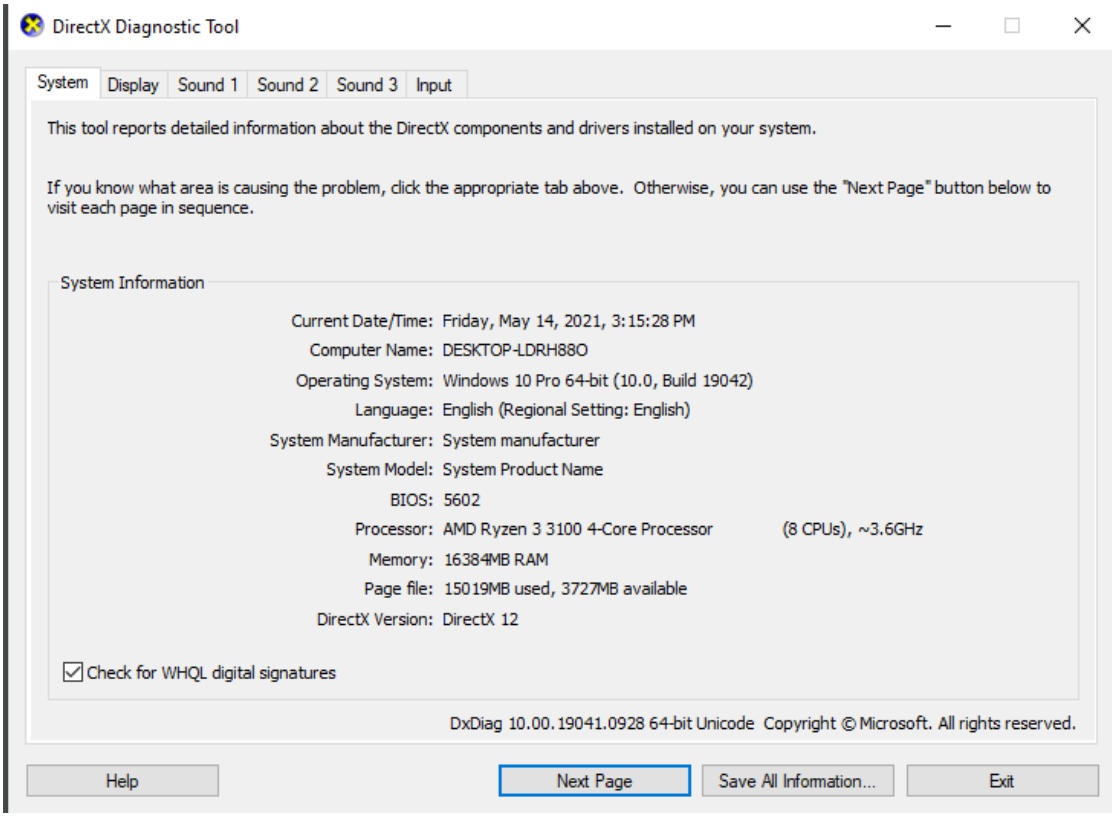
Problem Set: Trying out Searching Instructor: Neil Patrick Del Gallego

Emerson Paul P. Celestial
05/14/2021

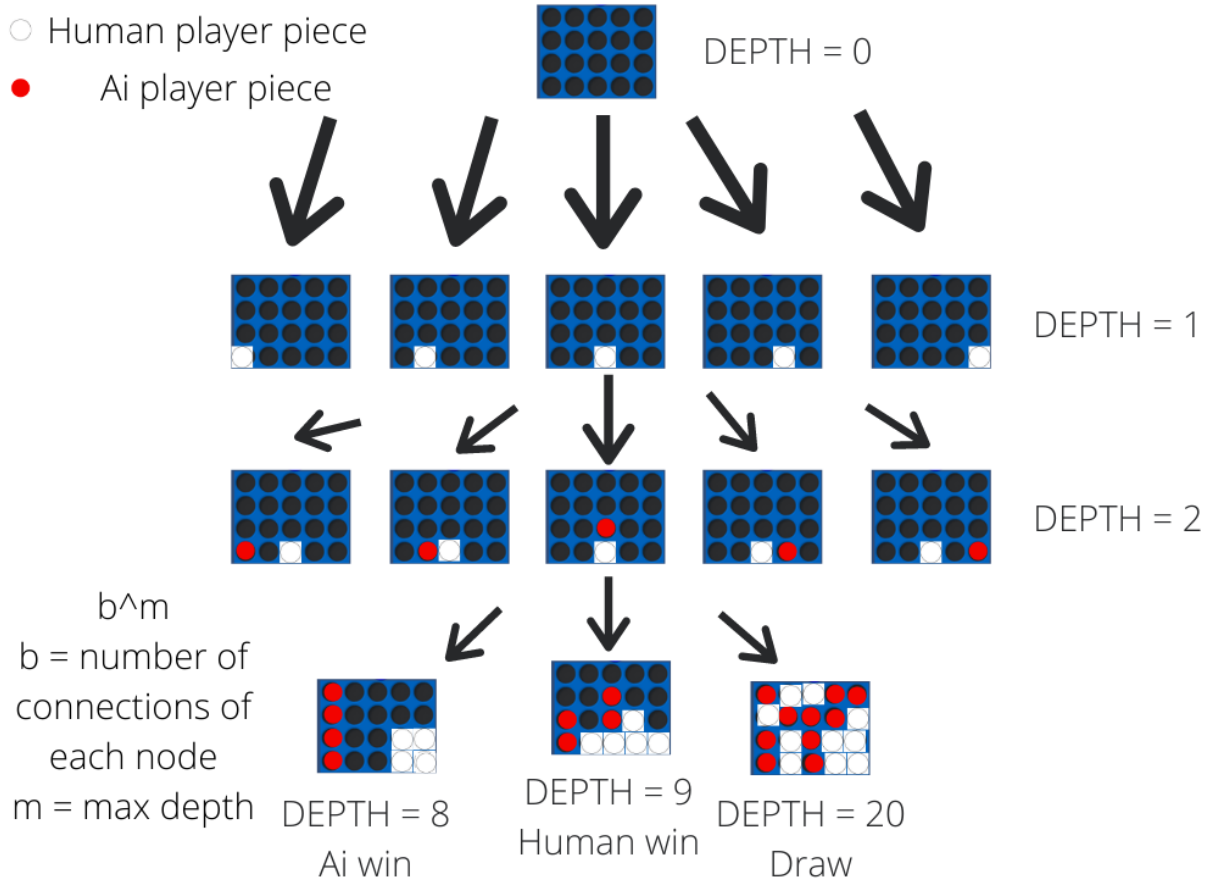
Board State Representation:

In my game platform, I've used Unity and choose 2D so that I can easily represent the pieces and operate their mechanics. On the art side, I've just looked for a free asset in the Unity store for Connect 4. This game will only need one player to play and the enemy would be the AI agent. The game will start with either the AI or the human since I made the 1st attacker of the game randomized. The Max_Depth will depend on whose player goes first (Human1st – 19 depth, AI1st – 20 depth). In my game mechanic in placing their pieces, during the player's turn, he/she should just point out to the column to where he/she want to place its move. After the human player makes a move, the AI will run its function to determine which move is the best also to foresight the best possible moves. The game can result in three different terminal states, (1) the human player win, (2) AI agent win, and (3)Draw. These states will be determined by the backend functions that check the updated board state by checking all kinds of conditions to set the game to overvalue to true.

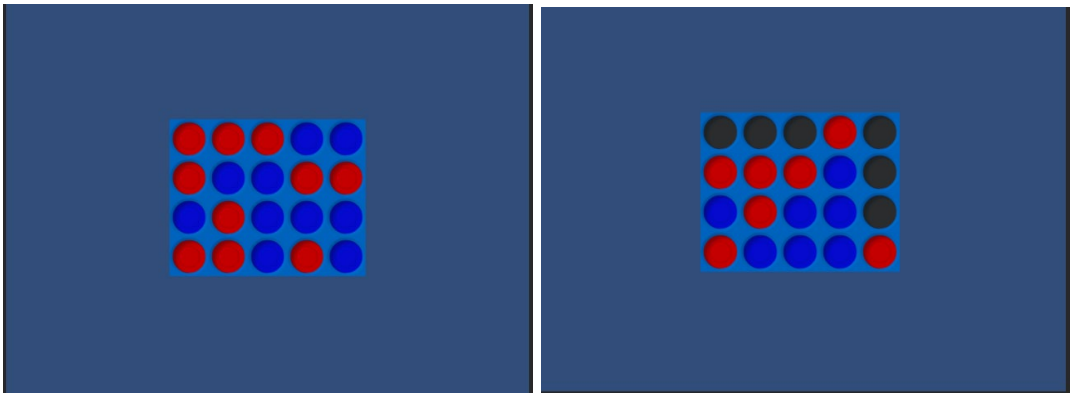
Hardware:



Board State Search Tree



This would be the representation of each node(board state) in the tree where it consists of the root node, child node, and leaf node (terminal state). This is graph tree is what the minimax function is generating wherein if the m = would be the max depth, then the agent can already have an advantage since it already knows the path to a winning terminal state. Possible terminal Board States:



DRAW AI WIN

Class Structures Used:
In my public classes, I've used BoardState, TilePosition, AdjacenyGraph, Agent, GameController to structure all the data in my game.

- In my BoardState class, this holds:
1. the position on whose player turn is,
2. the list positions (List-data handling and int[,]-representation),
3. the evaluation score of each boardState
4. current depth of the node,
5. The column and row size,
6. A function that returns a 2d array that holds the value of each tile position

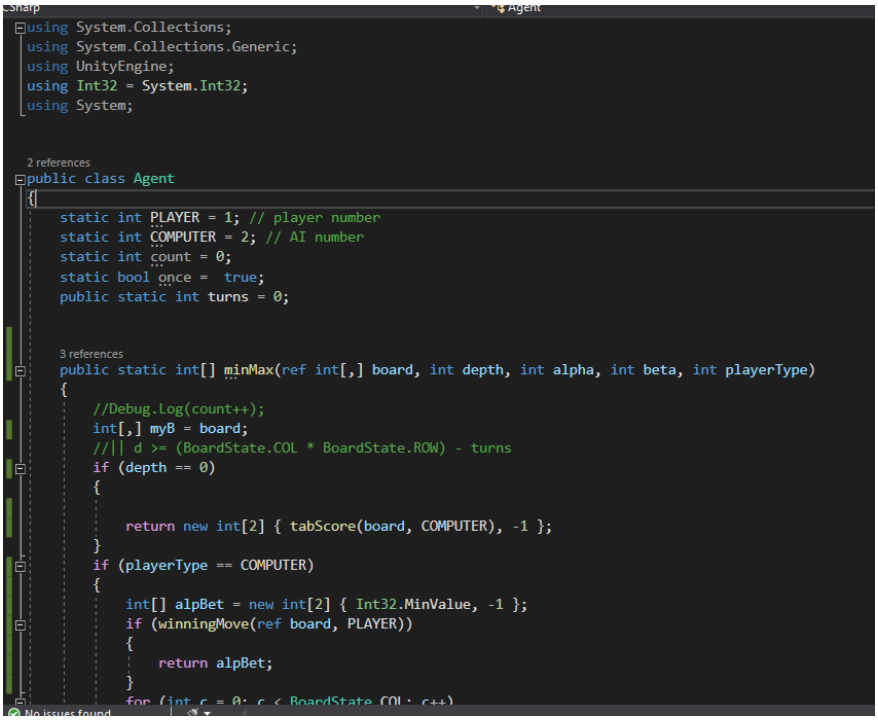
```
Assembly-CSharp                                   TilePosition
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
    2 references
5   public class TilePosition
6   {
        0 references
7       public enum Piece
8       {
9           Empty = 0,  //empty tile
10          Blue = 1, //player
11          Red = 2     //Ai
12      }
        0 references
13      public TilePosition(short row, short column)
14      {
15          this.row = row;
16          this.column = column;
17      }
18      public static string TAG = "Position";
19      public short pieceType; //0-empty 1-Human 2-Cpu
20      public short row;
21      public short column;
22  }
23
```

- In my TilePosition, this holds the:
1. pieceType that it contain(0-empty,blue-player,red-Ai agent)
2. Row and column index

```
mbly-CSharp                                       AdjacencyGraph
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
    0 references
5   public class AdjacencyGraph
6   {
7       //list of boardstates in a depth index
8       static public Dictionary<int, List<BoardState>> A;
9
10      //adds a new boardState connection
        0 references
11      public void addBoardState(int d, BoardState neighbor)
12      {
13          if (A[d] == null)
14              A[d] = new List<BoardState>();
15          A[d].Add(neighbor);
16      }
17
18      //overload connectiong multipple neighbors to the parent boardstate
        0 references
19      public void addBoardStates(int d, List<BoardState> neighbors)
20      {
21          if (A[d] == null)
22              A[d] = new List<BoardState>();
23          for (int i = 0; i < neighbors.Count; i++)
24              A[d].Add(neighbors[i]);
25      }
```

```
26
27      //returns a list of all the connections of the called boardsta
        0 references
28      public List<BoardState> connections(int d)
29      {
30          if (A[d] != null)
31              return A[d];
32          else
33              return null;
34      }
35
36      //returns a list of all the boardstate listed in the Graph
        0 references
37      public List<BoardState> allV()
38      {
39          List<BoardState> all_v = new List<BoardState>();
40
41          foreach (int key_pair in A.Keys)
42          {
43              foreach (var item in A[key_pair])
44              {
45                  all_v.Add(item);
46              }
47          }
48
49          return all_v;
50      }
51  }
```

-In my AdjacenyGraph, this holds the:
1. Dictionary list of depth with its list of boardState values(key-depth, value-BoardStates)
2. addBoardState() – adds a boardstate to the List<boardState> of the corresponding depth.
3. addBoardStates() - adds a list of boardstates to the List<boardState> of the corresponding depth.
4. Connections() – returns a List<boardState> of the called depth
5. AllV() – returns all the nodes in the graph



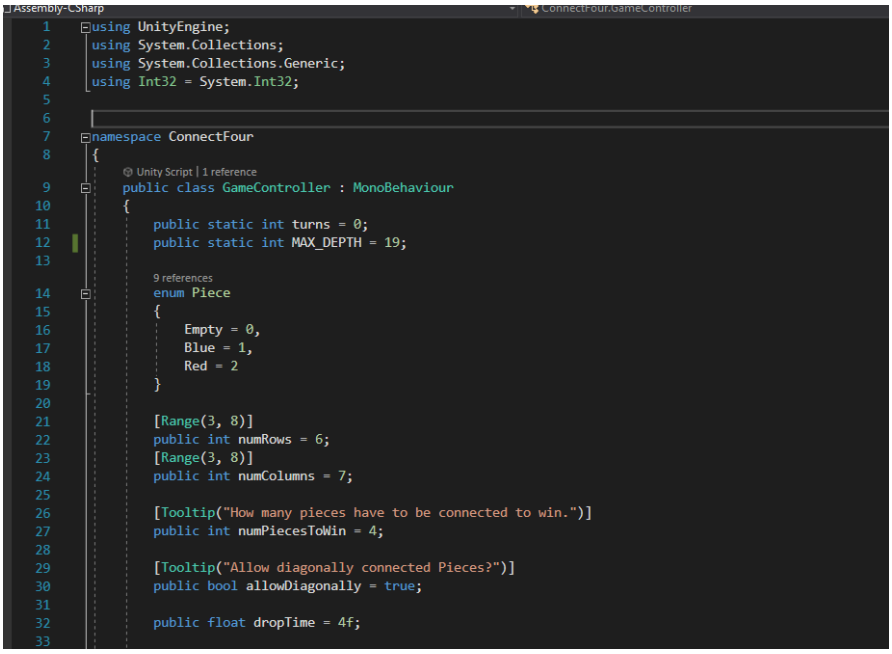
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Int32 = System.Int32;
using System;


2 references
public class Agent
{
    static int PLAYER = 1; // player number
    static int COMPUTER = 2; // AI number
    static int count = 0;
    static bool once =  true;
    public static int turns = 0;


    3 references
    public static int[] minMax(ref int[,] board, int depth, int alpha, int beta, int playerType)
    {
        //Debug.Log(count++);
        int[,] myB = board;
        //|| d >= (BoardState.COL * BoardState.ROW) - turns
        if (depth == 0)
        {

            return new int[2] { tabScore(board, COMPUTER), -1 };
        }
        if (playerType == COMPUTER)
        {
            int[] alpBet = new int[2] { Int32.MinValue, -1 };
            if (winningMove(ref board, PLAYER))
            {
                return alpBet;
            }
            for (int c = 0; c < BoardState.COL; c++)
```

-In my Agent, this holds the
1. minMax() – that returns the column with the best move.
2. constructMove() – returns a new board and sets the its depth and evalScore
3. heuristicScore() – that returns the evaluation value of the boardState



```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using Int32 = System.Int32;


namespace ConnectFour
{
    Unity Script | 1 reference
    public class GameController : MonoBehaviour
    {
        public static int turns = 0;
        public static int MAX_DEPTH = 19;

        9 references
        enum Piece
        {
            Empty = 0,
            Blue = 1,
            Red = 2
        }

        [Range(3, 8)]
        public int numRows = 6;
        [Range(3, 8)]
        public int numColumns = 7;

        [Tooltip("How many pieces have to be connected to win.")]
        public int numPiecesToWin = 4;

        [Tooltip("Allow diagonally connected Pieces?")]
        public bool allowDiagonally = true;

        public float dropTime = 4f;
```

In my GameController, this holds the:
1. Cusomize Max_Depth of the AI
2. Turn count in the game
3. Board representation

4. Game mechanics(dropping)
5. Customize number of Rows and Columns of the board
6. Game Condition checker

PseudoCode:

```
public double miniMax(BoardState curr, int depth, double alpha, double beta, bool
maximizingPlayer)
    {
        double maxEval = 0;
        double minEval = 0;
        double eval = 0;

        //reached the root node
        if (depth == 0)
            return curr.evalScore;

        if(maximizingPlayer)
        {
            maxEval = -infVal;
            foreach(BoardState child in AdjacencyGraph.A[curr])
            {
                eval = miniMax(child, depth - 1, alpha, beta, false);
                maxEval = maxEval > eval ? maxEval : eval;
                alpha = alpha > eval ? alpha : eval;
                if (beta <= alpha)
                    break;
                eval = maxEval;
            }
        }
        else
        {
            maxEval = infVal;
            foreach (BoardState child in AdjacencyGraph.A[curr])
            {
                eval = miniMax(child, depth - 1, alpha, beta, true);
                minEval = minEval < eval ? minEval : eval;
                betaa = beta < eval ? betaa : eval;
                if (beta <= alpha)
                    break;
                eval = minEval;
            }
        }
        return eval;
    }

static int heuristicScore (int up, int down, int even)
       {
              int evalScore = 0;
              if (up == 4) { evalScore += 10000; }
              else if (up == 3 && even == 1) { evalScore += 6; }
              else if (up == 2 && even == 2) { evalScore += 4; }
              else if (down == 2 && even == 2) { evalScore -= 4; }
              else if (down == 3 && even == 1) { evalScore -= 2; }
              else if (down == 4) { evalScore -= 10000; }
              return evalScore;
       }
```

In my minimax with Alpha-Beta pruning function, I've followed this pseudocode structure where the time complexity of this is O(b^m/2) where b = number of connections of each board state and m = maximum depth of the search tree. It will result in this complexity since the condition of minimax with alpha-beta pruning is to hold a score that is the highest and lowest value temporarily and these will become a basis of whether the function will still need to dive deeper into the tree looking if the recently visited node evaluated score is higher or lower the alpha or beta respectively depending on the Max and Min depth level of the node. For an Ai that has a max-depth of maximum depth, then the ai would have an advantage on the player since it already knows the best path and moves to take and the lowest condition it can only have is the draw which also makes the AI looks intelligent.

In my heuristicScore function, I've set the values for the conditions: (1)if 4 pieces are connected, (2) 3 pieces are connected, and (3) 2 pieces are connected both conditions are set for the two sides wherein the ai pieces will be evaluated into a positive score while the human player will be evaluated into a negative score. I've set the values of the 4 pieces that are connected to a 10000 far from the score of the other condition so that it won't let a chance for the other condition to overtake this crucial condition. All of the derived scores will be accumulated in the evaluation score of the board state and this will be used in the search Tree.

Time Complexity of Mini-Max Algorithm with Alpha-Beta Pruning
AI Goes first = Max_Depth == 20
Human Goes first = Max_Depth == 19
k = 5

Max_Depth = 20 ~ 13.70secs	Max_Depth = 14 ~ 0.80secs	Max_Depth = 7 ~ 0.50secs
Max_Depth = 19 ~ 10.54secs	Max_Depth = 13 ~ 0.80secs	Max_Depth = 6 ~ 0.50secs
Max_Depth = 18 ~ 5.96secs	Max_Depth = 12 ~ 0.80secs	Max_Depth = 5 ~ 0.50secs
Max_Depth = 17 ~ 0.96secs	Max_Depth = 11 ~ 0.80secs	Max_Depth = 4 ~ 0.50secs
Max_Depth = 16 ~ 0.96secs	Max_Depth = 10 ~ 0.80secs	Max_Depth = 3 ~ 0.50secs
Max_Depth = 15 ~ 0.50secs	Max_Depth = 9 ~ 0.50secs	Max_Depth = 2 ~ 0.50secs
Max_Depth = 15 ~ 0.50secs	Max_Depth = 8 ~ 0.50secs	Max_Depth = 1 ~ 0.50secs
		Max_Depth = 0 ~ 0.50secs

In my code, the agent follows the algorithm of minimax with alpha-beta pruning. The agent will construct a list of all the possible moves up to the terminal state of the current board state, then it will find which states path will have better moves (can end up to a win or draw) despite the human is always choosing the best moves.

Having a full depth (20 and 19) gives the agent more data of the winning possible moves wherein it looks all the nodes up to the terminal states. The pruning helps with the search because it lessens the space and efficiently finds the best path for the AI. Pruning keeps the highest score as the alpha and the opposite as the beta having a basis for the agent on when to cut off a node.

Having the Max_Depth to be lessened by 'k'(N-k //k-number of depth to reduce for the max Depth) means that the AI might not see the terminal states yet leaving it with the gathered board states data where the search depth depends on the 'k'. Making the N equal to 1 means that the agent can only look at the next possible moves and it will not let him generate a winning path of the agent making the possible moves for the agent spontaneous like.

As a result, having the depth to Max(20 or 19 depending on who goes first) means that the AI has already the path to take whether the human can always choose the best move while lowering the depth proportionally decreases the chance of having a perfect path moves since it

can only see limited moves in its depth. Configuring my depth to the max is more effective than lowering it since it gives the agent an advantage already against the player making it become a perfect connect 4 player.

Technical Challenges Encountered:

Creating the minimax function gives me a hard time constructing it since it holds a lot of data needed for its conditions and debugging it is exhausting for me because I need to analyze each of the conditions that occur before the suspected line. Modifying the algorithm also gives me a lot of time to work because the algorithm that I've followed in the internet was only a plain(pseudocode).

EMERSON PAUL P. CELESTIAL