Towards a fast and open-ended environment for curriculum learning in deep-reinforcement learning

Bachelor's Project Thesis

B. D. van Bruggen, S5220394, b.d.van.bruggen@student.rug.nl
Supervisor: G.A. Pourcel, g.a.pourcel@rug.nl

Abstract: Reinforcement learning (RL) struggles with complex, sparse-reward tasks due to delayed feedback and the need for large amounts of computational resources. Curriculum learning and reward shaping, especially when guided by large language models (LLMs), help agents learn complex tasks by providing goals of intermediate difficulty and intermediate feedback. In contrast, open-ended environments increase the challenge by introducing vast and unstructured goal spaces, which is crucial for studying agents that can generalize and adapt beyond fixed tasks. This thesis presents craftax-wiring (code provided at `https://github.com/pot4/Craftax-wiring`), an extension of the JAX-based Craftax-classic environment, designed to enable a wide range goals and curriculum learning for such goals. Inspired by Minecraft's redstone and Terraria's wiring systems, the environment introduces wires, logic gates, and inputs and outputs that simulate simple logic circuits. Experiments involving a half-adder construction task reveal that agents fail to learn without learning support, confirming the task's complexity. When guided through a handcrafted reward-shaping structure, agents show successful patterns that indicate the learning of the task, showing the environment's suitability for curriculum learning. Additionally, LLM evaluation confirms the high interestingness of the new goals. Performance tests show that the new environment runs at a similar speed when compared to the Craftax environment. Overall, Craftax-wiring shows promise for being a fast and open-ended environment for curriculum learning in RL research.

1 Introduction

Reinforcement learning (RL) is a prominent subfield of artificial intelligence (AI) that focuses on training agents to make decisions by interacting with an environment. However, despite great accomplishments and developments in the recent years, certain challenges still remain in scaling RL to solve complex tasks.

1.1 Complex, sparse reward tasks

Complex tasks in RL are usually associated with sparse rewards since feedback for actions can have very significant delays.

A solution for this is to provide more frequent feedback in the form of intermediate rewards. This is the main purpose of reward-shaping. Reward-shaping fortifies a certain policy for the agent which introduces the bias of the designer. This can result in the agent not finding the optimal policy for achieving the goal. To combat this, automated reward-shaping using Large Language Models (LLM) could provide a way for agents to learn a policy that is closer to optimal. This is because the LLMs can search the task space and select the best reward-shaping method. Another solution is to introduce intermediate goals that reward the agent for desired behavior which can be achieved using curriculum learning. This is a method that increases the complexity of the experience of an agent over time during the training process (Portelas et al., 2020). Such learning methods optimize the training process by prioritizing tasks according to the agent's skill level, ensuring that the agent first masters base level tasks before progressing. The agent starts with learning easier skills and progresses to learning more difficult skills using the knowledge acquired from the easier tasks. This learning progression falls in line with how humans learn tasks. Recently, new developments have shown promise in automated curriculum learning (ACL) leveraging the use of LLMs (Ryu et al., 2024). These LLMs generate tasks and design curricula for them as opposed to hand-crafted curriculum structures. Furthermore, LLMs can also generate and code goals themselves dynamically (Faldor et al., 2024).

1.2 Open-endedness

Open-ended learning systems are systems that "continue to learn new, interesting things up to the limit of modern computation" (Parker-Holder, 2022). A truly open-ended learning algorithm can have diverse and massive implications and it is believed that the existence of such systems are achievable. Open-ended systems can have important applications in practical domains: from engineering domains such as architecture to creative sectors such as music and art generation. Researchers argue that the challenge of open-endedness is related to the existence of AI itself: "If we accept that evolution on Earth is an open-ended process, then perhaps the deepest connection is that human-level intelligence on Earth is one of its many products. In other words, open-endedness could be a prerequisite to AI, or at least a promising path to it." (Stanley et al., 2017). Furthermore, open-ended environments are environments that allow for a wide range of goals with basically unlimited novelty. Agents within such environments are not trying to solve a specific goal but to keep learning a diverse amount of goals. For example: the game Space Invaders is not open-ended since there is a clear specific goal which is shooting aliens and survive. There is a single dominant strategy and not much possibility for creativity or novelty. In open-ended environments, there are often multiple ways to reach goals and goals are generated dynamically or not explicitly defined. This makes such environments are ideal for the goal and curriculum generating LLMs mentioned before.

1.3 GPU environment

One classic limitation is that RL typically requires large amounts of data and computational resources. Typically, deep reinforcement learning environments are run on CPUs, whereas the agent and it's learning algorithm (RL) is accelerated on the GPU (Mnih et al., 2013). This requires complex algorithms to schedule the CPU and GPU and shuffle data around to synchronize them. A solution for this is to run the whole setup on the GPU. GPUs can run thousands of agents and environments in parallel and this method avoids memory transfers to the CPU. Some GPU accelerated environments can run at hundreds of millions of steps per second compared to the same environment running using CPUs which achieves thousands of steps per second (Freeman et al., 2021). This can be achieved using JAX, a programming language which is similar to python but compiles the code to accelerated linear algebra (xla) which allows it to run on hardware accelerators such as GPUs.

1.4 Research focus

For these reasons, an environment should exist for RL agents in which they are able to learn complex tasks using, that is open-ended while also being optimized for the use of computational resources. This brings us to the question "Can we create a fast and open-ended environment for curriculum learning in deep-reinforcement learning?". For these purposes, an extension on the craftax-classic RL benchmark environment (Matthews et al., 2024) has been created. This original environment is a JAX based re-implementation of Crafter (Hafner, 2021) which is a top-down 2D open-world survival game. RL environments before this were relatively simple and generally focus on a single task. Crafter, however, is a simplified version of Minecraft (Mojang, 2011), a popular and a very open-ended video game which has been used for RL purposes by various researchers. Rewards in craftax-classic are primarily focused on survival aspects (e.g. DEFEAT_ZOMBIE, EAT_PLANT, COLLECT_DIAMOND), which is why an extension providing the necessities for more diverse goals is necessary for goal and curriculum generation using LLMs.

The extension presented by this thesis consists of a wire system that functions similarly to "redstone dust" from Minecraft and logic gates, inspired by another video game Terraria (Re-Logic, 2011). These additional features allow for interactions in the environment that loosely simulate electrical engineering systems and provide for a wide range of possible mechanisms and goals from a concise set of materials. This allows for the creation of complex goals such as logic gate systems within the environment. Furthermore, this wiring system unlocks new aspects related to survival, for example, automated trapping of mobs.

This environment will be tested on goal complexity, goal interestingness, goal learnability, and computational expensiveness. These aspects of the environment will show if it is a suitable fast, open-ended environment for curriculum learning.

2 Methods

2.1 Craftax-wiring

We introduce Craftax-wiring, an extension of the Craftax-classic environment that incorporates a comprehensive wiring system for creating logic circuits. This extension draws inspiration from block-based wiring systems found in games like Minecraft's redstone and Terraria's wire mechanics, enabling agents to construct and interact with logic gates.

This extension adds four main categories of functionality to the base environment: (1) new blocks for circuit construction, including wires, logic gates, and input/output components; (2) additional actions for crafting, placing, and manipulating these circuit elements; (3) updated game mechanics for signal propagation and circuit evaluation; and (4) novel reward functions that require agents to construct and operate specific logic circuits, such as binary adders and combinational logic systems.

This extension transforms the environment from a purely survival-based challenge into a platform that combines spatial reasoning, logical thinking, and sequential planning—creating opportunities for curriculum learning approaches that can guide agents through increasingly complex circuit design tasks.

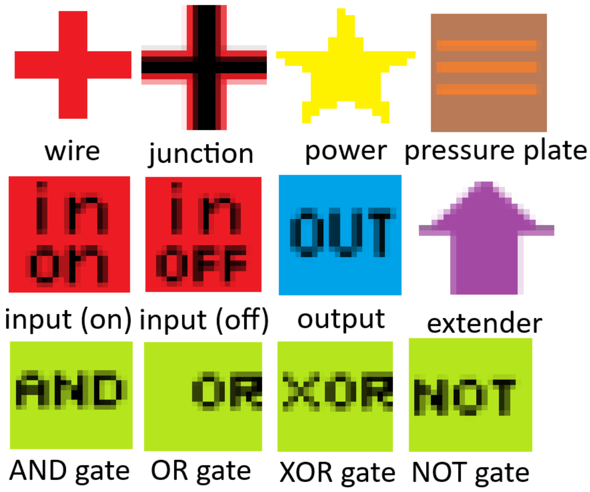

Figure 2.1: New blocks introduced in craftax-wiring extension, including wires, logic gates (AND, OR, XOR, NOT), power sources, and input/output blocks.

2.1.1 Blocks

All new blocks can be seen in Figure 2.1. These can all be crafted, placed and mined by the player character.

Power source blocks Power sources are blocks that initially power wires which can then in turn transmit that power to other blocks. Wires placed next to power source adopt $V_{max} = 10$. Power sources can be found in different forms in this environment including:

- Power blocks, these are simple blocks that always power neighboring blocks.
- Input blocks, these function similarly to levers in Minecraft which means they can be turned on (functions like a power block) and off (functions like a block) by the player character and are also used for the new reward functions (Section 2.1.4). This is so the input of logic gate circuits can be exactly measured.
- Pressure plates, these function as power blocks when the player or a mob is standing on the plate.

Wires The main function of these wires is to transmit integer signals (charge values 0-10) to neighboring blocks using a von Neumann neighborhood connectivity. Each wire carries a charge V_{wire}. The updating of the wires functions similar to a cellular automaton. (Details can be found in Appendix B) This was inspired by a 2D implementation of Minecrafts redstone as a cellular automata in typescript (Jackrekirby, 2024).

If a power source is found:

$$V_{wire} = V_{max} \tag{2.1}$$

If a power source is not found but a wire is found:

$$V_{wire}(t+1) = \max_{n \in N}(V_n(t)) - 1 \tag{2.2}$$

Where $V_{wire(t+1)}$ represents the wire's voltage at timestep t+1, N denotes the von Neumann neighborhood, and $V_n(t)$ is the voltage of neighbor n at timestep t.

This is to ensure a proper signal direction and to prevent looping signals between two wires. Furthermore, each wire is updated once per timestep and all wires are updated in parallel. The location of the wires is saved in the state and an updating function loops over all indices similar to how plants are updated in craftax-classic. Saving the locations and indices of all wire-related blocks is less computationally expensive that having an updating function that loops over all blocks in the map. This setup results in the signal travelling one block each step in the environment. These mechanics ensure that the signal propagation functions similarly to existing wiring systems while also retaining the computational speed of JAX based environments.

Junction The wire junction block functions as a crossroads that transmits signals in a straight line (instead of in a von Neumann neighborhood), which allows for the creation of more compact systems. They do not carry a charge, instead an adjacent wire adopts the charge of the wire on the other side of the junction. This also allows for faster signal transmission over longer distances.

Logic gates Logic gates are blocks that output a signal in the direction in which it was placed. For example, a logic gate placed while the player character is facing north will output a signal to the north. This output is based on the binary inputs received from the left and right wires (with the exception of the NOT gate). The inputs for the logic gates are wires with $V_{wire} > 0$. Wires placed in the output location adopt V_{max} if the condition of the logic gate has been met. Forms of logic gates include the following:

- AND
- OR
- XOR
- NOT

The NOT gate has a $V_{output} = V_{max}$ if $V_{input} = 0$, else $V_{output} = 0$ where V_{input} is the signal received from the opposite direction in which it was placed. These basic logic gates, serving as elementary functions, allow for the creation of complex logic systems (such as a binary adder, for example).

Finally, there are two output blocks:

- Output blocks, these are implemented like wires with the additional functionality of retrieving their signal converted to binary and are used for reward functions together with the input blocks (Section 2.1.4)
  $$f(V) = \begin{cases} 1 & \text{if } V > 0 \\ 0 & \text{otherwise} \end{cases}$$
- Extenders, these blocks allow for the system to interact with the environment by extending one block in the direction it was placed in which allows for the creation of doors or traps

2.1.2 Observation space

Similarly to Craftax-classic, Craftax-wiring allows for pixel-based and symbolic observations. The size of the pixel-based observation space remains unchanged which is a downscaled $63 \times 63 \times 3$ image. The symbolic observation space is a one-hot encoded representation of the block types and creatures in the player's visual area along with an array of the player's information such as health, inventory etc. resulting of a flat observation space of 2175 due to the addition of the blocks. The agent can only observe the existence of the new blocks and not the charge or other details.

Input A	Input B	SUM	CARRY
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Table 2.1: Half adder truth table

2.1.3 Actions

New actions were added to the environment that allow the crafting and placing of these blocks and the switching of the input block. This results in a larger action space of 40 as compared to the original 17 of craftax classic (43 in craftax). A full list of actions can be found in Appendix A.

2.1.4 Reward functions

The new reward functions for the RL agent consist of:

Logic gate systems The new reward functions regarding logic gate systems include:

- Half adder
- Full adder
- Binary to grey code converter

These reward functions retrieve the binary input and output values from the first n input and output blocks placed in the environment. These values are then compared to the truth table of the respective logic gate system (example for the half adder can be found in Table 2.1). When a match has been found with one of the rows in the truth table, that row is marked as achieved. This happens after the signal propagates from the input to the output which means the agent has to wait a small amount of time before this has been registered. When all rows have been marked as achieved, the agent is given the reward. This means that the agent has to build the circuit and then make the outputs match the inputs in all possible configurations.

The following is an example of the half adder reward function in pseudocode:

```
Function HALF_ADDER(state): //The state of the
    environment is used as an input and will
    be updated
    input_A <-- get input index 0 from state
    input_B <-- get input index 1 from state
    output_sum <-- get output index 0 from
        state
    output_carry <-- get output index 1 from
        state

    truth_table <-- state.
        truth_table_half_adder
    inputs <-- [input_A, input_B, output_sum,
        output_carry]

    real_truth_table <-- [
        [0, 0, 0, 0],
        [0, 1, 1, 0],
        [1, 0, 1, 0],
        [1, 1, 0, 1]
    ]

    matches <-- compare each row in
        real_truth_table with inputs
    updated_truth_table <-- logical OR of
        truth_table and matches
    full_table_complete <-- all entries in
        updated_truth_table are True

    achievements <-- copy of state.achievements
    achievements[HALF_ADDER] <-- achievements[
        HALF_ADDER] OR full_table_complete

    new_state <-- state with updated
        truth_table_half_adder and
        achievements

    return new_state
```

Interactive The new reward functions regarding environment interaction include:

- Trap
- Door

These reward functions are wiring systems that interact with the environment. The trap function rewards the agent when the extended part of an activated extender is found next to a pressure plate. This represents a simple trap that closes a "gate" near a mob or player that steps on the pressure plate which traps the respective mob or player. The door function rewards the agent when the location of the extended part of an inactive extender

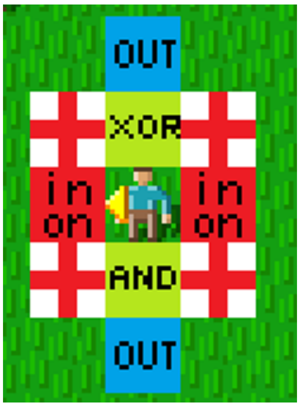

Figure 2.2: Simplest version of a succesful half adder

is next to a pressure plate while a NOT gate is powered. This represents a simple door that is activated by a pressure plate (which is similar to how doors are classically implemented by human players in Minecraft).

2.2 Experimental Setup

To find out if this environment is a suitable and computationally fast environment for curriculum learning purposes, several experiments are conducted.

Environmental setup The environment will be modified so that the agent can fully target the half adder task. This means several actions and achievements are disabled (See Appendix C for the full list of the actions and achievements used). Furthermore, the agent will start with all the necessary blocks needed to build the simplest version of the half adder (Figure 2.2).

2.2.1 Goal complexity

Firstly, it must be determined that the new reward functions are too difficult for a RL agent to achieve and learn without intermediate steps in this new environment, which would mean they are challenging enough for curriculum learning methods. This will be done with a JAX implementation of TransformerXL with PPO in a RL setup (transformerXL_PPO_JAX) (Hamon, 2024) which outperforms baseline methods on the Craftax benchmark. The default parameters will be used with 100 million timesteps (See Appendix D for a full list of the parameters). The results of this will show if the reward functions are achieved or not. The goal for this experiment is for the agent to create a half adder that performs a binary addition of two inputs into a sum and a carry output.

2.2.2 Goal interestingness

To evaluate how interesting such goals are, GPT-4.5 will be asked to score the new reward functions and the rewards from Craftax on several measurements which are also decided by the LLM. This method follows from the idea that foundation models contain human notions of interestingness (Faldor et al., 2024) and that asking a general LLM to rate the achievements would give a an automized result based on an average notion of interestingness compared to an individual evaluation. (See Appendix E.1 for the full prompt).

2.2.3 Goal learnability

A handcrafted, stage-based reward-shaping architecture for a reward function of the wiring extension was designed. This means that there are intermittend stages that reward the agent for completing subgoals that sequentially lead to the creation and activation of the half adder circuit. This is to see if the goal can be reached in the environment using intermittent rewards that encourage a natural learning progression.

The architecture rewards the agent for the intermediate stages which include:

- Stage 1: Placing two inputs
- Stage 2: Connecting two wires to each input
- Stage 3: Placing an XOR gate
- Stage 4: Activating an input
- Stage 5: Activating the XOR gate
- Stage 6: Placing an AND gate
- Stage 7: Placing two outputs next to the logic gates
- Stage 8-11: Checking values in truth table

We recognise that curriculum learning is complementary for achieving the goal of helping agents develop skills through gradually increasing complexity. However, for this initial research, we chose to implement a reward-shaping strategy for simplicity purposes. This decision was made to focus on identifying and assessing how well individual subgoals can be learned in the environment. This approach helps to validate the feasibility of tasks in a controlled manner, before implementing a more comprehensive curriculum framework.

2.2.4 Computational resources

Finally, the computational resources used for training transformerXL_PPO_JAX will be measured and compared to the training of the architecture on Craftax. Steps per second will be measured using different numbers of parallel environments running on the hardware. Three setups of environment workers will be measured: 1024, 2048, and 4096. This will show the scalability of the environment, how well the implementation leverages the hardware, and any inefficiencies in the implementation. This is to see whether the implementation of the extension is still computationally inexpensive. The training will be done on two Tesla V100 GPUs.

3 Results

3.1 Goal complexity

Without any reward-shaping, the transformer model was not able to learn any behavior that results in achieving the half adder goal. Figure 3.1 shows the result of the training. The average return remained zero throughout most of the training, with no upward trend which indicates no learning pattern. The small spike at 30 updates indicates that the agent was sometimes able to reach the goal, however, this spike can consistently be found in several runs and the cause for this remains unclear. Regardless, there is no evidence of the agent learning a suitable policy for building the half adder. This result highlights the necessity of reward shaping or curriculum learning for solving complex symbolic tasks with reinforcement learning.

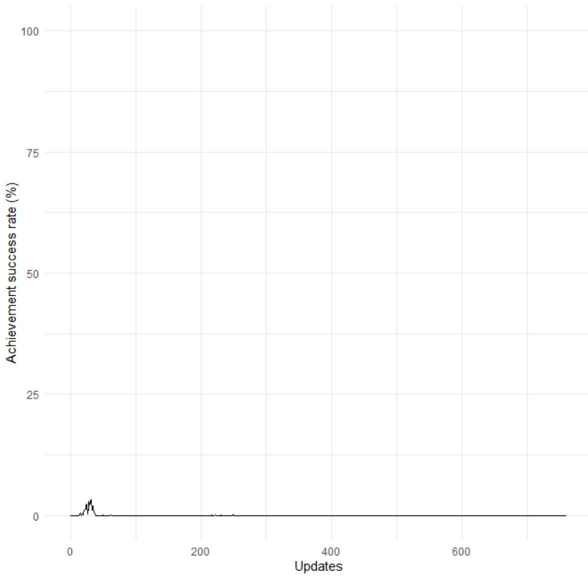

Figure 3.1: Half adder reward on craftax-wiring with transformer PPO run on 100 million timesteps without reward-shaping

3.2 Goal interestingness

The rewards are scored by the LLM on several aspects, chosen by the LLM itself:

- Complexity: How challenging is the logical or mechanical understanding required by the agent?
- Progression and Hierarchy: Does completing simpler achievements meaningfully lead to advanced achievements?
- Novelty and Uniqueness: Are the goals unique compared to typical RL environments, offering fresh challenges?
- Interaction Complexity: How many components or interactions must the agent manage simultaneously?
- Real-world or Educational Value: How well can the learned skills or concepts be generalized or related to real-world problems?

These aspects are scored on a 5 point scale. The scores can be found in Table 3.1. See Appendix E.2 for the full prompt and response.

The LLM provided the following conclusion: “The Craftax-classic environment incorporating

	Craftax	Craftax-wiring
Complexity	3	5
Progression and Hierarchy	4	4
Novelty and uniqueness	2	5
Interaction complexity	3	5
Real-world or Educational value	2	4
Overall score (out of 25)	14	23

Table 3.1: Interestingness scores given by GPT 4.5

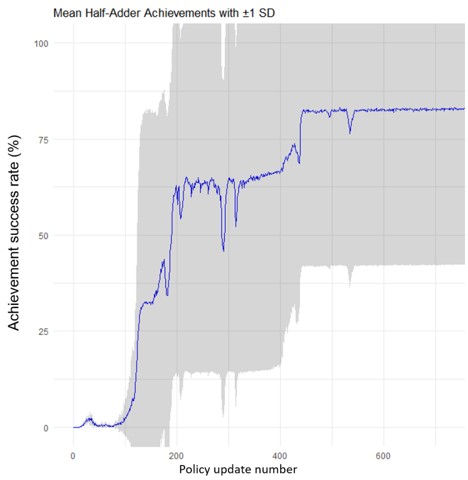

Figure 3.2: Half adder reward on craftax-wiring with transformer PPO run on 100 million timesteps with curriculum

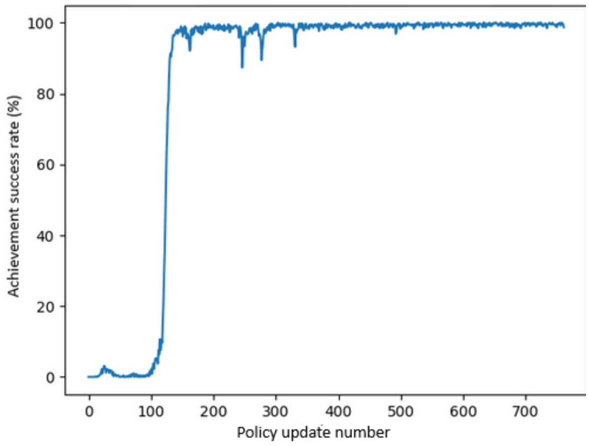

Figure 3.3: Best run on half adder reward with curriculum

Redstone wiring achievements is notably more interesting and valuable as an RL task compared to standard Craftax. It excels particularly in complexity, novelty, and educational value. This environment uniquely integrates spatial, logical, and computational reasoning challenges, providing richer, more stimulating learning scenarios for RL agents."

3.3 Goal learnability

Figure 3.2 shows the mean learning curve across 5 independent training runs over a 100 million timesteps equivalent in policy updates, with error bars indicating standard deviation. Figure 3.3 shows the best (largest area under curve) run out of these 5. In the first 100 updates, the agent learns to progress through the different stages that were established to allow for the learning of the creation and usage of the half adder. The plateaus indicates that the agent has the ability to successfully learn and reliably execute the half adder task. The high variance mostly occurs since out of the 5 runs, one run did not indicate a plateau and the return remained near 0 which indicated that the agent did not learn to build the half adder. Regardless, this shows that the agent can learn a complex task in this environment successfully.

3.4 Computational resources

Figure 3.4 shows that the wiring environment computes the same steps per second (SPS) as the craftax environment. Benchmarking done in the craftax paper indicate that craftax environment is 25% slower than craftax-classic which means craftax-wiring has the same SPS decrease. This environment is still orders of magnitude faster than crafter.

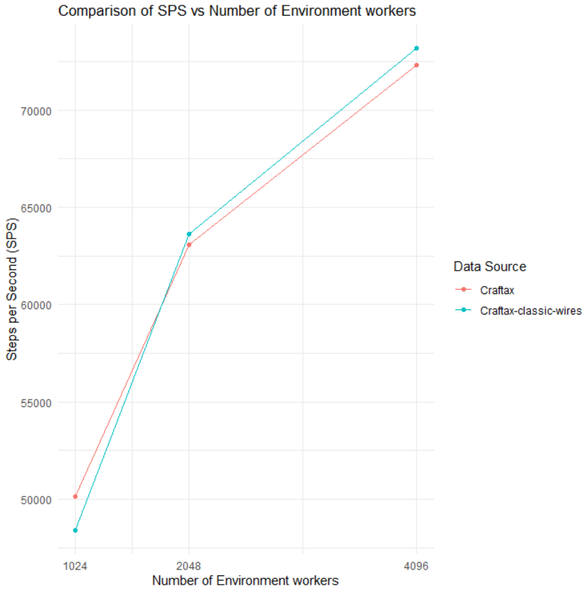

Figure 3.4: Steps per second for craftax and craftax-wiring

	Wiring	Classic
Complex goals	✓	X
Interesting goals	✓	X
Learnable goals (assisted)	✓	-
Computationally optimized	✓	✓+

Table 4.1: Overall comparison between craftax-wiring and craftax-classic

4 Discussion

4.1 Known issues

There are some known issues / bugs with craftax-wiring for which the reason remains unclear.

Stuck wires Sometimes the wires do not progress the signal correctly. More specifically, the wires do not succesfully obtain a charge from their neighbors and they remain chargeless. It seems to be somewhat related to the usage of the logic gates. Replacing the wires fixes the issue.

Stochasticity All rng should be deterministic based on the seed however two runs using the same parameters yield different results. This also occurs on craftax-classic which means this is an issue of the original environment. This results in problems regarding reproducibility of the research presented by this thesis.

Wiring observability The state of the wires, logic gates, and input and output blocks are not observable by the agent. If these charges, directions and active status were in the observation space, the agent could have a more comprehensive understanding of the circuits and could possibly find a more optimal policy or find the policy faster.

4.2 Future research

The purpose for the development of this environment is for future research. There are many things that can be done with this environment.

4.2.1 Curriculum learning and reward shaping

In this thesis, reward shaping was chosen over curriculum learning to focus on validating whether agents could learn specific subgoals in isolation. By assigning rewards to incremental stages (e.g., placing inputs, connecting wires, activating logic gates), we approximated a linear learning progression without implementing a full curriculum.

This staged approach proved effective: agents failed to learn the half-adder task without guidance but succeeded when intermediate rewards were given. While curriculum learning is better suited for general skill acquisition across tasks of increasing difficulty, reward shaping allowed controlled evaluation of learnability and environment mechanics.

Importantly, Craftax-wiring remains compatible with curriculum learning. The open-ended design, diverse goal structures, and logical progression of complexity make it ideal for future work on curriculum design.

4.2.2 LLM goal and curriculum design

An open-ended environment like craftax-wiring is optimal for goal designing by LLMs. It would be interesting to see a foundation model, similar to those used by OMNI-EPIC (Faldor et al., 2024), or an autotelic agent leveraging LLMs (Pourcel et al., 2024), create and code interesting goals inside craftax-wiring while also using a curriculum designing LLM similar to CurricuLLM (Ryu et al., 2024) to create and code a learning progression for an RL agent to learn the skills required for creating logic gate systems or other goals. Transfer learning could also be researched using agents trained to build simple circuits and applying that knowledge to novel circuits.

4.2.3 Minecraft in JAX

An even more advanced environment for similar purposes would be a reimplementation of Minecraft in JAX, which is basically a more complex 3-dimensional version of Crafter. It is an open-ended environment and a version in JAX would be an interesting benchmarking environment for curriculum learning in deep reinforcement learning. The more advanced redstone system in 3D with more blocks like comparators for example would allow for more interesting goals, for example: functional computers (mattbatwings, 2023) or neural networks (mattbatwings, 2024).

4.3 Conclusion

The experiments show that this new craftax-wiring environment has goals can not be solved with traditional RL, goals that are interesting, reward shaping is possible and helps the RL agent achieve the goal, and the environment is relatively computationally inexpensive. A comparison against Craftax-classic related to the experiments can be found in Table 4.1. From this we can derive that Craftax-wiring is a promising fast benchmarking environment for curriculum learning purposes. The new actions and blocks allow for a large number of combinational logic gate circuits. Notably, the modifications required to extend Craftax-classic were relatively minimal, yet they enabled a way broader and more open-ended range of tasks. To answer the research question of "Can we create a fast and open-ended environment for curriculum learning in deep-reinforcement learning?", the answer is yes and craftax-wiring is a promising environment for future research.

References

Faldor, M., Zhang, J., Cully, A., & Clune, J. (2024). Omni-epic: Open-endedness via models of human notions of interestingness with environments programmed in code. *arXiv preprint arXiv:2405.15568*.

Freeman, C. D., Frey, E., Raichuk, A., Girgin, S., Mordatch, I., & Bachem, O. (2021). Brax–a differentiable physics engine for large scale rigid body simulation. *arXiv preprint arXiv:2106.13281*.

Hafner, D. (2021). Benchmarking the spectrum of agent capabilities. *arXiv preprint arXiv:2109.06780*.

Hamon, G. (2024, July). *transformerXL_PPO_JAX.* Retrieved from `https://inria.hal.science/hal-04659863`

Jackrekirby. (2024). *minecraft-ca-ts.* `https://github.com/Jackrekirby/minecraft-ca-ts`. GitHub.

mattbatwings. (2023, March). *I made a working computer with just redstone!* `https://www.youtube.com/watch?v=CW9N6kGbu2I`. (YouTube video)

mattbatwings. (2024, May). *I made an ai with just redstone!* `https://www.youtube.com/watch?v=DQ0lCm0J3PM`. (YouTube video)

Matthews, M., Beukman, M., Ellis, B., Samvelyan, M., Jackson, M., Coward, S., & Foerster, J. (2024). Craftax: A lightning-fast benchmark for open-ended reinforcement learning. *arXiv preprint arXiv:2402.16801*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mojang. (2011). *Minecraft.* Retrieved from `https://www.minecraft.net`

Parker-Holder, J. (2022). *Towards truly open-ended reinforcement learning.* University of Oxford (United Kingdom).

Portelas, R., Colas, C., Weng, L., Hofmann, K., & Oudeyer, P.-Y. (2020). Automatic curriculum learning for deep rl: A short survey. *arXiv preprint arXiv:2003.04664*.

Pourcel, G., Carta, T., Kovač, G., & Oudeyer, P.-Y. (2024). Autotelic llm-based exploration for goal-conditioned rl. In *Intrinsically motivated open-ended learning workshop at neurips 2024*.

Re-Logic. (2011). *Terraria.* Video game. (Available at `https://terraria.org`)

Ryu, K., Liao, Q., Li, Z., Delgosha, P., Sreenath, K., & Mehr, N. (2024). Curricullm: Automatic task curricula design for learning complex robot skills using large language models. *arXiv preprint arXiv:2409.18382*.

Stanley, K. O., Lehman, J., & Soros, L. (2017). Open-endedness: The last grand challenge you've never heard of. *While open-endedness could be a force for discovering intelligence, it could also be a component of AI itself*.

A Full list of actions

```
NOOP
LEFT
RIGHT
UP
DOWN
DO
SLEEP
PLACE_STONE
PLACE_TABLE
PLACE_FURNACE
PLACE_PLANT
MAKE_WOOD_PICKAXE
MAKE_STONE_PICKAXE
MAKE_IRON_PICKAXE
MAKE_WOOD_SWORD
MAKE_STONE_SWORD
MAKE_IRON_SWORD
MAKE_WIRE
MAKE_POWER
MAKE_EXTENDER
MAKE_JUNCTION
MAKE_AND
MAKE_OR
MAKE_XOR
MAKE_NOT
MAKE_PRESSURE_PLATE
MAKE_INPUT
MAKE_OUTPUT
PLACE_WIRE
PLACE_POWER
PLACE_EXTENDER
PLACE_JUNCTION
PLACE_AND
PLACE_OR
PLACE_XOR
PLACE_NOT
PLACE_PRESSURE_PLATE
PLACE_INPUT
PLACE_OUTPUT
SWITCH_INPUT
```

B Updating of new blocks

B.1 Placing

When wires, logic gates, extenders, input and output blocks are placed in the environment, their position and other variables relevant to the respective block are saved in the state. This is the same as how plants are implemented in craftax-classic. Besides position, some blocks have the following properties.

Wires

- Charge (int)

Logic gates

- Type (int)
- Output direction (2D int vector)
- Power (bool)

Extenders

- Output direction (2D int vector)

B.2 Input and output value retrieval

When an input is switched on or off, the actual block changes. Therefore, when the value of a certain input, given the index, needs to be retrieved by a reward function for example, the position of the input with the corresponding index in the state map is found and we read the value of the block at that position.

When an output block is placed, a wire is also created in the state that corresponds with the output block. This results in a simple way of retrieving the output value: the position of the output which is saved in the state is found using the index. Then we find the wire on that position and if $V_{wire} > 0$ then the output value is True, otherwise it is False.

B.3 Updating

The following are pseudocode renditions of the updating functions of wires (output blocks function like wires), logic gates and extenders generated by ChatGPT 4o based on the functions used in the environment. The pseuodoce maps correctly to the actual code.

B.3.1 Wire charge

```
Define directions to check for neighboring
    tiles (up, down, left, right)

Function CHECK_MAX_CHARGE(carry, offset):
   wires_charge, nearby_charge, index <--
       carry
   pos <-- position of current wire + offset

   // Follow junctions in a straight line
       until junction ends
   While tile at pos is a JUNCTION:
      pos <-- pos + offset
      num_junctions += 1

   // Check if the current position is a valid
        power source
   is_power <-- tile at pos is POWER or
       INPUT_ON,
             OR it's a PRESSURE_PLATE being
                 stepped on by player or mob

   // Check if the current pos matches a logic
        gate input/output
   gate_found <-- does pos match any known
       logic gate?
   If gate_found:
      powered_gate <-- is that gate currently
          powered?
      gate_dir <-- its direction
      gate_pos <-- its position

      // Check if this wire is receiving
          output from the gate
      is_output <-- (gate_pos + gate_dir X (1
          + num_junctions)) == wire_position
          [index]
      receive_output <-- powered_gate AND
          is_output
   Else:
      receive_output <-- False

   // If any nearby source powers the wire,
       assign full charge
   If is_power OR receive_output:
      new_charge <-- max_wire_charge
   Else:
      new_charge <-- nearby_charge

   // Also check if this neighbor is a powered
        wire
   If pos is a wire and exists in wire list:
      neighbor_index <-- index of that wire
      neighbor_charge <-- wires_charge[
          neighbor_index] - 1 (decayed)
      new_charge <-- max(new_charge,
          neighbor_charge)

   // Update the charge value
   nearby_charge <-- max(nearby_charge,
       new_charge)
   wires_charge[index] <-- nearby_charge
   Return (wires_charge, nearby_charge, index)

Function CHECK_ONE_WIRE(wires_charge, index):
   nearby_charge <-- 0
   For each offset in neighboring_wire_area:
      (wires_charge, _, _) <--
          CHECK_MAX_CHARGE((wires_charge,
          nearby_charge, index), offset)
   Return wires_charge

// Update all wires in the world
For each wire index from 0 to 19:
   wires_charge <-- CHECK_ONE_WIRE(
       wires_charge, index)

// Save updated wire charges back into state
state.wires_charge <-- wires_charge
Return state
```

B.3.2 Logic gates

```
Define ROTATE_LEFT(direction):
   Return [-direction[1], direction[0]]

Define ROTATE_RIGHT(direction):
   Return [direction[1], -direction[0]]

Define INPUT_GATE(position, direction):
   pos_input_1 <-- position + ROTATE_LEFT(
       DIRECTIONS[direction])
   pos_input_2 <-- position + ROTATE_RIGHT(
       DIRECTIONS[direction])
   is_wire_1 <-- tile at pos_input_1 is WIRE
   is_wire_2 <-- tile at pos_input_2 is WIRE

   wire_index_1 <-- if is_wire_1, index of
       wire at pos_input_1, else max_wires +
       1
   wire_index_2 <-- if is_wire_2, index of
       wire at pos_input_2, else max_wires +
       1

   input_1 <-- wires_charge[wire_index_1] > 0
```

```
    input_2 <-- wires_charge[wire_index_2] > 0

    Return [input_1, input_2]

Define INPUT_NOT(position, direction):
    pos_input <-- position - DIRECTIONS[
        direction]
    is_wire <-- tile at pos_input is WIRE
    wire_index <-- if is_wire, index of wire at
         pos_input, else max_wires + 1
    input <-- wires_charge[wire_index] > 0

    Return [input, False]

Define UPDATE_NONE(input):
    Return False

Define UPDATE_AND(input):
    input_1, input_2 <-- input
    Return input_1 AND input_2

Define UPDATE_OR(input):
    input_1, input_2 <-- input
    Return input_1 OR input_2

Define UPDATE_XOR(input):
    input_1, input_2 <-- input
    Return input_1 XOR input_2

Define UPDATE_NOT(input):
    Return NOT input[0]

Define UPDATE_LOGIC_GATE(carry):
    power, index <-- carry
    type <-- logic_gates_type[index]
    position <-- logic_gates_positions[index]
    direction <-- logic_gates_direction[index]
    pos_output <-- position + DIRECTIONS[
        direction] // not used

    If type == 4:
        input <-- INPUT_NOT(position, direction
            )
    Else:
        input <-- INPUT_GATE(position,
            direction)

    update_functions <-- [UPDATE_NONE,
        UPDATE_AND, UPDATE_OR, UPDATE_XOR,
        UPDATE_NOT]
    output <-- update_functions[type](input)

    power[index] <-- output
    Return power
```

```
Define CHECK_ONE_LOGIC_GATE(logic_gates_power,
      index):
    new_logic_gates_power <-- UPDATE_LOGIC_GATE
        ((logic_gates_power, index))
    Return new_logic_gates_power

new_logic_gates_power <-- state.
     logic_gates_power

For index in 0 to 4:
    new_logic_gates_power <--
        CHECK_ONE_LOGIC_GATE(
        new_logic_gates_power, index)

state.logic_gates_power <--
     new_logic_gates_power
Return state
```

B.3.3 Extenders

```
neighboring_wire_area <-- [(-1, 0), (0, -1),
     (0, 1), (1, 0)]

Define NEARBY_WIRE(carry, loc_add):
    is_extending, index <-- carry
    pos <-- state.extenders_positions[index] +
        loc_add

    is_wire <-- tile at pos is WIRE
    wire_index <-- if is_wire, index of wire at
         pos, else max_wires + 1
    valid_index <-- wire_index >= 0 AND
        wire_index <= max_wire_charge

    extending <-- is_extending OR (wires_charge
        [wire_index] > 0)

    Return (extending, index)

Define CHECK_ONE_EXTENDER(new_map, index):
    extension_position <-- extenders_positions[
        index] + direction from
        extenders_direction[index]
    is_extending <-- False

    (is_extending, _), _ <-- scan NEARBY_WIRE
        over neighboring_wire_area

    placed_extender_block <-- if is_extending,
        EXTENSION block, else PATH block
```

```
    new_map[extension_position] <--
        placed_extender_block

    Return new_map

new_map <-- state.map

For index in 0 to max_extenders - 1:
    new_map <-- CHECK_ONE_EXTENDER(new_map,
        index)

state.map <-- new_map

Return state
```

C Experimental environment setup

C.1 Actions

```
NOOP
LEFT
RIGHT
UP
DOWN
DO
SLEEP
PLACE_TABLE
PLACE_WIRE
PLACE_JUNCTION
PLACE_AND
PLACE_XOR
PLACE_INPUT
PLACE_OUTPUT
SWITCH_INPUT
```

C.2 Achievements

```
PLACE_OUTPUT
PLACE_INPUT
PLACE_WIRE
ACTIVATE_INPUT
ACTIVATE_OUTPUT
ACTIVATE_LOGIC_GATE
TRAP
DOOR
HALF_ADDER
FULL_ADDER
BIN_TO_GRAY
```

D Training parameters

LR	2e-4
NUM_ENVS	1024
NUM_STEPS	128
TOTAL_TIMESTEPS	1e8
UPDATE_EPOCHS	4
NUM_MINIBATCHES	8
GAMMA	0.999
GAE_LAMBDA	0.8
CLIP_EPS	0.2
ENT_COEF	0.002
VF_COEF	0.5
MAX_GRAD_NORM	1
ACTIVATION	relu
ENV_NAME	craftax-wiring
ANNEAL_LR	True
qkv_features	256
EMBED_SIZE	256
num_heads	8
num_layers	2
hidden_layers	256
WINDOW_MEM	128
WINDOW_GRAD	64
gating	True
gating_bias	2
seed	randint(0, 10000000)

Table D.1: Parameters used in experimental setup

E Interestingness evaluation

E.1 LLM prompt

```
The following is a list of achievements from
    craftax:
class Achievement(Enum):
    COLLECT_WOOD = 0
    PLACE_TABLE = 1
    EAT_COW = 2
    COLLECT_SAPLING = 3
    COLLECT_DRINK = 4
    MAKE_WOOD_PICKAXE = 5
    MAKE_WOOD_SWORD = 6
    PLACE_PLANT = 7
    DEFEAT_ZOMBIE = 8
    COLLECT_STONE = 9
    PLACE_STONE = 10
    EAT_PLANT = 11
    DEFEAT_SKELETON = 12
    MAKE_STONE_PICKAXE = 13
    MAKE_STONE_SWORD = 14
    WAKE_UP = 15
    PLACE_FURNACE = 16
    COLLECT_COAL = 17
    COLLECT_IRON = 18
    COLLECT_DIAMOND = 19
    MAKE_IRON_PICKAXE = 20
    MAKE_IRON_SWORD = 21

    MAKE_ARROW = 22
    MAKE_TORCH = 23
    PLACE_TORCH = 24

    COLLECT_SAPPHIRE = 54
    COLLECT_RUBY = 59
    MAKE_DIAMOND_PICKAXE = 60
    MAKE_DIAMOND_SWORD = 25
    MAKE_IRON_ARMOUR = 26
    MAKE_DIAMOND_ARMOUR = 27

    ENTER_GNOMISH_MINES = 28
    ENTER_DUNGEON = 29
    ENTER_SEWERS = 30
    ENTER_VAULT = 31
    ENTER_TROLL_MINES = 32
    ENTER_FIRE_REALM = 33
    ENTER_ICE_REALM = 34
    ENTER_GRAVEYARD = 35

    DEFEAT_GNOME_WARRIOR = 36
    DEFEAT_GNOME_ARCHER = 37
    DEFEAT_ORC_SOLIDER = 38
```

```
    DEFEAT_ORC_MAGE = 39
    DEFEAT_LIZARD = 40
    DEFEAT_KOBOLD = 41
    DEFEAT_KNIGHT = 65
    DEFEAT_ARCHER = 66
    DEFEAT_TROLL = 42
    DEFEAT_DEEP_THING = 43
    DEFEAT_PIGMAN = 44
    DEFEAT_FIRE_ELEMENTAL = 45
    DEFEAT_FROST_TROLL = 46
    DEFEAT_ICE_ELEMENTAL = 47
    DAMAGE_NECROMANCER = 48
    DEFEAT_NECROMANCER = 49

    EAT_BAT = 50
    EAT_SNAIL = 51

    FIND_BOW = 52
    FIRE_BOW = 53

    LEARN_FIREBALL = 55
    CAST_FIREBALL = 56
    LEARN_ICEBALL = 57
    CAST_ICEBALL = 58

    OPEN_CHEST = 61
    DRINK_POTION = 62
    ENCHANT_SWORD = 63
    ENCHANT_ARMOUR = 64


INTERMEDIATE_ACHIEVEMENTS = [
    Achievement.COLLECT_SAPPHIRE.value,
    Achievement.COLLECT_RUBY.value,
    Achievement.MAKE_DIAMOND_PICKAXE.value,
    Achievement.MAKE_DIAMOND_SWORD.value,
    Achievement.MAKE_IRON_ARMOUR.value,
    Achievement.MAKE_DIAMOND_ARMOUR.value,
    Achievement.ENTER_GNOMISH_MINES.value,
    Achievement.ENTER_DUNGEON.value,
    Achievement.DEFEAT_GNOME_WARRIOR.value,
    Achievement.DEFEAT_GNOME_ARCHER.value,
    Achievement.DEFEAT_ORC_SOLIDER.value,
    Achievement.DEFEAT_ORC_MAGE.value,
    Achievement.EAT_BAT.value,
    Achievement.EAT_SNAIL.value,
    Achievement.FIND_BOW.value,
    Achievement.FIRE_BOW.value,
    Achievement.OPEN_CHEST.value,
    Achievement.DRINK_POTION.value,
]


VERY_ADVANCED_ACHIEVEMENTS = [
    Achievement.ENTER_FIRE_REALM.value,
    Achievement.ENTER_ICE_REALM.value,
    Achievement.ENTER_GRAVEYARD.value,
    Achievement.DEFEAT_PIGMAN.value,
    Achievement.DEFEAT_FIRE_ELEMENTAL.value,
    Achievement.DEFEAT_FROST_TROLL.value,
    Achievement.DEFEAT_ICE_ELEMENTAL.value,
    Achievement.DAMAGE_NECROMANCER.value,
    Achievement.DEFEAT_NECROMANCER.value,
]

The following is a list of achievements from
    craftax-classic with redstone wiring:
    MAKE_WIRE = 22
    PLACE_WIRE = 23
    MAKE_INPUT = 24
    PLACE_INPUT = 25
    MAKE_OUTPUT = 26
    PLACE_OUTPUT = 27
    ACTIVATE_INPUT = 28
    ACTIVATE_OUTPUT = 29
    ACTIVATE_LOGIC_GATE = 30
    HALF_ADDER = 31
    FULL_ADDER = 32
    BIN_TO_GRAY = 33
    TRAP = 34
    DOOR = 35
INTERMEDIATE_ACHIEVEMENTS = [
    Achievement.ACTIVATE_OUTPUT.value,
    Achievement.ACTIVATE_LOGIC_GATE.value,
    Achievement.TRAP.value,
]

VERY_ADVANCED_ACHIEVEMENTS = [
    Achievement.DOOR.value,
    Achievement.HALF_ADDER.value,
    Achievement.FULL_ADDER.value,
    Achievement.BIN_TO_GRAY.value,
]
I want you to evaluate the overall
    interestingness of these goals for a
    reinforcement learning agent on several
    variables which you can decide. Compare
    standard craftax to craftax-classic with
    redstone. Give a score for each variable
    and present a conclusion
```

E.2 LLM response

```
Here's a structured evaluation of the
    interestingness of goals for reinforcement
     learning agents, comparing standard
```

Craftax with Craftax-classic with Redstone wiring:
Variables for Evaluating Interestingness:

I suggest the following variables for assessing the interestingness of these RL goals:

Complexity:
How challenging is the logical or mechanical understanding required by the agent?

Progression and Hierarchy:
Does completing simpler achievements meaningfully lead to advanced achievements?

Novelty and Uniqueness:
Are the goals unique compared to typical RL environments, offering fresh challenges?

Interaction Complexity:
How many components or interactions must the agent manage simultaneously?

Real-world or Educational Value:
How well can the learned skills or concepts be generalized or related to real-world problems?

Each variable will receive a score from 1 (low) to 5 (high).
Evaluation of Standard Craftax:

1. Complexity: 3/5

Goals involve crafting, exploration, and combat, requiring a moderate understanding of item progression and basic environmental interaction.

2. Progression and Hierarchy: 4/5

Clear hierarchical structure: simple tasks (collect wood) progress logically to advanced tasks (craft diamond equipment, defeat bosses).

3. Novelty and Uniqueness: 2/5

Common survival-crafting elements present in various gaming environments.

4. Interaction Complexity: 3/5

Agent interaction with environment and enemies is straightforward, with some moderate resource and inventory management.

5. Real-world or Educational Value: 2/5

Primarily game-based goals; limited direct real-world applicability outside strategic and survival decision-making.

Overall Interestingness Score (Standard Craftax): 14/25
Evaluation of Craftax-classic with Redstone Wiring:

1. Complexity: 5/5

Redstone wiring involves logical circuits, gates, and arithmetic (half-adder/full-adder), significantly raising the complexity and cognitive demands.

2. Progression and Hierarchy: 4/5

Clear logical and technological progression from basic wiring to advanced circuits (e.g., trap, doors, adders).

3. Novelty and Uniqueness: 5/5

Highly unique for an RL environment; combining logic circuit understanding and spatial reasoning is rare in standard RL tasks.

4. Interaction Complexity: 5/5

Multiple interconnected components; agent must manage input-output relations, spatial configurations, and signal flow simultaneously.

5. Real-world or Educational Value: 4/5

Strong educational relevance in logic, computer engineering principles (boolean algebra, circuit logic), and foundational computation concepts.

```
Overall Interestingness Score (Craftax-classic
     with Redstone): 23/25
Conclusion:

The Craftax-classic environment incorporating
    Redstone wiring achievements is notably
    more interesting and valuable as an RL
    task compared to standard Craftax. It
    excels particularly in complexity, novelty
    , and educational value. This environment
    uniquely integrates spatial, logical, and
    computational reasoning challenges,
    providing richer, more stimulating
    learning scenarios for RL agents.
```