Artificial Intelligence Spring 2018
Homework 2: Adversarial Search
Due Date: Tuesday, March 20 at 11:59pm

ACADEMIC HONESTY

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

SUBMISSION

You will submit one zip file, named **hw2_myUNI.zip**, which contains two files:

- written.pdf, and
- driver.py or driver_3.py

You can submit as many times as you like before the deadline. Only your most recent submission will be graded. The written submission can be handwritten and scanned (PDF scanning mobile apps typically work well), or typed, but it must be legible in order to receive credit.

WRITTEN

Question 1: Association Rules

Consider the following items and transactions.

Item id	Item
1	diapers
2	beer
3	water
4	coffee
5	milk

TID	Transaction
100	1 2 4 5
200	2 5
300	1 2 4 5
400	1 2 5
500	1 2 3 4 5
600	3 4 5

1. For a minimum support of 50%, use the **Apriori algorithm** to find all frequent itemsets in the transaction table.

2. How many scans of the dataset were needed to find all frequent itemsets? What does this number represent?

3. For a minimum confidence of 80%, use the Apriori algorithm to find all strong association rules (report support and confidence) of the form:
   `Item 1 → Item 2 (support, confidence)`
   `Item 1 and item 2 → item 3 (support, confidence)`

4. Some variants of the Apriori algorithm leverage information provided by transaction identifiers (TIDs). These variants associate each frequent *k*-itemset with the set of TIDs in which it appears. For example, {diapers} is a frequent 1-itemset (set size of 1) and appears in the transactions with TID's {100, 300, 400, 500}, while the 1-itemset {coffee} appears in {100, 300, 500}.
   a. How can information provided by the TID sets of the most frequent *k*-itemsets be used to calculate the frequency of the potentially most frequent *(k+1)*-itemsets?
   b. Does the Apriori step of generating frequent itemsets still require scanning the entire transaction table? Discuss pros and cons of calculating frequency with this approach.

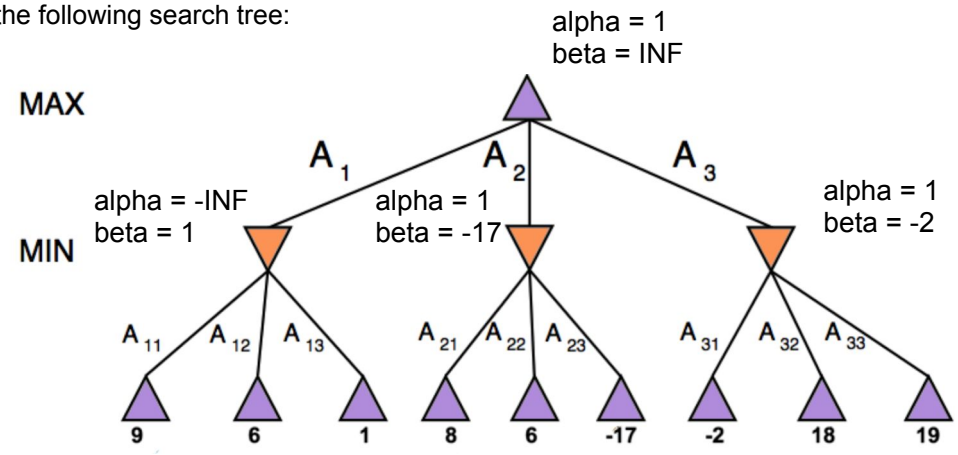
Question 2: Local Search Algorithms

There is a local search algorithm called ***simulated annealing*** which is based on statistical physics, specifically thermodynamics. In this question, we ask you to:

1. Describe simulated annealing (3-6 sentences). Feel free to use the textbook and online sources.
2. Compare simulated annealing to the genetic algorithm, and give an example application of each one. This paper provides a few examples: https://pdfs.semanticscholar.org/e893/4a942f06ee91940ab57732953ec6a24b3f00.pdf.

Question 3: Minimax and Alpha-Beta Pruning

Consider the following search tree:



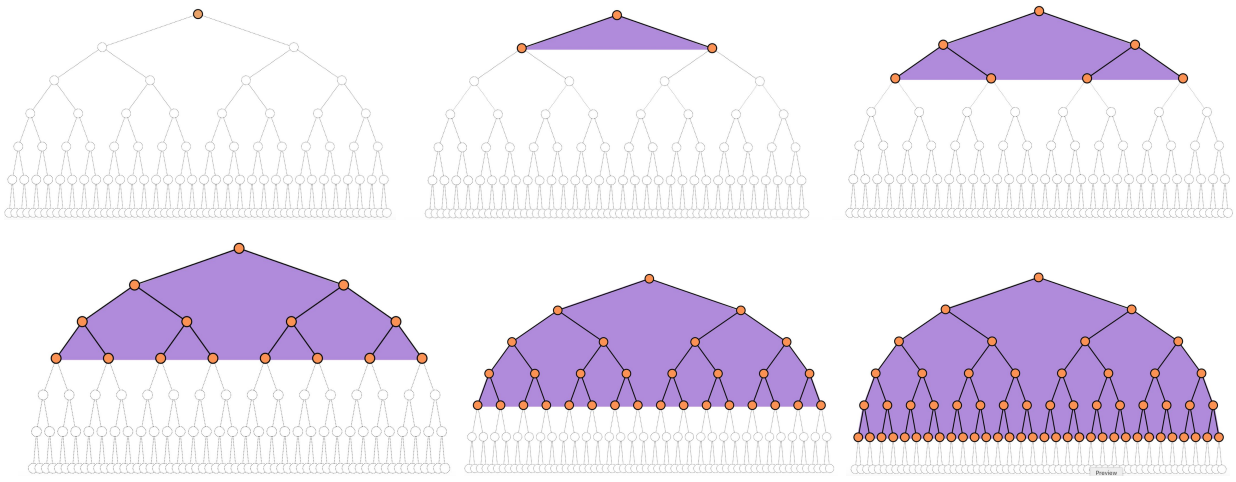
a. Using minimax, which of the three possible moves should MAX take at the root node? What is the value of Max at the root?

b. Using minimax with alpha-beta pruning, compute the value of alpha and beta at each node. Which branches are pruned?

Question 4: Iterative Deepening in Adversarial Search

Provide at least two reasons why **Iterative Depth Search** (also called **Depth First Iterative Deepening DFID**) is useful in solving adversarial two-player games like chess.

Suggested reading: Section 7 of Depth-First Iterative Deepening Korf 1985, found here: https://courseworks2.columbia.edu/courses/53318/files/folder/READING?preview=2364236



PROGRAMMING

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 **4×4 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.

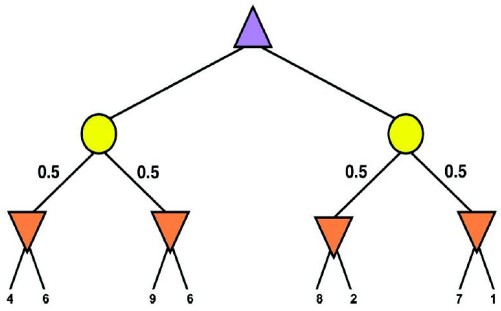
Usually, each role in a two-player games has a similar set of moves to choose from, and similar objectives (e.g. chess). In 2048 however, the player roles are inherently **asymmetric**, as the Computer AI places tiles and the Player moves them. Adversarial search can still be applied! Using your previous 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**: PlayerAI.py. You will create this file. The PlayerAI 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: $ 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. 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 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 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 he 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

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

We will only grade one version if two are given.
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 (expectiminimax 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.** 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 contains 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. 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 PlayerAI.py does not print anything to the screen. Printing gameplay progress is handled by Grid.py, and there should ideally be nothing else printed.