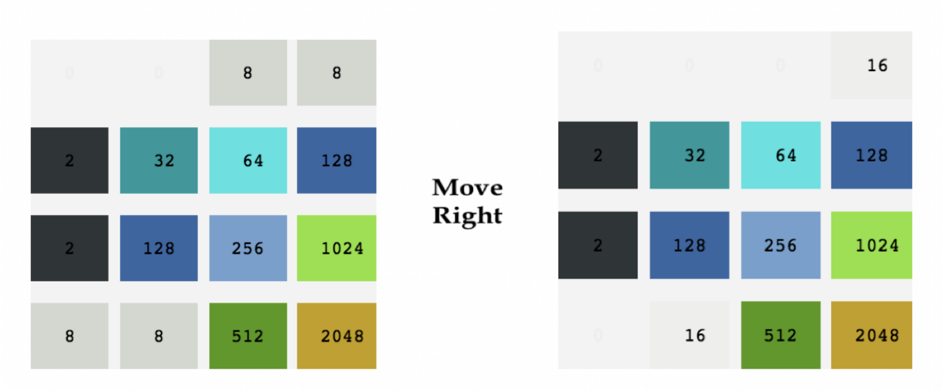
COMS 4701 Artificial Intelligence

Homework 4 Coding – Due date: March 27, 2021

In this assignment, you will create an adversarial search agent to play the 2048-puzzle game. A demo of the game is available here: gabrielecirulli.github.io/2048

I. 2048 As A Two-Player Game

2048 is played on a **4x4 grid** with numbered tiles which can slide up, down, left, or right. This game can be modeled as a two player game, in which the computer AI generates a 2- or 4-tile placed randomly on the board, and the player then selects a direction to move the tiles. Note that the tiles move until they either (1) collide with another tile, or (2) collide with the edge of the grid. If two tiles of the same number collide in a move, they merge into a single tile valued at the sum of the two originals. The resulting tile cannot merge with another tile again in the same move.

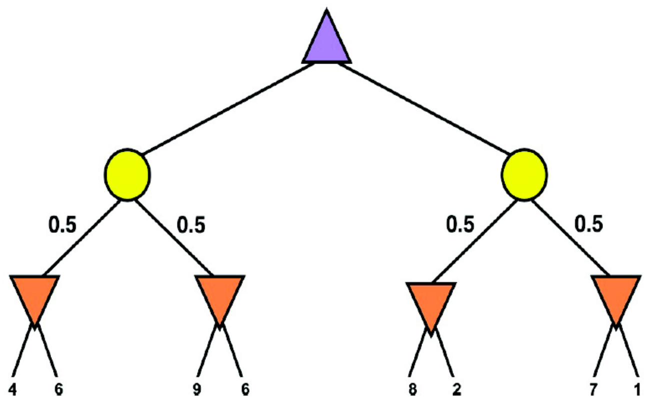


Usually, each role in a two-player games has a similar set of moves to choose from, and similar objectives (e.g. chess). In 2048 however, the player roles are inherently **asymmetric**, as the Computer AI places tiles and the Player moves them. Adversarial search can still be applied! Using your previous experience with objects, states, nodes, functions, and implicit or explicit search trees, along with our **skeleton code**, focus on optimizing your player algorithm to solve 2048 as efficiently and consistently as possible.

II. Choosing A Search Algorithm: Expectiminimax

Review the lecture on **adversarial search**. Is 2048 a zero-sum game? What are the minimax and expectiminimax principles? The tile-generating Computer AI of 2048 is not particularly adversarial as it spawns tiles irrespective of whether a spawn is the most adversarial to the user's progress, with a 90% probability of a 2 and 10% for a 4 (from GameManager.py). However, our Player AI will play **as if** the computer is adversarial since this proves more effective in beating the game. We will specifically use the **expectiminimax** algorithm.

With expectiminimax, your game playing **strategy** assumes the Computer AI chooses a tile to place in a way that minimizes the Player's outcome. Note whether or not the Computer AI is optimally adversarial is a question to consider. As a general principle, how far the opponent's behavior deviates from the player's assumption certainly affects how well the AI performs. However you will see that this strategy works well in this game.



Expectiminimax is a natural extension of the minimax algorithm, so think about how to implement minimax first. As we saw in the simple case of tic-tac-toe, it is useful to employ the minimax algorithm assuming the opponent is a perfect "minimizing" agent. In practice, an algorithm with the perfect opponent assumption deviates from reality when playing a **sub-par opponent** making silly moves, but still leads to the desired outcome of never losing. If the deviation goes the other way, however, (a "maximax" opponent in which the opponent wants us to win), winning is obviously not guaranteed.

III. Using the Skeleton Code

The skeleton code includes the following files. Note that you will only be working in **one** of them, and the rest are read-only:

- **Read-only:** `GameManager.py`. This is the driver program that loads your Computer AI and Player AI and begins a game where they compete with each other. See below on how to execute this program.
- **Read-only:** `Grid.py` This module defines the Grid object, along with some useful operations: `move()`, `getAvailableCells()`, `insertTile()`, and `clone()`, which you may use in your code. These are by no means the most efficient methods available, so if you wish to strive for better performance, feel free to ignore these and write your own helper methods in a separate file.
- **Read-only:** `BaseAI.py` This is the base class for any AI component. All AIs inherit from this module, and implement the `getMove()` function, which takes a Grid object as parameter and returns a move (there are different "moves" for different AIs).
- **Read-only:** `ComputerAI.py`. This inherits from BaseAI. The `getMove()` function returns a computer action that is a tuple (x, y) indicating the place you want to place a tile.
- **Writable:** `IntelligentAgent.py`. You will create this file. The IntelligentAgent class should inherit from BaseAI. The `getMove()` function to implement must return a number that indicates the player's action. In particular, **0 stands for "Up", 1 stands for "Down", 2 stands for "Left", and 3 stands for "Right".** This is where your player-optimizing logic lives and is executed. Feel free to create submodules for this file to use, and include any submodules in your submission.
- **Read-only:** `BaseDisplayer.py` and `Displayer.py`. These print the grid.

To test your code, execute the game manager like so: `$ python3 GameManager.py`

The progress of the game will be displayed on your terminal screen with one snapshot printed after each move that the Computer AI or Player AI makes. Your Player AI is allowed **0.2 seconds** to come up with each move. The process continues until the game is over; that is, until no further legal moves can be made. At the end of the game, the **maximum tile value** on the board is printed.

IMPORTANT: Do not modify the files that are specified as read-only. When your submission is graded, the grader will first automatically **over-write** all read-only files in the directory before executing your code. This is to ensure that all students are using the same game-play mechanism and computer opponent, and that you cannot "work around" the skeleton program and manually output a high score.

IV. What You Need To Submit

Your job in this assignment is to write `IntelligentAgent.py`, which intelligently plays the 2048-puzzle game. Here is a snippet of **starter code** to allow you to observe how the game looks when it is played out. In the following "naive" Player AI. The `getMove()` function simply selects a next move in random out of the available moves:

```
import random
from BaseAI import BaseAI

class IntelligentAgent(BaseAI):
    def getMove(self, grid):
      # Selects a random move and returns it
      moveset = grid.getAvailableMoves()
      return random.choice(moveset)[0] if moveset else None
```

Of course, that is indeed a very naive way to play the 2048-puzzle game. If you submit this as your finished product, you will likely receive a low grade. You should implement your Player AI with the following points in mind:

- Employ the **expectiminimax algorithm**. This is a requirement. There are many viable strategies to beat the 2048-puzzle game, but in this assignment we will be using the expectiminimax algorithm. Note that 90% **of tiles placed by the computer are 2's**, while the remaining 10% **are 4's**. It may be helpful to first implement regular minimax.

- Implement **alpha-beta pruning**. This is a requirement. This should speed up the search process by eliminating irrelevant branches. In this case, is there anything we can do about move ordering?

- Use **heuristic functions**. What is the maximum height of the game tree? Unlike elementary games like tic-tac-toe, in this game it is highly impracticable to search the entire depth of the theoretical game tree. To be able to cut off your search at any point, you must employ **heuristic functions** to allow you to assign approximate values to nodes in the tree. Remember, the time limit allowed for each move is 0.2 seconds, so you must implement a systematic way to cut off your search before time runs out.

- Assign **heuristic weights**. You will likely want to include more than one heuristic function. In that case, you will need to assign weights associated with each individual heuristic. Deciding on an appropriate set of weights will take careful reasoning, along with careful experimentation. If you feel adventurous, you can also simply write an optimization meta-algorithm to iterate over the space of weight vectors, until you arrive at results that you are happy enough with.

V. Important Information

Please read the following information carefully. Before you post a clarifying question on the discussion board, make sure that your question is not already answered in the following sections.

1. Note on Python 3

We will only accept homeworks in python 3 (python 2 is being discontinued).

To test your algorithm in Python 3, execute the game manager like so: `$ python3 GameManager.py`

2. Basic Requirements

Your submission **must** fulfill the following requirements:

- You must use adversarial search in your IntelligentAgent (expectiminimax with alpha-beta pruning).

- You must provide your move within the time limit of 0.2 seconds.

- You must name your file `IntelligentAgent.py` (Python 3).

- Your grade will depend on the maximum tile values your program usually gets to.

3. Grading Submissions

Grading is exceptionally straightforward for this project: **the better your Player AI performs, the higher your grade.** While this is straightforward, we admit that this Adversarial Search project is the most difficult project in this class because of its open-endedness. Your Player AI will be pitted against the standard Computer AI for a total of **10 games**, and the **maximum tile value** of each game will be recorded. Among the 10 runs, we pick and average **top 5** maximum tile values. Based on the average of these 5 maximum tile values, your submission will be assessed out of a total of **100 points**.

- Submissions that are no better than **random** will receive a score of zero.
- **Submissions which contain two 1024 runs and three 2048 runs will receive full credit.** For example, [256, 512, 512, 512, 1024, 1024, 1024, 2048, 2048, 2048] will receive full credit.
- Submissions that fall somewhere in between will receive partial credit on a **logarithmic scale**. That is, every time you manage to double your average maximum tile value, you will be moving your final grade up in equally-spaced notches (instead of doubling as well). For other credit examples, please see the FAQs.

VI. Optional Heuristics - In Addition to Required Heuristics

Your heuristics are key to your IntelligentAgent's consistent success. Watch some pro 2048 players' videos. Observe patterns you might try enforcing via heuristics, and compare these players' performances to that of your IntelligentAgent as a potential benchmark of whether your AI is performing well or not. Take into consideration both quantitative and qualitative measures, such as:

- the absolute value of tiles,
- the difference in value between adjacent tiles,
- the potential for merging of similar tiles,
- the ordering of tiles across rows, columns, and diagonals

We also encourage you to research 2048-specific heuristics.

This repo (`http://ovolve.github.io/2048-AI/`) is a basic 2048 AI written in JavaScript. Your IntelligentAgent should do much better. This particular StackOverflow thread (`https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048`) provides the concepts of "Monotonicity" and "Smoothness" which you may choose to implement.

Whatever ideas you use, remember to use expectiminimax and alpha-beta pruning as instructed to receive full credit.

VII. Before You Submit

- **Make sure** your code executes. In particular, make sure you name your file correctly according to the instructions specified above, especially regarding different Python versions.
- **Make sure** your `IntelligentAgent.py` does not print anything to the screen. Printing gameplay progress is handled by `Grid.py`, and there should ideally be nothing else printed.