Assignment #2: Adversarial Search and Games

ACADEMIC HONESTY

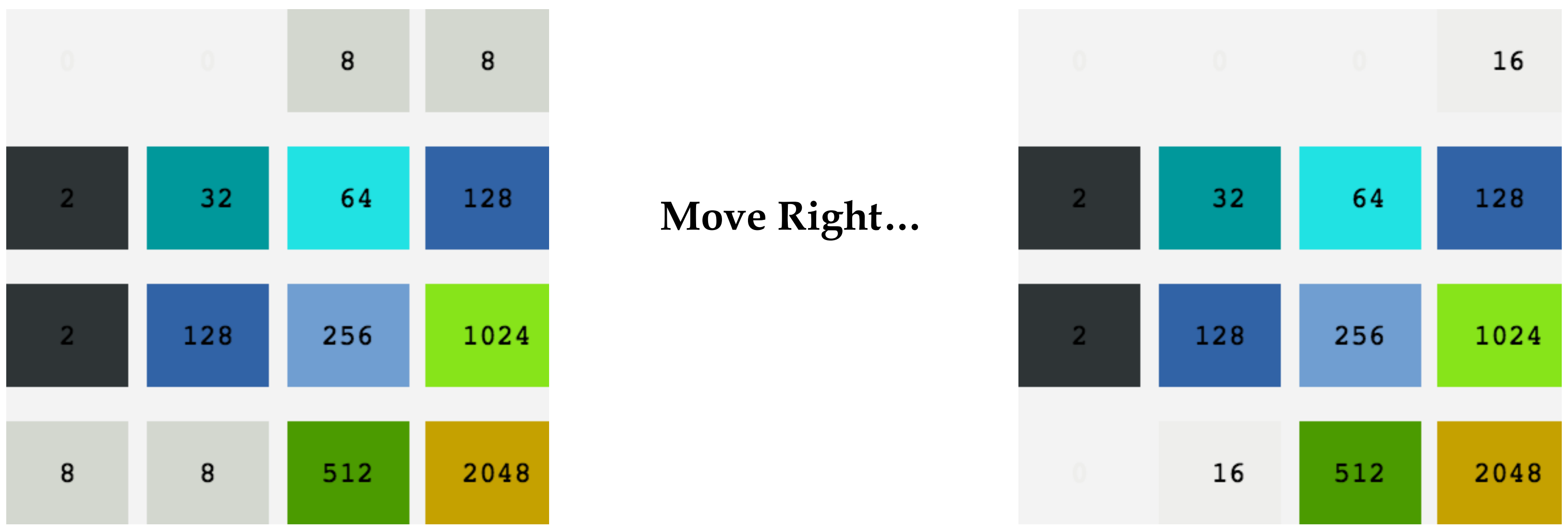
As usual, the standard honor code and academic honesty policy applies. We will be using automated **plagiarism detection** software to ensure that only original work is given credit. Submissions isomorphic to (1) those that exist anywhere online, (2) those submitted by your classmates, or (3) those submitted by students in prior semesters, will be detected and considered plagiarism.

INSTRUCTIONS

In this assignment you will create an agent to intelligently play the **2048-puzzle** game, using more advanced techniques to probe the search space than the simple methods used in the previous assignment. If you have not played the game before, you may do so at **gabrielecirulli.github.io/2048** to get a sense of how the game works. You will implement an adversarial search algorithm that plays the game intelligently, perhaps much more so than playing by hand. Please read all sections of the instructions carefully.

I. Introduction

An instance of the 2048-puzzle game is played on a **4×4 grid**, with numbered tiles that slide in all four directions when a player moves them. Every turn, a new tile will randomly appear in an empty spot on the board, with a value of either 2 or 4. Per the input direction given by the player, all tiles on the grid slide as far as possible in that direction, 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 flight, they merge into a single tile, valued at the sum of the two original tiles that collided. The resulting tile cannot merge with another tile again in the same move.



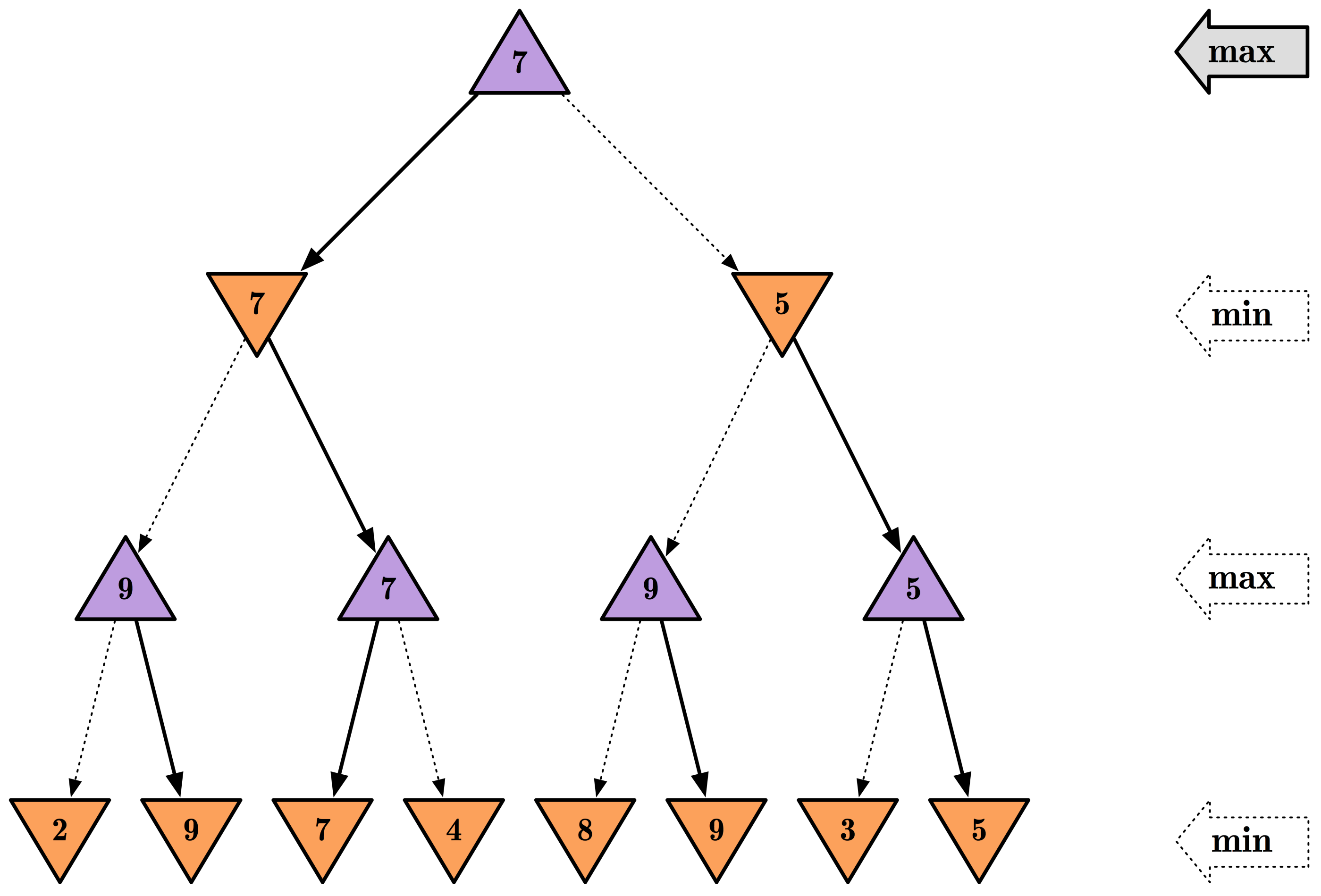
In the first assignment, you had ample experience with the process of abstracting ideas and designing functions, classes, and data structures. The goal was to get familiar with how objects, states, nodes, functions, and implicit or explicit search trees are implemented and interact in practice. This time, the focus is strictly on the ground-level details of the algorithms. You will be provided with all the **skeleton code** necessary to get started, so that you can focus solely on optimizing your algorithm.

With typical board games like chess, the two players in the game (i.e. the "Computer AI" and the "Player") take similar actions in their turn, and have similar objectives to achieve in the game. In the 2048-puzzle game, the setup is inherently **asymmetric**; that is, the computer and player take drastically different actions in their turns. Specifically, the computer is responsible for placing random tiles of 2 or 4 on the board, while the player is responsible for moving the pieces. However, adversarial search can be applied to this game just the same.

II. Algorithm Review

Before you begin, review the lecture slides on **adversarial search**. Is this a zero-sum game? What is the minimax principle? In the 2048-puzzle game, the computer AI is technically not "adversarial". In particular, all it does is spawn random tiles of 2 and 4 each turn, with a designated probability of either a 2 or a 4; it certainly does not specifically spawn tiles at the most inopportune locations to foil the player's progress. However, we will create a "Player AI" to play **as if** the computer is completely adversarial. In particular, we will employ the **minimax algorithm** in this assignment.

Remember, in game-playing we generally pick a **strategy** to employ. With the minimax algorithm, the strategy assumes that the computer opponent is perfect in minimizing the player's outcome. Whether or not the opponent is actually perfect in doing so is another question. As a general principle, how far the actual opponent's actual behavior deviates from the assumption certainly affects how well the AI performs [**1**]. However, you will see that this strategy works well in this game. In this assignment, we will implement and optimize the minimax algorithm.



[**1**] As we saw in the case of a simple game of tic-tac-toe, it is useful to employ the minimax algorithm, which assumes that the opponent is a perfect "minimizing" agent. In practice, however, we may encounter a **sub-par opponent** that makes silly moves. When this happens, the algorithm's assumption deviates from the actual opponent's behavior. In this case, it still leads to the desired outcome of never losing. However, if the deviation goes the other way (e.g. suppose we employ a "maximax" algorithm that assumes that the opponent wants us to win), then the outcome would certainly be different.

III. Using The Skeleton Code

To let you focus on the details of the algorithm, a skeleton code is provided to help you get started, and to allow you to test your algorithm on your own. The skeleton code includes the following files. Note that you will only be working in **one** of them, and the rest of them 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 available to get you started, but they are by no means the most efficient methods available. 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, i.e. a tuple (x, y) indicating the place you want to place a tile.

- **Writable**: `PlayerAI.py`. You will create this file, and this is where you will be doing your work. This should inherit from BaseAI. The `getMove()` function, which you will need to implement, returns 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". You need to create this file and make it as intelligent as possible. You may include other files in your submission, but they will have to be included through this file.

- **Read-only**: `BaseDisplayer.py` and `Displayer.py`. These print the grid.

To test your code, execute the game manager like so:

```
$ python 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. The 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 `PlayerAI.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:

```
from random import randint
from BaseAI import BaseAI

class PlayerAI(BaseAI):
    def getMove(self, grid):
        moves = grid.getAvailableMoves()
        return moves[randint(0, len(moves) - 1)] if moves 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 grade of zero. You should implement your Player AI with the following points in mind:

- Employ the **minimax algorithm**. This is a requirement. There are many viable strategies to beat the 2048-puzzle game, but in this assignment we will be practicing with the minimax algorithm.

- 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?

- In your readme, briefly **compare your results** between vanilla minimax and minimax with alpha-beta pruning. How can you tell if your pruning is actually helping?

- 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 will 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 are feeling 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

Each file in the skeleton code actually comes in **two flavors**: `[filename].py` (written in Python 2) and `[filename]_3.py` (written in Python 3). If you prefer to develop in Python 3, you will be using the latter version of each file in the skeleton code provided. In addition, you will have to name your player AI file `PlayerAI_3.py` as well, so that the grader will be alerted to use the correct version of Python during grading. For grading purposes, please only submit one of the following, but not both:

- `PlayerAI.py` (developed in Python 2, and relying on the Python 2 version of each skeleton code file), or
- `PlayerAI_3.py` (developed in Python 3, and relying on the Python 3 version of each skeleton code file).

If you submit both versions, the grader will only grade one of them, which probably not what you would want. To test your algorithm in Python 3, execute the game manager like so:

```
$ python3 GameManager_3.py
```

2. Basic Requirements

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

- You must use adversarial search in your PlayerAI (minimax with alpha-beta pruning).
- You must provide your move within the time limit of 0.2 seconds.
- You must name your file `PlayerAI.py` (Python 2) or `PlayerAI_3.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. Specifically, 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. Based on the average of these maximum tile values, your submission will be assessed out of a total of **200 points**.

- Submissions that are no better than **random** will receive a score of zero.
- Submissions for which the average maximum tile value falls **halfway** between 1024 and 2048 will receive full credit. As an approximation, you want to strive for half of your games to achieve 2048.

- Submissions that fall somewhere in between will receive partial credit on a **logarithmic** scale. That is, every time you double your average maximum tile, you will move your final grade up in equally-spaced notches (instead of doubling as well).

4. Evaluation Functions

Think very carefully about your heuristic valuations; these will likely make or break your player. Consider both qualitative and quantitative 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, … , etc.

Hints? **Here** is a basic AI written in JavaScript. You are supposed to do better! More hints? Here is an interesting StackOverflow **discussion**, which may be helpful. There are lots of great ideas everywhere, but remember that you are required to start with the minimax algorithm and alpha-beta pruning in this assignment.

5. Enforcing Time Limits

In the game manager, you will see that we enforce the time limit strictly. If the `getMove()` function takes longer than 0.2 seconds to return, the game will **immediately terminate**, and the maximum tile value will be assessed pre-maturely. The same goes for when your submission is graded. You are responsible for making sure that `getMove()` does not exceed the time limit. Once a violation is detected, that game is immediately over.

There are multiple tools out there to help you clock time. Make sure that you are relying on **CPU time**, instead of **wall time**, as your clocking mechanism. If you look at `GameManager.py`, you will see that we are using `time.clock()` on our end to enforce the limit. If you want to be safe, then use exactly that for your own implementation-side limit as well.

Please **do not** rely on a depth-limit as a basis for limiting the running time. While your algorithm may manage to abide by the time limit via a depth limit on your own computer, there is no guarantee that it will on other machines, in particular on the grading machine.

6. Skeleton Efficiency

As the astute student might observe, the skeleton code provided may not be the most efficient. In particular, the operations: `move()`, `getAvailableCells()`, `insertTile()`, and `clone()` are by no means the most efficient implementations available for achieving the desired functionality. However, the purpose of the assignment is to gain practice in adversarial search and heuristic evaluation, and not to optimize function- and object-design.

Everyone will be using the same skeleton code. If you wish, you are free to write your own helper methods in a separate file (remember, `Grid.py` is read-only). However, this is by no means necessary; in fact, students have written player AIs that have beaten the game handily by employing the given methods. You should focus your efforts on developing a smart algorithm **first**. If you have additional time, or as an extra boost (or last resort...), you are certainly encouraged to improve these methods by writing your own efficient ones.