Programming Assignment 3
Detailed Instructions

Overview

In this programming assignment, you're implement three different AI difficulties and collecting data about how they do against each other.

To give you some help in approaching your work for this assignment, I’ve provided the steps I implemented when building my assignment solution.

Step 1: Randomly select the board configuration

In this step, you're randomly selecting the number of bins and the number of teddy bears per bin using the appropriate game constants. The correct place to do this is in the `Board CreateNewBoard` method.

Remember, the `max` argument for the Unity `Random.Range` method is exclusive, not inclusive.

Step 2: Limit the search depth

In this step, you're limiting the depth of the minimax tree a player builds based on the difficulty setting for that player.

The simplest approach to take is to only add a child node to the tree and the `nodeList` in the `Player BuildTree` method if the child depth is <= the `searchDepth`. The root of the tree is depth 0, the root's children are depth 1, and so on. I wrote a separate method to determine the depth of a tree node by moving up the tree from the node to the root using the `Parent` property. The number of times I moved up to reach the root from the current node (the parent node of the child node I'm thinking of adding to the tree) is one less than the depth of the child node.

The code I provided to you sets both players to hard in the `DontTakeTheLastTeddy Start` method when it calls the `StartGame` method. You should modify those difficulties to see the differences in gameplay.

Step 3: Implement your heuristic evaluation function

In this step, you're implementing a reasonable heuristic evaluation function to score non-leaf nodes. The method implementing the function (I used the provided `AssignMinimaxHeuristicScore` method) needs to describe (in the comments) your motivation for each of the rules you're using to calculate the score.

To help with this, I added a `NonEmptyBins` property to the `Configuration` class to give me a list of the bin counts for the bins that aren't empty in the configuration. I also added a `NumBears` property to the `Configuration` class to give me the total number of teddy bears in the configuration.

Your evaluation function doesn't have to be perfect, but you should think of at least two scenarios for a configuration where you can assign a reasonable score. Don't forget to assign some score (0.5f is reasonable) if the configuration doesn't match any of your scenarios.

Step 4: Run multiple games

In this step, you're making the code run multiple games (you decide how many) and making it so the first player making a move alternates for each game. In other words, Player 1 makes the first move in the first game, Player 2 makes the first move in the second game, and so on.

I added a timer to the `DontTakeTheLastTeddy` class to pause between the end of one game and the start of the next one. I also added a `GameStarting` event that the `DontTakeTheLastTeddy` class invokes when it's starting a new game so the HUD can disable the game over message when a new game starts.

Step 5: Collect and display data

In this step, you're collecting and displaying data for 100 games each of easy vs easy, medium vs medium, hard vs hard, easy vs medium, easy vs hard, and medium vs hard. Display the total wins for each player for each of those combinations in a separate scene you move to once all the games are done. Set the `GameConstants.AiThinkSeconds` to 0.01f and the "between game delay" to 0.01f to make that data collection go quickly so your peer reviewers can see the results in less than approximately 1 minute. My "between game delay" was so short that I didn't see the game over message at the end of each game until the last game; that's fine.

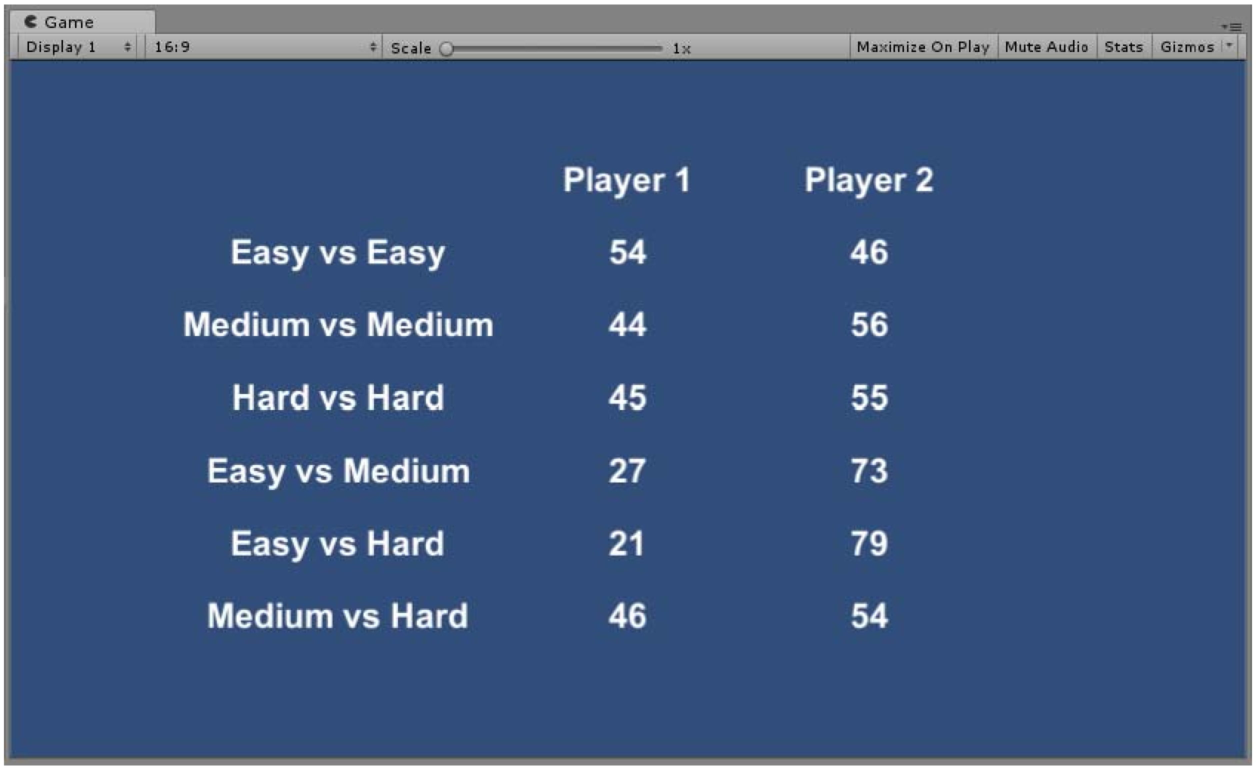
To store the data, I added a static `Statistics` class that listens for the `GameOver` event. I changed my `GameOver` event to include the winning player (as it did before) and the difficulties for Player 1 and Player 2 so the `Statistics` class could store the data for each game correctly.

In the `StartGame` method of my `DontTakeTheLastTeddy` class, I checked to see if a multiple of 100 games had been played (% comes in handy here) and set the player difficulties each time to make sure I gathered data for all the required difficulty combinations (I used a separate method to change the difficulties).

My `Statistics` class used a 2D array, where the columns were the wins for Player 1 and Player 2 and the rows were for each difficulty combination. I calculated the index by adding the difficulties, though I did have to handle one special case. My `Statistics` class also exposed a method to get the wins count for a given player and difficulty combination; I needed that method in the scene that displays the wins.

I provided the scene for you to use to display the statistics; that way, everyone's output looks the same (though the actual numbers will be different, of course).

The picture below shows my results. When the players are the same difficulty, the wins for each are reasonably close to each other. When the difficulties are different from each other, the easier difficulty consistently has fewer wins than the harder difficulty. If you get results have those characteristics, that's great.



	Player 1	Player 2
Easy vs Easy	54	46
Medium vs Medium	44	56
Hard vs Hard	45	55
Easy vs Medium	27	73
Easy vs Hard	21	79
Medium vs Hard	46	54