📖 **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. The project will include and expand what we have covered in class, in particular Adversarial Search. However, in order to do well in the project, groups will also have to demonstrate creativity!

Learning Objectives

By the end of this project you will have learned:

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

1. Game Description

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

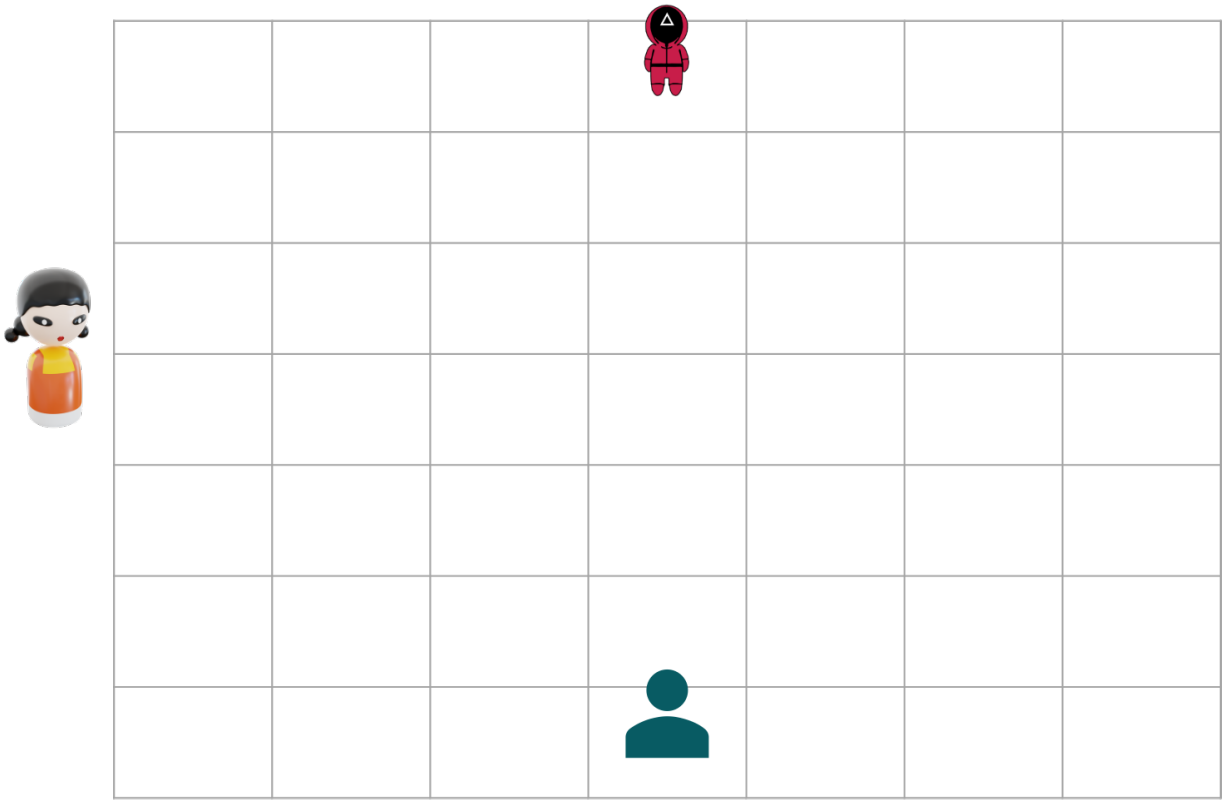
The players are placed on a gridded space and supplied with small traps which they can throw anywhere on the grid.

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.

To win the game, you will need to *trap the opponent*, that is, surround them with traps from all sides so that they cannot move - 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** and then **throws a trap** somewhere on the board.



.

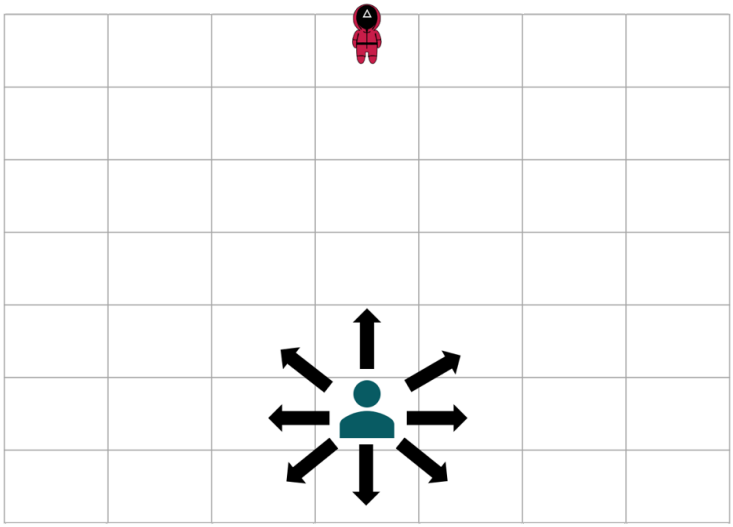
1.2 Movement

Each turn, a player can move one step in any possible direction, diagonals included (like King in chess), so long as:

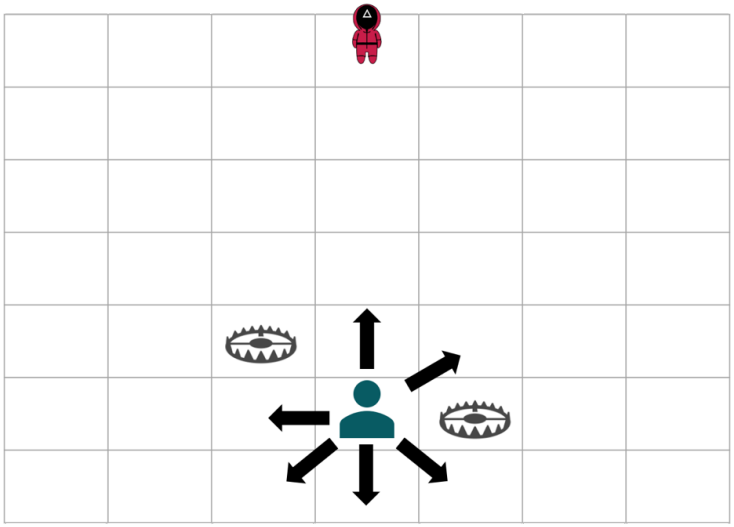
1. There is no trap placed in that cell
2. There is no player (i.e., the opponent) in that cell
3. It is within the borders of the grid

As you know, the Doll is extremely intolerant of cheating, so any invalid move will result in immediate loss!

no traps



with traps



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 onto another trap is possible but useless.

However, sadly, we are not on the olympic throwing team, and our aiming abilities deteriorate with distance. So, there is an increasing chance the trap will land on a cell abutting the intended target the further the target is from the player's location.

In fact, the chance *p* that it will land precisely on the desired cell is given as:

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

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

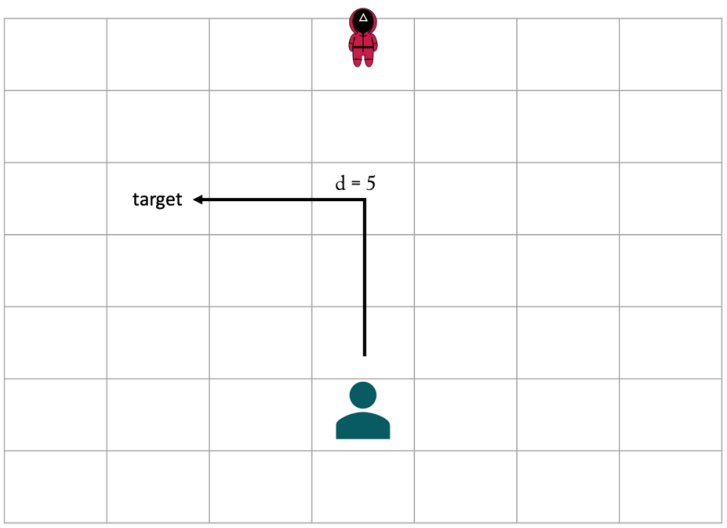
The *n* surrounding cells divide the remainder equally between them, that is:

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

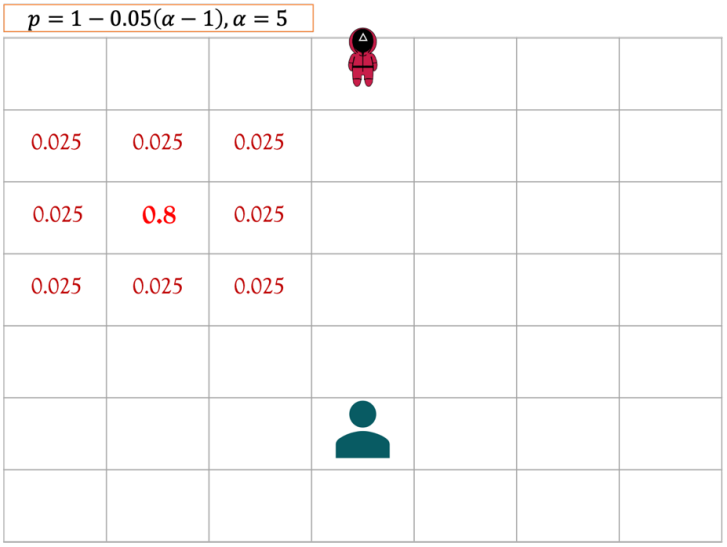
$$p_{neighbor} = \frac{1-p}{n} = \frac{0.05(\alpha - 1)}{n}$$

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.**

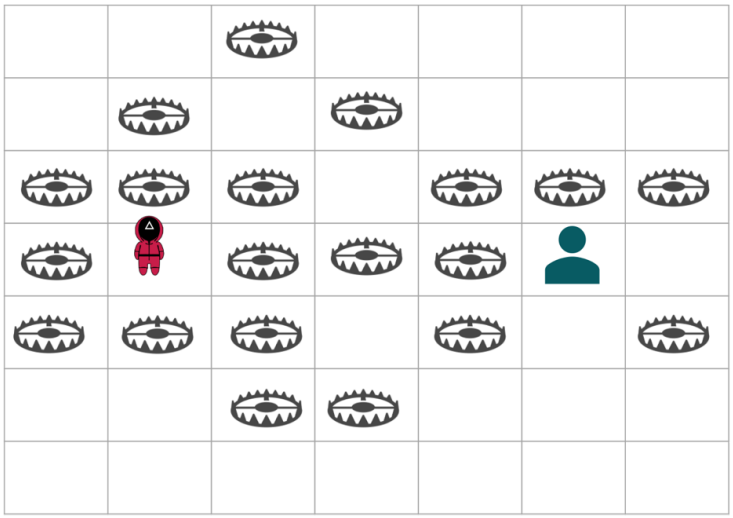
However, you will need to use the probability of success *p* when coding your Minimax (more on that soon).

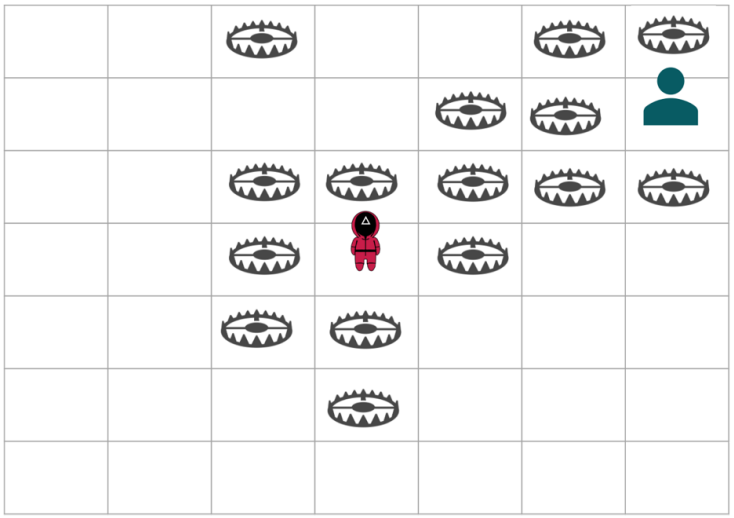
1.4 Game Over?

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. Therefore, you, technically, constitute a trap! (i.e., you can trap the opponent by surrounding them with traps and yourself).
- The function you implement should return the *intended* position for the trap. We take care of the stochastic throw!
- You will not be required to take into account the probabilities of the neighboring cells, *(1-p)/n*, only the probability of success, *p*.

2. What to Code (the fun part!)

Two thoughtful actions are performed every turn: **Moving**, which maximizes the chance of survival (or not losing), and **Trapping**, which maximizes the chance of winning the game. Accordingly, we have two *different* Adversarial Search problems to compute at each turn!

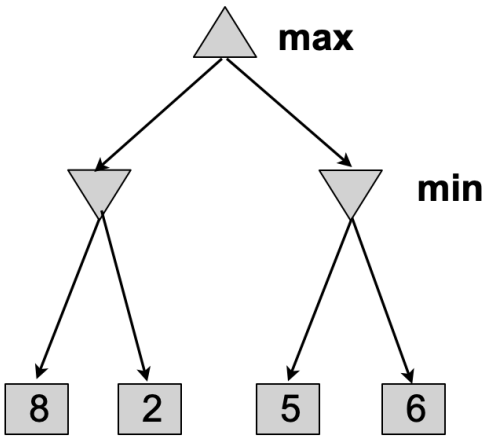
For each of the search problems, you will have to implement the **ExpectiMinimax** algorithm with **Alpha-Beta Pruning** and an imposed **depth limit**!

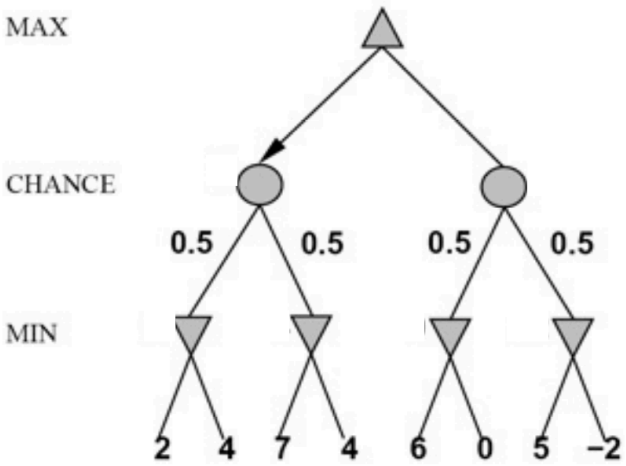
2.1 Expecti--What Now?

ExpectiMinimax (indeed a mouthful) is a simple extension of the famous Minimax algorithm that we covered in class (Adversarial Search lecture), first proposed by Donald Michie in 1966! The only difference is that in the ExpectiMinimax, we introduce an intermediate **Chance Node** that accounts for, well, chance.

As we have previously mentioned, throwing a trap is not always accurate, so we have to consider those probabilities of success as we navigate the game. The difference between the algorithms is demonstrated in the comparison below:

Minimax	ExpectiMinimax





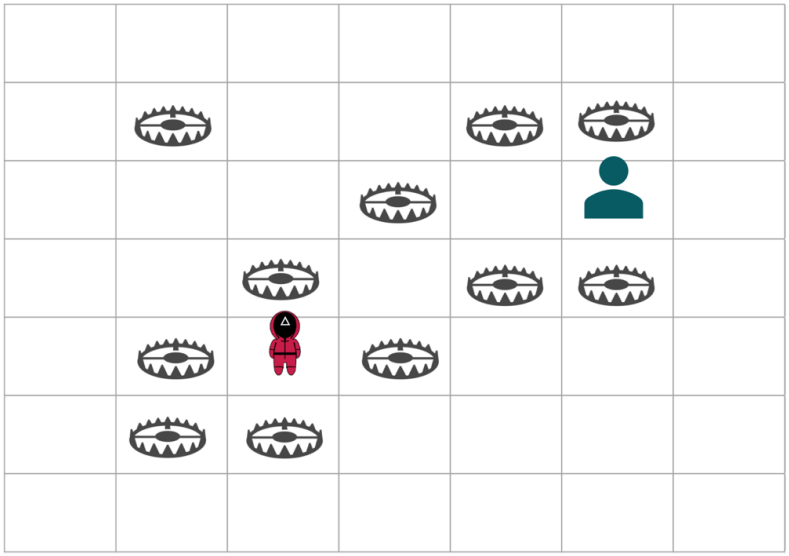
Very Important:

- The `getTrap` search tree maximizes expected utility over the Opponent's *Move* actions, which are accurate. In English, you're asking yourself "where should I place this trap if Opponent moves like a pro?" See examle 2.1.1 below for an illustration of that tree.
- The `getMove` search tree maximizes expected utility over Opponent's *Trap* actions, which are subject to chance. In English, you're trying to foresee "where should I move, if the Opponent throws traps like a pro?" For the sake of simplicity, you may assume that the Opponent is *static*, that is, it throws traps from the same position even deeper in the tree.

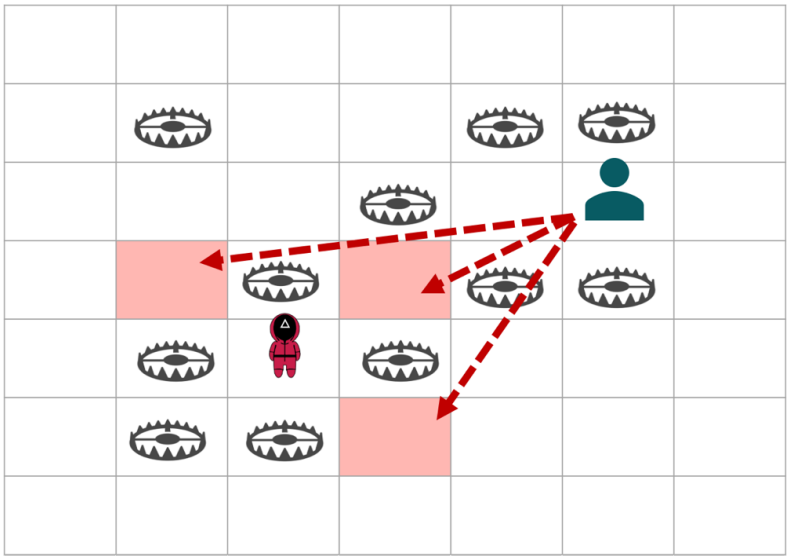
So, expect to have two distinct search trees, two distinct (yet similar) ExpectiMinimax algorithms. This is why this is a group project!

2.1.1 A concrete example

Let's say this is the current state:

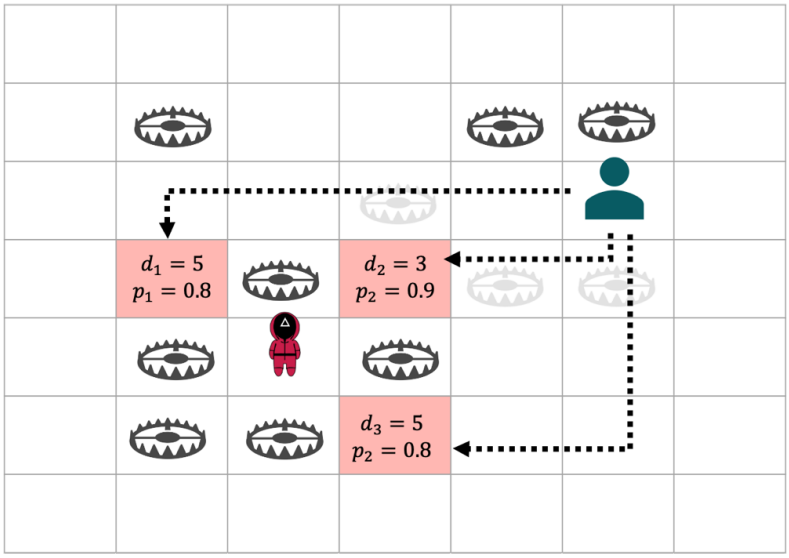


Given some heuristic, we decide to consider only the three available cells surrounding the Opponent as candidates for our trap:

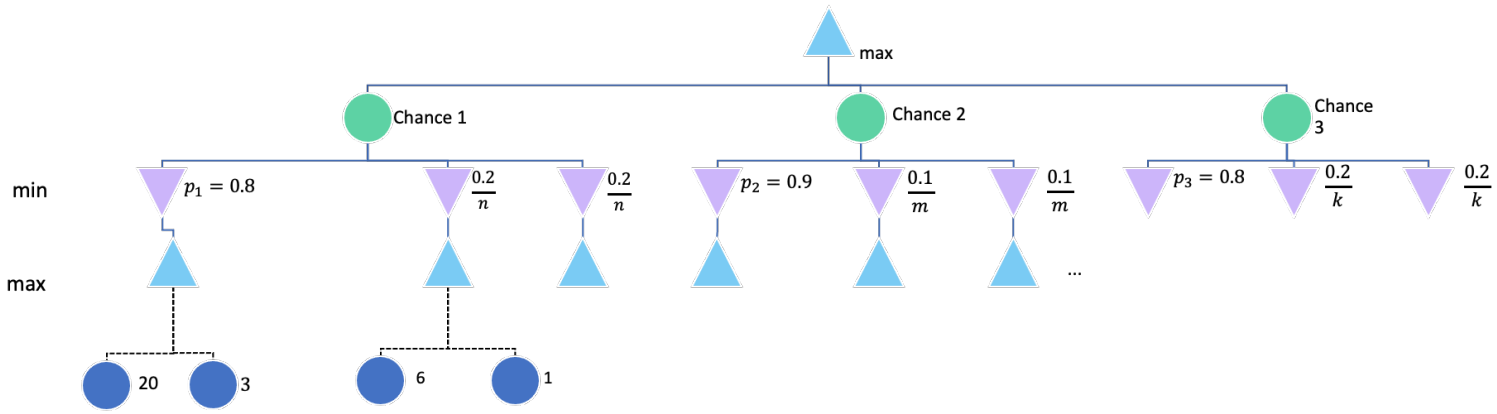


We compute the probabilities for each of the candidate positions according to the formula in 1.3, remembering that for each of those cells, there's a probability that it will land on a neighboring cell:

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



And the resulting search tree might look something like this:



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

2.2 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 too deep in the tree (i.e., what will happen 10 rounds from now) before visiting other, more immediate options. Therefore, we impose a maximum depth of 5. You may opt to do less than 5 after experimenting with different values.

To do that, there are two options:

1. You can use a famous algorithm 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*

2. You may also decide to pass `depth` as an argument when calling the recursive functions of the Minimax.

Up to you!

2.3 Heuristics!

2.3.1 Utility

Say we have reached our maximum depth of 5. Since we can't go further, we must evaluate the utility of the current state as if it is a leaf node in the tree. Since we're allowed to be selfish (at last!), we seek to evaluate: "How good is this state for US?"

To evaluate the utility of a state , we will need to come up with clever a heuristic function `utility(S : Grid)` that takes in a state and returns some numerical value that estimates the goodness of this state for *you*!

Note: The better the heuristics, the better your player will be able to navigate the game. **Do not underestimate the power of a good heuristic function.**

A few basic heuristic ideas you can start with:

- **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 want to come up with your own heuristics and/or combine multiple ones into a composite heuristic!

2.3.2 Search Space

The search space in this game could be huge (49 board cells!). One cannot reasonably consider all available cells 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.

A simple one that you can start with is the available cells around your opponent, returned by the function `.get_neighbors(opponent_position, available_only = True)` provided in `Grid.py` .

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

Fear not! Here's a very good recipe:

1. Start with making a random Move + Trap under five seconds.
2. Code basic heuristics for Moving and Trapping. Save Player and make it the Opponent from now on!
3. Implement a simple Minimax with basic Heuristics.
4. Enforce depth limit. Save Player and make it the Opponent from now on!
5. Add Alpha-beta pruning. Observe improvements!
6. Hack some more advanced heuristics of your creation. Almost there!!
7. Expand Minimax to Expectiminimax, taking probability of success into account.

DONE.

2.5 Things to think about while hacking

1. You have five seconds to move *and* throw a trap. Decide how much to allocate to each (you may want to test this!)
2. To test your Player's performance, 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. Think of and treat the two search problems as distinct though with some interaction. Draw the search trees of the two problems to get a better understanding of the recursive flow of information in each. Pay attention to where the chance node comes into play * (hint: it is not every turn!)
5. If things get too complicated, focus on the main part and ignore the rest. E.g., you do not *have* to account for the probabilities of the trap landing in a neighboring cells, only the probability that it lands where you want it to. So, your search tree can be much

simpler than the one in 2.1.1.

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-ish:** `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. By default, it loads two dummy player, so please change it to load your `playerAI()` once implemented. Also, you may play with the time limit (we'll test you on a 5-second time limit). 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 the computer decides to move to. Similarly, `getTrap()` function returns the *desired* position for the trap (recall where it lands is subject to chance).
- **Writable:** `PlayerAI.py` . You will code in this file. The `PlayerAI` class should inherit from `BaseAI` . The `getMove()` function must return an (x,y) tuple that indicates Player's new location. This must be a valid move, given the traps on the board, the players, and the size of the board. Likewise, the `getTrap()` function is for you to implement and must return an (x,y) tuple of the *desired* position for the trap. This is equivalent to the `decision` function in the Minimax pseudocode. So, expect it to call a `maximize` function.
- **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 a game, run `python3 Game.py` . Initially, the game is set so that two "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.

5. Q&A

What are we allowed to use?**

You are welcome to import libraries such as numpy, itertools, etc. (if you're uncertain, post a question on Ed) as well as use functions provided in the skeleton code. HOWEVER, you will need to implement all the 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.

6. Hints, Hacks, Bug Fixes (will get updated regularly)

- The Opponent's player number is always `3 - self.player_num` (useful when wishing to find the opponent on the board). Do not assume Opponnent will always be Player 2, because during the competition you might be!
- To make a copy of the grid/state (will be *very* useful), use `grid.clone()`
- to import a `BaseAI` to a player in the `test_players` subfolder, you can add the following lines to the top of your player file:

```
import sys
import os
# setting path to parent directory
sys.path.append(os.getcwd())
```

- There is a small bug in line ~191 of Game.py. It should be:

```
self.over = True
print(f"Tried to put trap in {intended_trap}") ## change trap to intended_trap
print("Invalid trap!")
```

- A super cool edge case: since we move before we trap, and You (Player) technically constitute a trap (since Opponent cannot move onto that cell), there is a possibility that you have trapped the opponent (and thus won) before `getTrap()` is even called, which means `getTrap()` might return `None` (depending on the scope of your search). You can address this edge case with a simple *if* statement in `getTrap()`.