Kage
The Abstract Strategic Game

AI Project

Group 4: **Ethan Page, Saviz Saei, Matthew Rester, Jonathan Butterfield, Caleb Byers**

Spring 2023

Kage Game Overview

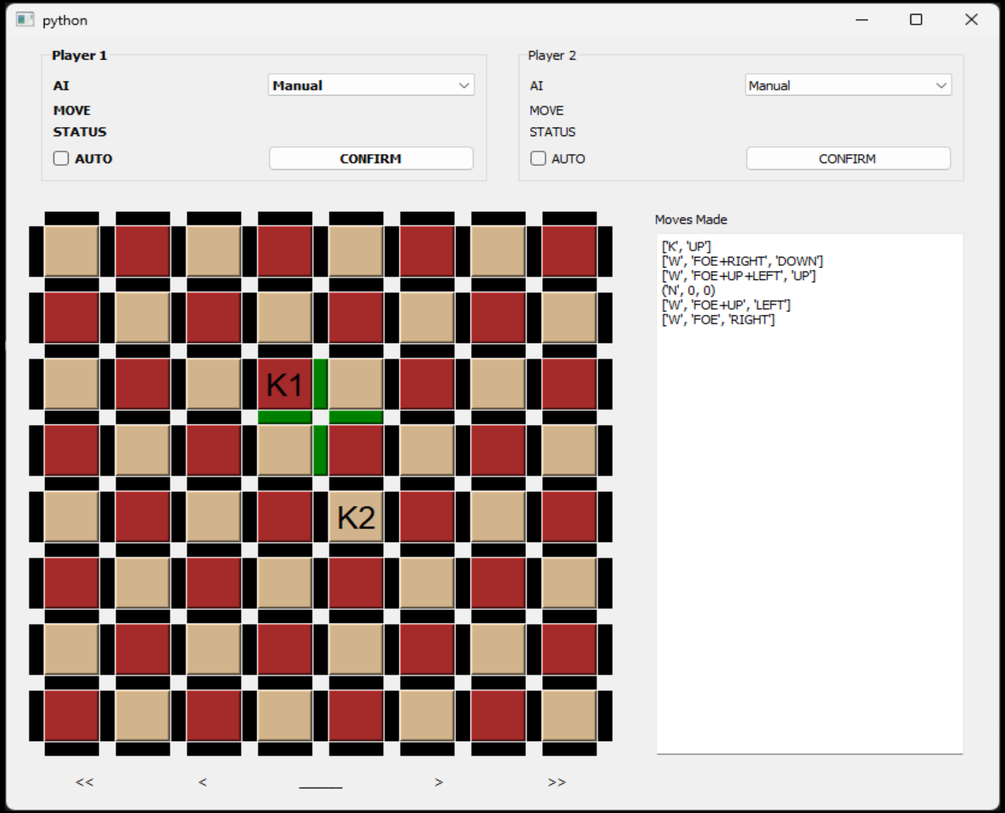
- Game Components: 8x8 chess board, 2 Kings
- Starting Position: Kings located on a diagonal in the center of the board
- Turn-Based Gameplay: Players can either move their King or place one wall on each turn
- King Movement: Move up, right, left, or down one square; cannot move diagonally, move into the space of another King, move off the board, or move through a wall
- Win/Lose Condition: Form a Kage (walls) around the opponent's King to win; form a Kage around your own King without the opponent's King included to lose

Kage: The Abstract Strategic Game

A similar game you might know "Kono" from Korea

- Two players attempt to capture each other's pieces by jumping over their own pieces and landing on the other player's pieces
- Objective: Capture the opponent's pieces by jumping over your own pieces and landing on the opponent's pieces
- Unique Features: Small but interesting strategic game with simple rules but complex gameplay
- Skills Developed: Enhances math, problem-solving, abstract thinking, and spatial perception skills

Visual Interface



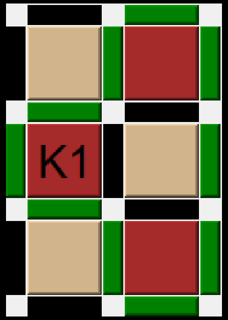
Moves Made

```
['K', 'UP']
['W', 'FOE+RIGHT', 'DOWN']
['W', 'FOE+UP+LEFT', 'UP']
('N', 0, 0)
['W', 'FOE+UP', 'LEFT']
['W', 'FOE', 'RIGHT']
```

K1: King 1
K2: King 2 as Opponent

Walls (Green lines): Can be placed by either player to block movement: 🟩

Condition to Win: K2 wins if they form a wall around K1



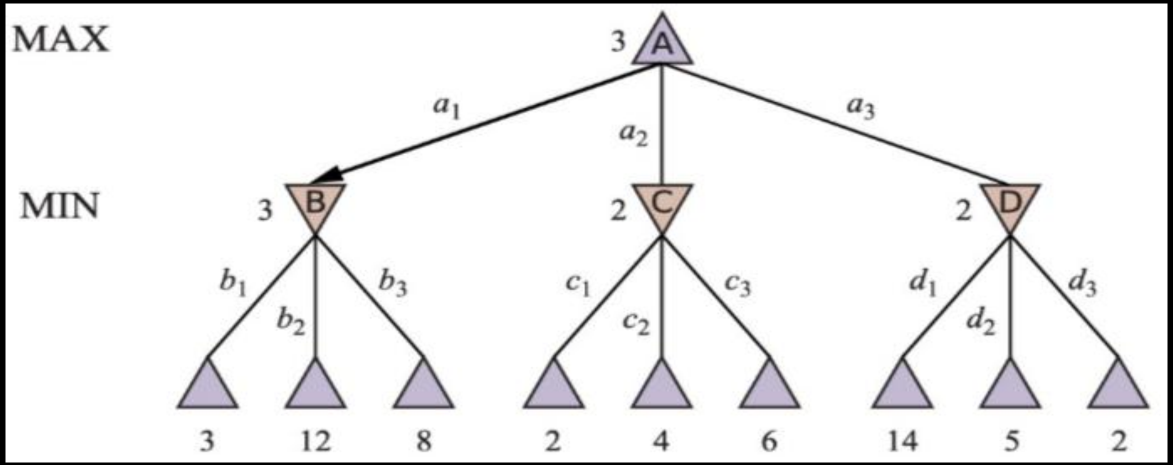
Demo
(minimax wins)

Methods and Algorithms

- Expert Systems: Incorporates knowledge and rules from domain experts to make decisions and suggest actions
- Alpha-Beta Pruning in Minimax Algorithm: Determines the best move to make by minimizing the maximum possible loss
- Genetic Algorithms: Uses evolutionary techniques to optimize game strategies and decision-making
- Deep Q-Network Reinforcement Learning: Learns through trial and error by receiving feedback or rewards based on actions taken

Minimax Algorithm

- The minimax algorithm is used to find the optimal move for the player, assuming that the other player is also playing optimally.
- The algorithm constructs a search tree that represents all possible moves in a game.
- The tree is explored recursively, with each level representing a player's turn, and each node representing a possible move.
- At each level, the algorithm alternates between maximizing and minimizing the potential outcome of the game.



Alpha-Beta Pruning in Minimax Algorithm

- Alpha-beta pruning is a technique used to optimize the minimax algorithm by reducing the number of nodes that need to be evaluated.
- It works by cutting off the search tree at nodes where it can be determined that a certain move will not be chosen.
- If the score of a node is worse than alpha (for the maximizing player) or better than beta (for the minimizing player), then the search can be stopped at that node, as it is already known that this node will not be chosen.
- This greatly reduces the number of nodes that need to be evaluated and can speed up the algorithm significantly.

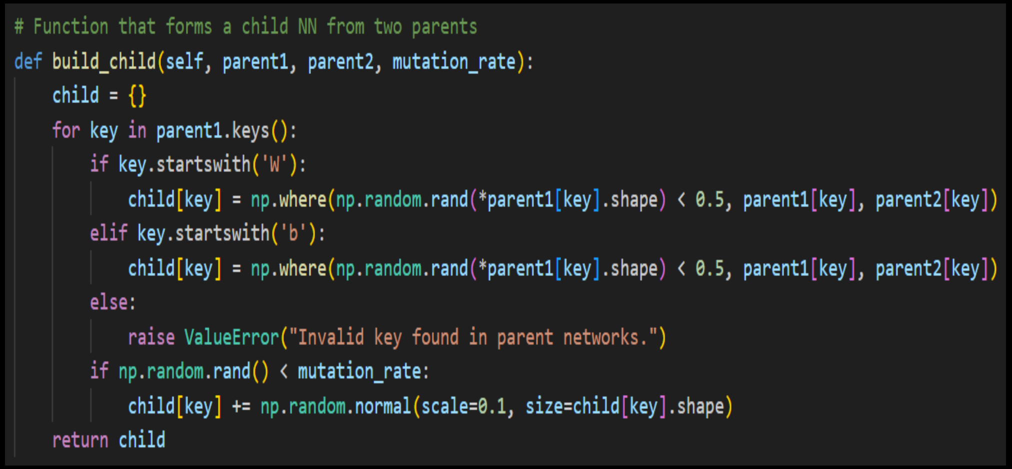
Genetic Algorithm

- The genetic algorithm here uses a neural network to make decisions. However, in this case, there is no backpropagation, or learning within the networks. Instead, a population of agents, where each agent is a neural network, is created with random weights and biases. They then compete against each other, and the best agents are used to populate the next generation. Over time, an optimal agent is approached.

Genetic Pseudocode

1. Generate a population of Neural Networks(NNs) with random weights and biases.
2. Let them compete against one another round-robin style, utilizing only feedforward, and tally up # of wins.
3. Select the best performers to be the parents of the next generation.
4. Repopulate using crossover and mutation.
    1. Crossover means that aspects of the parental NNs are chosen at random. For example, every NN here has 4 weight matrices. So, when forming the child’s W1, set it equal to either parent1’s W1 or to parent2’s W1.
    2. Mutation is the random introduction of new futures so that the population can change paths with time.
5. Repeat steps 2-4 for some # of generations.
6. Save the best agent after training to be a reusable competitor.

Example code: Crossover and Mutation



```python
# Function that forms a child NN from two parents
def build_child(self, parent1, parent2, mutation_rate):
    child = {}
    for key in parent1.keys():
        if key.startswith('W'):
            child[key] = np.where(np.random.rand(*parent1[key].shape) < 0.5, parent1[key], parent2[key])
        elif key.startswith('b'):
            child[key] = np.where(np.random.rand(*parent1[key].shape) < 0.5, parent1[key], parent2[key])
        else:
            raise ValueError("Invalid key found in parent networks.")
        if np.random.rand() < mutation_rate:
            child[key] += np.random.normal(scale=0.1, size=child[key].shape)
    return child
```

Genetic Challenges Faced

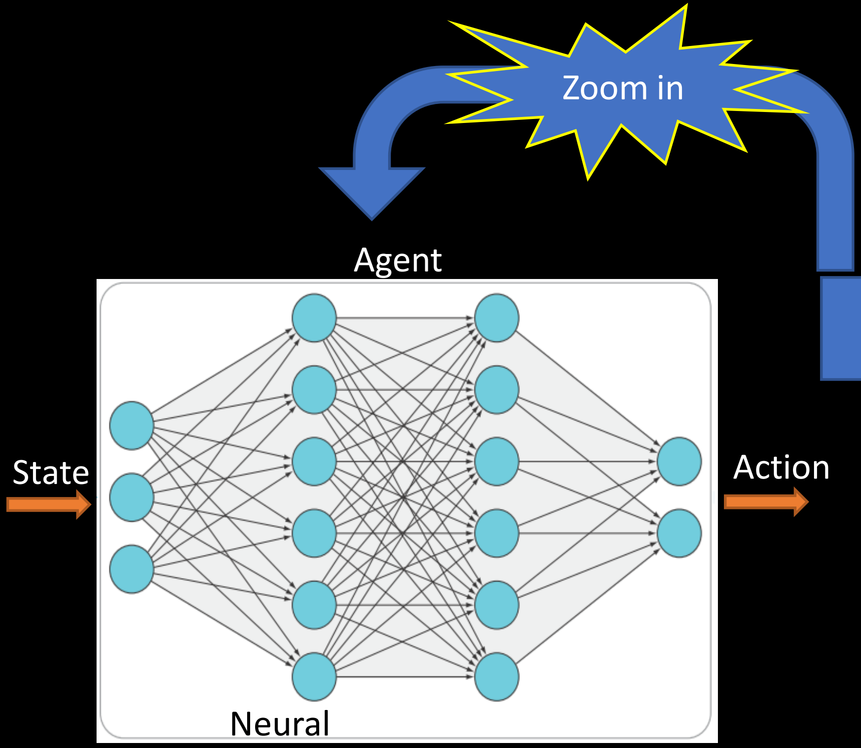
- Genetic algorithm was difficult to implement here for a number of reasons:
    1. This form of genetic algorithm is computationally expensive, so training takes a long time.
        - That being said, the process of feeding forward through the NNs was not time-consuming in comparison to other aspects of the project. Great changes in the size of the NNs had little effect on the time taken.
    2. There are many variables and parameters that can be changed (population size, number of generations, mutation rate, size, and number of the hidden layers of the NNs). There is much trial and error involved in tweaking these values.
    3. Overall, this is not the best AI model to fit this board game, but with some fine-tuning and extended training, it is capable of producing a good player.

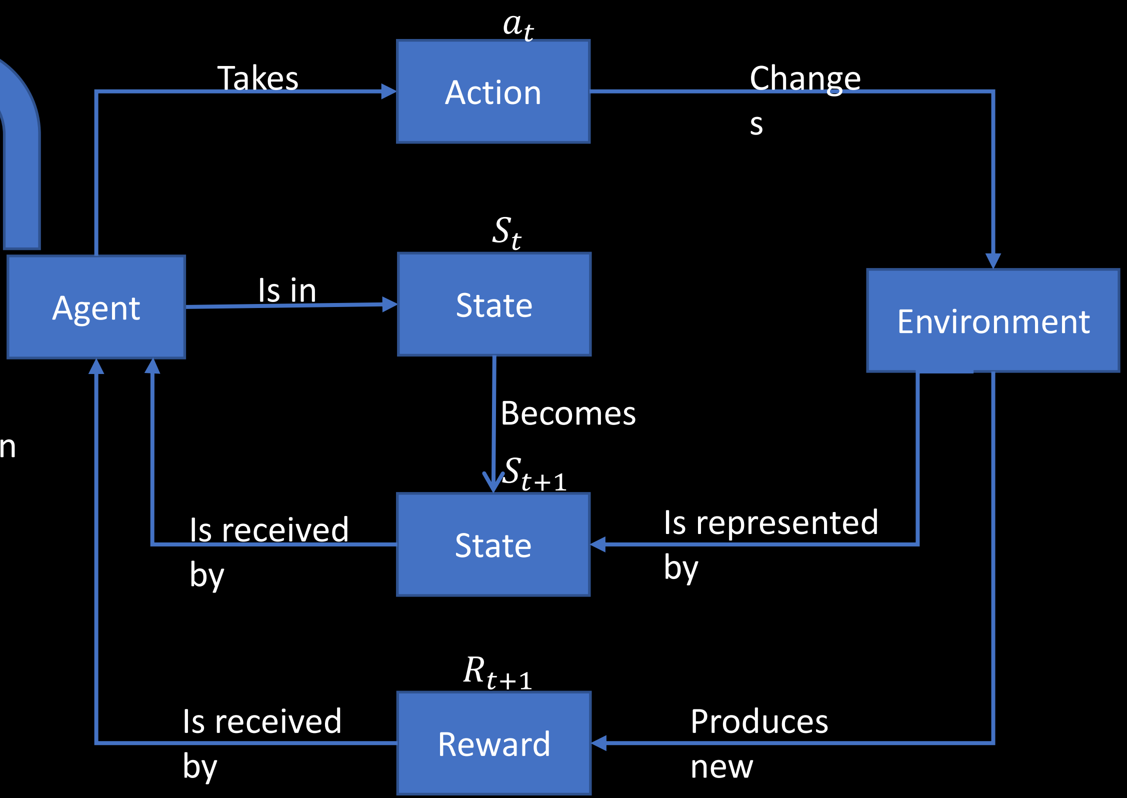
Deep Q-Network Reinforcement Learning (DQN-RL)

- Deep Q-Network reinforcement learning algorithm uses a neural network to approximate the Q-function, enabling an agent to learn a policy to maximize its long-term reward by iteratively adjusting its action-selection strategy while leveraging experience replay and target networks to stabilize the learning process.

$$\underbrace{Q(S_t, A_t)}_{\text{Updated Q value}} = \overbrace{Q(S_t, A_t)}^{\text{Current Q value}} + \underbrace{\alpha}_{\text{Learning rate/ Step Size}}[R_{t+1} + \overbrace{\gamma}^{\text{Discount factor}} \underbrace{\max Q(S_{t+1}, a)}_{\text{Max Q value for all actions}} - Q(S_t, A_t)]$$

The standard framework for RL algorithms

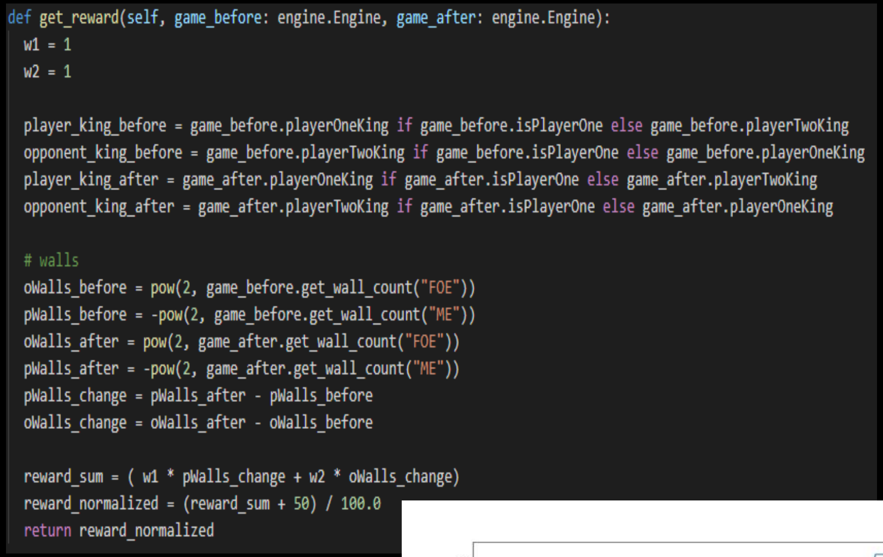




Deep Q-Network Reinforcement Learning (DQN-RL)

- The architecture of the DQN model is defined with a 5-layer neural network with dropout layers
- An Adam optimizer with L2 regularization is used, and the learning rate is scheduled to decay at specific intervals
- Experience replay is implemented with a deque to store experiences, and random mini-batches are sampled to update the DQN
- The reward function measures the effectiveness of the agent's actions by Calculating the number of walls for both the player and the opponent
- Assigning weights w1 and w2 in the reward function can help to adjust the importance of the number of walls and normalize the reward to be ensured that the reward values are compatible with the neural network's output range
- The agent plays the game and learns from its experiences through the 500 epochs

Example code: reward function

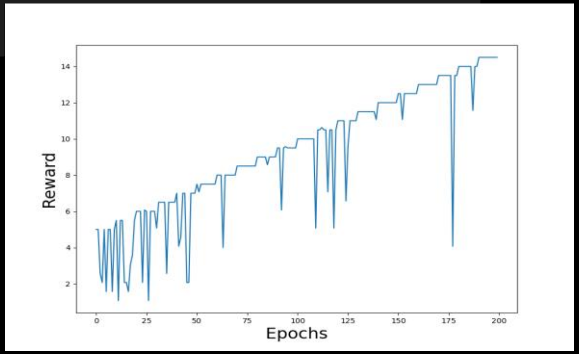


```python
def get_reward(self, game_before: engine.Engine, game_after: engine.Engine):
  w1 = 1
  w2 = 1

  player_king_before = game_before.playerOneKing if game_before.isPlayerOne else game_before.playerTwoKing
  opponent_king_before = game_before.playerTwoKing if game_before.isPlayerOne else game_before.playerOneKing
  player_king_after = game_after.playerOneKing if game_after.isPlayerOne else game_after.playerTwoKing
  opponent_king_after = game_after.playerTwoKing if game_after.isPlayerOne else game_after.playerOneKing

  # walls
  oWalls_before = pow(2, game_before.get_wall_count("FOE"))
  pWalls_before = -pow(2, game_before.get_wall_count("ME"))
  oWalls_after = pow(2, game_after.get_wall_count("FOE"))
  pWalls_after = -pow(2, game_after.get_wall_count("ME"))
  pWalls_change = pWalls_after - pWalls_before
  oWalls_change = oWalls_after - oWalls_before

  reward_sum = ( w1 * pWalls_change + w2 * oWalls_change)
  reward_normalized = (reward_sum + 50) / 100.0
  return reward_normalized
```



Challenges Faced

- Computational resources: Training a deep learning model, especially for a large number of epochs, can be computationally expensive.
- Learning rate and optimization: We are using the Adam optimizer with a learning rate scheduler to adjust the learning rate. However, the initial learning rate and the decay rate is challenging to have a trade-off between exploration vs. exploitation
- Network architecture: we need to experiment with different architectures, such as deeper or wider networks
- Reward function design
- Convergence and training stability

Results and Future Work

- Among all of the methods which are used, Alpha-Beta Pruning in Minimax Algorithm performs really well which takes only 2.06 seconds at depth=2 without pruning, and 0.252 with pruning
- Genetic Algorithm with some fine-tuning and extended training might be capable of producing a good player
- Deep Q-Network Reinforcement Learning (DQN-RL) with Monte Carlo Tree Search (MCTS) might be capable of selecting actions based on a combination of upper confidence bounds and estimated Q-values

Questions?