Lucas Summers (lsumme01@calpoly.edu)
Braeden Alonge (balonge@calpoly.edu)
Nathan Lim (nlim10@calpoly.edu)
Megan Fung (mfung06@calpoly.edu)

CSC 480 Team Project Proposal (Revised)

Project Idea (Problem Statement/ Question)

- What is the best design of an intelligent 2048 agent? Specifically, which AI methods amongst search-based, heuristic, Monte-Carlo, and reinforcement-learning techniques tend to produce the highest average scores in ?

Abstract (Summary of Project Purpose + Approach)

- This project aims to develop an AI agent capable of achieving the highest possible score in the single player puzzle game 2048. We will experiment with a variety of AI techniques, including Expectimax search, heuristic evaluation functions, and reinforcement learning methods. Our bot will dynamically decide optimal moves at each game state by predicting future outcomes and maximizing reward. We aim to develop a model that is able to achieve an average game score above the 2048 tile in a 4x4 board and achieve a best case high score as close to tile 131,072 as possible—the theoretical limit of a 4x4 board. Through this project, we hope to explore how AI planning and decision-making algorithms perform under uncertain, partially observable environments.

Motivation (Why)

- Games like 2048 offer an ideal environment for experimenting with AI techniques, balancing strategy, planning, and randomness. Building an AI that can consistently achieve high scores in 2048 is a challenging way to apply search algorithms, heuristic designs, and reinforcement learning, all while revealing how each different AI algorithm compares with one another. Success in this project would demonstrate practical mastery of game playing AI techniques, and deepen our understanding of how artificial agents can make decisions with incomplete information.

Background (Related Works + Existing Solutions)

- Previous attempts to build 2048 AI bots have primarily used the following approaches, which can be seen in our bibliography of existing work we examined.
    - Reinforcement Learning: some models apply deep reinforcement learning, allowing agents to learn policies over millions of games.
        - OpenAI: uses reinforcement learning agents in simple games to learn complex strategies autonomously

    - Expectimax Search: unlike minimax, Expectimax accounts for probabilistic outcomes (random tile spawns in 2048).
        - Matt Overlan's AI for 2048: shows that using Expectimax with a strong evaluation function can consistently reach 4096+ tiles.
    - MCTS: one of the most prevalent methods to solving 2048 because of its relative simplicity and efficiency.
- Hybrid approaches that combine approaches using reinforcement learning and search based algorithms like MCTS/Expectimax have also become more prevalent to create agents with the strengths of multiple approaches.
- Despite these successes, challenges remain: computational cost of deep searches, crafting an effective evaluation function, and balancing exploration vs exploitation in learned policies.

Relevance (Related Class Topics)

- This project directly ties into several AI topics we've covered in class:
    - Search algorithms– expectimax, heuristic based searches
    - Adversarial and Stochastic Planning– handling randomness in environments (non-determinism)
    - Reinforcement Learning– agent training through rewards over time

---

Bibliography

1. **Matt Overlan, "Building a 2048 AI with Expectimax,"** GitHub, 2014. https://github.com/nneonneo/2048-ai
   – Discusses applying Expectimax search to 2048 with heuristic evaluation functions.

2. **Guei, Hung. On Reinforcement Learning for the Game of 2048. Diss. National Yang Ming Chiao Tung University, 2023.** https://arxiv.org/abs/2212.11087
   – Explores the use of Reinforcement Learning as an approach to creating 2048 solving agents.

3. **P. Rodgers and J. Levine, "An investigation into 2048 AI strategies," 2014 IEEE Conference on Computational Intelligence and Games, Dortmund, Germany, 2014, pp. 1-2, doi: 10.1109/CIG.2014.6932920.** https://ieeexplore.ieee.org/document/6932920
   – Goes over various approaches to 2048 solving agents, including MCTS and ADLS (Averaged Depth Limited Search).

4. **Watanabe, Shota and Kiminori Matsuzaki. "Enhancement of CNN-based 2048 Player with Monte-Carlo Tree Search." 2022 International Conference on Technologies and Applications of Artificial Intelligence (TAAI) (2022): 48-53.** https://www.semanticscholar.org/paper/Enhancement-of-CNN-based-2048-Player-wit

h-Tree-Watanabe-Matsuzaki/7ea90a71c41240386da55bed2a1530e13b330de4
– Use of a deep learning CNN-based algorithm to solve 2048 while combining MCTS.

5. **Zhou, Yulin. "From AlphaGo Zero to 2048." (2019).**
https://www.semanticscholar.org/paper/From-AlphaGo-Zero-to-2048-Zhou/b9cc9a861531ede494a365e42b7ba788d638eaec
– Exploration of how the algorithm used in the successful AlphaGo Zero can be applied to solve 2048.

Timeline

Week 1:

- Set up Git repository
- Define a tech stack (likely Python, NumPy, PyTorch) and workspace to begin the coding portion of the project for all members
- Begin development of a custom 2048 grid to develop on
- Decide on size of grid (ie. 4x4, 5x5)

Week 2:

- Begin/continue research on how to implement Expectimax, and/or examine other possible algorithms to implement
- Finish development of custom 2048 grid
- Start final paper; begin with outline and move onto drafting afterwards
- Start code of the AI model using the finalized custom 2048 grid
- Revise proposal

Week 3:

- Continue development of code, experimenting with different algorithms and coming to conclusions on what the final algorithm used should be
- Continue adding to the final paper, completing at least half of it by the end of the week
- Attain a final paper rough draft

Week 4:

- Finish final touch ups of the “best” agent and run large-scale evaluation using at least 1000 games to get final stats of model
- Finish final paper
- Create final presentation slides and prepare to present
- Make collaboration documents

Methods

1. **Monte Carlo Tree Search (MCTS)**
   Implementation
   - Use the four basic phases: Selection, Expansion, Simulation, Backpropagation
   - Use UCB1 to balance exploitation and exploration ($C = \sqrt{2}$)
   - Simulation handled through random rollouts
   - Time limit for limiting simulation count based on the user requested speed

   Justification
   - Effectively handles the high branching factor (up to 4 moves × many possible tile placements)
   - Balances exploration of promising strategies with exploitation of known good moves
   - Avoids exhaustive search by focusing on statistically promising paths
   - Adapts search effort based on position complexity

2. **Expectimax**
   Implementation
   - Strategic pruning of for reducing complexity
     - Partial alpha-beta pruning at MAX nodes
     - Sparse sampling at chance nodes when many empty cells exist
     - Probability cutoffs for extremely unlikely outcomes
   - Progressive deepening for time-bounded decision-making
     - Max search depth probably going to be limited to 3-5 moves due to exponential branching
   - Common evaluation function for non-terminal states implemented as a weighted combination of several simple board heuristics:
     - Empty cell count (mobility)
     - Monotonicity (tiles increasing/decreasing in order)
     - Smoothness (difference between adjacent tiles)
     - Maximum tile placement (corners preferred)
     - Merge potential (adjacent tile similarity)

   Justification
   - It directly models the probabilistic nature of new tile spawns (90% chance of 2-tile, 10% chance of 4-tile)

- It can determine the expected outcome of all possible move sequences
- It handles alternating player moves and random environment actions
- It addresses the key uncertainty element that makes 2048 challenging for traditional search

3. **Deep Reinforcement Learning Approach**
   Implementation
   - State encoded as a 4x4x16 one-hot encoded grid (powers of 2 to represent tiles)
   - Use of a CNN combined with a 4-unit policy head to represent the probabilities of each of the four moves being the "optimal" move
   - We can train with different numbers of convolutional layers, kernels, and filters to see what works best
   - Training involves using a *very* large amount of self-play steps and the reward can be set to the delta in score

   Justification
   - Deep reinforcement learning removes dependence on handcrafted heuristics
   - Pure neural agent can provide a benchmark against comparatively search-heavy approaches
   - Can also provide the policy for the Hybrid agent, but the Hybrid approach is separated to isolate effects.

4. **Hybrid Approach**
   Implementation
   - Use MCTS for overall search strategy and move selection
     - Focuses computation on promising move sequences
     - Manages computational budget across the game tree
   - Two options for rollout phase:
     - Pure CNN policy/value networks trained via deep reinforcement learning
     - Limited-depth Expectimax for rollout phase (2-3 plies)
       - More accurately evaluate position potential than random play
       - Use the same evaluation function at their depth limit
       - Reinforcement learning for optimized Expectimax evaluation function
         - Uses Cross-Entropy Method (CEM) to iteratively update weights based on empirical performance
         - Applies learned weights across both MCTS and Expectimax components
   - **Interaction Flow**
     1. MCTS begins search from current game state
     2. UCB1 formula guides tree traversal to promising leaf nodes
     3. At leaf nodes, limited-depth Expectimax or a pure NN evaluates potential outcomes
     4. Expectimax uses the shared, RL-tuned evaluation function
     5. Results are backpropagated through the MCTS tree

6. Final move selection based on most visited or highest value child

Justification

- Combines MCTS's efficient exploration with accurate handling of simulations via either a neural network or Expectimax's accurate handling of randomness
- Replaces weak random rollouts with principled simulations
- Maintains consistent evaluation across all components
- Can re-use the policy made from the pure DRL approach in its rollout

---

Study Design

1. **Game Environment**
    a. Standard 4×4 grid implementation of 2048 with all standard game mechanics
    b. Customizable for experimentation (board size, winning condition)
    c. Visualization and performance logging capabilities for analysis and debugging
    d. Performance optimized for rapid simulation
2. **Experiment Design**
    a. **Agent Configurations**
        i. **Baseline Agents**
            1. Random
            2. Greedy score maximization
        ii. **Core Agents**
            1. Expectimax
            2. MCTS
            3. Deep Reinforcement Learning
            4. Hybrid: MCTS + DRL NN/Expectimax + RL weight tuning
        iii. **Variants**
            1. Hybrid with NN-based rollout vs. Expectimax w/ RL tuned weights or evenly distributed weights
            2. Expectimax with RL tuned weights vs. evenly distributed weights
    b. **Common Infrastructure**
        i. Modular design allowing agent interchangeability
        ii. Consistent evaluation framework
        iii. Performance profiling and logging
        iv. Game state serialization for analysis
    c. **Parameter Optimization**
        i. **For Expectimax**
            1. Max search depth (3-6 plies/based on time given)
            2. Evaluation function weights (use even weights when not using RL)
            3. Pruning thresholds (for CHANCE nodes)
        ii. **For MCTS**

1. Simulation count (based on amount of time given)
2. Exploration constant (0.5-2.0 but might just stick with standard $\sqrt{2}$)
3. Tree reuse policies (full? partial?)

iii. For Deep Reinforcement Learning

1. Tweaking the training learning rate, the convolutional layers, kernel size, activation functions, etc.
2. Tweaking learning rate

iv. For Hybrid

1. Division of computation time between MCTS exploration and NN based rollout/Expectimax depth
2. RL training episodes (500-5000)

d. Testing Protocol

i. Each agent configuration plays 1,000+ complete games
ii. Fixed random seed for fair comparison
iii. Fixed "thinking time" per move across all agents

e. Evaluation Metrics

i. Performance Analysis

1. Average score achieved
2. Maximum tile reached (2048, 4096, 8192, etc.)
3. Success rate (% of games reaching 2048 tile)

ii. Secondary Analysis

1. Average game length (moves)
2. Board utilization (average empty cells maintained)
3. Strategic patterns (corner usage, etc.)
4. Move distribution (left, right, down, up)

iii. Visualization of Results

1. Performance Comparison Charts
2. Game Progress Charts
3. Decision Analysis Charts
4. Computational Efficiency Charts
5. Learning Curve Charts (for RL component)

Preliminary Results

1. 2048 Game Environment

a. Implemented basic 2048 board logic and game mechanics
   i. Efficient NumPy board representation optimized for AI operations
   ii. Move generation and validation systems
   iii. Random tile generation (90% for 2-tiles, 10% for 4-tiles)
   iv. Scoring system consistent with original 2048 game
b. Web-based game interface using Flask

    i. Creates visual display with tile animations like the original 2048 game and connects to backend agents via API calls
    ii. Allows the user to switch between different AI agents and see them work in real-time with score and stats
    iii. User can also pause AI and play the game themselves using keyboard
2. **Agent and Heuristic Framework**
    a. Designed a modular agent interface, allowing different AI implementations to be easily created
    b. Designed an interface for creating different board heuristics and combining them together into a composite heuristic with weights
    c. Implemented a random move agent as initial baseline
3. **Basic Agent Evaluation Methods**
    a. Established an initial performance monitoring and logging system that runs game simulations on each agent
    b. In the process of creating a way to graph the logged results to compare agents