Petri Nets and Hierarchical Reinforcement Learning for Personalized Student Assistance in Serious Games

Ryan Hare
Dept. of Electrical and Computer Engineering
Rowan University
Glassboro, NJ, USA
harer6@rowan.edu

Ying Tang*
Dept. of Electrical and Computer Engineering
Rowan University
Glassboro, NJ, USA
tang@rowan.edu

2022 International Conference on Cyber-Physical Social Intelligence (ICCSI) | 978-1-6654-9835-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCSI55536.2022.9970680

***Abstract*—Adaptive serious games offer a new frontier for education, especially in complex topics. However, optimal methods for in-game adaptation are still being explored to address challenges such as limited educator resources, unpredictable or limited data, or complicated implementation procedures. This work offers an adaptable framework for personalized student assistance and directing within an adaptive serious game using reinforcement learning and Petri nets. Our proposed framework can be built upon by serious game developers and researchers to create adaptive serious games for improving student learning in other domains. Building on prior work, we address the challenge of adaptive in-game content through Petri net player modelling and a multi-agent deep reinforcement learning approach to gradually learn optimal personalized assistance. Finally, we provide proof-of-concept training performance for our proposed agent using a student simulation, demonstrating that the proposed hierarchical reinforcement learning approach offers significantly (effect size r = 0.8101) improved training performance over a tabular, single-agent approach.**

I. Introduction

As higher education becomes more complex, students will inevitably run into courses that they find confusing or boring, impairing their learning process. Furthermore, strain on educator time and resources compounds the issue, leading to students falling behind or failing due to lack of support tailored to their ideal style of learning. Adaptive educational systems offer one solution to this issue. By modelling the student's current mental state, ideal learning style, or other relevant aspects, adaptive educational systems seek to provide personalized, just-in-time support based on a student's current needs or misunderstandings.

A key challenge in creating adaptive educational systems for classroom use is lack of usable data. It is difficult to create and populate an accurate student model without sufficient data on a student's performance. However, by combining an adaptive educational system into a serious game or virtual learning environment, any and all data about that environment is made freely available. Furthermore, the environment is highly controllable, allowing students to learn and explore at their own pace. With serious games (SGs), there is the added benefit of the game aspect, which simultaneously entertains players while educating them.

Adaptive serious games (ASGs) extend the educational experience by integrating an adaptive educational system into a virtual environment, coupling adaptive educational support with an entertaining and engaging learning environment. Recent trends in educational technology continue to push for ASGs as a solution to limited instructor resources and unengaging course content [1]–[3]. Using the built-in adaptive educational system, ASGs adjust game content, game difficulty, virtual environments, or other aspects to optimize a student's learning. Furthermore, by "gamifying" the learning experience, ASGs further benefit students in terms of lesson enjoyment and engagement [4].

In prior work, we designed a serious game called *Gridlock*, a domain-specific game dealing with sequential logic design. *Gridlock* was integrated with a proposed Petri net-based student model that tracked and controlled player performance and player movement [5] to systematically model the evolution of students' knowledge. We then extended the system's ability to provide personalized assistance through reinforcement learning (RL), allowing it to provide automated support. While our approach was successful in modelling and controlling player movement, the method's ability to provide personalized instructional support was limited due to data availability, convergence issues, and the large size of the learning space.

To expand upon our prior work and deal with the exponentially increasing size of our learning space, we propose combining a Petri net-based game model and a hierarchical Markov decision process (MDP). Our general-purpose Petri net model focuses on recording and controlling player movement through a given serious game, including personalized support, while the MDP helps us to predict a player's performance as a function of the provided educational support. We then subdivide the overall learning space into a hierarchical MDP, greatly decreasing the size of the learning space and reducing data requirements. With RL, the system can automatically learn associations between student performance and personalized support, allowing it to provide optimal student support without external intervention. The end result is a general-purpose, modular system for integrating personalized support and player tracking into a serious game.

Due to difficulty in gathering sufficient real-classroom data, we verify system performance through student simulations informed by in-classroom observations. By testing the completed system on these student simulations, we were able to adjust system hyperparameters and environmental variables to improve system performance and verify that our proposed method provides more efficient learning compared to a single MDP approach, especially in the early stages of training when data are sparse and limited. To that end, Section II overviews our proposed Petri net structure. Section III formalizes our hierarchical MDP structure and combined Petri net/RL approach, including definitions of our learning space. Section IV shows our simulated results, followed by conclusions in Section V.

II. Agent-driven Petri Nets

A Petri net (PN) is a graphical model designed for modelling discrete event systems [6]. Visually, a PN can be drawn as a directed graph composed of places (circles) and transitions (bars), with directed arcs connecting from place to transition or vice versa.

This work was supported in part by the National Science Foundation under Grant 1913809 and by the U.S. Department of Education Graduate Assistance in Areas of National Need (GAANN) Grant Number P200A180055.

At any given time step t, a place can contain a number of tokens, which "flow" through the Petri net whenever a transition is fired. By representing the player as a token, a PN can easily model player movement through a given serious game. With a Petri net model, we create a generic structure for an adaptive serious game composed of N "learning blocks". Each learning block pertains to a different subset of knowledge to deliver to students, complete with hints, study materials, or other easily personalized assistance.

In order to properly integrate a Petri net with reinforcement learning (RL), we adapt extended Petri nets from our prior work [5] into an agent-driven Petri net (APN). The main focus of the APN is to integrate a PN with one or more RL agents. Specifically, we use one 'high-level' agent that deals with a student's overall movement through the game, and N 'low-level' agents, each of which deals with personalized assistance in one of the N learning blocks. Throughout the paper, h and l in superscript will be used to refer to the high- and low-level agents, respectively (for example, X^h and X^l). With that in mind, the full definition of an APN is as a six-tuple in the form $\langle P, \Upsilon, I, O, M, \delta \rangle$:

- P is a set of places that compose the Petri net. In our extended APN, P is divided into two subsets of places, $P = \{P_g, P_s\}$:
  1) P_g, where each place $p_g \in P_g$ represents a location in or state of the game.
  2) P_s, where each place $p_s \in P_s$ represents a storage place for instances of personalized support.
- Υ is a set of transitions that connect places in the Petri net. In our extended APN, Υ is divided into $N+3$ subsets, 3 subsets for grading, system-driven, and high-level agent-driven transitions, and N subsets for low-level agent-driven transitions in each of the N learning blocks. Thus, $\Upsilon = \{\Upsilon_g, \Upsilon_s, \Upsilon_d^h, \Upsilon_{dn}^l : n \in [1, N]\}$:
  1) Υ_g, where each transition $\upsilon_g \in \Upsilon_g$ represents a grading transition which, upon firing, will update one or more pieces of internal data in the relevant student marker.
  2) Υ_s, where each transition $\upsilon_s \in \Upsilon_s$ represents player- or system-driven movement of the player from one place to another.
  3) Υ_d^h, where each transition $\upsilon_d^h \in \Upsilon_d^h$ represents a high-level agent-driven transition. The number of high-level agent-driven transitions is determined by the number of learning blocks, but they are all encapsulated in a single set.
  4) Υ_{dn}^l represents low-level agent-driven transitions in learning block n. Each transition $\upsilon_{dn}^l \in \Upsilon_{dn}^l$ represents a specific low-level agent-driven transition in learning block n.
- I is an input function, defining all arcs that connect from a place $p \in P$ to a transition $\upsilon \in \Upsilon$.
- O is an output function, defining all arcs that connect from a transition $\upsilon \in \Upsilon$ to a place $p \in P$.
- M is a marker function, defining the number of markers as $M(p) = \mathbb{R} \forall p \in P$. In our extended APN, M also contains student data, defined as:
  - In addition to a single value for each place, the marker stores student data $M(p) = \{\mathbb{R}, \Omega\}$, where Ω is a set of student data. For non-student markers, $M(p) = \mathbb{R}$.
- δ is unique to the extended APN, and defines a set of transition probabilities partitioned into $N + 1$ subsets, 1 for the high-level agent-driven transitions, and N for low-level agent-driven transitions in each of the N learning blocks. δ, then, is defined as $\delta = \{\delta^h, \delta_n^l : n \in [1, N]\}$:
  1) δ^h, which is defined for each transition $\upsilon_d^h \in \Upsilon_d^h$, and is mapped to an high-level RL agent.
  2) δ_n^l, which is defined for each transition $\upsilon_{dn}^l \in \Upsilon_{dn}^l$, and is mapped to the nth reinforcement learning agent.

As stated, the structure assumes a game divided into N so-called subject-specific blocks, each of which focuses on a subset of the overall knowledge that the game intends to deliver. Using the grading transitions, a player is tested, both to establish an initial benchmark and to assess the player's specific knowledge in each learning block. The high-level agent-driven transitions allow the system to dynamically select what blocks to visit, and the low-level agent-driven transitions provide students with personalized support inside the learning blocks.

We focus on integrating reinforcement learning (RL) with the APN. Since RL involves probabilistic decision-making, the δ function allows us to map RL decision-making directly to transition probabilities in the APN. Additionally, we store student feature vectors in markers in the net, allowing student data to "flow" through the net as the student moves. In our definition, Ω is a set of student feature vectors in one instance of the game (for one student). $\Omega = \{\lambda_n, \hat{\omega}_n : n \in [1, N]\}$ such that $\hat{\omega}_n$ is the feature vector for learning block N. Additionally, Ω also contains N grade values, $\lambda_n, n \in [1, N]$, which are numerical "grades" assigned to each feature vector. This "grading" is discussed in greater detail in Section III-A in our discussion on agent states.

III. Hierarchical MDPs and Deep RL

A. Hierarchical Markov Decision Processes

The basic principle of reinforcement learning (RL) is to allow some intelligent agent to interact with an external environment [7]. At any given time step, t, an RL agent observes the environment's current numerical state, s_t, and chooses an action, a_t, from a pool of possible actions. Based on the resulting new state, s_{t+1}, the agent is provided a numerical reward, r_{t+1}, determined from a reward function. The agent then gradually learns what actions to take to maximize obtained reward. The entire flow is called a transition, and a transition can be represented as a 4-tuple, $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$.

In this implementation, we mainly focus on the combination of Petri nets and reinforcement learning, applying standard RL methods to implement intelligent agents in our game system. With our approach, we use a hierarchical MDP as a method to subdivide the state space and allow for quicker and more accurate agent learning. Other methods to solve this problem can be found in more depth in the space of multi-agent reinforcement learning [8]–[11], which inspired our more simplified implementation. In multi-agent reinforcement learning, agents often work cooperatively to complete a shared task. In our hierarchical MDP, the agents instead learn separately on individual subtasks all with the shared goal of providing student support.

This sequential model for reinforcement learning problems is known as a Markov decision process (MDP). MDPs assume the Markov property, which states that future states only depend on the current state. The MDP is a 6-tuple in the form $\langle S, A, R, \phi, \gamma, \pi \rangle$, which we extend as stated here into a hierarchical MDP:

- S is the set of all possible states. At time step t, the agent observes state $s_t \in S$, selects an action, and arrives at a new state $s_{t+1} \in S$. In the hierarchical MDP, the state space is a set of $N+1$ state spaces, 1 state space for the high-level agent, and N state spaces for N low-level agents (one for each learning block). Thus, $S = \{S^h, S_n^l : n \in [1, N]\}$:

1) S^h is the high-level state space.
2) S^l_n are N low-level state spaces that each deal with learning block $n \in [1, N]$.

- A is the set of all possible actions. The agent's chosen action at time step t is denoted $a_t \in A$. In the hierarchical MDP, A is a set of $N+1$ action spaces, 1 for the high-level agent and N for the low-level agents. To summarize, $A = \{A^h, A^l_n : n \in [1, N]\}$:
  1) A^h is the high-level action space that directly maps to transition set Υ^h_d in the APN. For example, Action $a^h \in A^h$ directly maps to transition $\upsilon^h_d \in \Upsilon^h_d$.
  2) A^l_n are N low-level action spaces that each deal with learning block $n \in [1, N]$ and each map to transition set Υ^l_{dn} in the APN. For example, action $a^l_n \in A^l_n$ directly maps to transition $\upsilon^l_{dn} \in \Upsilon^l_{dn}$.
- R is the reward function determining what reward the agent receives. After firing action a_t and arriving at state s_{t+1}, the agent receives reward r_t as determined by R. In the hierarchical MDP, R is a set of reward functions, where $R = \{R^h, R^l\}$:
  1) R^h is the high-level reward function, used to reward the high-level agent.
  2) R^l is the low-level reward function, used to reward all low-level agents.
- $\phi(s_t, a_t, s_{t+1})$ is the transition probabilities that create a mapping from state to state based on the chosen actions. In the hierarchical MDP, ϕ is a set of transition probabilities, where $\phi = \{\phi^h, \phi^l_n : n \in [1, N]\}$ for the high- and low-level state spaces.
- γ is the discount factor in range $(0, 1)$. The discount factor is used to weigh future rewards, determining how near- or far-sighted the agent is. When $\gamma = 1$, the agent will weigh all rewards equally, and when $\gamma = 0$, the agent only considers possible reward for the next time step. To vary agent behavior between high- and low-level, the discount factor in the hierarchical MDP, $\gamma = \{\gamma^h, \gamma^l\}$, where:
  1) γ^h is the high-level discount factor.
  2) γ^l is the low-level discount factor.
- π is the agent's policy, with $\pi(s, a)$ determining the probability of selecting action $a \in A$ given that state $s \in S$ is observed. $\pi(s, a) \in [0, 1] \forall s \in S, a \in A$. In the hierarchical MDP, the policy π is a set of $N + 1$ policies, 1 policy for the high-level agent, and N policies for the N low-level agents. Thus, $\pi = \{\pi^h, \pi^l_n : n \in [1, N]\}$:
  1) π^h is the high-level agent's policy, creating a probability distribution over S^h and A^h.
  2) π^l_n are the low-level agent policies for each of n agents, each creating a distribution over S^l_n and A^l_n for $n \in [1, N]$.

Using this extended hierarchical MDP, agents can be appropriately configured to work with the agent-driven Petri net in an adaptive serious game. To provide more detail about the reasoning behind the extension into a hierarchical MDP, we now give a detailed explanation of the configurations for states, actions, reward functions, and agent policies in terms of applying the proposed system in an adaptive serious game.

1) State: Initial student data are gathered with a baseline quiz to establish students' starting level of knowledge. As stated, for each of the N subject-specific blocks, each student is assigned a set of feature vectors $\Omega = \{\lambda_n, \hat{\omega}_n : n \in [1, N]\}$ such that $\hat{\omega}_n$ is the student's feature vector for learning block n and λ_n are numerical "grades" assigned to each feature vector.

For the high-level agent, the state space could be represented as the set of N feature vectors, Ω. However, this state space would be exceedingly large, so we apply a "grading" process to obtain a set of integer values in place of the set of feature vectors. The grading process uses the grading function, $\Lambda(\hat{\omega})$, to obtain an integer value, λ_n. By representing the high-level agent's state space as the set of grades, $\{\lambda_n : n \in [1, N]\}$, we reduce the size of the high-level agent's state space significantly. The grading function shown in Equation 1 can be thought of as a generic translation of a feature vector $\hat{\omega}_n$ into an integer value λ_n.

$$\Lambda(\hat{\omega}_n) = \lambda_n \tag{1}$$

For the low-level agent, the student feature vectors can be directly translated into the state values used by the RL agent to make decisions. The state space for low-level agent $n \in [1, N]$ is then defined by feature vector $\hat{\omega}_n$. For the actual values used in the feature vectors, they could represent score on questions, time taken to complete sections, emotional indicators transformed into numerical values, or any other number of relevant numerical factors indicative of student performance.

2) Action: The high-level agent makes broad decisions about the student's game route, deciding which subject-specific blocks to visit. These actions directly correspond to each transition $\upsilon^h_d \in \Upsilon^h_d$ in the APN.

The low-level agent receives more specific information to make better decisions about student assistance. Like the high-level agent, the available actions for low-level agent n directly correspond to the transitions $\upsilon^l_{dn} \in \Upsilon^l_{dn}$, where $n \in [1, N]$.

3) Reward Functions: The agent's reward structure is highly important since the goal of the agent is to maximize reward. In other words, reward structure determines final agent behaviour. In our proposed reward structure, we focused on minimizing total game completion time and maximizing student performance.

The high-level agent's rewards are based on the following rules: 1) avoid entering blocks in which a student has already proven mastery; and 2) minimize total number of blocks entered. As such, the high-level agent's step-by-step reward can be derived from changes in student grades, $\{\Delta\lambda_n : n \in [1, N]\}$. At the end of the game, the agent can also be provided some large positive reward, σ^h, reduced based on the total number of time steps in which the high-level agent acted.

The low-level agent's rewards are based on the following rules: 1) improve the student's performance indicators given from the feature vector; and 2) enable the student to reach content mastery in as few time steps as possible. Thus, step-by-step reward within block n can be determined as the percentage change in feature vector $\hat{\omega}_n$ that was triggered by the prior help action. Upon finishing a learning block, the low-level agent can also be provided a large positive reward, σ^l, reduced by the number of steps taken to complete the relevant learning block.

4) Agent Policies: Finally, to translate policies into transition probabilities, the agents must first generate policies. Agent policies are determined by $\Pi \rightarrow \pi$, which assigns a probability distribution based on past observed rewards. The specifics of the distribution Π is determined by implementation. π^h, the high-level agent's policy, can then be directly mapped to δ^h in the agent-driven Petri net (APN). Likewise, π^l_n for $n \in [1, N]$ can each be mapped to δ^l_n in the APN.

B. Solving MDPs with Deep Reinforcement Learning

Q-learning is a tabular method that tracks and updates a Q-function [12]. The Q-function, $Q(s, a)$, is a function that determines the

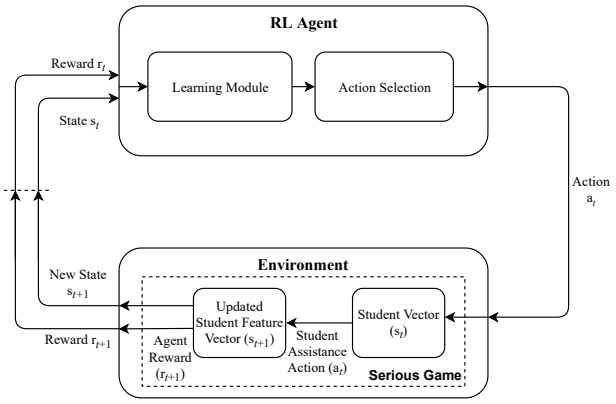

Fig. 1. The standard cycle of learning for an RL agent.

agent's expected reward when selecting an action $a \in A$ while observing a state $s \in S$. By looking at the expected reward values in this function, the agent can know which action will provide the best expected reward [12]. The agent also applies a discount factor, $\gamma \in [0, 1]$, to weigh future rewards. Whenever a new transition is observed at time step t, the Q-function is updated as per Equation 2.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + (r_{t+1} + \gamma \max_a Q(s_{t+1}, a)) \quad (2)$$

The main issues we address from our prior work are the large learning space, lack of generalization between similar states, mediocre early performance, and long convergence times we faced when using a tabular single-agent approach [5]. Further, we hope to extend the system's data efficiency and improve system performance when faced with a small amount of real-world data. We address these issues by extending the tabular single-agent method into a multi-agent hierarchical approach using deep Q-learning, an extended form of Q-learning that uses a neural network to predict $Q(s, a)$ [13]. Whenever the agent observes a new transition, the neural network is retrained, adjusting the internal network weights θ to minimize the difference between the network's estimated reward and the actual observed reward plus discounted future reward, as estimated by the network. The difference is also called the cost function, and is shown in Equation 3.

$$Cost = Q(s_t, a_t; \theta_t) - (r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \vartheta_t) \quad (3)$$

As shown in Equation 3, a second neural network with weights ϑ is used for future reward predictions. This network is an exact copy of the main network with different weights. This method is called double deep Q-learning [13]. Using a copy of the main network with weights ϑ for future predictions helps to stabilize the main network during training. Every ρ time steps, the copy network weights ϑ are updated to reflect the main network weights θ.

Another optimization for deep Q-learning is experience replay. By storing a set of past transition tuples, $\Psi = \{\langle s_t, a_t, r_{t+1}, s_{t+1}\rangle\}$, the agent can randomly sample from the pool whenever new training is needed. Using experience replay improves data efficiency of the network since past experience is reused. It also breaks any correlations between data, as would occur when observing a concurrent sequence of transitions. Finally, experience replay helps prevent the agents from "forgetting" past information by periodically "reminding" the agent what it has seen in the past.

Agent-driven Petri Net Operation with RL

Require: Set $Z = \{p_0\}$ where p_0 is the place $\in P$ where all player markers are initialized
while $(Z \neq \emptyset)$ **do**
  for each place $p \in Z$ **do**
    $H(p) = \emptyset$
    if $\upsilon \in \Upsilon_s \forall \upsilon \in I(p)$ **then**
      Fire υ and deposit token into $O(\upsilon)$
    else if $\upsilon \in \Upsilon_g \forall \upsilon \in I(p)$ **then**
      Record updated $\hat{\omega}_n^*$ vector(s)
      Populate updated Ω^* using Equation 1
      Calculate r_{t+1} from R^h
      Call Algorithm III-B on last agent to act with transition tuple $\langle \Omega, a_t, r_{t+1}, \Omega^* \rangle$
      Fire υ and deposit token into $O(\upsilon)$
    else if $\upsilon \in \Upsilon_d^h \forall \upsilon \in I(p)$ **then**
      Observe high-level state, $s_t^h = \{\lambda_n : n \in [1, N]\}$ from Ω
      Update high-level policy, $\pi^h(s_t^h)$ from Π
      Update $\delta^h \leftarrow \pi^h(s_t^h)$
      Fire available transition υ based on probabilities δ^h and deposit token into $O(t)$
    else if $\upsilon \in \Upsilon_{dn}^l \forall \upsilon \in I(p)$ **then**
      Observe low-level state for agent n, $s_t^l = \hat{\omega}_n \in \Omega$
      Update low-level policy for agent n, $\pi_n^l(s_t^l)$ from Π
      Update $\delta_n^l \leftarrow \pi_n^l(s_t^l)$
      Fire available transition υ based on probabilities δ^l and deposit token into $O(t)$
    end if
    $Z = Z \cup \{O(\upsilon) \cap P\}$
  end for
  $Z = Z - \{p\}$
end while

Deep Q-learning Q-function Update

Require: New transition tuple $\psi_t = \langle s_t, a_t, r_{t+1}, s_{t+1}\rangle$
Replay memory Ψ
Batch size b
Target agent
Append $\psi_t \rightarrow \Psi$
Sample batch of b transitions from Ψ
for Each transition $\psi_x \in$ batch **do**
  Predict $Q(s_{t+1}, a; \vartheta_t)$ from copy network
  Using ψ_x and $Q(s_{t+1}, a)$, adjust θ_t to minimize Equation 3
end for

C. Deep Q-learning for Hierarchical MDPs

In a hierarchical Markov decision process (MDP), deep Q-learning is extended into a multi-agent configuration [14]–[16]. In the proposed system, the game makes use of a high-level agent that controls student movement from block to block and N low-level agents that each control student assistance inside an assigned learning block. By splitting the problem to a multi-agent approach, the overall size of the learning space is greatly decreased for all agents, allowing the agents to better learn optimal policies with a lower data requirement.

The key connection between the deep Q-learning and the agent-driven Petri net (APN) is in the APN's transition probabilities function, δ. As stated, this function assigns probabilities to firing every available transition from a given place. By mapping δ to the

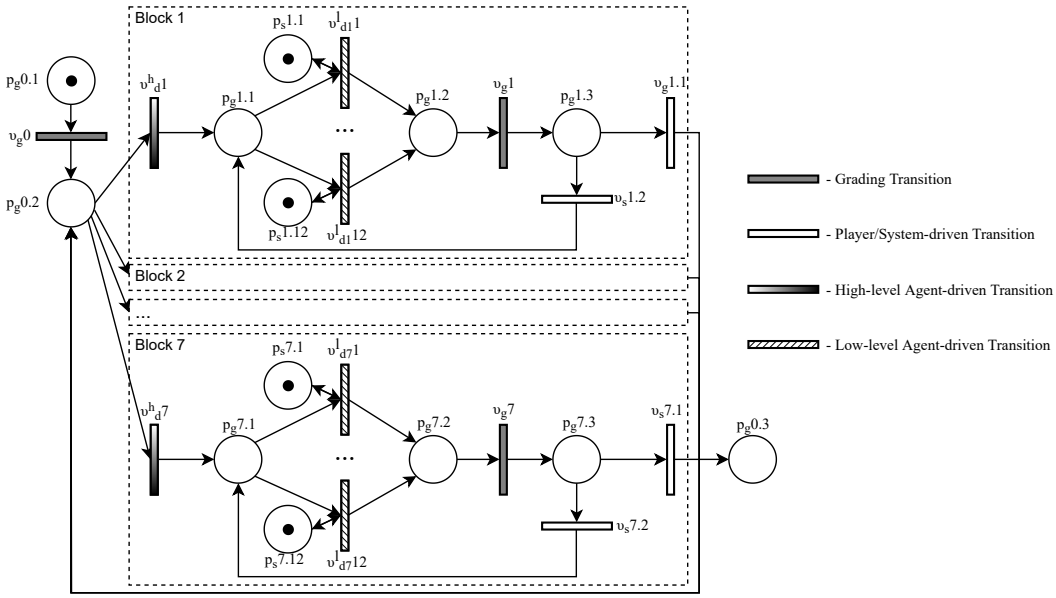

Fig. 2. APN structure in *Gridlock*.

RL agent's policy, π, the RL agent's actions are therefor assigned directly to the APN's transitions.

Algorithm III-B shows the full adaptive optimization process. In this process, transitions are fired when available based on player marker positioning. Depending on the type of transition, the system observes different results, including updated feature vectors in the APN, stored knowledge in the reinforcement learning replay memory, Q-learning updates, and updated transition probabilities.

IV. Experimental Results

A. Parameters and Network Layout

For testing, we integrated our proposed system into a serious game called *Gridlock*. Gridlock is a domain-specific serious game intended for intro-level university students, designed to instruct them on the basics of digital logic and digital circuit design. In Gridlock, $N = 7$ is the number of blocks and the size of the high-level state vector while $|\hat{\omega}_n| = 15$ is the size of all low-level state vectors. For the structure of the agent-driven Petri net (APN) in *Gridlock*, refer to Figure 2. As shown, the agent-driven transitions allow the student marker to be moved into any of the learning blocks. Once in a learning block, the student marker enables all help transitions, allowing the low-level agents to decide what help transition to fire, ultimately providing assistance to the student.

For our RL agent exploration policy, we determined agent policy using Boltzmann exploration [17], as shown in Equation 4. Both exploration factor, ϵ, and the Boltzmann temperature values, τ^h and τ^l, were determined experimentally. Finally, reward functions were determined as defined in Section III-A, with ω^h and ω^l determined experimentally as well.

$$\pi(s_t, a) = \left(\frac{e^{\frac{Q(s_t,a)}{\tau}}}{\sum_{i=1}^{|A|} e^{\frac{Q(s_t,a_i)}{\tau}}} \right) \forall a \in A \tag{4}$$

All parameter values used in testing are shown in Table I. Some parameters were gradually changed over the course of simulated agent training, and are indicated by an arrow pointing from the initial value to the final value.

TABLE I
Network and Agent Hyperparameters

Name	Description	Value
σ^h	High-level completion reward	50
σ^l	Low-level completion reward	0
α	Neural network learning rate	0.0001
γ^h	High-level discount factor	0.85
γ^l	Low-level discount factor	0.85
ϵ	Exploration factor	$0.6 \rightarrow 1.0$
ρ	Update target network every	5 steps
b	Training batch size	200
τ^h	High-level agent Boltzmann temperature	200
τ^l	Low-level agent Boltzmann temperature	8
$\max \lvert\Psi\rvert$	Max replay memory size	50000
$\min \lvert\Psi\rvert$	Min replay memory size for training	100

For our simulated student testing, our hierarchical deep Q-learning agents and simulation setup were coded in Python [18] using Keras [19] to create the deep Q-networks for the high- and low-level agents. The networks used had an input layer with the size of the state vector (7 for high-level, 15 for low-level), two dense layers with the relu activation function with 60 and 40 neurons, respectively, and then a dense output layer sized to the number of possible actions ($|A^h| = 7$ for high-level, $|A^l_n| = 12$ for all low-level) with the relu activation function. Finally, for our grading function, Λ, we used a trained random forest classifier to classify student feature vectors into integer grade values in the range $[1, 3]$.

B. Results

Initial results focused on early performance of a single-agent deep Q-learning approach compared to the proposed hierarchical agent structure. We compare the cumulative reward obtained by both agents per episode (one instance of the game, from start to finish) over 5000 episodes. We also compare the rolling average of the cumulative reward, with both shown in Figure 3.

By observation, the hierarchical approach has visibly improved average reward in the early stages of training. By the Mann-Whitney

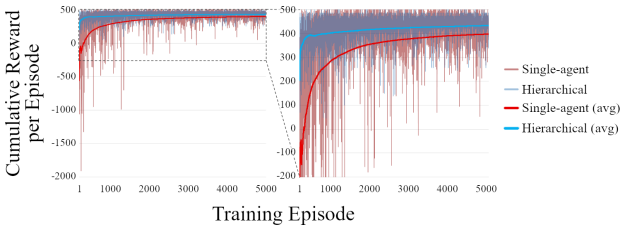

Fig. 3. Single-agent RL vs hierarchical RL cumulative reward obtained per episode.

U test, the cumulative reward showed a significant improvement in the hierarchical agent's performance with an effect size $r = 0.8101$. Figure 4 compares the number of steps taken to complete each episode for the single-agent and hierarchical approach, as well as the rolling average of the number of steps taken. For the hierarchical agent, number of steps taken is cumulative over the high- and low-level agents. In step size, the hierarchical agent showed a similar level of improvement with an effect size $r = 0.6113$.

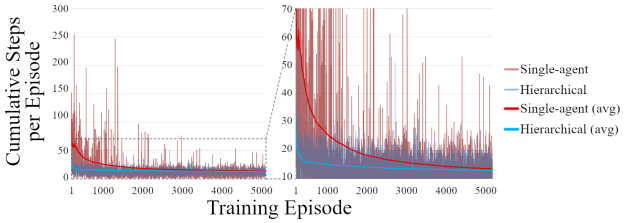

Fig. 4. Single-agent RL vs hierarchical RL total steps taken per episode until completion.

This preliminary testing shows the improvements made by the proposed method over the method used in our prior work. The new multi-agent approach with deep Q-learning allows for the agent to perform better in the early stages of training, demonstrating that, at least for simulated students, the system can be effective for student assistance with small amounts of data.

V. Conclusion

Our proposed work focuses on developing a general-purpose RL system to provide automated, adaptive content in an educational serious game. We also put a heavy focus on operating in low-data environments. To that end, our proposed method extends our prior work with learning-embedded attributed Petri nets to add a hierarchical Markov decision process. We also adapt deep Q-learning as a reinforcement learning approach to optimize the aforementioned Markov decision process. All these methods combine to create a modular and extensible model for adaptive serious games with automated learning optimization and high data efficiency. Through a student simulation, we show a proof-of-concept of the proposed system on an existing game. We demonstrate that an agent using the proposed method achieves better average reward and more efficient operation in terms of steps taken to reach episode completion. Further, the proposed method provides improved performance in the early stages of training and more consistent performance overall.

To expand upon this research, we intend to focus on collecting classroom data for system verification. Other future areas of exploration for this work we will explore are more advanced methods for RL to further improve agent performance such as learning from expert-generated policies. Other emerging methods such as transfer learning and more advanced human-like simulations could also serve to further improve data efficiency and lead to overall better adaptive system performance.

References

[1] L. Shoukry, S. Göbel, and R. Steinmetz, "Learning analytics and serious games: Trends and considerations," in *Proceedings of the 2014 ACM International Workshop on Serious Games*, ser. SeriousGames '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 21–26. [Online]. Available: https://doi.org/10.1145/2656719.2656729

[2] D. Hooshyar, M. Yousefi, and H. Lim, "A systematic review of data-driven approaches in player modeling of educational games," *Artificial Intelligence Review*, vol. 52, no. 3, pp. 1997–2017, Oct 2019. [Online]. Available: https://doi.org/10.1007/s10462-017-9609-8

[3] K. Chrysafiadi and M. Virvou, "Student modeling approaches: A literature review for the last decade," *Expert Systems with Applications*, vol. 40, no. 11, pp. 4715–4729, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095741741300122X

[4] J. Krath, L. Schürmann, and H. F. von Korflesch, "Revealing the theoretical basis of gamification: A systematic review and analysis of theory in research on gamification, serious games and game-based learning," *Computers in Human Behavior*, vol. 125, p. 106963, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0747563221002867

[5] J. Liang, Y. Tang, R. Hare, B. Wu, and F.-Y. Wang, "A learning-embedded attributed petri net to optimize student learning in a serious game," *IEEE Transactions on Computational Social Systems*, pp. 1–9, 2021.

[6] R. Zurawski and M. Zhou, "Petri nets and industrial applications: A tutorial," *IEEE Transactions on Industrial Electronics*, vol. 41, no. 6, pp. 567–583, 1994.

[7] C. Watkins, "Learning from delayed rewards," 01 1989.

[8] T. Rashid, M. Samvelyan, C. Schroeder, G. Farquhar, J. Foerster, and S. Whiteson, "QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 4295–4304. [Online]. Available: https://proceedings.mlr.press/v80/rashid18a.html

[9] J. Zhao, Y. Zhao, W. Wang, M. Yang, X. Hu, W. Zhou, J. Hao, and H. Li, "Coach-assisted multi-agent reinforcement learning framework for unexpected crashed agents," 2022. [Online]. Available: https://arxiv.org/abs/2203.08454

[10] Y. Yang, R. Luo, M. Li, M. Zhou, W. Zhang, and J. Wang, "Mean field multi-agent reinforcement learning," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 5571–5580. [Online]. Available: https://proceedings.mlr.press/v80/yang18d.html

[11] Z. Zhou and G. Liu, "Romfac: A robust mean-field actor-critic reinforcement learning against adversarial perturbations on states," 2022. [Online]. Available: https://arxiv.org/abs/2205.07229

[12] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992698

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[14] M. Brittain and P. Wei, "Hierarchical reinforcement learning with deep nested agents," *CoRR*, vol. abs/1805.07008, 2018. [Online]. Available: http://arxiv.org/abs/1805.07008

[15] T. G. Dietterich, "Hierarchical reinforcement learning with the MAXQ value function decomposition," *CoRR*, vol. cs.LG/9905014, 1999. [Online]. Available: https://arxiv.org/abs/cs/9905014

[16] P. Dayan and G. E. Hinton, "Feudal reinforcement learning," in *Advances in Neural Information Processing Systems*, S. Hanson, J. Cowan, and C. Giles, Eds., vol. 5. Morgan-Kaufmann, 1993. [Online]. Available: https://proceedings.neurips.cc/paper/1992/file/d14220ee66aeec73c49038385428ec4c-Paper.pdf

[17] N. Cesa-Bianchi, C. Gentile, G. Lugosi, and G. Neu, "Boltzmann exploration done right," 2017.

[18] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[19] F. Chollet *et al.*, "Keras," https://keras.io, 2015.