README.md

AI Squid Game: Trap! (Fall 2021)

The final coding challenge for COMS W4701 - Artificial Intelligence, Columbia University.

Professor : Ansaf Salleb-Aouissi

Written by: Tom Cohen

Code: Tom Cohen, Adam Lin, Gustave Ducrest, Rohith Ravindranath

DEADLINE: December 10 at 11:59 PM (EDT)

Preface and Learning Objectives

In this (super exciting) project, we will use Adversarial AI to defeat a strategic opponent at a mind game.

Learning Objective

By the end of this project you will have learned:

1. **Minimax, ExpectiMinimax, Alpha-Beta Pruning, IDS**
2. Inventive **Heuristics**!
3. Adversarial Seach under **Time Constraints**
4. How to **collaborate effectively**

1. Game Description

It is you against another contestant, and only one will survive!

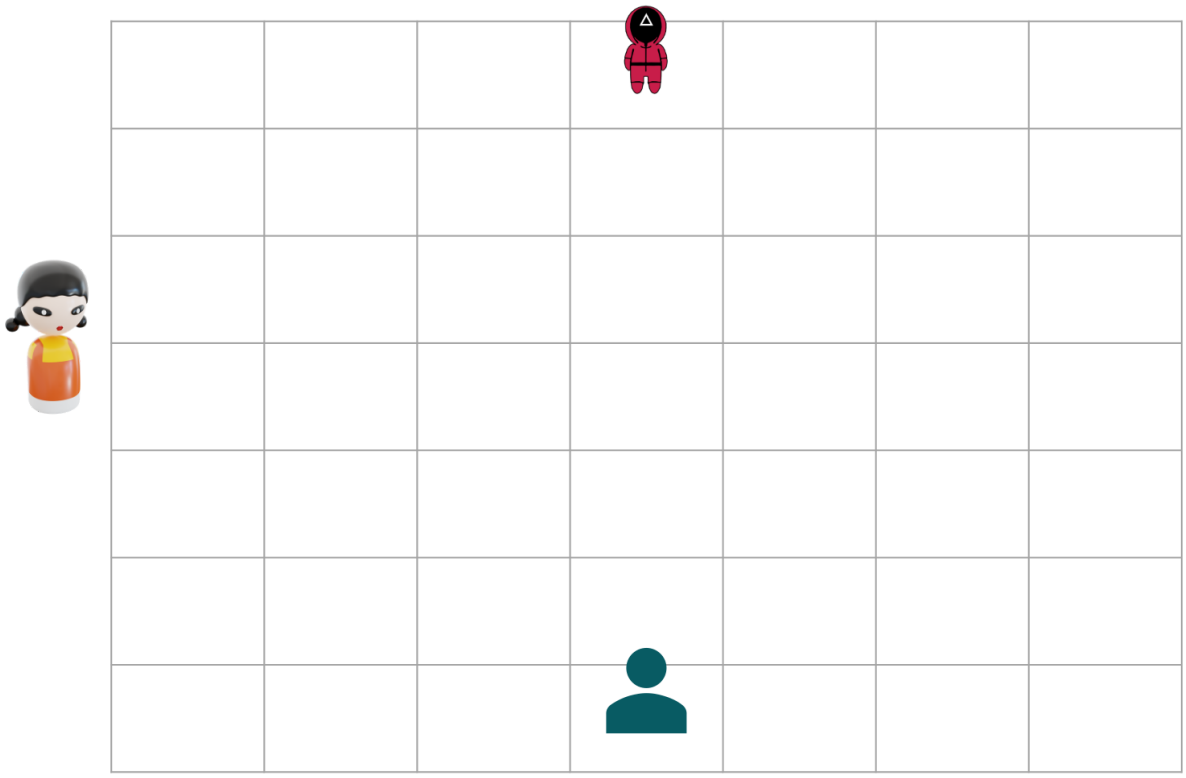
The players are placed on a gridded space and supplied with small traps which they can throw anywhere on the grid.

The game is simple: At each turn, a player has 5 seconds to act - move and throw a trap - or else the notorious Squid Game doll will shoot them down.

In order to win the game, you will need to trap the opponent, i.e., surround them with traps from all sides so that they cannot move anywhere - meaning the doll will take care of them - before they do that to you!

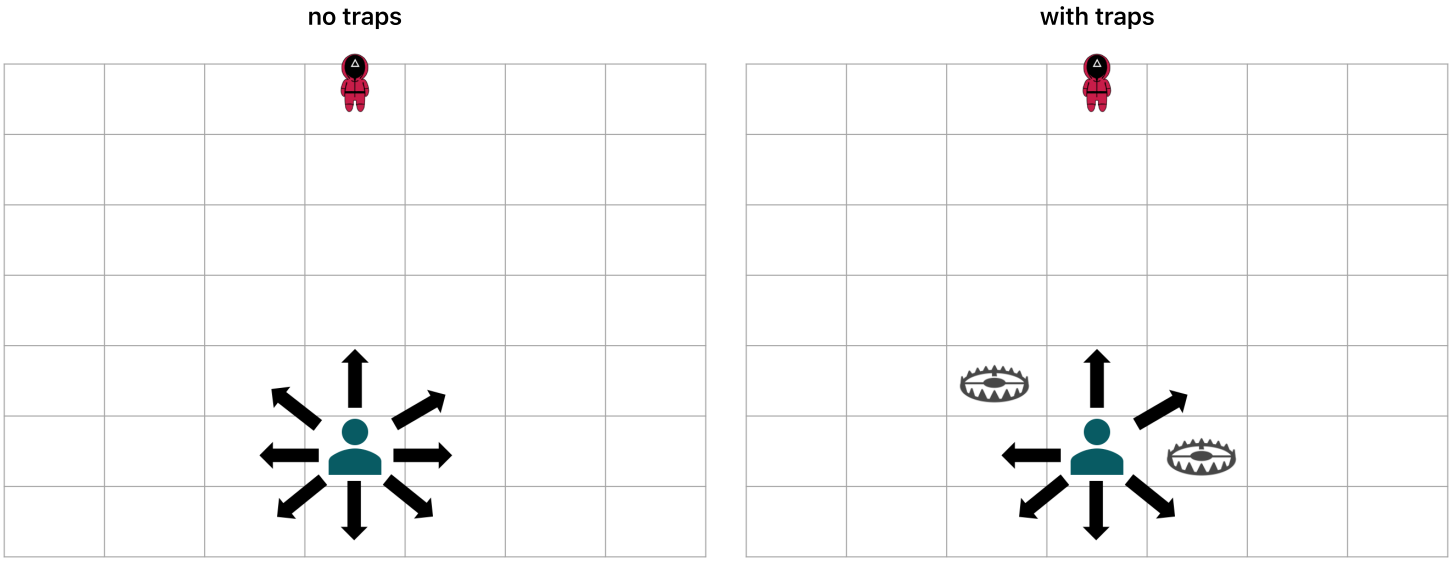
1.1 Organization

The game is organized as a two-player game on a 7x7 board space. Every turn, a player first **moves** (to make sure they wouldn't die) and then **throws a trap** somewhere on the board.



1.2 Movement

Each turn, a player can move one step in any possible direction, diagonals included (like King in chess), *so long as there is no trap placed in that cell and that it is within the borders of the grid*



1.3 Throwing a Trap

Unlike movement, a trap can be thrown to *any cell in the board*, with the exception of the Opponent's location and the Player's location. Note that throwing a trap on top of another trap is possible but useless.

However, sadly, we are not on the olympic throwing team, and our aiming abilities deteriorate with distance such that there is an increasing chance the trap will land on a cell that abuts the intended target. In fact, the chance p that it will land precisely on the desired cell is given as:

$$p = 1 - 0.05 * (\alpha - 1)$$

$$\alpha = manhattan(position, target) = |X_{position} - X_{target}| + |Y_{position} - Y_{target}|$$

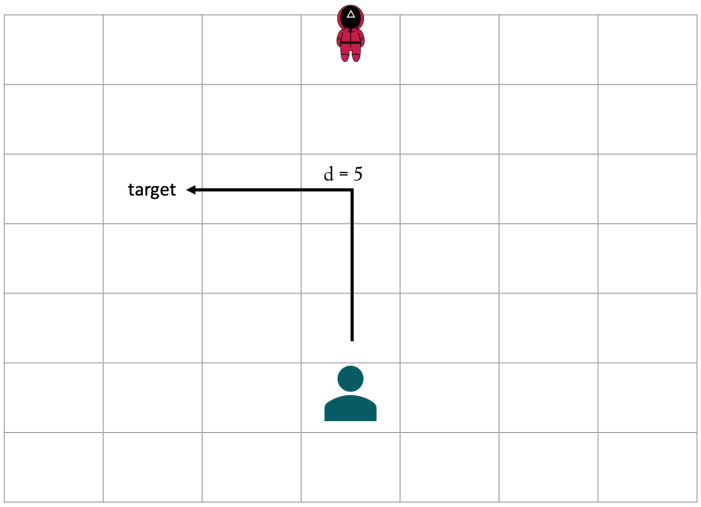
The *n* surrounding cells divide the remainder equally between them, that is:

$$p_{miss} = 1 - p = 0.05 * (\alpha - 1)$$

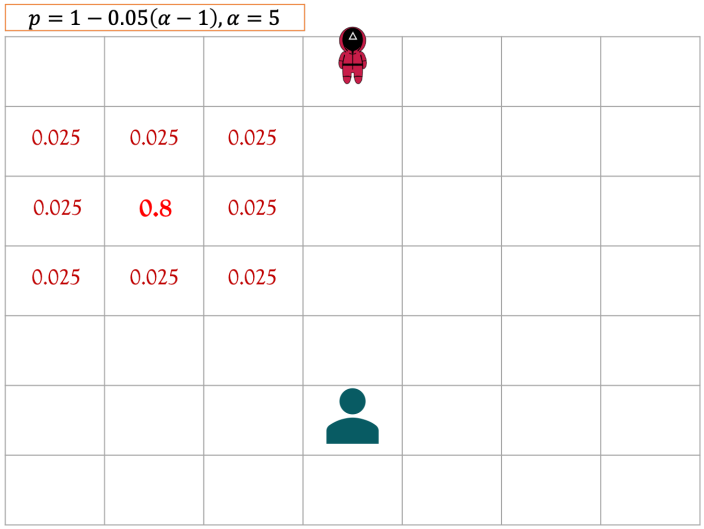
$$p_{neighbor} = \frac{1-p}{n} = \frac{0.05(\alpha - 1)}{n}$$

Example:

Target



Probabilities of trap placement

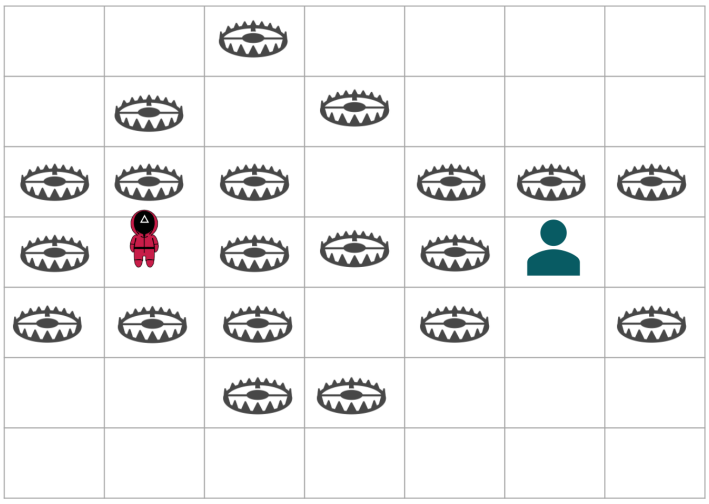


This is implemented for you in a function called `throw` which takes in the position to which you want to throw the trap and returns the position in which it lands. So no need to worry about implementing this. But you will need to consider those probabilities when coding your Minimax.

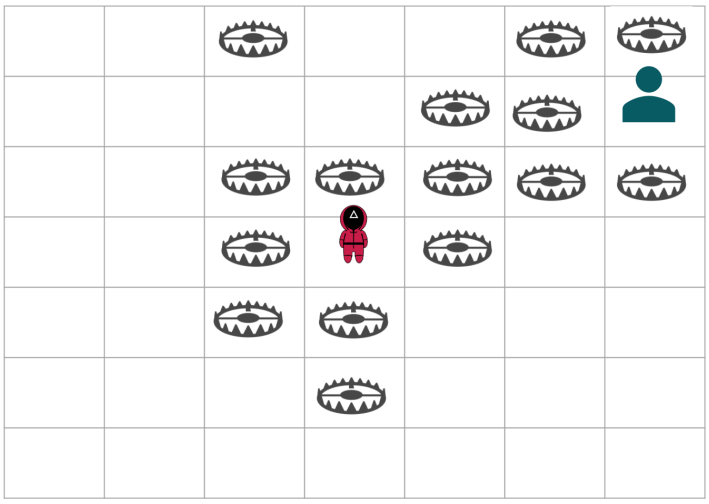
1.4 Game Over?

Again, to win, you must "Trap" you opponent so that they could not possibly move in any direction, while you still can! Examples:

Example 1 (Win)



Example 2 (Loss)



In example 1: the player can still move down or right, whereas the opponent has no valid moves.

In example 2: the player cannot move but the opponent still has a diagnoal move.

1.5 More clarifications (Will get updated as we go):

- The two players cannot stand on the same cell.
- You can throw a trap on a cell that is either free or has a trap on it (which is useless). You cannot throw a trap on a cell where there's a player.

2. What to Code

Two thoughtful actions are performed every turn: Moving maximizes the chance of survival, and Trapping maximize the chance of winning the game. Accordingly, we have two different Adversarial Search problems to compute at each turn!

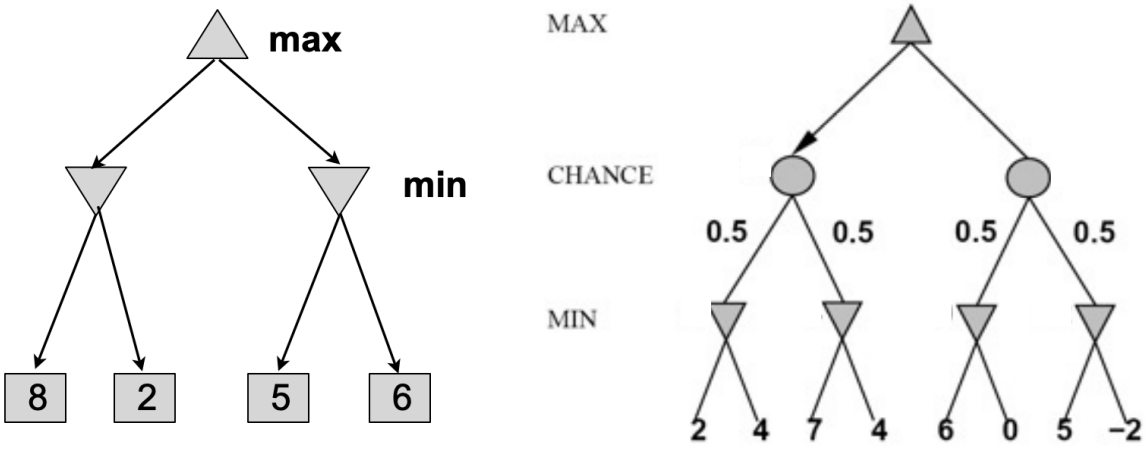
2.1 The Search Algorithm

For **each** of the search problems, you will have to implement the ExpectiMinimax algorithm with Alpha-Beta Pruning and an imposed depth limit!

2.2 Expecti--What Now?

Expectiminimax (indeed a mouthful) is a simple extension of the famous Minimax algorithm that we covered in class (Adversarial Search lecture)! The only difference is that in the Expectiminimax, we introduce an intermediate **Chance Node** that accounts for chance and uncertainty. As we have previously mentioned, throwing a trap is not always accurate, so we have to consider the consequences of that as we navigate the game. The difference between the algorithms is demonstrated in the comparison below:

Minimax **ExpectiMinimax**



For example, if we are to throw a trap and want to maximize our chance of winning, our tree will have a chance node with value *p* as well as *n* nodes with values *(1-p)/n* (see equations in 1.3: Throwing a Trap). The rest is the same!

As you can see, you are now trying to maximize an outcome, given the *chance* of it happening, that it, the *Expected Utility* of every action (and if that sounds not-necessarily-optimal to you, you are not wrong!).

Note: the Throw tree search maximizes over the Opponent's **move** actions, which in turn minimizes the Player's winning chances by moving strategically. The Move search tree, on the other hand, maximizes survival over the Opponent's **throw** actions (i.e., we ask ourselves: "Where would the Opponent throw their trap to, given that they're trying to minimze my chance of surviving?").

2.2.1 Depth Limit of 5

Given the time constraint, we would like to limit the depth of our search to ensure maximum coverage (both breadth and depth). We certainly do not want to go into deep rabbit holes (i.e., what will happen 10 rounds from now) before visiting other, more immediate options. Therefore, we impose a maximum depth of at most 5. You may opt to do less than 5 after experimenting with different values.

To do that, you can use one of the algorithms we've seen in class - the Iterative Deepening Search (IDS), which allows to control the maximum depth of the search. Pseudocode:

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  for *depth* = 0 to d; **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*,*depth*)
    if *result* ≠ cutoff **then return** *result*

That said, you may also decide to pass the depth as an argument when calling the recursive functions of the Minimax. Up to you!

2.3 Heuristics!

To evaluate the utility of an action, you will need to come up with clever heuristics for both *restricting the search space* as well as *evaluating utility.*

A few basic ones you can start with for evaluating utility:

- **Improved Score (IS):** the difference between the current number of moves Player (You) can make and the current number of moves the opponent can make.
- **Aggressive Improved Score (AIS):** Same as IS only with a 2:1 ratio applied to Opponent's moves.
- **One Cell Lookahead Score (OCLS):** The difference between the Player's sum of possible moves looking one step ahead and the Opponent's sum of possible moves looking one step ahead

You may choose to combine multiple heuristics and come up with your own heuristics!

Also, since the seach space here could be huge, one cannot possibly check all available cell in the board to find an optimal throw (imagine trying to compute the first, optimal move in chess). Therefore, you will need to think of a clever way to reduce your search scope (Hint: it could be some number of cells in the vicinity of the Opponent's location).

You do not need to implement the stochastic trap throw (it's implemented for you).

2.4 Things to think about

1. You have five seconds to both move and throw the trap. Decide how much to allocate for each (you may want to test this!)
2. You are only provided with a "dumb" player to play against. To test your player, you should create more sophisticated Opponents. No need to be extra creative here - as you're coding and improving your Player, export your code into an `Opponent_[indicative description].py` file and place it in the folder `test_players`. Then, you'll be able to import that AI player (e.g., `from test_players.Opponent_minimax_no_pruning import PlayerAI as Opponent`) into `Game.py` and make it the opponent. That will help you prepare for other people's players!
3. Limit both your search *breadth* (i.e., what is the scope of cells you're checking given Player's/Opponent's position) and depth to make sure you're covering as much as possible given the time constraint!!!
4. Keep it simple: When coding your Expectiminimax, you can disregard the neighboring cells with the smaller probabilities that your trap might land on, and only focus on the ones you consider throwing the trap to.

2.5 Hmm that was a lot. Where Do We Start?

Fear not! Here's a very good recipe:

1. Start with making a move + trap under five seconds.
2. Code basic heuristics
3. Implement a simple Minimax. Observe improvements.
4. Add depth Limit.

5. Add Alpha-beta pruning. Observe improvements!
6. Expand minimax to Expectiminimax
7. Code advanced heuristics
8. DONE.

3. 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:** `Game.py` . This is the driver program that loads your Player AI and Computer 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(),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, andimplement thegetMove()function, which takes a Grid object as parameter and returns a move (there aredifferent "moves" for different AIs).
- **Read-only:** `ComputerAI.py` . This inherits from `BaseAI` . The `getMove()` function returns a random computer action that is a tuple (x, y) indicating the location they decide to move. Similarly, the getTrap() function is where they *desire* to place the trap to (recall this is subject to chance).
- **Writable:** `PlayerAI.py` . You will code in this file. The `PlayerAI` class should inherit from `BaseAI` . The getMove() function to implement must return a tuple that indicates the player's new location. This must be a valid move, given the traps on the board as well as the size of the board. Likewise, the `getTrap()` method is for you to implement and must return a tuple of the *desired* position for the trap. This is also where your player-optimizing logic lives and is executed. Feel free to create sub-modules for this file to use, and include any sub-modules 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

To simulate the game, run `python3 Game.py` . Initially, the game is set so that to "dumb" ComputerAI player will play against each other. Change that by instantiating and using Player AI instead.

4. Grading

The competition is designed so that **all groups will be able to get a good grade regardless of their placement in the competition**. We will test your code against players of three difficulty levels: Easy, Medium, and Hard (still very doable as long as you implement everything). Those will account for at least 85% of your grade on the project. After evaluating all teams against our players, we will create a tournament where you will compete against other groups and will be rewarded more points based on the number of groups you defeat!

5. Submission

Please submit the entire folder with all the files. The name of the folder must be the UNIs of all groups members, concatenated with underscores. For example, tc1234_gd5678. Additionally, you *MUST* submit a text file, named exactly as the folder (e.g., tc1234_gd5678.txt) describing each member's contribution to the project. We may take that into account when computing individual grades in the case that one member contributed significantly more than the other(s)!

Note that we will only test the PlayerAI.py file so make sure all necessary functions are there!

5. Q&A

What are we allowed to use?**

You are definitely allowed to import commont libraries (e.g., numpy, itertools, etc.) as well as use/modify functions provided as part of the source code (e.g., the probabilistic trap throw). HOWEVER, you will need to implement all the Search Algorithms and Heuristics yourselves!

How should we divide the work?

- **A group of 2:** We suggest dividing the Expectiminimax algorithms - one group member should code the Move Expectiminimax and its accompanying heuristics and the other should code the Throw expectiminimax and its heuristics. Additionally each member should code one Opponent player to play against.
- **A groupd of 3:** We suggest one group member responsible for the Move Expectiminimax, another for the Throw Expectiminimax, and the third for both heuristics as well as coding AI Opponents to test the group's code against.