Chess Algorithms

Noah Caplinger

September 1, 2021

ISBN 978-1-304-20194-2

Thanks to all my friends who read drafts of this book,

especially my internet-stranger friends Finn Eggers and Kim Kåhre,
who endured my numerous questions about their engine, Koivisto.

<3 Brito

Contents

Preface

I am not a professional computer scientist, nor a professional chess player. My training is in mathematics, and I might *charitably* be described as an 'advanced amateur' in chess and programming. At the intersection of these two topics are chess engines: computers that play chess. To me, engines are a constant source of fascination: the way they work, the way they play, and the ways they are optimized are enchanting topics. It was inevitable that I would build my own. Ever since, chess programming has been my one and only hobby which I am now sharing with the world.

My main goal in writing this book is the creation of a single (hopefully) well-written introduction to chess programming with the explicit goal of being accessible to newcomers. This stems from a general disappointment with the state of online chess programming exposition. Essentially everything I've learned on this topic has been from poorly-written wikis, blogs that must be found on the internet archive, stackexchange answers, papers from the 80s, and forums from the 90s. It's a pretty bleak picture. I'd like to change that.

In particular, most online resources suffer from the same illness of math expositors: they feel the need to present every topic in its full generality and abstraction the first time it is introduced. I strive to, whenever possible, give examples first and code later. This is the only true way to learn. I've also made an intentional decision to include as many diagrams as possible. There are plenty "explanations" of alpha/beta pruning without a single picture in sight—if you're lucky you'll get *one*. The corresponding chapter in this book has 26.

The audience I have in mind is roughly the person I was before taking up this amazing hobby: a non-(computer scientist) programmer with an interest in algorithms and chess. There are few prerequisites other than a basic grasp of recursion and algorithmic thinking. On the off chance a non-programming chess-enthusiast has picked up this book, I've also included a chapter on programming which can (and should) be skipped by all who have the skills.

As hard as I've tried to make the ideas in this book accessible, there is no escaping that they are complex and require real thought. Needing to re-read sections is not at all shameful. The worst quality a student can have is the fear of not understanding the first time. I can guarantee that there will be sections of the book you don't understand immediately. You will read and re-read these sections many more times than you think is reasonable. This is completely normal.

All that said, thank you for picking up this book. I sincerely hope you find it as fascinating as I do.

Part I.

The Basics

1. Chess

What follows is a *very* basic introduction to chess. Anyone with any familiarity with the game should skip this chapter.

1.1. The Game

Chess is a 2-player turn-based game played on an 8×8 board. The two players alternate moving pieces, (King, pawn, knight, etc.) which move in different ways and may sometimes "capture" opponent's pieces, removing them from the board. The objective of the game is to capture the opponent's King.[1]

In principal, the above paragraph is all that is needed to understand 90% of this book (only skip chapter 7), but it seems silly not to include a section on

1.2. Chess Fundamentals.

If you want to learn the rules of chess, there are excellent resources online which can teach chess much better than this book ever could. (I recommend https://lichess.org/learn#/). For the remainder of this chapter I will assume a passing understanding of the rules of the game, and will fill in any necessary details in the following sections.

1.2.1. The Relative Value of Pieces

Pieces are, of course, not all of the same value. Pawns aren't as important as Knights, which aren't important as rooks, etc. The below table displays the approximate value of each piece.

A few things to note here: first, why am I using multiples of 100? Why not divide everything out and deal with small numbers? This is a valid point, and indeed pieces are usually valued at 1/3/5/9, but it is sometimes preferable to use larger numbers when dealing with computers—it allows for more precise evaluations while still using whole-number arithmetic. Second, I put "value" in quotation marks, because chess is

[1] Ok *technically* we never capture the king and deliver "checkmate" instead. If this distinction is important to you, you're already too advanced for this chapter.

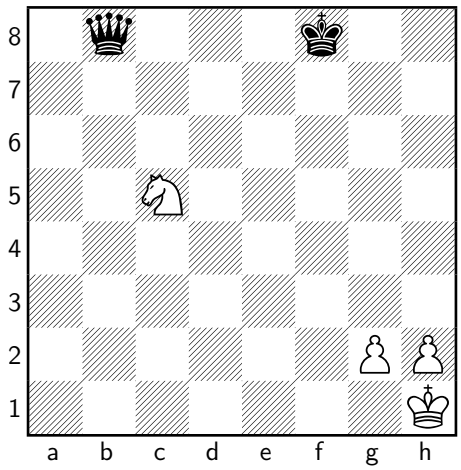
Pawn	100
Knight	300
Bishop	300
Rook	500
Queen	900
King	∞

not nearly that simple. The "true" value of any given piece fluctuates during a game depending on a myriad of factors. These are guidelines, not rules.

We've now brushed up against two vocabulary words: 1. centipawn - a unit used to measure how favorable a board position is for one player or another (positive for White winning, negative for Black). Strong players often require only a hundred or so centipawn advantage to win. 2. Material (advantage) - One player is said to have a material advantage if the sum of the value of their pieces is larger than that of their opponent's.

1.2.2. Tactics

Players can gain a material advantage through *tactics*, which are best explained through an example:



Here, white will play Nd7.[2] Note that this move attacks both the King and Queen. Black is now *forced* to move his king, say Ke7, and white will then play Nxb8, taking the queen and gaining a significant material advantage. White now has more than enough

[2]Quick overview of chess notation: There are letters on the bottom and numbers on the side of the board to denote columns and rows respectively. Then a1 is the bottom left square, the white knight is on c5, etc. **P**awns, K**n**ights, **B**ishops, **R**ooks, **Q**ueens and **K**ings are denoted P, N, B, R, Q, and K respectively. A move is specified by a piece and a square, so Nd7 means "the knight moves to d7". If there is a capture involved, you put an x between the piece and the square, so Nxb8 reads "Knight takes b8".

material to win the game. Black's move is *forced* in the sense that he has no choice but to move the king and give up the queen. This is a defining feature of tactics (sequence of moves that limits the opponent's options and result in tangible gain). I did not draw another board for you to see these moves as played—you had to imagine them. This process of imagining moves that might happen in the future is called *calculation*. As you might imagine, computers are very good at this.

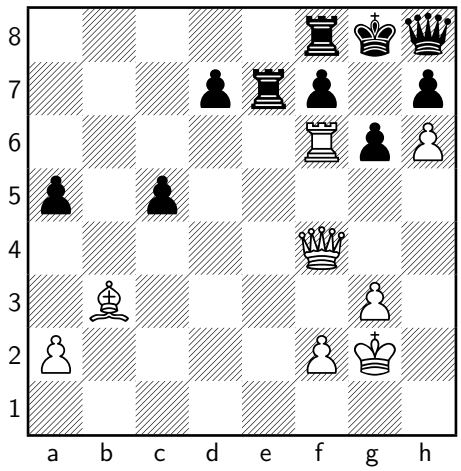
Having a lead in material is a significant advantage in chess. All else being equal, professionals require only the slightest of margins to convert a material advantage into a win. However, all is not usually equal, and these discrepancies are explained by

1.2.3. Positional Chess.

The finer points of chess revolve around one basic idea:

Pieces are most effective when they have the freedom to move around the board.

Thousands of books have been written on this basic idea, so I won't be able to do it justice in a page or so. Hopefully I'll be able to get the point across with an example:



This position is taken from a 100 game match Google's AlphaZero played against Stockfish 8 (two chess computing gods) in which AlphaZero won 28 and drew the remaining 72. There was some controversy regarding the fairness of this match, but I think it's fair to say Google impressed the chess community.

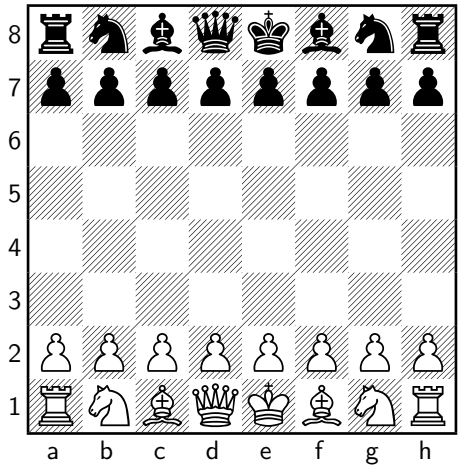
The first thing beginners will notice about this position is the material imbalance: Stockfish (Black) has an extra rook and two pawns to AlphaZero's bishop. Under normal circumstances, this would be more than enough to seal the deal, but this position is *far* from ordinary.

A more advanced player will notice is the positioning of Black's queen: it is in the corner, trapped by friendly pieces and the rook on f6. This piece, ostensibly valued at 9 pawns is stuck in the corner doing nothing. White also has an enormous amount of pressure on f7. The white Rook, Bishop and Queen are all pointed directly at the pawn. Both of Black's rooks are stuck defending f7—if either moves, white will quickly play Rxf7, and black's position falls apart.

Stockfish has *no* useful moves. Each of its major pieces are tied down, while AlphaZero's have free range. It will quickly play g4, g5 (defending the rook) then gobble up black's pawns with the queen, and march the `a` pawn to victory. Despite the material disparity, white has an *enormous* advantage.

1.2.4. The Opening: Development and Controlling the Center

Every game of chess starts with the same board configuration:



As such, this is the most studied position in all of chess. Thousands of books have been written about positional chess, and thousands more have been written on opening theory, a very modest amount of which will be important to creating a chess engine. These ideas are really just consequences of those discussed in the previous section, but it's worth stating them explicitly.

As any kid coming home from chess camp will tell you, there are three opening principles:

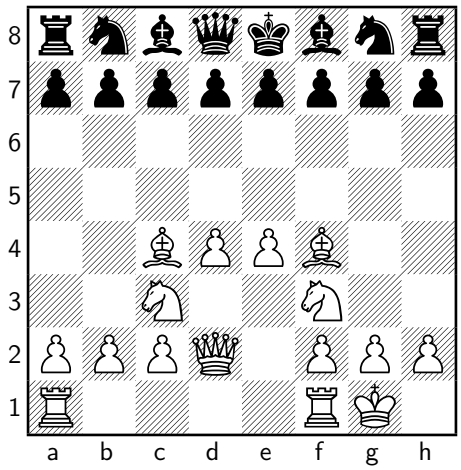
1. Protect your king
2. Develop your pieces
3. Control the center

Given that the the game hinges on the safety of the king, point 1 hardly needs justification. The other two are less obvious. By "develop your pieces," I mean to move

you pieces off their starting squares and into battle. The purpose of this is to give your pieces more freedom—they can hardly move around the board when they are trapped in the back.

The four central squares, and to a slightly lesser extent, the center 16 are of importance for two main reasons. First, every piece but the pawn has increased mobility there. Second, pieces in the center affect the entire board. Players without an established control of the center generally struggle to mobilize their pieces.

With these ideas in mind, the following position is often cited as the "best possible opening for white." (or at least that's what we tell kids)



Of course, Black will not let this happen without a fight—he will contest this bid for the center while developing his pieces and protecting his king. This is how most chess games begin: with a fight for the center and the deployment of pieces.

Chess masters have dedicated their lives to recognising and capitalizing on positional advantages. The difficulty of this task stems from its vagueness—what exactly is a positional advantage? How does one obtain one, and how can it be used to win? There are various heuristics which give partial answers to these questions, but real skill in this area comes from intuition and experience.

As you might imagine, it is very difficult to tell a computer to handle these imprecise ideas of positional chess, but much easier to get computers to find tactics. When definitions of success are precise and problems are constrained, computers tend to do very well. When success is vague and problems are open-ended, computers flounder. Our main task in building a chess engine will be to leverage the computer's strengths—raw calculation and speed—to play passable positional chess. But before we can do that, we must learn to program.

2. Programming

While it is not necessary to know how to play chess in order to understand this book, I find it very difficult to imagine a non-programmer following some of the later chapters on algorithms. This has nothing to do with understanding any particular language, or the ability to read pseudocode—it's a general familiarity with algorithmic thought.

That said, I do think there are things a non-programmer can learn from this book, and for these people I have included a section on the basics of coding. If you are one of these people, just be aware that this book may require supplemental programming tutorials, particularly in recursion.

This chapter can also serve as a gentle review for those who haven't coded in awhile.

2.1. Computers

For our purposes, a computer is simply a machine that follows instructions. You tell it to `print("hi")`, and it does. You tell it to compute, $2+2$, and it does. You tell it to find if $e^\pi > \pi^e$, and it will (provided it knows what π and e are). The only real catch here is that we can only use "simple" instructions: add/subtract/multiply/divide/compare... . The exact list depends on the programming language in question, but essentially all languages have the simple things you expect. I will be writing in pythonic "pseudocode."

We typically write lots of these instructions in one file, and have the computer run all of them in sequence. For example, running the following program

```
print("computers are awesome!")
print(2 + 2)
print(3 > 4)
```

will result in

```
computers are awesome!
4
False
```

Perhaps the most fundamental tools in programming are variables—things which store values. For instance the program

```
x = 3
```

```
print(x)
x = 5
print(x)
```

Will give

```
3
5
```

2.2. If Statements, For Loops and Functions (Oh my!)

So far, programming isn't all that interesting. Maybe useful if you want to do calculations, but not much more. However, among these "simple instructions" are tools for more complicated ideas which we can use to solve interesting problems.

2.2.1. If Statements

Using an "if statement," we can execute a specific bit of code *if* something is true. For instance,

```
if 3 < 4:
    print("hi mom")
    print("told you three is less than four!")
```

will give

```
hi mom
told you three is less than four!
```

The indentations on the third and fourth lines indicate the code which is to be run *if* the statement is true.

If statements by themselves aren't too useful—I already knew $3 < 4$, so I might as well have omitted the first line entirely. Their use becomes much more evident when used in conjunction with for loops.

2.2.2. For Loops

The following code

```
for x in [1,2,5,"hi mom"]:
    print(x)
```

will print

```
1
2
5
hi
```

Hopefully this example makes it very clear what's going on. The computer iterates the following process: 1. set `x` equal to the next element in the list, 2. `print(x)` 3. repeat until the list is empty. As with if statements, indentations are used to show which code is to be repeated. I will sometimes denote the "list" of legal moves in a chess position as `position.legal_moves`.

For loops are incredibly useful. The following bit of code you will see in some form many times throughout this book

```
alist = [1,2,3,10,3]

highestSoFar = -infinity

for x in alist:
    if x > highestSoFar:
        highestSoFar = x

print(x)
```

Give yourself a moment to try to figure out what this does. Answer in the footnotes[1].

2.2.3. Functions

Functions are a way to teach the computer "new instructions" from ones it already knows. Perhaps you want your program to find the maximum of many different lists in many different parts of your program. It would be silly to repeat the above code *every time*, so instead we can write a function (read: new instruction):

```
def max_of_list(alist):
    highestSoFar = -infinity
    for x in alist:
        if x > highestSoFar:
            highestSoFar = x
    return x
```

Now whenever you want to find the maximum of a list `L`, you simply write `max_of_list(L)` and the computer will execute the indented code, evaluating that line to the maximum element of `L`.

Our main task in this book is to find a function that takes in a board position, and returns what it believes is the "best move."

[1] It will print the largest element of `alist` which in this case is 10.

2.3. Recursion

Recall the definition of the factorial $n!$

$$n! = n(n-1)(n-2)\cdots 3 \cdot 2 \cdot 1.$$

This can be rewritten as

$$n! = n \cdot (n-1)!.$$

Note that together with the base case $1! = 1$, the above equation actually *defines*[2] the factorial function. To see this, note that for instance $5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$. This definition is 'recursive' in the sense that it involves repeatedly applying the definition until we reach the 'base case'.

Recursive functions work in much the same way: solving a problem by repeatedly solving a smaller version of the same problem until we reach the base case. For instance, a function which takes in an integer n and returns $n!$ can be written as

```
def factorial(n):
    if n = 1:
        return 1
    return n*factorial(n - 1)
```

We will be using recursion extensively in this book. If you are not familiar with it, I recommend finding some online resources and coming back.

[2] yes, yes, $0! = 1$, but we can also find an analytic extension of ! on $\mathbb{C}$, so I don't want to hear it.

Part II.

Fundamental Algorithms

3. The Evaluation Function

This chapter is a brief discussion of the most naive chess-playing algorithm and why it does not work. Its primary purpose is to motivate chapter 4, and is skippable if the reader is comfortable with the phrase "a function which takes in a board position and returns a 'value' of the position which *roughly* corresponds to who is winning".

3.1. The Physicist's Method

Citizens of the internet with nerdy inclinations will recognise the following xkcd:[1]

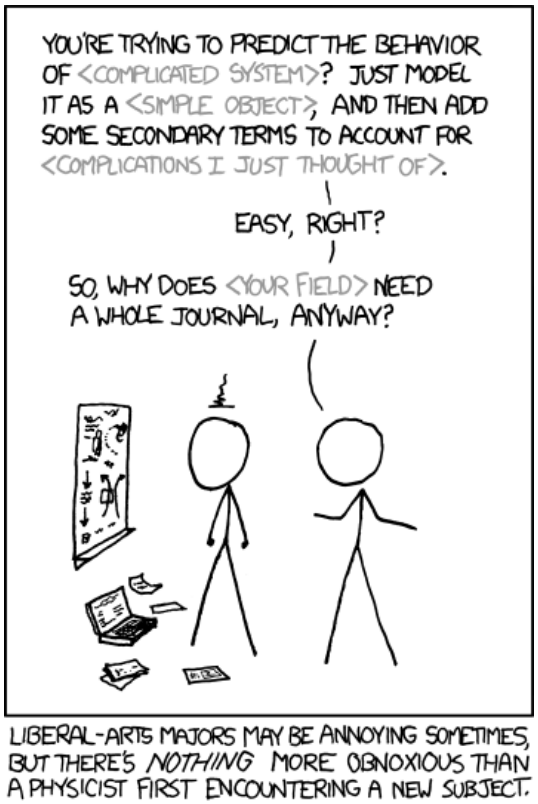


Figure 3.1.: a relevant xkcd.

[1] xkcd is a wonderfully nerdy webcomic. I imagine the people who voluntarily pick up this book form a strict subset of the people who would like xkcd. This particular comic can be found at https://xkcd.com/793/

"You're trying to find a good chess move? Just write a function that assigns a number to each board position which reflects material imbalance, then add/subtract from that number according to development and central control. Choose the move that results in the board state with the highest number.

...

Why did someone write a book about this?"

This isn't too hard to implement:[2]

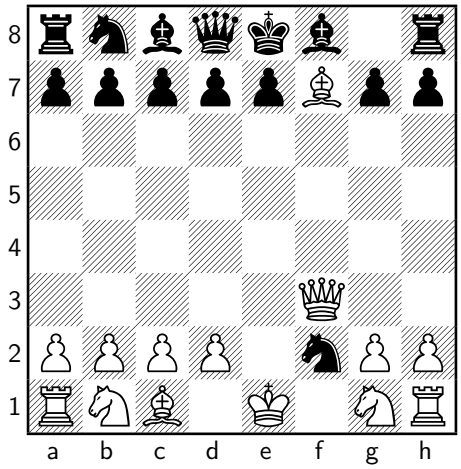
```
def evaluate(position):
    score = 0
    for piece in position.pieces():
        score += piece.value()
        if not piece.isDeveloped():
            score -= 20
        if piece.isInCenter():
            score += 20
    return score

def bestMove(position):
    bestSoFar = -infinity
    candidate = None
    for move in position.legal_moves:
        value = evaluate(after_move(position,move))
        if value > bestSoFar:
            bestSoFar = value
            candidate = move
    return candidate
```

So how does this algorithm do?

3.2. Physicists are Stupid

I beat it in four moves:

[2] As discussed later, the algorithms we discuss will only be concerned with finding the *value* of the best move, not the best move itself. This is the final example where I actually return a move.



The game starts 1. e4 Nf6. e4 is a very common first move, and the computer chooses the move which most immediately improves the 'score', Nf6 in this case because it develops a piece towards the center. I then play Bc4 because I'm crafty and have a plan. The computer then play Nxe4, immediately grabbing material. I play Qf3, threatening checkmate on f7. The computer, oblivious to this fact continues on gobbling up pawns, playing Nxf2. I then have two options to end the game, and I chose the more stylistic.

The computer is suffering from a complete lack of foresight.

The (perhaps obvious) point I am trying to make is that chess is complicated. It's difficult to create a simple function that tells you when a position in chess is winning or losing.

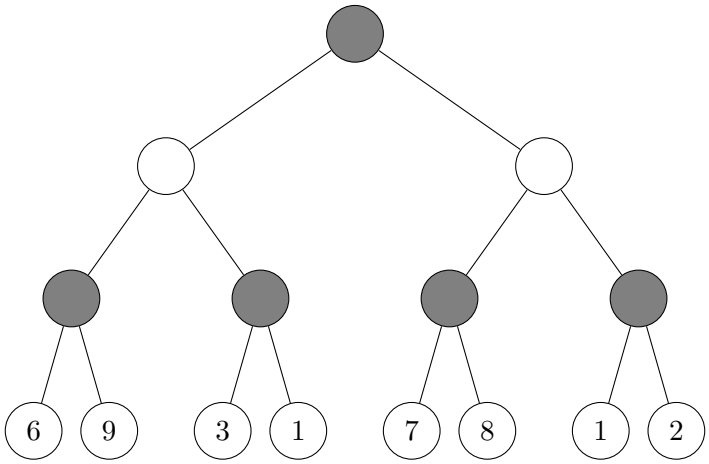
You might object that I simply need to tell my computer to calculate future positions. This is indeed the direction we are headed, but first I want to point out two things 1. by no means is calculating future positions 'simple' (we will spend the remainder of the book on this topic) and 2. calculate future positions *and look for what*? It should definitely look for checkmates so as to not play silly games like the one above, but what else? At some point, we will have to use some kind of value judgment to determine what board positions are favorable. Whether we like it or not, the physicist might have to stay awhile.

4. MinMax

This Chapter gives an introduction to the MinMax[1] algorithm, and the NegaMax framework. We discuss its limitations and performance. If you are comfortable with these topics, the relevant sections may be skipped. Note that the NegaMax framework will be used throughout the rest of this book.

4.1. Game Trees and Tree Games

Before we start talking about chess, I want to introduce a related game between two players, 'White' and 'Black'. Consider the following tree:



Each circle is called a 'node', and the colors are purely cosmetic.

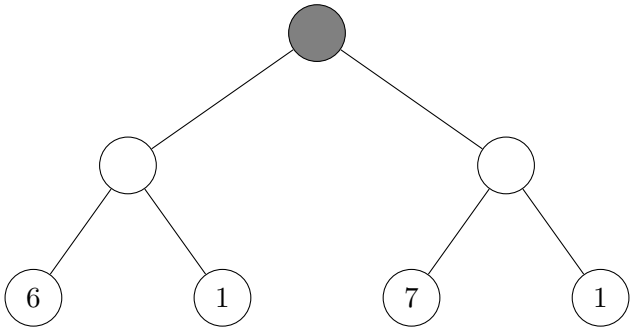
Here's the game: we start at the top (root) node. Black goes first, and chooses one of the two nodes immediately below the root (a 'child' node). We move to the chosen node, White chooses one of its children, and we move to that node. Black then chooses one of its children, and they continue alternating until they reach a node with a number, called a 'terminal' or 'leaf' node. Say this number is x. White receives x dollars, and Black receives $\$(10 - x)$.

A very natural question arises from this game: Assuming perfect play (that is, Black minimizes x, White maximizes x, and both are perfect logicians), who gets what amount of money? I'll let you think about it.

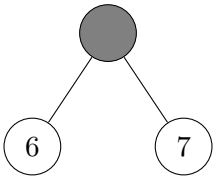
[1] Most people use "MiniMax" instead of "MinMax." I find this extra i abhorrent.

This is the way I (a human) think about the problem: If they ever arrive at the leftmost black node, it will be Black's turn, in which case he will choose the node with 6, as he is trying to minimize the number. Similarly, if they ever arrive on the rightmost node, it will also be Black's turn, and he will pick the node with 1. Continuing with this process, we can see that if we arrive at the 1^{st}, 2^{nd}, 3^{rd}, or 4^{th}, lower black nodes, x will be 6, 1, 7 or 1 respectively.

Crucially, White is aware that Black will make these choices, so these black nodes may as well just be terminal nodes with values 6, 1, 7 and 1. Then, from White's perspective, the tree looks like this:



Now we can use the same logic one level up: deduce what White will do if they arrive at the right and left nodes in the second layer, and fill in the appropriate values. Then we get



Black will clearly choose 6, so we can conclude that *with perfect play*, the two players will traverse the tree by always choosing the leftmost node.

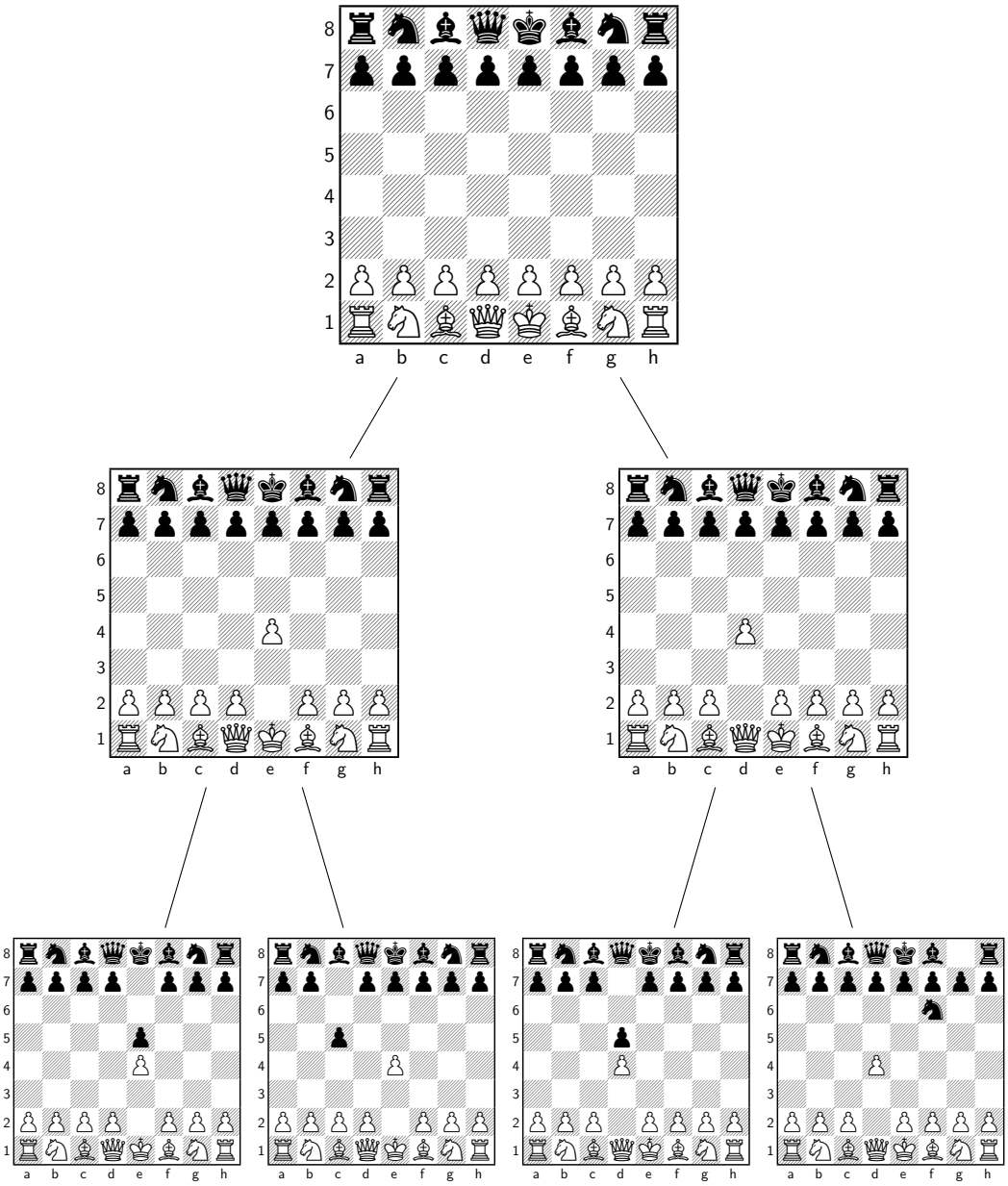
It's clear how to generalize this process: Starting at the bottom, label each node with the min/max of it's children (depending on the color of then node, or equivalently the parity[2] of the distance to the root). Continue doing this until each node is labeled.[3] The label of the root node is the value x. I encourage you to perform this process yourself.

[2] 'parity' is the even or odd-ness of a number. For instance, you might say "2 and 3 have opposite parity."

[3] If you want, you can think of this as an elaborate knockout bracket, where the rounds alternate between being contests of minimum and maximum value.

But what does this have to do with chess? As it turns out, chess is really just this tree-game dressed up as a board game. To see this, consider each board position[4] a node, and the resulting board positions of each legal move it's children. Also consider the starting board position to be the root node, and label every game-ending position with -1, 0 or 1 if it is a Black win, draw or White win respectively. The resulting tree is called the 'game tree' of chess. Hopefully this picture of an incomplete game tree provides some clarity:

[4] where each board position contains not only piece positions, but also information about castling rights, who's turn, en passant, etc.



Now every possible game of chess can be interpreted as a sequence of steps along the game tree, and playing a game of chess can be understood as playing a version of the tree-game. This means that in order to analyze chess, we can instead analyze the tree-game, which is in many ways easier to conceptualize.

Knowing just the value of child nodes is enough information to find the best move (just pick the one with the largest/smallest value). To simplify matters, our algorithms will be concerned with computing the *value* of a particular position, not explicitly finding the best move, even though these things are functionally equivalent.

Because there are only finitely many possible games of chess, you might think we can simply use the above algorithm to 'solve' the game. In theory, this would work. Just one problem: there are more than 10^{120} possible games[5]. This number is so phenomenally large that you can do silly thought experiments: If you were to assign every atom in the observable universe a supercomputer, and let them all run in sync until our sun explodes, they would still not calculate all possible games of chess. Not even close. I'm sure you can come up with your own and use them to annoy your family.

For now, we will use a rather crude solution to this problem: choose some arbitrary depth d, maybe it's 5. Starting with the game tree, cut off every node whose distance from the root node is larger than d. Then, we will assign values to terminal nodes based on the evaluation function we talked about in the previous chapter. This depth 5 tree is much more manageable (read: can be searched completely before the sun explodes), but contains less information. By searching this stunted tree, the computer is effectively choosing the move which best optimizes the evaluation function 5 moves in the future.

There are several problems with this approach, but for the moment, we've solved the problem identified in the previous chapter: our computer can now 'think ahead' and anticipate moves.

In chess lingo, a 'move' is a turn by each player, so when I said "...optimizes the evaluation function in 5 moves," is technically incorrect (at least in this terminology). I really meant 'half-move' or 'turn'. To avoid ambiguity, we instead call these half-moves 'ply'. So the above algorithm "searches at a depth of five ply".

4.2. The Algorithm

The algorithm described in the previous section is fine. It works, it's intuitive and easy to explain, but if you sit down and try to write it,[6] you may run into some difficulties. For one, how do you know which nodes are unlabeled? When the tree only has a few nodes this is easy, but if we are going to be modeling a game of chess, each node will have 30 or so children, and the tree explodes quickly. You certainly don't want to loop over every node every time you iterate the process—that would be wildly inefficient. Instead, you might keep track of which ones are labeled and which are unlabeled. But then you still have to determine which nodes do and do not have unlabeled children, so you might try to find another fix for that... The code is a mess.

Fortunately, we can salvage the main ideas of the previous algorithm and repackage them. The first key idea is that we can assign a 'value' to intermediate nodes by assuming perfect play from both players and deducing their actions. The second is that the value of any node is simply the minimum or maximum of the value of their children. This suggests a recursive approach:

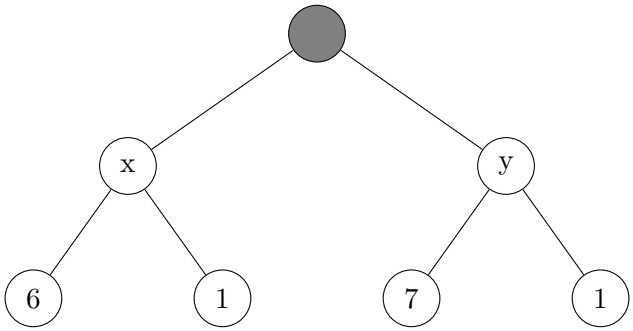
[5] This is the oft-cited 'Shannon's number', which was first mentioned in exactly this context 70 years ago when chess programming was in its infancy.

[6] which I encourage you to do!

```
def MinMax(node): // To find the value of node
    if node.isTerminal():
        return node.value()
    if turn = White:
        bestSoFar = -infinity
        for child in node.children():
            // Finding the maximum value of node's children
            value = MinMax(child)
            if value > bestSoFar:
                bestSoFar = value
        return bestSoFar
    if turn = Black:
        bestSoFar = infinity
        for child in node.children():
            // Finding the minimum value of node's children
            value = MinMax(child)
            if value < bestSoFar:
                bestSoFar = value
        return bestSoFar
```

Where `node.isTerminal()` is true when node is a terminal node, `node.value()` is the value of a terminal node, and `node.children()` is a list containing the node's children.

At a high-level, this algorithm is essentially saying "to find the value of a node, first find the values of its children,[7] then take appropriate minimum or maximum." It's important to have this kind of high-level understanding of your code, but it's equally important to understand exactly what's going on. The rest of this book is fundamentally about variations of this algorithm. To give you a nuts-and-bolts feel for the above algorithm, I will explicitly trace the code for the intermediate tree we found earlier. Here it is again:



I adopt the convention that nodes are searched left to right. Let `r` be the root. I will now explicitly compute `MinMax(r)`. All lines with the same indentation have the same local scope.

[7] This is done recursively. If that didn't make sense to you, it may be helpful to learn/review recursion.

`r` is not a terminal node, and it is black, so we will jump down to the third 'if statement' and set $\texttt{bestSoFar}_r = \infty$. We now loop over `r`'s children, computing the value of each using the MinMax function. We will compute them left to right:

> {Computing `MinMax(x)`: `x` is not a terminal node, and it is white, so we will enter the second if statement. $\texttt{bestSoFar}_x$ (which is a different variable than the one above) is now negative infinity. Now we must compute the values of it's first child:
>
> > {Computing `MinMax(6)`: This node *is* a terminal node, so we simply return the value 6 }
>
> 6 is larger than $\texttt{bestSoFar}_x = -\infty$, so we set $\texttt{bestSoFar}_x = 6$. For the next iteration of the for loop, we will find the value of `x`'s next child:
>
> > {Computing `MinMax(1)`: This node *is* a terminal node, so we simply return the value 1 }
>
> 1 is not larger than 6, so we don't change $\texttt{bestSoFar}_x$. We now exit the for loop, and return 6.}

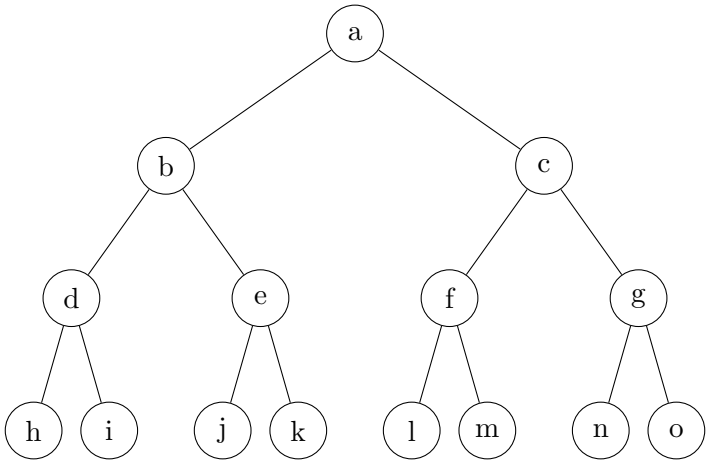
6 is less than $\texttt{bestSoFar}_r = \infty$, so $\texttt{bestSoFar}_r$ is now 6. Onto the next iteration of the for loop: computing `MinMax(y)`

> {Computing `MinMax(y)`: `y` is not a terminal node, and it is white, so we will enter the second if statement. We set $\texttt{bestSoFar}_y = -\infty$ and proceed to compute the value of `y`'s first child:
>
> > {Computing `MinMax(7)`: This node *is* a terminal node, so we simply return the value 7. }
>
> 7 is larger than $\texttt{bestSoFar}_y = -\infty$, so we set $\texttt{bestSoFar}_y = 7$. For the next iteration of the for loop, we will find the value of `y`'s next child:
>
> > {Computing `MinMax(1)`: This node *is* a terminal node, so we simply return the value 1 }
>
> 1 is not larger than $\texttt{bestSoFar}_y = 7$, so we don't change $\texttt{bestSoFar}_y$. We now exit the for loop, and return 7. }

Now, the value of `MinMax(y)` is larger than $\texttt{bestSoFar}_r = 6$, so we do not change $\texttt{bestSoFar}_r = 6$, exit the for loop, and return 6.

If you found this at all confusing, re-read the section on recursion, and possibly find some resources for it online. I can't overstate how important it is in this book.

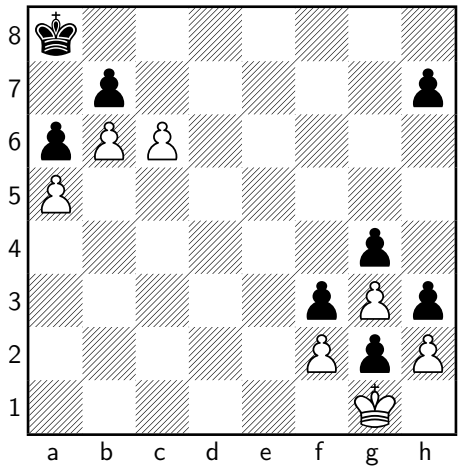
Note the order our algorithm traversed the nodes: b, x, 6, 1, y, 7, 1. As an exercise, find the order the nodes are traversed in the tree from the beginning of the chapter:



Answer in the footnotes[8]

4.2.1. MinMax-ing a Mate-in-two.

Consider the following puzzle:

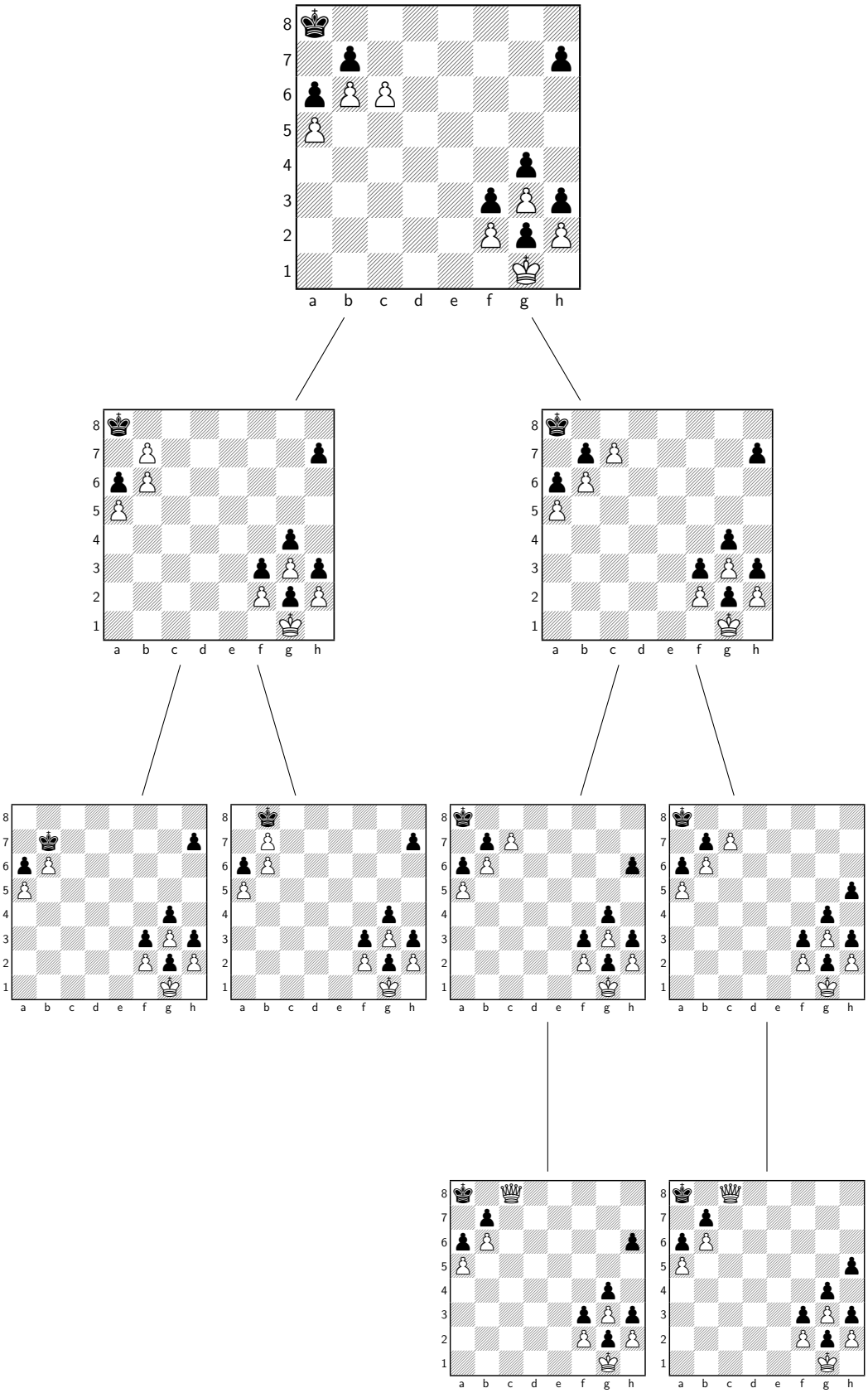


If you would like to take a stab at it, the answer is in the footnotes.[9] If you used any chess intuition to solve it, try to also answer this question using pure brute force—calculate every possible sequence of moves, and choose the best one for White.

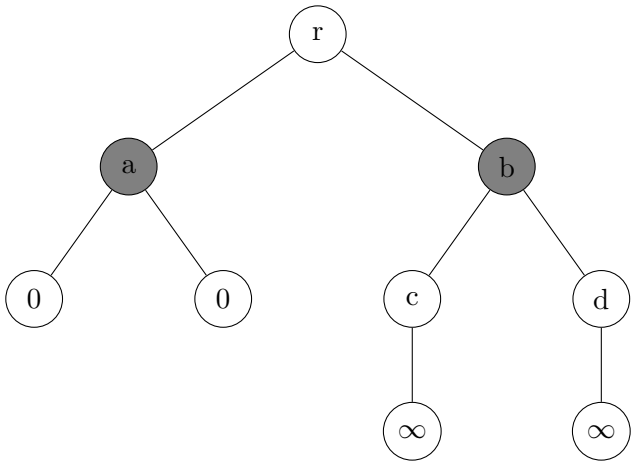
Let's look at how the MinMax algorithm solves this puzzle at depth 3. The first step is to make a game tree.

[8] a, b, d, h, i, e, j, k, c, f, l, m, g, n, o

[9] White should play c7, then promote to a queen/rook on the next move.



Next, we must assign each terminal node an evaluation. Reading left to right, the first two terminal nodes are stalemates—value 0—and the last two are checkmates for white—$+\infty$. We can now forget about the chessboards and work abstractly with trees:



I will be following the code on page 19. As before, all lines with the same indentation have the same local scope.

{Computing `MinMax(r)`: `r` is not terminal and it is white, so we move to the first if statement. We set $\texttt{bestSoFar}_r = -\infty$ and begin searching `r`'s children, going left to right.

> {Computing `MinMax(a)`: `a` is not terminal and it is black, so we move to the second if statement. We set $\texttt{bestSoFar}_a = \infty$ and begin searching `a`'s children.
>
> > {Computing `MinMax(0)`: `0` *is* a terminal node, so we just return its value, 0.}
>
> $0 < \texttt{bestSoFar}_a = \infty$, so we update $\texttt{bestSoFar}_a = 0$, and proceed to the next child of `a`.
>
> > {Computing `MinMax(0)`: `0` *is* a terminal node, so we just return its value, 0.}
>
> $0 \not< \texttt{bestSoFar}_a = 0$, so we do nothing. This is the final child of `MinMax(a)`, so we exit the for loop and return $\texttt{bestSoFar}_a = 0$}.

Then `a` has value $0 > -\infty$, so we update $\texttt{bestSoFar}_r = 0$, and proceed to the next child of `r`.

> {Computing `MinMax(b)`: `b` is not terminal and it is black, so we move to the second if statement. We set $\texttt{bestSoFar}_b = \infty$ and begin searching `b`'s children.
>
> > {Computing `MinMax(c)`: `c` is not terminal and it is white, so we move to the first if statement. We set $\texttt{bestSoFar}_c = -\infty$ and begin searching `c`'s children.

{Computing `MinMax(`∞`)`: ∞ *is* a terminal node, so we just return its value, ∞.}

$\infty > -\infty$, so we update $\texttt{bestSoFar}_c = \infty$. There are no more children of c, so we return $\texttt{bestSoFar}_c = \infty$.}

$\infty \not< \infty$, so we do not update $\texttt{bestSoFar}_b$ (which is currently set to ∞). Proceeding to the next child of `b`...

{Computing `MinMax(d)`: `d` is not terminal and it is white, so we move to the first if statement. We set $\texttt{bestSoFar}_d = -\infty$ and begin searching `d`'s children.

{Computing `MinMax(`∞`)`: ∞ *is* a terminal node, so we just return its value, ∞.}

$\infty > -\infty$, so we update $\texttt{bestSoFar}_d = \infty$. There are no more children of d, so we return $\texttt{bestSoFar}_d = \infty$.}

We again see that $\infty \not< \infty$, and do not update $\texttt{bestSoFar}_b$. There are no more children of b, so we return $\texttt{bestSoFar}_b = \infty$.}

$\infty > 0$, so we update $\texttt{bestSoFar}_r$ (which is currently set to 0) to be ∞. `r` has no more children, so we return ∞.}

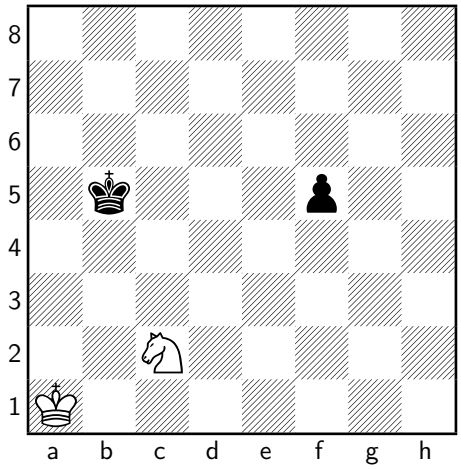
Can you see how this process matches the brute force method?

4.3. Performance

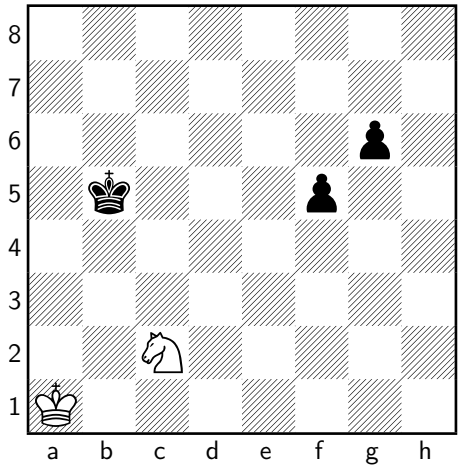
MinMax is... ok.[10] Depending on your language of choice and specific implementation, it can search somewhere between 3 and 5 ply in a reasonable amount of time. Playing against it can feel a little weird. It'll sometimes make perfectly normal, intelligent moves, and other times makes decisions that defy logic (unless of course you know how it's making those decisions).

Take for example the following board:

[10] it sucks



Even at a depth = 3, the computer will recognise the obvious tactic: Nd4+, king moves, say Kc5, then Nxf5, winning the pawn. Here, the computer sees this because it gains a material advantage (something the evaluation function understands) within 3 ply. However, this 'thought process' can be fooled:



Here, the depth = 3 engine will still play Nd4+! Even though the f5 pawn is defended, the computer still believes it gains a material advantage through exactly the sequence from before: Nd4+, Kc5, Nxf5. As humans, we can see that if sequence is actually played, black's g6 pawn will take on f5, and white will lose the knight. But this sequence is 4 ply long, and is therefore invisible to the engine.

In this situation, after playing Nd4+, Kc5, the engine would wise up to the g5 pawn (as it is now inside the 3 ply horizon), and move the knight elsewhere. In this example, it makes little difference where the knight goes, but it's not difficult to imagine a situation in which that wasted move could be extremely consequential.

This is the principal weakness of MinMax: it simply doesn't search far enough in the game tree. You can't blame it. An average chess position has 30ish legal moves, so a...

... ply search	requires searching ... nodes.
1	31
2	931
3	27,931
4	837,931
5	25,137,931
6	754,137,931
7	22,624,137,931
⋮	⋮

But here's the thing: almost none of those 22 billion nodes we could be searching make any sense whatsoever. Human players routinely calculate 7 ply by considering only 40 or so board positions. 22,624,137,931 vs 40. This difference is absurd! Chess algorithms (and indeed the contents of this book) are all about finding clever ways of reducing this gap to something more manageable.

4.4. NegaMax and Cleaning Code

There are two quality of life changes I will be making to the code as written in section 4.2. These will not affect the algorithm itself, but will make our lives easier down the road.

The first change relates to our understanding of a terminal node. Instead of "cutting off" branches the game tree with large depth, it will be easier to keep a `depth` counter, and use it to determine what is and isn't a terminal node. I will also be switching from tree terminology (node, child, etc) to chess terminology (position, move, etc.). Our code now looks like this:

```
// To find the value of the position at a certain depth
def MinMax(pos, depth):
    if depth = 0:
        return evaluate(pos)
    if turn = White:
        bestSoFar = -infinity
        for move in pos.legal_moves:
            value = MinMax(after_move(pos, move), depth - 1)
            if value > bestSoFar:
                bestSoFar = value
        return bestSoFar
    if turn = Black:
        bestSoFar = infinity
        for move in pos.legal_moves:
            value = MinMax(after_move(pos, move), depth - 1)
            if value < bestSoFar:
                bestSoFar = value
        return bestSoFar
```

Convince yourself that the above code is essentially the same as that in section 4.2.

Technical Aside: When actually implementing this code, real programs will make the move, evaluate the board position recursively, then 'unmake' the move. This has significant speed advantages, but makes the code a little jumbled, so I'm going to omit it by using `after_move(pos, move)` to represent the board position after the move.

The code written above is a bit unwieldy. It's split into two parts which both do roughly the same thing, only opposite. This becomes a nightmare to maintain. You want to make a minor tweak? Do it twice, but make sure you get the signs right. Debugging? Have fun discovering the $<$ was supposed to be a $>$. Fortunately, there is a clever way to combine the White and Black cases called NegaMax. It may take some getting used to, but I speak from experience when I say it's well worth your time.

To illustrate the idea of NegaMax, I'm going to start with a rather silly analogy.

Imagine I'm sitting on the West side of a (very long) table which has model train-tracks running North/South and a train in the middle. My brother and I have decided to play the following game: we take turns pushing the train down the tracks,[11] each time starting from where the other's push stopped. If I can push the train off the North side of the table, I win. If he can push it off the South side, he wins.

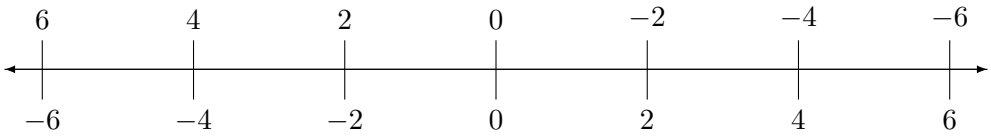
We're interested in writing instructions for each player. The simplest instructions

[11] standing still and pushing down the tracks, not running alongside the table

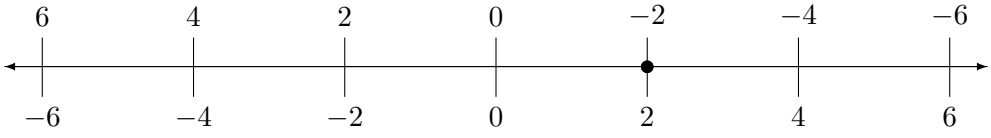
might be "If you're name is Noah[12], push North. If you are Noah's brother, push to the South." This has the disadvantage of having two different sets of instructions (one for each player). If you thought a little harder, you might come up with an alternative: "stand on the opposite side of the table from your brother, and push to the right" which uses a single set of instructions for each player. This is (roughly) the idea of NegaMax.

If you need a way to relate this back to chess, imagine the position of the train as an indication of who is winning—North for me, South for my brother. Instead of pushing, we are choosing moves which will result in a favorable train position. The first set of instructions is analogous to the MinMax algorithm described earlier in the chapter.

To make this example more concrete, imagine the number line... err... the train tracks are labeled on each side by numbers increasing to the right:



Further imagine that when referencing positions on the tracks, I use the numbers on my side of the table, and my brother uses the numbers on his side. So if the train is currently at the point



and you were to ask "where is the train," my brother would reply "−2", while I would say "2". It's easy to see that in order to "translate" between our two perspectives, you multiply by −1.

This has the potential to introduce some confusion. If our father walks in on us furiously pushing his model train back and forth (and we skip the part where he becomes a combination of angry and confused) he might ask "what's the score?" I would say "2," but my brother would say "−2". We can resolve this by agreeing to use the numbering system of the player to move.

This might all seem needlessly complicating, but now **both** players are attempting to maximize their score. This is significant because we can now follow the same set of instructions: "push the train in the direction of increasing numbers."

---

[12]That's me, the author

NegaMax is just MinMax when we use this new scoring system—positive numbers favor the player to move.

MinMax can be summarized as follows:

- if depth is 0, evaluate
- move 1: find the value of the position after I make move #1 (at a smaller depth).
- move 2: find the value of the position after I make move #2 (at a smaller depth).
- ⋮ ⋮ ⋮
- (last move): find the value of the position after I make move #(last move) (at a smaller depth).
- choose the min/max of the values I found above.

Go back, look at the code, and make sure this makes sense.

NegaMax is the same as the above, except every time I "find the value of the position after I make move #n," I am doing so from the perspective of my opponent (as making a move changes the turn), meaning I have to multiply the values by -1 to "translate" them. Each player is maximizing their score, which makes the code much more succinct:

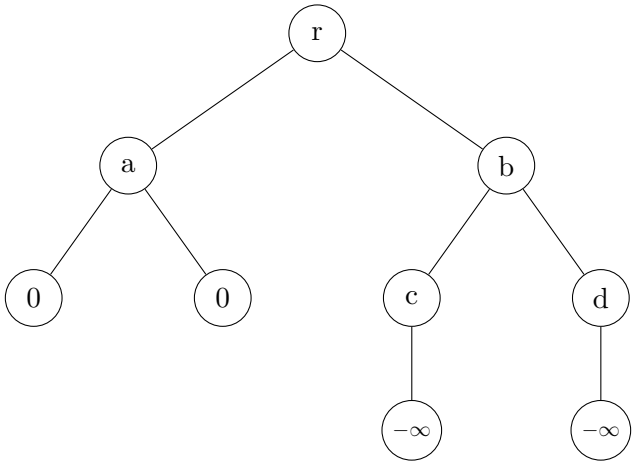
```
def NegaMax(pos, depth):
    if depth = 0:
        return evaluate(pos)
    bestSoFar = -infinity
    for move in pos.legal_moves:
        value = -NegaMax(after_move(pos, move), depth - 1) // note the negative
        if value > bestSoFar:
            bestSoFar = value
    return bestSoFar
```

You may not see the importance of this now, (and you may think MinMax is easier) but as we continue adding bells and whistles to the code, the search algorithm becomes *massive*. Much of this book can be understood without NegaMax, but the same can't be said of building an engine.

To avoid ambiguity regarding using or not using NegaMax with game trees, I will adopt the following convention: trees which alternate white/black nodes will be traced with white maximizing and black minimizing, while trees with no shaded nodes will be traced with NegaMax.

I'll finish this chapter with a worked example of NegaMax. Below is the tree from the mate-in-2 example, with two changes: 1. The nodes are not shaded 2. the ∞ nodes

have changed to $-\infty$. The first change simply indicates that we are searching this tree using NegaMax. The second change is because we are now measuring the evaluation of terminal nodes from the perspective of the player to move. Those nodes are black,[13] and from black's perspective, being checkmated is the worst possible thing, hence the value $-\infty$.



I will be following the code on page 30. Note the similarity to the worked example on page 24. As always, all lines with the same indentation have the same local scope.

[13]These nodes are black in the sense that the nodes alternate white-black-white-black-$\cdots$. I previously haven't been giving shading in terminal nodes color so you can see the entry clearly. Besides, does it make sense to say that when black is checkmated, it is his turn?

{Computing `MinMax(r)`: `r` is not terminal, so we set $\texttt{bestSoFar}_r = -\infty$ and begin searching `r`'s children, going left to right.

{Computing `MinMax(a)`: `a` is not terminal so we set $\texttt{bestSoFar}_a = -\infty$ and begin searching `a`'s children.

{Computing `MinMax(0)`: `0` *is* a terminal node, so we just return its value, 0.}

$(-1) \cdot 0 > \texttt{bestSoFar}_a = -\infty$, so we update $\texttt{bestSoFar}_a = 0$, and proceed to the next child of `a`.

{Computing `MinMax(0)`: `0` *is* a terminal node, so we just return its value, 0.}

$(-1) \cdot 0 \not> \texttt{bestSoFar}_a = 0$, so we do nothing. This is the final child of `MinMax(a)`, so we exit the for loop and return $\texttt{bestSoFar}_a = 0$}.

Because $(-1) \cdot \text{value}(a) = (-1) \cdot 0 > -\infty$, we update $\texttt{bestSoFar}_r = 0$, and proceed to the next child of `r`.

{Computing `MinMax(b)`: `b` is not terminal so we set $\texttt{bestSoFar}_b = -\infty$ and begin searching `b`'s children.

{Computing `MinMax(c)`: `c` is not terminal, so we set $\texttt{bestSoFar}_c = -\infty$ and begin searching `c`'s children.

{Computing `MinMax(`$-\infty$`)`: $-\infty$ *is* a terminal node, so we just return its value, $-\infty$.}

$(-1) \cdot -\infty = \infty > -\infty$, so we update $\texttt{bestSoFar}_c = \infty$. There are no more children of c, so we return $\texttt{bestSoFar}_c = \infty$.}

$(-1) \cdot \infty \not> -\infty$, so we do not update $\texttt{bestSoFar}_b$ (which is currently set to $-\infty$). Proceeding to the next child of `b`...

{Computing `MinMax(d)`: `d` is not terminal, so we set $\texttt{bestSoFar}_d = -\infty$ and begin searching `d`'s children.

{Computing `MinMax(`∞`)`: $-\infty$ *is* a terminal node, so we just return its value, $-\infty$.}

$(-1) \cdot -\infty = \infty > -\infty$, so we update $\texttt{bestSoFar}_d = \infty$. There are no more children of d, so we return $\texttt{bestSoFar}_d = \infty$.}

We again see that $(-1) \cdot \infty \not> -\infty$, and do not update $\texttt{bestSoFar}_b$. There are no more children of b, so we return $\texttt{bestSoFar}_b = -\infty$.}

$(-1) \cdot -\infty = \infty > 0$, so we update $\texttt{bestSoFar}_r$ (which is currently set to 0) to be ∞. `r` has no more children, so we return ∞.}

5. Alpha/Beta Pruning

Alpha/Beta pruning is a partial solution to MinMax's problem of computing many, many useless lines. The basic idea is that we can *prove* that searching some nodes is pointless in certain situations. Alpha/Beta pruning is *the* the most important topics in all of chess programming. Do not skip this chapter.

5.1. (Shallow) Pruning

Consider the following hypothetical story:

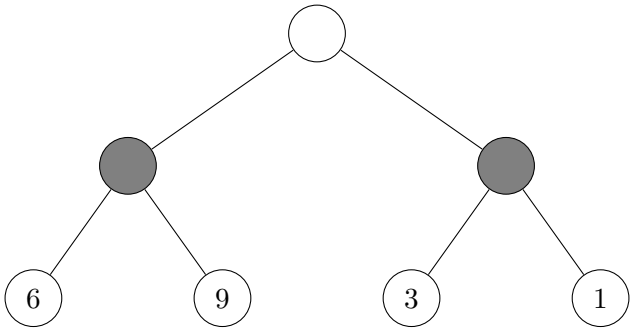
My brother and I really enjoy this card game. It's one of those games where you physically go to a store and buy a pack of cards. Unlike most of those unethical, borderline underage-gambling games, each pack comes labeled with its contents—you know what you're getting when you buy.

I, being the older brother, have some extra spending money and invite my brother to the store. Because I'm a good guy, I let him choose which pack I'll buy, and promise to give him a card from the pack. But I also tell him I'm no charity—I'm always going to give him the worst card in the pack (let's assume each card has some numerical value). When we get to the store, there are only two packs. The first has only two cards valued at 6 and 9. The second pack is very thick and when we flip it over it has 100 cards. The values are listed: $3, 7, 9, 11, 8, \ldots$

Despite the fact that it would take him at least a minute to read all 100 numbers, he immediately tells me I should buy the first pack.

Hopefully it's clear how he makes this choice: my brother doesn't actually need all the information available to him in order to make an informed decision. If I buy the first pack, he's guaranteed to get a 6. When he starts reading the back of the second pack, he see that the first number is a 3. This means that if I buy the second pack, he's getting *at most* a 3 (and possibly worse). This is all he needs to realize that the first pack will give him a better card.

This is the basic idea behind alpha/beta pruning. Sometimes, you don't need to know the exact value of a node in order to know for sure that it's worse than another node. This idea is so important that I'm going to re-state and re-solve the card-problem a few more times, all in slightly different ways. It might become a bit tedious, but I'd much rather over explain this topic than under explain it. Consider the intermediate sub-tree from the beginning of chapter 4:



We proceed in the same way we did in the previous chapter (no NegaMax yet):

First, visit the leftmost black node, and find the minimum of its children. This is 6, so we know leftmost black node has value 6. We move on to the second black node, and look at its children. The first is a 3. This means that the second black node has value *at most* 3. Because we are only interested in finding the maximum of the two black nodes, and that

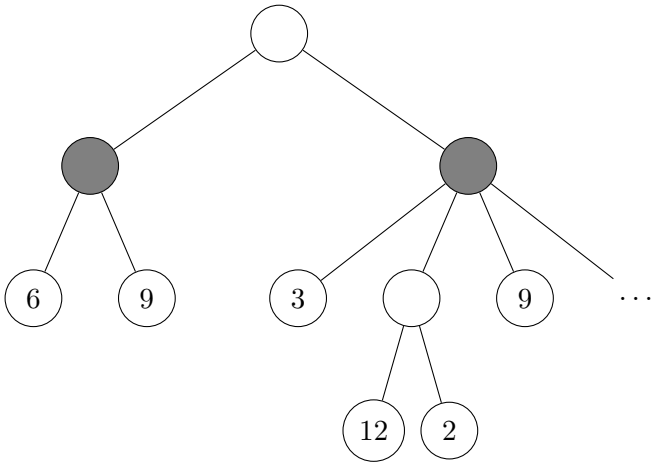
$$\text{value of rightmost black node} \leq 3 < 6 = \text{value of leftmost black node}, \tag{5.1}$$

we know

$$\text{value of rightmost black node} < \text{value of leftmost black node} = 6$$

and therefore that the maximum of the black nodes is 6.

Note that the thought process would be exactly the same if the graph looked like this:



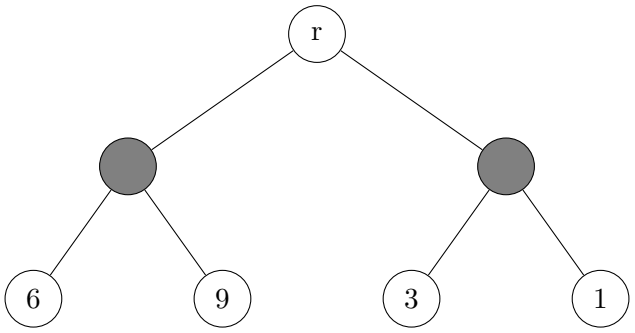
We simply do not need to search the subtree of all other nodes connected to the leftmost black node, so we may 'prune' this branch from the game-tree. This may not

seem so impressive with small trees, but it can make a *massive* difference when each node has 30 children.

When comparing two nodes, A and B, if we discover A is inferior to B, we should not waste resources determining *exactly* how inferior. This is the fundamental idea of alpha/beta pruning. We are looking for situations where the value of a node is *at best* worse than an alternative.

Everything up to this point has been fairly straightforward, but the actual implementation in code can be tricky. Instead of jumping directly into that, first I want to investigate the exact circumstances in which we can use this trick. That is, situations similar to the inequality 5.1.

But we need to be much more precise than "situations similar to..." if we are going to code this up. To do this, I'm going to explicitly trace through the tree and see exactly what's going on. I've redrawn it for your convenience. Note that we are still not using NegaMax, so the white nodes are maximizing and the black nodes are minimizing.



We call `MinMax(r)` (page 19). `r` is not a terminal node, but it is a white node, so we go to the first if statement. We set `bestSoFar` to $-\infty$ and enter the for loop. On the first iteration, we (recursively) evaluate the left most black node, and find it's value to be 6. We then set `bestSoFar` to be 6. Upon the next iteration of the for loop, we evaluate the rightmost black node (call it `rbn`), and set *its* `bestSoFar` to be ∞. after searching `rbn`'s first child, we can improve `rbn`'s `bestSoFar` to be 3.

stop.

At this point, we already know that the value of `rbn` is ≤ 3, so no matter what, it will not be larger than 6, the value of `r`'s `bestSoFar`. Then, the value of `r`'s `bestSoFar` will not change, so searching any more of rbn's children is pointless.

This gives us the precise circumstances we were looking for: If `r` is a white node,[1] and c is its child (a black node) we may stop searching c's children whenever the value of c's `bestSoFar` becomes smaller than `r`'s `bestSoFar`.

[1] think about what happens when `r` is black

This may take a moment to digest. Don't move on unless you are comfortable with the above statement.

5.2. Code, Negamax, and an Example

We can implement the idea at the end of the previous section by adding a parameter which specifies the previous `bestSoFar`:

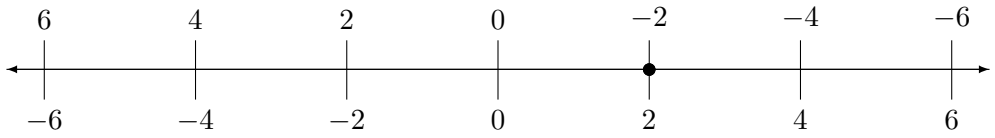
```
// Note: we are not using NegaMax
def MinMax(pos, depth, previousBestSoFar):
    if depth = 0:
        return evaluate(pos)
    if turn = White:
        bestSoFar = -infinity
        for move in pos.legal_moves:
            value = MinMax(after_move(pos, move), depth - 1, bestSoFar)
            if value > bestSoFar: // to update bestSoFar
                bestSoFar = value
            if bestSoFar > previousBestSoFar: // check if you can prune
                return BestSoFar
    if turn = Black:
        bestSoFar = infinity
        for move in pos.legal_moves:
            value = MinMax(after_move(pos, move), depth - 1, bestSoFar)
            if value < bestSoFar: // to update bestSoFar
                bestSoFar = value
            if bestSoFar < previousBestSoFar: // check if you can prune
                return BestSoFar
    return bestSoFar
```

When we prune, we stop the process of searching *c's* children, meaning this pruning happens when we call `c`. `r` is white when `c` is black, so the statement in the previous section applies when `c` is a black. If we do decide to prune, we return the current value of `bestSoFar`. This is analogous to my brother using 3 as a stand-in for the value of the second pack.

There is still a question of what the first value of `previousBestSoFar` should be—there is no previous move! We don't want to prune anything on the first level, so `previousBestSoFar` should be set to ∞ when the root is white, and $-\infty$ when the root is black.

I think we can all agree that the above code is ugly and unwieldy (only gets worse from here). The solution, of course, is NegaMax, but this can make things a bit confusing.

Because we're using NegaMax, both sides are maximizing the negative of the opponent's positions. This messes up our '`previousBestSoFar`' logic, as the value of the previous `bestSoFar` now represents the opponent's `bestSoFar` *from to opposite perspective*. The solution to this problem is best explained using the model train tracks example from before:



Say I (from the perspective of the bottom numbers) have calculated my `bestSoFar` to be 2, and continue searching my brother's moves. Under the old pruning system, we would see that he, as the minimizing player from the perspective of the bottom numbers, should prune whenever he finds a node with value less than `previousBestSoFar` (see the code on the previous page). However, when using NegaMax, he uses the top numbers. Looking at the picture, it's clear that being less than 2 on the bottom numbers is equivalent to being larger than -2 for the top numbers. Multiplying by -1 in order to change perspective also swaps the direction of $<$ and $>$.

Similarly, say my brother calculates *his* `bestSoFar` to be -2 (from the top perspective, 2 from the bottom), and starts searching my moves. From the old pruning system (which uses only the bottom numbers), I (being the maximizing player from the perspective of the bottom numbers) should prune whenever the value of my `bestSoFar` exceeds $2 = -(-2)$ (here, we use only the bottom numbers).

Then *each* player should prune whenever their `bestSoFar` is larger than the negative of the previous `bestSoFar`. Re-read the previous two paragraphs and convince yourself of this.

To implement this change, we will pass the negative `bestSoFar` through as `previousBestSoFar`, (which "changes the perspective") then play as though we are always the maximizing player:

```
// The NegaMax version of shallow pruning
def NegaMax(pos, depth, previousBestSoFar):
    if depth = 0:
        return evaluate(pos)
    bestSoFar = -infinity
    for move in pos.legal_moves:
        value = -NegaMax(after_move(pos, move), depth - 1, -bestSoFar) // here
        if value > bestSoFar:
            bestSoFar = value
        if bestSoFar > previousBestSoFar:
            return bestSoFar
    return bestSoFar
```

This may make more sense with math:[2] Say `c` is a child of `n`, and let bsf(n), bsf(c) be the `bestSoFar` of `n,c`, and pbsf(c) be `c`'s `previousBestSoFar`. That is, pbsf(c) = −bsf(n). *If* we are ever in a situation where bsf(c) > pbsf(c), then we know

$$\texttt{NegaMax(c, *, *)} \geq \text{bsf}(c) > \text{pbsf}(c).$$

We can then multiply everything by −1 to the inequality from `n`'s perspective:

$$-\texttt{NegaMax(c, *, *)} \leq -\text{bsf}(c) < -\text{pbsf}(c) = \text{bsf}(n)$$

Then, we know that

$$-\texttt{NegaMax(c, *, *)} < \text{bsf}(n)$$

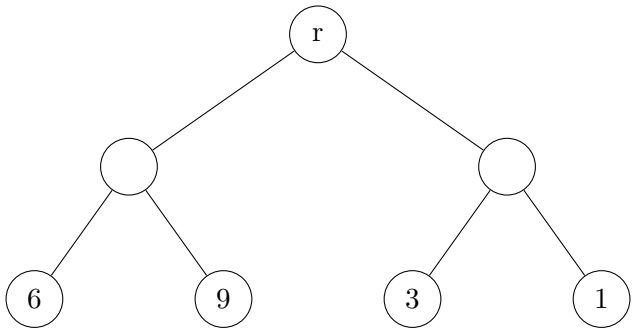
which means that the value of `-NegaMax(c, *, *)` will never exceed `n`'s `bestSoFar`, and that is is pointless to continue searching this line.

To summarize: *If* bsf(c) > pbsf(c), then we should prune. This explains the final if statement.

If you found the above confusing, don't worry. I remember being *completely* stumped by this business of passing through negative bounds. If this is you, I *highly* recommend manually working through all of the exercises in the next section with pencil and paper (and eraser!). Do them first with normal MinMax, then with NegaMax, and compare the methods. Find intuitive explanations for why white or black choose what they do ("If white chooses this node, then black can force white to choose..."). If this isn't enough, make your own examples and test them with your computer.

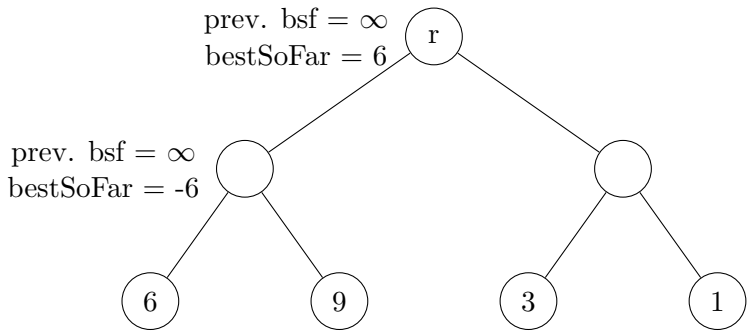
There isn't any escaping that these are difficult ideas, but believe me when I say the payoff is worth it.

We've now made great progress! Below is a copy of the tree from the beginning of the chapter. I'll now trace through it using the NegaMax algorithm

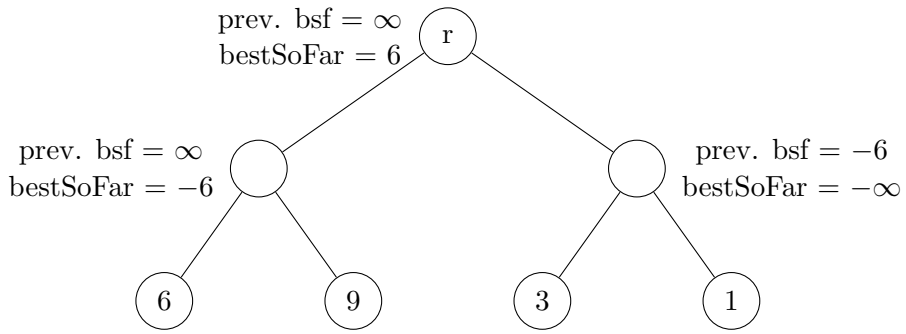


[2]Careful readers will note that this argument is the exact same as the one above, only with mAtH.

We start by calling `MinMax(r, 2, ∞)`. We aren't at depth 0 yet, so we set `bestSoFar` to $-\infty$ and enter the for loop. The first child/move is the left black node (lbn). We set value equal to `-MinMax(pos, depth - 1, -bestSoFar)`. Hopefully, it's easy[3] to see that `MinMax(lbn, 1, ∞)` is -6. Then, we set value equal to $-(-6) = 6$, and update the value of `r`'s `bestSoFar` to be 6. If you were tracing it out on paper, it might look like this:



On the next iteration of the for loop, we arrive at the right black node (rbn), and set value equal to `-MinMax(rbn, 1, -6)`, so we will now evaluate `MinMax(rbn, 2, -6)`. The tree now looks like this:



We now loop through the children of `rbn`, first finding `-MinMax((3), 0, ∞)`. This evaluates to -3, so we update `bestSoFar` to -3. Obviously, $-3 > -6$, so the pruning kicks in and we return -3. This means `-MinMax(rbn, 2, -6)` evaluates to 3. We see that $3 < 6$, so we do not update the value of `r`'s bestSoFar. Finally, 6 is not larger than ∞, so we're done, and conclude that the value of the tree is 6.

[3] if it isn't easy, review chapter 4

5.3. Exercises

I thought for awhile about whether I should include exercises in this book. Demanding my reader do problems seems reminiscent of not-so-fond memories of school, and I want chess programming to be a source of fascination, not drudgery. On the other hand, reading text in order to understand something can only go so far. You really have to get your hands dirty to learn. I've made my decision.

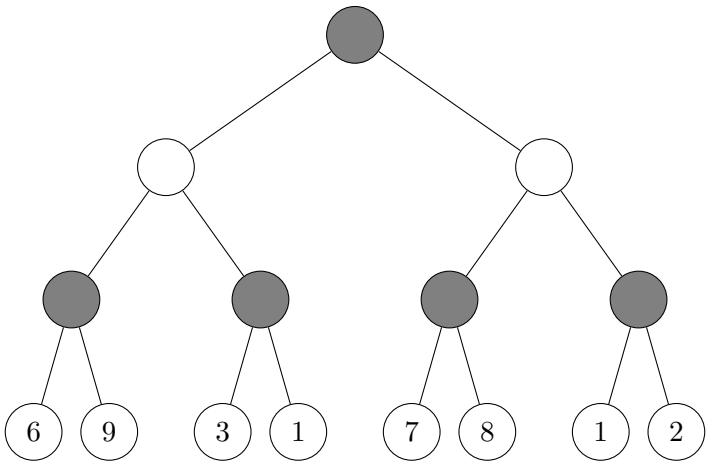
Eat your broccoli.

The instructions for each of these Exercises are as follows: Trace through the game tree using MinMax, NegaMax, MinMax with pruning, and NegaMax with pruning. After completing each algorithm, tell yourself a story about why the result is what the algorithms say ("white choose this node because she wanted black to choose between..." or "We don't have to search that node because..."). Convince yourself that the algorithms work. *Believe* it.

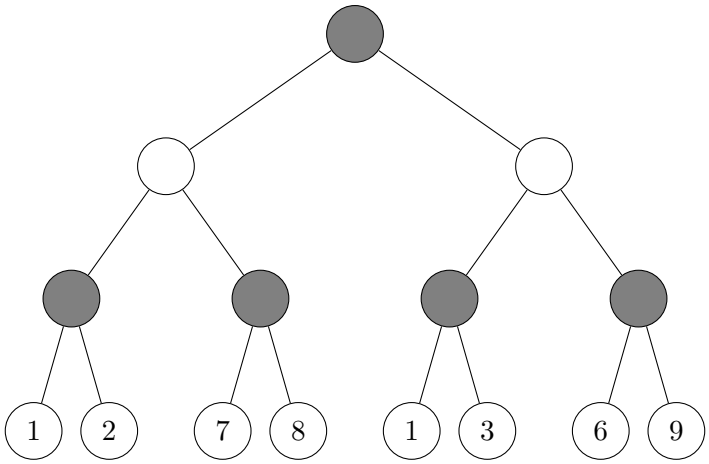
You don't have to do each algorithm for each exercise, but you *should* feel comfortable with them all when you're done. This is not the time to have a wishy-washy understanding. Ignore my warnings at your own peril.

Be sure you get the same answers for each algorithm. The nodes that have been 'pruned away' and not searched will be listed at the end of the section for you to check your answers.

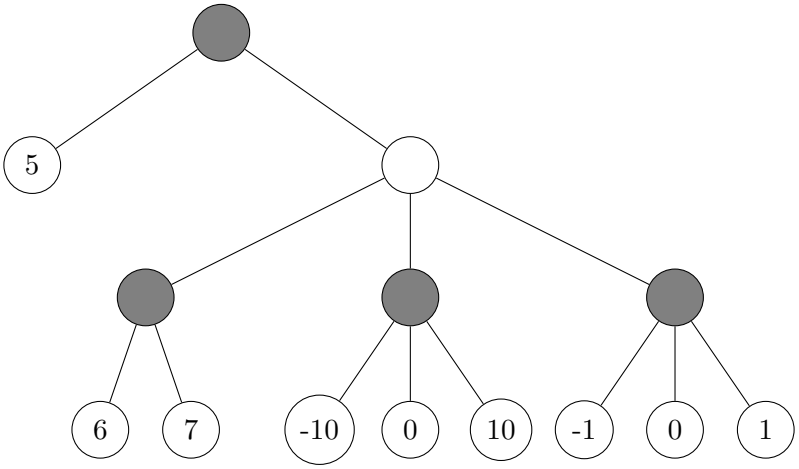
Exercise 1: The tree from the beginning of the chapter 4.



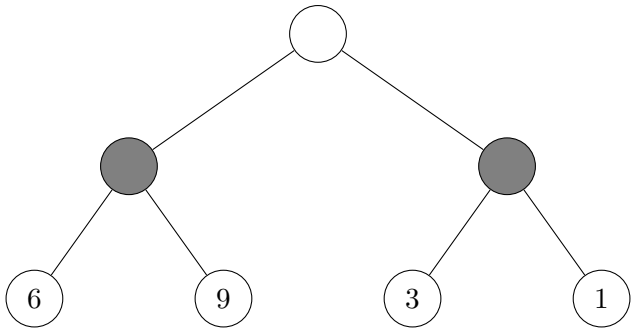
Exercise 2: The 'same' tree, but shuffled around.



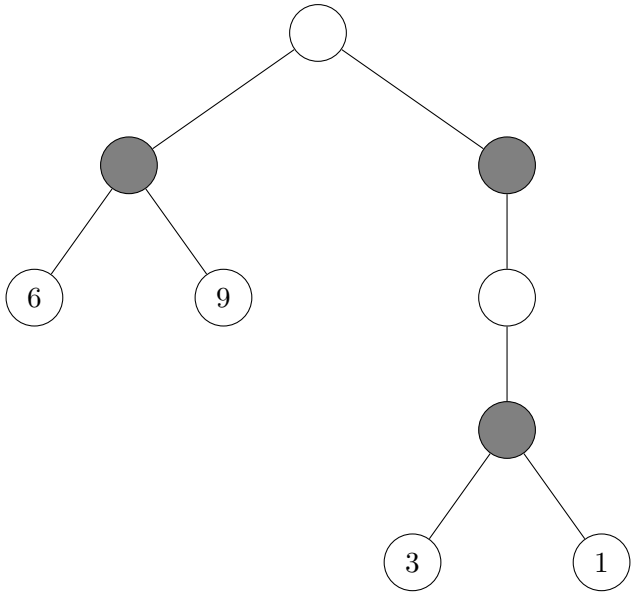
Exercise 3: Easier than it looks.



Exercise 4: Should be very familiar



Exercise 5: Suspiciously similar...



Answers

1: As before, the value of the root is 6. When pruning, we don't search the leftmost 1, or the rightmost black node (or its children).

2: Again, the value of the root is 6, but this time we don't prune—all nodes are searched.

3: The value of the root is 5. If pruning, we don't visit either of the two rightmost nodes or any of their children. Compare the work done with and without pruning, and imagine

what would happen if that middle white node had 30 children instead of 3. Kinda gives you an appreciation for the algorithm.

4: If you need me to answer this, you haven't been paying any attention.

5: The value of the root is 6, but we prune NOTHING. Because of the similarities between this problem and the previous one, you might expect that we would prune the rightmost white node, but alas we don't. At least not yet.

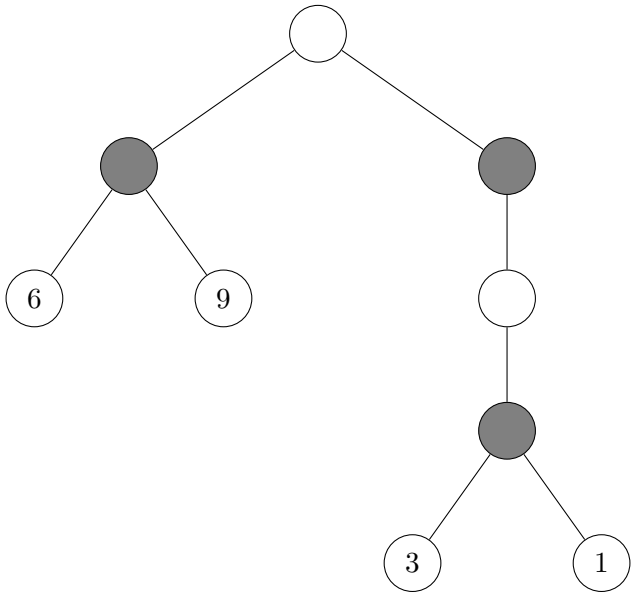
5.4. Alpha/Beta Pruning

The algorithm described in the previous section can be improved significantly. Fortunately, this optimization doesn't require any new ideas, so it shouldn't be too painful. If you are having difficulty with this section, I'd recommend going over the previous ones more carefully.

Until the last subsection, we will *not* be using NegaMax, so White is maximizing, and Black is minimizing.

5.4.1. The Problem

You may have noticed something interesting in the last exercise. If you trace through it as we did at the beginning of the chapter, using pruning logic, you can see that we "should" prune at the rightmost white node. Here it is again:



After tracing through the tree and arriving at 3, we know that the bottom black node (bbn) has value ≤ 3. Now I pose the question: does the exact value matter? It doesn't!

We already know that the white player has the option of a guaranteed 6 via the first left branch. To her, a 3, 1, and -100 are the exact same: worse. We don't need to know the exact value of bbn; we already know that it's inferior.

However, the algorithm in the previous chapter doesn't recognise this situation as a pruning possibility. That algorithm only uses information about the previous node (`previousBestSoFar`), and can't see "multi-generationally". This is the "shallow" in the title of section 5.1.

Make sure you understand the problem before we discuss the solution.

5.4.2. The Solution

Instead of recording the `bestSoFar` of the previous node, as we did before, we will record pruning information of *every* ancestor. This may seem like a lot, but it can be encapsulated in only two numbers (I wonder what we will call them...). Before we do that, I will make two clarifications on what I mean by 'ancestor':

1. everyone is ancestor of themselves
2. Only 'direct ancestors' count (no aunts/uncles).

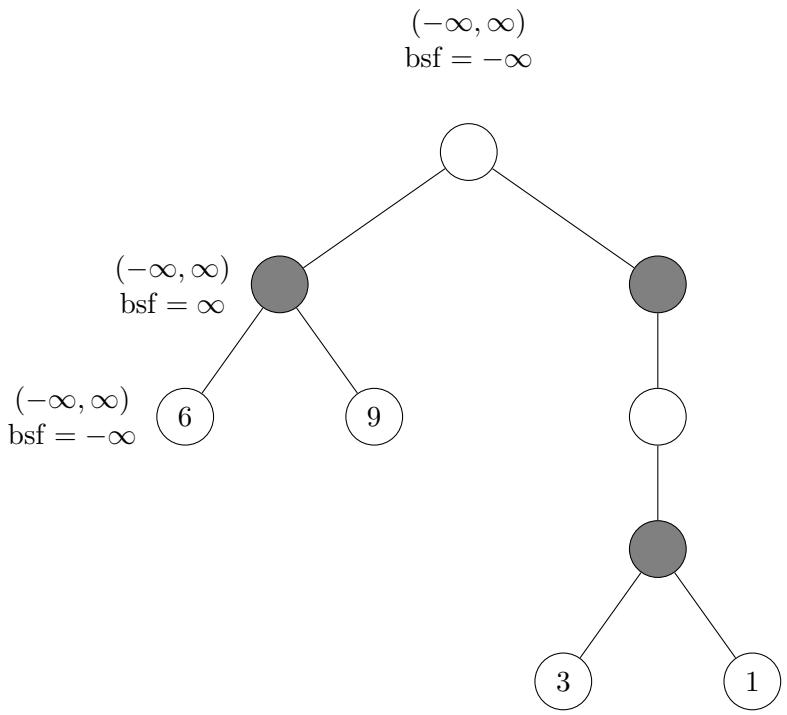
Definition 1 (α and β)**.** *To each node, we will assign the numbers α and β. The first number, α is the maximum of the `bestSoFar`'s of all white ancestors. Similarly, β is the minimum of the `bestSoFar`'s of all black ancestors.*

Note that this is a dynamic definition; the values α and β of a particular node will change depending on the value of that node's `bestSoFar`. The reason for these particular numbers will become apparent, but for now just roll with it.[4]

Let's see how this α/β system works in MinMax by labeling each node with its (α, β) as we trace through the tree. After recursing down to the node 6, the tree will look like this:[5]

[4] In general, I think this is a bad pedagogical strategy, but in this case, it's easier to show rather than tell.

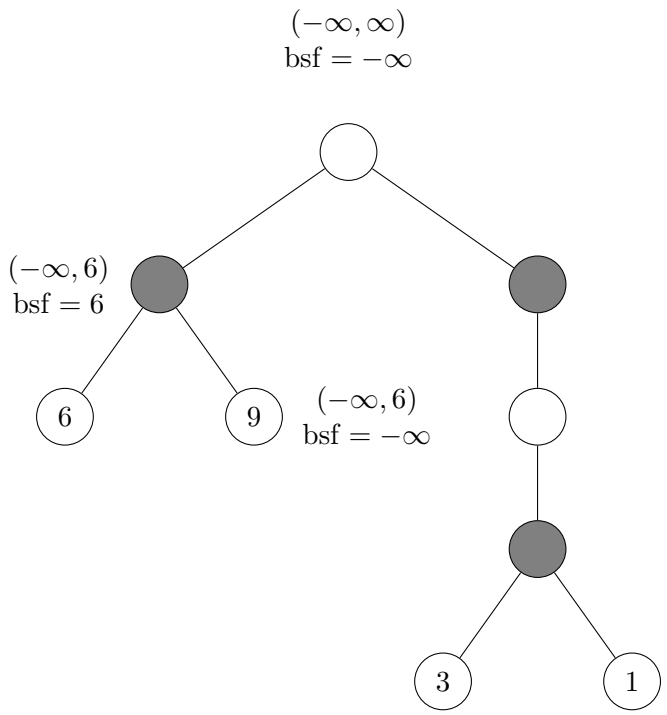
[5] You may note that the root node has no black ancestors. If you know any real analysis, you know what to do. Otherwise, you can adopt the convention that the (α, β) of a root node is always $(-\infty, \infty)$.



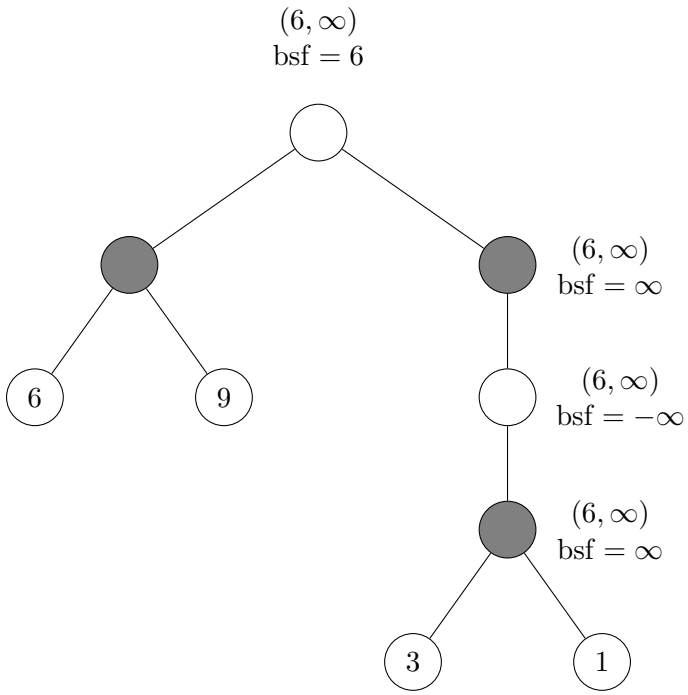
After evaluating the first terminal node, the `bestSoFar` of the left black node (lbn) is updated to 6. This changes the value of its β to 6, as the lbn is its own ancestor, and $6 < \infty$ (see the definition again). lbn's next child now inherits the (α, β) of its parent. Think about why this is always true: when we begin looking at a node, it has the same (α, β) values.[6]

If I really wanted to be precise with the definition, I could continue updating the values of (α, β) of the first terminal node searched. This has the potential to cause confusion, and the node in question is never searched again. Instead I will simply erase the (α, β) and `bestSoFar`information of nodes we are done searching—it is irrelevant.

[6]hint: a child and its parent share almost all of their ancestors.



We continue with our normal MinMax algorithm, first by updating the `bestSoFar` of the root to 6 This will cause its α to also update to 6, as the root node is its own ancestor. We move on to the next iteration, and recurse until we hit the node 3:



We now search bbn's first child, which has value 3. We update the `bbn`'s `bestSoFar` accordingly and notice that it is less than α.

Stop.

Let's think about what this means. Because bbn's α is 6, bbn must have some white ancestor with the option of a guaranteed 6 (this is the definition of α). We know bbn's `bestSoFar` is less than 6 and will only get smaller, so bbn *must be inferior to its ancestor's other option.* As I've said a thousand times, it doesn't matter *how* inferior, only that it is.

Another way to think about this is by asking yourself the question: *if* the value bbn propagates up the tree all the way to that (white) ancestor, will it matter if you searched the rest of bbn's children? No! You already know it's less than 6, so there's no way it can be better than the ancestor's alternative. In other words, we have identified a pruning opportunity. This solves our multi-generational problem.

Note that this would not work if bbn were white. Once we know that bbn's `bestSoFar` is less than α, we *need* to know that it can only get smaller in order to conclude it is inferior. There is an analogous situation where the `bestSoFar` of a white node exceeds β, in which case we know that a black ancestor has a guaranteed better option. Think about this.

The range of numbers between α and β is sometimes called the "window," as it is the window of values which are acceptable to both white and black ancestors.

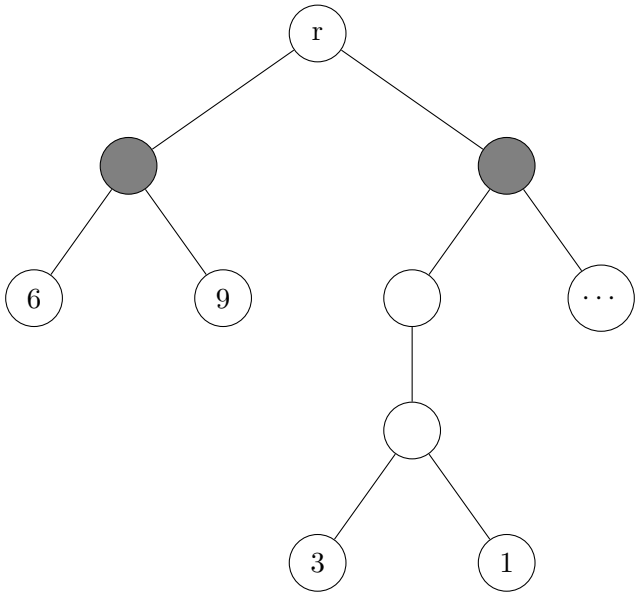
Nodes inherit (α, β) from their parent, then update them according to their `bestSoFar` values. This eliminates the need to actually (max/min)-imize over all ancestors—a parent's (α, β) values contains all relevant information.

Once everything slots into place, it isn't all that hard to code up:

```
def AlphaBeta(pos, depth, alpha, beta):
    if depth = 0:
        return evaluate(pos)
    if turn = White:
        bestSoFar = -infinity
        for move in pos.legal_moves:
            value = AlphaBeta(after_move(pos, move), depth - 1, alpha, beta)
            if value > bestSoFar: // to update bestSoFar
                bestSoFar = value
            if bestSoFar > beta: // pruning condition
                return bestSoFar
            alpha = max(alpha, bestSoFar) // update alpha
        return bestSoFar
    if turn = Black:
        bestSoFar = infinity
        for move in pos.legal_moves:
            value = AlphaBeta(after_move(pos, move), depth - 1, alpha, beta)
            if value < bestSoFar: // to update bestSoFar
                bestSoFar = value
            if bestSoFar < alpha: // pruning condition
                return bestSoFar
            beta = min(beta, bestSoFar) // update beta
        return bestSoFar
```

As always, make sure you understand the above code before you move on. I recommend re-doing some of the exercises using Alpha/Beta until it feels routine. This will help you understand the next section much more easily.

A Small Broccoli Floret. Trace the following tree using alpha/beta pruning.



Hint: you don't need to know what's in the · · ·—it could be terminal or it could have 1000 more generations attached to it.

5.4.3. NegaMax

You may want to re-read the section on implementing NegaMax for the 'old' system of pruning.

Just like before, NegaMax will be maximizing the negative of the opponent's perspective, and just as before, this negation messes with our pruning system.

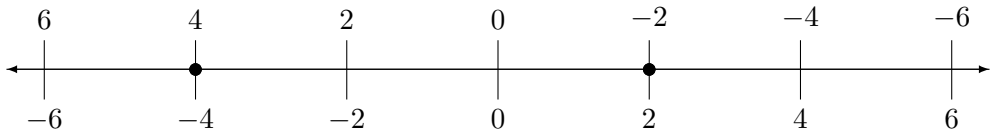
If we think about the algorithm intergenerationally,[7] alpha/beta can be summarized as follows:

- (white node) calculate `value` and `bestSoFar`. If `bestSoFar` $> \beta$, then prune. Otherwise, update alpha if appropriate.
- (black node) calculate `value` and `bestSoFar`. If `bestSoFar` $< \alpha$, prune. Otherwise, update beta if appropriate.
- (white node) calculate `value` and `bestSoFar`. If `bestSoFar` $> \beta$, prune. Otherwise, update alpha if appropriate.
- (black node) calculate `value` and `bestSoFar`. If `bestSoFar` $< \alpha$, prune. Otherwise, update beta if appropriate.

[7] that is, each bullet describes what happens in the next vertical level of the tree, ignoring what happens laterally

- ⋮ ⋮ ⋮

Let's see what this means for the train-tracks model where $\alpha = -4$ and $\beta = 2$ (using the bottom numbers).



- (white node) calculate `value` and `bestSoFar`. If `bestSoFar` is to the right of the dots, prune. If `bestSoFar` is between between the dots, move the left dot over to `bestSoFar`.
- (black node) calculate `value` and `bestSoFar`. If `bestSoFar` is to the left of the dots, prune. If `bestSoFar` is between between the dots, move the right dot over to `bestSoFar`.
- (white node) calculate `value` and `bestSoFar`. If `bestSoFar` is to the right of the dots, prune. If `bestSoFar` is between between the dots, move the left dot over to `bestSoFar`.
- ⋮ ⋮ ⋮

That is, white nodes check to see if the value is too far to the right, then possibly move α to the right. Black nodes check to see if the value is too far to the left, then possibly move β to the left. Notice how the two sets of instructions are "the same thing" only in opposite directions. Just as before, we can model this symmetry by multiplying by -1 and reversing inequalities:

- (white node) calculate `value` and `bestSoFar`. If `bestSoFar` $> \beta$, prune. If `bestSoFar` $> \alpha$, change α to `bestSoFar`.
- (black node) calculate `value` and `bestSoFar`. If `bestSoFar` $> -\alpha$, prune. If `bestSoFar` $> -\beta$, change $-\beta$ to `bestSoFar`.
- ⋮ ⋮ ⋮

The black nodes are doing *exactly* the same thing as the white nodes, only replacing α with $-\beta$ and β with $-\alpha$! This observation leads to some clean, compact code:

```
def NegaMax(pos, depth, alpha, beta):
    if depth = 0:
        return evaluate(pos)
    bestSoFar = -infinity
    for move in pos.legal_moves:
        // Note the order of -beta and -alpha
        value = -NegaMax(after_move(pos, move), depth - 1, -beta, -alpha)
        if value > bestSoFar: // update bestSoFar
            bestSoFar = value
        if bestSoFar > beta: // pruning condition
            return bestSoFar
        alpha = max(alpha, bestSoFar)
    return bestSoFar
```

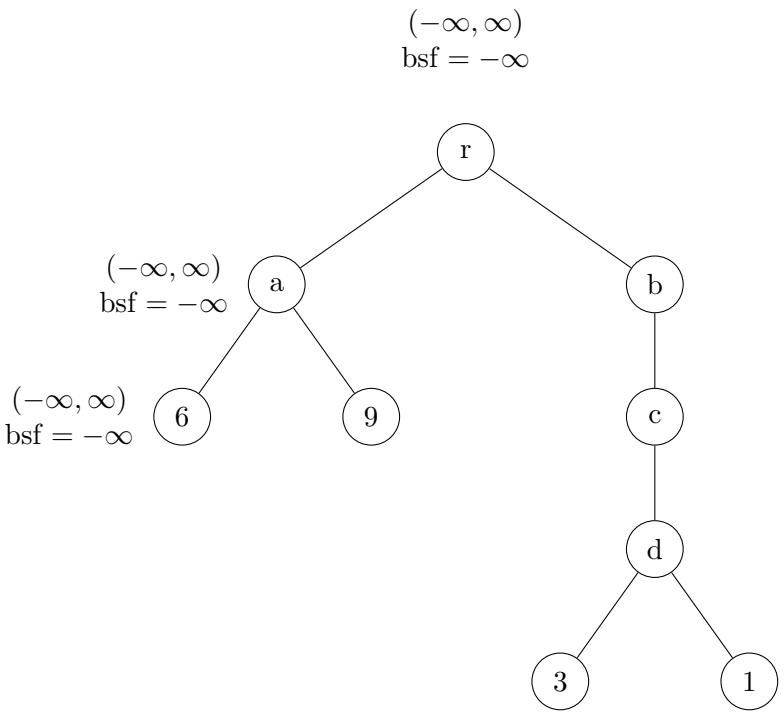
And that's it!

As always, convince yourself that this code does what I say it does, and work some examples.

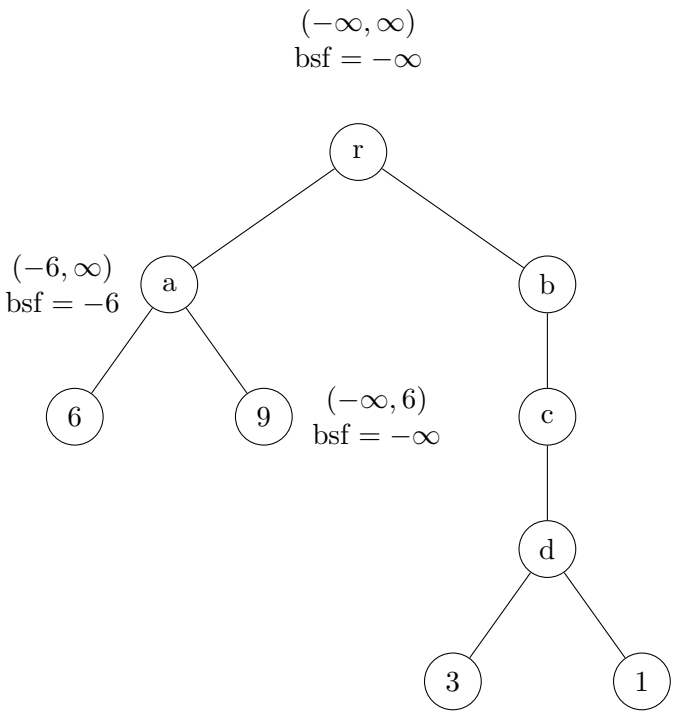
If you want to, you can interpret α as the maximum of the `bestSoFar`s of ancestors an even distance away, and β as the negative of the maximum of the `bestSoFar`s of ancestors an odd distance away, but this view is needlessly confusing. If you understand the non-NegaMax version of alpha/beta, then you can just view the NegaMax version as a more compact way of writing the old algorithm. Just know that the definitions of α, β aren't the same in the NegaMax context (as NegaMax is colorblind).

For the sake of completeness, I'll now trace through the tree at the beginning of the section using NegaMax (but you try it first!)

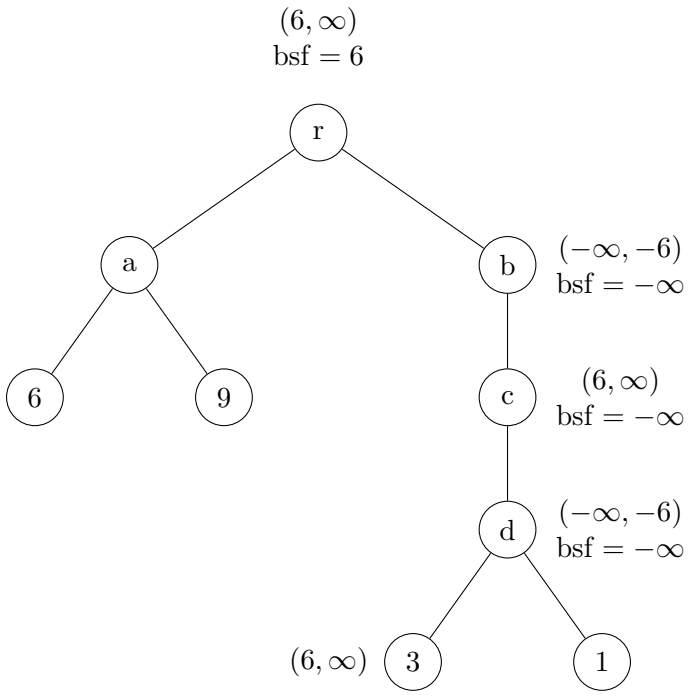
First, we recurse all the way down to the leftmost node. Note that at each stage, α and β are the negatives of the parent's β and α respectively.



We see that the leftmost terminal node has value 6. This causes $\texttt{bestSoFar}_a$ to be -6, (you negate each time you cross a generation). Because $\texttt{bestSoFar}_a > \alpha = -\infty$, we update α to -6, and proceed to a's next child. This node now has a window of $(-\infty, 6)$.



The node 9 is terminal, so (from the perspective of a) its value is -9. Clearly $-9 < -6 = \texttt{bestSoFar}_a$, so we don't update anything. We've now searched all of a's children, so we return $\texttt{bestSoFar}_a = -6$. From the perspective of `r`, a has value $-(-6) = 6 > -\infty$, so we update both $\texttt{bestSoFar}_a$ and α to be 6, and recurse all the way down to the node 3.

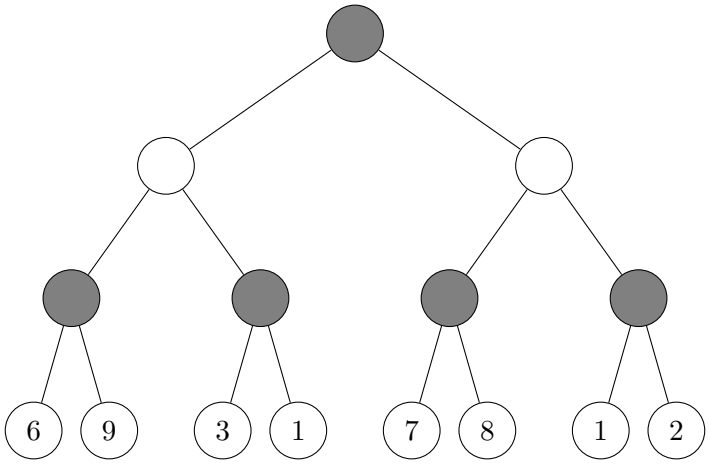


The node 3 is a terminal node, so we evaluate it, and pass it's value back to d, whereupon it is negated to -3. Because $-6 < -3$, we prune, immediately returning -3 up to c, which then negates it. c then passes the value 3 up to b, which negates it again. Finally, b passes the value -3 up to the root, which negates it a third time. So `r` sees that its second child has value 3, doesn't update anything (as $3 < 6$) and returns 6.

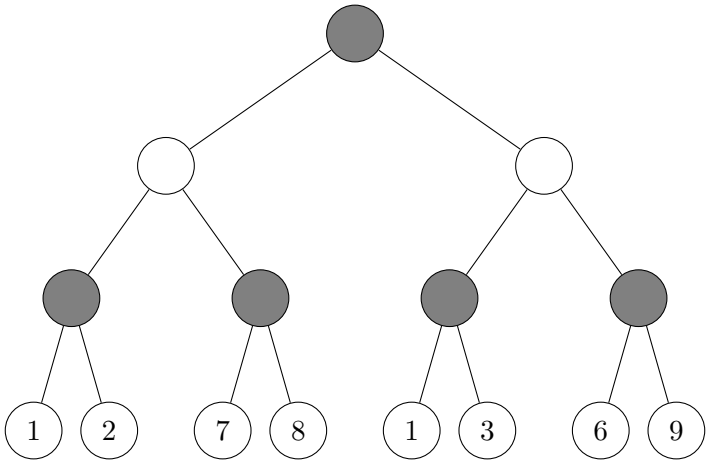
Compare these three trees with the ones we found earlier using non-NegaMax alpha/-beta. Notice how the white nodes are all labeled with the same (α, β) windows, but the black ones are flipped and negated. Why is that? Can you find more patterns? What if the root were black?

5.5. Move Ordering

Recall the first two exercises from section 5.3:



and



I will proceed assuming you have traced both these with alpha/beta (optionally with NegaMax)

These are 'the same' tree in the sense that they represent identical tree-games. The branches of the tree are shuffled, but the connections between the branches is the same. For instance, the sequence RLR (R=right,L=left) on the first tree involves the exact same decisions as the sequence LRR in the second tree even though the literal right/left move is different.

However, there's a huge difference in how much effort is required to search the two trees. Thanks to pruning, we only need to search 5 (!!) of the 8 terminal nodes in the first tree, while alpha/beta searches all 8 in the second.

When I explain this to people, they usually don't get as excited as I do about a savings of 3 nodes. They lack perspective. These diagrams are toy examples we use to conceptualize game trees. The actual ones are *massive*. Good engines can search *millions* of positions each second, with a depth far larger than 3. Pruning a single node on the third level now represents an *enormous* subtree that we don't have to search completely.

But why is one tree easier to search than the other? The answer is actually fairly straightforward: the first tree searches the best nodes first, which makes it much easier to prove subsequent subtrees inferior. Read that a few times to let that sink in. If it's still confusing, think about it in terms of my brother choosing the card pack: He calculates the value of the first pack to be 6, and can stop searching (prune) a pack whenever he discovers a card which is less than 6. If the first pack had value 100 instead of 6, he would be able to prune more often. This advantage would disappear if the value 100 pack were searched last. Thus, trees are easier to search when their best nodes are searched first.[8]

When the computer is playing a game of chess, the search tree isn't handed down from on high. The computer generates the legal moves in a particular position, and (here's the key part) *gets to decide* what order the moves are searched in. So ideally, our computer chooses to search the best move first. Of course, this is paradoxical: "Directions to find the best move: First search the best move..." however, there are fairly good half-solutions to this problem—using various heuristics, computers can guess what will and won't be a good move. There is a huge upside to doing this correctly, so I have dedicated an entire chapter (9) to it.

5.6. Classification of Nodes and Other Terminology

This section introduces some new terminology surrounding alpha/beta pruning, and re-caps old definitions.

5.6.1. Node Types

One of the more important set of definitions surrounding alpha/beta pruning is the classification of nodes. This is the language we will use to discuss pruning. There's some confusion online regarding two subtly different definitions of the following terms[9] Just know that these are the definitions I am using.

[8] This is usually but not always true. See exercise 5 in section 5.7

[9] Online, "all/cut/PV-node" sometimes means what I would call an "*expected* all/cut/PV-node," that is, a node you *expect* to be all/cut/PV. This concept is useful in some search techniques I won't be going into in this book.

Cut-Nodes

I have been using the word 'pruning' to describe the act of not searching a subtree. I like this word because it's evocative, but the term 'beta-cutoff'[10] (or just 'cutoff') is slightly more standard. This gives us the etymology of our first definition:

A **cut-node** is a node at which a cutoff is performed.

Cut-nodes are nodes which are "too good" for the side to move—they have an ancestor (of the opposite color) with a better option. In NegaMax, this happens when the value of the node exceeds β. Cut-nodes are sometimes called "type 2," "fail-high," or "beta" -nodes, but "cut-node" is most frequently used.

All-Nodes

If cut-nodes are "too good," then all-nodes aren't good enough.

An **all-node** is a node whose value does not exceed its initial value of α (in NegaMax).

This happens when it has an ancestor of the same color with a better option. It is called an all-node because it must search all its children to verify that its value is less than α. All-nodes are sometimes called "type 3," "fail-low," or "alpha" -nodes, but "all-node" seems to be the most common.

PV-nodes

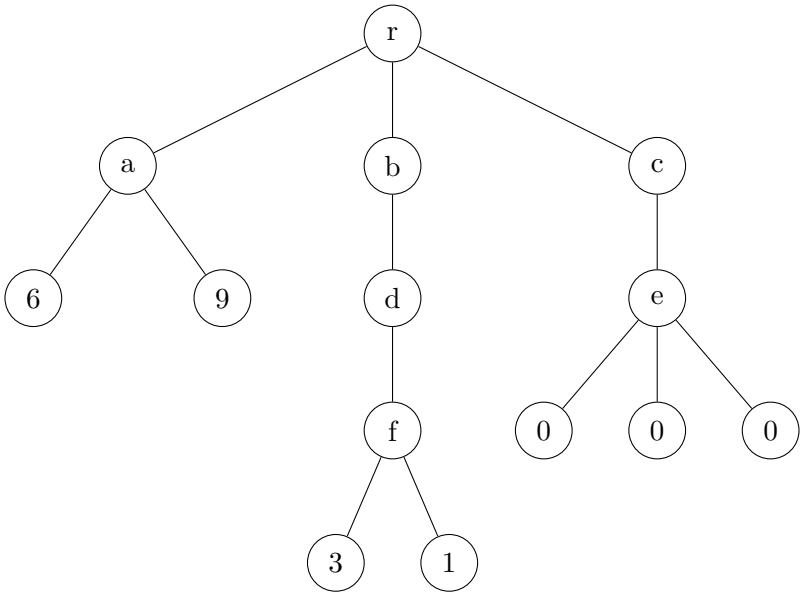
Cut and all-nodes classify the two ways a node can be inferior to an ancestor—either too good or not good enough. PV-nodes aren't deemed inferior by anyone:

A **PV-node** is a node whose value is inside the (α, β) window.

Being inside the (α, β) window guarantees a node is better than anything else that has been searched up to that point. PV stands for Principal Variation, so called because it's the best variation the computer has seen up to that point of search. PV-nodes are sometimes called "type 1" nodes.

As an example, consider the following tree:

[10] The beta refers to the fact that (in NegaMax) cutoffs are performed when the value of the node exceeds β



If you ignore the rightmost subtree, this is just the example we worked through before. In light of that, it should be very easy to trace—I've already done most of the work (do it!!). I'll now classify each non-terminal node in the tree.[11]

r is a PV-node. Its window is $(-\infty, \infty)$, so it's value must be inside it.

a is a PV-Node for the same reason as above.

b is a cut-node. It's value, -3, is larger than its β, which was -6.

d is an all-node. It's value, 3, is less than its α, which was 6.

f is a cut-node. It's value, -3, is larger than its β, which was -6.

c is an cut-node. It's value, 0, is larger than its $\beta = -6$.

e is an all-node. It's value, 0, is less than its $\alpha = 6$. Notice that *all* its children are searched.

5.6.2. Definitions

- **Branching Factor** (of a tree): the approximate number of children per node in a tree. Most of the diagrams I have been drawing have had a branching factor of 2.

[11] If you want, you can also classify terminal nodes, but we don't ever "do" anything other than evaluate them, so you almost never refer to terminal nodes as cut, all or PV -nodes.

Chess has a branching factor of around 30. It can be approximated as the number of nodes at one level divided by the number of nodes at the preceding level.

- (Effective) **Branching Factor**: the number of *searched* nodes at a particular level divided by the number of *searched* nodes at the preceding level. (usually this ratio is averaged over multiple levels.) Effective branching factor and regular branching factor coincide in MinMax.
- (Beta-)**Cutoff**: When the algorithm decides no to search a subtree
- **Ply**: one half-move in chess, or one step down the game tree.
- **Fail-high** (In NegaMax) A node fails-high when its value exceeds β in α/β pruning. e.g. "if a node fails high, it will trigger a cutoff." Note that this definition only works in NegaMax.
- **Fail-low** (In NegaMax) A node fails-low when its final value is less than α. Note that this definition only works in NegaMax.
- (alpha/beta) **Window**: The values between α and β. Also the values of a node which are "acceptable" by all the nodes ancestors.

5.7. More Exercises!

Unlike section 5.3, these exercise will only ask you to think about something. This will require much more effort than you might expect. Each of the following are true statements. Figure out why.

* indicates a more difficult problem.

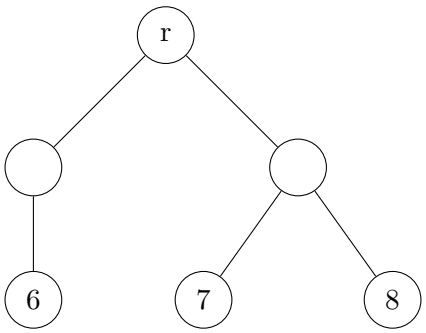
1. * The only way to understand alpha/beta pruning is to work exercises by hand.
2. * It's always the case that $\alpha \leq \beta$.
3. The line of code `bestSoFar > beta`: can be changed to `bestSoFar >= beta`: (changing $>$ to $\geq$) without affecting the pruning logic. (This is a minor improvement to the code as written in 5.4.3. Moving forward, I will use $\geq$.)
4. With the change in exercise 3, the (α, β) window is an open interval. (This gives the notation (α, β) more meaning.)
5. * Searching the best moves first doesn't guarantee you search the fewest nodes. (find a counterexample)
6. Every PV node causes the (α, β) window to decrease in size.
7. The root node and all left-most nodes are PV-nodes (if the root is called with $(\alpha, \beta) = (-\infty, \infty)$).

8. * For the following problems, assume we are dealing with a *perfectly ordered tree*, that is a tree where the best child is always searched first. (see the diagram on page 58 and generalize)
   a) The parent of an all-node is a cut-node.
   b) The child of a cut-node is an all-node.
   c) The child of an all-node is a cut-node.
   d) The children of PV-nodes are PV-nodes or cut-nodes.
   e) (summary of last 3 exercises) Cut-nodes appear as children of PV-nodes, and after that they alternate all, cut, all, cut...
   f) PV-nodes are pure-bloods. (all ancestors are also PV-nodes)
   g) The distance of a cut-node to its closest PV ancestor is odd
   h) The distance of an all-node to its closest PV ancestor is even.

5.8. An Important Fact about Failing High

I will mention up front that there's something called "search instability" which can make everything I say in this section false. However, we shouldn't run into this *too* much in this book, so don't worry about it.

Search the following tree with $(\alpha, \beta) = (-\infty, 5)$ (everything in this section uses NegaMax).



If you did it correctly, you will only search the leftmost branch, and obtain the value 6.

However, the true value of this tree is 7 ($\neq 6$). This shouldn't be too surprising: the mantra of the pruner is that once we find the value of the tree to be greater than β, we don't care by how much. Here, we discover `r` to be larger than 5, so alpha/beta says "good enough!" and returns 6. In this case, the true value is *larger* than the returned

value. This holds in general, and is the "important fact about failing high" mentioned in the title of this section. More precisely,

If `pos` is a cut-node, and x is the value of `AlphaBeta(pos, depth,` α`,` β`)`, then

$$x \leq \texttt{AlphaBeta(pos, depth, } -\infty\texttt{, } \infty\texttt{)}$$

This happens because cutoffs stop the search, possibly before `bestSoFar` (which can only increase) has reached the true value of `pos`.

A similar effect can be seen for slightly increased windows:

If `pos` is a cut-node and $\beta \leq \gamma$ then

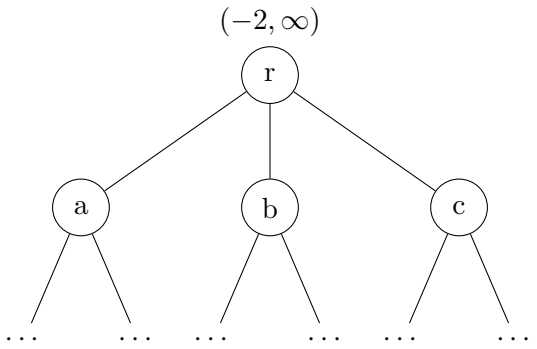
$$\texttt{AlphaBeta(pos, depth, } \alpha\texttt{, } \beta\texttt{)} \leq \texttt{AlphaBeta(pos, depth, } \alpha\texttt{, } \gamma\texttt{)}$$

.

I'll let you think that one through yourself.

We get this inequality because cut-nodes sometimes don't search all their children. This isn't the case for all-nodes, so you might expect there *isn't* an analogous inequality for failing low. In fact there is, but the reason for it is more subtle.[12]

Let x be the value of `AlphaBeta(r, depth, -2,` ∞`)`, where `r` is in the following tree.



Say $x = -4$, that is we fail low. Then `r` is an all-node, so we will search each of its children. `a` is searched with $(\alpha, \beta) = (-\infty, 2)$, and we know for sure it fails high, as its parent failed low. The same can be said for each of `r`'s children—they will all return values at least 4 (and one of them will return *exactly* 4).

However, using what we learned from the previous page, we know that the *true* value of each of `r`'s children might be larger, which in turn means the true value of `r` might be smaller than -4. This is the "dual" version of the important fact about failing high.

[12] We actually won't need this next part for the rest of the book (and is therefore skippable), but I think it's a nice exercise.

6. Mathematical Analysis of Alpha/Beta Pruning

Alpha/beta search has been the subject of hundreds of scholarly works throughout the years. Much of it has been the same stuff we're doing in this book—finding clever optimizations of established algorithms—but some of it has a much more theoretical bent. As a mathematician, I am more interested in theoretical bits. This book isn't about theory, but I'd betray my discipline to not include *something*.

To that end, I want to present two of my favorite proofs related to alpha/beta pruning, both of which appear in Donald Knuth and Ronald Moore's 1975 paper *An Analysis of Alpha-Beta Pruning*. Of course, this chapter is 100% skippable, but I hope you'll give it a read.

Prerequisites

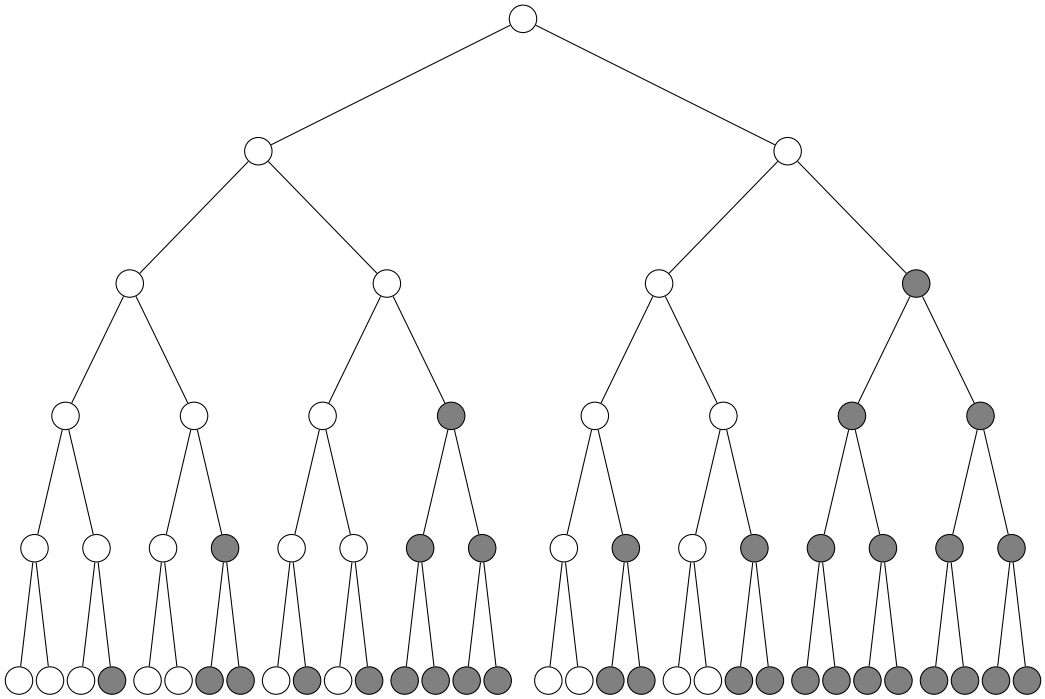
I will assume you have completed exercise 6 in section 5.7, reproduced below:

> Each of the following are true statements. Figure out why.
>
> For the following problems, assume we are dealing with a *perfectly ordered tree*, that is a tree where the best child is always searched first. (see the diagram on page 58 and generalize)
>
> 1. The parent of an all-node is a cut-node.
> 2. The child of a cut-node is an all-node.
> 3. The child of an all-node is a cut-node.
> 4. The children of PV-nodes are PV-nodes or cut-nodes.
> 5. (summary of last 3 exercises) Cut-nodes appear as children of PV-nodes, and after that they alternate all, cut, all, cut...

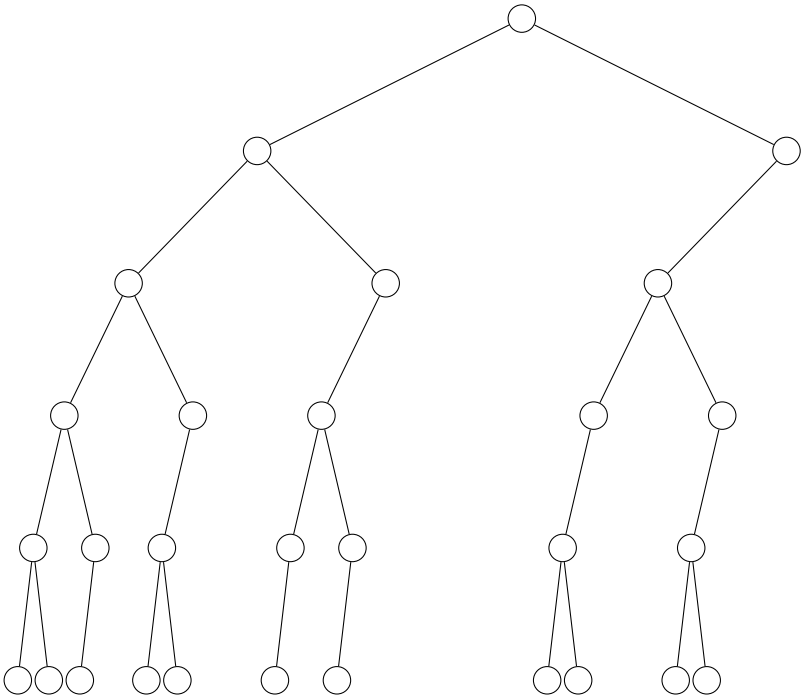
This exercise should be doable if you have a good understanding of alpha/beta pruning and *access to pen and paper*. If you cannot solve them[1] you should take it as an indication that you need to review alpha/beta pruning and/or (PV/cut/all)-node terminology.

To get a feel of the 'shape' of the tree, this is what a perfectly ordered tree looks like after shading in nodes that aren't examined by alpha/beta.

[1] I have failed you, I apologise



Below is the tree with the shaded nodes removed. I love this picture because you can really *see* the extent to which alpha/beta pruning cuts down on the density. This effect is more pronounced when the branching factor is larger.



6.1. Just How Good is Alpha/Beta Pruning?

This raises the natural question "What does 'good' mean?". The natural answer is to simply count the number of nodes searched. MiniMax is searches b^l nodes at level l of a tree with uniform branching factor. What is this number for alpha/beta pruning?

Because the effectiveness of alpha/beta pruning is highly dependent on the ordering of the tree, let's look at the best case scenario—a perfectly ordered uniform tree.[2] As it turn out, this is not as poor of an assumption as you may think—move ordering techniques (chapter 9) can consistently get near-perfect results. This has the practical benefit of being able to test a move ordering mechanism by how close it is to the theoretical best case. (Perhaps surprisingly,) The node count for Alpha/beta pruning admits a simple formula:

Theorem 1 (Nodes searched in best case)**.** *Let T be a perfectly ordered game tree of uniform branching factor b. At level l of this tree, alpha/beta pruning will examine exactly*

$$b^{\lfloor l/2 \rfloor} + b^{\lceil l/2 \rceil} - 1$$

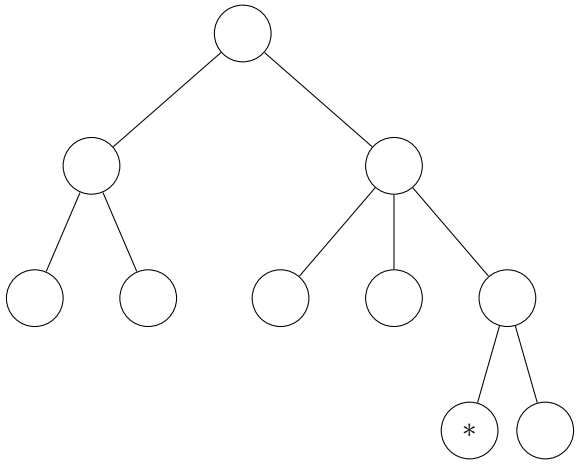
nodes, where $\lceil x \rceil$ and $\lfloor x \rfloor$ denote x rounded up and down respectively (called the "ceiling" and "floor" functions).

This was first discovered by Michael Levin in 1961, but apparently not proven. In an informal memo, he wrote "For a convincing personal proof using the new heuristic hand waving technique, see the author of this theorem."[3]

Proof. To prove the above theorem, we will establish a convenient notation for each position node on the game tree. It's fairly straightforward: a sequence of numbers $a_0, a_1, \ldots, a_n$ represents the node found by starting at the root, and choosing the a_0^{th} option, then the a_1^{th} option, etc. So the sequence $1, 1, 1, \ldots$ represents the leftmost branch (the PV). The sequence $2, 3, 1$ represents the node found by making the second move, then the third move, then the first move, as indicated in the diagram below.

[2] If you have done exercise 5 on page 59, you may object to this statement. However, exercise 5 is *not* true if we consider only uniform trees.

[3] <3 computer scientists.



We know from the discussion at the beginning of the chapter that in a perfectly ordered tree, all branches examined by alpha/beta pruning follow the same pattern: They start by going some distance down the PV, then alternating cut, all, cut, all... . Cut nodes always choose the first move in a perfectly ordered tree, so the corresponding sequence starts with 1s, then after some period of time, starts alternating between 1s and not 1s: $1, a_1, 1, a_3, \ldots$.

Then, in order to count the nodes examined by alpha/beta pruning at level l, we need only to count the number of l-length sequences which start with 1s, then eventually alternate between 1s and (not 1s).

I will separate these sequences into two groups. Group 1: sequences with 1s in all even positions. Group 2. Those with 1s in all odd positions. The stipulations from the previous paragraph mean that each sequence we want to count falls into one of the two groups. They overlap in only one sequence: $1, 1, \ldots, 1$.

A sequence in the first group of sequences looks like $1, a_1, 1, a_3, \ldots, a_l$. Each a_i has b options, and there are $\lfloor l/2 \rfloor$ number of a_i's (i odd), meaning group 1 contains $b^{\lfloor l/2 \rfloor}$ sequences. Similarly, there are $\lceil l/2 \rceil$ number of a_i's in $a_0, 1, a_2, 1, \ldots, a_l$, so group 2 contains $b^{\lceil l/2 \rceil}$. Because the two groups overlap in one sequence, we have to subtract one from the sum to find the answer: $b^{\lfloor l/2 \rfloor} + b^{\lceil l/2 \rceil} - 1$.

□

6.1.1. The odd/even effect

If you've taken a computer science course, you might squint at theorem 1 and say to yourself "ahh, so basically $b^{l/2}$". This is correct for a certain type of analysis, but it masks something interesting that only becomes apparent after plugging in values: (b = 10)

The ratio between the number of nodes at one level and the number of nodes at the next is not the same! It alternates between almost doubling, and almost increasing tenfold.

level	number of nodes
1	10
2	19
3	109
4	199
5	1099
6	1999
7	10999
8	19999

6.2. Is there Anything Better than Alpha/Beta Pruning?

This question demands more precision on two fronts. First, what does 'better' mean? Second, what exactly is 'anything'?

Starting with the second question, we want to consider algorithms which examine some subsets of the nodes of of a game tree, and outputs the value of its root. We will further stipulate that the algorithm be *provably correct*. When an algorithm 'examines' a node, it gains information about its value (if it is terminal) or its children (if not). An algorithm may only examine a node if it also examines its parent.

In general, it is difficult to compare the effectiveness of algorithms. In the case of alpha/beta pruning, the effectiveness is contingent on the ordering of the tree, which sheds some doubt on the idea that general algorithms can be compared *at all*. However, there is a general notion that if one algorithm examines fewer nodes than another, it performs better. As it turns out, this is all you need to formulate the idea that alpha/beta pruning is in fact the best way to search a game tree.[4]

Theorem 2 (α/β is optimal)**.** *If A is some algorithm which finds the (finite) value of a game tree, then there is some re-ordering of the tree so that the alpha/beta algorithm examines only (but not necessarily all) nodes examined by A.*

That is, if you give me an algorithm A, I can always beat it with alpha/beta pruning if you let me re-order the tree. Before we spend the next few pages on the proof, take a moment to appreciate the strength of this statement. Any algorithm you dream up, *anything* can be beaten by alpha/beta pruning.

Upon reading this theorem, you may wonder what the rest of this book is about if alpha/beta is the best there is. Of course, there is always the task of finding good move orderings, but beyond that, theorem 2 is too high-minded and theoretical. It is concerned with only algorithms which are *provably correct*, and we have no such requirements in the real world—very often chess programmers will intentionally sacrifice *correctness* for *strength*.

[4] To the best of my research, this was first proven by five Soviet mathematicians—Adelson-Velski, Arlazarov, Bitman, Zhivotovskii, and Uskov in 1970. The proof I am presenting is adapted from the aforementioned paper *An Analysis of Alpha-Beta Pruning* by Donald Knuth and Ronald Moore.

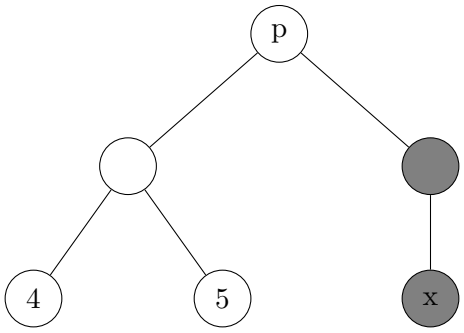
The proof will need some setup. Fix some algorithm A which searches the game tree.

Upper Bounds and Lower Bounds

Clearly, A must examine some subset of the game tree, but will likely not search all the nodes. Although A can figure out the exact value of the root, it may not be able to find the precise value of the other nodes in the tree by only searching the root. This should not be surprising—beta cutoffs do exactly this.

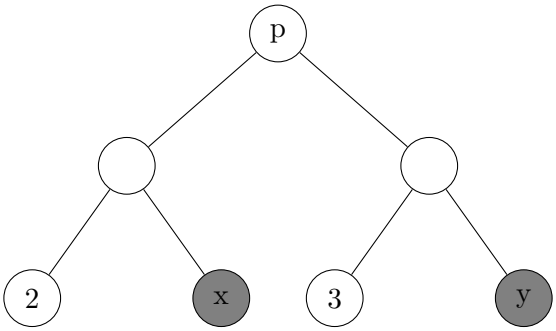
When A searches only the root, but also examines enough nodes to determine something else about the tree, I will say "A can prove that..." or "A can deduce that..." even though nominally A does nothing other than find the value of the root. This makes a sort of sense because in principal, A *could* do those other things without examining any additional nodes. A will only ever be called on the root of a tree, and when I say "A can determine that..." I mean implicitly that it can make this determination by only searching the root.

We will need some way to examine A's imprecise evaluations of non-root nodes. Let A_l be a function which takes as input any node of T, and outputs the *lowest* value that node can attain and still be consistent with the examined nodes. Take for instance the following tree: (note that we are using the negamax framework)



Say A does not examine the shaded nodes. Because A doesn't have a full picture of the tree, it can't tell exactly what the value of p is, but it's not completely blind! The left path guarantees a 4, so no matter the value of x (or indeed that entire subtree), the value of p will be at least 4. This is the smallest such guarantee, so $A_l(p) = 4$. You can think of $A_l(p)$ as A's pessimistic estimate of the value of p.

We will make a similar definition for upper bounds—A_u is a function which takes as input any node of T, and outputs the *largest* value that node can attain and still be consistent with the examined nodes. Consider the next tree:



No matter which branch is chosen at p, the opponent will always to choose a terminal node with value *at most* 3. Despite not knowing the complete tree, A can deduce the value is no larger than 3. This is the largest such guarantee, so $A_u(p) = 3$. You can think of A_u as A's optimistic estimate of the value of p.

We can now state some obvious facts with the new terminology: If p is unexamined, then the optimist will say it has value ∞ and the pessimist will say it has value $-\infty$. That is, $A_l(p) = -\infty$ and $A_u(p) = \infty$. It's also generally true that the pessimist will think every node is worse than it actually is, and optimist will think every node is better than it actually is. In other words, $A_l(p) \leq \text{value}(p) \leq A_u(p)$ for every node p. In fact, we can be even more precise about our characterization of $A_l(p)$ and $A_u(p)$. Let $p_1, \ldots, p_k$ be the children of p. Then

$$A_l(p) = \begin{cases} -\infty & \text{if } p \text{ is unexamined} \\ \text{value}(p) & \text{if } p \text{ is examined and terminal} \\ \max\{-A_u(p_1), \ldots, -A_u(p_k)\} & \text{if } p \text{ is examined and non-terminal} \end{cases} \tag{6.1}$$

$$A_u(p) = \begin{cases} \infty & \text{if } p \text{ is unexamined} \\ \text{value}(p) & \text{if } p \text{ is examined and terminal} \\ \max\{-A_l(p_1), \ldots, -A_l(p_k)\} & \text{if } p \text{ is examined and non-terminal.} \end{cases} \tag{6.2}$$

Intuitively, the pessimist will be extremely optimistic for their opponent—considering the best outcome for the opponent at each turn, then choosing the option which is least bad. Hence, $A_l(p) = \max\{-A_u(p_1), \ldots, -A_u(p_k)\}$ when p is examined and non-terminal.

By the same logic, the optimist will be very pessimistic for their opponent—considering the worst outcome for the opponent at each turn, then choosing the best option. Hence, $A_u(p) = \max\{-A_l(p_1), \ldots, -A_l(p_k)\}$ when p is examined and non-terminal.

I will spend the next page or so proving three lemmas that cover different situations for non-terminal, examined nodes, p. These will turn out to be loose analogs of PV/cut/all nodes in a more general setting.

Situation 1

Suppose A examines enough nodes to give an exact (finite) value for p.

This means $A_l(p) = A_u(p) = \text{value}(p)$. From 6.1 and 6.2, we know p has children p_k and p_j such that $A_l(p) = -A_u(p_j)$ and $A_u(p) = -A_l(p_k)$. Also from 6.2, we know $-A_l(p_k) \geq -A_l(p_j)$ (because $A_u(p) = -A_l(p_k)$ is the *maximum* of $A_l(x)$ over all children x of p). Multiplying by -1, we obtain $A_l(p_k) \leq A_l(p_j)$.

Combining the above facts, we can see that

$$A_l(p_k) \leq A_l(p_j) \leq A_u(p_j) = -A_l(p) = -A_u(p) = A_l(p_k)$$

The right and left side of this expression is the same, so everything in the middle must be equal (if $a \leq b \leq a$, then $a = b$). In particular, $A_l(p_j) = A_u(p_j)$, meaning A can *also deduce an exact value for* p_j. Furthermore, $\text{value}(p_j) = -\text{value}(p)$.

If you didn't follow all of that math, it's ok. It's all just a rigorous way of saying that if you can find the exact value of a node, you should be able to point to where that value came from—in this case p_j.

But what about the other children, p_i, of p? If p_i were unexamined, we would have $-A_l(p_i) = \infty$, contradicting the finiteness of $A_u(p)$. Then all the children of p are examined.

From equation 6.2, we know $-A_l(p_i) \leq -A_l(p_k)$ (again, because $-A_l(p_k) = A_u(p)$ is the maximum of the $-A_l(x)$'s), so $A_l(p_i) \geq A_l(p_k) = -\text{value}(p) = \text{value}(p_j)$. The inequality $A_l(p_i) \geq \text{value}(p_j)$ is just a rigorous way of saying that A must be able to prove that p_j is the best option—all other children are at least as good for the opponent.

Astute readers will note that the inequality $A_l(p_i) \geq -\text{value}(p)$ looks suspiciously similar to the inequality that triggers a beta cutoff. I will summarize this discussion in a lemma:

Lemma 1. *Suppose A can determine the exact value of some node p. Then*

1. *A examines all of p's children.*
2. *p has some child, p_j, with value equal to $-value(p)$.*
3. *A can determine the exact value of p_j.*
4. *A can prove that $A_l(p_i) \geq -value(p) = value(p_j)$ (a finite value) for all children p_i of p. That is, A can prove p_j is the best option.*

That's as complicated as it gets. I promise.

Situation 2

We will now turn our attention to nodes which can be bounded from below. Let p be an examined node, b be some (finite) number, and suppose $A_l(p) \geq b$.

Equation 6.1 tells us that p has some child p_i such that $A_l(p) = -A_u(p_i)$. Then, $A_u(p_i) = -A_l(p) \leq -b$. If p_i were unexamined, this would imply $\infty = A_u(p_i) \leq -b$, which is impossible. Thus, p_i must be examined. In summary:

Lemma 2. *Let b be some (finite) number and p be some examined node. Suppose A can determine that the value of p is bounded from below by b, that is $b \leq A_l(p)$. Then*

1. *A examines at least one of p's children.*
2. *A can determine that one such examined child is bounded from above by $-b$. If this child is p_i, we know $A_u(p_i) \leq -b$.*

Situation 3

Now we will look at nodes which can be bounded from above. Let a be some (finite) number, and p some examined node with $A_u(p) \leq a$.

If any child, p_i, of p is unexamined, then $A_l(p_i) = -\infty$, so $A_u(p) = \infty \leq a$, which is impossible. Then every child of p is examined. From equation 6.2, we know that for all children, p_i, of p we have $-A_l(p_i) \leq A_u(p) \leq a$. This means $-A_l(p_i) \leq a$ and therefore $A_l(p_i) \geq -a$. In summary:

Lemma 3. *Let a be some (finite) number and p be some examined node. Suppose A can prove that the value of p is bounded from above by a, that is $A_u(p) \leq a$. Then*

1. *All of p's children are examined.*
2. *A can determine that each child p_i of p is bounded from below by $-a$, that is $-a \leq A_l(p)$.*

Notice how the last part of each of these lemmas play off each other—if we start with a node with an exact value (say, the root) lemma 1 tells us that we can bound p's children from below. Then lemma 2 tells us that p grandchildren can be bounded from above, which lets us apply lemma 3 to see that p's great-grandchildren can be bounded from below, bringing us back to lemma 2, etc. Hopefully this reminds you of exercise 6 in section 5.7.

Proof of Theorem

We are finally ready for the proof of the main theorem, which I will now restate.

Theorem 2 (α/β is optimal)**.** *If A is some algorithm which finds the (finite) value of a game tree, then there is some re-ordering of the tree so that the alpha/beta algorithm examines only (but not necessarily all) nodes examined by A.*

In order to prove the theorem, we will need to exhibit an ordering of the tree so that the alpha/beta algorithm examines only nodes examined by A. This will be done by starting at the root, and then letting lemmas 1, 2 and 3 play off each other.

Proof. By assumption, A can find the exact (finite) value of the root. Then lemma 1 tells us that A can find the exact value of one of its children. We can apply lemma 1 to this child to see that A can find the exact value of one of *its* children (ie a grandchild of the root). We can apply lemma 1 repeatedly in this way to find a chain of children from the root to a terminal node, where A can find the exact value of each node in the chain.

The first step of our re-ordering will be to move this chain to the left of the tree, so that the leftmost branch contains only nodes that A can determine the exact value of. We will call this branch the principal variation, or PV.

Lemma 1 tells us much more than just the existence of a principal variation. It also guarantees that A can bound each child of a PV-node from below by the negative value of its parent. That is, if p is a PV node, and p_1 its child, then $A_l(p_1) \geq -\text{value}(p)$. This puts us in exactly the situation of lemma 2, which tells us that A can bound at least one (examined) child of p_1 (call it p_2) from above by value(p), that is, $A_u(p_2) \leq \text{value}(p)$. The next step of our re-ordering is to move p_2 to the left of all of its siblings.

Now that we know $A_u(p_2) \leq \text{value}(p)$, we are in the situation of lemma 3. From this, we know A examines all of p_2's children, and furthermore that A can bound any child of p_2 (say, p_3) from below by $-\text{value}(p)$, that is $-\text{value}(p) \leq A_l(p_3)$. There is no re-ordering at this step.

Now that we know $-\text{value}(p) \leq A_l(p_3)$, we are back at lemma 2. We will use this lemma to re-order a single child of p_3, as before, then use lemma 3 again. In this way, we keep on bouncing back and forth between lemmas 2 and 3, re-ordering at the child of every instance of lemma 2.

Take a step back and think about what we've done. The tree now looks suspiciously similar to the one described at the beginning of the chapter, with a PV on the left, and alternating types of nodes after deviating from the PV. With this in mind, what happens when we search the tree using alpha/beta pruning?

Because it gives the optimal value, the left side of the tree will be the usual principal variation, and whenever we arrive at a node re-ordered by lemma 2, we will find that its value exceeds the negative value of its most recent PV ancestor—or on other words, it will cause a beta cutoff (ie it is a cut-node) and alpha/beta will not examine any of its siblings. Alpha/beta will examine all children of the cut-nodes, but this is ok, because lemma 3 establishes they are examined by A.

Then, when searching the re-ordered tree, alpha/beta pruning examines *only* nodes examined by A, which proves the theorem.

□

Part III.

Optimization

7. The Evaluation Function Revisited

It's obvious that the evaluation function given in chapter 3 is inadequate. It's perhaps less obvious how to improve it. This chapter outlines some rough and ready heuristics that can be used to write an evaluation function.

Throughout this chapter, you will notice that I very rarely attach centipawn values to particular types of advantages/disadvantages. The reason for this is simple: I don't know the correct values. This part of chess programming is more art than science, and I am no artist. People have written lots on this topic, and to my knowledge there is no consensus—most hobbyists make it up as they go, then fine tune through observation and experimentation.

7.0.1. A Note on Efficiency

The implementation of the evaluation function is highly dependent on specific board representation, so I'm not going to provide any code, only describe what an evaluation function might do. It should be fairly straightforward to implement most of the ideas here.

That being said, I will give you one warning: The Evaluation function is called at *every* leaf node. **EVERY** leaf node! We can't tolerate excessive looping or costly computations. This part of the chess program deserves strict scrutiny—be on the lookout for unnecessary or inefficient code. In particular, be wary of needless loops—if you're searching every square on the chessboard *multiple times*, you're doing it wrong.

7.1. State of the Game

The main difficulty in writing a good evaluation function is that the value of certain aspects of a position are always changing and evolving based on the particularities of the game. In an attempt to describe this fluidity, we will classify positions based on certain recurring themes which affect various aspects of our evaluation.

7.1.1. Phase of the Game

Just as a story, every game of chess has a beginning, middle and an end.[1] Appropriately, these phases are called the "opening," "middlegame," and "endgame." In the opening, players develop their pieces and vie for control of the center. The middlegame sees players positioning their pieces and poking their opponent, trying to find a weakness. If

[1] always in that order.

the middlegame isn't decisive, the pieces will eventually trade down and we enter the endgame, where players try to promote their pawns and push for the win.

Opening vs Middlegame

"The opening is just a special case of the middlegame with the distinguishing feature that it is when players develop their pieces."

I forgot where I first heard this. I may have read it somewhere, or perhaps someone told it to me when I was a kid, but I can't seem to find a source for it. Regardless, this quote gives an interesting perspective most chess players don't have—the opening is a part of the middlegame, not separate. The idea here is that the *value* of various chess resources are more or less the same between the opening and the middlegame.

The evaluation function is only ever called *after a search*, while the "opening phase" usually lasts 8 or so moves (16 ply). Depending on how deep your engine searches, this means the evaluation function will only ever "see" an opening position a handful of times—the search horizon is constantly being pushed back.

For these reasons, It doesn't really make sense for the evaluation function to distinguish between the opening and the middlegame. As long as it values the development of pieces, control of the center, and king safety (which it should) the middlegame evaluation function can be used at the beginning of the game.

Middlegame vs Endgame

When I was a small aspiring chess player, my parents sent me to a chess camp ...I remember quite distinctly the coach saying

> "The Ghost of Bobby Fischer is never going to jump out at you
> and scream 'YOU ARE IN THE ENDGAME!!'"

I have no idea why I remember this particular encounter (and not anything else about him), but it's true. There's no clear dividing line between the middlegame and an endgame, but a line must be drawn,[2] and I put it at 30 pawns. That is, if you add up the value of each piece on the board and the number is less than 30, you're in the endgame.

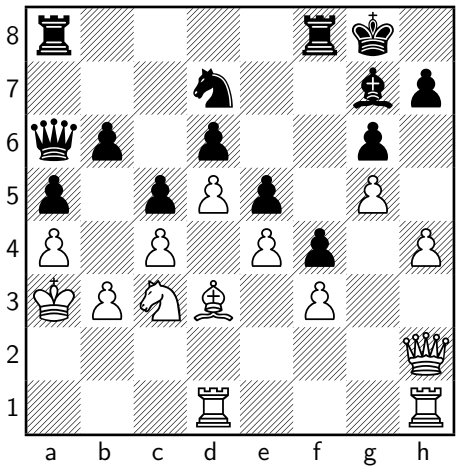
7.1.2. Open, Closed, Semi-Open

The second way we will classify positions regards the general mobility of all pieces on the board. In chessland, the terms 'open', 'closed', and 'semi-open' will sometimes refer to a particular class of *openings*, but here I use them to describe positions. Most chess players would still recognise this usage.

[2] Not always—there is something called a "tapered evaluation," which is a sort of a sliding scale between the middlegame and the endgame.

Closed

Closed positions are characterized by a locked pawn structure with most pawns still on the board.



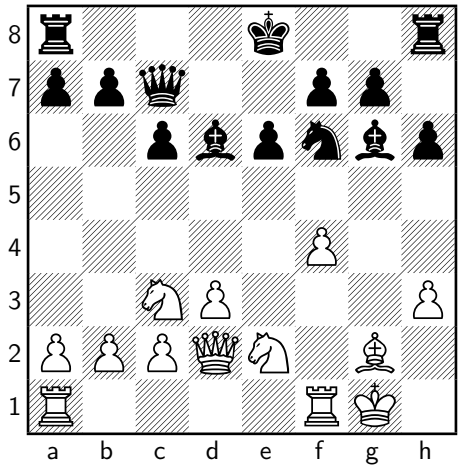
By "locked pawn structure," I mean situations where the pawns are in a diagonal chain where none of them can advance due to the opponent's pawns. Such pawns are said to be 'fixed'. Closed positions can be identified by looking at the four central squares: if 3 of them are occupied by fixed pawns, the position is likely closed.[3]

In general, closed positions require long-term strategy, which engines are typically very bad at. We usually think of closed positions favoring Knights (as opposed to Bishops) as their movement can't be blocked by pawns.

Open

Loosely, open positions are the opposite of closed positions—those where the few pawns still on the board are not locked.

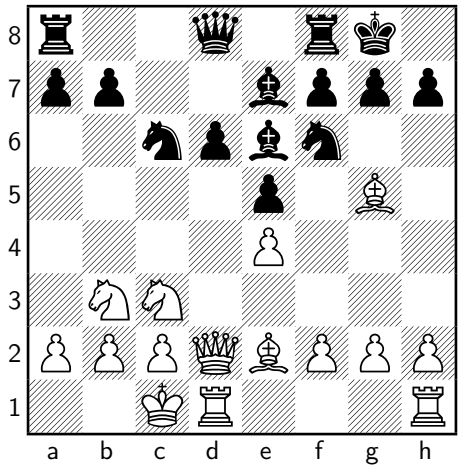
[3]Note that this doesn't *define* a closed position. It's just a hard and fast rule that works maybe 80% of the time. Same goes for everything else.



Open positions can again be identified by looking at the four central squares: if none of them contain fixed pawns, then the position is open. Open positions *usually* favor bishops over knights, as they generally have more mobility.

Semi-open

Loosely, a semi-open position is one which is neither open nor closed.



7.2. Material Considerations

By "material considerations," I mean information we can gain about the evaluation of a position solely from which pieces are on the board, and not where they are situated.

7.2.1. Relative Value of Pieces

I've had the good fortune to meet a few titled players[4] in my life. Whenever I get the chance, I always ask them about the relative value of pieces. This is how the conversation usually goes:

Me: Exactly how much is a bishop worth?
Master: Chess isn't that simple. Piece values are highly situational, and it doesn't make sense to assign them fixed values.
Me: Yes, I know, but perhaps you could give an average?
Master: Chess isn't that simple. Piece values are highly situational, and it doesn't make sense to assign them fixed values.
Me: Yes, yes, but if you had to.
Master: Chess isn't that simple. Piece values are highly situational, and it doesn't make sense to assign them fixed values.

(conversation continues like this with length roughly proportional to age of the master)

Master: Would this conversation end if I gave you a number?
Me: Yes.
Master: Very well. A knight is worth __ pawns, a Bishop __, a Rook __, and a Queen __. I'm leaving now.

And what answers do I get? Depends on who you ask! I've had one GM tell me bishops are 3.5 pawns, and another tell me 3—a huge difference. From my informal survey with extremely small sample sizes, my only real conclusion is that no one can agree what pieces are actually worth.

The Wikipedia page on the relative value of chess pieces bears this out: it contains a table of 24 ostensibly reputable sources which give the knight a value somewhere between 2.4 and 4.16 pawns, and the queen somewhere between 8.5 and 10.4 pawns. Below is a table of the mean/median of the values listed on Wikipedia: (switching to centipawns)

	Mean	Median
Pawn	100	100
Knight	326	315
Bishop	337	341
Rook	525	500
Queen	973	950

Given the spread of the values, I don't feel comfortable using these numbers for anything more than saying "Knights are worth slightly more than 3 pawns, and Bishops only slightly more than that. Rooks are a little over 5 pawns, and Queens are at little under

[4]Chess players with a 'title', for instance, "International master" or "Grandmaster."

10." and attaching large error bars around each evaluation. Be wary of the precision in the above table—it does not reflect accuracy.

7.2.2. Material Imbalances

A material imbalance (as distinct from a material advantage) refers to a situation where the material count is equal, but the pieces that make up this count on either side are different. For instance, two bishops and a knight vs queen is ostensibly equal, but the particularities of this situation typically favor the side with minor pieces. Looking at material imbalances is useful because the raw material counts often don't tell the entire story.

As always, these below statements are *huge* generalizations and should not be given too much weight. With the exception of the bishop pair, these considerations shouldn't be given any more weight than 15-20 or so centipawns.

The Bishop Pair

By far the most significant material imbalance is the bishop pair. Because they move diagonally, bishops can only cover half of the board by themselves, while two bishops together can exert their influence over the entire board. Essentially everyone—GMs *and* data miners—value the bishop pair at a half a pawn.

Bishop vs Knight

ahh the age-old question of BvN, along with the age-old answer: Knights are better in closed positions, bishops better in open ones. This is a rough and ready rule. The actual evaluation depends on many other factors, some of which we will discuss in the next section.

Queen vs 2 Rooks

In the endgame with open lines for both the rooks and the queen, two rooks will typically be able to coordinate in attack and defense better than a solitary queen. However, if you throw in some complications—a closed center, minor piece for each side, the rooks have difficulty working together and will succumb to the queen's mobility.

Number of Pawns

Sliding pieces (ie Bishops, Rooks, Queens) benefit from open files/diagonals. Pawns normally block off these lines, so sliding pieces typically gain value as pawns are traded. Knights on the other hand do not have their range limited by pawns, and have difficulty traversing the entire board, so they *lose* value as pawns come off the board.

If you are interested in an in-depth analysis of this topic, I will point you to GM Larry Kaufman's 1999 article[5] "The Evaluation of Material Imbalances," which gives an excellent in-depth, data-driven take on evaluating pieces. Disregarding its relevance to chess programming, I *highly* recommend it to all chess players.

7.3. Positional Considerations

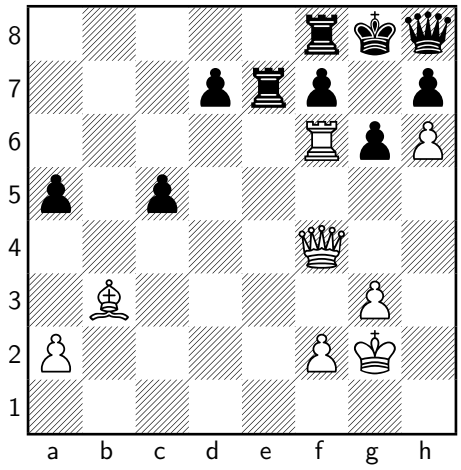
By "Positional Considerations," I mean information we can gain about the evaluation of a board position from where the pieces are placed on the board.

7.3.1. Mobility

As mentioned in chapter 1, most of so-called 'positional chess' revolves around the idea that

Pieces are most effective when they have the freedom to move around the board.

There's a fairly direct way to measure this: simply count the number of legal moves.



In chapter 1, I explained why this is is positionally dominant for white—Black's pieces are tied down while White's have free reign. This is reflected in the number of legal moves for each side—42 for white and only 18 for Black.

In theory, this is a decent thing to optimize for—playing moves to increase mobility is good. Only one problem: it's inefficient. The move generator requires loops upon loops with cascading if statements. It simply takes too long.

[5] https://www.danheisman.com/evaluation-of-material-imbalances.html

This is not to say that this kind of analysis couldn't ever work. Perhaps there are some clever way to (roughly) count the number of moves in an efficient manner[6]. But for now, we are left with indirect measures of positional chess.

7.3.2. Piece-Square Tables

Perhaps the easiest way to tell the computer where pieces *should* go are piece-square tables. The basic idea is that pieces get "bonuses" for being placed on good squares, and "penalties" if they are on bad ones.

We keep track of the bonuses and penalties in 'piece-square tables', where each entry of the table represents the bonus/penalty. Below is a piece-square table for a knight.[7] (For viewing purposes, it is laid out as if it were a chessboard, so a1 has a penalty of 24).

```
{
        {-20, -16, -12, -12, -12, -12, -16, -20},
        {-8,  -4,  0,   0,   0,   0,   -4,  -8 },
        {-12, 4,   12,  12,  12,  12,  4,   -12},
        {-12, 2,   6,   10,  10,  6,   2,   -12},
        {-12, 2,   6,   10,  10,  6,   2,   -12},
        {-6,  10,  8,   6,   6,   8,   2,   -6},
        {-16, -8,  0,   2,   2,   0,   -8,  -16},
        {-24, -50, -12, -12, -12, -12, -50, -24},
}
```

Here, we're incentivising moving a knight to the center, and disincentivising moving it to a corner. Also note that the knight's starting squares are disincentivised heavily. This is one of the mechanisms you can use to get the computer to develop its pieces.

Determining exactly what values you should place here is a bit of an art, but the basic idea is clear: disincentivise bad squares, (including starting squares), and encourage good ones.

However, what constitutes a "good" and "bad" square for a particular piece can change as the game progresses. Most notably, you really *do not* want your king out in the open in the middlegame—that's a fast way to loose. On the other hand, you *really do* want to activate your king in the endgame, when checkmate isn't a constant threat. The attacking value of the king in the endgame is often valued at around 4 pawns.

I've found the best solution to this problem is simply to have two separate piece-square tables—one for the middlegame and one for the end game. It's inelegant, but it works.

7.3.3. Pawns

Quite famously, Phillidor[8] said

[6] looking at you, bitboards

[7] for purposes of efficiency, I usually don't leave them as 2-dimensional arrays, and instead flatten them.

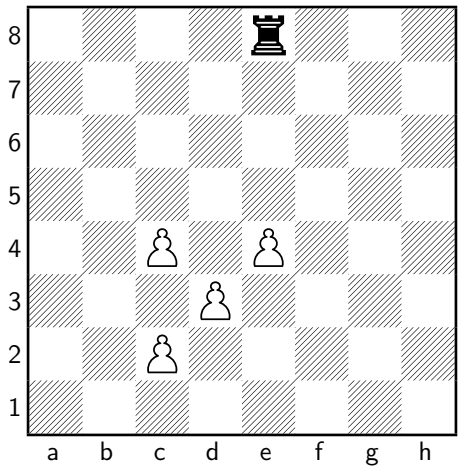
[8] The greatest chess player of the 16th century

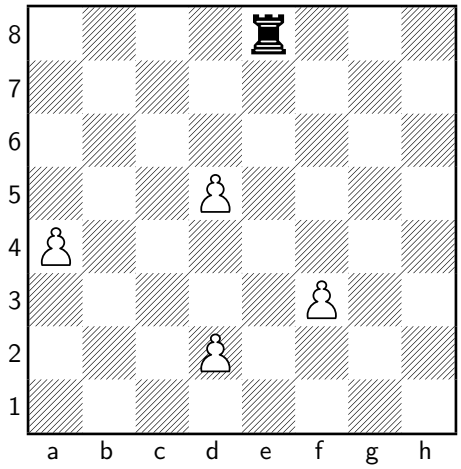
> ...to play the pawns well; they are the soul of chess: it is they which uniquely determine the attack and the defense, and on their good or bad arrangement depends entirely the winning or losing of the game.

Or more commonly, "Pawns are the soul of chess."

The grandiose language isn't entirely unwarranted. Pawns determine the "shape" of the board, and affect how the pieces interact with each other.

If pieces are best when they have the freedom to move around, pawns are best when they can defend each other. Consider the following two positions:





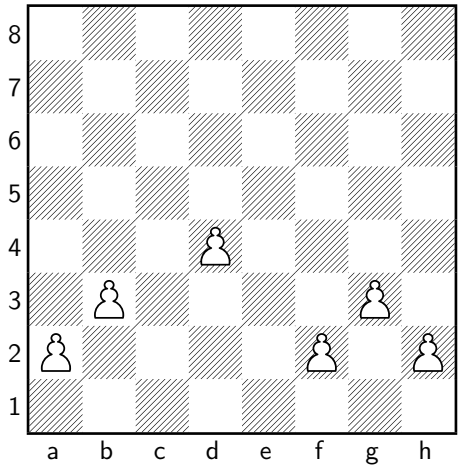
In the first, the black rook has only one weakness to attack: the pawn on c2. Every other pawn is defended. If white had more pieces on the board, it would be fairly simple

to defend c2, and therefore secure all the pawns. The opposite is true of the second position: black can take any of the pawns in any order, and if she had pieces on the board, white would struggle to defend all her pawns.

These are the basic ideas surrounding the following definitions:

Isolated Pawn

An isolated pawn is a pawn without a friendly pawn on an adjacent file.



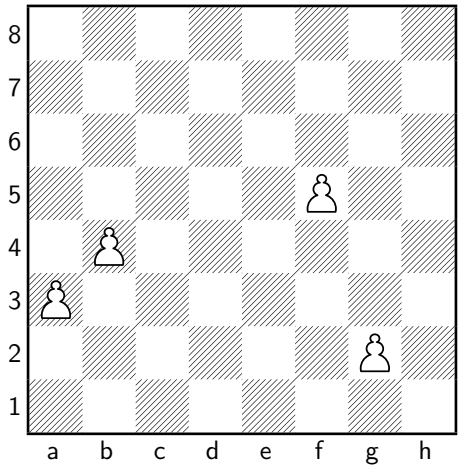
The d pawn is isolated.

These pawns cannot be protected by another pawn, and are usually considered a liability, especially if there isn't an opposing pawn on the same file.[9] The opposite of an isolated pawn is a

Connected Pawn,

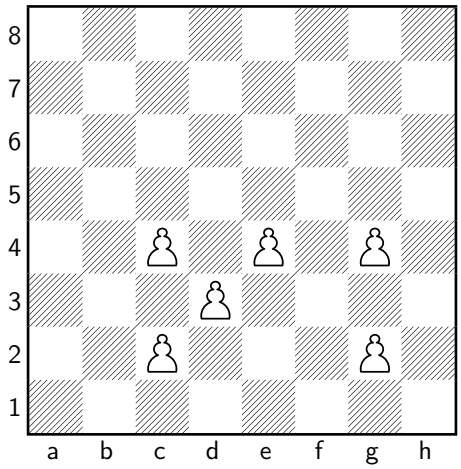
that is a pawn which is flanked by friendly pawns. Note that one doesn't have to be defending another to be considered connected—the a, b, f, and g pawns are all considered connected in the following position.

[9] This allows the opponent's rooks to put pressure on the pawn.



Doubled Pawns

Doubled pawns are friendly pawns which occupy the same file, so the c and g pawns are doubled in the following position.



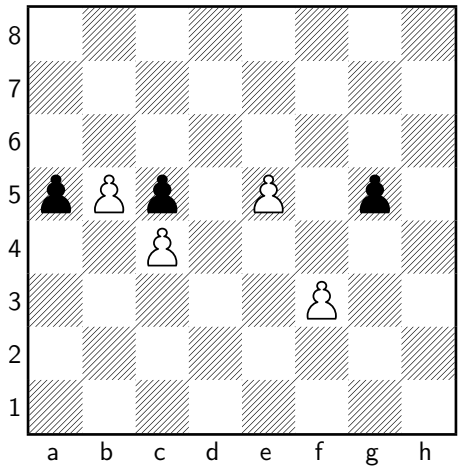
In general, doubled pawns are a liability, but there is some subtlety involved. They can sometimes offer additional support in the center, and usually open up files which can be used to good effect.

Passed Pawns

If you get a pawn to the other side of the board, it can be 'promoted', and turned into another piece (usually a queen). The most effective way for your opponent to stop this is with a pawn of their own. If your opponent can't do this, the pawn is said to be 'passed', and will likely cause some trouble in the future.

There is some subtly to this definition: pawns with opposing pawns on an adjacent file and in front are not considered passed—they have an opponent's pawn preventing

them from marching up the board. So, the a, b and e pawns are passed in the following position, but the f and g pawns are *not*.

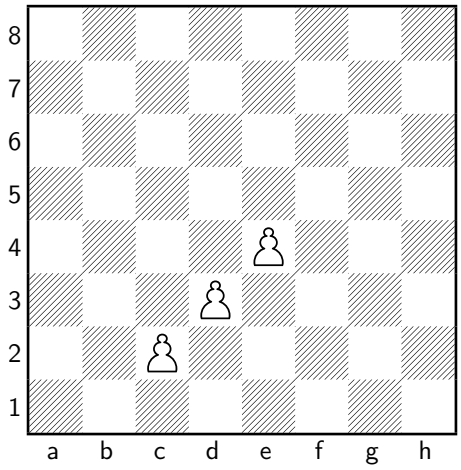


Multiple-Adjective Pawns

Of course, some pawns can satisfy many of these definitions at once, usually with compounding effects. I (and no one else) call these "multiple-adjective pawns." The most common examples are doubled isolated pawns (bad), and connected passed pawns (good).

Pawn Chains

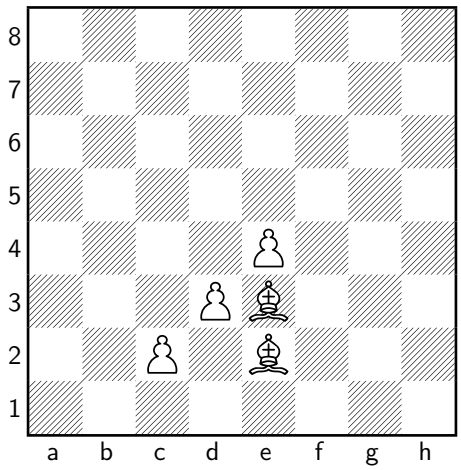
Because pawns love to defend each other, they naturally form these nice diagonal lines:



These lines are called "pawn chains," and must be unravelled starting at the base.

7.3.4. Good Bishop, Bad Bishop

If we take those naturally-occurring pawn chains, and add two bishops, something interesting happens:

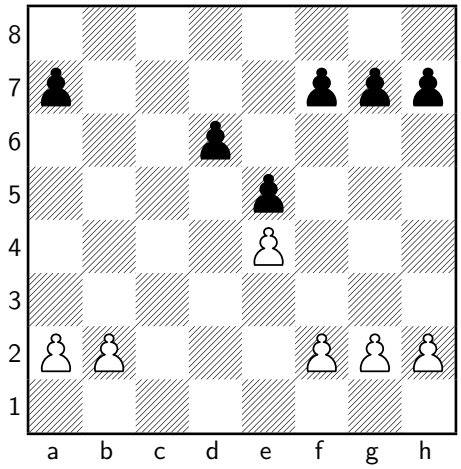


The two bishops—ostensibly equal in value—have very different prospects. The dark-squared bishop is unhindered by its friendly pawns (indeed, it can move as if they weren't there), while those same pawns severely stunt the mobility of the light-squared bishop. Since central pawns are usually static, this disparity between this bishops will likely be maintained throughout the game. Because of this, white has a real incentive to favor the dark-squared bishop over the light squared one.

This provides motivation for the following definition: If a bishop is on the same color squares as two of its central pawns, it is said to be "bad," and the other bishop is "good". Bad bishops are not always bad pieces. They are sometimes deployed outside the pawn chain where they make themselves useful. Similarly, good bishops are not always good pieces, but it's a rough heuristic you can use to determine the favorability of minor piece trades (among many other factors).

7.3.5. Rooks: Open Files, doubling up and the 7^{th} Rank

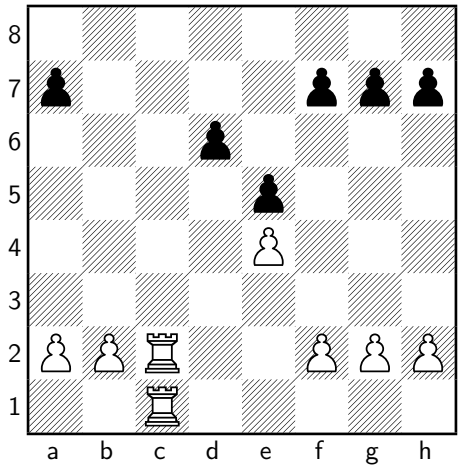
The positional considerations for rooks are entirely analogous to those of bishops: rooks want to be on files unobstructed by pawns. There are two primary ways this happens: on open and half-open files



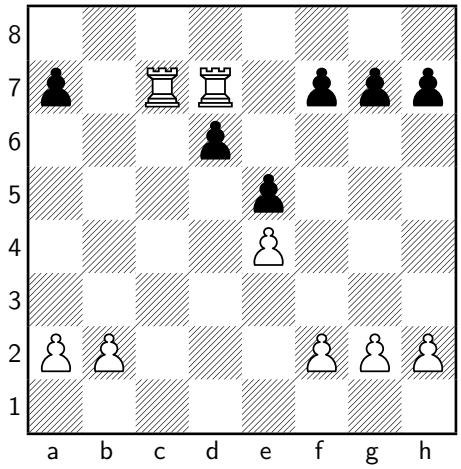
As you might expect, in this example, the c file is open, while the a,e,f,g,h files are closed—the c file has no pawns and a,e,f,g,h each have two pawns. Naturally, the rooks want to be on open files.

More interestingly, the b and d files are both *half*-open. However, this does *not* mean they should be treated the same. A white rook on b1 does nothing but stare at the back of a friendly pawn, while a white rook on d1 exerts its influence on most of the d file while putting pressure on the d6 pawn. Similarly, Black wants his rook on b8 instead of d8. Due to this asymmetry, we sometimes refer to the d file as half-open *for white*, and the b file as half-open *for black*.

Another important aspect of positional play with rooks is that they coordinate *very* well when connected aggressively, most commonly on (half-)open files and on the 7th rank.



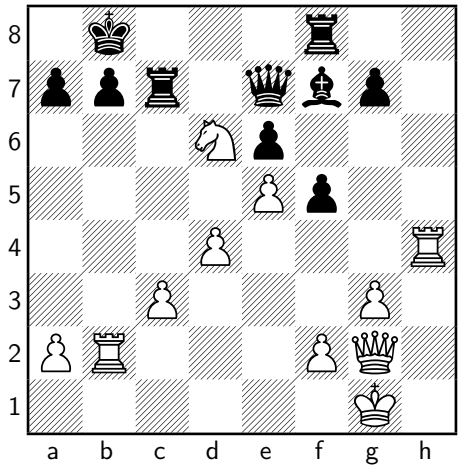
In this position, black has a very firm grip over the c file, and black has to tip-toe carefully around all the squares that white controls. For this reason, "doubling" rooks on an open file is very powerful.



Here, the white rooks are an absolute force of nature. White is putting pressure *everywhere*, and black will have a hard time defending. Because pawns (and sometimes pieces) usually hang out around on the 7^{th} rank (2^{nd}, if playing black), doubling rooks there is very powerful.

7.3.6. Knight Outposts

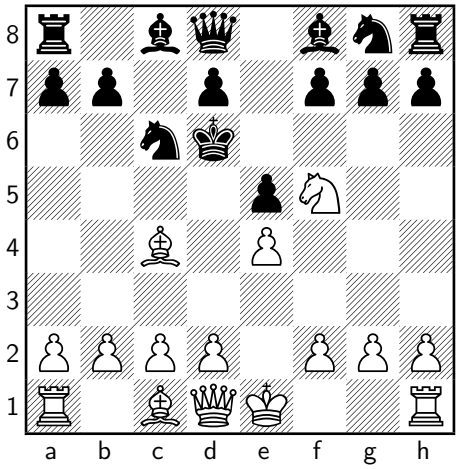
An outpost is a protected square in the opponent's camp which they can't easily attack with a pawn.



In the above position, d6 is a outpost for the knight. Since no black pawns are in position to attack d6, and the black bishop is on a light square (d6 is dark), black won't be able to remove the knight from its position. Outposts are generally most useful for knights, especially when they are in the opponent's face, typically strongest on the 6^{th} rank (for white, 3^{rd} for black).

7.3.7. King Safety

Consider the following position, and don't ask too many questions about how it came to be:



Black is in check and has two legal moves: Kc5 and Kc7. Which is better?

Even inexperienced players will be able to give you the correct answer for the correct reason: Kc7, because Kc5 makes the king *very* vulnerable to attack. But how does a computer come to this conclusion? Kc5 doesn't lead directly to any mate threats, nor does it seem to lose substantial material[10] So how do you do it?

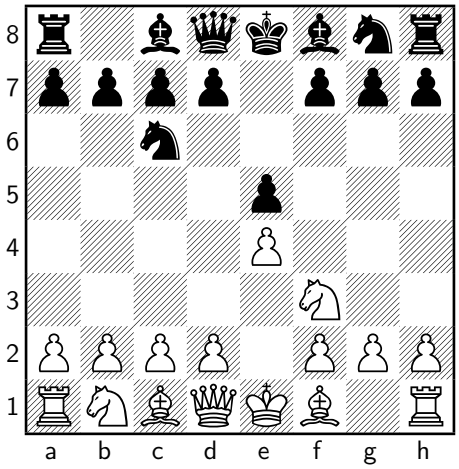
There are many ways to approach this problem. The simplest is to incentivize generic "safe squares" for kings in the middlegame while disincentivizing "unsafe" ones using piece-square table. This might solve our problem from before, but its a rather inflexible solution. It's easy to imagine situations where typical "safe" squares are in fact *very* unsafe.

One way of attacking this problem is to pay special attention to the number of attacked squares around the king. We might define a "king-zone" of one square in every direction from the king, and two steps towards the enemy.[11] Counting the number of times enemy pieces attack a square in the king-zone (possibly giving different weights to different pieces) gives you an idea of how vulnerable the king is.

[10] A naive engine with limited search capabilities might reasonably think this.
[11] in the previous position, the rectangle c4, e4, e7, c7)

8. Transposition Tables

There are four ways to arrive at the following board position on move 2:



White's first move can be either e4 or Nf3, and Black's can be either e5 or Nc6. Each of these four possibilities appear as distinct branches in the game tree, meaning vanilla alpha/beta pruning searches them all. This is redundant! If we already know what the value/bestmove is for a given position, there's no point in searching it again—we'll get the same answer.[1] Transposition tables are a way to store information about previously searched board positions for future use.

8.1. What Information Do You Remember?

Answer: anything you think is helpful, although there are some things that every transposition table should store:

- Score—you want to remember the result of calculations you've already done. (very important)
- Depth—Perhaps you are searching a position at depth 5 and remember a previous result at depth 3. You can't completely trust the depth 3 calculation (it's less accurate) but you can still use it for some things.

[1] This is a slight lie. Same advanced search techniques can cause "search instability," but for almost everything in this book that won't be an issue.

- Node type—this plays in to how previous results can be used.
- "best" move—Sometimes (chapter 9) it will be useful to know the best move found from that position. Although this isn't exactly what the computer thinks is best here (beta cutoffs prevent from searching everything), it's still useful to record this information.

8.2. How Do You Use That Information

"So if I ever encounter a position I've already searched and the depth is agreeable,[2] I just return the value I found on the first search right?"

No. Wrong. Bad.

Beta cutoffs can throw a monkey wrench in this process. Say you search some position, `p`, and it fails high, returning the value v. If, elsewhere in the search, you encounter `p` again, *you can't immediately conclude* the value of `p` is v. The Important Fact About Failing High (section 5.8) tells us that the "true" value of `p` is $\geq v$, or in other words v is a lower bound on the "true" value.

Upon the second search of `p`, we can do two things with the information that `p` failed high, and returned v. First, if $\alpha < v$, we can update $\alpha = v$, as v is an improved lower bound on the true value of `p`. Intuitively, the previous time the algorithm searched `p`, it discovered something at least as good as v. Second, if $\beta \leq v$, we know for sure the "true" value is larger than β, so we can *completely skip the move search* and preform a beta cutoff.

In code, this all looks like

```
def Search_with_TT(pos, depth, alpha, beta):
    if in_transposition_table(pos):
        tt_entry = TT[pos]
        if tt_entry.depth >= depth: //does it give accurate information
            v = tt_entry.value
            if tt_entry.node_type == CUT:
                if beta <= v:
                    return v
                if alpha < v:
                    alpha = v
```

which goes *before* the big for loop searching all the moves in a `pos`.

The other side of this coin is the case where `p` is an ALL node. In this case, we use the second part of The Important Fact About Failing High in the same way as before, only flipping the inequality signs and swapping α and β.

[2] that is, the depth of the second search is less than or equal to the depth of the TT search.

```
    if tt_entry.node_type == ALL:
            if v <= alpha:
                return v
            if v < beta:
                beta = v
```

Of course, if `p` is a PV node, its value is exact, so we don't have to do anything special:

```
    if tt_entry.node_type == PV:
        return v
```

Putting all this together, the transposition table access will look something like this:

```
def Search_with_TT(pos, depth, alpha, beta):
    if in_transposition_table(pos):
        tt_entry = TT[pos]
        if tt_entry.depth >= depth: //does it give accurate information
            v = tt_entry.value
            if tt_entry.node_type == CUT:
                if beta <= v:
                    return v
                if alpha < v:
                    alpha = v
            if tt_entry.node_type == ALL:
                if v <= alpha:
                    return v
                if v < beta:
                    beta = v
            if tt_entry.node_type == PV:
                return v
```

8.3. How do you Store Information: Zobrist Hashing

This section describes Zobrist Hashing, the primary way computers store information about chess positions.

8.3.1. Hash Functions

You are standing in an empty, infinite library with rows and rows of empty shelves and have a massive pile of books you need to organize for efficient retrieval. Here's what you do: pick up a book and pick a random shelf. Write down the book title and shelf number on a piece of paper and place the book on that shelf. Repeat $\times 10,000$. Now

when someone asks for a particular book, you look through your big list, find the entry, and retrieve the book from the corresponding shelf.

This system works, but it's slow. Every time someone asks for a book, you have to go through this big list. It would be so much easier if, given a book title, you could just figure out the shelf number directly.

The next time you're trapped in an empty, infinite library, you decide on a better sorting scheme: give each word in the English language a number by multiplying together the positions of each letter in the alphabet. So "CHESS" is $3 \cdot 8 \cdot 5 \cdot 19 \cdot 19 = 43,320$ and "ALGORITHMS" is $1 \cdot 12 \cdot 7 \cdot 15 \cdot 18 \cdot 9 \cdot 20 \cdot 8 \cdot 13 \cdot 19 = 8,066,822,400$. To find the shelf number of a book, add together the numbers associated to each word in the title, so "CHESS ALGORITHMS" is placed in shelf number

$$3 \cdot 8 \cdot 5 \cdot 19 \cdot 19 + 1 \cdot 12 \cdot 7 \cdot 15 \cdot 18 \cdot 9 \cdot 20 \cdot 8 \cdot 13 \cdot 19 = 8,066,865,720$$

Now when someone asks for a book, you simply compute the shelf number instead of painfully going through a list.

The ladder technique for storing values (appropriately digitized) is known a 'hash table' in computer science. The general strategy is to create a *hash function* which tells us where every object should be stored in memory.

Not all hash functions are perfect. For instance, the infinite library hash function doesn't play nice with anagrams: books titled "CHESS ALGORITHMS", "HCESS LAGORITHMS" or "ALGORITHMS CHESS" would all be sent to the same shelf. These are called *collisions* and may or may not be a problem depending on how large your shelves are, and whether you are willing to manually go through each book on the shelf to find the book you are searching for.

We want to create a hash function which sorts chess positions into memory for easy retrieval. For our purposes, three things that make a good hash function:

1. It is easy to compute
2. It is collision resistant
3. Similar positions (books) do not produce similar hashes (shelf numbers)

Hopefully by now you can appreciate why 1 is important in the context of chess programming. Although some level of collisions are unavoidable,[3] they give inaccurate or incomplete information and therefore need to be avoided. Furthermore, each entry in the hash/transposition table will come from the same chess game. Then we need to take care to ensure that similar board positions are not more likely to collide.

[3] By the "pigeonhole principle": the number of possible chess positions is far, far larger than the number of spaces in memory

8.3.2. Zobrist Hashing

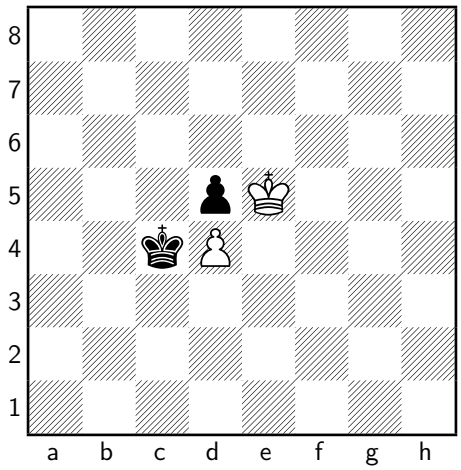
In this section, I will construct a hash function that satisfies all three criteria.

Step 1 is to generate a whole bunch of random numbers. Specifically, we will generate a random number (which I will call a "Zobrist numbers") for each piece at each board position.[4]. So somewhere in memory, the computer knows

$$\begin{aligned}
\text{white rook at a1} &= 9836436391762152168 \\
\text{white rook at a2} &= 8860853227890611409 \\
&\vdots \\
\text{black rook at a1} &= 9444523028451467091 \\
\text{black rook at a2} &= 16389479467785772769 \\
&\vdots
\end{aligned}$$

It's important that these numbers be big—64 bits if possible—but we'll get to that later.

Step 2: To find the Zobrist key of a given board position, XOR together[5] every Zobrist number which appears in the position. For instance, the Zobrist key of



is simply

(white pawn at d4) XOR (black pawn at d5) XOR (white king at e5) XOR (black king at c4)

More generally, the algorithm looks like this:

[4] You can (and should) add information about castling rights, en passant, whose turn, etc. but I just want to indicate the idea here

[5] to XOR two numbers a, b together, write them both in binary, perform an XOR on each position separately. So 5 XOR 6 = 101 XOR 110 = 011 = 3.

```
def zobrist_key(position):
    x = 0
    for piece in position:
        x ^= zobrist_number(piece, square)
    return x
```

Where ^ is the XOR operation.

8.3.3. Is Zobrist Hashing any Good?

Recall we want three things out of a hash function:

1. It is easy to compute
2. It is collision resistant
3. Similar objects do not produce similar keys

Easy to compute?

You might think a simple `for` loop is quite easy to compute, but remember we are calling this function on *every* node in the tree. We need this function to be *fast*, and an extra `for` loop won't cut it. Fortunately, there is an easy modification of the above algorithm which is lightning fast. Imagine you already have a board with a known Zobrist key, and you place a white rook on a1. Assuming a1 was empty beforehand, how does the Zobrist key change?

(please think about it...)

Because a XOR b = b XOR a, we can move the (white rook at a1) term in the expression for the Zobrist hash all the way to the back to see that

(new Zobrist key) = (old Zobrist key) XOR (white rook at a1)

Because a XOR $a = 0$, *removing* a white rook from a1 is preformed similarly:

(new Zobrist key) = (old Zobrist key) XOR (white rook at a1)

Here's the best part: Because every position searched in the game tree is the child of another searched position, we can always use the Zobrist key of the previous node to compute the Zobrist key of the current node. If the rook was moving from a1 to a2, we first remove the a1 rook from the Zobrist key, then add an a2 rook to the Zobrist key, that is

(new Zobrist key) = (old Zobrist key) XOR (white rook at a1) XOR (white rook at a2)

If the rook captures a piece at the target square, we must also remove that piece via an XOR.

This process lets us compute the Zobrist key of a position in just a few bit operations, which is about as fast as it gets for computers.

Are Zobrist keys collision resistant?

Short answer: Yes if you have 64 bit keys, no if you have 32 bit keys. Long answer: math!

We would like to analyze the collision resistance of Zobrist hashing. More precisely, given two distinct (randomly chosen) positions, what is the probability that they have the same Zobrist hash? Because each Zobrist number was chosen uniformly at random, the Zobrist key to also distributed uniformly.[6] If N is the number of possible Zobrist numbers (in the 64 bit case, $N = 2^{64}$), there are N different way to have matching Zobrist keys, and N^2 possible pairings of Zobrist keys. Then the probability of having a matching Zobrist key is $1/N$.

But this doesn't really tell us what we want to know. In practice, we will have millions of different positions, and we want to analyze the collision resistance *in aggregate*: if we store $m = 100,000,000$ positions, what is the *expected* number of positions which collide with another? This will take slightly more careful analysis.

The basic strategy will be to first calculate the probability that a particular position collides with something, then multiply by the number positions under consideration. You might object that the collision pattern of positions are not independent events. This is true: if I hash three positions A, B, C, it's unlikely that any will collide, but if I tell you A collides with something, the probability that B collides with something goes way up. But this doesn't matter! *Even if you have dependent quantities*, the expected value of the sum is simply the sum of the expected value.[7]

Let A be some position. We're looking for the probability that A collides with something. This is the same as one minus the probability that everything else does not collide with A. From the first paragraph of this subsubsection, we know that the probability another random position does not collide with A is $1 - 1/N$. Then the probability that *every* other position does not collide with A is $(1 - 1/N)^{m-1}$, which means the probability that A collides with *something* is $1 - (1 - 1/N)^{m-1}$. From the previous discussion,

[6] In a strict sense, this requires justification, but it should be intuitive.

[7] This is known as "linearity of expectation". The precise way it is being used here is as follows: Let X_i be a random variable which takes the value 1 whenever the i^{th} position collides with another, and 0 otherwise. Then the total number of positions which collide with another is $X_1 + X_2 + \cdots + X_m$, and we are searching for the expected value:

$$\mathbb{E}(X_1 + \cdots + X_m) = \mathbb{E}(X_1) + \cdots + \mathbb{E}(X_m) = m \cdot \mathbb{E}(X_1)$$

Finally, note that $\mathbb{E}(X_i)$ is simply the probability that the i^{th} position collides with something.

this means the expected number of positions which collide with another is

$$m \cdot \left(1 - \left(1 - \frac{1}{N}\right)^{m-1}\right)$$

Using this equation and setting $m = 100,000,000$ and $N = 2^{32}$, we can expect $2,301,410$ colliding positions, while $N = 2^{64}$ gives only $0.0005\ldots$.

We do not have a library with 2^{64} shelves

$2^{64} = 18,446,744,073,709,551,616$ is a (very) big number and unfortunately computers don't have enough shelf space to accommodate our 64-bit needs. Solution? Allow multiple positions to occupy the same entry in the transposition table, but tag them with their Zobrist key. This is performed as follows: Let $N < 64$ be some number small enough that 2^N is an acceptable number of shelves (say $N = 17$ or so). Then the shelf number of a position with Zobrist key k will be simply the first N bits of k. From our previous discussion, this process is all but guaranteed to put multiple books on the same shelf. If N is chosen large enough, the number of books per shelf will be small, making it feasible to search them all until a matching Zobrist key is found. If this number is still too large, some programs artificially limit the number of books per shelf.

In summary, a position is stored in the transposition table by first calculating its Zobrist key k, then placing it in the shelf corresponding to the first N bits of that key. Each entry is also tagged with its Zobrist key. In order to retrieve an entry with a Zobrist key k, it will first find the shelf corresponding to the first N bits of k, then search every entry of that shelf until it finds a matching Zobrist key.

Do similar objects produce similar keys?

No, they don't. Similar but distinct board positions must differ in at least one piece of information. This means one of the positions has a Zobrist number in it's XOR decomposition which is *not* included in the XOR decomposition of the other position. Because this Zobrist number was chosen at random, the two Zobrist keys of the given positions will be randomly different.

An Interesting Mathematical Digression

Zobrist hashing (as with everything in life) is just linear algebra.

Let $\mathbb{F}_2$ be the field of two elements, and $V = \mathbb{F}_2^{64}$. Then every vector in V is a 64 bit number, and addition of two vectors is simply the XOR operation. With this language, the Zobrist hash of a position is a linear combination of the constituent Zobrist numbers. A collision, therefore, is a linear dependence of Zobrist numbers.

Not all sets of Zobrist numbers are created equal—some may be more prone to collisions than others.[8] A set of Zobrist numbers which span a proper subspace of $\mathbb{F}_2^{64}$ will have "less space" to avoid collisions than one spanning the full $\mathbb{F}_2^{64}$.

Pure dimension considerations aren't enough—consider a Zobrist system where white rooks at square i have Zobrist number $e_i = (0, \ldots, 0, 1, 0, \ldots, 0)$, (where the 1 is in the i^{th} position) and every other piece/square combination has Zobrist number 0. Then every position without white rooks will have the same hash (ie *lots* of collisions), even though the Zobrist numbers span all of $\mathbb{F}_2^{64}$.

The "best" set of Zobrist numbers Z ($|Z| \approx 800$) will somehow minimize the probability that small (< 32) subsets $T \subset Z$ will be linearly independent. I am unsure if this is the best mathematical formulation, but the question presents itself: what is the best set of Zobrist numbers? To my (admittedly, meager) knowledge, there is no known answer.

I mentioned this problem to a real-life combinatorialist, and he mentioned the words "binary matroid". After a bit of googling, I now believe that the mathematical machinery exists to solve this kind of problem. If some grad student in combinatorics wants to take a stab at it...

[8] Choosing randomly is basically good enough. This section is for the mathematicians.

9. Move Ordering

As mentioned several times in chapters 5 and 6, the order in which moves are searched plays a pivotal role in the efficacy of alpha/beta pruning. This chapter outlines the primary techniques used to sort moves.

9.1. How to Think About Move Ordering

The purpose of move ordering is not to find the best moves and search them first—it's to create more beta-cutoffs faster.

If these two things seem like essentially the same thing to you, it's because they are. However, there is a subtle shift in perspective here that I found illuminating when I first learned it. In this chapter, we will *not* be attempting the quasi-paradoxical task of finding the best move without searching,[1] rather we are simply finding the moves which we think will lead to beta-cutoffs. This is the philosophy which guides our efforts to sorting moves.

9.2. Not the Physicist Again...

We already have a rather crude way to approximate the value of a move—the evaluation function! Of course, we should expect this to not work out too well. The evaluation function isn't built for this—it's far too imprecise to be useful at the root node. But it can't hurt to try:

[1] how to find the best move: step 1: search the best move first...

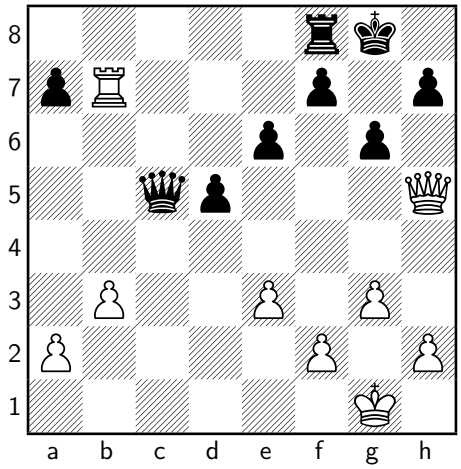
Depth	Vanilla α/β	α/β with physicist sorting
1	44	44
2	414	389
3	12,813	2,158
4	113,364	13,447
5	650,504	91,094
6	11,830,531	347,633

It... works?

Well... yeah. Sorta. A factor of 30 improvement is certainly significant,[2] but it speaks more to the power of move ordering than the efficacy of this approach. We'll eventually learn about more advanced techniques, but in the meantime I think it's worthwhile to investigate exactly why "physicist sorting" works so well.

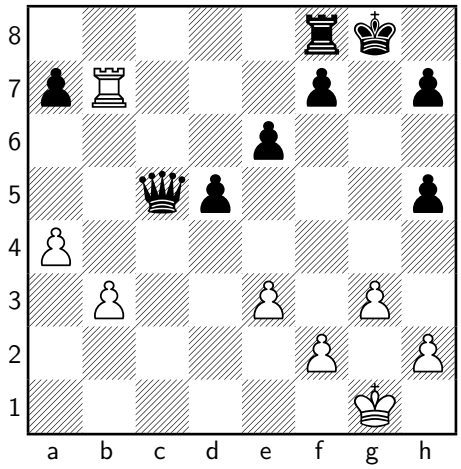
Physicist sorting does one thing right: it pushes captures to the top of the list. The best way to increase your "score" according to the (blind) evaluation function is to take a piece. The fact that this is a positive feature may seem strange to chess players—most available captures are bad moves! Indeed, in the following position, white has 5 legal captures, each of which looses the game.

[2]This is a poor way to measure improvement in the algorithm—it depends on both depth and position.



Why then do we *want* to search captures first?

This line of reasoning is too human. The computer doesn't see this single position, it's looking at *millions* of subsequent positions. While it's true that all of the captures are bad *now*, that may not be the case in the future, especially if we're calculating so many (possibly dubious) lines. White has 37 legal moves in the above position. In all but nine of them, white's queen will be taken on the next move: if White plays a4, Black will respond gxh5.



If White plays a3, Black will also respond with gxh5. If White plays h3, Black will respond with gxh5. In *76%* of White's legal moves, Black's best response will be to take a piece. We don't search captures first because we think they will give the best moves. We do it because they give the best *refutations* to bad moves. Similar reasoning applies to (capture) threats, and re-captures.

9.2.1. Implementing Physicist Sorting

If you'd rather not learn/write the other move ordering methods, physicist sorting is the way to go. It's simple, doesn't muck up the rest of your code, and is *far* better than no move ordering at all.

However, given that the process essentially performs a 1 depth search on *all* nodes with depth ≥ 2, it's worth thinking about how to do it efficiently.

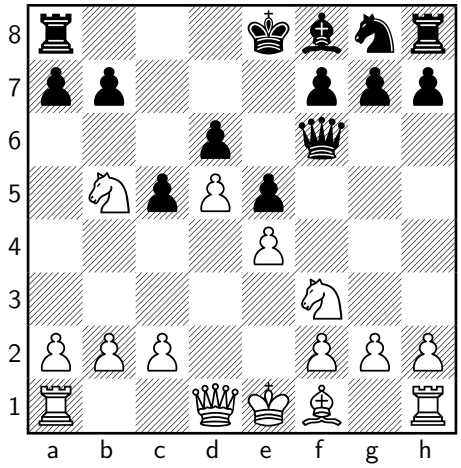
Most of the information the evaluation function uses (bishop pair, doubled pawns, king safety, etc.) isn't especially relevant for sorting purposes. Because physicist sorting only compares moves *from the same position*, it only needs to know the *change* in evaluation to sort the moves, not the true evaluation.

If you only care about the (approximate) change in evaluation, you can get away without looping through the entire board. The only relevant pieces of information are the piece that was moved, where it moved from, where it moved to, and what piece it took (if any). Using that information, you can calculate the change in material, and the change in piece-square table evaluation. Everything else stays the same, so this encapsulates value is roughly the change in overall evaluation.

Of course, this is a very bare-bones calculation, but that's ok. We don't need too much precision here, only enough to give a decent-ish sort.

9.3. Killer Moves

Consider the following position, black to move.



White is threatening to play Nc7+, forking the king and rook. Chess players know that black has only a few reasonable ways of defending this, but the computer has no clue. It'll search plenty of moves—a6,a5,Be7,Ne7,... etc. each time (eventually) discovering that Nc7+ is a refutation. It'd be nice if the computer could somehow "remember" that, "hey! the reason a6 doesn't work also means that a5 doesn't work," instead of spending resources re-discovering that Nc7+ is a problem.

Remembering things is easy for computers—just write down the move to be remembered, and place it towards the front of the move ordering. We just have to figure out what's worth remembering.

The distinguishing feature of Ne7+ is that it refutes *lots* of moves. After black has found a viable solution, this can be measured by the number of beta cutoffs Ne7+ causes. These are killer moves—moves which cause a large number of beta cutoffs elsewhere in the tree.[3] Moves which caused beta cutoffs in the past are likely to cause beta cutoffs in the future, so we ought to search them first (or close to first). This is the main idea of Killer moves—remember previous moves which caused beta-cutoffs, then give those moves higher priority in the future.

You can think of this process as a way of gathering information about which moves are good, then using that information to sort the tree more effectively.

My example only deals with sibling nodes—it also works with cousins and second-cousins! If a move caused a cutoff, it is likely that the same move also causes a cutoff somewhere else in the tree *at the same generational level*. Because of this, Killer moves are usually stored in one big table indexed by distance from the root node.[4]

There are a few different ways to implement this table. The most common is to have two "killer slots" for each generational level. Once a move fails high, we put it in the first slot, move what was in the first slot to the second, and forget what was in the second slot. We then give both killers high priority in the search.

You could imagine a fancier system that keeps track of more information, but this has the virtue of being very quick—no looping necessary. You might also imagine we could add more killer slots. This is something you could experiment with, but the number is usually fairly low (ie 2 or 3) because there usually aren't that many significant killers in a typical position, and we don't want to waste space giving bad moves high priority.

Another *very* important detail is that we **do not put captures in the killer table.** There are two main reasons for this: 1. captures are important enough that we deal with them separately (section 9.5) 2. they "clog the system." When captures refute moves, it's usually because the previous move left a piece hanging (see the first position in section 9.2), which doesn't indicate that it might cause cutoffs elsewhere in the tree, just that the opponent did a stupid.

9.4. Iterative Deepening

Here's a stupid idea: If, at depth d, we want to find the best move to search first, why not just find the best move using the search function at depth $d-1$, then search that move at depth d?

[3] This is sometimes called the "history heuristic."

[4] In part IV, I will introduce techniques which distinguish between the *depth* of a search (ie the parameter in alpha/beta), and the distance to the root. These can be different things.

Because that's a terrible idea. Horrific, really. The problem isn't that it wouldn't find a good move—it would—the problem is that it would be *horrendously* inefficient. Effectively, what you're doing is taking your game tree (which, remember, grows exponentially), then attaching another exponentially growing tree (of size one smaller) to *each node*, then adding *another* exponentially growing tree (of size two smaller) to each of *those* nodes, and on and on. This will never work.

However, this kind of approach smacks of memoization, and we already have a framework (Transposition Tables, chapter 8) to remember the results of previous calculations, so maybe there's some clever way to get around this runtime blowup.

The basic idea from before was that we can use a shallow search to inform the move ordering of a deeper search—the information from a depth 3 search is useful when performing a depth 4 search. To avoid the recursive blowup, we will do this process *iteratively* instead of recursively:

- Search the root with $d = 2$, and record the relevant sorting information (ie best move) in the transposition table.
- Search the root with $d = 3$ using information from the previous step to help sort moves, then update the transposition table with more accurate information.
-
- Search the root with $d = 4$ using information from the previous step to help sort moves, then update the transposition table with more accurate information.
- $\quad\vdots\qquad\vdots\qquad\vdots$

This process is repeated as many times as we can within our given time frame.

What's amazing about this process is that it doesn't just help sort at the root—it helps sort at *every* node in the tree. Searching the root at depth 4 entails searching its child at depth 3. When we perform next iteration, the root is searched at depth 5 (the depth 4 search helps), while it's child is searched at depth 4 (and the previous depth 3 search helps!). Indeed, at the n^{th} iteration, every node is searched at depth one more than it was in the previous iteration.

You might reasonably ask if this is more efficient than simply doing a single search—it certainly looks like we're doing *additional* work. Do the benefits of better move ordering outweigh the costs?

Short answer: yes. Long answer: math. The amount of time it takes for each step is roughly a constant multiple of the amount of time for the previous step[5], or equivalently a

[5]This is actually *not* true, due to the odd-even effect, but for conceptual approximations it works.

constant fraction of the time of the next step. Say this constant multiple (the "branching factor") is b, and x is the amount of time the final step takes. Then the total time can be approximated as

$$\sum_{k=0}^{d} x \left(\frac{1}{b}\right)^k$$

Using that calculus formula you forgot awhile ago, we can see that

$$\sum_{k=0}^{d} x \left(\frac{1}{b}\right)^k \leq \sum_{k=0}^{\infty} x \left(\frac{1}{b}\right)^k = \frac{x}{1-\frac{1}{b}}$$

If your effective branching factor is 10 (which is somewhat reasonable without advanced techniques), we can be fairly confident the process will take no longer than

$$\frac{x}{1-\frac{1}{b}} = \frac{x}{1-\frac{1}{10}} = \frac{x}{\frac{9}{10}} = \frac{10}{9}x$$

...which is barely more than x. This means we actually don't loose that much from searching the first $d-1$ depths, while the potential upside of good move ordering is *huge*.

9.5. Wining and Losing Captures

Absent any other considerations, pawn captures knight is a good move. Even if the pawn is recaptured, we still win material. The same cannot be said for queen takes knight—if the knight's protected, we're out of luck. With this in mind, we will distinguish between "winning" and "losing" captures—a "winning" capture happens when the capturing piece is valued less than or equal to the captured piece, and all other captures are "losing". We want to search winning captures first, because they are most likely to be good.

9.6. Putting it all Together

There's a lot going on in this chapter, and we still need to know how the individual pieces play together. Moves are sorted primarily according to the following list:

1. The "hash move" (the move in the transposition table found using iterative deepening)
2. Winning captures
3. Killer moves
4. Losing captures
5. Other

...and then are sorted secondarily according to our friend the physicist.[6] More advanced engines make this list more detailed, adding in places for (under)promotions, castling, and further dividing the above categories.

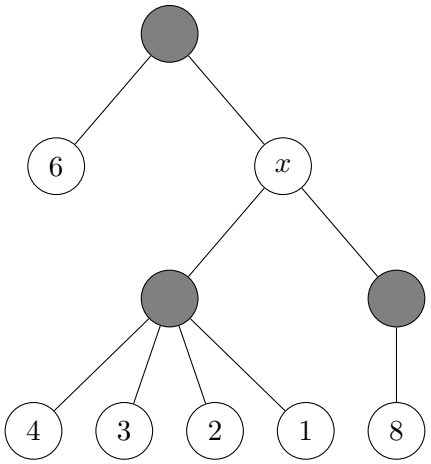
[6] To the people making their own: make sure there are no duplicates. Moves can often appear in multiple categories.

10. Minimal Windows and PVS

The vast majority of lines calculated in a typical game tree are complete nonsense. Of course, we have no way of knowing beforehand exactly which of these lines are useless, but if we did, we could leverage this information by directing the computer to search in places where its computational power is most useful. Obtaining such information does not come for free, but it can be done efficiently enough that its benefits outweigh its costs. This is the premise of Principal Variation Search.

10.1. Minimal Windows

This section is concerned with identifying inferior subtrees. In particular, we would like to know when the value of a tree is lower/higher than some specified number. For instance, in the following tree, we would like to know if the right subtree (with root x) is greater than 6.

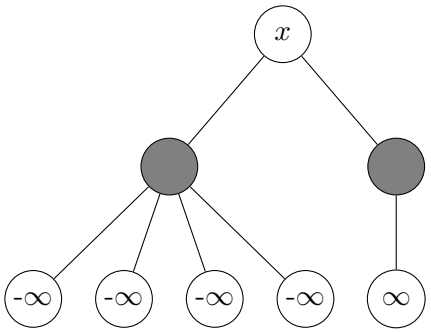


If you trace through this tree using the alpha/beta pruning (which you should!), you will search *every* node. Alpha/beta needs to do this because the value of x *might*[1] become the value of the entire tree, in which case it needs to be exact.

If we *only* care about whether the subtree is greater than 6, we need no such precision. In particular, 4, 3, 2, 1 are all less than 6, and therefore can be regarded as 'the same' for our purposes. Setting aside alpha/beta for a moment, let's can carry through with this logic.

[1] alpha/beta has no way of knowing ahead of time—try replacing 1 with -10, and 8 with (-9 or -11).

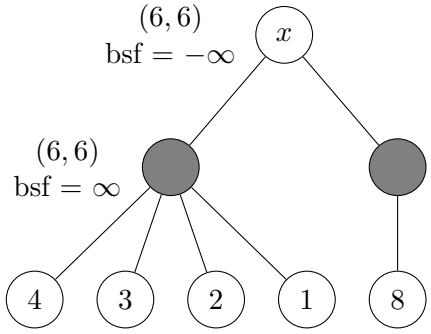
Instead of labeling nodes with numbers, we can categorize them into one of two types: those that are less than or equal to 6, and those that are greater than 6. I will use $-\infty$ and ∞ to denote these nodes respectively:



Tracing the tree now becomes very easy: upon searching the left black node (lbn), we see that it's first child is less than 6. Because lbn is black and therefore minimizing, the value of lbn will also be less than 6, *so we don't have to look at any of its other children* (sound familiar?). Moving onto rbn, we an see that its value is greater than 6. Then, the value of x is the maximum of (greater than 6) and (less than 6), ie greater than 6.

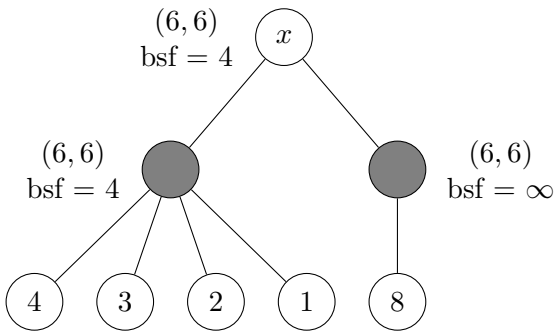
We arrived at our answer *without* visiting all 4 of lbn's children, meaning our approach is more efficient than the simply calling alpha/beta.[2] The key is the logic game we played with the lbn's first child: "...it's first child is less than 6. Because lbn is black and therefore minimizing, the value of lbn will also be less than 6, so we don't have to look at any of its other children." Or to put it another way: once we know that lbn is less than 6, we don't care *how* much less, only that it is.

That last sentence should've rung a bell—it's the exact same language I used when talking about pruning. This is no accident! In fact, this process of labeling nodes with $\pm\infty$, then using pruning logic is just a particular case of the alpha/beta algorithm from the previous chapter. To see this, let's trace through the tree using $\alpha = \beta = 6$.



[2]Think about why this does not contradict theorem 2 from chapter 6.

At this stage, we visit lbn's first child, and see that it is 4. $4 < 6$, so this triggers a cut-off. The value of lbn is set to 4, and we visit rbn:



We proceed, checking rbn's children. 8 is the only child, so its value is passed up to rbn, and onto x. From there, we see that $8 > 4$, and update bestSoFar accordingly. x has no more children, so the process stops and we conclude that x **is greater than[3] 6**. By setting $\alpha = \beta = 6$, we are telling the algorithm that anything above $\beta = 6$ for White or below $\alpha = 6$ for Black is good enough—exactly matching the $\pm\infty$ logic from before.

Appropriately enough, this technique for determining if a tree is less than or greater than a specified value is called a minimal (or null) window search, referring to the fact that the alpha/beta window is as small as can be. Because the window is so small, we get *lots* of cutoffs, making minimal windows very fast.

The main takeaway here is that minimal windows can be used to *efficiently* determine when a (sub)tree is larger or smaller than a specified value.

Many places online will have the "minimal window" actually be $(b-1, b)$ (when b is the value we're testing against) instead of the (b, b) I am using here. This difference is due to a choice of implementation of alpha/beta—I'm using 'fail-soft', while many people online use 'fail-hard'. The difference is subtle and can be confusing to beginners, so I omitted its explanation. Just know that it exists, and that it affects the implementation of minimal window.

10.2. Principal Variation Search (PVS)

Minimal windows are a sort of tool in the chess programmer's toolbelt. Our first use of this tool will be Principal Variation Search, or PVS.

[3]The fact that the value of x is also 8 is no accident. We'll get to that in section 10.3.

As discussed before, most lines calculated by our algorithm are complete nonesense. We don't know beforehand which moves are garbage, but we do know that *most* of them are. We're going to take advantage of this fact by *assuming* each move is terrible. This assumption might be wrong, but we now have a convenient way to figure out if we are: minimal windows.

So here's the plan:

1. Search the first move like normal
2. For each subsequent move, assume it is worse than the current bestSoFar
3. Check if this assumption is correct using minimal windows
4. If we are correct, move onto the next move—we've established it's worse, so there's no sense in searching it any further
5. If we're wrong, darn. We'll have to find the exact value, which means searching the position again, this time with the full window.

There are two conflicting forces acting upon the efficiency of this algorithm. The first is that we *don't* have to search inferior moves deeply—minimal window search is fast—which speeds up the search. The second is that good moves, the ones that fail-high on the minimal window search, must be searched *twice*—once using the minimal window and once using the full window. This makes our search slower.

Our gamble in PVS is that the benefits of searching with minimal windows outweigh the costs. Whether or not this actually happens depends on the position in question, but the gamble usually pays off.

Here's the code:

```
def PVS(pos, depth, alpha, beta):
    if depth = 0:
        return evaluate(pos)
    bestSoFar = -PVS(after_move(pos, firstMove), depth - 1, -beta, -alpha)
    for move in pos.legal_moves[1:]:  // excluding the first move
        // search minimal window centered at -bestSoFar
        value = -PVS(after_move(pos, move), depth - 1, -bestSoFar, -bestSoFar)
        if value < bestSoFar: // If it's worse, we don't care about it
            continue
        // re-search
        value = -PVS(after_move(pos, move), depth - 1, -beta, -alpha)
        if value > bestSoFar:
            bestSoFar = value
        if bestSoFar >= beta: // pruning condition
            return bestSoFar
        alpha = max(alpha, bestSoFar)
    return bestSoFar
```

And that's principal variation search! There are many different ways of implementing it, but this is the basic idea.

There are a few improvements we can make on this code. These changes don't make a *significant* impact on performance, and they also rely on somewhat difficult ideas. I wouldn't worry too much about it if you don't understand/implement the changes in the remainder of this section, but I do think they are things worth thinking about.

The first improvement is about this business of treating the first move differently. Ostensibly, we do this so we know what value to use with the minimal window search for subsequent moves, but we already have a value which does this: α. If the first move ends up having value less than α do we care how much less? Nope! Then, we can use α for the first minimal window search. In fact, because we update α to be bestSoFar whenever bestSoFar improves on α, we can just use α for *all* minimal window searches. Our code now looks like this:

```
def PVS(pos, depth, alpha, beta):
    if depth = 0:
        return evaluate(pos)
    bestSoFar = -infinity
    for move in pos.legal_moves:
        value = -PVS(after_move(pos, move), depth - 1, -alpha, -alpha)
        if value < bestSoFar:
            continue
        value = -PVS(after_move(pos, move), depth - 1, -beta, -alpha)
        if value > bestSoFar:
            bestSoFar = value
        if bestSoFar >= beta:
            return bestSoFar
        alpha = max(alpha, bestSoFar)
    return bestSoFar
```

You may even ask why we use the variable bestSoFar at all, simply using α in it's place. Indeed you can, and many programmers do. This is another difference between 'fail-soft' and 'fail-hard' frameworks.

The next change we will make to the code as written above has to do with re-searches. Of course, we only need to re-search when the minimal window search finds `value` to be larger than bestSoFar. But if `value` is also larger than β, The Important Fact About Failing High (section 5.8) tells us that the re-search will also be larger than β. In that case, we're going to fail-high anyway, so there's no point in re-searching:

```
def PVS(pos, depth, alpha, beta):
    if depth = 0:
        return evaluate(pos)
    bestSoFar = -infinity
    for move in pos.legal_moves:
        value = -PVS(after_move(pos, move), depth - 1, -alpha, -alpha)
        if value < bestSoFar:
            continue
        if value < beta:
            // only re-search if value is in the window
            value = -PVS(after_move(pos, move), depth - 1, -beta, -alpha)
        if value > bestSoFar:
            bestSoFar = value
        if bestSoFar >= beta:
            return bestSoFar
        alpha = max(alpha, bestSoFar)
    return bestSoFar
```

This modification reveals why it is called principal variation search: we only ever use a full window when we find a move which is better than `bestSoFar` (in fail-hard, α) but worse than β, that is whenever we've found a PV-node.[4]

The last tiny optimization will also make use of the Important Fact About Failing High. If we determine that we have to re-search (that is, the minimal window value is in (α, β)), the IFAFH tells us that the re-search can't be lower than[5] the value found by the minimal window. This means we can update the bounds we pass through to the re-search. Specifically, we change $-\alpha$ to negative the value of the minimal window search:

[4] Ok *technically* we could still fail-high after a re-search, but the *spirit* of PVS is only using the full window on PV-nodes.

[5] Major asterisk here: there's something called search instability that can make this statement false. Don't worry about it right now.

```
def PVS(pos, depth, alpha, beta):
    if depth = 0:
        return evaluate(pos)
    bestSoFar = -infinity
    for move in pos.legal_moves:
        value = -PVS(after_move(pos, move), depth - 1, -alpha, -alpha)
        if value < bestSoFar:
            continue
        if value < beta:
            // change in the following line
            value = -PVS(after_move(pos, move), depth-1, -beta, -value)
        if value > bestSoFar:
            bestSoFar = value
        if bestSoFar >= beta:
            return bestSoFar
        alpha = max(alpha, bestSoFar)
    return bestSoFar
```

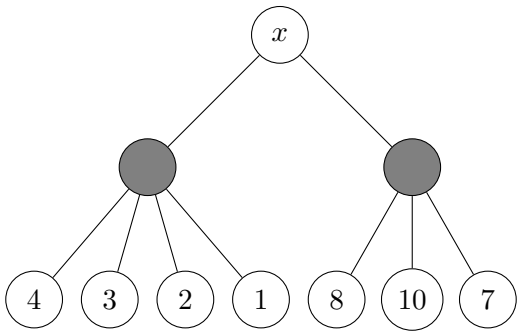
This change is the defining feature of 'NegaScout'.

As said, don't panic if you didn't understand all of this. It's not all that important.

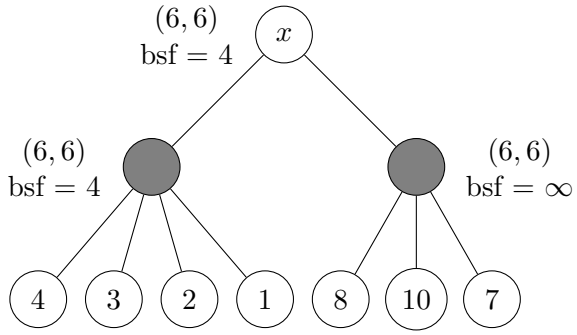
10.3. Depth = 2

In general, minimal windows can *only* distinguish between better/worse, and don't give exact values. Despite this, our first example *did* give its exact value. It will be worthwhile to look into other situations where minimal windows can be this precise.

Let's make a slight variation on the first example:



I've added the two nodes 10 and 7, which changes the value of x to 7. When tracing this tree (with window $(6, 6)$), lbn preforms the same cutoff—only searching the node 4 and pruning the rest.



However, when we get to rbn, we search all its children:[6] the algorithm continues to search in hopes that it can find a child less than 6 (as rbn is black), but it never does. No matter how many children over 6 I add to rbn, it will still never prune any nodes. This leads to a nice conclusion: if minimal window search says rbn is larger than 6, it also returns the exact value. This effect is felt a level up. All of x's children with values over 6 will be given exact values by minimal window search. x chooses the maximum of its children, so *if* x has value over 6, it will choose an exact value, giving x an exact value.

To summarize: *if* x is at depth 2, *and* minimal window search says it is larger than 6, it also gives the exact score.

This means our re-searches are sometimes redundant! In particular, when PVS makes a minimal window search, fails high *and* is at depth 1 or 2, we know it is exact, so re-searching is pointless. Then, we can add another condition to our re-search:

```
def PVS(pos, depth, alpha, beta):
    if depth = 0:
        return evaluate(pos)
    bestSoFar = -infinity
    for move in pos.legal_moves:
        value = -PVS(after_move(pos, move), depth - 1, -alpha, -alpha)
        if value < bestSoFar:
            continue
        // change in the following line
        if value < beta and depth > 2:
            value = -PVS(after_move(pos, move), depth-1, -beta, -value)
        if value > bestSoFar:
            bestSoFar = value
        if bestSoFar >= beta:
            return bestSoFar
        alpha = max(alpha, bestSoFar)
```

[6] remember that this is vanilla α/β search, not NegaMax.

```
return bestSoFar
```

10.4. Broccoli!!

1. (When the root node is white) Verify that minimal window search correctly identifies subtrees that are *less than* a specified number (the example in the text showed greater than).

2. Verify that the minimal window search described in section 10.1 works when the root is black.

3. Verify that the code at the beginning of section 10.2 reflects the ideas in section 10.1 in a color-neutral way.

4. Find an example of a tree for which minimal window search *does not* give the exact value of the tree.

5. Fill in the details of the depth = 2 idea when the root node of the subtree is black.

6. Verify that the NegaMax code on the previous page reflects the ideas of section 10.3 in a color-neutral way.

7. Show that the depth = 2 idea *doesn't* work when depth ≥ 3. (Find a counterexample, see problem 4)

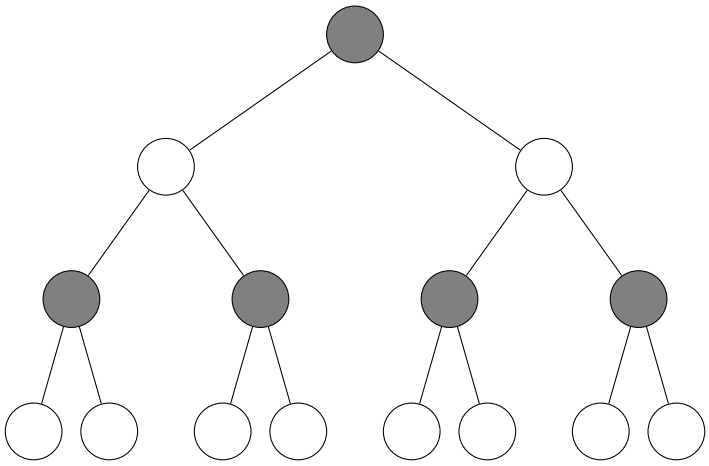
Part IV.

Selectivity

Selectivity: What is?

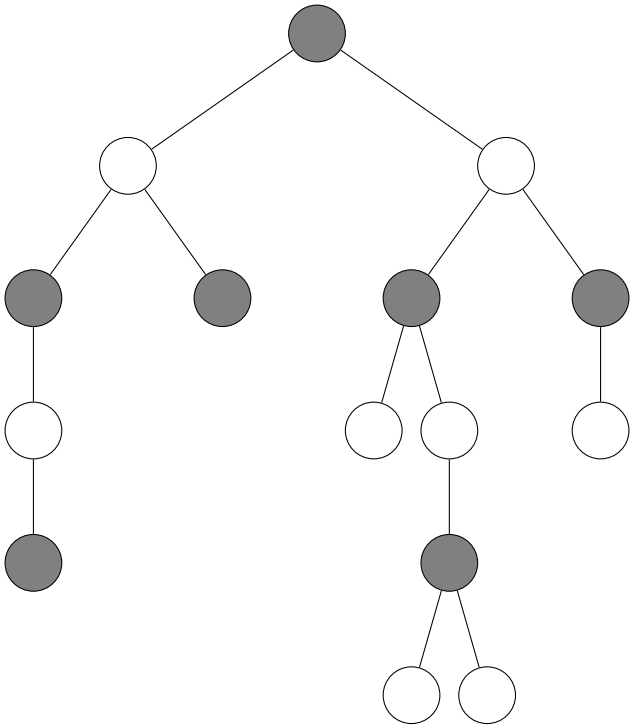
The fundamental problem of chess programming is the enormous number of nonsensical branches of the game tree. So far we've tackled this problem with theoretically sound algorithms—alpha/beta is "correct" in the sense that it always agrees with MinMax. However, there are only so many improvements we can make in this direction, and at some point we must give up this "correctness" in order to gain strength. Welcome to the dark arts of chess programming.

The primary way we gain strength at the cost of correctness by leveraging the fact that some lines are more important than others. So far, we have only dealt with trees that look like this:



That is, trees where every terminal node has the same depth—the tree is "short and fat," searching *every* branch. In an effort to limit the number of senseless lines, we can *select* those that look promising, and do away with the ones which don't look like they lead to anything. A selected game tree might look something like this:[7]

[7] Of course, alpha/beta won't search all these nodes, but hopefully you can understand the point I'm trying to make.



This game tree is "tall and skinny," searching a few lines deeply, and other more shallowly. Of course, this is a dangerous game to play—we're losing information by skipping some branches of tree, and banking on the benefits of a deeper search.

10.4.1. There is more to life than Ply

Selectivity is very powerful—it can give you incredible gains in both search depth and strength, but those are not the same thing. It's very easy to tweak some of the parameters in the coming sections to dramatically increase the search depth, but sacrifice some other virtue and ultimately make your engine weaker. If you are making an engine yourself, you must tread this line carefully. Constantly test your engine, and don't get too caught up in trying to search $n + 1$ ply.

11. Null Move Pruning

Null move pruning is yet another solution to the problem of nonsensical lines in a huge game tree. It has the advantage of being relatively easy to understand, (usually) painless to implement and a very powerful optimization in its own right.

11.1. The Null Move Heuristic

Here's a proposition which may or may not be true:

> In each position, the player to move can improve his or her situation my making a move. That is, given the option between making a move and passing (ie opting to not move), the player to move will never choose to pass.

Chess players know that this proposition is in fact *false*, but in this section we will assume it to be true—as it turns out, this assumption (the "null move heuristic") is quite useful.

We are interested in expediting the process of a beta cutoff. As a brief review, a beta cutoff happens at a node p when know for sure that p has an ancestor of the opposite color which has a better option than p. In this situation, p will never be played (by a perfect player) and therefore we shouldn't waste resources searching p any further. The value of the ancestor's other option is β, and the cutoff is triggered when the value of p is known to be larger than β.[1]

The null move heuristic can help us determine when value(p) $> \beta$. This usually happens with a series of recursive calls on p's children (and stopping whenever they exceed β), but we can do better: from the node p, make a "passing" move, *then* evaluate the position recursively. If the position is *still* $\geq \beta$ at node p, the null move heuristic says that making some other move will have value $\geq \beta$.

To flesh this out in math, say you have established that value(null move) $\geq \beta$. The null move heuristic says value(some other move) $\geq$ value(null move), and *together* these facts say

$$\text{value(some other move)} \geq \text{value(null move)} \geq \beta$$

[1] If this was confusing, you might want to review chapter 5

and therefore that value(some other move) $\geq \beta$. This allows you to conclude value(p) $\geq \beta$ without searching any of p's children.

11.2. The Execution

Code for null move pruning, as described in the previous section, would look something like this:

```
def null_search(pos, depth, alpha, beta):

    ...Transposition table stuff...

    value = -null_search(after_pass(pos), depth - 1, -beta, -alpha)
    if value >= beta:
        return value

    ...usual search stuff...
```

...which goes *before* the big loop searching all the moves from `pos`, but *after* the transposition search. In principal, the value of the null search could be used to update alpha with `alpha = max(value, alpha)` (in accordance with the null move heuristic), but this turns out to not make a huge difference if the engine is using minimal windows.

There is still the lingering question of whether this is actually an improvement. It seems as though we are giving *each* node on the tree an additional child[2] with the hope of finding a cutoff earlier. Is this expansion of the tree worth it?

Well no, not really. To begin with, we can only hope to make an improvement on cut-nodes,[3] as we can only prune when the value of the null move is larger than beta. But we don't even improve on all cut-nodes—only the ones that are *so much better* than beta that they can make a passing move and still be good. But even if you do make an improvement, how much better do you do? If you have good move ordering, (and no null-move shenanigans) most cut-nodes ($> 90\%$) perform a beta cutoff within the first few moves searched—often the very first. If you're using the above algorithm and are lucky enough to hit a null-move cutoff, you're guaranteed to prune after the very first (null) move. This means you save *maybe* 1 or 2 nodes on average, even when you get lucky. Things aren't looking good for null moves.

---

[2] as well as an additional child to each added child. Recursion!

[3] there are more advanced techniques not discussed here which revolve around descendant whether a node will be cut or all, and making decisions accordingly.

We're going to solve this problem by *drastically* reducing the cost.

The inequalities established by the null move heuristic (which are value(best move) $\geq$ value(passing move)) is very rough. The exact difference between those two values is something like 25 centipawns, with huge error bars attached. Furthermore, we don't really care about the precise value of the null move—It is only ever used to create a cutoff, and will never determine the value of a node.

Taken together, these facts mean we can afford to be a little loosey-goosey with our evaluation of the null move. Specifically, we will search it at a lower depth than normal. The difference in precision in a depth n and a depth $n-1$ or $n-2$ search isn't all that big[4]—almost always lower than that 25 centipawn threshold, so we (usually) don't lose anything by lowering the depth. This is simple to implement:

```
value = -null_search(after_pass(pos), depth - 1 - R, -beta, -alpha)
if value >= beta:
    return value
```

...where `R` is a parameter which signifies the number of moves to reduce, commonly set to 2.[5] Each of those added children from the previous page now cost a tiny fraction of what they did previously, to the point of being negligible. It seems there's only upside.

Again, no. This depth reduction business *does* lose information. It's entirely possible we miss something important at `depth - 1` ply that the algorithm won't see when searching at `depth - 1 - R`. In other words, we've introduced a way for the computer to make a mistake; it will sometimes confidently assert a position is very strong when in fact it missed a key move at a depth we though null moves didn't have to search.

What do we do about this problem? Nothing. The computer only has so much processing power, and we need to allocate resources in the most cost-effective manner possible. Through testing, programmers have found that trading faster cutoffs for possible mistakes is worth it, so we do.[6] We aren't trying to make a correct engine, we're trying to make a strong one.

When we make the null move search

```
value = -null_search(after_pass(pos), depth - 1 - R, -beta, -alpha)
```

we *only* care about it if `value` $\geq \beta$. This makes it a prime candidate for a minimal window search (see section 10.1). This is an easy change:

[4] and gets smaller as n gets bigger

[5] Stockfish 10, the strongest open source engine at the time of writing, has the line `Depth R = (737 + 77 * depth) / 246 + std::min(int(eval - beta) / 192, 3);`, so there is quite a bit of room for optimization in this number.

[6] If you find this disturbing, consider that the "but if I only searched a little farther..." was already a problem way back in MinMax. We've only added the same problem to a different region of the tree.

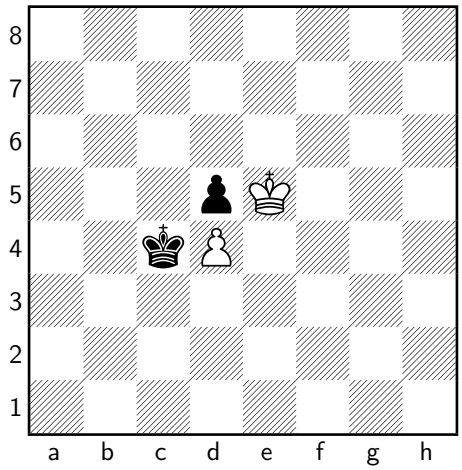
```
value = -null_search(after_pass(pos), depth - 1 - R, -beta, -beta)
```

One more thing: while we are temporarily suspending the rules with passing moves, we can't do this willy-nilly. For instance if we make a passing move while in check, our opponent will be able to capture our king, and will throw a monkey wrench in our program. This is a simple fix—just encase the above code in an if statement:

```
if pos.safe_to_reduce()
    value = -null_search(after_pass(pos), depth - 1 - R, -beta, -alpha)
    if value >= beta:
        return value
```

11.3. Zugzwang

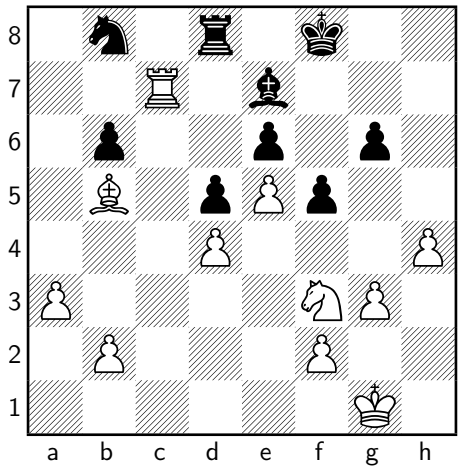
As mentioned earlier, the null move heuristic is a helpful untruth. To understand how it fails, consider the following position—who is winning?



Trick question—I haven't given you enough information! Whose turn is it? The result of the game hinges on the answer. If white is on move, she is *forced* to move the king away from the defense of d4, giving up the pawn and the game. Likewise, if it is black's turn, he is *forced* to give up his d5 pawn. In this position, you *don't* want it to be your turn. Every legal move is worse than passing, but the rules of chess force you to

make a move even though you would rather not. This peculiar situation is known as "zugzwang."[7]

Of course, it must not happen to both players at once as in the previous example. In 2019, world champion Magnus Carlsen reached the following position while streaming:



Carlsen has trapped every black piece, saying "I think this is a case of absolute zugzwang". White doesn't have an immediate threat—if black is allowed to pass, he will survive for the moment, but the rules obligate him to move. Carlsen played the waiting move 1. Kg2, and black resigned after 1...Kf7 2. Ng5 Kf8 3. Nxe6

The existence of zugzwang presents a bit of a problem for null move pruning. An engine could conceivably be on the losing side of a zugzwang position, make a(n illegal) passing move, and say to itself "wow this passing move was great! I don't have to search any farther since the null move heuristic says there's another even better move!" when in reality no such move exists.

This is clearly suboptimal, but how do we solve this problem. Here are a couple ideas:

1. Don't use null moves.
2. Do use null moves, but do nothing about zugzwang.
3. Try to detect zugzwang, and selectively use null moves.

In light of the aberrant decisions null moves can sometimes create, you might be tempted to go for option 1. But remember, we are interested in making a *strong* engine, not a *correct* one.

As it turns out, the speed benefits of null move pruning outweigh the occasional blunder it can cause. It's generally better to opt for option 2. over option 1. Nevertheless

[7]Stolen from German, "zug" meaning "move" (in this context) and "zwang" meaning forced/compelled/obligated

the drawbacks of null move pruning are real, and we should think about how about mitigate them.

The first thing to note is that zugzwang are somewhat rare, and almost always happens in the endgame. The above Carlsen zugzwang was noteworthy because black still had 3 pieces on the board. This is *ridiculously* rare. So rare in fact that's it's the only thing we really need to check for—that is, check to see if there are any non-(pawn or king) pieces on the board for the side to move. If there are, it is very, very unlikely you will have to worry zugzwang.

Of course, if you really wanted to you could create/use some more sophisticated (and computationally expensive) zugzwang checking algorithms[8], but sure to test whether it is worth the drawbacks.

The end algorithm (fragment) is

```
if pos.safe_to_reduce() and !pos.possible_zugzwang()
    value = -null_search(after_pass(pos), depth - 1 - R, -beta, -alpha)
    if value >= beta:
        return value
```

[8] google "double null move chess programming"

12. Late Move Reduction

In the previous chapter, we got around the cost of searching the null move by reducing the search depth. We paid for this in precision, but it didn't matter because we were testing a "fake" move. Late move pruning applies the same idea to moves which are actually on the game tree. This comes with significant risks and rewards.

12.1. Reductions

Let's say you've got a kickass move orderer. You've implemented all of chapter 9, and the beta cutoffs are flying off the tree so fast, the local landscaping company wants you on their payroll. What now?

If you *really* have faith in your move orderer, you can use it for much more than ordering moves—it's a catch all algorithm for determining the importance of searching a particular node. We can leverage this information to make decisions about what to search and how deeply. In practice, this usually means not searching the moves further down in the move ordering list (ie low priority) as deeply as we otherwise would. Whenever we make a decision to search a move at a lower depth than normal, it is called a *reduction*,[1] and reducing moves with low priority in the ordering is called a *late move reduction*. In general, it will look something like this:

```
def AB(pos, depth, alpha, beta):

    ...bla bla....

    for move in pos.legal_moves:

        ...bla bla....

        reduce = how_much_to_reduce(move)
        value = -AB(after_move(pos,move), depth - 1 - reduce, -beta, -alpha)

        ...bla bla....
```

The exact amount to reduce (determined by `how_much_to_reduce` above) is a bit of an art. You generally don't want to be too aggressive (default can be one ply), but some of the best engines in the world get away with it using *lots* of tuning.

[1] If you want, you can think about null move pruning as a kind of reduction on steroids.

Just like in null move pruning, we might be surprised and find a low-priority move that exceeds the value of α. In this case, we really need to know the precise value of the node (it has the chance of turning into a PV!), so we must re-search:

```
...bla bla....

reduce = how_much_to_reduce(move)
value = -AB(after_move(pos,move), depth - 1 - reduce, -beta, -alpha)
if value > alpha:
    value = -AB(after_move(pos,move), depth - 1, -beta, -alpha)

...bla bla....
```

In the true spirit of selectivity, this might all blow up in our face. Those extra nodes we aren't searching contain potentially useful information which we aren't considering. We're making a guess—a justified guess—that those resources are better spent elsewhere.

12.2. When and How Much to Reduce

In the previous section, I brushed aside all the details of late move reduction in the line `reduce = how_much_to_reduce(move)`. There are some important questions that need to be answered by this line—when should it be zero? ie when should we (not) reduce? What factors other than position in ordering should we take into consideration? How do we decide the magnitude of the reduction? Answering these questions is more art than science, so I'll only be able to give vague, borderline unhelpful answers. The general idea is that interesting or dynamic moves deserve to be fully searched, while boring ones maybe don't.

When should we (not) reduce? A line needs to be drawn between "late" and non-reduced moves. The lazy way to do this is fix a value, say 10, then search the first 10 moves at full depth, and reduce all remaining moves. A perhaps more interesting approach would be to take the move ordering list in section 9.6 (page 104), and reduce everything in the "other" category. This means searching hash moves, killer moves, and captures at full depth.

It is in general a good idea to give high search priority to forcing and dynamic moves. There are two main reasons for this: 1. These moves are more likely to be relevant to the rest of our search—they produce lots of beta cutoffs, and are more likely to become a PV. 2. Dynamic and forcing moves are by nature very double edged, so it's much easier to miss something important—they deserve to be searched fully.

What factors other than position in ordering should we take into consideration? Anything that indicates the move is (not) interesting (see section 12.3). Interesting moves get a pass on reductions. Often this means looking at the move's history. Has it caused beta cutoffs in the past? Has it failed low in the past? Is it now or has it ever been a PV? etc. Trying to think of your own indicators is a fun exercise.

How do we decide the magnitude of the reduction? There's a lot that goes into this (see subsection 12.2.2). The safest option is to just set all reductions to 1, but we can be more creative than that.

In general, the deeper the search, the more we can afford to reduce. For this reason, some engines will *slowly* increase the amount of reductions as the depth increases using functions like `log(depth)`. In principal, the additional factors mentioned above can also be used to affect the size of the reduction, but this can create more problems that it solves.

12.2.1. Fractional Plies

Nominally, you can only reduce by a whole number of ply. This is a bit of a coarse scale—perhaps some reduction criterion isn't good enough to warrant a full ply reduction, but you still believe it says something useful. You'd like to be able to reduce the move by a "half ply," whatever that means.

You can! simply decrement the depth by a half (set[2] `reduce = .5`), and change your base case from `if depth == 0` to `if depth <= 0`. With only a single application, this reduction won't cause any change in the search tree, but if a node is reduced by a half ply, *and* has an ancestor which was reduced by a half ply, the rest of that line will be searched as if our original node were reduced by a full ply. These "fractional plies" allow for more refined reductions.

12.2.2. The strength of the engine affects LMR

Reductions are sort of "cheating" the search procedure—you're diverting resources from (supposedly) less important branches to (supposedly) more important branches. It's much easier to get away with this if your engine is strong.

If the less important branches have very few resources to begin with, siphoning them off can be disastrous. Reducing by 2 ply in a depth 6 search is a big deal, and probably a mistake. Reducing by 2 ply in a depth 30 search is little problem. This is part of the reason top engines have such aggressive selectivity—the code is so efficient that the downsides aren't as severe. Amateurs with inefficient code should play it safe with late move reductions.

[2]Sometimes, people set "one ply" to be a larger number, say 10, and (in this case) the reduction to 5. This way you're still dealing with integers, but you have more granular parameters.

The same idea plays out in the interaction between move ordering and late move reductions. If the move orderer sucks, we'll end up reducing good moves. The better the move orderer is, the less we have to worry about intentionally not searching something important.

12.3. Reductions in Stockfish

I find the extent of the reduction tuning in Stockfish equal parts fascinating and hilarious, and I want to share.[3] I will warn you against trying to replicate all of these parameters. They're a bit... extreme. There are a lot of things professionals can get away with that us amateur's can't. Hopefully it does give you some ideas.

Below is (part of) the code in Stockfish which corresponds to my `how_much_to_reduce()`.

```
    Depth reduction(bool i, Depth d, int mn) {
        int r = Reductions[d] * Reductions[mn];
        return (r + 503) / 1024 + (!i && r > 915);
}
```

Where `i` is a boolean flag telling us whether the position is improving, `d` is the depth, `mn` is the move number, (ie position in the ordering), and `Reductions` is some array initialized as

```
  for (int i = 1; i < MAX_MOVES; ++i)
      Reductions[i] = int((21.3 + 2 * std::log(Threads.size())) *
      std::log(i + 0.25 * std::log(i)));
}
```

But we're still not done. After setting `Depth r = reduction(improving, depth, moveCount);`, Stockfish spends the next *62 lines of code* tweaking and refining `r`. I'll give an executive summary of the modifications:[4]

1. Decrease `r` if we're getting lots of transposition table hits
2. Increase `r` if other threads are searching this position
3. Decrease `r` (by 2 ply!) if the position is or was a PV.
4. Increase `r` if the best move hasn't been changing.
5. Increase `r` if the position was never in PV
6. Decrease `r` if the opponent has lots of moves

[3] If you have some time to kill, open up the "search" file in the Stockfish source code. Endless hours of entertainment.

[4] mostly paraphrasing the comments in the code

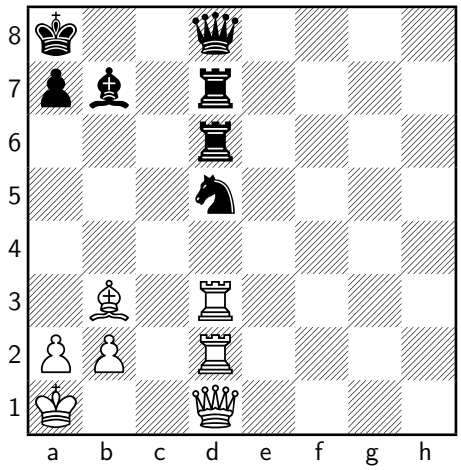
7. Decrease `r` if the move has been singularly extended (see chapter 14)
8. Increase `r` (by 2 ply!) for (expected) cut nodes
9. Decrease `r` for moves that escape a capture
10. Decrease or increase `r` for moves with (respectively) good and bad history

13. Quiescence Search

We will now address a lingering problem with the application of tree searching algorithms to chess. Specifically, none of algorithms can see beyond the 'search horizon', which can cause significant problems that must be dealt with.

13.1. The Horizon Problem

There's something that may have been bugging you since chapter 4. We've been making the rather crude decision to cut off the game tree at a fixed depth. If we search at depth n, what if we miss something important at depth $n + 1$? This can happen quite easily:



Should white play Bxd5?

Bxd5 will result in a sequence of captures, and it's not immediately clear who comes out on top. Human players will count the number of defenders and attackers (assuming the pieces are of equal value). White has four pieces attacking d5 (we count the queen and d2 rook), and black has 4 pieces defending. Because he is defending, black will get the last capture and come out on top.

Fixed depth searches usually[1] have no such heuristics, and will analyze the position according to the depth of its search:[2][3]

[1] look up static exchange evaluation

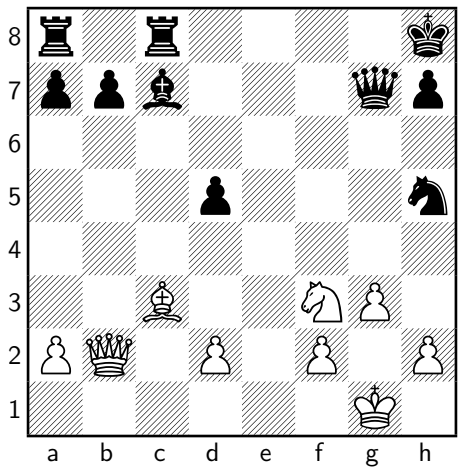
[2] The lines listed are the principal variation—what the engine *thinks* will happen, not what actually will.

[3] If you need to, you can play out the moves online by going to https://lichess.org/editor, setting up the board yourself and clicking "analysis mode"

- **1 ply: 1.Bxd5** This is great! I won a knight!
- **2 ply: 1.Bxd5 Bxd5** Maybe not so great—I trade my knight for a bishop
- **3 ply: 1.Bxd5 Bxd5 2.Rxd5** This is great! I traded a bishop for a bishop and a knight!
- **4 ply: 1.Bxd5 Bxd5 2.Rxd5 Rxd5** This is terrible! I lost a bishop and a rook for a bishop and a knight!
- **5 ply: 1.Bxd5 Bxd5 2.Rxd5 Rxd5 3.Rxd5** This is great! I traded R + B= 8 for a R + B + N= 11.
- **6 ply: 1.Bxd5 Bxd5 2.Rxd5 Rxd5 3.Rxd5 Rxd5** This is terrible! I traded R + R + B= 13 for R+ B + N= 11!
- **7 ply: 1.Bxd5 Bxd5 2.Rxd5 Rxd5 3.Rxd5 Rxd5 4.Qxd5** This is great! I traded R + R + B= 13 for R + R + B + N= 16.
- **8 ply: 1.Bxd5 Bxd5 2.Rxd5 Rxd5 3.Rxd5 Rxd5 4.Qxd5 Qxd5** This is terrible! I traded Q + R + R + B= 21 for R + R + B + N16.

We have a bit of a problem. Blind to the future, the engine will flip-flop between good and bad until the position settles down. This is the first type of horizon problem: the computer will make a move at the search horizon without realizing that the final move hangs a piece.

This second type of horizon problem is more subtle. (Black to play)



Human players will quickly identify that Black's Queen is cooked (it's pinned to the king), and conclude that no resources should be devoted to saving it. Stockfish suggests Bc6, and gives black a 250 centipawn advantage.

Naive fixed depth searches do something interesting:

- **1 ply: 1...Qxc3** I won a bishop!
- **2 ply: 1...d4 2.Bxd4** (anthropomorphizing...) hmmm. If I play Qxc3, I will lose my queen for a bishop. not good. If I do nothing, white will take my queen on the next move. not good. I can't move my queen because it's pinned to the king, so I will block the white bishop by playing d4. I'll lose the pawn on the next move, but that's better than losing the queen.
- **3 ply: 1...d4 2.Bxd4 Qxd4** I won a bishop!
- **4 ply: 1...d4 2.Bxd4 Be5 3.Bxd5** Ok If I do the 3 ply plan, I will lose the queen on the next move. with *black* playing Qxd4. After 1...d4 2.Bxd4, I have the same problem as before: I'm about to lose my queen on the next move if I do nothing. I'll block that threat with Be5. I'll lose a pawn and bishop in the next two moves, but thats better than losing a queen.
- **5 ply: 1...d4 2.Bxd4 Be5 3.Bxd5 Qxe5** I traded a pawn and a bishop for a bishop. not too bad.
- **6 ply: 1...d4 2.Bxd4 Be5 3.Bxd5 Nf6 4.Bxf6** The 5 ply line doesn't work, since the next move will be Qxe5, losing my queen. I need to protect my queen just like on plies 2 and 4. I'll go Nf6. I'll lose the knight on the next move, but that's better than losing the queen.
- **7 ply: 1... d4 2.Bxd4 Be5 3.Bxd5 Nf6 4.Bxf6 Qxf6** I lost a pawn, bishop and knight, but now I get a Bishop back. Not ideal, but its better than losing the queen outright.
- **8 ply:** nooo!!!! Losing the queen is inevitable. oh wait. Is it that bad? I lose a queen for his bishop, but I'm still up in material. I'll just play something like Bc6 and accept the loss of the queen.

Do you see what happened there? Up until 8 ply, Black was planning to stall out the inevitable loss of the queen by giving up material. It is far better to simply allow the queen to be taken, and play up in material.

So what is it about these two positions that messes our current search algorithms? In both cases, the terminal nodes "unstable" in the sense that the evaluation function is extremely unreliable (more so than usual). The horizon problem happens whenever the engine makes important decisions based on calling the evaluation function at unstable nodes, thereby giving an incorrect picture of what's actually happening.

13.2. The Solution

The evaluation does ok when no pieces are hanging and there aren't any crazy tactics available that could create a big material swing in the near future. We call such positions 'quiet'. These positions are fairly easy to identify—just check to see if there are any available checks[4] or captures. If there are none, the position is likely quiet. If there are checks or captures, something exciting might happen in the next few moves and we need to be prepared.

When the search procedure hits a node at maximal depth and this terminal node also happens to be quiet, we can safely call the evaluation function. If the node *isn't* quiet, we have reasonable fear that the evaluation function might miss something in the next few moves.

The naive solution is to simply 'extend' the search by looking one move ahead. Of course this only kicks the problem one ply down, so instead you might search *only captures*. Essentially the program is saying "If there are no outstanding captures, I trust my evaluation function. If there *are* outstanding captures, I need to resolve them to find their 'true' value." This is usually done with a separate function, `qsearch`.[5]

```
def qsearch(pos):
    if pos.is_quiet():
        return evaluate(pos)
    bestSoFar = -infinity
    for move in pos.captures:
        value = -qsearch(after_move(pos, move))
        if value > bestSoFar:
            bestSoFar = value
    return bestSoFar


def main_search_function(pos, depth, alpha, beta):

    ...bla bla...

    if depth == 0:
        return qsearch(pos)
        \\previously "return evaluate(pos)"

    ...bla bla....
```

[4] Finding whether a move is a check can be an expensive procedure. Considering just captures does the job.

[5] for "quiescence search". Quiescence, meaning "inactivity or dormancy".

Nice, clean and simple. Only one problem: it doesn't work.

Although it's easy to think of "resolving all captures" as a short, straightforward procedure, in practice it creates a big mess of captures and recaptures and captures and recaptures—`qsearch` doesn't have a depth limit. These "mutual plunder" situations (where two pieces on opposite colors take turns gobbling up the opponents pieces) will quickly bog down the search procedure. Depending on the position, the code as written above might not finish within the week.

The basic solution to this is to realize that we can transplant many of the optimization techniques used in MinMax to `qsearch`. Specifically, we can inherit the alpha/beta values from the main search procedure, and a do basic application of null move heurisitc:

```
def qsearch(pos, alpha, beta):

    stand_pat = evaluate(pos)

    if pos.is_quiet():
        return stand_pat

    if stand_pat > beta:
        return stand_pat

    bestSoFar = stand_pat // by the null-move heuristic
    for move in pos.captures:
        value = -qsearch(after_move(pos, move), -beta, -alpha)
        if value > bestSoFar:
            bestSoFar = value
        if bestSoFar > beta:  // usual beta cutoff
            return bestSoFar
        alpha = max(alpha, bestSoFar)
    return bestSoFar


def main_search_function(pos, depth, alpha, beta):

    ...bla bla...

    if depth == 0:
        return qsearch(pos, alpha, beta)

    ...bla bla....
```

The term "stand pat" is taken from poker. Here, it's used to represent the baseline lower bound given by the null move heuristic, which we presumptively use as the value for `bestSoFar`.

13.3. MVV/LVA

I'd like to bring your attention to two things:

1. We have not implemented move ordering in the `qsearch` code. Because we are dealing with only captures, we have extra information that we can use to sort the moves (namely, the captured piece).
2. The obvious way of generating all captures (generate all move, filter) is very inefficient for our purpose. Because we are dealing with only captures, we don't have to deal with extraneous moves.

MVV/LVA (**M**ost **V**aluable **V**ictim, **L**east **V**aluable **A**ttacker) will address both of these points.

Here's the algorithm: (the code will of course depend on implementation)

1. Look at the opponent's queen. If I can capture it, find all queen captures and sort them by the value of the attacker in ascending order.
2. Look at the opponent's rook(s). If I can capture either, find all rook captures and and sort them by the value of the attacker in ascending order.
3. ⋮ ⋮ ⋮

Then the moves RxQ, PxB, BxR, PxQ, QxR will be found and sorted in the order PxQ, RxQ, BxR, QxR, PxB. This is not ideal! We would prefer the moves be sorted according to the *difference* in piece value (ie PxQ, RxQ, BxR, PxB, QxR), as those are the ones that "win" us the most material.

However, one of the chief advantages of MVV/LVA is that it generates moves in order *one by one*. Instead of generating a list of all captures and sorting, MVV/LVA finds the first one, searches it, then finds the second one, searches it... etc. This is nice because if (when) we hit a beta cutoff, we don't have to waste resources finding the remaining captures.

14. Extensions

If reductions decide to search uninteresting moves shallowly, extensions try to look further into interesting moves. In a relative sense, reductions and extensions are two sides of the same coin, but most modern programmers prefer reductions because they are easier to control. Nevertheless, extensions do play a role in modern engines, and there is ELO to be gained. This chapter reuses many old ideas, so it will be shorter than the rest.

14.1. The Idea

Extensions are just reductions in reverse:

```
def AB(pos, depth, alpha, beta):

    ...bla bla....

    for move in pos.legal_moves:

        ...bla bla....

        extension = how_much_to_extend(move)
        value = -AB(after_move(pos,move), depth + extension - 1, -beta, -alpha)

        ...bla bla....
```

Determine how interesting a move is, then *add* that to the depth in the recursive call. Qsearch can be understood as a sort of extension which is only invoked at leaf nodes.

Adding depth to a search can easily cause the search tree to blow up into a massive intractable mess. To work around this, extensions must be constrained in some way. This could mean depth quotas, depth limits, fractional plies, or something else entirely. The difficulty with restricting extensions is that they do not have a uniform effect on various positions. You may have a system of extensions that behaves perfectly nicely for 99 positions, but on the 100^{th} it hits a pocket of *very* interesting moves, and blows up. These exponential blow-ups tend to not respect human timescales, so the engine may as well have crashed. Extra constraints on extensions certainly diminish their effectiveness, but they are also extremely necessary.

14.2. A Zoo of Extensions

I'll now list a few common ideas for extensions. By far the most important, and the most relevant to modern chess programming are singular extensions, subsection 14.2.5.

14.2.1. Capture

The most naive extension comes directly from quiescence searching—just extend all captures. There are two main reasons one might do this:

1. Fear of a "deep horizon effect". A long sequence of moves which gives one side an advantage can be pushed across the horizon by inserting captures early in the sequence. Because these captures don't happen at the search horizon, they aren't caught by `qsearch`. Simply extending all captures solves this problem.
2. Captures are inherently interesting and deserve to be searched in depth.

The difficulty is that raw capture extensions are simply untenable. Instead of just extending captures at leaf nodes as in `qsearch`[1] we are extending *every* capture *everywhere* in the game tree. This adds a lot of garbage lines which wouldn't otherwise be searched (deeply), especially "mutual plunder" situations where the players take turns capturing the other's pieces.

In light of these mutual plunder situations, we may want to rethink point 2. It is true that (typical) captures deserved to be searched, but maybe not all of them. I'll modify it to "*normal* captures are inherently interesting and deserve to be searched in depth," where "normal capture" has some vague definition that excludes mutual plunder.

14.2.2. Recapture

If raw capture extensions don't work, perhaps we can modify the basic idea to something more tractable. Indeed we can! Instead of extending captures directly, we will extend only immediate *re-captures*—a capture is extended when the piece it is capturing captured a piece the previous turn.[2]

Since most reasonable in-game captures are trades, re-capture extensions don't freak out too much in mutual plunder situations while still addressing points 1 and 2.

14.2.3. Check

Checks are another obvious candidate for extensions for much the same reasons as captures, with the additional incentive of discovering a mate at the end of the line. Though there are generally only a few responses to checks, computers are still quite capable of giving meaningless checks which can cause deep horizon effects. These situations need to be distinguished from (genuine) perpetual check opportunities which the computer needs to recognise.

[1] which *also* needed safeguards against search blowups.
[2] TLDR when captures capture capturers. I am deep in the throes of semantic satiation.

14.2.4. One Reply Extensions

If the side to move has only one option, we should be interested in the outcome—it often leads to checkmate or other (un)desirable outcomes. It is an easy target for an extension. Beware of overlap between one-reply extensions and check extensions.

14.2.5. Singular

We've already seen three instances of a "interesting" moves in (re-)captures, checks and one-reply's, but there's a broader notion of the word that partially encompasses the previous ones and can be applied to a variety of positions.

A move is *interesting* if it is significantly better than all alternatives.

In chess lingo, moves that give your opponent only one reasonable option are called "forcing". General chess wisdom dictates that forcing moves should be calculated deeply, so why not apply the same idea to computers? This is great because it appeals to a general criteria for extensions instead of an explicit list of circumstances. Just one problem: how do you actually measure it?

Just like in move ordering and null move pruning, we are in a situation where already knowing the value of a search would help to perform the search. Just like before, the solution to this problem is to appeal to another search at a smaller depth. In this case, we will make a significantly reduced search on all other moves in order to determine whether the candidate move is .

Assuming we are using iterative deepening, it only ever makes sense to test for singularity on the hash move—any other move is likely worse, and has no hope of being "interesting" by the above definition. Furthermore, we are only concerned with whether the other moves are close enough to the top move. Then the reduced search should be performed with a null window at value(`tt_move`) - `margin`, where `margin` is how much better the `tt_move` must be in order to be considered singular.

The following code represents the basic idea of singular extensions, but *is not* how it is usually implemented.

```
...bla bla....

for move in pos.legal_moves:

    ...bla bla....

    extension = 0

    // some criteria for a move to be singular
    // in particular, it must be a ttmove
    if move == tt_move(pos) and other_criteria(move):
        sing_baseline = tt_eval(pos) // value of potential singular move
        sing_margin = sing_margin(pos) // how much better must it be?
        to_beat = sing_baseline - sing_margin // the value other moves are tested against
        sing_reduction = sing_reduction(move) // how much are we reducing?

        // finding the value of the next best move
        nextBestSoFar = -infinity
        for m in pos.legal_moves:
            if m == move:
                continue  //skip current move
            value = -AB(after_move(pos,m), depth - sing_reduction, -to_beat, -to_beat)
            nextBestSoFar = max(nextBestSoFar, value)

        // are we *that much* better than the next best?
        if nextBestSoFar + sing_margin < sing_baseline:
            extension += 1 // do the extension

    // finally, do the recursion
    value = -AB(after_move(pos,move), depth + extension - 1, -beta, -alpha)

    ...bla bla....
```

The implementation details of singular extensions can be very tricky. Indeed, one of it's (co-)inventors, Feng-hsiung Hsu[3] said

> Simple as it is, the idea of singular extensions is actually fraught with hidden implementation surprises.

I don't think it is particularly productive to discuss implementation details here other than to mention two things:

[3]Spearheaded computer chess from the late 80's and 90's. Lead author of Deep Though. Big deal.

1. The `other_criterion(move)` line usually contains several other things—node type, depth, etc.

2. Instead of the somewhat unwieldy pseudo-code written above, most programs will perform a *single* recursive search on the same position with a parameter that says "don't search this move." This does functionally the same thing as the other bit of code.

15. How to Trick an Engine

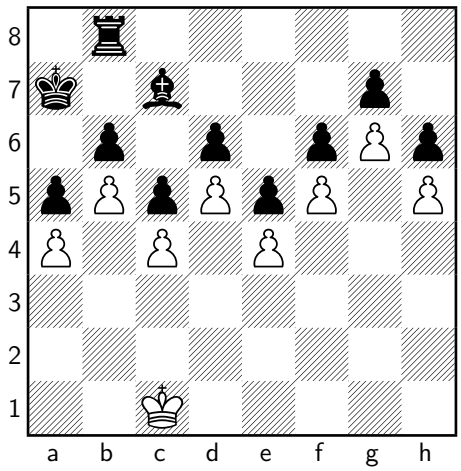
Modern computers are, for all intents and purposes, gods of chess, but they are not omnipotent. In this chapter, I will display a few instructive instances of this, and explain what's going on.

Since my readers exist in the (possibly distant) future, I will mention that these types of things don't age well. In my research, I've found many instances (some as recent as 2015) of someone saying "everyone look at this position. My computer can't solve it!" I would then feed that position into Stockfish and have the correct move in two seconds. I would not be surprised if in a few years you will be able to do much the same to my examples. For purposes of your historical curiosity: this book was written in various parts of 2019, 2020 and 2021, and I am using Stockfish 10 for my analysis.

15.1. A Few Puzzles

15.1.1. A Lazy Example

I occasionally see the following type of position on various internet chess forums:



Perhaps accompanied by "lolz enginez r stoopid. it tinks black iz winning."

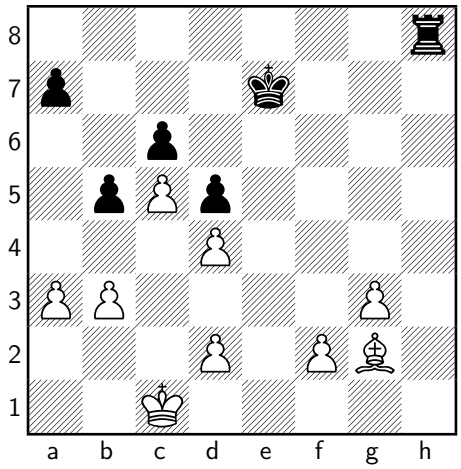
Stockfish gives me -9.9. Why? Because of course it does. It has no way of seeing that despite his enormous material advantage, Black has no way to make progress. By its very nature, the evaluation function can't figure this out. Making progress is a job for the search procedure, which can only conclude the position is a draw after 50 moves (100 ply), something far out of the reach of modern hardware.

I dislike touting these types of positions. It's equivalent to taunting a child for not being able to reach your king as you're holding it over your head. Oh, and the kid is rated 3500.

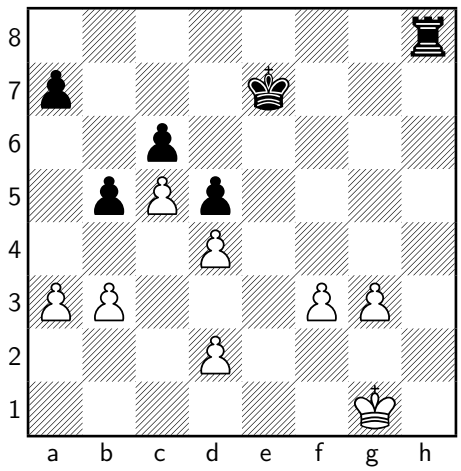
Although it is a bit of an annoying example, it does illustrate the point that in some contrived situations, human judgment can overrule engines.

15.1.2. No Progress to be Made

But do they all have to be so contrived? Oh no. Let's test your endgame form.[1] White to move:[2]



This isn't a chess chapter, so I won't go into the gory details, but the correct continuation is 1.Kd1 Rh2 2.Ke1 Rxg2 3.Kf1 Rh2 4.Kg1 Rh8 5.f3! which results in the following position:



[1] This is *very* difficult.

[2] This study is due to chess composer Vitaly Chekhover

As long as the rook is on the h file, White will move her king back and forth between g1 and g2. If the rook is on the e file, she will oscillate between f1 and f2. White's pawns prevent infiltration from the black king, and the queenside pawns can always lock up the black pawns should they advance.

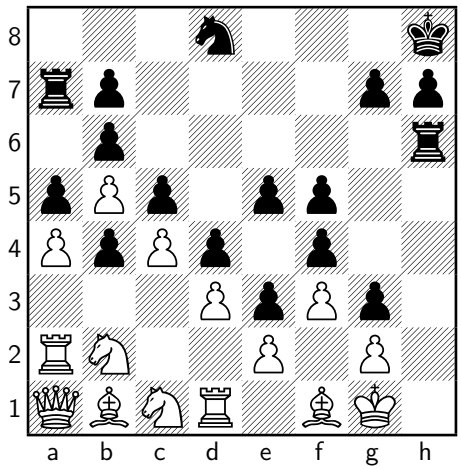
I've omited some details, but the short version is White can hold this position indefinitely, making the game a draw.

Despite this, the Stockfish suggests 1.Bf3 and gives black a 170 centipawn advantage. After capturing the bisohp in the correct line, that number jumps to 400. The computer can't comprehend the idea that giving up a piece allows white to stall indefinitely—it can't see that far.

Unlike the previous position, however, the "no progress" idea is very important in calculation—whether or not you spot it changes your decision making, and in this case affects whether you draw or lose. Knowledge of these types of "fortresses" is one of the areas computers always have (and always will?) struggle with.

15.1.3. Too Many Quiet Moves

Moving back to an extremely contrived example, see if you can spot a mate for Black:



Hopefully you got it. Black spends 7 (!!) moves getting both his rooks on the h file, then delivers checkmate on h1. True to form, the computer gives White a 630 centipawn advantage *at depth 40*.

This position is different from the previous two examples in at least two respects:

1. There *is* progress to be made. If Black just shuffles his king back and forth, white can sacrifice some material to crack open the black pawns. For instance, Nb3 Bc2 Qc1 Na1 Bb1 Nc2 Nxb4, after which White plays a5 or c5. Of course, this is all too slow.

2. There *actually is* a forcing mating sequence, which the computer just misses. This isn't a case of computers not being able to "understand the position"—there's a straightforward tactic, and the computer can't find it.

Furthermore, this is a *mate in 8*, (ie 15 ply out) which the still can't see at *depth 40*. What's going on here?

Hopefully you understand by now that the "depth" does not indicate a level of the search tree as in chapter 4. Rather it is a parameter which roughly indicated how far we are looking ahead. Some combination of null moves, quiescence searches, late move reductions and extensions can make a depth n search look at a particular branch *much* deeper or *much* shallower than n.

This is precisely what is happening in this position. The solution requires a series of quiet (boring) moves which, to a computer, don't do much to distinguish themselves from other quiet moves. In other words, the correct moves are perfect candidates for reductions.

Because there are 7 of these quiet moves in a row, the reductions build on each other. Stockfish has very aggressive selectivity, which is why it to fails to see the line all the way through, even though it less than half the nominal "depth".
