' status, operation durations (with positional encoding), and joint sequence of jobs and machines. During each step of the episode, the encoder network is called and produces joint embeddings of the jobs and machines. These embeddings, along with the latent vector, are then fed into the decoder network. The decoder network receives the encoder embeddings and the latent vector as inputs and concatenates each dimension of the embedding with the latent vector before passing it through a multi-layer perceptron. At each step, the decoder network generates N marginal categorical distributions for each machine and provides a value generated by the critic. It is important to note that the actor and critic networks are separate entities with distinct sets of weights, ensuring that they do not share any parameters.

E Latent Policy Space

E.1 Design

The latent policy space defines a set of vectors that the single model can condition itself upon, and thus, this latent space provides us with an infinitely large set of policies (which becomes specialized and diverse through our training procedure). There are several ways to define the latent space, and the simplest approach is to use a set of N one-hot encoded vectors. This way is very similar to the definition of the skill space in the RL Skill-Discovery literature (Eysenbach et al., 2019; Sharma et al., 2019) and the difference between a set of independent policies and a single conditioned policy is reminiscent of the opposition between RL-based methods and Quality-Diversity methods for Skill Discovery (Chalumeau et al., 2023a). Through preliminary experiments, we achieved a similar performance to Poppy 16 (Grinsztajn et al., 2022) (current SOTA) with a set of 16 one-hot encoded vectors. However, the drawback of using a discrete latent space is that we cannot have an infinite set of policies and cannot interpolate within the latent space to adapt our model. Therefore, we define our latent space as a continuous n-dimension square. This enables us with an infinite number of policies that can be uniformly sampled during training and strategically searched during inference. During the experimentation phase, we investigated several different distributions and space sizes and concluded with a 16-dimensional space constrained to $[-1, 1]^{16}$. In practice, we multiply the latent vector by a factor 100, which is equivalent to sampling in $[-100, 100]^{16}$.

E.2 Visualisation

In section 4.3, we train COMPASS with a two-dimensional latent space and report the visualization of its performance landscape on a randomly selected instance of TSP150. To obtain this visualization, COMPASS was trained on a distribution of TSP100, and then we evaluated 32 000 latent vectors on a randomly sampled TSP150 instance. This enables us to create a precise heat-map (contour plot) of the performance landscape of the latent space on this instance. In Figure 10, we report 7 additional visualizations for 7 new instances (the first one is the one reported in the main paper).

We can draw three main observations out of this plot: (i) the landscape is instance-dependent: we can see that the whole landscape changes and, in particular, the performant regions are different for each of the 8 problems studied. This shows that the whole latent space is used at inference time, and there does not seem to be a subpart dominating the others. (ii) They are usually several performant areas in the latent space for a given instance. This motivates the use of several independent search components at inference time. It enables us to avoid having all the search budget used on a suboptimal

area. In case there is a clear area of interest, the components are likely to all converge there, hence having no loss of performance. This observation is insightful as it illustrates that several distinct solving-strategies can lead to solutions of similar quality (even if those solutions are distinct). (iii) Interestingly, they are clear discontinuities in the performance landscape. We can observe those in all problems visualized, but *Problem 5* is a particularly relevant example. We can see several clear frontiers in the landscape, which shows that they can be a clear discontinuity in the mapping between solving-strategy and solution quality. We can make the hypothesis that there is one important decision that differs at the frontier, leading to a completely different performance.

Note that the deeper analysis of these latent spaces can provide very interesting insights about the solving process (e.g. the important decisions taken during the solving process), which could help to improve it (e.g. focusing the search on the important decision nodes). We leave this for future work.

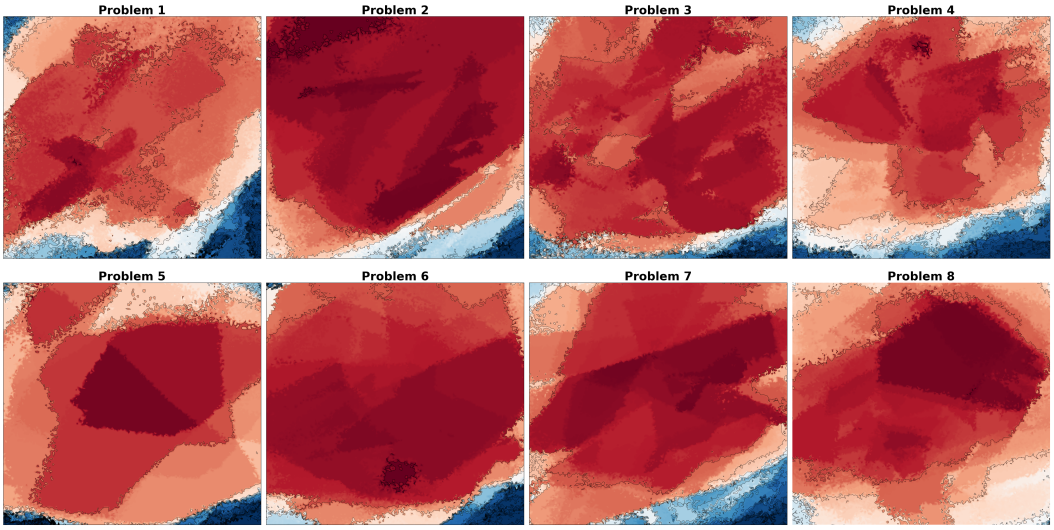

Figure 10: Latent space's heat map on 8 problem instances. We train a 2-dimensional COMPASS latent space and report its heat map on new unseen instances.

E.3 Exploring the Latent Policy Space at Inference Time

In this section, we present a comparison of various search methods applied to our latent policy space. Specifically, we analyze the performance of 4 strategies to illustrate choices made in the design of our method. In particular, we compare (1) a naive strategy, consisting of sampling 16 random vectors from the latent space, and sampling stochastically the resulting policies. This strategy is referred to as 'fixed policies' because we are not re-sampling in the latent space. (2) a strategy consisting of sampling uniformly from the latent space and rolling out the resulting policy, referred to as 'uniform-sampling'. This strategy can already illustrate the interest in having a latent space of diverse and specialized policies compared to sampling from a fixed set of policies. (3) CMA-ES search to navigate the latent space (4) our strategy, CMA-ES with several components (three), to illustrate the interest of being able to focus on distinct areas of the latent space.

Figure 11 illustrates the performances of the different search strategies on a set of TSP150 instances. We report the global performance as well as the latest batch performance. We can highlight three observations: (i) Sampling from the latent space brings significant improvement compared to sampling stochastically from a fixed set of policies. This shows the interest in having access to a space of diverse and specialized policies. (ii) Using an Evolution Strategy (like CMA-ES) to focus the search helps to make better use of the budget. We can see that the latest sampled solutions get better as the budget is used and the global performance improves faster compared to uniform sampling (iii) The final performance of the search gets better with three independent CMA-ES components rather than one. Being able to focus on several areas of the latent space enables us to avoid local optima and helps exploration. Interestingly, we can see that using all the budget for one component gives faster improvement but gets surpassed at the end of the budget. The choice of the accurate number of

components is a middle ground between risking staying stuck in local optima and not having enough time to converge. In most of our tasks, using two or three components proved to work best.

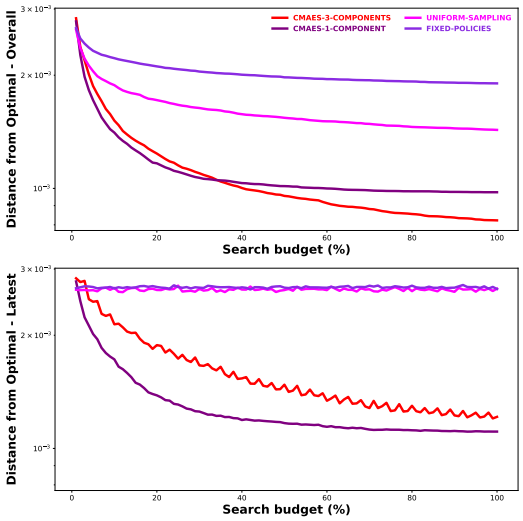

Figure 11: The performance of the different search strategies for TSP150 instances.

E.4 Alternative Strategies

At inference time, our goal is to explore the latent space to efficiently find promising areas of the latent space in order to obtain high-quality solutions within the given computational budget. As explained in Section 3.2, we decide to use multiple components of an evolution strategy to navigate our search space, because this approach is able to focus the search while being robust to local optima. Additionally, its parameters are easy to tune and work well on a wide range of tasks. Using multiple independent components of CMA-ES is also used in other works from the literature (Fontaine et al., 2020; Cully, 2021), out of the CO scope. In practice, our implementation of CMA-ES is inspired by the GitHub package QDax (Chalumeau et al., 2023b; Lim et al., 2022).

In Appendix E.3, we provide a comparison of our search strategy (CMA-ES with multiple components) with alternatives. In particular, we compare CMA-ES with a single component to uniform sampling in the latent space. Nevertheless, we also considered other alternatives for this search. First, we explored Bayesian Search, as it is a data-efficient search that can hence be appropriate when the budget is limited, which is our case. Nevertheless, this approach was never able to do better than random sampling, although we tried multiple sets of hyper-parameters. A potential limitation of Bayesian Search is that it needs to model the space with a Gaussian Process, which can be very tedious if the space is noisy. Our main hypothesis is that the latent space of COMPASS is too noisy to be correctly modeled under the budget constraint. We see two mitigation of this effect. The first one is to add a regularization term during the training to create a 'better-defined' latent space. The second would be to use a distribution to sample the space and use a Bayesian Search to optimize the parameters of this distribution. We leave these for future work.

Another alternative to Evolution Strategy is Gradient Descent, since we do have access to the derivatives. Nevertheless, gradient descent can easily be stuck in local optima, which is what we observe when analyzing the search of EAS in Section 4.3. That being said, note that EAS and COMPASS could be combined, which could be expected to provide very good results, particularly in CVPR. We also defer this to future work.

F Training Procedure

In this section, we describe our model's training process. We first have a pre-training phase where we either reuse an existing model or train a single model with an encoder and a non-conditioned decoder using the REINFORCE algorithm.

Next, we begin the training procedure which aims to create a diverse set of specialized policies. In this process, we designate our set of policies with a 16-dimensional latent space that contains vectors that can be used to condition our decoder model (i.e., a policy is parameterized by the decoder model parameters and the latent vector). Specifically, the vector is concatenated with the key, query, and value inputs of the decoder model for each step of the episode. In order to use the learned decoder from the pre-training phase, we initialize the extra weights with zeros such that they do not initially affect the output.

In the training procedure, the policy is trained to use the latent space to specialize to subareas of the problem distribution and this is achieved by solely training the policy conditioned on the latent variable that yields the highest reward for this instance. The details of the COMPASS training procedure is presented in Algorithm 1 and can be understood as follows. At each iteration, we sample N vectors from the latent space $\mathcal{Z}$ and a batch $\mathcal{B}$ of instances from the problem distribution $\mathcal{D}$. Then, for each instance ρ_i where $i \in 1, \ldots, \mathcal{B}$ and sampled vector z_k where $k \in 1, \ldots, N$, we rollout the conditioned policy $\pi_\theta(\cdot|z_k)$ on the problem instance (i.e., generate a trajectory which represents a solution to the instance). Next, for each instance, we determine the best-performing latent vector and this is done by computing which conditioned policy obtained the highest reward on the instance. Finally, we only train the best latent vector for each problem instance and use the REINFORCE loss to perform backpropagation through the network parameters of our model (including both the encoder and decoder networks).

Note, we only train on instances that have a conditioned policy performing strictly better than the remaining policies. For example, if two policies have the exact same performance which is the maximum amongst the set of sampled policies, neither of the policies is trained on the instance. This approach enhances specialization and promotes a more balanced distribution of instances solved by each policy, resulting in a better spread of contributions within the latent space.

Lastly, it is also important to note that the number of vectors sampled during training (i.e., the number of conditioned policies competing for each instance) plays a significant role in the training process. The number of sampled latent vectors can be potentially infinite, the constraint lies in hardware or runtime limitations. Increasing the "number of samples N in COMPASS promotes greater specialization and competition among policies in the latent space, which can be leveraged during inference when tackling new instances.

The final COMPASS neural solvers are trained until convergence, on a TPU v3-8. For each problem the training time and environment steps are: 4.5 days (110M steps) for TSP, 5.5 days (76.5M steps) for CVRP and 4.5 days (4.2M steps) for JSSP.

Algorithm 1 COMPASS Training

Input: problem distribution $\mathcal{D}$, latent space $\mathcal{Z}$, number of samples N, batch size $\mathcal{B}$, number of training steps K, policy π_θ with pre-trained parameters θ.
$\pi_\theta \leftarrow \texttt{Augment}(\pi_\theta)$ {Augment the pre-trained policy to take as input the latent variable.}
for step 1 to K **do**
  $\rho_i \leftarrow \texttt{Sample}(\mathcal{D})\ \forall i \in 1, \ldots, \mathcal{B}$
  $z_i \leftarrow \texttt{Sample}(\mathcal{Z})\ \forall i \in 1, \ldots, N$
  $\tau_i^k \leftarrow \texttt{Rollout}(\rho_i, \pi_\theta(\cdot|z_k))\ \forall i \in 1, \ldots, \mathcal{B}, \forall k \in 1, \ldots, N$
  $k_i^* \leftarrow \arg\max_{k \leq N} \mathcal{R}(\tau_i^k)\ \forall i \in 1, \ldots, \mathcal{B}$ {Select the best vector for each problem ρ_i.}
  $\nabla L(\theta) \leftarrow \frac{1}{B} \sum_{i \leq \mathcal{B}} \texttt{REINFORCE}(\tau_i^{k_i^*})$ {Backpropagate through these only.}
  $\theta \leftarrow \theta - \alpha \nabla L(\theta)$

G Hyper-parameters

In Table 5, we report all the hyper-parameters of our method. Interestingly, our method is quite robust to these parameters and we almost use the same for all types of tasks. Note that for TSP sizes 125, 150, and 200, we report the hyper-parameters used during inference for the multi-components CMAES algorithm but there is no training hyper-parameters to report as we model used was trained on instances of size 100.

Table 5: The hyper-parameters used in COMPASS.

Phase	Hyper-parameters	TSP100	TSP(125, 150)	TSP200	CVRP	JSSP
Train time	latent space dimension	16	-	-	16	16
	training sample size	128	-	-	128	128
	instances batch size	8	-	-	8	8
Inference Time	policy noise	1	0.1	0.1	0.1	0.1
	num. CMAES components	3	3	2	2	3
	CMAES init. sigma	100	100	100	100	100
	sampling batch size	16	16	16	16	16

H Model Checkpoints

We compare our method COMPASS to three main baselines: POMO (Kwon et al., 2020), Poppy (Grinsztajn et al., 2022), and EAS (Hottung et al., 2022) on three CO problem, TSP, CVRP, and JSSP. The checkpoints used to run the POMO and Poppy models for TSP and CVRP are taken from Grinsztajn et al. (2022), and the EAS baseline is executed using the same POMO checkpoint. Those checkpoints are publicly available at https://github.com/instadeepai/poppy. For JSSP, we trained the single agent and Poppy models ourselves as they were not available. Lastly, we provide the COMPASS checkpoints used to obtain the results reported in this work. All those are available at https://github.com/instadeepai/compass.

I Performance with a Small Evaluation Budget

In Table 6, we provide the performances of COMPASS and the baseline methods on the standard benchmark dataset with a smaller budget (typically 10% of the usual budget). It can be seen that COMPASS outperforms the baselines on a majority of the tasks (specifically, 3/4 for TSP, 2/4 for CVRP, and all 3/3 for JSSP). In practice, a solver is particularly useful if it is performing well for a wide range of budget, in other words, able to provide good solutions fast, but also able to continually improve if any additional budget is given. COMPASS is able to provide an efficient to improve with a budget while being competitive (or outperforming) state-of-the-art methods with low budget.

J Limitations

In this section, we address three potential limitations of our method, COMPASS. First, our generalization capacity is directly linked to and potentially limited by the diversity created by specializing on the training distribution. In particular, our objective function does not explicitly incorporate any term to promote further diversity within the latent space. The specialization we observe during training is a result of training only the top-performing conditioned policy on each instance. Although this approach is straightforward and effective, it can be argued that adding an unsupervised term during training could lead to a more diverse and ultimately higher-performing set of policies.

Another limitation lies in our training process: when sampling the latent space uniformly, the latent vectors evaluated may not include the true best vector, resulting in training a policy conditioned on a sub-optimal vector. This is both data-inefficient (evaluating several conditioned policies to only train on one trajectory) and sub-optimal (as specialization decreases by training a vector that does not correspond to the best conditioned policy).

To address this issue, we can consider two approaches. First, we can incorporate a search in the latent space during training to increase the likelihood of sampling the best vector. This way, we explore a wider range of latent vectors and improve the chances of discovering the optimal conditioning for each instance. Alternatively, we can train a prior distribution that, given an instance, provides information about the promising regions in the search space. By utilizing this prior distribution, we can sample latent vectors that are more likely to lead to better-performing conditioned policies. By leveraging prior knowledge or conducting a targeted search, we can enhance the efficiency and effectiveness of our training process.

Table 6: Results of COMPASS against the baseline algorithms for few-shot task on (a) TSP, (b) CVRP, and (c) JSSP problems. The methods are evaluated on instances from training distribution as well as on larger instance sizes to test generalization. The methods are only given 10% of the standard budget.

(a) TSP

	Training distr.		**Generalization**					
	$n = 100$		$n = 125$		$n = 150$		$n = 200$	
Method	Obj.	Gap	Obj.	Gap	Obj.	Gap	Obj.	Gap
Concorde	7.765	0.000%	8.583	0.000%	9.346	0.000%	10.687	0.000%
LKH3	7.765	0.000%	8.583	0.000%	9.346	0.000%	10.687	0.000%
POMO (greedy)	7.796	0.404%	8.635	0.607%	9.440	1.001%	10.933	2.300%
POMO	7.785	0.263%	8.619	0.424%	9.424	0.835%	11.066	3.545%
Poppy 16	7.769	0.045%	**8.593**	**0.115%**	9.373	0.293%	10.867	1.683%
EAS	7.783	0.226%	8.611	0.33%	9.399	0.564%	10.855	1.571%
COMPASS (ours)	**7.769**	**0.044%**	8.595	0.138%	**9.373**	**0.283%**	**10.787**	**0.933%**

(b) CVRP

	Training distr.		**Generalization**					
	$n = 100$		$n = 125$		$n = 150$		$n = 200$	
Method	Obj.	Gap	Obj.	Gap	Obj.	Gap	Obj.	Gap
LKH3	15.65	0.000%	17.50	0.000%	19.22	0.000%	22.00	0.000%
POMO (greedy)	15.874	1.430%	17.818	1.818%	19.750	2.757%	23.318	5.992%
POMO	15.767	0.75%	17.690	1.085%	19.610	2.031%	23.817	8.258%
Poppy 32	15.722	0.459%	**17.633**	**0.758%**	19.558	1.757%	23.907	8.666%
EAS	15.736	0.55%	17.636	0.779%	**19.526**	**1.593%**	23.202	5.463%
COMPASS (ours)	**15.681**	**0.201%**	17.648	0.846%	19.532	1.626%	**22.946**	**4.298%**

(c) JSSP

	Training distr.		**Generalization**			
	10×10		15×15		20×15	
Method	Obj.	Gap	Obj.	Gap	Obj.	Gap
OR-Tools	807.6	0.0%	1188.0	0.0%	1345.5	0.0%
L2D (Sampling)	871.7	8.0%	1378.3	16.0%	1624.6	20.8%
Single	869.7	7.684%	1312.4	10.47%	1517.6	12.787%
Poppy 16	857.1	6.126%	1306.1	9.94%	1516.2	12.686%
EAS	868.3	7.516%	1313.6	10.572%	1519.3	12.918%
COMPASS (ours)	**851.9**	**5.48%**	**1301.1**	**9.518%**	**1504.6**	**11.827%**

Lastly, we believe there is potential for improving the efficiency of our search process with a better-defined latent space. We have observed that the latent space is noisy, which poses a challenge when employing a Bayesian search method for exploration. Moreover, we anticipate that our CMA-ES search could achieve faster convergence if the space exhibits smoother characteristics. One possible approach to address this is by incorporating a regularization term into our training objective, which would promote a smoother and more well-behaved latent space.

K Extended related work

We focus the literature review of the main paper (section 2) on construction methods trained with reinforcement learning. In this section, we present several improvement methods, i.e. methods that start with an existing solution and learn to directly modify this solution to create a new one; close to the concept of local search. Those are interesting alternatives to construction methods, although there limitation lies in that they are usually more problem-specific, and are also highly biased by the choice of the initial solution used.

NeuRewriter (Chen and Tian, 2019) learns a policy that updates the solution, factorized in two steps: choosing the part of the solution, and then choosing the rule used to create a new solution. Note that this assumes the existence of a set of existing update rules to pick from. Concurrently, Hottung and Tierney (2020) builds upon the Large Neighborhood Search framework, which consists of destroying and repairing solutions. This method relies on heuristics to destroy the solution and a neural policy cor the reconstruction mechanism, and this is learned end-to-end with reinforcement learning. Their

approach is solely assessed on vehicle routing problems. Kim et al. (2021) reduces the dependence on the initial solution by learning a constructive model that generates diverse solutions, called seeds, that are used as starting points for an improvement neural policy. de O. da Costa et al. (2020) learns a neural policy that selects 2-opt operators to improve solutions of the TSP, and Wu et al. (2022) extends it to CVRP.

Ma et al. (2021) improves the performance of improvement methods (in VRP) based on transformers neural models by ensuring that the embedding that is learned to encode the instance being solved better capture the structure of the vehicle routing problems.

Overall, once a first solution has been created, using improvement approaches can be seen as a concurrent approach to policy adaptation to make the best use of a given budget to reach the best possible solution.

L Analysis of time consumption in COMPASS

We provide an analysis of the time consumption of COMPASS at inference time, by looking at its time performance on TSP100 on a TPU v3-8. The inference procedure can be decomposed into four steps: (1) encoding an instance (once per episode), decoding steps (99 times per episode), (3) environment steps (99 times per episode), and (4) CMA-ES steps (namely the sampling of the vectors and the update of the distribution parameters, once per episode). All values are estimated by averaging 500 evaluations and are reported in milliseconds (ms).

These results show that the adaptation mechanism of COMPASS (CMA-ES sampling and updates) is negligible compared to the remaining steps in the inference procedure, with a difference of three orders of magnitude. This explains the similarity in runtimes reported in Table 1 between COMPASS, Poppy and POMO: the encoding and decoding are mostly identical, and the additional adaptation mechanism is not significant.

Interestingly, the encoding step is done only once for the whole budget. Hence, the bigger the budget, the more negligible the encoding steps become compared to the decoding and environment steps. Furthermore, the time difference between CMA-ES steps and all the other steps (aggregated over an episode) is only expected to grow with the size of the instance. First, because those individual steps depend on the instance size (larger matrix to encode or decode, more computations to be carried in the environment) whereas the CMA-ES adaptation mechanism does not depend on the instance size. Second, the number of the decoding and environment step increases with the instance size, which is not the case for the CMA-ES steps.

On the contrary, the adaptation mechanism used in EAS is time-consuming, making the method much slower than POMO, Poppy and COMPASS. Additionally, EAS' adaptation time increases as the instance size increases.

Table 7: The time taken in a full episode rollout of COMPASS for TSP100. All values are averaged over 500 evaluations. CMA-ES (sampling and update) takes three orders of magnitude less time than the decoding steps.

Phase	Encoding	Decoding	Env. step	CMAES (sample & update)
Time for single event (ms)	32.27	3.01	0.69	0.28
Occurrence in an episode	1	99	99	1
Time over an episode (ms)	**32.27**	**297.99**	**68.31**	**0.28**

M Impacts of neural solver and search procedure in overall performance

M.1 Base solver vs adaptation mechanism

Our method COMPASS has two important aspects: a conditioned neural solver, that enables to it capture specialised policies in a latent space; and an adaptation mechanism that searches the latent space at inference time. In this subsection, we provide more insight into the impact of both aspects on the overall performance observed.

Our experimental results provide evidence that both aspects – (1) a well-trained conditional neural solver and (2) an efficient search algorithm – are critical for strong performance.

Point (1) is illustrated by Fig. 5 and Fig. 10 which shows high-performing regions for a given instance. Point (2) is demonstrated by Fig. 11 which shows the principled search method significantly outperforms random search. Additionally, we see that random search outperforms POMO and Poppy, confirming that the latent space "contains" high-performing and diverse policies.

To further illustrate the importance of those combined aspects, we provide two additional experiments. First, we under-train a conditioned neural solver by stopping the training procedure well before convergence and compare two COMPASS models (fully- vs. under-trained) solving TSP150 instances with two search methods (CMA-ES and uniform sampling). Those results are reported in Fig. 12. The results demonstrate that: (i) both search methods for the fully trained model outperform those for the under-trained model, showing the importance of our training procedure. (ii) uniform search on the fully-trained solver outperforms CMA-ES search on the under-trained model, showing that the search alone is not sufficient.

Second, we present the evolution of the latent space during training on a TSP150 instance, on Fig. 13. It can be seen that initially, the space is uniform (no specialized regions exist). However, as training progresses, high-performing regions emerge (shown in red) which indicates the specialization of policies within the latent space, and we also see the improved performance of the best conditioned policy.

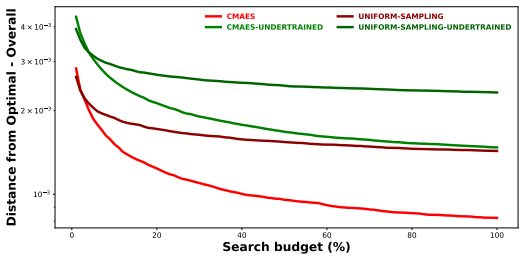

Figure 12: Performance of random and CMA-ES search on a conditioned solver that has not been trained until convergence (green) and a fully trained conditioned solver (red). Those are evaluated on 1000 instances of TSP150.

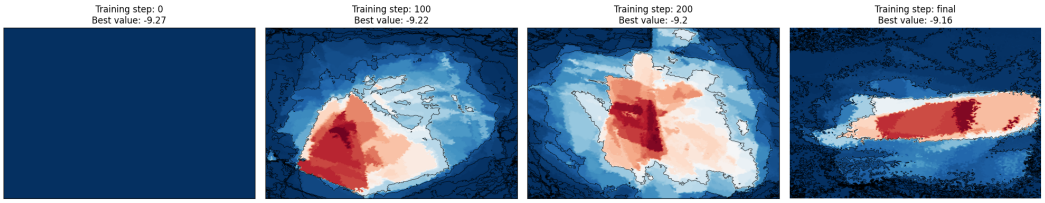

Figure 13: Evolution of the latent space during training on a given instance of TSP150.

M.2 Adaptation mechanism vs beam search

Several strategies can be consider to optimally leverage a neural solver within the budget constraints to find the best possible solutions. To evaluate the efficiency and performance of our adaptation mechanism, CMA-ES search, we can compare to an alternative approach, such as Beam search.

These two approaches are different paradigms: our approach searches the policy space and uses the policy to find a solution; Beam search explores directly the solution space, armed with a fixed policy. This can be approximated by comparing COMPASS with SGBS (Choo et al., 2022), a heuristic approach to improve the performance of POMO (Kwon et al., 2020). A naive application of this heuristic on COMPASS (sampling a random policy with no latent space search) is equivalent to POMO+SGBS. We report the results of POMO+SGBS in Table 3 and show that COMPASS outperforms POMO+SGBS on the whole benchmark. This validates that it is worth searching for a

good latent condition with the budget rather than fixing a random policy and using a beam search. Nevertheless, there may be a trade-off between search in latent space and heuristic solution search, which we leave for future work.

N Performance of our implementation of EAS

In order to report the performance of EAS on our whole set of experiments and in similar conditions as POMO, Poppy and COMPASS, we have re-implemented EAS in our codebase, in Jax. This implementation is open-sourced[2]. In this section, we report the results stated in the paper introducing EAS (Hottung et al., 2022) and report the results of our implementation in the same conditions.

In our implementation of EAS-Emb, we backpropagate the gradients through the whole decoder to update the embeddings instead of backpropagating only through the last attention layer. This might explain why the EAS performance are slightly better than those reported in the original paper, but incurs a higher computational cost.

Table 8: Results of EAS (paper results vs. our implementation) with instance augmentation for (a) TSP and (b) CVRP.

(a) TSP

	Training distr.			**Generalization**								
	$n = 100$			$n = 125$			$n = 150$			$n = 200$		
Method	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time
EAS (paper)	7.769	0.052%	5H	8.591	0.093%	57M	9.363	0.182%	2H	10.730	0.402%	4H
EAS (our implem.)	7.768	0.039%	7H	8.590	0.082%	66M	9.360	0.175%	138M	10.724	0.350%	206M

(b) CVRP

	Training distr.			**Generalization**								
	$n = 100$			$n = 125$			$n = 150$			$n = 200$		
Method	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time
EAS (paper)	15.63	-0.13%	9H	17.47	-0.17%	93M	19.22	0.00%	3H	22.19	0.86%	6H
EAS (our implem.)	15.62	-0.21%	13H	17.462	-0.22%	2H	19.213	-0.037%	5H	22.162	0.73%	7H

[2]Implementations available at https://github.com/instadeepai/compass