Assignment 7.1 Game Trees

The basic concept of a game tree is pretty easy to understand, and you've made similar things in the past. Game trees are not a data structure, which means their similarity to AVL, Red/Black, and B-Trees is in name only. These are closer in design to the eight queens, the knights tour, and our marble game in that they are recursive and the pattern that laying out all the recursive calls produces looks quite like a tree. For example, one recursive call my lead to 7 recursive calls at the next level. Each of those subsequently turns into 7 more recursive calls and so on and so on. Eventually, you reach some stopping point and start returning from recursion.

The big difference between game trees and these previous recursive games we've examine is that they are useful for examining multiplayer games in which you trying to generate an optimal solution with an adversarial element intermixed. Below we'll look at some simple examples before trying to dive into our assignment.

Seven Pennies in a pile game

This is a two-player game

To Start the game: Create a single pile of seven pennies

Playing the game: Players take turns taking a pile of pennies and breaking it up into two uneven piles. If your opponent can't break one of the stacks of pennies into uneven piles they lose, and you get to rub it in their face.

So, starting with seven pennies, you (**Player 1)** can break up the one pile to piles of:

3 and 4
2 and 5
6 and 1

Your opponent (Player 2) only has to be able to split a single pile. So, for example, if Player 1 chooses 2 and 5, Player 2 cannot split up the 2, but can split the 5.

So, let’s simulate a game:

If Player 1 was to pick 3 and 4, the possible divisions could be:

1, 2 and 4

3, 3 and 1

Player 2 might choose 3, 3 and 1

Player 1 can choose to break either 3 pile up. Either way, the results are:

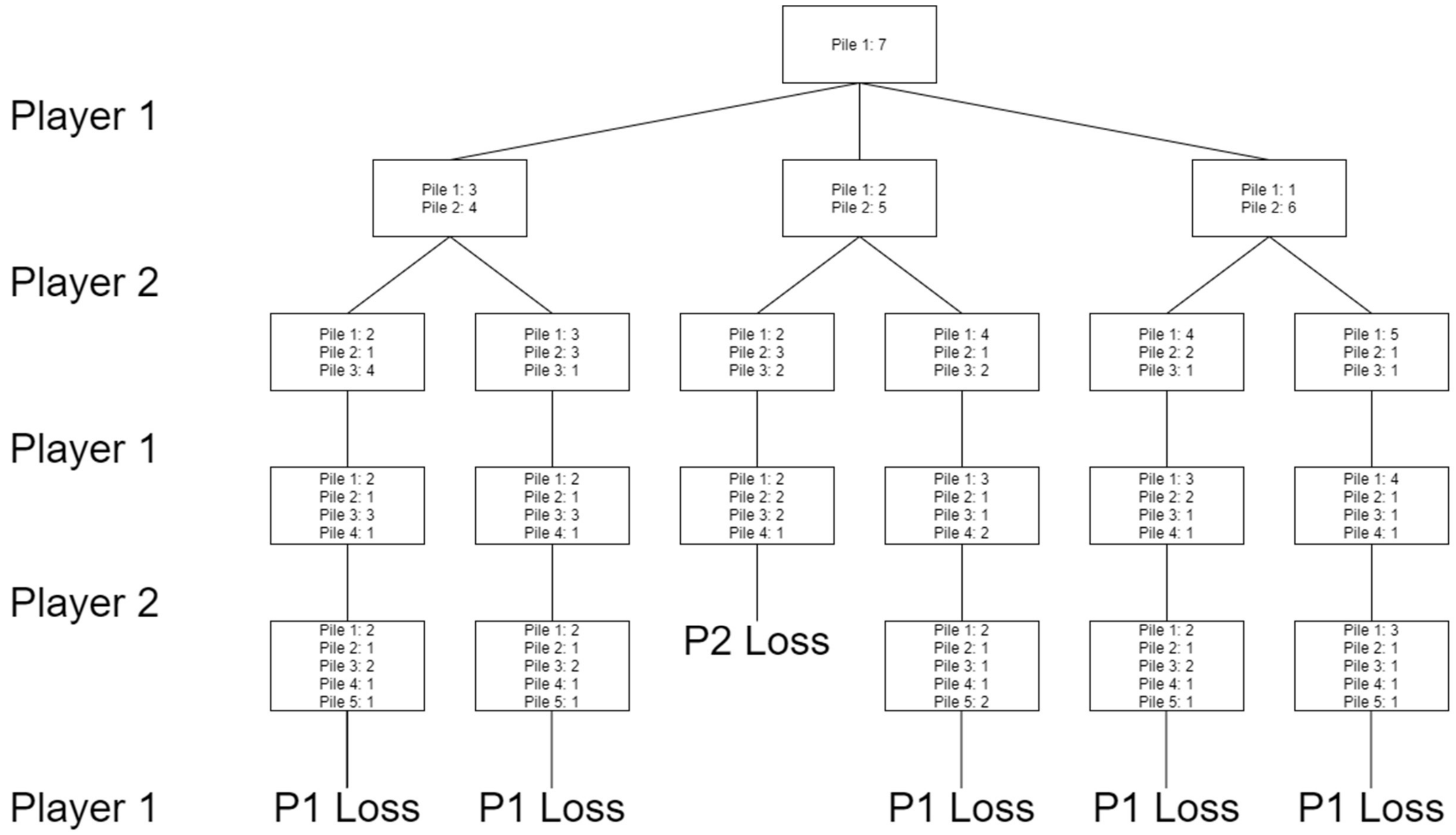
2, 1, 3 and 1

Player 2 then only has the option to break up the 3 to 2 and 1 leaving:

2, 1, 2, 1, and 1.

From here, Player 1 has no move and loses

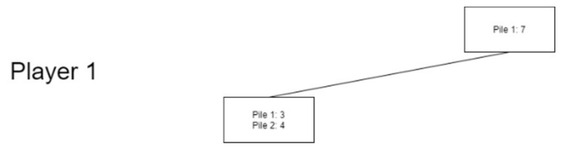
However, this is only **one** possible outcome. If we were able to map out every possible outcome, we would produce a tree that looks something like this:


Player 1
Player 2
Player 1
Player 2
Player 1
Pile 1: 7
Pile 1: 3 Pile 2: 4
Pile 1: 2 Pile 2: 5
Pile 1: 1 Pile 2: 6
Pile 1: 2 Pile 2: 1 Pile 3: 4
Pile 1: 3 Pile 2: 3 Pile 3: 1
Pile 1: 2 Pile 2: 3 Pile 3: 2
Pile 1: 4 Pile 2: 1 Pile 3: 2
Pile 1: 4 Pile 2: 2 Pile 3: 1
Pile 1: 5 Pile 2: 1 Pile 3: 1
Pile 1: 2 Pile 2: 1 Pile 3: 3 Pile 4: 1
Pile 1: 2 Pile 2: 1 Pile 3: 3 Pile 4: 1
Pile 1: 2 Pile 2: 2 Pile 3: 2 Pile 4: 1
Pile 1: 3 Pile 2: 1 Pile 3: 1 Pile 4: 2
Pile 1: 3 Pile 2: 2 Pile 3: 1 Pile 4: 1
Pile 1: 4 Pile 2: 1 Pile 3: 1 Pile 4: 1
Pile 1: 2 Pile 2: 1 Pile 3: 2 Pile 4: 1 Pile 5: 1
Pile 1: 2 Pile 2: 1 Pile 3: 2 Pile 4: 1 Pile 5: 1
P2 Loss
Pile 1: 2 Pile 2: 1 Pile 3: 1 Pile 4: 1 Pile 5: 2
Pile 1: 2 Pile 2: 1 Pile 3: 2 Pile 4: 1 Pile 5: 1
Pile 1: 3 Pile 2: 1 Pile 3: 1 Pile 4: 1 Pile 5: 1
P1 Loss
P1 Loss
P1 Loss
P1 Loss
P1 Loss

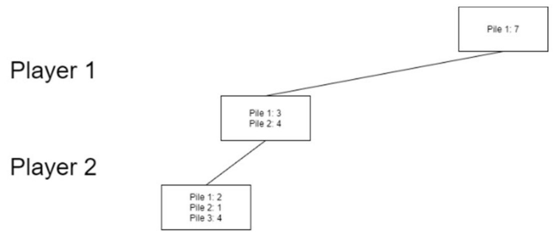
Looking at this it seems like you'd never want to be player 1 as almost all the possible paths end in a loss for you. However, you do get a choice of which move you will choose after exploring the tree. Obviously looking at this player 1 would want to choose to split the pile of 7 pennies into stacks of 2 and 5, but how do we know this programmatically? Like A*, game trees use a heuristic function to evaluate the quality of the position we're in when we reach the bottom most layer of our prediction process. This 'quality' score is based on our determination of our likelihood of winning. We use infinity (or int_max) to represent a guaranteed known for sure win and -infinity (or int_min) to represent a guaranteed loss. This value is returned by our recursive function to the previous level and once we know the heuristic value for all possible paths we chose the one that gives us the best chance at winning. Essentially, this means after all of the recursive calls are made we evaluate the heuristic values returned and then print out the move associated with the highest score.

However, the flip site of this is our competitor is trying to stop us from winning, so on player 2's turn rather than choosing the move associated with the highest score we assume that player 2 would make the move that would be the worst for us or the lowest heuristic value. In practice what this means is that even though we would want to split the seven pennies into a pile of two and a pile of five, player two would know that splitting the pile of 5 into a pile of 2 and 3 (giving us 2 3 2) would guarantee they lose and would not chose this option. We have to assume they would chose to split into 4 1 2 which would be a win for them and a loss for us. Thus the value returned at that level would be -infinity even though there is a possible win down this path. Here is an example of the recursive process moving through the left site of the tree:

1. Player 1 determines the possible moves that can be made and then makes the first move splitting the pile of 7 into 3 and 4.

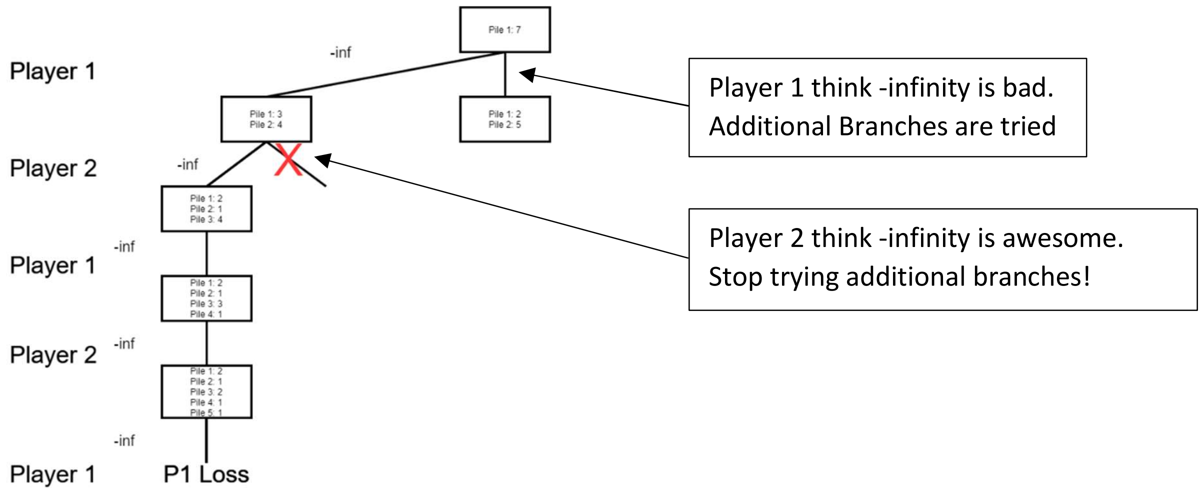


2. Player 2 determines the possible moves that can be made and then makes the first move splitting the pile of three into 1 and 2



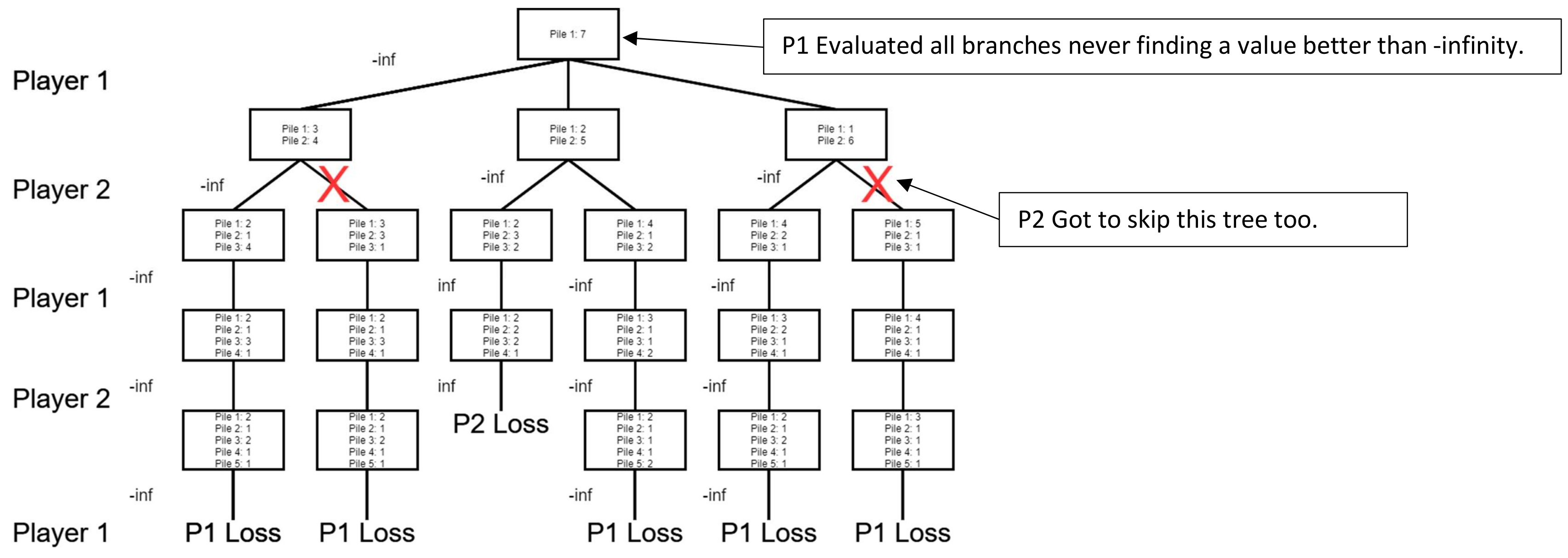
3. This process continues until you hit the bottom of the tree. Once at the bottom, Player 1 evaluates the state of each stack of coins and determines that none of them can be split, which is a guaranteed loss. Player 1 returns -infinity to the previous level. There are no other moves at Player 2's level (2nd from the bottom), so the best found is returned. The same for Player 1 (3rd from the bottom). The next level is a little more interesting as Player 2 does have additional moves that can be tried.

   However, this recursive process is quite time consuming and, as far as Player 2 is concerned, an optimal value of -infinity has already been identified. As a result, a little tree trimming occurs, and the second branch skipped and -infinity just returned.



   We're now back to the first call of our game and we've received a score of -infinity back. As this is Player 1's level and their goal is to maximize the heuristic value additional trees **are** tried. We move on to the second possible way to break up our stacks (2 & 5) and repeat the process.

4. Eventually the whole tree has been explored and it looks something like this:



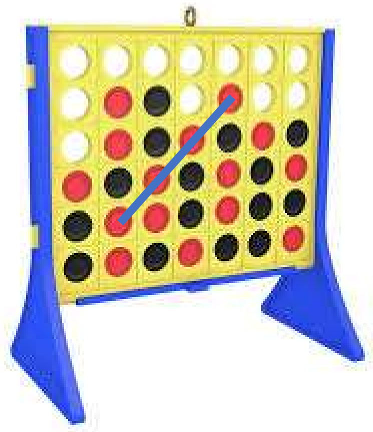
Player 1
Player 2
Player 1
Player 2
Player 1
Pile 1: 7
P1 Evaluated all branches never finding a value better than -infinity.
-inf
Pile 1: 3 Pile 2: 4
Pile 1: 2 Pile 2: 5
Pile 1: 1 Pile 2: 6
-inf
-inf
-inf
P2 Got to skip this tree too.
Pile 1: 2 Pile 2: 1 Pile 3: 4
Pile 1: 3 Pile 2: 3 Pile 3: 1
Pile 1: 2 Pile 2: 3 Pile 3: 2
Pile 1: 4 Pile 2: 1 Pile 3: 2
Pile 1: 4 Pile 2: 2 Pile 3: 1
Pile 1: 5 Pile 2: 1 Pile 3: 1
-inf
inf
-inf
-inf
Pile 1: 2 Pile 2: 1 Pile 3: 3 Pile 4: 1
Pile 1: 2 Pile 2: 1 Pile 3: 3 Pile 4: 1
Pile 1: 2 Pile 2: 2 Pile 3: 2 Pile 4: 1
Pile 1: 3 Pile 2: 1 Pile 3: 1 Pile 4: 2
Pile 1: 3 Pile 2: 2 Pile 3: 1 Pile 4: 1
Pile 1: 4 Pile 2: 1 Pile 3: 1 Pile 4: 1
-inf
inf
-inf
-inf
Pile 1: 2 Pile 2: 1 Pile 3: 2 Pile 4: 1 Pile 5: 1
Pile 1: 2 Pile 2: 1 Pile 3: 2 Pile 4: 1 Pile 5: 1
P2 Loss
Pile 1: 2 Pile 2: 1 Pile 3: 1 Pile 4: 1 Pile 5: 2
Pile 1: 2 Pile 2: 1 Pile 3: 2 Pile 4: 1 Pile 5: 1
Pile 1: 3 Pile 2: 1 Pile 3: 1 Pile 4: 1 Pile 5: 1
-inf
-inf
-inf
P1 Loss
P1 Loss
P1 Loss
P1 Loss
P1 Loss

You can see a couple of important things have happened here. First, Player 2 again got to skip evaluating an entire branch of the tree because the first path they went down returned an optimal value from their perspective. Second, at the topmost level Player 1 had to evaluate every branch because none of the branches returned his optimal value (infinity). The score he would return back to Player 1 would be -infinity as no matter which decision he makes initially will lead to a win.

Applying this idea to larger problems

One nice thing about the penny game was that we were able to scan the whole tree. If there was a win, we reported back that we found a win. If not, we reported back a loss. What we're going to find as we work on this is that in most games, it is impossible to do a complete analysis in a reasonable amount of time.

Consider the game Connect Four. If you've never played the game board looks like this:



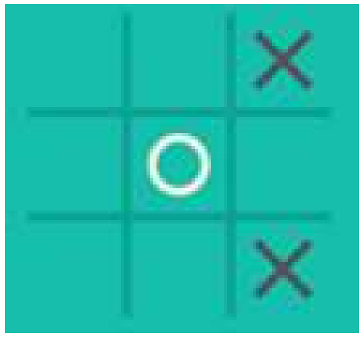
The goal of the game is to be the first player to connect four checkers of your color vertically, horizontally, or diagonally. In the above image red has won diagonally in the position indicated by the blue line.

If we were creating a game tree for connect four each position would max out at 7 branches from each level as there are 7 columns you can place a piece in. Obviously, this would get smaller as columns are filled up. Portions of the tree would be very short because they lead to quick wins, but there are also play variations where the whole board may fill up. The tree is 42 levels deep at these points If we try and search the whole tree for solutions our program may take 30 minutes or more to make a move each turn!

To combat this, we add a sort of governor that controls how deep down the tree into the future we are looking. For example, with a little experimentation we may discover that it takes about 30 second to look 9 levels deep in our tree. This is a reasonable amount of time to decide each time we take our turn. However, if we are never looking more than 9 moves into the future we will have A LOT of board states that have not yet reached a conclusion. This is where our heuristic function really helps. Determining if you've won a game is quite easy, but what we need to do now is try and create a function that will tell us if we are **winning** a game. Also, if we are winning a game, by **how much** are we winning?

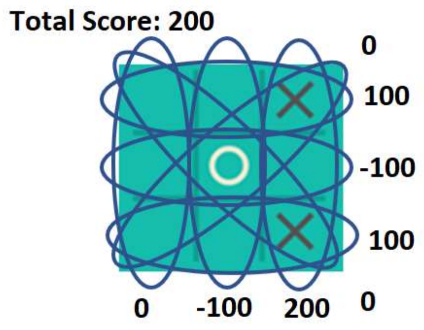
For example, obviously finding a board that has 3 in a row is quite valuable, but what about a board that has two in a row for us? What if it also has two in a row for the opponent? Is a board with two 'two in a rows' more valuable than one with one two in a row? How much more valuable? What if we have a two in a row, but it has an opening at both ends? What if it's just one opening, but it will take 3 or 4 or 5 more checkers to get to the opening? The job of the heuristic function needs to evolve from just telling us if we've won or lost to consider the importance of each of these scenarios and assign an overall score to the board state. In many ways, the quality of this function will dictate the quality of your overall algorithm.

Let's switch back to an easier game for a minute and consider tic-tac-toe. The tree for tic-tac-toe is large enough that it would be hard to fit it all in a drawing but is still quite easily solved by a modern computer. The tree would look different than connect for as in connect four you still have 7 possible positions choose after the first player goes. In fact, you have 7 choices all the way until a column is full. In tic-tac-toe you start out with 9 choices, then at the next level you have 8, then 7, 6 …. 0.

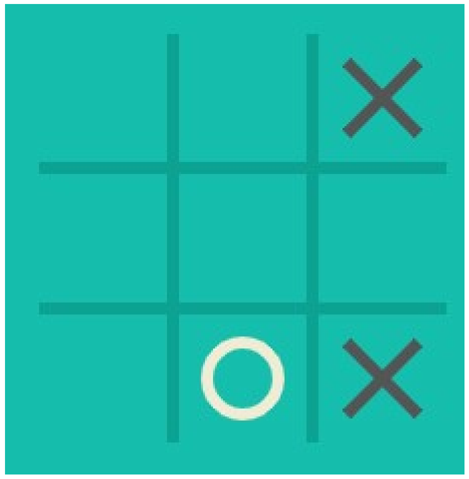


Let's assume we're writing the code for tic-tac-toe to evaluate the board state three moves into the future. So, at the start of the game one possible board state that could be generated would look like the board above. As the game progresses, you'll be evaluating the board with more and more pieces on the board as you are always looking three moves past whatever the current board state is. For right now though, assume we're playing as X and this is where we're at. How would you evaluate the quality of this board state?

One way to do this would be to examine every row, column and diagonal on the board and each time you encounter a symbol on the board you add or subtract 100 points accordingly. So, for the above board you have the following situation:



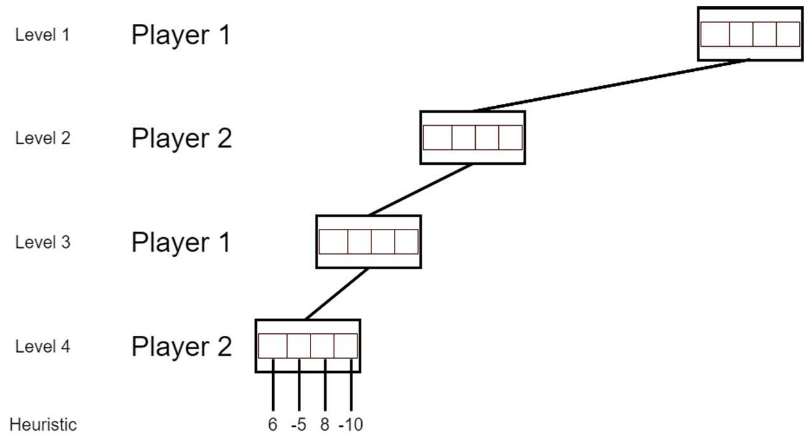
This board isn't bad, but also isn't great. Yes we have a positive score which means it's better for us, but we've also take one additional turn. The score for this one should be 400. See if you can figure it out:



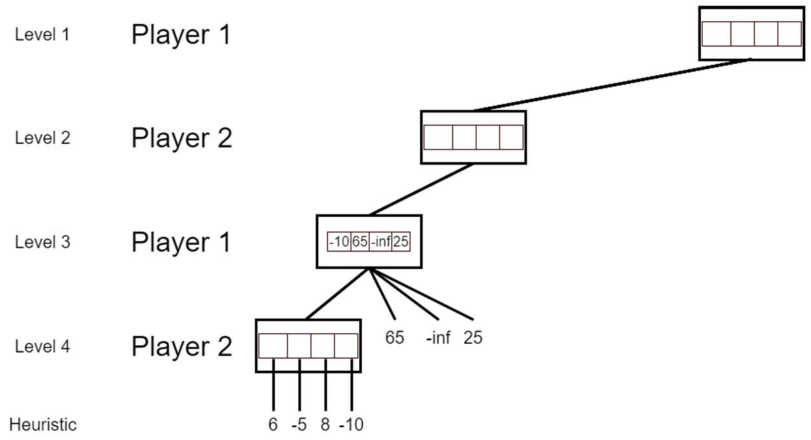
You can see from this example you don't always need the most sophisticated heuristic. Often something that objectively runs 'quickly' while also giving a good estimate is preferable to something that has a ton of calculations to get the perfect answer but is quite slow. Remember this function likely will get called up thousands upon thousands of times for larger problem spaces. One simple optimization you could add to this would be if you ever score two rows at 200, then you're guaranteed a victory and can return infinity. The opposite is also true for scoring two rows -200.

We'll come back to tic-tac-toe in a minute and try and program it, but what is something that's kind of unique about the trees we will be generating with checkers compared to the tic-tac-toe tree? One thing is that as we get more space on the board and more kings, we often have a very large number of possible moves. This may mean in some cases; you have a tree **adding branching levels** as you go down.

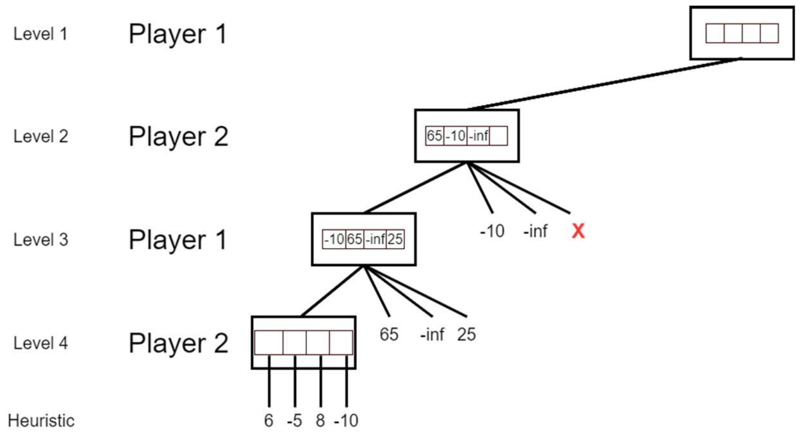
Before we start programming tic-tac-toe, lets walk through another simple example of how heuristic values are determined now that we have a height limit and the heuristic function concept flushed out. For this one, imagine we're programming some imaginary game where you have four different moves that can be made at each game state. In the picture below, we've made the first recursive call at each level until we've reached a depth of four. At this depth, the heuristic was called rather than performing additional recursive calls. You can see that for the four board states available at level 4 heuristics were calculated:



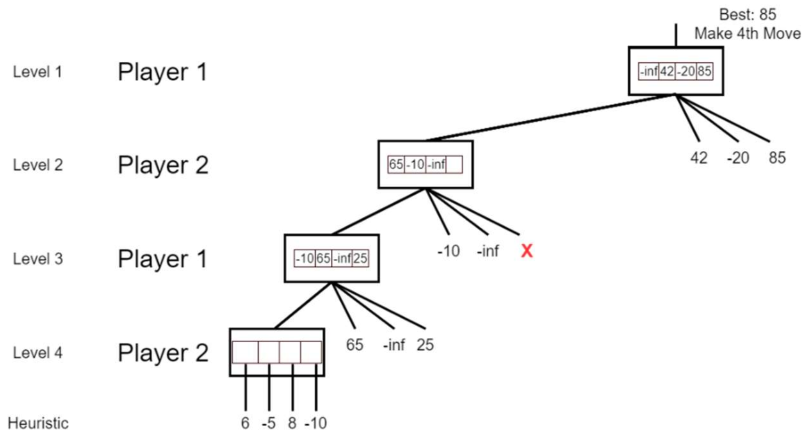
Since Level 4 is a Player 2 level, our program will want to minimize the heuristic value at this point and return -10 as the value to the previous level. Assume the remaining heuristic calls at level three were made down to their level, player 2 picked the lowest value available at each level and returned them. When we need to make our decision on what to return at Level 3, our program state might look something like this:



At Level 3, Player 1 will want to maximize the score returned. Notice that even though a -inf was returned all branches had to be explored as we can only skip branches if and ideal value for the Player tied to the level is returned (infinity in this case). The max value in this case is 65 and returned as the value of the tree:



At Level 2, Player 2 is again minimizing. The third recursive call found -infinity was the best option, so the fourth call is unnecessary and -infinity is returned. However, Player 1 will only choose that option if there is nothing better, so it will likely get ignored at Level 1:



You can in the previous graphic that the -infinity was indeed ignored and the algorithm chose to make the 4th move that had a score of 85.

Programming Tic-Tac-Toe

I will post the finished in-class code for tic-tac-toe from the last session of this course in Blackboard, but here's a quick overview of what's needed to program it.

The board

The recursive process will need some method to keep track of the game board. For tic-tac-toe, this can simply be a 2D character array. Much like knights, this can just be done at the class level as we can put on and take off the last move pretty easilty:

```
private:
	char board[3][3];
```

This board will need to be initialized to all spaces when the game starts. When a call is made to determine which move the computer should make, the current state of this board will be used to determine which moves are available and then one at a time each move will be added and a recursive call will be made. To determine the set of moves that can be made, will just scan the board and anywhere there is a space will be a valid potential move. However, we somehow need to track what each of these moves is without making any of them. After the recursive process, we'll need to track the heuristic value associated with each move. To support this a struct is created to track the row and column where each potential move can be placed and an int to store the value:

```
struct TMove
{
	int row;
	int column;
	int value;
};
```

The loop then to fill in the array of potential moves then would look like this:

```
void FindMoves(TMove allmoves[], int  & nummoves)
{
        nummoves = 0;
        for (int r = 0; r<3; r++)
        {
                for (int c = 0; c<3; c++)
                {
                        if (board[r][c] == ' ')
                        {
                                allmoves[nummoves].row = r;
                                allmoves[nummoves++].column = c;
                        }
                }
        }

}
```

The recursive process then would need to create and fill the array of potential moves and then loop through each move add them to the board and make a recursive call. However, the first thing that happens is if we are at the bottommost level of recursion, we need to calculate the heuristic and return it:

```
        if (ply == 0) {
                return Heuristic();
        }
```

Ply here is passed into the recursive function as a parameter and controls the height of our tree. Each recursive call subtracts 1 from it until we hit zero.

After this we find our moves and loop through them:

```
TMove arrayofMoves[9];
int numberofMoves;
FindMoves(arrayofMoves, numberofMoves);

for (int curMove = 0; curMove < numberofMoves; curMove++) {
        board[arrayofMoves[curMove].row][arrayofMoves[curMove].column] = player;

        if (win()) {
                //...
        }
        else {
                if (player=='X')
                        arrayofMoves[curMove].value = ComputerMove(ply-1,'O', false);
                else
                        arrayofMoves[curMove].value = ComputerMove(ply-1,'X', false);

                if (!TopLevel) {
                        board[arrayofMoves[curMove].row][arrayofMoves[curMove].column] = ' ';


                        //leave early on a win?!!?
                        if (arrayofMoves[curMove].value == INT_MAX && player == 'X')
                                return INT_MAX;
                        else if (arrayofMoves[curMove].value == INT_MIN && player == 'O')
                                return INT_MIN;
                }
                else {
                        //leave early on a win?!!?
                        if (arrayofMoves[curMove].value == INT_MAX && player == 'X')
                                return INT_MAX;
                        else if (arrayofMoves[curMove].value == INT_MIN && player == 'O')
                                return INT_MIN;

                        board[arrayofMoves[curMove].row][arrayofMoves[curMove].column] = ' ';
}
        }
}
```

There's a bit going on here. First you can see the call to FindMoves that fills in arrayofMoves. Next a loop that goes through each move and marks it with an X or O (This is player, which is passed in as a parameter). Once the piece is on the board we have two possibilities either the player has won, or the game isn't over (if win() …). If a win as occurred we take the piece off and return INT_MAX or min as appropriate. Otherwise, subtract 1 from ply, switch which players turn it is and make another recursive call.

Once the recursive call has returned, if either INT_MAX or INT_MIN (depending on who's turn it is) was given as a value we break out of the loop and return without any more recursion. Otherwise the piece is turned back into a space and we try the next position.

```
int bestIndex=0;
for (int curMove = 1; curMove < numberofMoves; curMove++) {
       if (player == 'X' && arrayofMoves[curMove].value > arrayofMoves[bestIndex].value)
              bestIndex = curMove;
       else if (player == 'O' && arrayofMoves[curMove].value < arrayofMoves[bestIndex].value)
              bestIndex = curMove;
}

return arrayofMoves[bestIndex].value;
```

This small piece of code at the bottom is responsible for finding the best value and returning it in the case MAX or MIN wasn't found earlier.

The Heuristic function works as described earlier but is modified slightly using a system we developed in class to award 10,000 points if one of the scoring directions has two spaces marked for one player and none for the other player. 100 points is still awarded for a diagonal having a single space marked with none for the other player. None is awarded for a diagonal that is blocked even if you outnumber the other player.

The loop to play the game then looks like:

```
       do {
              game.ComputerMove(1, 'X', true);
              game.show();
              if (game.over())
                     break;
              game.ManualMove('O');
              game.show();

       } while (!game.over());
```

The third parameter on the call to ComputerMove (true here) is used so that when at the topmost level, the best move is left on the board.

Look through the code and feel free to ask questions about it during our daily video calls.

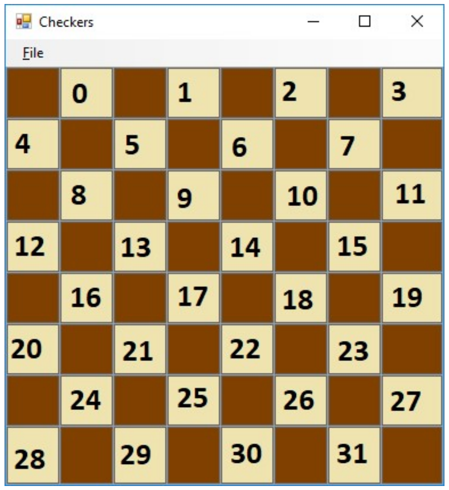
Our Checkers Game

For our checkers game you have the option of developing the entire game in C++, or if you prefer, I've created a Visual C# board that can be used. The prebuilt C# app has several useful features built in. Most importantly, it'll already perform all valid human movements for you. Things like stopping the human from being able to make a move when they have a jump to perform or stopping them from switching the active piece when in the middle of a jump take a fair amount of time to program and that is all done for you.

If you're unfamiliar with C# I would suggest you still try your best to work with this project rather than building your own. To get started, go to the solution explorer right click on Checkers.cs and choose 'View Code'. Scrolling all the way to the bottom you'll see a function called ComputerMove() that becomes the only major piece of code you need to implement. Some important things to know about this version of Checkers:

Because you're just writing primarily this one function, you really won't have to interact with the interface of the program too extensively. With this excluded, the base C# language looks very similar to C++ and you'll pick it up quickly. Some examples of bitwise manipulation are given below to help you get started.

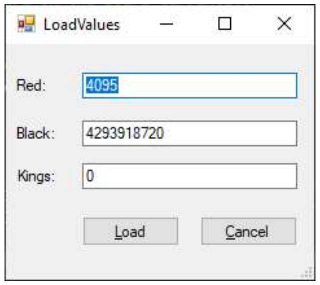
To enhance the performance of the game, the state of the board is stored entirely in 3 unsigned ints called Red Black and Kings. Red and Black each respectively denote which positions on the board are occupied by checkers of that color. The Kings variable stores which checkers of both colors have been upgraded to kings. There are exactly 32 usable squares on a checkerboard, so this works out perfectly! Here is a map of how the C# implementation of the game associates each bit with a square on the board:



Normally you'd play checkers on the dark squares, but I felt the checkers stood out better on the light squares. If you design your own version of the board, the only important factor is that the non-playable corner is in the upper left / bottom right. Rotating the board 90 degrees clockwise and playing on the opposite color would produce the exact same board.

I've built a few helpful features into the game already. First, if you choose File->New the board will be setup with a complete set of checkers. The human plays as black from the bottom and goes first by default. However, if you'd like the computer to take a turn, at anytime you can choose File-> Force Computer Turn.

File->Load can be used to retrieve the values for Red Black and King at any time. Choosing it brings up this dialog:



Red:	4095
Black:	4293918720
Kings:	0

The values already in the boxes indicate the state of the current game. You can also tweak these numbers and choose load to change the board to any state you'd like. This is handy for debugging if you're running your computer code and it is consistently making poor choices at some specific spot. Record the numbers, fix your code and load them back in again.

Lastly, choosing File->Setup allows you to click on the individual squares on the board and change their color. Clicking on a square will cycle it from empty to red to red king to black to black king and then back to empty again. You can do this for as many squares as you choose, but when you're done you must choose File->Setup again to switch the game back to play mode.

Help in Developing Checkers – Bit Manipulation

As you can imagine, Bitwise manipulation plays a major role in the development of this game. However, bit manipulation in C# looks very similar to what most of you learned in 242. For example, if I have one unsigned int with the number 1 in it, the bitwise storage of this would look like:

```
uint var1 = 0000 0000 0000 0000    0000 0000 0000 0001
```

Another unsigned int with the number 16 in it would look like this in bitwise form:

```
uint var2 = 0000 0000 0000 0000    0000 0000 0001 0000
```

Using a bitwise or on these to produce a new variable would look like this:

```
uint var3 = var1 | var2;
```

which would set any bit that existed in either var1 or var2:

```
        0000 0000 0000 0000    0000 0000 0000 0001
[bw or] 0000 0000 0000 0000    0000 0000 0001 0000
---------------------------------------------------
        0000 0000 0000 0000    0000 0000 0001 0001
```

A bitwise and would then set any bit that existed both var2 or var3 if this operation was done:

uint var4 = var2 & var3;

```
         0000 0000 0000 0000    0000 0000 0001 0000
[bw and] 0000 0000 0000 0000    0000 0000 0001 0001
-----------------------------------------------------------------
         0000 0000 0000 0000    0000 0000 0001 0000
```

There is also an XOR operator ^ and a negation operator ~ and shift operators << and >> if needed. If you've never used a bit shift operator before they are very useful when you know you want to flip the state of a specific bit in a number. For example, if I wanted to add a black checker at position 14 to my existing set of black checkers, the command would look something like:

Black = Black | (uint)(1 << 14);

This takes the number 1 which is:

0000 0000 0000 0000   0000 0000 0000 0001

And shifts it over 14 spaces

0000 0000 0000 0000   0100 0000 0000 0000

And then ORs this intermediate number with the existing set of black checkers, which we'll say looked like this

```
        0000 0000 0000 0000    0001 0000 1100 1111
[bw or] 0000 0000 0000 0000    0100 0000 0000 0000
-----------------------------------------------------------------
        0000 0000 0000 0000    0101 0000 1100 1111
```

When you've completely finished determining what move you want to make, the class level Red Black and King variables should be set to their new values and drawBoard should be called to update the visual interface. This example is kind of nonsensical, but let's say for example, I just wanted to write ComputerMove() to set to clear the entire set of black checkers and then place one black checker at position 13 in the above picture. To do this, the entire function of ComputerMove() would look like:

```
        private void ComputerMove()
        {
            MessageBox.Show("Computer's Turn");
            Black = 8192;
            drawBoard(Red,Black,Kings);
        }
```

If the 8192 doesn't make sense to you remember that each bit of Black represents one position on the board. The 13th board position is bit 13 of Black, or the number 2^{13} which is 8192. If you wanted a checker at position 13 and position 0, you'd change the number to 8193 or $2^{13}+2^{0}$. It makes little sense to use these raw numbers in your code though, so you'd likely write this as something like:

```
        Black = (1 << 13) | (1 << 0);
```

Other useful things you might want to do with bit manipulation:

Get all the empty squares: uint empty = ~(red | black);

Check if red has a specific position set: if ((red & (1U << position)) != 0)

Get all red kings: uint rkings = red & kings;

Move a king to a new position: kings = (kings | (1U << newPosition)) ^ (1U << oldPosition);

Help in Developing Checkers – The Game Structure

Much of the structure of the checkers game can be used here. Rather than a 2D grid, we have the three unsigned ints to work with. In order to help keep moves grouped together I created a struct that would keep an entire board state together as a single unit. In c# this looks a little different. I right-clicked on the Checkers project and chose Add->New Item->Class and named it CheckerMove. This class then looked something like this:

```
    class CheckerMove
    {
        private uint _red;

        …
        public uint Red
        {
            get
            {
                return _red;
            }
            set
            {
                _red = value;
            }
        }
        …
}
```

You'll notice that in C# each item has its own declaration of visibility rather than something like:

Public:

above a bunch of functions.

Red here is called a property and is the same as creating functions to set and retrieve the value of _red. It allows us to use the function like we would a variable. For example, the get function could be used doing this:

uint color = Whateverthevarnameis.Red;

and the set one could be used doing this:

Whateverthevarnameis.Red = color;

Note because these are actually functions, you still have the ability to add protection in if desired (for example limiting the range the user is allowed to set).

Inside your class you'll want to have a variable and a property for each of red, black, kings, as well as the heuristic value.

Recording the moves:

Your main ComputerMove function will need to determine a list of valid moves before making recursive calls. If you're using the Computer Move class this will look something like this:

```
            InitialMoves = new CheckerMove[48];
            for (int i = 0; i < 48; i++)
                InitialMoves[i] = new CheckerMove();
```

I then wrote a function to identify all moves that could be made by either color:

```
       GetMoves(InitialMoves, ref moveCount, CheckerType.Red, black, red, kings);
```

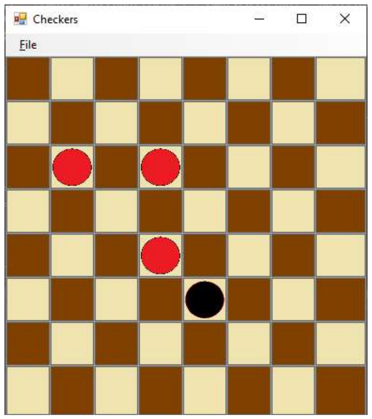
InitialMoves here is the array that will be filled in with moves. moveCount is the number of moves found by the function. Notice that C# uses ref here to indicate the variable is being passed by reference. CheckerType.Red indicates which color we are trying to find moves for and black red and kings are the current state of the board passed into the function when started. The function prototype then would look like this:

```
private void GetMoves(CheckerMove[] move, ref int moveCount, CheckerType color, uint black, uint red, uint kings)
```

CheckerType is an enumeration mostly used as part of the visual code, but it works here for informing which site we're building a move list for:

```
enum CheckerType { Empty,Red,RedKing,Black,BlackKing};
```

GetMoves is a fairly complicated and lengthy function and will take you some time to develop. Keep in mind you have to figure out all regular moves, king moves, jumps, double jumps, triple jumps, etc.. Further, you may have a jump that has multiple jumps off of it. For example, suppose you are black here:



Obviously, you have two double jumps that need to be added as possible moves here, but as important is that the single jump NOT get added as a possible move. This may require a recursive function to properly identify all possible jumps.

Heuristic:

In checkers you win if you eliminate all your opponent's checkers, so obviously having more checkers than your opponent is the main thing you should be scoring on. Other things you may choose to use in your scoring include having more kings than your opponent and keeping pieces in your back row (preventing your opponent from getting kings). Feel free to look up checkers strategy and try and include other things as well. As mentioned earlier, the better your heuristic the better the quality of your predictions will be.

End Game:

Checkers, Chess, and other games built with game trees can be difficult to beat. In fact, if you can routinely beat your game when this is finished, you probably have a bug in your code. However, a skilled opponent will often be able to beat your game as the end of a game nears, game trees have a difficult time finding paths to victory. This is because the number of possible moves that lead to victory may be a high number of moves away and difficult to judge as superior in any way. To combat this, game trees are often accompanied with an end game database. End game databases are large files that store optimal moves for any given position with a set number of pieces (for example 3 red v. 3 black or fewer). They can be quite difficult to work with, but some are available for download around the Internet. Working with an End Game database is **not** part of the assignment, but it's important to understand the advantage these can afford you.

Help in Developing Checkers – What else?!?

There are two major topics still to cover that will be addressed in another document. To help improve the performance of your game we will look at how to introduce threading to this algorithm. Secondly, we'll explore the topic of alpha-beta pruning. AB pruning can be thought of an augmentation of our idea of eliminating branches if a max/min score is found. It drastically improves the performance of your game tree, but does add some difficulty to the overall assignment.