Implementing Othello in ML
(using Alpha-Beta Pruning)

OR

Teach Yourself

Board Game Programming

in **14** Minutes!

Step 1- We Need to Somehow Represent a Board in ML

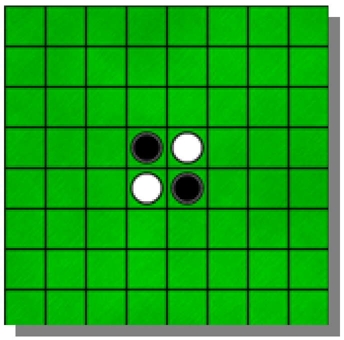
Cannot do much of pattern matching with this structure.

It is kind of thinking in C++! :)

Solution 1 - Use a list of strings of the length 64, with items “B” for black , “W” for white, and “E” for empty .

Example

[“E”, “E”,“E”, “E”,“E”, “E”,“E”, “E”,“E”, “E”,“E”, “E”,“E”, “E”,“E”, “E”, “E”, “E”,“E”, “E”,“E”, “E”,“E”, “E”, “E”,“E”, “E”, “B”, “W”, “E”,...]



What occupies each square can be Black, White or Empty

A square is made of an X and Y coordinates and what sits in it

A board, then, is simply a list of squares

Solution 2 - Define new data types to present each element of a board:

```
datatype occupyingType = Black
                       | White
                       | Empty;
```

```
datatype square = Square of int*int*occupyingType;
```

```
datatype board = Board of square list;
```

Now we can do All sorts of Pattern Matching!

Example:

```
fun oppositeType Black = White
 |  oppositeType White = Black
 |  oppositeType Empty = Empty;
```

Step 2– Now, We Need to Find All Valid Moves in a Board

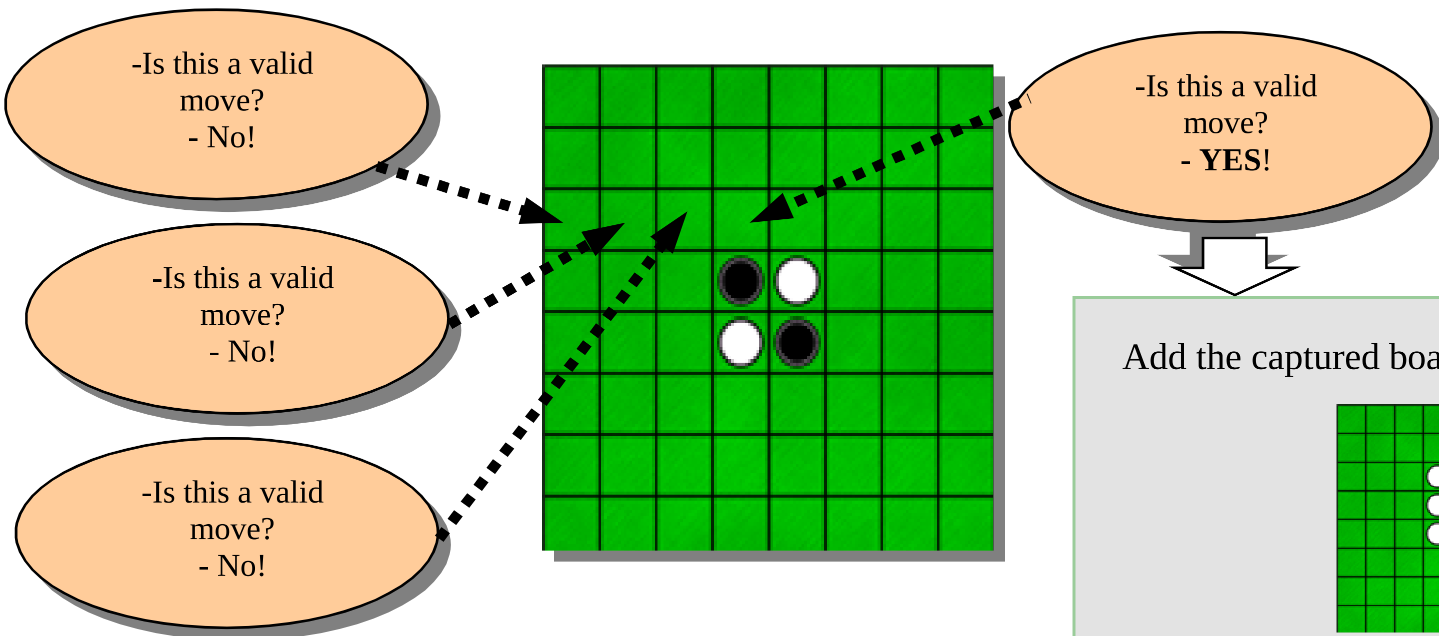
Brute Force Solution:

1. Get a board and a player
2. Go through all 64 squares in the board, and see when the player puts a piece in that square it will be a valid move
3. If it is valid, capture the enemy pieces in between, and add the board to the list of possible moves

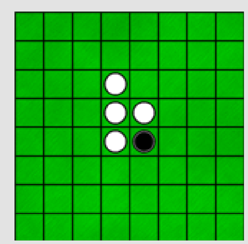
Example:

Bellow is a few steps of the algorithm, when we are looking for all possible moves for the White player:

This may not be the most efficient approach to solve this problem, but hey, it works! :)



Add the captured board to the Valid Moves list



Step 3– We need to represent all possible moves

Solution– Create a search tree of all possible moves from the current board

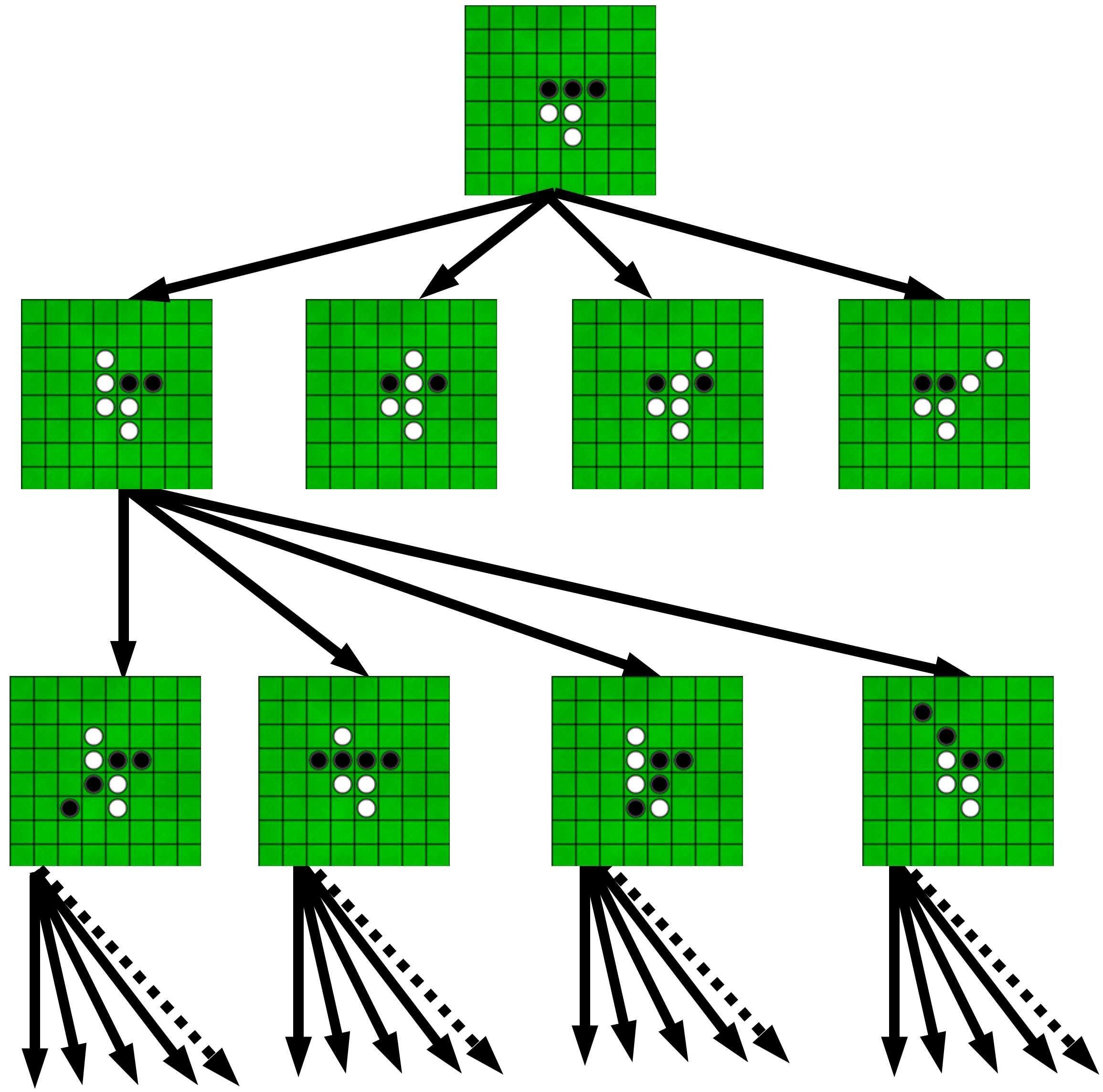
Example: see the tree bellow

On this board, white has 4 possible moves

We assign each of the possible moves as one of its children

Now in the case of the leftmost child, black also has 4 possible moves

Using this search tree, we can represent every possible move in a game



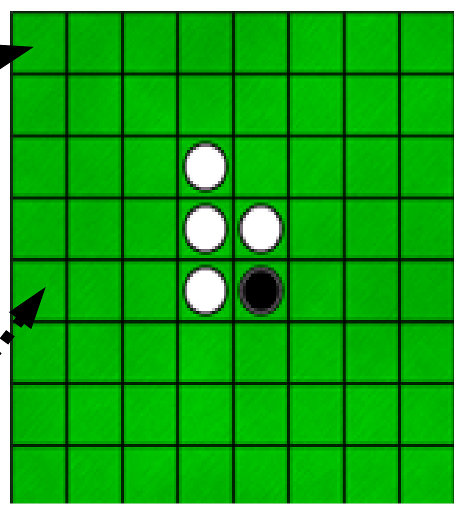
Step 4– What Is a Good Move?

- If we want to find a good move, we need to compare the boards that are the result of different moves
- ML cannot distinguish a board that has a good chance to win from a board that is about to lose
- But ML understands numbers!

We need a heuristic function from Boards to Numbers

Corners cannot be re-captured — They are worth **4** regular squares

Edges are hard to re-capture — They are worth **2** regular squares



Value of this board = 4 - 1 = 3

A Simple Heuristic Board Evaluator:

- Count all player pieces on the board
- Count all enemy pieces on the board
- Take into account that corner and edge squares are more worthy than other squares
- The value of the board is simply:

# of Player Pieces - # of Enemy Pieces

- The higher this number, the more likely it is that the player will win the game in this board.

Step 5- So How Do We Find this Good Move?

This Is Called: **MIN/MAX**

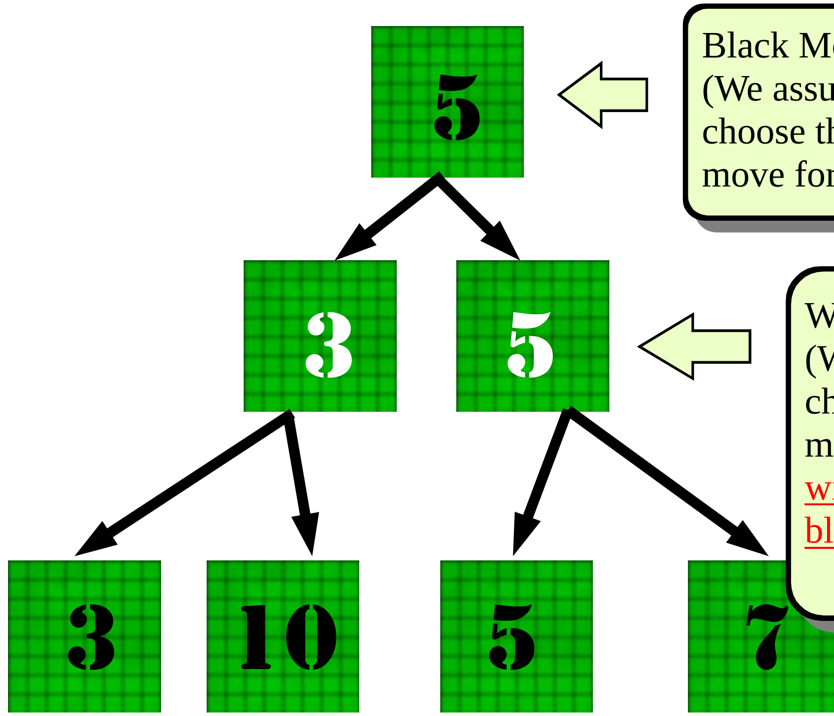
- Remember that we already have a search tree that represents all possible moves from current board till the end of the game.
- We also have a function that can evaluate a board and tell us how “good” the board is.
- We only need one assumption, and then we are good to go:

ASSUMPTION: Each player on its turn will choose the Best possible move

Here Is the Idea:
Traverse the tree, depth first, until you reach the leaves . Evaluate each leaf board (these are the boards that present a finished game).

Now on each level, if it is your turn to play, you want to get the value of the best child. If it is your enemies turn, she wants to do you the most harm, so she will choose the child with smallest value.

Recursively repeat this process, until all nodes of depth 1 have a value. Now, the best possible move is simply the node with highest value.



Black Moves Here (We assume it will choose the best possible move for Black)

White Moves Here (We assume it will choose the best possible move for White, which will be the worst possible move for Black)

Step 6– Making the Ultimate Othello Game -that Always Wins-

We have a search tree that contains all possible moves in a game

+

We have a process that can find the best move in a list of valid moves

→

We are able to make an Othello game that always wins!

NOT TRUE!

Branching Factor is the problem.
With an average branching factor of 10, to traverse all game nodes until the leaves, we have to evaluate this many boards:
$\mathbf{10^{64}}$
(=100,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000)

This is not easy!

Solution: Instead of traversing the whole tree –which is close to impossible, we choose a depth (for example 5), and only search the tree within that depth.

Step 7– Increasing the Depth of our Search

- As we saw earlier, we have to limit our game tree to a certain depth.
- But when the depth is too small, Min-Max plays stupidly
- Somehow, we need to go deeper in the tree, and yet don't traverse as many nodes as the branching factor forces us to.

A Lesson from the Gardener:
"Some branches of a fruit tree will never give any fruit. If you cut these branches early in the year, the rest of the branches will become stronger and give more fruit."

If we could somehow determine the branches that have no chance of being chosen as the best move and are only adding to the branching factor, we could prune them by simply ignoring them and their children.

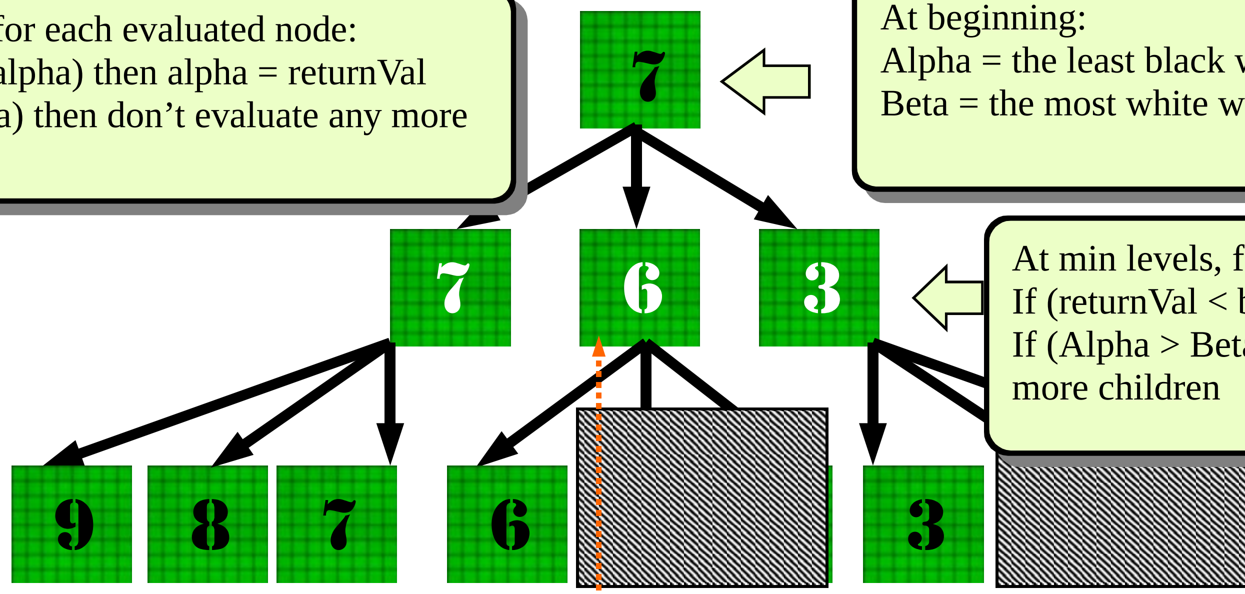
Step 8- Pruning the Fruitless Branches

This Is Called:
Alpha-Beta Pruning

- We need a way to predict that a branch is no good, so we can prune it without traversing it
- One method to do this is called Alpha-Beta pruning.
- For each node we define two new values:
  - Alpha = Floor = A value that, after this point, the player will at least gain
  - Beta = Ceiling = A value that, after this point, the opponent will at most give

At max levels, for each evaluated node:
If (returnVal > alpha) then alpha = returnVal
If (Alpha > Beta) then don't evaluate any more children

At beginning:
Alpha = the least black will gain = -Infinity
Beta = the most white will give = +Infinity



At min levels, for each evaluated node:
If (returnVal < beta) then beta = returnVal
If (Alpha > Beta) then don't evaluate any more children

All this makes no sense?! -- Try an example:
If we blindly follow the algorithm, at the indicated board, after the Min process evaluated the left child to 6, it will have: Alpha = 7, Beta = 6. Alpha is greater than Beta so it won't need to evaluate the rest of the children.
- WHY?!! Because at this point it knows that "after this point, the opponent will at most give" 6, and it also knows that "after this point, the player will at least gain" 7, so no matter what the rest of the children evaluate to, the player (Max process) will not choose what we return from this branch. Therefore, we can prune it

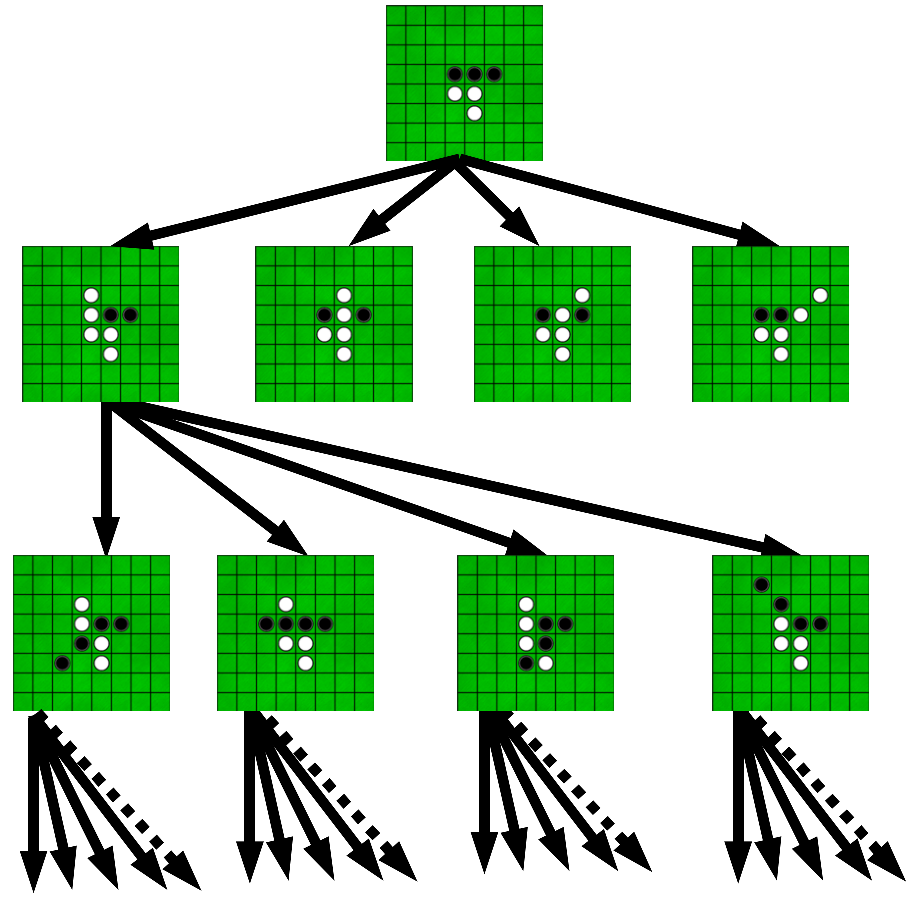
Step 9– Major Disappointment! Our Pruned Tree is Not Much Faster

Remember our game tree was made of a board on its root and the list of all possible moves (presented by boards) as its children.

Remember that

ML is an Eager Language

ML creates the whole tree at the moment that we define it, and before we get to prune anything



For our pruning method to be effective, we need to force ML to wait and create the nodes of the game-tree as we need them.

Step 10– A Good Othello Player Is a Lazy Othello Player!

To make our game-tree lazy, the main thing that we have to do is to replace the list of children with a sequence of children.

Example
Here is the structure of a general infinite tree:

```
datatype 'a itree = NULL
                  | iTree of 'a * (unit -> 'a itree seq);
```

Here is a simplified lazy game-tree generator:

```
fun genGameTree (board) =
   [....]
   if (isEmpty possibleMovesSeq) then
      NULL
   else
      (iTree(board, (fn () => (iMap genGameTree possibleMovesSeq))));
```

So Why is this any Better?

After we made our game structure lazy, the nodes of the game tree will only be created when they are needed. This means that when we ignore a branch at a top level (for instance by using alpha-beta) we save all the computation time and storage necessary to create that board and its children, and their children, all the way to about 60 levels!

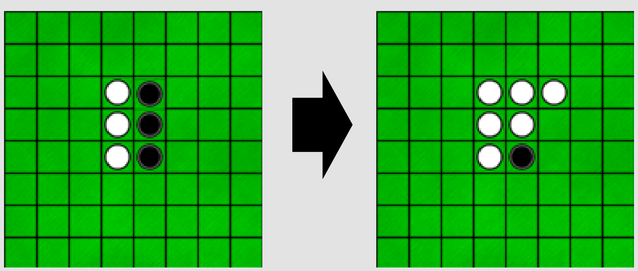
Step 11– Why Do the Good Guys Lose?

Observation:
If we let two instances of the game –with different search depths- play against each other, it is not always the case that the larger depth wins.

White Depth	Black Depth	Winner
0	1	Black
1	0	White
1	1	Black
0	2	White-Anomaly
2	0	White
1	2	Black
2	1	White
1	3	White-Anomaly
0	3	Black
3	1	Black-Anomaly
2	4	Black
4	2	Black-Anomaly
4	4	Black

The nature of Othello is such that the score can be completely reversed in only few turns

Here is a simple example:



Notice how the scores changed from (3 - 3) to (6 - 1) in only one move.

So why does the larger depth lose?

Because the larger depth algorithm is choosing a move that looks very good up to the depth 4, for instance. What it is missing is that if it had examined a few more turns, it would find out that this move is actually a terrible move and hands half the board to the opponent.