

Curiosity Driven Reinforcement Learning for Job Shop Scheduling

Alexander Nasuta^a, Marco Kemmerling^b, Hans Zhou^c, Anas Abdelrazeq^d
and Robert Schmitt^e

Chair of Intelligence in Quality Sensing, RWTH Aachen, Aachen, Germany

{alexander.nasuta, marco.kemmerling, hans.zhou, anas.abdelrazeq, robert.schmitt}@wzl-iqs.rwth-aachen.de

Keywords: Curiosity, Reinforcement Learning, Job Shop Problem, Combinatorial Optimization.

Abstract: The Job Shop Problem (JSP) is a well-known NP-hard problem with numerous applications in manufacturing and other fields. Efficient scheduling is critical for producing customized products in the manufacturing industry in time. Typically, the quality metrics of a schedule, such as the makespan, can only be assessed after all tasks have been assigned, leading to sparse reward signals when framing JSP as a reinforcement learning (RL) problem. Sparse rewards pose significant challenges for many RL algorithms, often resulting in slow learning behavior. Curiosity algorithms, which introduce intrinsic reward signals, have been shown to accelerate learning in environments with sparse rewards. In this study, we explored the effectiveness of the Intrinsic Curiosity Module (ICM) and Episodic Curiosity (EC) by benchmarking them against state-of-the-art methods. Our experiments demonstrate that the use of curiosity significantly increases the amount of states encountered by the RL agent. When the intrinsic and extrinsic reward signals are of comparable magnitude, the agent is with ICM module are able to escape local optima and discover better solutions.

1 INTRODUCTION


Production planning is a critical challenge across industries, with resource allocation and task scheduling being complex decisions. The Job Shop Problem (JSP) is particularly relevant in manufacturing, where efficient scheduling is essential for producing customized products and managing small batches (Błażewicz et al., 2019). In a JSP, each product is treated as a job consisting of tasks that must be processed on specific machines in a particular order. This flexibility makes JSP widely applicable but also highly complex, as it is an NP-hard problem, meaning that finding exact solutions is computationally infeasible for large instances. To tackle this complexity, heuristic and metaheuristic approaches are often used to find near-optimal solutions. Recently, reinforcement learning (RL) has emerged as a promising approach, allowing agents to learn adaptive scheduling heuristics through interaction with the environment. However, applying RL to JSPs poses challenges, particularly due to the sparse reward structure. In JSP,


the quality of a schedule is typically assessed only after all tasks are scheduled, leading to delayed rewards and a slow learning process.


One of the key issues in RL is the exploration-exploitation dilemma, where the agent must balance trying new actions (exploration) with using known successful actions (exploitation). This dilemma is especially problematic in sparse reward environments like JSPs, where it's difficult for the agent to identify which actions contributed to success or failure.


Curiosity-based exploration algorithms offer a solution approach by introducing intrinsic rewards that encourage the agent to systematically explore new states or actions, even without immediate external rewards. This approach has been shown to accelerate learning in various RL tasks, particularly in environments with sparse rewards (Pathak et al., 2017; Savinov et al., 2018). Despite its potential, curiosity-driven exploration has not been widely studied in the context of JSPs to the best of our knowledge.


This research aims to explore the impact of curiosity-based exploration on RL agents in a job shop environment. By integrating curiosity algorithms into a typical RL setup, we seek to improve the effectiveness of scheduling solutions.

^a  <https://orcid.org/0009-0007-5111-6774>

^b  <https://orcid.org/0000-0003-0141-2050>

^c  <https://orcid.org/0000-0002-7768-4303>

^d  <https://orcid.org/0000-0002-8450-2889>

^e  <https://orcid.org/0000-0002-0011-5962>

2 RELATED WORK

This section provides an overview of the existing literature on curiosity algorithms in RL and the application of RL to the JSP. It is divided into two subsections: Curiosity Algorithms in RL, which explores various approaches to intrinsic motivation in RL, and RL approaches for the JSP, which reviews how RL has been applied to solve JSP instances.

2.1 Curiosity Algorithms in RL

Curiosity-driven exploration has emerged as a possibility to guide RL agents to explore the state space in a systematic way in environments with sparse rewards. The primary challenge in sparse reward environments is the exploration-exploitation dilemma, where an agent must decide between exploring new actions or exploiting known successful actions. While this dilemma holds true for any RL setting it is especially severe, where rewards are infrequent or delayed, leading to slow and inefficient learning. To address this challenge, curiosity algorithms introduce intrinsic rewards, which encourage an agent to explore states or actions that are novel or uncertain, even in the absence of external rewards.

One of the simplest curiosity mechanisms is count-based curiosity, where the agent receives higher rewards for visiting less-explored states. The method's limitation lies in its infeasibility for environments with a vast state space. Bellemare et al. (2016) addressed this issue by introducing pseudo-counts, which estimate state visitation counts using a neural network, thereby enabling more efficient exploration in complex environments like Atari games.

Prediction-based curiosity, initially proposed by Schmidhuber (1991), is another prominent approach. It involves the agent predicting the next state based on the current state and action, with the intrinsic reward being the prediction error. This method allows the agent to focus on learning areas of the environment where its predictive model is less accurate, fostering more efficient exploration. Pathak et al. (2017) enhanced this approach with the Intrinsic Curiosity Module (ICM), which transforms states into a feature space to filter out statistical noise. The ICM has demonstrated significant improvements in learning efficiency in sparse reward environments like VizDoom and Super Mario Bros compared to approaches without a curiosity approach.

Episodic Curiosity (EC), introduced by Savinov et al. (2018), suggests a different concept of curiosity by defining novelty in terms of the agent's ability to reach a new state from previously encountered

states within a limited number of actions. If a state is deemed novel, it is stored in memory, and the agent is rewarded, thus promoting exploration of genuinely new areas of the environment. EC has shown superior performance in environments like VizDoom, DM-Lab, and MuJoCo, outperforming the ICM in the evaluated use cases.

2.2 RL Approaches for the JSP

The Job Shop Problem is a classic combinatorial optimization problem, widely recognized for its computational complexity as an NP-hard problem. The goal in JSP is to determine an optimal schedule for a set of jobs, each consisting of multiple tasks that must be processed on specific machines in a predefined order. Given its complexity, JSP has traditionally been addressed using heuristic and metaheuristic approaches. However, with recent advances in reinforcement learning, RL has emerged as a promising tool for solving JSPs.

Samsonov et al. (2021) introduced a reinforcement learning framework that utilizes a sparse reward function for JSP. In this approach, a discrete time simulation of the job shop environment is employed, where an RL agent assigns tasks to machines. The reward is sparse, provided only at the end of the scheduling process, and is inversely proportional to the makespan. This method allows the agent to focus on achieving near-optimal schedules but faces challenges due to the delayed nature of the reward.

In contrast, Tassel et al. (2021) proposed a dense reward function for JSP that is based on machine utilization. Here, the reward at each time step is determined by the area occupied by tasks in the Gantt chart and the extent of idle times. By optimizing the scheduled area, this method indirectly minimizes the makespan, allowing for more frequent rewards and thus faster learning.

Zhang et al. (2020) took a different approach by modeling the JSP as a disjunctive graph. The RL agent uses a graph neural network (GNN) to transform states into a latent space. The reward function in this model is based on the critical path in the disjunctive graph, allowing the agent to adapt to JSP instances of varying sizes. This method has demonstrated the ability of RL agents to generalize across different JSP scenarios, offering a scalable solution to the problem.

Nasuta et al. (2023) introduced a highly configurable RL environment, adhering to the Gym standard¹, aimed at JSPs. The study compared various

¹<https://www.gymnasium.dev/index.html>

reward functions, including sparse and dense formulations. While no single reward function emerged as universally superior across a wide range of JSP instances, dense reward functions based on machine utilization, such as those proposed by Tassel et al. (2021), tended to optimize the makespan more effectively than sparse rewards that target makespan directly. These findings suggest that dense reward signals can provide more consistent learning signals, leading to improved scheduling performance.

Recent approaches have also explored the combination of RL with Monte Carlo Tree Search (MCTS), a heuristic search method originally developed for combinatorial games. MCTS represents the solution space as a tree and employs random sampling to guide search efforts. Algorithms that combine RL with MCTS, often referred to as neural MCTS, have been notably applied in high-profile examples like AlphaGo and AlphaZero (Kemmerling et al., 2024b).

Oren et al. (2021) introduced an approach that integrates Deep Q-Learning for policy training in an RL setting but incorporates MCTS during production runs to generate superior solutions compared to relying solely on the learned policy.

Kemmerling (2024) explored in which cases neural MCTS outperforms conventional model-free reinforcement learning by analyzing job and operation diversity using Shannon entropy. Datasets with varying entropy levels were generated to assess each approach’s performance under different problem complexities. The results showed that while model-free agents struggled with high-entropy instances, neural MCTS could overcome this challenge by performing additional planning during decision-making, leading to solutions closer to the optimal even in more diverse and complex scheduling scenarios.

While RL approaches have shown promising results, they are prone to learn policies that result in local optima, leaving room for improvement. Sparse rewards, in particular, remain a significant challenge. We aim to address this by integrating curiosity-based algorithms into the RL framework for JSP, with the goal of investigating the learning efficiency and solution quality of curiosity-based algorithms.

3 METHODOLOGY

This section introduces the underlying optimization problem of a JSP and outlines the methodology used in our study, focusing on the design of a JSP environment as a Markov Decision Process (MDP) and the experimental setup employed to evaluate two curiosity-driven exploration algorithms: ICM intro-

duced by Pathak et al. (2017) and EC introduced by Savinov et al. (2018).

JSP Formalization. The JSP is a classical scheduling problem where a set of tasks $\mathcal{T} = \{T_1, T_2, \dots, T_N\}$ must be processed on a set of machines $\mathcal{M} = \{M_i\}_{i=1}^m$. Each job, representing the production of a specific product, is composed of a sequence of tasks, with each task corresponding to a specific production step. The problem considers a set of jobs $\mathcal{J} = \{J_j\}_{j=1}^n$, where the total number of tasks N is determined by the number of jobs n and the number of machines m , such that $N = n \cdot m$. A solution to the JSP, a feasible schedule, assigns a start time \hat{s}_α to each task T_α , while ensuring that the precedence constraints and machine availability constraints are not violated. Our work leverages a disjunctive graph approach to model the JSP, like Zhang et al. (2020) and Nasuta et al. (2023). In this approach, the disjunctive graph G consists of a set of nodes \mathcal{V} , directed edges \mathcal{A} , and undirected edges \mathcal{E} . The set of nodes \mathcal{V} includes the tasks \mathcal{T} and two fictitious nodes, the source T_0 and the sink T_* , such that $\mathcal{V} = \mathcal{T} \cup \{T_0, T_*\}$. The directed edges \mathcal{A} , termed conjunctive edges, represent precedence relations and are initially derived from the sequence of tasks within each job. The undirected edges, known as disjunctive edges, are introduced between tasks that require the same machine, reflecting the yet undecided order of processing on that machine. To generate a valid schedule, all disjunctive edges must be directed in such a way that the resulting graph remains acyclic. The makespan, defined as the total time required to complete all jobs, can be determined by finding the longest path, also known as the critical path, from T_0 to T_* in the fully scheduled graph.

The discrete optimization problem for minimizing the makespan can be formally stated as follows: Minimize s_* subject to (Błażewicz et al., 2019):

$$\hat{s}_\beta - \hat{s}_\alpha \geq p_\alpha \quad \forall (T_\beta, T_\alpha) \in \mathcal{A} \quad (1)$$

$$\hat{s}_\alpha \geq 0 \quad \forall T_\alpha \in \mathcal{T} \quad (2)$$

$$\hat{s}_\beta - \hat{s}_\alpha \geq p_\alpha \vee \hat{s}_\alpha - \hat{s}_\beta \geq p_\beta \quad \forall \{T_\beta, T_\alpha\} \in \mathcal{E}_i, \quad (3)$$

$$\forall M_i \in \mathcal{M}$$

Markov Decision Process. In this study the JSP is modelled as a Markov Decision Process to facilitate the training of RL agents. The state space spans all possible configurations of the partial schedule, while the action space consists of choices regarding which job’s next unscheduled operation to schedule.

Observation Space. For a RL setup the state representation needs to encode all information of a par-

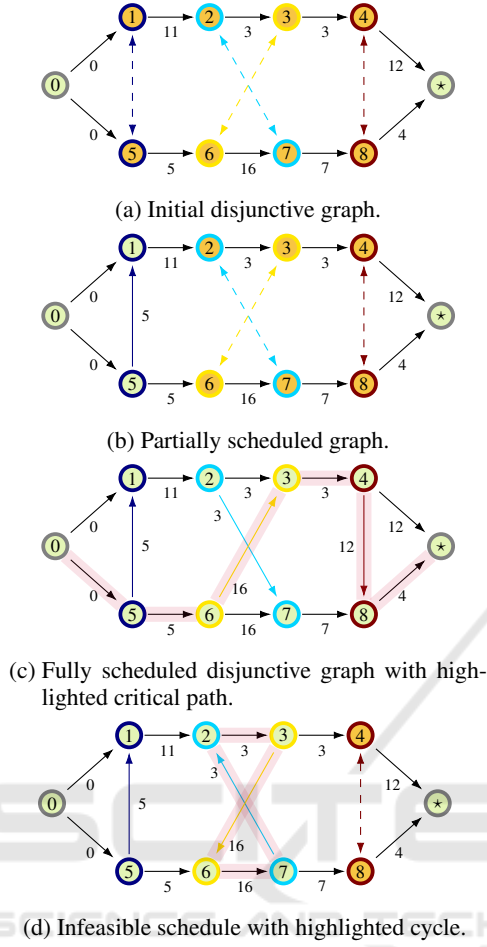


Figure 1: Disjunctive graph scheduling.

tial solution. When formulating the JSP using a disjunctive graph approach, any that allows to construct graphs, such as seen in Figure 1a- 1c is a valid state representation. We represent the graph by an adjacency matrix, that is extended with columns for the machine and the processing duration of a task. The state representation corresponding to Figure 1b is provided in the appendix, in Table 3.

Action Space. The action space consists of the set off all tasks along with an action mask, that masked out actions that might lead to infeasible schedules. In our setup tasks within a job are scheduled left to right similar to Zhang et al. (2020) and Nasuta et al. (2023), which ensures an acyclic graph at any point in time. For the instance in Figure 1 consists of the tasks $\{t_1, t_2, \dots, t_8\}$. Valid actions in Figure 1a are $\{t_1, t_5\}$. In Figure 1b $\{t_2, t_6\}$ are valid actions.

Reward Function. The reward function is a sparse reward function incentivizes actions that minimize the makespan:

$$r(s_t) = \begin{cases} -\frac{C}{C_{LB}} & \text{end of episode} \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

C_{LB} denotes a lower bound of the makespan for a specific instance. Therefore the reward is always in the same range across different instances. If C_{LB} equals the optimal makespan, the reward r will approach -1 as the agent finds better and better solutions. This reward function was introduced by Nasuta et al. (2023) as a trivial reward function, due to its simplicity. Despite Nasuta et al. (2023) incentives to formulate a dense reward function based on the machine utilization, we chose to use a sparse reward function to study the effects of curiosity algorithms, because curiosity was specifically introduced to excel on environments with a sparse reward structure.

Curiosity Algorithms. We consider the ICM approach by Pathak et al. (2017) and the EC approach introduced by Savinov et al. (2018) in our work, since these approaches are the most advanced curiosity algorithms demonstrating the best performance scoring in the domain of Atari games with sparse rewards. The following paragraphs cover both approaches in more detail.

Intrinsic Curiosity Module. The ICM, as introduced by Pathak et al. (2017), is an approach designed to address the challenge of sparse reward environments in reinforcement learning. It builds on the concept of intrinsic motivation by generating rewards based on the agent's inability to predict the outcomes of its actions, encouraging exploration in uncertain regions of the environment. The ICM operates by transforming raw states s_t and s_{t+1} into a feature space $\phi(s_t), \phi(s_{t+1})$, to reduce the influence of stochastic noise in the environment. The forward model predicts the next state in the feature space, and the intrinsic reward is calculated as the error between the predicted and actual subsequent state $\|\hat{\phi}(s_{t+1}) - \phi(s_{t+1})\|$. This reward drives the agent to explore areas where its predictions are less accurate. Additionally, the ICM includes an inverse model, which predicts the action taken by the agent based on the feature representation of the current and next state. This ensures that the feature space captures relevant dynamics, filtering out noise. The combination of the forward and inverse models is optimized using a shared optimizer, with a weighted sum of their errors guiding the network parameters.

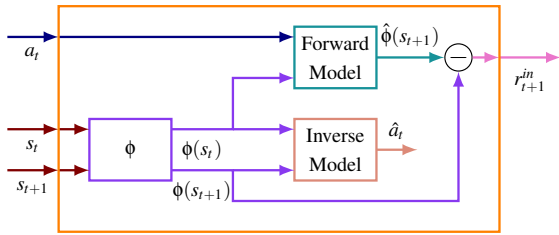


Figure 2: Signal block diagram for the ICM.

Episodic Curiosity. The Episodic Curiosity Module (EC), proposed by Savinov et al. (2018), provides a mechanism for detecting novelty in an agent’s environment by comparing the agent’s current state to previously encountered states stored in a memory buffer. Novelty, in this context, is defined by how many actions are required to move from a known state to the current state. Specifically, if more than a predefined number of actions, k , are needed to reach the current state s_t from any state in the memory buffer M , the state is considered novel and an intrinsic reward is generated. The EC utilizes two neural networks: an embedding network E , which transforms the current state s_t into a feature space $E(s_t)$, and a comparator network C , which estimates the reachability of s_t from stored states $s_m \in M$. The comparator C predicts a continuous reachability score between 0 and 1, with 1 indicating that s_t is reachable within k actions. These predictions form a reachability buffer R , which is then aggregated into a scalar value using a function F , typically set as the 90th-percentile to account for neural network approximation errors. This scalar is passed to a bonus function B , and if the resulting value b exceeds a novelty threshold $b_{novelty}$, the state is deemed novel, and $E(s_t)$ is added to the memory buffer. The intrinsic reward r_{t+1}^{in} is set to b when novelty is detected, or 0 otherwise. The shown in Figure 3 in this paper illustrates the interaction between these components and the flow of information within the EC system.

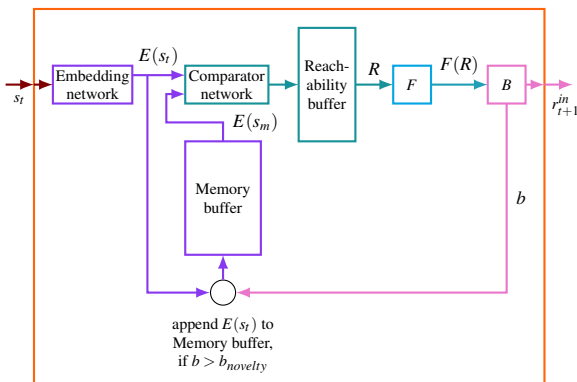


Figure 3: Signal block diagram for the EC.

3.1 Experimental Setup

This section covers the details on the experiment design and the on the specific software and frameworks used in this study. First the details on the implementation of of the investigated curiosity algorithms are covered, then we highlight or RL setup and the used libraries for RL and Experiment tracking.

Curiosity Module Implementation. We consider implementations of the ICM and EC within the reinforcement learning (RL) frameworks RLLib² and Stable Baselines3 Contrib (SB3)³. RLLib provides an implementation of ICM that can be integrated into an RL setup via its exploration API. We implemented EC using RLLib’s exploration API and validated it with the validation process used for RLLib’s ICM implementation. For the SB3 approach, we implemented both ICM and EC in the form of Gymnasium wrappers and evaluated the implementations in the FrozenLake environment. The metrics for these evaluation test cases are available on the Weights and Biases (WandB) platform^{4,5}.

Reinforcement Learning Setup. This study utilizes the Gymnasium environment introduced by Natusa et al. (2023), which was configured to realise the Observation and Action space described above along with the described sparse reward function. The curiosity modules in our setup are realised as gymnasium wrappers. For experiment tracking we use the WandB platform. After the validation of our curiosity implementations, we decided to use the SB3 setup for the evaluation curiosity algorithms on the JSP, because in is more convenient to realize action masking and incorporate custom WandB metrics. We consider the metrics described in Table 1 in our experiments: The Proximal Policy Optimization (PPO), originally proposed by Schulman et al. (2017), algorithm was chosen as the baseline reinforcement learning algorithm for this study. PPO, a robust and efficient actor-critic method, is well-suited for tasks with complex action spaces like JSP.

Hardware. All computations in this study were performed on an Apple Macbook Pro with a M1 Max chip with 64 Gb of shared memory.

²<https://docs.ray.io/en/latest/rllib/index.html>

³<https://sb3-contrib.readthedocs.io/en/master/>

⁴<https://wandb.ai/query/frozenlake-ray>

⁵<https://wandb.ai/query/frozenlake-sb3>

Table 1: Experiment Metrics.

Metric	Formula	Description
extrinsic return	G_{ex}	The return resulting from the environment
intrinsic return	G_{int}	The return resulting from the investigated curiosity module
total return	$G_{total} = G_{ex} + G_{int}$	The sum of intrinsic and extrinsic return.
visited states	$\ S\ $	The number of distance states the agent encountered.
loss	L	The loss from the optimiser, that trains the neural network inside a curiosity module

3.2 Experiment Design

There are two main approaches in the literature for solving the JSP using RL. The first approach aims to find the best possible solution for a specific instance within a given time budget, as explored by Tassel et al. (2021) and Kemmerling et al. (2024a). The second approach seeks to train an agent on a variety of instances, with the goal of generalizing to solve any new instance, as pursued by Zhang et al. (2020).

In this work, we focus on solving individual instances of the JSP. The source code for our experiments is available on GitHub⁶.

For evaluating the impact of curiosity modules in a RL setup for the JSP we utilize the well-known benchmark JSP instances from Fisher (1963): *ft06*, an instance with 6 jobs and 6 machines (size 6×6), and *ft10*, an instance with 10 jobs and 10 machines (size 10×10). These instances represent a range of complexities, with *ft06* being smaller and less computationally demanding, while *ft10* presents a more challenging scheduling scenario due to its larger size. The environment was configured with a sparse reward function, as described above, providing feedback only upon completion of a full schedule, thus creating a challenging exploration scenario ideal for evaluating curiosity-driven algorithms. To ensure that the PPO algorithm was well-tuned for each JSP instance size, a hyperparameter tuning process was conducted using the sweep functionality of WandB. The tuning focused on parameters such as the discount factor, neural network architecture, whether to turn off the intrinsic reward signal at some point and hyperparameters of the curiosity modules. An exhaustive list of the resulting parameters is provided in the appendix.

We divided the hyperparameter tuning process for the *ft06* instance into two stages. In the first stage, we performed hyperparameter tuning using randomly selected hyperparameters. Subsequently, we conducted a grid search over the parameters of the best-performing runs from the first stage. For the *ft06* instance, we applied this two-stage process to three

setups: a plain PPO, a PPO with an ICM, and a PPO with EC. The first stage consisted of 300 runs, each with 75k timesteps, followed by the second stage with 192 runs, each with 50k timesteps. In total, 1,504 hyperparameter tuning runs were completed. For each setup (PPO, PPO with ICM, and PPO with EC), the best-performing hyperparameter configuration—resulting in the lowest makespan—was further evaluated over 10 runs with a budget of 500k timesteps.

For the *ft10* instance, we performed 100 runs for the plain PPO and 49 runs for the ICM, both with randomly chosen hyperparameters and 1 million timesteps. After 500k timesteps, ICM exploration was turned on during hyperparameter tuning. The best-performing PPO configuration was selected as the baseline. Two ICM configurations were chosen for further evaluation: one that resulted in the lowest makespan and one that explored the highest number of states. Due to computational challenges, we decided to exclude the EC module in the larger-scale 10×10 scenario.

4 RESULTS AND DISCUSSION

This section presents the key findings of the experiments and provides a discussion of the outcomes. All recorded metrics are publicly available on WandB⁷.

The best performing hyperparameter configurations for *ft06* can be found for PPO, PPO with ICM and PPO with EC in Listings 1, 2 and 3, respectively. The performance of the agent in evaluation runs after hyperparameter tuning is visualized in Figure 4.

There are no significant differences in extrinsic returns or makespan among the evaluated groups. The performance of the PPO agent and the PPO agent with ICM is quite similar in terms of visited states. The PPO with the EC module visits slightly fewer states on average but remains largely within the range of the PPO and PPO with ICM. This difference is likely due to statistical fluctuations.

⁶<https://github.com/Alexander-Nasuta/Curiosity-Driven-RL-for-JSP>

⁷<https://wandb.ai/querry/MA-nasuta>

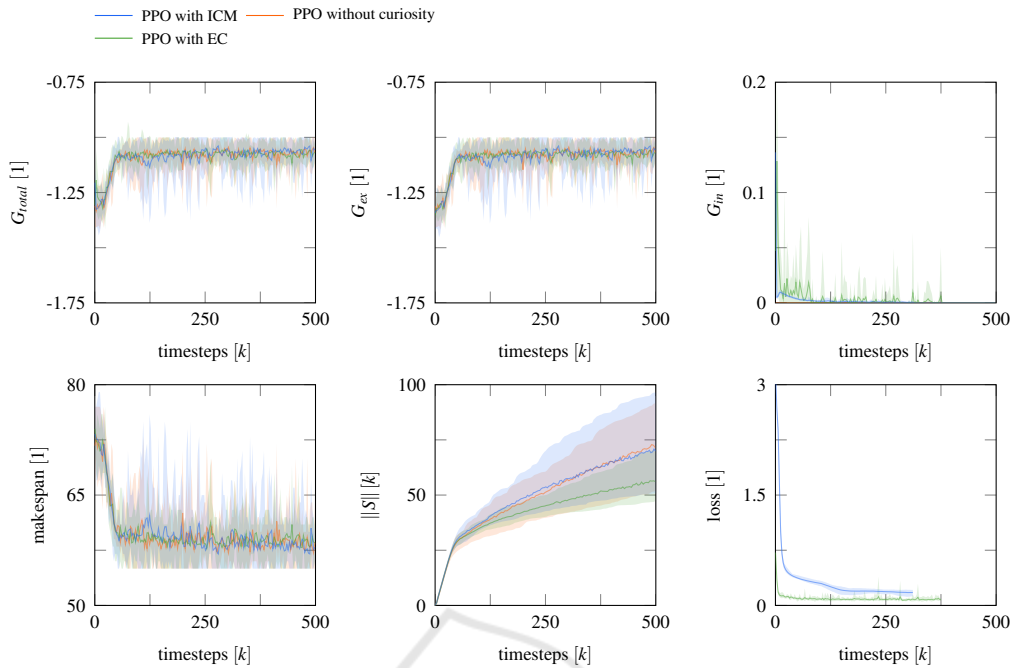


Figure 4: Evaluation runs on the ft06 instance.

No significant differences were observed between the baseline PPO and the PPO variants with curiosity modules, given the selected parameterizations. A possible explanation for this is the relatively low scale of the intrinsic reward signal, which might be too small to significantly influence the agent’s behavior. As shown in Figure 4, the intrinsic rewards are one to two orders of magnitude lower than the extrinsic rewards, making the intrinsic reward function more akin to noise than a meaningful goal-directed signal.

Additionally, we observed a considerable increase in computational demand when incorporating the EC module. This is due to the EC requires a prediction for every entry in the memory buffer at each timestep, in contrast to the ICM, which only requires two predictions per timestep. Another contributing factor to the increased computation is that our implementation of EC utilizes native Python data structures, rather than optimized implementation for managing the memory buffer, as used in the original implementation by Savinov et al. (2018). Due to these computational challenges, we decided to exclude the EC module in the larger-scale 10×10 scenario.

The 10×10 sized JSPs are significantly more complex than 6×6 sized ones, making them more appropriate for evaluating the impact of curiosity. Figure 5 presents selected runs from the hyperparameter tuning on the ft06 instance that illustrate how intrinsic curiosity affects scheduling.

Runs with high intrinsic rewards (pink shades in Figure 5) exhibit noticeably different behavior from those with low intrinsic rewards (green and blue shades). With low intrinsic rewards, agent behavior is similar to that without the ICM module, with the makespan dropping quickly and stabilizing and low exploration. In contrast, high intrinsic rewards initially result in more exploration and higher makespans. After 500k steps, the ICM is turned off in all runs, causing intrinsic rewards to drop to zero. Immediately afterward, the makespans of the green curves improve, indicating a shift from exploration to exploitation.

Higher intrinsic rewards clearly lead to increased exploration, while low intrinsic rewards appear to act as noise, with little impact on behavior. In the case of an ICM, the intrinsic reward is the prediction error for the next state, scaled by the parameter η . Hence, η is crucial in determining the magnitude of the intrinsic reward signal. Models with poor predictions generate higher intrinsic rewards, driving more exploration.

For the comparison between plain PPO and PPO with ICM, two runs from the ICM hyperparameter tuning were selected: the best-performing run (kind-sweep-18 with $\eta = 0.0276$) and the run with the highest level of exploration (still-sweep-27 with $\eta = 0.0615$). In the following, we refer to PPO with ICM parameterized as kind-sweep-18 as setup I, PPO with ICM parameterized as still-sweep-27 as setup II, and PPO without curiosity-driven explo-

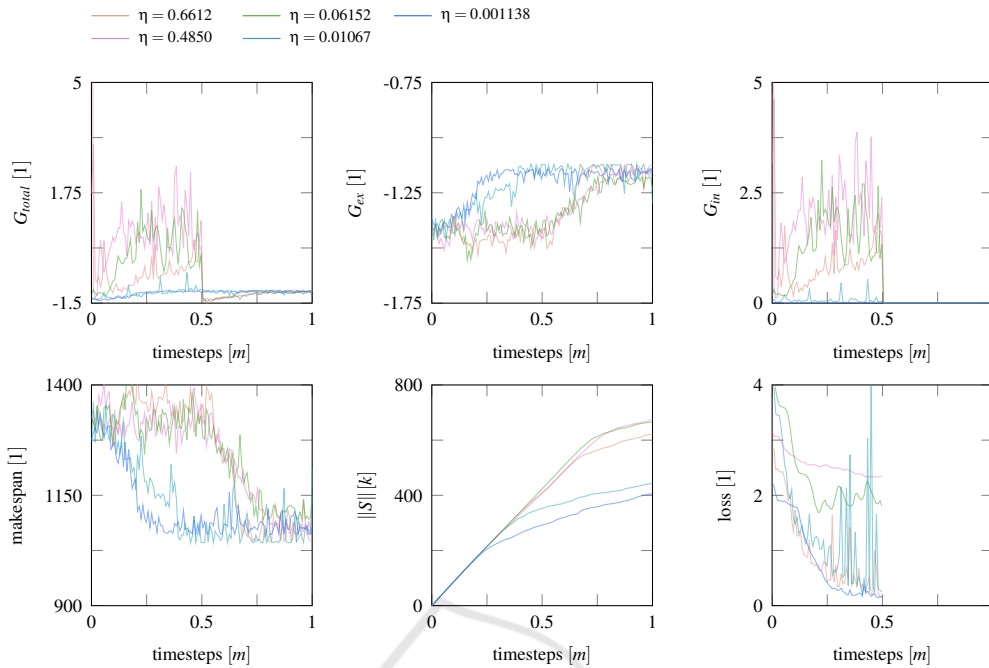


Figure 5: Selected hyperparameter tuning runs for a PPO with ICM on the ft10 instance.

ration as setup III. Both curiosity setups were compared to a PPO agent without curiosity, using $4m$ timesteps. These resulting runs are illustrated in Figure 6.

Setup II has a low η of 0.0615 but high, fluctuating intrinsic rewards due to large prediction errors from the ICM model. In contrast, Setup I achieved the lowest makespan, with intrinsic rewards ranging from 0.1 to 0.3 and less fluctuation. These variations likely stem from differences in the complexity of the internal neural networks, with Setup I having more layers and nodes than Setup II. After $2m$ timesteps, the ICM was switched off. Figures 6 shows that agents with ICM explore significantly more states pairs than PPO without curiosity in setup III. The makespan of Setup II remains high while the ICM is active, but improves after it is turned off. Across all setups, the makespan improves initially but eventually reaches a plateau. Periods of decreasing makespan correspond with high exploration rates, while stagnation in the makespan is associated with reduced exploration. Setup I suggests that a well-tuned ICM can drive exploration into new regions, potentially breaking out of local minima.

A comparison to other approaches found in the literature is presented in Table 2. This table also includes specific run names that can be used to locate the corresponding experiments on the WandB platform. Our setup I slightly outperforms other disjunctive graph-based methods, such as the *trivial* and *graph-tassel* approaches proposed by Nasuta et al..

However, the time-based approach introduced by Tassel et al. performs significantly better than both our method and the other disjunctive graph approaches examined in Nasuta et al. (2023). Nasuta et al. concluded that a dense reward function based on machine utilization is most effective for solving Job Shop Scheduling Problems (JSPs) using reinforcement learning. Our results suggest that introducing curiosity into a disjunctive graph-based approach offers only marginal improvements, while requiring additional computational resources for calculating the intrinsic reward and training the neural networks in the curiosity module. The *tassel* approach demonstrates that a lower makespan can be achieved by leveraging machine utilization to densify the reward structure in JSPs. We presume that, although curiosity can help escape local minima, a reward structure based on machine utilization is more likely to yield better results when computational resources are constrained. This is because it enables more timesteps within a given timeframe, avoiding the overhead introduced by the intrinsic curiosity signal.

5 CONCLUSION

Our experiments demonstrate that the use of curiosity significantly increases the number of states encountered by the RL agent. Moreover, when the intrinsic and extrinsic reward signals are of comparable mag-

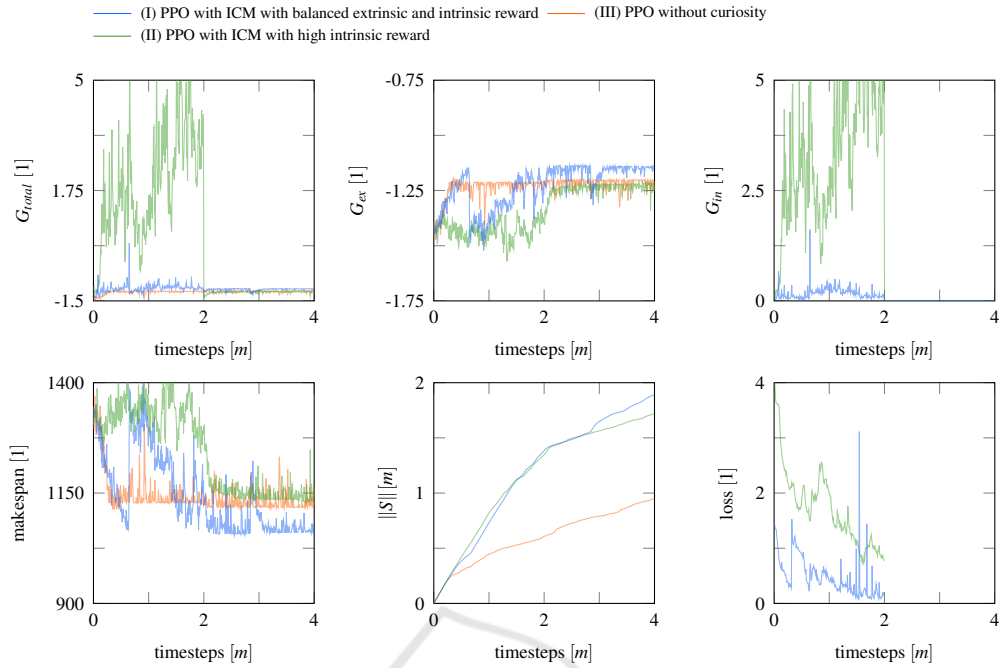


Figure 6: Evaluation runs on the ft10 instance.

Table 2: Comparing curiosity to other approaches in the literature.

Reward signal	Formula	Makespan after 2.5 m timesteps	Makespan after 4 m timesteps
tassel (Nasuta et al., run dazzling-sweep-3)	$r(s_t) = p_{aj} - \sum_{m \in \mathcal{M}} \text{empty}_m(s_t, s_{t+1})$	975	-
graph tassel (Nasuta et al., run skilled-sweep-17)	$r(s_t) = \frac{\sum_{\alpha} p_{\alpha}}{ \mathcal{M} \max_{\alpha} \delta_{\alpha} + p_{\alpha}}$	1147	-
trivial (Nasuta et al., run dulcet-sweep-18)	$r(s_t) = \begin{cases} -\frac{c}{c_{LB}} & \text{end of episode} \\ 0 & \text{otherwise} \end{cases}$	1216	-
trivial (run upbeat-pyramid-1785)	$r(s_t) = \begin{cases} -\frac{c}{c_{LB}} & \text{end of episode} \\ 0 & \text{otherwise} \end{cases}$	1132	1120
trivial + icm (run divine-butterfly-1795)	$r(s_t) = \frac{\eta}{2} \ \hat{\phi}(s_{t+1}) - \phi(s_{t+1})\ ^2 + \begin{cases} -\frac{c}{c_{LB}} & \text{end of episode} \\ 0 & \text{otherwise} \end{cases}$	1153	1158
trivial + icm (run worthy-morning-1787)	$r(s_t) = \frac{\eta}{2} \ \hat{\phi}(s_{t+1}) - \phi(s_{t+1})\ ^2 + \begin{cases} -\frac{c}{c_{LB}} & \text{end of episode} \\ 0 & \text{otherwise} \end{cases}$	1070	1065

nitude, the agent, along with the curiosity module, is able to escape local optima and discover better solutions. Low intrinsic reward signals do not affect the agent’s learning behavior and can be regarded as statistical noise. On the other hand, high intrinsic reward signals promote greater exploration. However, when the intrinsic reward dominates, the agent performs excessive exploration, which does not necessarily benefit the overall objective—namely, makespan optimization. Beneficial outcomes were only observed when the intrinsic and extrinsic rewards were balanced in magnitude.

We found that the ft06 instance (6x6) is likely too simple to provide a competitive advantage for a curiosity-based approach. However, for the larger ft10 instance (10x10), we observed notable benefits from incorporating an ICM. Therefore, we conclude that an ICM is beneficial only when the JSP is sufficiently complex—at least of size 10x10, according to our observations.

We also observed that incorporating both ICM and EC introduces additional computational demands. Specifically, integrating EC is computationally infeasible without an optimized memory buffer implementation. While the ICM allowed the agent to escape lo-

cal optima, further research suggests that approaches based on machine utilization may hold more promise for optimizing JSPs, particularly in terms of computational efficiency.

ACKNOWLEDGEMENTS

This work has been supported by the FAIR-Work project (www.fairwork-project.eu) and has been funded within the European Commission’s Horizon Europe Programme under contract number 101069499. This paper expresses the opinions of the authors and not necessarily those of the European Commission. The European Commission is not liable for any use that may be made of the information contained in this paper.

REFERENCES

- Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., and Munos, R. (2016). Unifying count-based exploration and intrinsic motivation. *Advances in neural information processing systems*, 29.
- Błażewicz, J., Ecker, K. H., Pesch, E., Schmidt, G., Sterna, M., and Weglarz, J. (2019). *Handbook on scheduling: From theory to practice*. Springer.
- Fisher, H. (1963). Probabilistic learning combinations of local job-shop scheduling rules. *Industrial scheduling*, pages 225–251.
- Kemmerling, M. (2024). *Job shop scheduling with neural Monte Carlo Tree Search*. PhD thesis, Dissertation, Rheinisch-Westfälische Technische Hochschule Aachen.
- Kemmerling, M., Abdelrazeq, A., and Schmitt, R. H. (2024a). Solving job shop problems with neural monte carlo tree search. In *ICAART (3)*, pages 149–158.
- Kemmerling, M., Lütticke, D., and Schmitt, R. H. (2024b). Beyond games: a systematic review of neural monte carlo tree search applications. *Applied Intelligence*, 54(1):1020–1046.
- Nasuta, A., Kemmerling, M., Lütticke, D., and Schmitt, R. H. (2023). Reward shaping for job shop scheduling. In *International Conference on Machine Learning, Optimization, and Data Science*, pages 197–211. Springer.
- Oren, J., Ross, C., Lefarov, M., Richter, F., Taitler, A., Feldman, Z., Di Castro, D., and Daniel, C. (2021). Solo: search online, learn offline for combinatorial optimization problems. In *Proceedings of the international symposium on combinatorial search*, volume 12, pages 97–105.
- Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning*, pages 2778–2787. PMLR.

- Samsonov, V., Kemmerling, M., Paegert, M., Lütticke, D., Sauermann, F., Gützlaff, A., Schuh, G., and Meisen, T. (2021). Manufacturing control in job shop environments with reinforcement learning. In *ICAART (2)*, pages 589–597.
- Savinov, N., Raichuk, A., Marinier, R., Vincent, D., Pollefeys, M., Lillicrap, T., and Gelly, S. (2018). Episodic curiosity through reachability. *arXiv preprint arXiv:1810.02274*.
- Schmidhuber, J. (1991). A possibility for implementing curiosity and boredom in model-building neural controllers.
- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.
- Tassel, P. P. A., Gebser, M., and Schekotihin, K. (2021). A reinforcement learning environment for job-shop scheduling. In *2021 PRL Workshop—Bridging the Gap Between AI Planning and Reinforcement Learning*.
- Zhang, C., Song, W., Cao, Z., Zhang, J., Tan, P. S., and Chi, X. (2020). Learning to dispatch for job shop scheduling via deep reinforcement learning. *Advances in Neural Information Processing Systems*, 33:1621–1632.

APPENDIX

```
# Discount factor
gamma: 0.99013
# Factor for trade-off of bias vs variance for Generalized
# Advantage Estimator
gae_lambda: 0.9
# Whether to normalize the advantage or not
normalize_advantage: True
# Number of epoch when optimizing the surrogate loss
n_epochs: 28
# The number of steps to run for each environment
# per update
n_steps: 432
# The maximum value for the gradient clipping
max_grad_norm: 0.5
# The learning rate of the PPO algorithm
learning_rate: 6e-4
policy_kwargs:
  net_arch:
    # Hidden layers of the policy network
    pi: [90, 90]
    # Hidden layers of the value function network
    vf: [90, 90]
  # Whether to use orthogonal initialization or not
  ortho_init: True
  # Activation function of the networks
  activation_fn: torch.nn.ELU
  optimizer_kwargs:
    # For the Adam optimizer
    eps: 1e-7
```

Listing 1: PPO hyperparameter tuning results for the *ft06* instance.

Table 3: Normalized representation of the disjunctive graph in Figure 1b.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	M_1	M_2	M_3	M_4	p
T_1	0	$\frac{11}{16}$	0	0	0	0	0	0	1	0	0	0	$\frac{11}{16}$
T_2	0	0	$\frac{3}{16}$	0	0	0	0	0	0	1	0	0	$\frac{3}{16}$
T_3	0	0	0	$\frac{3}{16}$	0	0	0	0	0	0	1	0	$\frac{3}{16}$
T_4	0	0	0	0	0	0	0	0	0	0	0	1	$\frac{12}{16}$
T_5	$\frac{5}{16}$	0	0	0	0	$\frac{5}{16}$	0	0	1	0	0	0	$\frac{5}{16}$
T_6	0	0	0	0	0	0	$\frac{16}{16}$	0	0	0	1	0	$\frac{16}{16}$
T_7	0	0	0	0	0	0	0	$\frac{7}{16}$	0	1	0	0	$\frac{7}{16}$
T_8	0	0	0	0	0	0	0	0	0	0	0	1	$\frac{4}{16}$

```

# Weighting for the ICM loss function
beta: 0.161
# Scaling factor for the intrinsic reward
eta: 0.0012
# Learning rate of the ICM optimizer
lr: 0.00059
# Dimension of the feature space
feature_dim: 1440
# Hidden layers of the feature network
feature_net_hiddens: [80]
# Activation function of the feature network
feature_net_activation: "relu"
# Hidden layers of the inverse model
inverse_feature_net_hiddens: [80]
# Activation function of the inverse model
inverse_feature_net_activation: "relu"
# Hidden layers of the forward model
forward_fcnet_net_hiddens: [100, 100]
# activation function of the forward model
forward_fcnet_net_activation: "relu"
# Memory capacity for (s_t, a_t, s_{t+1}) triples
memory_capacity: 12852
# Number of samples used for an optimization step
maximum_sample_size: memory_capacity * 0.875,
# Whether to shuffle the samples for optimization or not
shuffle_samples: True
# Whether to clear the memory after an episode or not
clear_memory_on_end_of_episode: False
# Whether do an optimization step at the end of an
# episode or not
postprocess_on_end_of_episode: True
# Whether to clear the memory every X steps.
# None = no clearing
clear_memory_every_n_steps: None
# Whether to do an optimization step every X time steps.
postprocess_every_n_steps: None
# Number of timesteps the ICM provides intrinsic rewards
exploration_steps: total_timesteps * 0.625

```

Listing 2: ICM hyperparameter tuning results for the *ft06* instance.

```

# Scaling factor for the intrinsic reward
alpha: 0.0025
# Parameter for the reward bonus function
beta: 0.5
# Threshold for novelty
b_novelty: 0.0
# Comparator hidden Layers
comparator_net_hiddens: [80, 80, 80]
# Comparator activation function
comparator_net_activation: "relu"
# Feature space dimension
embedding_dim: int = 288
# Hidden Layers of the embedding network
embedding_net_hiddens: [80]
# activation function of the embedding network
embedding_net_activation: "relu"
# Learning rate of the EC network training
lr: 0.005
# Spacing factor between positive and negativ examples
# for training
gamma: 2
# Memory capacity of the EC module
episodic_memory_capacity: 500
# Whether to clear the memory on the end of an episode
clear_memory_every_episode: False
# Number of timesteps the EC provides intrinsic rewards
exploration_steps: total_timesteps * 0.75

```

Listing 3: EC hyperparameter tuning results for the *ft06* instance.

```

# Discount factor
gamma: 0.9975
# Factor for trade-off of bias vs variance for
# Generalized Advantage Estimator
gae_lambda: 0.925
# Whether to normalize the advantage or not
normalize_advantage: True
# Number of epoch when optimizing the surrogate loss
n_epochs: 5
# The number of steps to run for each environment per update
n_steps: 600
# The maximum value for the gradient clipping
max_grad_norm: 0.5
# The learning rate of the PPO algorithm
learning_rate: 0.0004908203073130629
policy_kwargs:
  net_arch:
    # Hidden layers of the policy network
    pi: [75, 75, 75]
    # Hidden layers of the value function network
    vf: [75, 75, 75]
  # Whether to use orthogonal initialization or not
  ortho_init: True
  # Activation function of the networks
  activation_fn: torch.nn.ELU
  optimizer_kwargs:
    # For the Adam optimizer
    eps: 1e-8

```

Listing 4: PPO hyperparameter tuning results for the *ft10* instance.

```

# Weighting for the ICM loss function
beta: 0.10214627450350428
# Scaling factor for the intrinsic reward
eta: 0.06152020441380021
# Learning rate of the ICM optimizer
lr: 0.00029639942323414395
# Dimension of the feature space
feature_dim: 576
# Hidden layers of the feature network
feature_net_hiddens: [25, 25, 25, 25]
# Activation function of the feature network
feature_net_activation: "relu"
# Hidden layers of the inverse model
inverse_feature_net_hiddens: [25, 25, 25, 25]
# Activation function of the inverse model
inverse_feature_net_activation: "relu"
# Hidden layers of the forward model
forward_fcnet_net_hiddens: [25, 25, 25, 25]
# Activation function of the forward model
forward_fcnet_net_activation: "relu"
# Memory capacity for (s_t, a_t, s_{t+1}) triples
memory_capacity: 6500
# Number of samples used for an optimization step
maximum_sample_size: memory_capacity * 0.875,
# Whether to shuffle the samples for optimization or not
shuffle_samples: True
# Whether to clear the memory after an episode or not
clear_memory_on_end_of_episode: False
# Whether do an optimization step at the end of an episode or not
postprocess_on_end_of_episode: True
# Whether to clear the memory every X steps. None = no clearing
clear_memory_every_n_steps: None
# Whether to do an optimization step every X time steps
postprocess_every_n_steps: None
# Number of timesteps the ICM provides intrinsic rewards.
exploration_steps: total_timesteps * 0.5

```

Listing 5: ICM parameters of run still-sweep-27.

```

# Weighting for the ICM loss function
beta: 0.16775958687453613
# Scaling factor for the intrinsic reward
eta: 0.0276453876243576
# Learning rate of the ICM optimizer
lr: 0.000043689437240868784
# Dimension of the feature space
feature_dim: 288
# Hidden layers of the feature network
feature_net_hiddens: [75, 75, 75, 75, 75, 75]
# Activation function of the feature network
feature_net_activation: "relu"
# Hidden layers of the inverse model
inverse_feature_net_hiddens: [75, 75, 75, 75, 75, 75]
# Activation function of the inverse model
inverse_feature_net_activation: "relu"
# Hidden layers of the forward model
forward_fcnet_net_hiddens: [125, 125, 125, 125, 125, 125]
# Activation function of the forward model
forward_fcnet_net_activation: "relu"
# Memory capacity for (s_t, a_t, s_{t+1}) triples
memory_capacity: 1300
# Number of samples used for an optimization step
maximum_sample_size: memory_capacity * 0.75,
# Whether to shuffle the samples for optimization or not
shuffle_samples: True
# Whether to clear the memory after an episode or not
clear_memory_on_end_of_episode: False
# Whether do an optimization step at the end of an episode or not
postprocess_on_end_of_episode: True
# Whether to clear the memory every X steps. None = no clearing
clear_memory_every_n_steps: None
# Whether to do an optimization step every X time steps.
# None = no step based optimization
postprocess_every_n_steps: None
# Number of timesteps the ICM provides intrinsic rewards.
exploration_steps: total_timesteps * 0.5

```

Listing 6: ICM parameters of run kind-sweep-18.