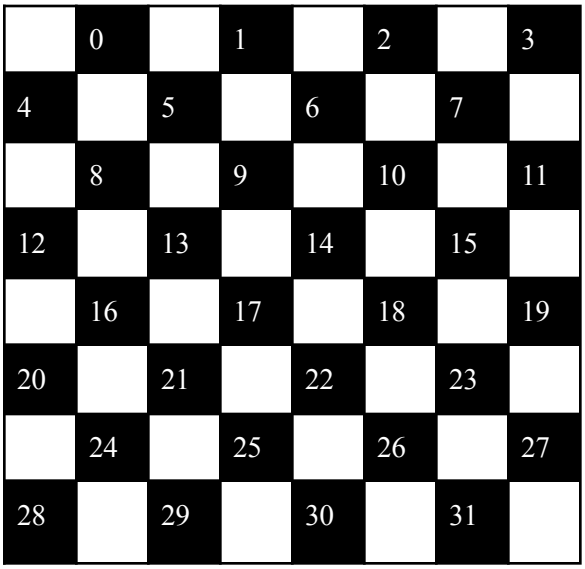
CHECKERS:

In checkers we have to store the location of the pieces, each type of piece, the number of pieces taken, and whose turn it is for any given state. I made a class Board to store these - a Board has a boolean for whose turn it is, two integers to store how many pieces have been taken for either side, and an array of 32 characters to represent the squares of the board.

The checkers board has 64 squares, but we only need 32 because checkers pieces can only ever be on one color since they all move diagonally. Here is a model of the board with indexes:



The board class has a few helper functions, but the most important one is move(). This function takes a move as input and returns the board that results from that move.

To store moves, you would think that all that needs to be stored is the index of the piece being moved, and the index of the spot it is moving to. However, to represent a move, I had to use the piece index and a quaternary tree of characters to store the move that it makes. It's ridiculous. Sadly, we have to do it this way because of a quirk of checkers gameplay: double jumps.

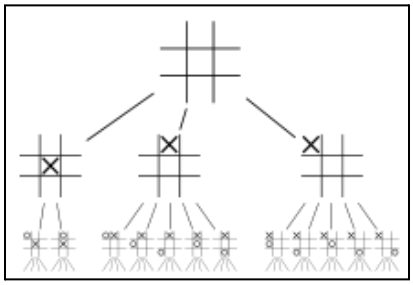
When a piece jumps another, it can perform another jump if the pieces line up. Sometimes, this results in a branching path of jumps that can be represented in a quaternary tree - the four children represent jump directions for up-right, up-left, down-right, and down-left. This made implementing checkers surprisingly hard, and forced me to write a few lines of code that can be very hard to follow. For example:

```
C/C++
std::unordered_map<char, std::unordered_set<QTreeNode<char>*>> movemap;
```

This data structure contains all of the moves that can be made on a given board. It maps the indexes of the pieces that can move to the possible moves they can make: each movable index has a set of moves. If an index can't move, it is not included in the map. This is used frequently during the engine to see all of the possible moves from a given board.

The Engine:

Surprisingly, making the checkers robot was easier than making the game itself. The checkers engine uses the minimax algorithm to search to a given depth, and optimizes it using alpha-beta pruning to increase the efficiency, and allow us to search to a farther depth.



What the bot does first is create a "game tree": a tree of boards that result from a certain root position. In the game tree, each edge represents a move, and each node represents a game board that results from its parent by the move adjacent to it.

Next, the engine "rates" all of the leaf nodes using a specified rating function. This rating function is the most important function in the entire program, as it can determine the behavior and "personality" of the bot. My rating system rates positions that favor the red side as positive, and positions that favor the black side as negative. I used a rating system that promotes keeping back rank pawns on their rank, which makes the bot defensive, but some ratings that I tried made the bot much more offensive. It was cool to experiment with different rating systems and see how they changed.

After that, the engine will use the minimax algorithm to rate nodes further up the tree. Basically, the engine will choose whichever move is best for whoever's turn it would be: the maximum rating is found for red's turn, and the minimum rating is found for black's - hence the name "minimax". We let that rating represent the rating for the node above. It does this recursively up the tree until the top, where it finally chooses the best move for the best rated board.

Depending on the game, a node in the game tree can have a lot of children depending on how many moves are available. In checkers, the average number of moves available in any given position is 8, but the number will vary. That means that searching down the game tree by depth+1 will result in ~8 times as many nodes being processed: an $O(8^n)$ operation! That can get really big really fast, so we have to use alpha-beta pruning as an optimization to cut the amount of nodes we need to look at.

← *Accurate depiction of how alpha-beta pruning affects the game tree.*

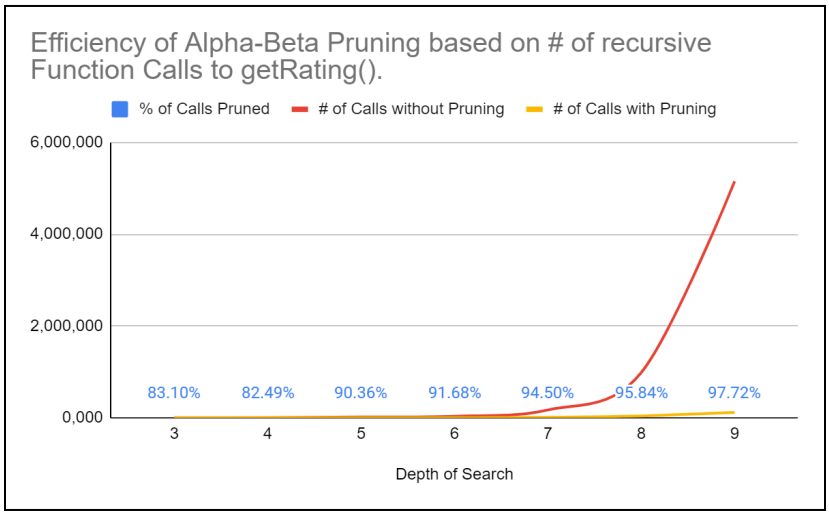
Alpha-Beta Pruning:

When the engine is looking for the maximum or minimum rating of a node, it has to look through every child to see if one of them has a better rating than the one that its parent currently has. However, sometimes we can stop looking halfway through if we know that a child will never choose a move that can "beat" its parents move. Since a parent will choose the maximum value, and a child will choose the minimum value, if any of a child's nodes have a rating that is lower than the parent's current maximum, then the child's pick will never be chosen by the parent. This allows us to cut off huge parts of the game tree, and is the essence of alpha-beta pruning.

My minimax function recursively traverses each node, so to calculate how many nodes are pruned, I used a counter variable to count how many times the function is called with and without the pruning. I sampled the first few moves from the starting position of a game and averaged the number of calls across all of the samples along a few different depths. Here are my results:

Depth	Calls without α-β Pruning	Calls with α-β Pruning	% of Calls Pruned
3	493.6	83.4375	83.10%
4	3172.77	555.604	82.49%
5	15661.4	1510.19	90.36%
6	33295.7	2769.16	91.68%
7	170210	9360.71	94.50%
8	984125.2	40964.6	95.84%
9	5150912.5	117422	97.72%

Alpha-beta pruning does a lot to help out the efficiency of this program. It saves on average 90% of calls to the function, and the number of calls pruned actually increases as the depth does! The number of calls still increases exponentially, but an improvement from traversing 5 million nodes to only traversing 100,000 is very impressive.



Rating Tournament:

Since the rating function changes the behavior of the engine, an engine with one rating function must win over an engine with another. So, I created four different rating functions that rate a board differently from each other and held a "tournament" to see which rating function was the best. Each rating function will go down to a depth of 9. Note that, for all rating functions, I tested pitting them against a lower depth, and the higher depth seemed to always win. Here are the results:

Basic	Rates only based on material advantage, rating kings as 5 points and pawns as 3 points.
Defensive	Rates the same as the basic engine, but gives a bonus for keeping pieces on the back-rank
Central	Rates the same as the basic engine but gives a bonus for back-rank pieces and for keeping pieces in the center.
Weighted	Rates the same as a defensive engine, but weighs situations with less pieces higher. Ex: a 2-1 advantage would be better than a 10-9 advantage, despite the same material advantage.

After running a round robin tournament where every engine fought every other engine with one game playing first, and one game playing second, here are the results:

Engine	**W**	**L**	**T**	**Comments**
Basic - 9 #4	1	4	1	Didn't perform very well, but was very clearly the fastest when loading at depth = 9.
Defensive - 9 #2	2	1	3	Often caused ties by holding up pieces in the corner. The opponent would capitalize on this by sacrificing a piece in a lost position to secure a tie instead of outright losing.
Central - 9 #3	1	3	2	Performed poorly and was very slow. Was able to tie with the defensive engine both times, though.
Weighted - 9 #1	5	0	1	Consistently won by a significant margin. This is partly because it would trade pieces to immediately simplify to an endgame.

Generally, the engines are good, but their performance in the later parts of games was very poor. They would run quickly, but wouldn't be able to close out a game for a long time, even when they were at an advantage.

REFERENCES:

https://lazyfoo.net/tutorials/SDL - Basic SDL2 guide

- Taught basic SDL over the summer, great tutorial. Helpful for setting up an SDL environment. No real impact on the checkers engine.

https://www.youtube.com/watch?v=STjW3eH0Cik - MIT Minimax Search Lecture

- Basic overview of minimax search.