Model Learning to Solve Minecraft Tasks

Yarin Benyamin, Argaman Mordoch, Shahaf Shperberg, Roni Stern

Ben Gurion University in Be'er Sheva, Israel
{bnyamin,mordocha}@post.bgu.ac.il, {shperbsh,sternron}@bgu.ac.il

Abstract

Minecraft is a sandbox game that offers a rich and complex environment for AI research. Its design allows for defining diverse tasks and challenges for AI agents, such as gathering resources and crafting items. Previous works have applied both Reinforcement Learning (RL) and Automated Planning methods to accomplish different tasks in Minecraft. RL methods usually require a large number of interactions with the environment, while planning methods requires a model of the domain to be available. Creating planning domain models for Minecraft tasks is arduous. Algorithms for learning a domain model from observations exist, yet have mostly been used on planning benchmarks. In this work, we explore the use of such algorithms for solving Minecraft tasks. We focus on the task of crafting a wooden pogo stick and explore different ways to represent states in this domain. Then, propose an agent that learns domain models from observations — either generated by an expert or collected online — and uses them with an off-the-shelf domain-independent planner.

Introduction

Minecraft is a widely popular sandbox game that offers a rich and complex environment for AI research. Its design allows for defining different tasks for Artificial Intelligence (AI) agents to perform, such as gathering resources and crafting items. Building AI agents that can play Minecraft and accomplish such tasks has been acknowledged as a major AI challenge and have received significant interest in the academic community. This includes an annual competition in NeurIPS (Guss et al. 2019a) and a dedicated game mod and framework for evaluating AI agents (Goss et al. 2023).

The state-of-the-art approach to building AI agents for solving Minecraft tasks is by applying Reinforcement Learning (RL) methods (Tessler et al. 2017; Frazier and Riedl 2019; Scheller, Schraner, and Vogel 2020). When using these methods, the AI agent learns to make decisions by interacting with an environment and receiving feedback through states and rewards (or penalties). The disadvantage of applying RL methods is that the learning stage requires heavy computations, many training hours, and many environmental interactions. The latter disadvantage is significant when designing agents for video games in general, and Minecraft in particular, since interacting with the environment requires initializing and performing actions in the game environment, which is often intentionally slowed down to allow humans to play it.

An alternative to RL used in building automated agents for video games is *Automated Planning* (Ghallab, Nau, and Traverso 2004). Automated planning algorithms use a *domain model* that specifies the dynamics of the world to determine which actions to perform to achieve a given set of goals. Planning technology has been used successfully to build AI agents for various video games (Duarte et al. 2020). For example, Bartheye and Jacopin (2009) used real-time planning methods to solve tasks in the Iceblox Virtual Battle Space 2 game. Planning has also been explicitly used in Minecraft. For example, Wichlacz et al. (2019) used classical and hierarchical planning models to solve construction tasks in Minecraft. One of the advantages of using planning is that it does not require interacting with the environment to decide which actions to perform to achieve desired goals. Additionally, due to the symbolic nature of most planning algorithms, the resulting plans for the planning tasks are more explainable than the policies generated by RL agents (Hoffmann and Magazzeni 2019).

A significant limitation of automated planning is that it requires a domain model, which includes defining how to represent states in the domain, what actions are available to the agent, and the action model of these actions, i.e., their preconditions and effects. Each of these modeling tasks can be difficult even for human experts. Prior work proposed automated methods for learning from observations state representations (Konidaris, Kaelbling, and Lozano-Perez 2018) and action models (Juba, Le, and Stern 2021; Wang 1994; Aineto, Celorrio, and Onaindia 2019). While action model learning methods have shown some promise, they have rarely been used in domains that are not standard automated planning benchmarks.

This work explores how automated planning can be used to build a Minecraft-playing agent. We explore different representations of states and actions in the domain. Unlike prior work on planning-based Minecraft agents, we do not assume that a Minecraft action model is given and instead learn it from observations. Specifically, we use Numeric Safe Action Model Learning (N-SAM) (Argaman Mordoch 2023), a state-of-the-art action model learning algorithm, to learn a numeric domain model from observations. Then, we provide the learned domain model to an off-the-shelf domain-

independent numeric planner, namely ENHSP (Scala et al. 2016), to select which actions our Minecraft-playing agent should perform. This agent is designed to work in two settings: offline and online. In the *offline* setting, it receives observations of an expert acting in the domain, and then it has to generate a plan to perform some Minecraft task. In the *online* setting, no observations are given apriori. The agent must choose how to interact with the environment to learn how to solve the desired Minecraft task.

We evaluated our agent in both settings in Polycraft, a symbolic wrapper to Minecraft, and compared it with multiple RL-based agents. We considered three symbolic representations of the domain and evaluated our agents in both offline and online settings. Our results show that in the online setting, the N-SAM-based agents outperform all baseline agents. We also show that our N-SAM planning agent is the only one to solve all the test set problems while the other RL agents are too overfitted towards their training data and cannot solve any test set problems.

Background and Related Work

In this section, we present the background and the previous works related to this work.

Minecraft Learning Environments

MineRL (Guss et al. 2019b) is an OpenAI-Gym (Brockman et al. 2016) compatible research environment that provides a Minecraft-based platform for the development, testing, and evaluation of RL algorithms. It offers various tasks and challenges commonly present in open-world games, such as navigation, resource gathering, and combat. It's large imitation learning datasets that contain more than 60 million frames of human player data enabling researchers to train and assess their algorithms efficiently. As a result, MineRL is a significant resource for the RL research community. In this work, we do not consider a visual, pixel-based, representation, and thus we did not use the MineRL environment.

Instead, we used the Polycraft (Palucka 2017) Minecraft mod Polycraft World. Polycraft provides an interface to Minecraft, as part of the Polycraft World AI Lab (PAL) (Goss et al. 2023)[1]. PAL allows AI agents to easily interact with Minecraft's environment by sending commands to the API and awaiting a response. Each command has predefined preconditions, effects, and costs. This mechanism enables RL algorithms to use the API to train their agents and easily solve various tasks. Compared to MineRL, PAL supports *symbolic* observations, which is best suitable for planning algorithms since they require a symbolic model of the environment to solve problems.

Reinforcement Learning Algorithms

RL (Sutton and Barto 2018) is a field of AI in which agents learn to make decisions by interacting with an environment and receiving feedback in the form of rewards (or penalties). RL and video games often go hand in hand (Justesen et al. 2019), as many games provide rewards for successful strategies. A prominent example of an RL algorithm is DQN (Mnih et al. 2013, 2015). DQN is an implementation of Q-Learning (Watkins 1989; Watkins and Dayan 1992) that uses deep neural networks to solve RL problems when the state-space is large. DQN agents can solve multiple Atari games, and the algorithm was the first deep-learning model to learn control policies from high-dimensional sensory input. This method outperformed all previous approaches and even surpassed human experts in some experiments. DQN is more suitable for large domains, and in preliminary experiments on our domain, it did not work well. Thus, we do not report results for DQN in this work.

However, open-world games present a complex environment for RL algorithms due to their lack of a defined reward structure, extensive exploration opportunities, and extensive player autonomy. Imitation Learning (IL) (Pomerleau 1991) offers a sound methodology to address these open-world challenges. In IL, the agent acquires a policy by observing expert demonstrations, also known as expert trajectories. The objective is to learn a policy that effectively mirrors the expert's performance in the game environment. The simplest form of IL is Behavioral Cloning (BC) (Bratko, Urbančič, and Sammut 1995), which focuses on learning the expert's policy using supervised learning. Inverse reinforcement learning (IRL) (Abbeel and Ng 2004) refers to learning the agent's objectives, values, or rewards by observing its behavior. The agent aims to find a reward function from the expert's demonstrations that explain the expert behavior. Generative Adversarial Imitation Learning (GAIL) (Ho and Ermon 2016) learns a policy by simultaneously training it with a discriminator that aims to distinguish expert observations against the observations from the learned policy.

Proximal Policy Optimization (PPO) (Schulman et al. 2017) is presently considered state-of-the-art in RL. The algorithm, introduced by OpenAI in 2017, alternates between sampling data through interaction with the environment and optimizing a "surrogate" objective function using stochastic gradient descent. PPO outperforms other online policy gradient methods This work focuses on the following algorithms: Q-Learning, BC, GAIL, and PPO.

The most widely adopted method for playing Minecraft is the Hierarchical Deep Reinforcement Learning Network (H-DRLN) (Tessler et al. 2017). This approach enables the agent to continuously learn multiple policies and adapt to new challenges within the game. The H-DRLN leverages a deep neural network to model the policy and value functions, resulting in high effectiveness across a variety of Minecraft tasks such as navigation, mining, and combat. Despite its success, this approach requires intensive training time and a less restrictive environment for it to be successful. Since we limit the learning time and restrict the agents to specific interactions with the environment, we determined that this approach is not appropriate for the scope of this research.

Planning in Minecraft

Planning is a well-studied field in AI that involves determining the sequence of actions an agent can take to achieve a specific goal. Planning algorithms are used in various AI applications, including robotics, natural language processing, and computer games, enabling agents to make decisions and

[1]https://github.com/PolycraftWorld/PAL

act in complex environments.

Formal planning languages express the "physics" of the world, i.e., what predicates define the state of the world, the possible actions for the agents, and the actions' preconditions and effects. The Planning Domain Definition Language (**PDDL**) (Aeronautiques et al. 1998) is a formal planning language and an extension of STRIPS (Fikes and Nilsson 1971) that includes conditional effects, universal quantification, domain axioms, and even object equality. Wichlacz et al. (Wichlacz, Torralba, and Hoffmann 2019) used PDDL modeling to solve complex construction tasks in Minecraft. The researchers modeled the house-construction task as classical and Hierarchical Task Network (HTN) (Georgievski and Aiello 2015) planning problems. They observed that even simple tasks present difficulties to current planners as the size of the world increases. The HTN planner scaled well when the size of the world increased but was too coupled with the specific task.

Action Model Learning

We focus on planning problems in domains where action outcomes are deterministic, states are fully observable, and the states are described with discrete and continuous state variables. Such problems are commonly modeled using the PDDL2.1 (Fox and Long 2003) language. We introduce the following notation to define a numeric planning problem in PDDL2.1. A domain is defined by a tuple $D = \langle F, X, A \rangle$ where F is a finite set of Boolean variables, X is a set of numeric variables (fluents), and A is a set of actions. A state is an assignment of values to all variables in $F \cup X$. For a state variable $v \in F \cup X$, we denote by $s(v)$ the value assigned to v in state s. Every action $a \in A$ is defined by a tuple $\langle \mathit{name}(a), \mathit{pre}(a), \mathit{eff}(a) \rangle$ representing the action's name, preconditions, and effects, respectively. The preconditions of action a are a set of assignments over (possibly a subset of) the Boolean variables and a set of conditions over (possibly a subset of) the numeric variables. These conditions are of the form (ξ, Rel, k) where ξ is an arithmetic expression over X, $Rel \in \{\leq, <, =, >, \geq\}$, and k is a number. The effects of action a denoted $\mathit{eff}(a)$, are a set of assignments over F and X, representing how the state changes after applying a. An assignment over a Boolean variable is either True or False. An assignment over a numeric variable $x \in X$ is a tuple of the form $\langle x, op, \xi \rangle$ where ξ is a numeric expression over X and op is either increase ("+="), decrease ("-="), or assign (":="). The set of actions with their definitions is referred to as the *action model* of the domain. We say that an action a is *applicable* in a state s if s satisfies $\mathit{pre}(a)$. Applying a in s, denoted $a(s)$, results in a state that differs from s only according to the assignments in $\mathit{eff}(a)$.

A planning problem is defined by $\langle D, s_0, G \rangle$ where D is a domain, s_0 is the initial state, and G are the problem goals. The problem goals G are assignments of values to a subset of the Boolean variables and a set of conditions over the numeric variables. A solution to a planning problem is a *plan*, i.e., a sequence of actions applicable in s_0 and resulting in a state s_G in which G is satisfied.

The Numeric Safe Action Model Learning (N-SAM) (Argaman Mordoch 2023) algorithm learns an action model that includes all actions observed in the given trajectories $\mathcal{T}$. First, it uses Safe Action Model Learning (SAM) learning (Juba, Le, and Stern 2021) to learn every observed action's Boolean preconditions and effects. Then, it creates numeric preconditions for every observed action a by constructing a convex hull over the relevant numeric variable values observed in states before a was applied. Finally, it creates numeric effects by solving a linear regression problem for every numeric variable that is part of the effects of that action. N-SAM has several attractive properties. First, it runs in time that is polynomial in the input data. Second, it is guaranteed to return an action model that is *safe*, i.e., the plans generated with it will also work on the real, unknown, domain model.

Problem Definition

Figure 1: A plan to accomplish the Craft Wooden Pogo task.

This work focuses on solving the Craft Wooden Pogo task, as defined in the PAL Minecraft environment. (Goss et al. 2023). In this task, the Minecraft agent (often called Steve) is located in a field comprising 30×30 blocks and surrounded by unbreakable bedrock walls. The field includes five trees and one crafting table placed in arbitrary locations in the environment. Steve is tasked with crafting a pogo stick, which requires performing the following actions:

1. Harvest at least three wood blocks from trees.
2. Use the wood to craft planks.
3. Use planks to craft sticks.
4. Use some of the sticks and planks to craft a tree tap and place the tree tap near a tree to collect rubber.

5. Use the remaining sticks, planks, and rubber to craft a wooden Pogo stick.

This sequence of actions is illustrated in Figure 1. The PAL environment supplies five different scenarios to solve this task.

We consider the problem of accomplishing the Craft Wooden Pogo task in two settings: *offline with expert demonstrations* and *online*.

Offline Learning from Expert Demonstrations

In the *offline with expert demonstrations* setting, our agent is given a set of *trajectories* created by observing an *expert* solving the Craft Wooden Pogo task, i.e., successfully crafting the wooden pogo stick. The PAL environment includes a hard-coded implementation of an expert planning-based agent that relies on a pre-defined PDDL domain model.[2]

The agent objective in this setting is to use the set of expert trajectories and output a *plan* or a *policy* specifying how to act in order to accomplish the Craft Wooden Pogo task. Both learning and planning in this setting are done *offline*, that is, after processing the expert trajectories and outputting a plan or a policy, the agent executes the actions in the plan or the policy until either the task is accomplished or not. Thus, this setting corresponds to the Offline RL setting (Sutton and Barto 2018) and Imitation Learning (Bratko, Urbančič, and Sammut 1995). The main measure we consider in this setting is how many expert trajectories our agent needs until it can output a successful plan or policy.

Online Learning

In the *online* setting, our agent no longer receives expert trajectories as its input. Instead, it must perform actions in the environment, actively exploring the environment and aiming to accomplish the task of crafting a wooden Pogo stick. Specifically, in the online setting the agent is allowed to interact with the environment in a sequence of *episodes*. Every episode starts from some initial state of the environment and ends after either the agent crafted the wooden pogo stick or the agent performed more than 64 actions.

A key difference between this setting and the previous one is that here planning and learning are interleaved: in every episode, the agent needs to plan which actions to perform, and then learn how to perform better based on the outcomes of performing these actions. This setting corresponds to the standard RL setting. The main measure we consider in this setting is how many episodes an agent performs until it can output a successful plan or policy.

Modeling the Pogostick Task in Minecraft

Efficient modeling and knowledge representation are key to solving hard learning and planning tasks. In this work, we explored three alternative approaches to model the Craft Wooden Pogo task.

[2]In our experiments, we implemented our own expert agent, manually encoding our own PDDL domain for different modeling of the Craft Wooden Pogo task.

- **All Blocks.** In this modeling, the field is represented as a 30×30 grid where each grid cell corresponds to a single block in the field and defines the content of that block.
- **Relevant Blocks.** This modeling, which is significantly smaller than All Blocks, ignores all blocks in the field that are not relevant for constructing the wooden pogo stick.
- **Item Counts.** This modeling is even more compact than Relevant Blocks, and completely ignores the location of different items in the field, storing only the number of blocks of each item type that currently exists in the field.

All our modeling approaches mask some internal mechanisms of PAL. Specifically, we do not store the wall blocks and ignore the agent's orientation. We also made collecting ingredients transparent to the agent, e.g., logs are automatically added to the inventory once a break action is executed in front of a tree. In addition, to avoid navigation computations we utilize PAL actions to directly teleport to any desired reachable block. Next, we describe in detail the differences between the proposed modeling approaches in detail, specifying how a state is defined and what actions the agent can perform.

All Blocks and Relevant Blocks

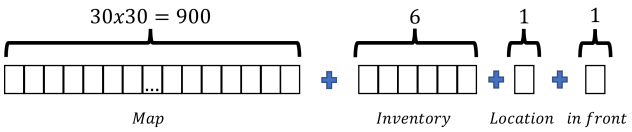

Figure 2: The vector representing the All Blocks modeling of a state in the environment, including the map dimensions. The agent is aware of the map and its own location in addition to its knowledge about the number of resources.

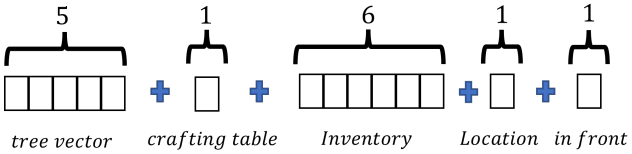

Figure 3: The vector representing the Relevant Blocks modeling of a state in the environment. The agent receives a smaller representation of the map and the information about its inventory.

In the All Blocks modeling, the agent is aware of the map's size and the trees' locations. A state corresponds to a 30×30 grid, as described above, a 6-array vector specifying the content of the agent's inventory, and an extra two state variables specifying the location of the agent and the content of the block in front of the agent. Figure 2 illustrates this state representation. In the Relevant Blocks modeling, we ignore all blocks in the field except those that contain a tree or the crafting table. That is, we consider a reduced version of the field in which there are only six cells, one for

each tree and one for the crafting table. The crafting table can change its location as well as the trees on the map. Figure 3 illustrates how a state in the Relevant Blocks modeling is represented. This modeling is, of course, significantly smaller than All Blocks, but it is also more specific as it assumes at most 5 trees in the map.

The actions the agent can perform in the All Blocks and the Relevant Blocks modeling are:

1. **TP_TO** - teleport from the current location to another cell on the map.
2. **BREAK**- breaks a tree to extract and add the logs to the inventory.
3. **CRAFT_PLANK** - craft planks from the logs in the inventory.
4. **CRAFT_STICK** - craft sticks from the planks in the inventory.
5. **CRAFT_TREE_TAP** - teleport to the crafting table, craft one tree tap, and add it to the inventory.
6. **PLACE_TREE_TAP** - when in front of a tree, move left, place the tree tap on it, collect the polyisoprene sack, and add it to the inventory.
7. **CRAFT_WOODEN_POGO** - teleport to the crafting table, craft a wooden pogo stick, and add it to the inventory.

Item Counts

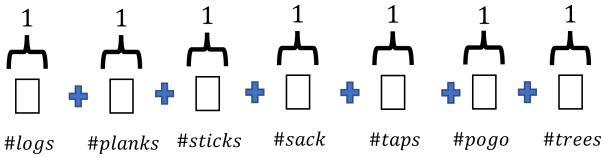

Figure 4: The vector representing the Item Counts modeling of a state in the environment. Each cell represents the number of resources of each type.

The Item Counts modeling of the environment relies on the understanding that the agent does not need to differentiate between the different trees in the domain. This allows us to define higher level *macro actions* for the agent to choose from, and correspondingly define a more compact state representation. Macro actions (also known as *options* (Sutton, Precup, and Singh 1999)) allow agents to optimize their gameplay and reduce the amount of time spent on repetitive tasks. It allows the agent to make more strategic decisions and react quickly to changing circumstances. OpenAI's use of macro actions in Starcraft (Vinyals et al. 2019) is an excellent example of how this approach can lead to more competitive and engaging gameplay.

In our context, we define the following macro actions that encapsulate multiple lower-level PAL actions:

1. **GET_LOG** - executes teleportation to a tree, breaking it, collecting the logs, add them to the inventory.
2. **CRAFT_PLANK** - craft planks from the logs in the inventory.
3. **CRAFT_STICK** - craft sticks from the planks in the inventory.
4. **CRAFT_TREE_TAP** - teleport to the crafting table, craft one tree tap, and add it to the inventory.
5. **PLACE_TREE_TAP** - teleport to a tree, move left, place the tree tap on it, collect the sack of polyisoprene, and add it to the inventory.
6. **CRAFT_WOODEN_POGO** - teleport to the crafting table, craft a wooden Pogo stick and add it to the inventory.

Problem analysis: the Item Counts map branching factor is 6 and state space is 11^6.

When using these macro actions, the agents do not need to know about the dimensionality of the map and to the placement of the trees. The agents are only aware of the quantity of each resource available in their inventory and the map itself. Thus, in the Item Counts modeling a state is composed of a (7×1) vector, representing the number of resources of each type, as well as the number of trees available in the map. Figure 4 illustrates a state in this modeling. In the initial state, five trees are available on the map, and the rest of the resources are all set to zeros. An example of a state during the agents' execution can be the vector $[1, 4, 0, 0, 0, 0, 3]$ that represents the state in which there is one log, and four planks in the inventory, three trees available in the map, and the rest of the ingredients are all zeros. We note that if an agent cuts all five trees in the map before using one to place a tree tap on it, then the problem becomes unsolvable.

Solving the Craft Wooden Pogo Task

Given the chosen modeling approach, we now describe two main approaches for the agent's decision-making process: based on RL techniques or based on domain model learning and planning techniques. We describe each of these approaches for offline learning from the expert observation setting and for the online learning setting.

Offline Learning from Expert Observations

The offline learning from expert observations setting can be viewed as an Offline RL problem (Kumar et al. 2020). Thus, one way to solve it is by using appropriate offline RL algorithms such as Q learning.[3] A different RL-based approach is to view our offline setting as an Imitation Learning problem and solve it with appropriate RL imitation learning algorithms such as BC and GAIL.

For the planning-based approach, we propose to provide the expert observations as input trajectories to the domain model learning algorithm N-SAM. The output of N-SAM is a PDDL domain model representing the environment. This PDDL domain is used as input to an off-the-shelf domain-independent numeric planner, which solves the resulting planning problem and outputs a plan for crafting the wooden pogo task. While any domain-independent numeric planner can be used to solve the resulting planning problem, we used in this work ENHSP (Scala et al. 2016), which is a state-of-the-art numeric planner.

[3] Q learning can also be applied in the online setting when coupled with an action selection method such as ϵ-greedy.

Note that N-SAM was designed to only return safe domain models, and avoids actions that were not sufficiently learned to guarantee safety. This is useful in domains where the safety requirement is needed but comes at the cost of slower learning. Since we do not need this requirement in our context and the number of expert observations may be limited, we modified N-SAM slightly to allow the agent to use unsafe actions.

In more detail, if there are n numeric state variables relevant to a certain action a, N-SAM requires that there are at least $n + 1$ linearly independent observations of that action. These independent observations are then used to compute a single solution to the set of polynomial equations that compose the action's effects. Otherwise, the action is considered to be unsafe and is not learned. We removed this requirement and allowed the linear regression algorithm to learn actions' effects even when the input data is insufficient to learn a safe domain model.

Online

Our online learning setting is the classical RL setting: an agent performs actions in the environment, receives observations, and adapts its behavior accordingly over time. Thus, we can use any off-the-shelf RL algorithm to solve it. In our experiments, we used PPO and Q learning. Since the rewards in our domain are only received at the end of an episode, we modified Q learning to only perform Bellman updates after it completed an episode.[4]

For the planning-based approach, we required an online algorithm to select which actions to perform when collecting trajectories. Here, we propose a hybrid of RL and planning approach in which an online RL algorithm is used to choose actions and generate trajectories, and N-SAM is used to learn a domain model from these set of trajectories. At some stage, N-SAM will have received enough observations to learn a sufficiently useful domain model that enables the planner to find a plan to accomplish the Craft Wooden Pogo task. In this work, we allow our planning-based agent to continue with the exploration process to try and improve the resulting domain model until it is halted externally. Future work may explore choosing to halt the exploration process earlier if cumulative regret considerations are relevant.

Note that since the agent in the online setting is exploring the environment while executing actions, it may attempt to perform an action in a state where it cannot. For example, attempting to break a tree when the agent is not near a tree block. This inconsistency is not supported by N-SAM, which is designed to learn from only valid trajectories. To address such cases, we assume that the result of applying an action in such a non-valid state is nothing, i.e., no change occurs to the state. This assumption is valid in Minecraft. Based on this assumption, we modified N-SAM to only consider transitions in which there is a difference between the pre-and post-states. This approach might be restrictive and there might be cases in which an action was correctly performed and still, the post-state has not changed.

[4]We also experimented with Vanilla Q learning without this enhancement and observed it performed poorly.

Experimental Results

In this section, we evaluated the RL-based approaches and the planning-based approach over the three proposed models — All Blocks, Relevant Blocks, and Item Counts— and the two considered settings — offline learning from expert observations and online learning.

RL Learning Configurations

We used the standard models from the python library `stable_baselines3` and `imitation`.[5] [6] The PPO and GAIL network architecture is a fully connected neural network of size 512×256×256, i.e., three layers in which the first is composed of 512 units, and the second and third have 256 units, and the selected activation function for each layer is $tanh$. The architecture of the BC agent's network is a fully connected neural network with two layers with 32 units each. We selected the $tanh$ activation in each layer to match the previous algorithms. Finally, we trained the Q-Learning agent with the following configuration: a learning rate of 0.01 and a discount factor of 0.99. Additionally, since we used Q-Learning as an online learning approach, we configured its random exploration parameter to start with maximal exploration, i.e., 1, and to decay with a factor of 0.02.

Evaluation Measures

The primary evaluation measure we consider is the *success rate*, which is the ratio of problems solved by the evaluated agent with the given amount of expert observations (for the offline learning setting) or episodes (for the online learning setting).

We divided our dataset into a single training problem and five test problems, and we measured the success rate on both the training problem and the test set problems. For the Item Counts modeling approach, this division has no meaning since the train and test problems are not sufficiently different. Thus, in this setting, we measured the success rate on the map that the agents trained on. In the online setting, we also measured the number of steps until the agent reaches the goal. Recall that an episode ends after at most 64 steps.

All Blocks and Relevant Blocks Modeling Results

Algorithm	# Trajectories - Train	# Solved - Test
GAIL	NA	0/5
BC	2000	0/5
Q-Learning	1	0/5
N-SAM	1	5/5

Table 1: The learning statistics of the offline setting on the All Blocks and Relevant Blocks modeling approaches.

The results for All Blocks and Relevant Blocks were practically the same. Consider first the results for the offline learning with expert observations setting. Table 1 show the number of iterations until the evaluated agent could solve the

[5]https://stable-baselines3.readthedocs.io/en/master/

[6]https://imitation.readthedocs.io/en/latest/algorithms/bc.html

train set problems perfectly (the "#Trajectories - Train" column), and the number of test set problems solved (out of 5) using the learned model (the "#Solved - Test"). The results show that GAIL could not learn even how to solve the train set problems even after 100,000 iterations. The BC algorithm was able to learn the policy for the train set problems but required 2,000 expert trajectories. In contrast, N-SAM and Q-Learning learned a model that correctly solves the training problem much faster, requiring only one expert trajectory. However, only N-SAM's model was able to generalize so that it could be used to solve the test set problems. Thus we can deduce that the RL algorithms overfitted to the training set while the action model learned by N-SAM allowed easy generalization to solve other problems in the domain.

Unfortunately, none of the evaluated online learning algorithms were able to solve the given task with the restriction of 100,000 iterations. The reason for this is that finding a sequence of actions that accomplishes the task in the All Blocks and Relevant Blocks representations is difficult. We also experimented with several reward-shaping techniques to help it converge but were not able to do so. This highlights the difficulty of this task and the importance of choosing effective state representation.

Item Counts Modeling Results

Next, we discuss the results for the Item Counts modeling approach. For the offline learning setting, we observed that GAIL required more than 1000 iterations to start improving. We also observed that both N-SAM and the Q-Learning algorithms learned their model after a single iteration. On the other hand, the success rate of the BC algorithm was lower until it learned the correct policy after 180 iterations. Since these trends present no difference in the learning rate between the Q-Learning and N-SAM, and the difference between these algorithms and the rest is vast, we decided to not present the data graphically.

In Figure 5, we present the moving average of the rewards the learning algorithms gained as a function of the number of iterations on the input trajectory for the online learning setting. In this setting, we averaged the values the same way as the previous experiment, i.e., we averaged over ten samples. When we experimented with the online algorithms, we observed that the N-SAM-based algorithms outperformed the RL algorithms. PPO based N-SAM presented the best results in terms of success rate. After three trajectories, the algorithm learned the model perfectly and the planning task was always solved. We also observed that the Q-Learning based N-SAM behaves unexpectedly. It starts with an average success rate of 0.4, and it declines after six trajectories, after which it stabilizes on a perfect score.

To understand the reason for this behavior, we examined the PDDL domains that N-SAM had output. We observed that N-SAM learned complex inequalities for the CRAFT_WOODEN_POGO whereas after fewer trajectories, the preconditions for the actions were disjunctions of numeric equalities. This fits the assumption that N-SAM can learn restrictive inequalities that become more relaxed once the algorithm receives more trajectories.

Figure 6 presents the average number of steps the agents had taken until they reached the goal as a function of the number of iterations. We note that the expert agent completed the task in eleven steps. Additionally, we set the maximum number of steps the agents can use to roam the environment to 64. Thus, any agent performing 64 steps is considered to not have solved the task. We observe that the fastest algorithm to reach the expert's number of steps is the PPO based N-SAM when on average, after 26 using 26 trajectories, its model was accurate enough for the solver to solve the planning task with the optimal number of steps. Following this algorithm is the Q-Learning based N-SAM that converged to 15 steps. The RL algorithms, on the other hand, did not converge and, at best, solved the problem in the same number of steps as the Q Learning N-SAM did.

This shows another advantage of using N-SAM as a part of the modeling framework. While the RL algorithms did not converge, and on occasion even failed to solve the input task, the N-SAM based improved over time and even provided the optimal solution to the problem.

In summary, we observe that in all modeling approaches, the N-SAM-based approach was either the same or significantly better than the RL-based approach. In addition, we observed that only with the more compact Item Counts state representation, were we able to solve the problem without expert observations.

Conclusions and Future Work

In this work, we explored a new approach to solving the wooden Pogo crafting task. We proposed three ways to represent states and actions in this domain, namely All Blocks, Relevant Blocks, and Item Counts. Then, we presented our approach that learns an action model of the environment using N-SAM and then uses a state-of-the-art planning algorithm to solve the planning task. We compared our approach to several RL algorithms in two settings: offline learning with expert observations and online learning. Experimental evaluation showed the benefit of a planning-based approach coupled with an action model learning algorithm, outperforming all RL-based approaches. Also, we showed that online learning was only possible with the most compact modeling (Item Counts), which emphasizes the importance of finding the right domain representation. In future work, we intend to explore solving the complete task without masking any internal functionality. This setting will present challenges since actions can have conditional effects depending on the states they are executed on. In addition, we intend to explore improving the action selection process for the online learning approach so action generation will focus on model improvement rather than the goal-oriented approach used in this research.

References

Abbeel, P.; and Ng, A. Y. 2004. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, 1.

Aeronautiques, C.; Howe, A.; Knoblock, C.; McDermott, I. D.; Ram, A.; Veloso, M.; Weld, D.; SRI, D. W.; Barrett,

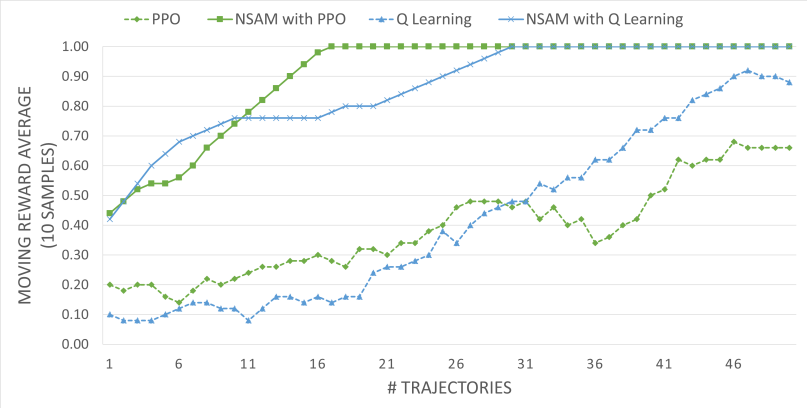

Figure 5: Moving average of every ten samples for the online learning algorithms as a function of the number of iterations. The results are measured for the Item Counts modeling setting.

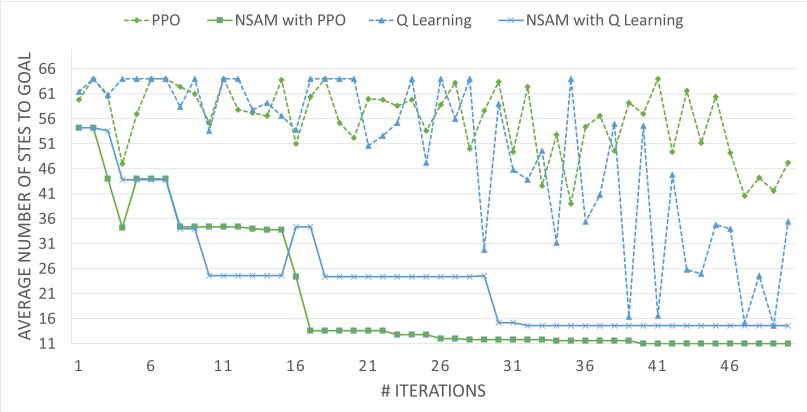

Figure 6: The average number of steps till solution for each online learning algorithm. The results are measured for the Item Counts modeling setting.

A.; Christianson, D.; et al. 1998. PDDL— The Planning Domain Definition Language. *Technical Report, Tech. Rep.*

Aineto, D.; Celorrio, S. J.; and Onaindia, E. 2019. Learning action models with minimal observability. *Artificial Intelligence*, 275: 104–137.

Argaman Mordoch, B. J., Roni Stern. 2023. Learning Safe Numeric Action Models. In *AAAI Conference on Artificial Intelligence (AAAI)*.

Bartheye, O.; and Jacopin, E. 2009. A real-time PDDL-based planning component for video games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 5, 130–135.

Bratko, I.; Urbančič, T.; and Sammut, C. 1995. Behavioural cloning: phenomena, results and problems. *IFAC Proceedings Volumes*, 28(21): 143–149.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. Openai gym. *arXiv preprint arXiv:1606.01540*.

Duarte, F. F.; Lau, N.; Pereira, A.; and Reis, L. P. 2020. A survey of planning and learning in games. *Applied Sciences*, 10(13): 4529.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4): 189–208.

Fox, M.; and Long, D. 2003. PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *Journal of artificial intelligence research*, 20: 61–124.

Frazier, S.; and Riedl, M. 2019. Improving deep reinforcement learning in minecraft with action advice. In *Proceedings of the AAAI conference on artificial intelligence and interactive digital entertainment*, volume 15, 146–152.

Georgievski, I.; and Aiello, M. 2015. HTN planning: Overview, comparison, and beyond. *Artificial Intelligence*, 222: 124–156.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: theory and practice*.

Goss, S. A.; Steininger, R. J.; Narayanan, D.; Olivença, D. V.; Sun, Y.; Qiu, P.; Amato, J.; Voit, E. O.; Voit, W. E.; and Kildebeck, E. J. 2023. Polycraft World AI Lab (PAL): An Extensible Platform for Evaluating Artificial Intelligence Agents. *arXiv preprint arXiv:2301.11891*.

Guss, W. H.; Codel, C.; Hofmann, K.; Houghton, B.; Kuno,

N.; Milani, S.; Mohanty, S.; Liebana, D. P.; Salakhutdinov, R.; Topin, N.; et al. 2019a. NeurIPS 2019 competition: the MineRL competition on sample efficient reinforcement learning using human priors. *arXiv preprint arXiv:1904.10079*.

Guss, W. H.; Houghton, B.; Topin, N.; Wang, P.; Codel, C.; Veloso, M.; and Salakhutdinov, R. 2019b. Minerl: A large-scale dataset of minecraft demonstrations. *arXiv preprint arXiv:1907.13440*.

Ho, J.; and Ermon, S. 2016. Generative adversarial imitation learning. *Advances in neural information processing systems*, 29.

Hoffmann, J.; and Magazzeni, D. 2019. Explainable AI planning (XAIP): overview and the case of contrastive explanation. *Reasoning Web. Explainable Artificial Intelligence*, 277–282.

Juba, B.; Le, H. S.; and Stern, R. 2021. Safe Learning of Lifted Action Models. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 379–389.

Justesen, N.; Bontrager, P.; Togelius, J.; and Risi, S. 2019. Deep learning for video game playing. *IEEE Transactions on Games*, 12(1): 1–20.

Konidaris, G.; Kaelbling, L. P.; and Lozano-Perez, T. 2018. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research*, 61: 215–289.

Kumar, A.; Zhou, A.; Tucker, G.; and Levine, S. 2020. Conservative q-learning for offline reinforcement learning. *Advances in Neural Information Processing Systems*, 33: 1179–1191.

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

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *nature*, 518(7540): 529–533.

Palucka, T. 2017. Polycraft World teaches science through an endlessly expansive universe of virtual gaming: https://polycraft.utdallas.edu. *MRS Bulletin*, 42(1): 15–17.

Pomerleau, D. A. 1991. Efficient training of artificial neural networks for autonomous navigation. *Neural computation*, 3(1): 88–97.

Scala, E.; Haslum, P.; Thiébaux, S.; and Ramirez, M. 2016. Interval-based relaxation for general numeric planning. In *European Conference on Artificial Intelligence (ECAI)*, 655–663.

Scheller, C.; Schraner, Y.; and Vogel, M. 2020. Sample efficient reinforcement learning through learning from demonstrations in minecraft. In *NeurIPS 2019 Competition and Demonstration Track*, 67–76. PMLR.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2): 181–211.

Tessler, C.; Givony, S.; Zahavy, T.; Mankowitz, D.; and Mannor, S. 2017. A deep hierarchical approach to lifelong learning in minecraft. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31.

Vinyals, O.; Babuschkin, I.; Czarnecki, W. M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D. H.; Powell, R.; Ewalds, T.; Georgiev, P.; et al. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782): 350–354.

Wang, X. 1994. Learning planning operators by observation and practice. In *Second International Conference on Artificial Intelligence Planning Systems (AIPS)*, 335–340.

Watkins, C. J.; and Dayan, P. 1992. Q-learning. *Machine learning*, 8: 279–292.

Watkins, C. J. C. H. 1989. Learning from delayed rewards.

Wichlacz, J.; Torralba, A.; and Hoffmann, J. 2019. Construction-planning models in minecraft. In *Proceedings of the ICAPS Workshop on Hierarchical Planning*, 1–5.