An Architecture for Novelty Handling in a Multi-Agent Stochastic Environment: Case Study in Open-World Monopoly

Tung Thai
Tufts University
Medford, MA
tung.thai@tufts.edu

Ming Shen
Arizona State University
Tempe, AZ
mshen16@asu.edu

Neeraj Varshney
Arizona State University
Tempe, AZ
nvarshn2@asu.edu

Sriram Gopalakrishnan
Arizona State University
Tempe, AZ
sgopal28@asu.edu

Utkarsh Soni
Arizona State University
Tempe, AZ
utkarsh.soni@asu.edu

Matthias Scheutz
Tufts University
Medford, MA
matthias.scheutz@tufts.edu

Chitta Baral
Arizona State University
Tempe, AZ
chitta@asu.edu

Jivko Sinapov
Tufts University
Medford, MA
jivko.sinapov@tufts.edu

Abstract

The ability of AI agents and architectures to detect and adapt to sudden changes in their environments remains an outstanding challenge. In the context of multi-agent games, the agent may face novel situations where the rules of the game, the available actions, the environment dynamics, the behavior of other agents, as well as the agent's goals suddenly change. In this paper, we introduce an architecture that allows agents to detect novelties, characterize those novelties, and build an appropriate adaptive model to accommodate them. Our agent utilizes logic and reasoning (specifically, Answer Set Programming) to characterize novelties into different categories, as to enable the agent to adapt to the novelty while maintaining high performance in the game. We demonstrate the effectiveness of the proposed agent architecture in a multi-agent imperfect information board game, Monopoly. We measure the success of the architecture by comparing our method to heuristics, and vanilla Monte-Carlo Tree Search approaches. Our results indicate precise novelty detection, and significant improvements in the performance of agents utilizing the novelty handling architecture.

Introduction

Recent applications of game AI employ sophisticated search and learning algorithms to train the agent in the hopes of defeating a human opponent. One such example is AlphaGo (Silver et al. 2016), a computer program designed by Google DeepMind. AlphaGo and other AI agents often focus heavily on perfect information games, which are *closed-world* environments where the rules of the game, the goals of the players and the full state of the board are always known by all agents. This characteristic often simplifies the training of AI agents due to the assumption that the environment does not change over the course of training (Brown, Sandholm, and Amos 2018; Nash 1951). Many games rely on the notion of incomplete information in which the opponents' initial states, game rules, or even actions are unknown, and information must be discovered through interactions (Boney et al. 2021; Wanyana and Moodley 2021). Incomplete information games are even more challenging when novelties are added to the game. Novelties are changes to the environment that occur after the agent has been trained or deployed in an operational setting. Such changes can include novel entities present in the game, novel environmental dynamics, and even novel game rules. Novelties present a special challenge to existing methods for training AI agents as those methods typically make the *closed-world* assumption. (Ganzfried and Sandholm 2011; Koller and Pfeffer 1995; Heinrich and Silver 2016; Brown and Sandholm 2019).

To address the challenges of open-world environments, this paper proposes an architecture that enables AI agents to detect, characterize, and adapt to novelties in a multi-agent stochastic games. We utilize a general novelty-handling framework as an extension to the Distributed Integrated Affect Reflection Cognition (DIARC) architecture (Schermerhorn et al. 2007; Muhammad et al. 2021; Scheutz et al. 2019), and logical reasoning to detect, learn and adapt to novelties in *open-world* environments. Our paper provides: (1) a new architecture framework to handle novelties in a multi-agent stochastic environment and (2) a new logical reasoning approach to characterize novelties in order to learn and accommodate them during the planning phase. After the system detects the novelty, our architecture accommodates the planning agent, in this case, a Monte-Carlo Tree Search (MCTS) agent, by updating the planning agent's custom evaluation function and knowledge base.

We tested the architecture with four novelty classes: 1) *action-effect*; 2) *action-precondition*; 3) *state triggers*; and 4) *novel entities*. We injected more than 20 novelties from these four classes into the Monopoly environment. The re-

sults indicate that our novelty handling can detect and support more than 20 novelties with high precision. A third-party team then evaluated our architecture by injecting novelties hidden from our research team. This evaluation allows us to maximize the results' integrity and quantify the effectiveness of the architecture in a multi-agent imperfect information environment. The third-party team also evaluated our agent's performance against other approaches that attempted to solve the game of Monopoly in terms of novelty detection accuracy and win ratio. The results indicate that our agent can detect novelty with a high accuracy rate while maintaining a dominant performance against other competitors.

Background & Related Works

Methods based on model-free deep reinforcement learning (RL), imitation learning, and variants of Markov decision processes (MDP) have shown great potential in dealing with multi-agent game environments such as Monopoly (Haliem et al. 2021; Brown, Sandholm, and Amos 2018). However, these approaches are not explicitly developed to deal with novelties in the environment. Even though these methods have shown excellent results in imperfect information environments, addressing novelties remains a challenge.

RL methods have been proposed for non-stationary environments (Padakandla, K. J., and Bhatnagar 2020; Choi, Yeung, and Zhang 2000; Hester and Stone 2012), e.g., online learning and probabilistic inference were used to learn an offline policy in a lifelong non-stationary environment (Xie, Harrison, and Finn 2020). This method uses past and present experience to learn a new representation of the world. Other approaches have attempted to develop algorithms that learn a suitable control policy in dynamically-changing environments. However, RL approaches do not explicitly characterize novelties, and adaptation to novelties may often take as long as training the agent from scratch (Goel et al. 2021).

Deep RL methods have been utilized to solve imperfect information games (Heinrich and Silver 2016). Such methods have been shown to converge reliably to approximate the Nash equilibrium of the game. However, solving the problem in a larger and more complex environment remains a challenge. Other model-free methods for non-stationary environments include context detection and environmental model for reinforcement learning (da Silva et al. 2006; Choi, Yeung, and Zhang 2000). While such methods can handle small expected changes in the environment such as traffic light control in rush hour. It remains a challenge to deal with more complex changes, especially in a stochastic environment. One example of such complex changes is traffic control when an unexpected event blocks the road.

Integrated symbolic planning and learning methods are new approaches to handle novelties in the *open-world* environment (Sarathy et al. 2021; Chitnis et al. 2021; Muhammad et al. 2021; Neary et al. 2021; Kaelbling and Lozano-Perez 2013; Pinto et al. 2021; Boult et al. 2021). However, most of the current target environments are deterministic environments with complete information known to the agent. In contrast, we propose an architecture that combines symbolic reasoning, and stochastic planning to detect and accommodate novelties on the fly in imperfect information environments. Improvements are needed for these approaches to perform well in an imperfect information environment, where a part of the information is hidden, such as adversary mental models, other agent's behavior models, or random components in the environment.

Preliminaries

Domain Knowledge and Symbolic Planning

At the beginning of the planning process, we assume the agent starts with a complete domain knowledge of environment(Gizzi et al. 2021). The domain knowledge contains information such as entities, predicates, symbolic states, and operators in the environment. Formally, we denote the domain knowledge $\mathcal{K} = \langle \mathcal{E}, \mathcal{P}, \mathcal{F}, S, \mathcal{O} \rangle$. The set $\mathcal{E} = \{e_1, \cdots, e_n\}$ is a finite set of all entities in the environment. Predicates set $\mathcal{P} = \{p_1(x), \cdots, p_n(x)\}$, $p \subset \mathcal{E}$, contains all the predicates in the environment and their negation. The negation of any particular predicate $p_i(x)$ is defined as $\neg p_i(x)$, with $p_i(x), \neg p_i(x) \in \mathcal{P}$. The set $\mathcal{O} = \{o_1, \cdots, o_n\}$ is a finite set of all operators in the environment. Fluents set $\mathcal{F} = \{f_1(x), \cdots, f_n(x)\}$, $f \subset \mathcal{E}$, contains all fluents in the environment. Fluents in the environment are simply numeric variables, or boolean variables that the agent may deal with during task performance. The set of symbolic states set is represented by S. We encode all the information above into a planning domain using Answer Set Programming (ASP) (Baral 2003). Each operator o_i in operator set $\mathcal{O}$ is defined by a preconditions set $\delta_i(o) \in \mathcal{P}$ and an effects set $\beta_i(o) \in \mathcal{P}$. A preconditions set $\delta_i(o)$ of an operator o_i includes all the predicates that need to be satisfied in order to execute the operator. Meanwhile, the effects set β_i of an operator o_i indicates the expected predicates or expected results after a successful execution of operator o_i. For negation of predicate $\neg p_i(x)$, preconditions set δ_i must be unsatisfied before execution, and effects set β_i must be false after execution. Finally, we construct the planning task $\mathcal{T}$ which includes entities set, predicates set, operators set, initial state and goal state of the environment, $\mathcal{T} = \langle \mathcal{E}, \mathcal{P}, \mathcal{O}, s_0, s_g \rangle$, so the agent can solve all the tasks appropriately.

Multi-Agent Markov Decision

In order to model the decision making agent in stochastic environment, we formalize the environment in which the agent operates as a multi-agent Markov Decision Process (MMDP) (Boutilier 1970). An MMDP model $\mathcal{M}$ is defined by a tuple $\mathcal{M} = \langle \alpha, S, \{A_i\}_{i \in \alpha}, \Sigma, R, T, \gamma \rangle$, where α is a finite set of agents, identified by $i \in \{1, 2, \cdots, n\}$; S is a finite set of states in the environment; a finite set of actions A contains all actions of each agent i; Σ is a joint transition function so that $\Sigma : S \times A \times S \rightarrow (0, 1]$, i.e, the probability mass function, such that $\Sigma(s, a_1, a_2, \cdots, a_n, s) = P(s'|s, a_1, \cdots, a_n)$. R is the joint reward function so that $R : S \times A \times S \rightarrow \mathbb{R}$. T is a finite set of discrete time steps $t \in \{1, 2, 3, \cdots, T\}$. Finally, $\gamma \in (0, 1]$ is the discount factor which determines the importance of immediate and future rewards. A solution to an MMDP problem is a policy π. Policy π is defined as a procedure for the action selection of an agent at a specific state. A policy π is a map from states

to actions so that $\pi : S \rightarrow A$, where A is represents the set of joint actions in the multi-agent case.

Constrained MMDP in Monopoly

We customize the traditional MMDP to describe the Monopoly domain in a high-level concept. In traditional MMDPs, actions are performed as a sequence $\{A_i\}_{i\in\alpha}$, in which each player takes turns to take actions. However, in Monopoly, on each turn, a subset of agents ω_k, such that $\omega_k \subset \alpha$ and $k = 1, 2, 3, \cdots, n$ or all agents are allowed to act. For example, when a player decides to auction property, all other players can bid on the property. There is also a case where only two players interact with each other in a trade.

Novelty

Novelty is a change in the environment where the agent can neither detect the change from logical deduction nor past experience (not in the knowledge base). Novelty set $\mathcal{N}$ can be defined as $\mathcal{N} = \langle \alpha', S', \{A'_i\}_{i\in\alpha'}, \mathcal{E}', \mathcal{F}' \rangle$m where, α' represents a finite set of novel agents, so that $\alpha' \cap \alpha = \emptyset$. S' is a set of novelty set, such that $S' \cap S = \emptyset$. A' represents a set of novel actions of each agent i, such that $A' \cap A = \emptyset$. We assume that the preconditions δ' and effects β' of the new action set A' are completely unknown to the agent, both must be discovered through agent's interactions. $\mathcal{E}'$ is used to denote the set of novel entities in the environment, such that $\mathcal{E}' \cap \mathcal{E} = \emptyset$. Finally, $\mathcal{F}'$ is used to denote the set of novel fluents in the environment, such that $\mathcal{F}' \cap \mathcal{F} = \emptyset$.

Problem Formulation: Novelty Detection and Novelty Adaptation

We utilize the DIARC framework to plan, learn and adapt to novelties in the environment (Schermerhorn et al. 2007; Scheutz et al. 2019). The framework allows us to map entities $\mathcal{E}$, symbolic states $\mathcal{S}$, predicates $\mathcal{P}$, fluents $\mathcal{F}$ and operators $\mathcal{O}$ in the current environment to the knowledge base $\mathcal{K}$. Based on this information, we can determine the planning task $\mathcal{T}$. However, since novelties are injected into the environment, the plan must be changed in order to adapt to new conditions or rules. In an effort to adapt to novelty, we must detect and identify the new changes, update the knowledge base accordingly, then adapt with a new plan.

As described in Section 3.2, the pre-novelty environment is described as a MMDP model $\mathcal{M} = \langle \alpha, S, \{A_i\}_{i\in\alpha}, \Sigma, R, T, \gamma \rangle$. We define a detection function $d(s, a)$, used to determine if there is any change in the environment at a particular state s, and action a; and an identification function $\iota(s, a)$, used to determine the cause of the change based on logical reasoning. The objective of these functions is to model the environment after novelty (post-novelty) $\mathcal{M}'$, such that $\mathcal{M}' = \langle \alpha', S', \{A'_i\}_{i\in\alpha'}, \Sigma', R', T, \gamma' \rangle$. In which, α' is the new number of agents in the environment post-novelty. S' is the finite states set post-novelty, this set may includes a new initial state s'_o and a new goal state s'_g. The set $\{A'_i\}_{i\in\alpha'}$ is the finite actions set with respect to each agent α in the environment. Σ' is the new transition function. R' is the new reward function post novelty. Finally, γ' is the new discount factor value post novelty.

From the new model of the world $\mathcal{M}'$, we can modify the planning task $\mathcal{T}$ into $\mathcal{T}' = \langle \mathcal{E}', \mathcal{P}', \mathcal{O}', s'_0, s'_g \rangle$.

Novelty Handling Architecture

Agent Architecture

The previous version of novelty-centric DIARC can support novelty detection in single-agent MDP and deterministic environments. We extend the novelty-centric DIARC architecture to handle multi-agent stochastic environments. The detailed architecture is shown in Figure 1. The architecture includes four main components: environment, novelty handling, knowledge base, and planning agent. In detail, the environment component contains the game simulator, game interface, and a goal management system. The knowledge base includes game information such as actions, interactions, agent information, and relations. Then, the novelty handling component operates to detect and identify novelty that may get injected at the beginning of the operation process. A deeper explanation of the novelty handling system is located in *Novelty Handling* section. After the change is determined, the novelty handling component interacts with the knowledge base by updating the detected novelty. The system then updates this novelty in the knowledge base with its new effects or new states. When the agent receives the updated information, the Monte-Carlo Tree Search (MCTS) planning agent can plan accordingly. Our MCTS agent's custom value function accounts for the property value of all the agent's assets $\mathcal{M}_{assets}$, short-term expected gain $\mathcal{R}_s$, long-term expected gain $\mathcal{R}_l$, and a Monopoly beneficial gain term $\mathcal{M}_{Monopoly}$ which is discussed in depth in *Monte Carlo Tree Search Planning Agent* section.

Novelty Categorization and Handling

We categorize novelty into four different types: *action-effect*, *action-precondition*, *state triggers*, and *novel entities*.

Action Effect Novelty In this novelty class, a new effect $\beta'(a_i)$ is triggered by an existing action, one that was previously unknown to the agent's knowledge. This new effect is different from the expected effects set $\beta(o)$ that is known to the agent. In this case, effect $\beta(a_i)$ triggered by action a_i has been changed. We illustrate this novelty using *auction tax* novelty, where there is a new additional fee on top of the player's bids in the auction. In order to detect this novelty type, we apply action a_t to state S_t with effect $\beta(a_t)$ which will result in a state S_{t+1} (expected result). If the expected state S_{t+1} does not equal to actual state S'_{t+1}, and $\beta(a_t)$ can not be found in the overall effect set $\beta(o)$, there is at least one novelty that was injected into the environment. After the novelty is detected, we update the new effect to our knowledge base by adding the new effect which correlated to the action into the effect set planning agent.

Action Precondition Novelty This class of novelty refers to changes in the preconditions of an action. To illustrate this novelty in the Monopoly game, we inject a *restricted color* novelty, where building houses in a specific color set is not permitted. In this class of novelty, the precondition set $\delta(a_i) \subset \delta(o)$ of at least one action has been changed in

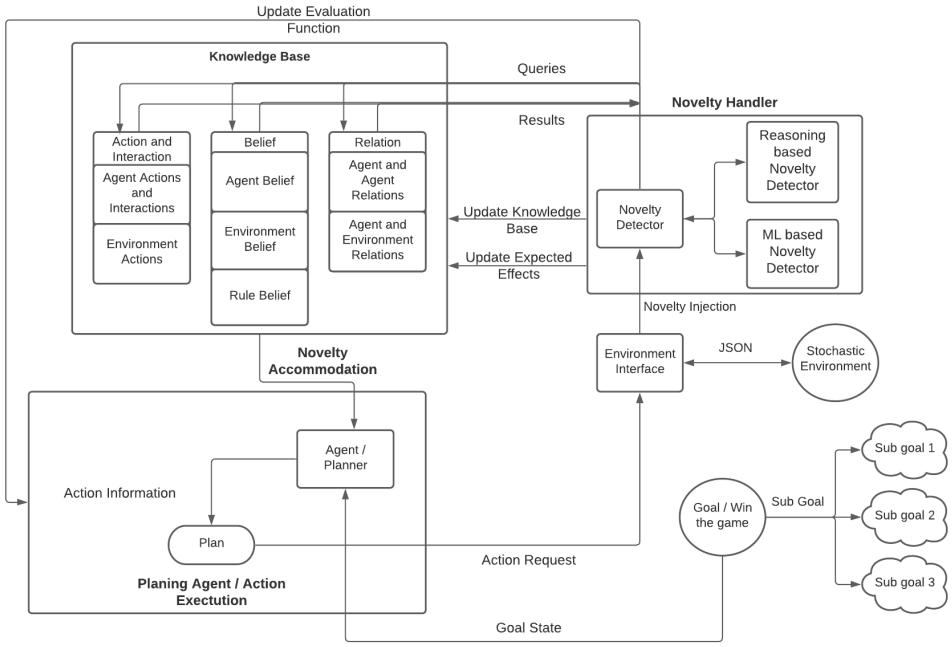

Figure 1: The overall architecture of the novelty handling framework in a stochastic environment

the environment; this includes the case where new precondition sets are added. We apply Algorithm 1 to detect the novelty and update the agent's knowledge base accordingly. Two scenarios may occur in this type of novelty. In the first scenario, all preconditions for action are satisfied. However, the action fails to be completed successfully during its execution. In the second scenario, the action is successfully executed despite some unmet preconditions. After the novelty is detected, we compute the new set of preconditions of the action. We also evaluate the novelty's effects on our plan and update our planning custom value function accordingly. For example, *restricted color* novelty may negatively affect our long-term expected gain and Monopoly beneficial gain. After the novelty is detected, we adjust the value function as to avoid investing in the restricted color.

State Trigger Effect Novelty A new effect is triggered in this novelty class when the game or the player is in a specific state. The new effects $\beta'(s_i)$ are entirely unknown to the agent's knowledge. In a typical scenario, state S_t and next state S_{t+1} would be the same if the agent does not perform any action. However, state S_t can now trigger an effect in the game that can cause state S_{t+1} to change without any action. For instance, we demonstrate this novelty in Monopoly by changing the *Go Increment*, where we change the amount of money players may receive after they pass Go. In order to detect this class of novelty, we must compare the discrepancy between the actual next state and the expected next state. If no action is taken in the current time step, the actual next state of the environment has changed. Thus, there is a difference between the actual and expected next state. This event means there is at least one state trigger effect novelty is injected into the environment. After the novelty is detected, we update the state knowledge base by adjusting the state-transition Σ according to the new changes, $\Sigma \rightarrow \Sigma'$. We also update all the components of the MCTS value function accordingly in order to accommodate the novelty.

Novel Entities In this novelty class, a completely new entity is added to the environment. These new entities include additional new dices, new properties, or new property types to new rules, new win conditions. Some of the new entities are categorized as below:

1. New Action: This class of novelty contains new actions available to the agent. In detail, this novelty extends the action space A to a new action space A' which includes all the old actions and new actions.
2. New State: This class of novelty contains new states to the environment. In detail, this novelty extends the state space S to a new space S'.

If new entities are injected into the environment, changes may occur in the state space S, and action space A. Hence, we provide a simple detection mechanism to detect this type of novelty. In this detection mechanism, we monitor the state space, action space, and goal space. Suppose there is any certain change at any particular time that results in the different new state space S', and action space A'. The system detects these changes and reports the novelty immediately. The system updates the state space, and the action space, based on the novelty detection results. Finally, this new knowledge is applied to the MCTS planning agent. Novelties that can change action and goal spaces are essential to detect and

Algorithm 1 Action Precondition Novelty Detection

Initialization: State Space S, Action Space A, precondition Set of all actions $\delta(o)$,
$ND = False$ ▷ Novelty Detected
$t = 0$ ▷ Time step
while Game is not end **do**
    Case 1: All precondition $\delta(a_{t+1})$ for action a_{t+1} are satisfied but action a_{t+1} is not executable
    if $\delta(a_{t+1}) == True \wedge a_{t+1} == False$ **then**
        $ND = True$
        Case 2: At least one precondition for action A_{t+1} is not satisfied but action a_{t+1} is executable
    else if $\delta(a_{t+1}) == False \wedge a_{t+1} == True$ **then**
        $ND = True$
    else
        $ND = False$
    end if
    if $ND == True$ **then**
        $\delta(o).insert(\delta(a_{t+1}))$ ▷ Update Precondition Set
        $\mathcal{M}_{assets} \leftarrow \mathcal{M}'_{assets}$ ▷ Update assets value
        $\mathcal{R}_s \leftarrow \mathcal{R}'_s$ ▷ Update short-term gain
        $\mathcal{R}_l \leftarrow \mathcal{R}'_l$ ▷ Update long-term gain
        $\mathcal{M}_{Monopoly} \leftarrow \mathcal{M}'_{Monopoly}$ ▷ Update Monopoly beneficial value
    end if
    t = t + 1
end while
return ND

adapt since they can dramatically change the game dynamic (i.e. winning conditions, or new important actions).

Novelty detection and identification using Answer Set Programming (ASP)

We monitor the game board as communicated by the game server and continually compare it with our "expectation" of the game board state. To compute our expectation of the game board state, we use answer set programming (ASP) to represent states of the game board, represent various actions and their preconditions, end effects, and do hypothetical reasoning about the effect of actions on the game board state. The game server gives us the real game board states and actions that have occurred between two states. Starting with a particular game board state, when we notice a discrepancy between our expectation of the next game board state and the real next game board state we surmise that something must have changed with respect to the game, i.e., a novelty may have been introduced which makes some aspect of our domain (of the game) representation incorrect.

Next, the agent uses a novelty identification module to characterize the novelty. This module has several sub-modules (which can be run in parallel), each focused on determining a specific type of novelty. Each novelty identification sub-module uses the same ASP code (except two changes) that is used for hypothetical reasoning about the effect of an action. The first change is that, a particular parameter, which is the focus of that particular sub-module, which was originally a fact, is now replaced by "choice" rules of ASP that enumerate different values that the parameter can take. The second change is that constraints are added to remove possible answer sets where the predicted game board state does not match with the observed game board state. The answer sets of the resulting program give us the values of the parameter which reconcile the predicted game board state and the observed game board state. If there is only one answer set, and thus a unique parameter value, then if this value is different from, the value that we had earlier, then we have identified a novelty. Now we can update our ASP code that was used for hypothetical reasoning by simply replacing the earlier value of the parameter by the new value.

Monte Carlo Tree Search Planning Agent

Our agent uses Monte-Carlo Tree Search with a custom value function as a method for selecting its next action. MCTS is a method which can be used to make an optimal decision or choose the winning path based on the statistical simulation of each state or move in the game.

Selection: In this stage, a leaf node is chosen that has the highest score. The score is computed using a customized value function. The function accounts for property value of all the agent's assets $\mathcal{M}_{assets}$, short term expected gain $\mathcal{R}_s$, long term expected gain $\mathcal{R}_l$, and a Monopoly beneficial gain term $\mathcal{M}_{Monopoly}$. i.e.

$$\mathcal{V} = \mathcal{M}_{assets} + \mathcal{R}_s + \mathcal{R}_l + \mathcal{M}_{Monopoly}$$

A leaf node is a node that contain no child nodes. To ensure the algorithm thoroughly explores different paths, it selects a leaf node that has been ignored for a certain amount of iterations instead of the leaf node with the highest score, or most potential to win.

Expansion: Child nodes are created that contains the next board state from the selected parent node. The expansion stops after a fixed search depth or level. Due to the randomness of the dice's roll, we only applied a one-step look-ahead for the expansion. This constraint prevents the agent from taking too long to go through irrelevant nodes. This optimization results in an asymmetrical tree growth where the search more frequently focuses on visiting relevant nodes. Further enhancements were made to MCTS to encode domain-specific game knowledge into the search.

Simulation: After expansion, a simulation of the game is run to calculate the success probability of each node.

Back Propagation: Based on the simulation results, MCTS updates every node in the winning path. The optimal decision is chosen according to a desired distribution.

Environment Implementation

Monopoly

Traditional Monopoly, shown in Figure 2, is a multi-player adversarial game where players roll dice to move across the board. Monopoly is an imperfect information game since community cards, chest cards, and trades potential are unknown. The game can support up to 4 players, describe in Table 1. The 4 players start at the same position, the *Go*

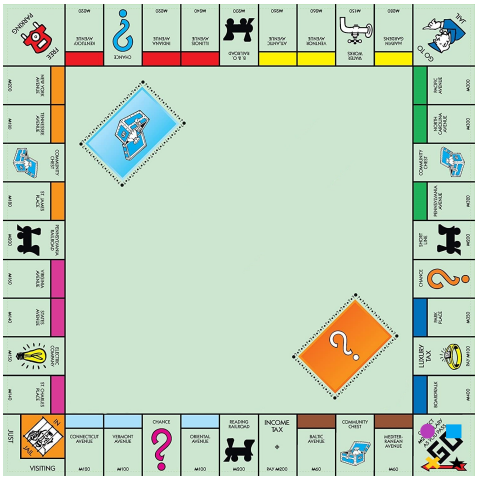

Figure 2: Classic Monopoly Board

tile. The game's objective is to be the last player standing in the game after bankrupting others. This goal is reached by buying properties, monopolizing color sets, and developing houses on properties. If one player lands on a property owned by another player, they get charged rent or a fee. After monopolizing color sets and developing houses, players can charge higher rent or fees when the other players land on their properties. Any plan of action in the game needs to be adapted to dice rolls and decisions of other players. These characteristics of the game make it more challenging for integrated planning and execution. In the game simulator, novelties can be injected on top of the standard game to study how the agent reacts to these changes.

Novelties in Monopoly

We implement all four types of novelty discussed in *Novelty Handling* section into a classic Monopoly game. Some examples of actual Monopoly novelty are described as below:

- We illustrate action effect novelty in the Monopoly game using a *sell property percentage* novelty for action effect novelty. In a classic Monopoly game, when a player sells their property to the bank, they receive 50% value of the property. However, when we inject *sell property percentage* novelty, the sell rate goes from 50% to 75%.
- To demonstrate the action precondition novelty class in a Monopoly game, let us look at the *restricted color* novelty which is shown in Figure 3. The figure describes the game's current state where properties ownership is illustrated using players' icons, and buildings are shown using house and hotel icons. When players monopolize a property's color set in the Monopoly game, players can build houses and hotels on those properties. However, when the *restricted color* novelty is injected into the game, players can not build houses and hotels on a specific color set.
- We illustrate state trigger effect novelty using *assigned properties* novelty, show in Figure 4. In a classic Monopoly game, all players start at the same position and own no property. The bank owns all the properties at the beginning of the game. When *assigned properties* novelty occurs, all players are assigned random properties at the beginning of the game. All the properties are distributed equally to 4 players and the bank do not own any properties at the beginning of the game.
- For new entities novelty, we create a new board orientation by swapping positions of properties and making multiple copies of these properties, show in Figure 5.

Player	Symbol
1	Dark Star
2	Blue Square
3	Yellow X Mark
4	Purple Circle

Table 1: Symbol description for players in the game

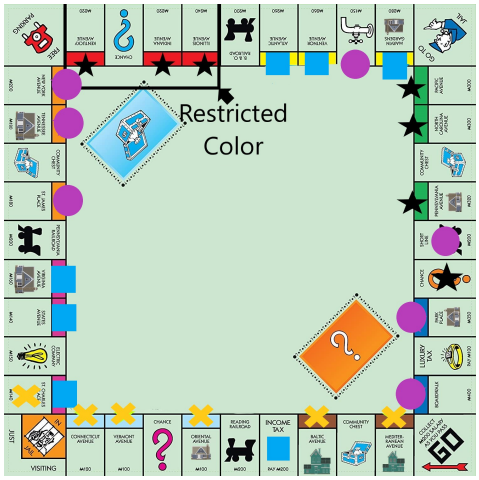

Figure 3: Restricted Color - Action Precondition Novelty

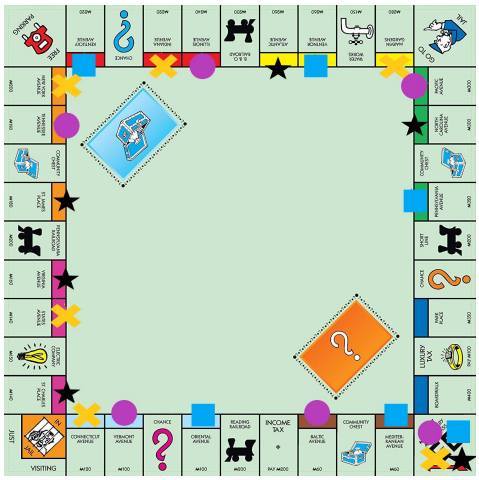

Figure 4: Assigned Properties - State Trigger Effect Novelty

Evaluation & Results

Internal Evaluation

For the internal evaluation, we conduct experiments to test the novelty detection performance, and adaptive agent per-

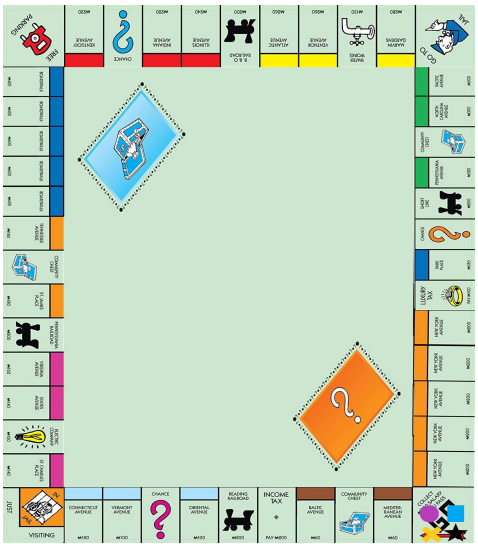

Figure 5: Swap and Extend Board - New Entities Novelty

formance for all four types of novelty. We collect data in a tournament set up, in which a series of games are played to determine the winner. To measure the novelty detection performance, we perform 10 tournaments of 20 games for each novelty within a novelty type. We test at least 5 individual novelties for each type. Each novelty is injected at a random point of the tournament. We then measure the percent of correctly detected trials (CDT), recall, and detection delay in terms of the number of games after novelty was injected. In this case, a trial includes all the tournaments that are tested for one novelty. Percent of CDT is the percent of trails that have at least one True Positive and no False Positives. To evaluate adaptive agent performance, we measure the win rate of the MCTS agent with and without the support of novelty handling architecture against a heuristic agent which embedded some of the most common strategies in Monopoly. Finally, we compute novelty reaction performance (NRP) of the agent based on the following formula:

$$NRP = \frac{\mathcal{W}_{PostNovelty}}{\mathcal{W}_{PreNovelty}}$$

Where, $\mathcal{W}_{PostNovelty}$ is the win rate of the MCTS agent post-novelty. $\mathcal{W}_{PreNovelty}$ is the win rate of the MCTS agent pre-novelty. Table 1 demonstrates the novelty detection performance in 200 tournaments. The results show that our model has 100% percent of correctly detected trials for novelty detection across four types of novelty. The overall results are expected though it is important to note that the recall rate of action effect novelty, and action precondition novelty is lower than the other two types of novelty. It also takes more games for the agent to detect these novelty categories. These results are due to the nature of action novelty and action precondition novelty. We can only detect these novelties types when a specific action is executed. Due to the randomness of the Monopoly game, the action may not happen throughout one entire tournament. For example, we cannot detect restricted color novelty when the agent does not have a chance to monopolize and try to build houses on the specific color set. Table 2 shows the overall performance against a heuristic agent which knows the novelty. Even though the agent with architecture support outperforms the vanilla MCTS agent without novelty handling, there is not much of a difference between them. Some novelties can have an essential effect on the game. For example, restricted color and extended board novelty can significantly affect the agent's strategies for buying and trading properties. On the other hand, other novelties such as sell house, or sell property rate can have minimal effects on the game.

External Evaluation

The external evaluations were performed by a third party, which created a different set of novelties and evaluated our agent based on three different novelty levels below:

- Level 1 [Class]: In this level of novelty, a new class of object or a change in existing objects is injected in the Monopoly game. *Auction tax* novelty is a good example to present this class.
- Level 2 [Attribute]: For attribute novelty, some existence actions or effects are changed. For instance, jail fine can be changed from $50 to $200.
- Level 3 [Representation]: In this level, a new novelty can change the board orientation or structure. An example of this novelty is swap and extend the Monopoly board.

A total of 500 tournaments were run to collect the data. More than 50 novelties were injected in total during the evaluation process. Tournaments were started with a classic Monopoly game with no novelty. At some random point throughout the tournament, novelties were injected. To avoid ambiguity between novelties, only one level of novelties was injected in a tournament. The overall results are shown in Table 4. The results consist of three performance metrics: M1 (CDT), M2 (NRP), and M3, overall win rate of the agent against the baseline agent.

Table 4 shows the results of our agent and the best competitor's performance in different novelty setups. With respect to the percent of CDT, our agent outperformed our

Novelty Type	Percent of CDT	Recall	Detection Delay (number of games)
Action Effect Novelty	100.00%	95.56% ± 50.00%	2.60±3.65
Action Precondition Novelty	100.00%	84.44% ± 50.62%	2.37±2.90
State Trigger Effect Novelty	100.00%	100.00%	0.20±0.40
New Entities Novelty	100.00%	100.00%	0.13±0.18

Table 2: Internal Evaluation Results for Novelty Detection

Novelty Type	Win rate of MCTS agent with the architecture	Win rate of non-adaptive MCTS	NRP
Pre novelty win rate: 76.48%			
Action Effect Novelty	79.94% ± 1.74%	73.20% ± 2.04%	1.05
Action Precondition Novelty	82.09% ± 2.17%	68.00% ± 2.83%	1.07
State Trigger Effect Novelty	74.85% ± 6.93%	67.60% ± 11.13%	0.98
New Entities Novelty	75.50% ± 6.38%	68.80% ± 10.93%	0.99

Table 3: Internal Evaluation Results for Agent's Performance Against Heuristic Agent with Novelty

Novelty Level 1: Class		
Metrics	Performance	Top Competitor's Performance
	Mean ± SD	Mean ± SD
M1. Percent of CDT	30.94% ± 8.16%	14.78% ± 7.21%
M2. NRP	86.49% ± 4.94%	73.65% ± 4.73%
M3. Win rate	54.97%±18.62%	54.50% ± 8.13%
Novelty Level 2: Attribute		
M1. Percent of CDT	71.89% ± 8.27%	67.67% ± 7.56%
M2. NRP	113.44%±4.30%	64.82% ± 5.42%
M3. Win rate	75.47% ± 13.9%	59.02% ± 9.42%
Novelty Level 3: Representation		
M1. Percent of CDT	23.33% ± 6.44%	9.44% ± 4.33%
M2. NRP	116.97%±3.96%	78.97% ± 4.23%
M3. Win rate	78.22% ± 2.69%	58.99% ± 6.87%

Table 4: External Evaluation

competitor across all the novelty levels. In detail, our agent performed two times better in all novelty types. Despite some limitations on novelty detection in class and representation novelty, the agent also achieved a very high NRP rate across all different settings. The results indicate that our agent can adapt to changes in Monopoly environments, especially in level 2 and level 3 novelty. Our agent win ratio was 76.48% against the baseline agent before novelties were injected. After novelties were injected, our agent performance remained relatively the same at 75.47% and 78.22% for level 2 and 3 novelty. In comparison, the win ratio of the next best agent is only around 60%. The results also suggest that we should improve novelty detection on level 1 and level 3 novelty in which our architecture only had around 30% of CDT. Overall, the evaluation results reflect our agent's capability to detect novelties accurately and adapt to those novelties to enhance the agent's performance.

Discussion

The results indicate that the MCTS agent provides outstanding solutions for the game despite the complexity of the architecture and different levels of novelty. The win rate is relatively high (above 70%) for most novelty types and different accommodation methods. Regarding novelty identification, we compose individual ASP novelty detection components for each type of novelty, e.g., the novelty of factor change when freeing a mortgaged property or the novelty of penalty change when getting out of jail. If one specific novelty is injected in the game, say the novelty of penalty change when getting out of the jail, when the action related to this novelty type is executed ($get\_out\_of\_jail$ in this case), the ASP novelty detection component composed explicitly for this novelty type will be triggered, so that we will know which type of novelty is injected.

Conclusion and Future Work

This paper presented a new agent architecture for novelty handling in a multi-agent stochastic environment that can detect, characterize, and accommodate novelties. Results from internal and external evaluations highlight the effectiveness of the architecture at handling different levels of novelty. However, the results also show some architecture limitations in novelty detection and identification.

A limitation of our architecture is that it cannot explicitly handle multiple novelties at the same time. The novelty detection mechanism may not distinguish which novelty is causing the observed changes from what is expected. Another limitation is that some novelties can only be detected if the agent performs an action or is put into a specific scenario. For example, when the agent successfully builds a hotel, the agent is most likely in a winning position and will not sell the hotel, hence any novelties related to selling properties will not be encountered. In other words, our agent does not explicitly explore the environment looking for novelties.

These limitations guide us to new approaches to improve the architecture. The first approach is probabilistic reasoning, which can help us overcome the multiple novelties limitation. For this approach, the agent can operate an add-on classification model to identify the exact novelty once the novelty is detected. Additionally, probabilistic reasoning may provide a more robust way to identify novelties instead of solely relying on logical reasoning. The second approach is to learn and explore the environment fully. This approach tackles the missing novelty limitation. This method can suggest the agent take actions that it has not taken before, and help the agent to discover new novelty. We can also explore a new direction to detect and accommodate novelty (Piotrowski and Mohan 2020; Peng, Balloch, and Riedl 2021; Li et al. 2021). While the new potential approaches may cause the agent to worsen its performance to explore the environment thoroughly, it may benefit the agent in the long run when the agent can utilize that novelty to reach the final goal.

Acknowledgements

This work was funded in part by DARPA grant W911NF-20-2-0006. We would like to thank Mayank Kejriwal, Shilpa Thomas, Hongyu Li and other members of the University of Southern California team for the Monopoly simulator.

References

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Boney, R.; Ilin, A.; Kannala, J.; and Seppanen, J. 2021. Learning to play imperfect-information games by imitating an oracle planner. *IEEE Transactions on Games*.

Boult, T. E.; Grabowicz, P. A.; Prijatelj, D. S.; Stern, R.; Holder, L. B.; Alspector, J.; Jafarzadeh, M.; Ahmad, T.; Dhamija, A. R.; Li, C.; Cruz, S.; Shrivastava, A.; Vondrick, C.; and Scheirer, W. J. 2021. Towards a unifying framework for formal theories of novelty. In *In Proceedings of the 35th AAAI Conference on Artificial Intelligence*, AAAI.

Boutilier, C. 1970. Planning, learning and coordination in multiagent decision processes. *Proceedings of the 6th Conference on Theoretical Aspects of Rationality and Knowledge*.

Brown, N., and Sandholm, T. 2019. Solving imperfect-information games via discounted regret minimization. *Proceedings of the AAAI Conference on Artificial Intelligence* 33(01):1829–1836.

Brown, N.; Sandholm, T.; and Amos, B. 2018. Depth-limited solving for imperfect-information games. In Bengio, S.; Wallach, H.; Larochelle, H.; Grauman, K.; Cesa-Bianchi, N.; and Garnett, R., eds., *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Chitnis, R.; Silver, T.; Tenenbaum, J. B.; Lozano-Pérez, T.; and Kaelbling, L. P. 2021. Learning neuro-symbolic relational transition models for bilevel planning. *ArXiv: 2105.14074*.

Choi, S.; Yeung, D.-Y.; and Zhang, N. 2000. An environment model for nonstationary reinforcement learning. In Solla, S.; Leen, T.; and Müller, K., eds., *Advances in Neural Information Processing Systems*, volume 12. MIT Press.

da Silva, B.; Basso, E.; Bazzan, A.; and Engel, P. 2006. Dealing with non-stationary environments using context detection. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, 217–224.

Ganzfried, S., and Sandholm, T. 2011. Game theory-based opponent modeling in large imperfect-information games. In *International Foundation for Autonomous Agents and Multiagent Systems*, AAMAS '11, 533–540.

Gizzi, E.; Amel, H.; Lin, W. W.; Rhea, K.; and Sinapov, J. 2021. Toward creative problem solving agents: Action discovery through behavior babbling. *2021 IEEE International Conference on Development and Learning (ICDL)* 1–7.

Goel, S.; Tatiya, G.; Scheutz, M.; and Sinapov, J. 2021. Novelgridworlds: A benchmark environment for detecting and adapting to novelties in open worlds. In *International Foundation for Autonomous Agents and Multiagent Systems*, AAMAS.

Haliem, M.; Bonjour, T.; Alsalem, A. O.; Thomas, S.; Li, H.; Aggarwal, V.; Bhargava, B.; and Kejriwal, M. 2021. Learning monopoly gameplay: A hybrid model-free deep reinforcement learning and imitation learning approach. *ArXiv: 2103.00683*.

Heinrich, J., and Silver, D. 2016. Deep reinforcement learning from self-play in imperfect-information games. *ArXiv: 1603.01121*.

Hester, T., and Stone, P. 2012. Intrinsically motivated model learning for a developing curious agent. In *AAMAS Adaptive Learning Agents (ALA) Workshop*.

Kaelbling, L. P., and Lozano-Perez, T. 2013. Integrated task and motion planning in belief space. *The International Journal of Robotics Research* 32:1194 – 1227.

Koller, D., and Pfeffer, A. 1995. Generating and solving imperfect information games. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, 1185–1192. Morgan Kaufmann Publishers Inc.

Li, R.; Hua, H.; Haslum, P.; and Renz, J. 2021. Unsupervised Novelty Characterization in Physical Environments Using Qualitative Spatial Relations. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, 454–464.

Muhammad, F.; Sarathy, V.; Tatiya, G.; Goel, S.; Gyawali, S.; Guaman, M.; Sinapov, J.; and Scheutz, M. 2021. A novelty-centric agent architecture for changing worlds. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS.

Nash, J. 1951. Non-cooperative games. *Annals of Mathematics* 54(2):286–295.

Neary, C.; Verginis, C. K.; Cubuktepe, M.; and Topcu, U. 2021. Verifiable and compositional reinforcement learning systems. *ArXiv: 2106.05864*.

Padakandla, S.; K. J., P.; and Bhatnagar, S. 2020. Reinforcement learning algorithm for non-stationary environments. *Applied Intelligence* 50(11):3590–3606.

Peng, X.; Balloch, J. C.; and Riedl, M. O. 2021. Detecting and adapting to novelty in games. *ArXiv: 2106.02204*.

Pinto, V.; Xue, C.; Gamage, C. N.; and Renz, J. 2021. The difficulty of novelty detection in open-world physical domains: An application to angry birds. *arXiv preprint arXiv:2106.08670*.

Piotrowski, W. M., and Mohan, S. 2020. Model-based novelty adaptation for Open-world AI. In *Proceedings of the 31st International Workshop on Principles of Diagnosis 2020*.

Sarathy, V.; Kasenberg, D.; Goel, S.; Sinapov, J.; and Scheutz, M. 2021. SPOTTER: Extending symbolic planning operators through targeted reinforcement learning. In *International Foundation for Autonomous Agents and Multiagent Systems*, AAMAS.

Schermerhorn, P.; Kramer, J.; Brick, T.; Anderson, D.; Dingler, A.; and Scheutz, M. 2007. DIARC: A testbed for natural human-robot interaction. In *Mobile Robot Competition and Exhibition - Papers from the 2006 AAAI Workshop,*

Technical Report, AAAI Workshop - Technical Report, 45–52.

Scheutz, M.; Williams, T.; Krause, E.; Oosterveld, B.; Sarathy, V.; and Frasca, T. 2019. *An Overview of the Distributed Integrated Cognition Affect and Reflection DIARC Architecture*. 165–193.

Silver, D.; Huang, A.; Maddison, C.; Guez, A.; Sifre, L.; Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529:484–489.

Wanyana, T., and Moodley, D. 2021. *An Agent Architecture for Knowledge Discovery and Evolution*. 241–256.

Xie, A.; Harrison, J.; and Finn, C. 2020. Deep reinforcement learning amidst lifelong non-stationarity. *arXiv: 2006.10701*.