­­­**Introduction**

In our project we are solving the problem of developing an AI agent that performs well in the game of Jenga. For those not familiar with the rules, the game starts with 54 wooden rectangular blocks (although there are variations of the game with less blocks) stacked in a tower. At each level of the tower there are three blocks placed side by side along their longer side, and the levels alternate direction relative to each other. For example, if in the downmost level the blocks are oriented horizontally by their longer sides, in the next level the blocks should be oriented vertically. The players make moves one after the other, so that in each move a player removes one block from any layer, except the top three, and places it on top of the tower. The player who causes any of the blocks to fall loses the game.

The more blocks are removed from the tower, especially from the lower levels and from the sides, the more unstable the tower becomes and the more likely it is to fall. To succeed in the game, a player needs to make a move that is cautious enough for the tower not to fall in their move, but also risky enough for the tower to become significantly more unstable in the adversary’s move. Due to this tradeoff, there is no trivial strategy the player can adhere to in order to win, and this makes the problem interesting.

As can be inferred already from the description of the game, Jenga is a physical game. In the real-world Jenga game, noise is introduced to the players’ strategies due to imperfect tower stacking, manufacturing tolerances, and friction between blocks. That is, no matter how good their strategies are, the tower can fall if they accidentally touch other blocks when pulling out the block they want to. Because of this, normally players can “probe” the blocks to find the ones that are loose enough to be safely pulled out. Therefore, to create a solver for the real-world Jenga, we would need to use robotics. Since this is not a project in robotics, we decided to simulate the game in its simplified form.

For simulation of the Jenga tower, we turned to Unity and its physics engine. It allowed us to control the parameters of static and dynamic friction, as well as gravity. It also allowed us to know the positions of all the blocks at any given time. First and foremost, it allowed us to isolate the strategic component of the Jenga game by eliminating the noise when pulling out blocks. That is, we were able to implement the process of pulling out of the blocks as if they are pulled without any friction with the neighboring blocks. Furthermore, it allowed our agents to get understanding of the current stability of the tower based on the blocks’ angles and, for example, make more cautious moves when the tower is less stable.

**Previous Work**

In our search for previous work that tried to make an automated solver for Jenga, we managed to find only one paper on the matter, which is a project by a team of researchers from MIT (N. Fazeli, M. Oller, J. Wu, B. Tenenbaum, and A. Rodriguez) published in the [Science Robotics](https://www.science.org/doi/10.1126/scirobotics.aav3123?utm_source=the+new+stack&utm_medium=referral&utm_content=inline-mention&utm_campaign=tns+platform) journal, with a short video available on [Youtube](https://www.youtube.com/watch?v=o1j_amoldMs) about it. Their project aimed to create a solver for Jenga in the real-world setting and described creating a robot that is capable of seeing and touching. One of the downsides of this approach is that physical learning is very costly. Our project, on the contrary, only aims to solve Jenga in a simulation, although we believe that with parameters tweaking in Unity we can achieve high degree of realism.

Similarly to the “probing” technique used by the human players, the robot uses tactile feedback to locate loose blocks that are safe to remove. It selects a block at random, tries to push it for some time, and if the push doesn’t make the block stick out, it retracts and goes to the next random block. As well as us, the researchers assumed that the blocks are rigid and don’t change form when force is applied, are absolutely equal, and there are no external forces like the wind and vibration.

During a short training period that the researchers call exploration phase, the robot learns the underlying physics model of the game: it categorizes the blocks into groups and adjusts the force and its direction according to the current block’s group. After the exploration phase the robot’s performance is measured. Essentially, the robot learns to play Jenga without the opponent, and so its main goal is to prolong the game by making the safest move. On the other hand, the solver in our project has an opponent, and so its objective is to try to make the riskiest move that doesn’t fall the tower in order to win.

The researchers used Markov Chain Monte Carlo (MCMC) Sampling with Hamiltonian dynamics to approximate the distribution over states and abstractions after receiving noisy observations. While the regular Monte Carlo draws independent samples, MCMC draws samples so that every sample depends on the previous one. Hamiltonian dynamics means that the sampling is done according to some clues. The researchers also used Deep Residual Learning, which learns the difference between the desired output and the input. In our project we either tried or actually used the simpler counterparts of these algorithms, Monte Carlo Sampling and Deep Learning.

The robot’s performance was evaluated by how many moves it could make in a randomly generated tower until the game ended. In our project we made a similar metric of the average game duration against the adversary. As baselines, the researchers used a Feed-Forward NN model and PPO implementation from the OpenAI Gym. In our project we also used a Feed-Forward NN model that was trained with fewer samples, as well as an adversary with random strategy.

**Methodology**

The codebase of the project is divided into two parts: the Unity implementation, and the Python code for all other aspects of the project and uses the Unity build created from the first part. The first part can be found [here](https://github.com/yonatanvolog/jenga-unity-build), and the second part with the instructions on how to run every part of the project can be found [here](https://github.com/yonatanvolog/jenga-game). All the external packages used in the project are listed in the requirements file.

When running the project, Python code and Unity build run together at all times and communicate over TCP. Unity physical engine is responsible for handling block collision simulations, as well as detecting whether the game ended using colliders around the tower. Unity is capable of resetting the tower, removing a block, detecting any of the blocks falling, and calculating the blocks angles. The agents in Python send commands to Unity to perform the removal of the block they need and get feedback on the state of the tower after the action was performed.

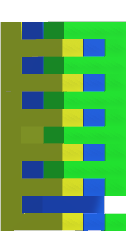
A diagram of steps and steps

Description automatically generated with medium confidence

*Figure 1: High-level communication process*

We assume in the project that the blocks are rigid and don’t change form when force is applied, are absolutely equal, and there are no external forces like the wind and vibration. We also assume that Unity models everything correctly according to the environmental parameters passed, and that these parameters don’t give advantage to any specific strategy of the player. Other assumptions are mentioned later in this section.

Since blocks can tilt and shift, representing a game state using a binary string that has zeros and ones depending on whether blocks are present or not wouldn’t be enough. And since blocks that are arranged parallelly hide those behind them, observing only one side of the tower wouldn’t suffice either. Therefore, we use a “spread” of the tower – combined screenshots from two adjacent sides, – which provides enough spatial context.



*Figure 2: Example of a spread of the tower*

The spread is later converted to greyscale and normalized, scaling pixel values to the range from -1 to 1. So, the state space of the problem is as follows:

Using image files while communicating between Unity and Python required us to introduce delays as sometimes the images were not registered by the operation system yet upon request from the agents.

To manage the large branching factor in Jenga, especially at the start of the game, and to make the learning process more feasible for the agents while keeping the game challenging, we reduced the tower height from 18 to 12 levels. We then numbered each level starting from zero, where zero is the top-most level. Also, for discerning between blocks at the same level from different points of view, we colored blocks at each level in Unity in three colors and assigned them numbers: yellow (0), blue (1) and green (2). Each action is represented by a tuple of the number of the level and the color of the block to remove. Therefore, the action space is as follows:

The reward function was defined the same way for all the agents. We assume that the lower level a block is taken from, the more unstable this makes the tower. This is because in the simulation, on the contrary to the real world, all blocks lie firmly on each other, since there aren’t any manufacturing errors or inaccuracy in assembling the tower. Because we want the agent to make the tower more unstable for the adversary, we give a bonus for the number of the level. However, we want the agent to not make the tower too unstable for it to fall, and so we give a bonus for a small tilt, and a penalty for a big tilt after the move. The information on the tilt is calculated by the Unity build after block removal as the maximum average tilt of all the blocks, where the tilt is the maximum between the roll and the pitch for every block.

Initially we thought that Monte Carlo Tree Search (MCTS) would be a good fit for the project. However, with the branching factor still being big after reducing the tower to 12 levels, and with games ending relatively quickly, expanding beyond a single level in the search tree would be not only computationally costly, but also unnecessary. With only one level being expanded, it was no longer MCTS but rather a greedy algorithm which we called Greedy Simulation-Based Action Search (GSBAS).

Every time before choosing a move, the GSBAS agent randomly chooses ten actions out of possible actions and simulates them. We chose to simulate only ten because simulations are slow, and we can count that even within these ten there will be good actions. GSBAS then takes the action that achieved the highest reward in the simulation. The problem in this approach is that the reward function is significantly smaller if the tower fell. So, the agent knows after which actions the tower falls in the current game, and this gives the agent unfair advantage over its opponent. Of course, there is still chance that the chosen action will go differently than in the simulation and will make the tower fall, but this chance is small. Additionally, this algorithm has other disadvantages: it is slow due to the simulations, and it is overly cautions due to maximizing the reward for the particular game rather than taking an action that performed good overall but not necessarily is best for this particular game.

The other two agents we implemented are based on reinforcement learning and use the Deep Q-Network (DQN) approach to estimate Q-values of actions. They both are called hierarchical agents because they use two separate Neural Networks for the estimation: one for selecting the level in which a block is removed, and the other to select the color of the block. Here we are assuming that blocks in the same level and blocks of the same color can be grouped together. The only difference between the agents is in their learning process: while the regular Hierarchical DQN agent assumes the environment is deterministic and learns from the maximum estimated future reward (it updates the Q-values based on the best action, no matter if this action was actually taken, which is called off-policy RL), SARSA assumes the environment is partially stochastic and learns from the actions taken (it updates the Q-values based on the action it takes next, which is called on-policy RL).

Since the next action the SARSA agent chooses can be an exploration action, this leads to more conservating learning and less overestimations.

**Results**

We created several baseline strategies for adversaries to evaluate the performance of our models against:

* Random Strategy – the adversary chooses a random block in their move.
* Optimistic Strategy – the adversary thinks the tower is more stable than it actually is and chooses a random block from the same level or lower than the agent did in its last move.
* Pessimistic Strategy – the adversary thinks the tower is less stable than it actually is and chooses a random block from the same level or higher than the agent did.