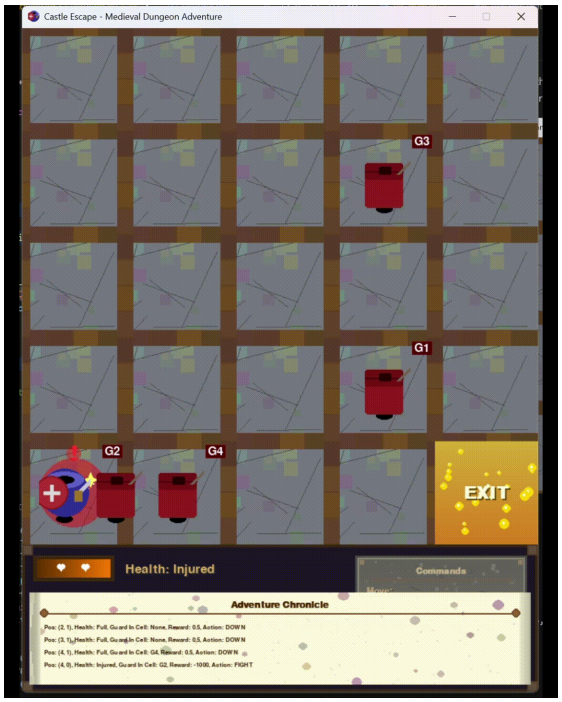
Castle Escape: Project Writeup

Introduction

Castle Escape is a medieval-themed reinforcement learning project that simulates a dungeon escape scenario. The project combines elements of game design, artificial intelligence, and reinforcement learning to create an environment where agents can learn to navigate a complex, stochastic environment with multiple risk factors. This write-up explores the development process, the technical implementation, and the results of training reinforcement learning agents in this custom environment.



Project Motivation

The primary motivation behind Castle Escape was to design a reinforcement learning environment that:

1. Presents complex decision-making challenges (fight vs. hide vs. navigate)
2. Incorporates varying levels of risk and reward
3. Features stochastic mechanics that require adaptive strategies
4. Provides a visually engaging way to observe agent learning

The medieval dungeon theme offers an intuitive framing for these challenges. When selecting actions, the player must consider both long-term goals (reaching the exit) and immediate dangers (encounters with guards).

Technical Implementation

Environment Design

The Castle Escape environment is built on the OpenAI Gym framework, allowing for standardized interaction between learning agents and the game world. The core environment consists of:

- A 5×5 grid representing castle rooms
- A player agent with three health states (Full, Injured, Critical)
- Four guard NPCs with unique attributes (strength and keenness)
- Six possible actions (UP, DOWN, LEFT, RIGHT, FIGHT, HIDE)
- Stochastic outcomes for movement, combat, and hiding attempts

The environment is implemented in `mdp_gym.py` as the `CastleEscapeEnv` class, which handles state transitions, reward calculations, and terminal state detection.

Visualization Layer

To provide an intuitive representation of the environment, a visualization layer was developed using Pygame. This layer, implemented in `vis_gym.py`, includes:

- Medieval-themed sprites and textures
- Dynamic health indicators
- Action console to display recent events
- Victory and defeat screens
- Animation effects for combat and movement

The visualization layer serves two purposes: enabling human players to interact with the environment and providing a way to observe the behavior of trained agents.

Reward Structure:

Standard Q-learning

The reward structure in the standard Q-learning implementation (`Q_learning.py`) relies entirely on the environment's native rewards without modification. It processes direct rewards from the environment (**+10000** for reaching the goal, **+10** for winning combat, **-1000** for losing combat/defeat) and incorporates them into Q-value updates using the standard Bellman

equation. This approach creates a sparse reward landscape where meaningful feedback is only received upon significant events (reaching the goal or combat outcomes). The absence of intermediate rewards requires the agent to explore extensively to discover successful paths, leading to slower learning that depends heavily on random exploration to discover the goal state's high reward.

Advanced Q-learning Reward Structure

The advanced Q-learning implementation (`Advanced_Q_learning.py`) enhances the reward structure with strategic shaping. Beyond the environment's native rewards, it implements additional reward signals: a bonus of **+0.5** when the agent moves closer to the goal (measured by Manhattan distance) and a penalty of **-0.3** when revisiting previously explored positions. This reward shaping creates a smoother gradient toward the goal, providing immediate feedback for progress while discouraging inefficient cycles. This enhanced reward structure guides exploration more effectively, enabling the agent to learn efficient paths with significantly fewer episodes by turning the sparse reward problem into a more informative learning landscape that rewards incremental progress toward the objective.

MCTS Concepts Implemented:

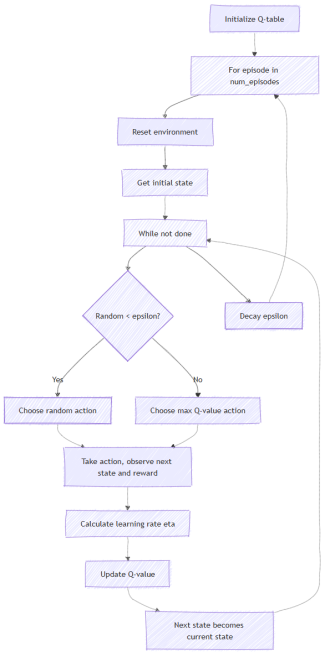
- Our Castle Escape Q-learning implementation embodies the four fundamental MCTS components while using different algorithmic machinery.
- The advanced implementation's smart exploration strategy with health-based and goal-directed action weighting represents the **Selection** phase, intelligently choosing promising paths through the state space.
- Our valid action filtering and position tracking serve as **Expansion**, determining which states to explore next.
- Though Q-learning doesn't use explicit playouts, our exploitation phase, when epsilon is low, acts as an implicit **Simulation**, estimating long-term value from current knowledge.
- Finally, our reward shaping and Q-value updates mirror **Backpropagation**, propagating reward signals through the state space to improve future decisions—creating a learning system that, like MCTS, effectively balances exploration with exploitation to navigate complex environments.

Reinforcement Learning Implementations

Two distinct Q-learning approaches were implemented to train agents for the Castle Escape environment:

Standard Q-Learning (`Q_learning.py`)

The standard implementation uses classic tabular Q-learning with:



This implementation features:

- Epsilon-greedy exploration strategy with decaying exploration rate
- Dynamic learning rate that decreases with the number of state-action visits
- A discount factor (gamma) of 0.9 to balance immediate and future rewards
- A hashing function to convert multi-dimensional state representations into table indices

The Q-learning update rule is a fundamental concept in reinforcement learning. Q-learning trains an agent to make decisions by learning the optimal action-value function Q(s, a), which estimates the expected cumulative reward for taking action a in state s and following the optimal policy thereafter. The formula updates the Q-value for a state-action pair as follows:

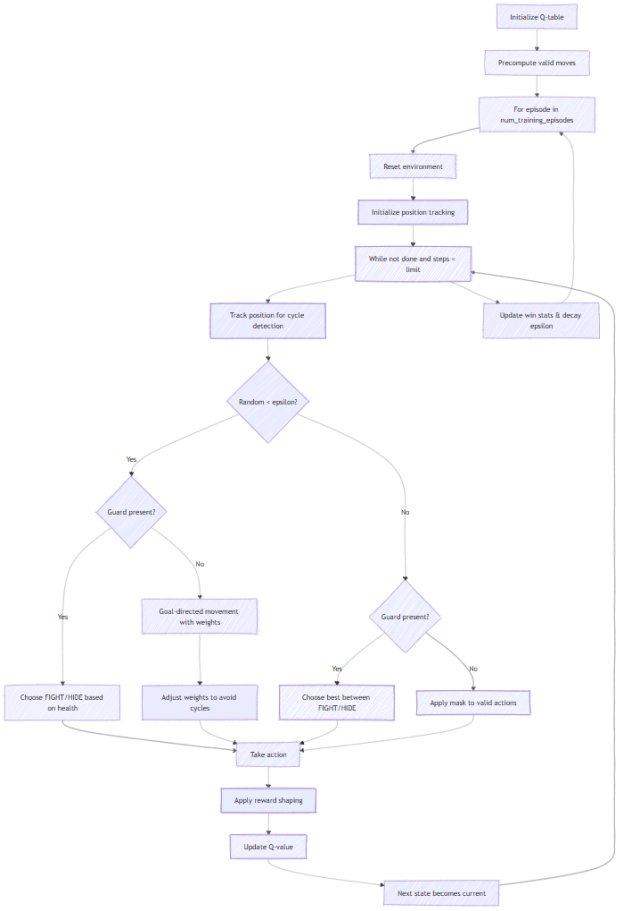
$$Q^{new}(S_t, A_t) \leftarrow (1 - \underbrace{\alpha}_{\text{learning rate}}) \cdot \underbrace{Q(S_t, A_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \underbrace{\left(\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(S_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}_{\text{new value (temporal difference target)}}$$

- α: Learning rate (controls how much new information overrides old knowledge).
- Rt+1: The immediate reward received after taking action
- γ: Discount factor (determines the importance of future rewards).
- maxaQ(St+1, a): The max estimated future reward from the next state St+1.

This update rule combines the current Q-value with a "temporal difference target," which includes both the immediate reward and the discounted estimate of future rewards. Over time, this iterative process enables the agent to learn an optimal policy through trial and error.

Advanced Q-Learning (`Advanced_Q_learning.py`)

The advanced implementation builds upon the standard approach with several enhancements:



Key enhancements include:

- Smart exploration biases that favor movements toward the goal
- Guard-specific decision making that adjusts strategy based on health state
- Cycle detection to prevent the agent from getting stuck in loops
- Position-based filtering of valid actions to avoid wasted exploration
- Reward shaping to provide intermediate feedback on progress toward the goal

Results and Analysis

Both Q-learning implementations successfully learned to navigate the Castle Escape environment, but with notable differences in performance and behavior:

Standard Q-Learning Performance

The standard implementation required a large number of training episodes (100,000+) to achieve consistent success. The agent gradually learned to:

- Avoid guards when possible
- Choose appropriate actions (FIGHT vs. HIDE) based on the current state
- Find paths to the exit that minimize the risk of health loss

However, the standard agent occasionally:

- Got stuck in cyclical patterns
- Made suboptimal decisions when encountering guards
- Required extensive exploration to discover effective strategies

Advanced Q-Learning Performance

The advanced implementation learned more efficiently, requiring fewer episodes (2,000+) to achieve similar or better performance. This agent demonstrated:

- More direct navigation toward the goal
- Better decision-making in guard encounters based on health state
- Reduced cyclical movement patterns
- More consistent success across multiple test runs

The enhancements in the advanced implementation (particularly the smart exploration biases and cycle detection) significantly improved learning efficiency and the quality of the learned policy.

In conclusion, Castle Escape demonstrates how even simple Q-learning algorithms can tackle complex decision problems when enhanced with domain knowledge and careful environment design. Future work could explore DQNs, procedurally generated layouts, or multi-agent scenarios.