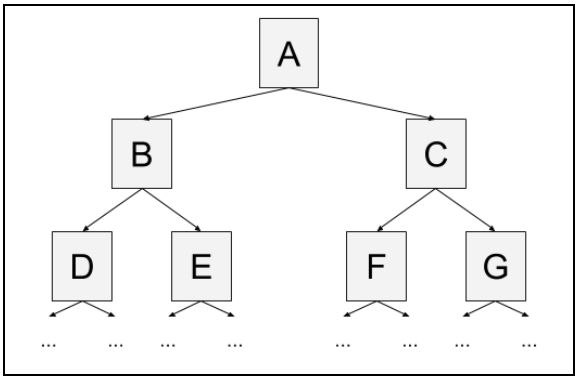
NegaMax Chess Engine

1. Tree Search

Developing an effective AI algorithm for chess requires the use of a tree based search. A tree, in simple terms, is a collection of nodes which are connected in a pyramid-like structure. Despite the term being described as this, it is often illustrated with the opposite: with a “home” node at the top with some branch nodes, and so forth until terminal nodes are reached.

Below is a simple illustration of a tree structure:



In this example, a “home” node of A leads to two branches B and C, which further lead to two branches, and so on. This tree is said to have a branching factor of 2.

Chess, comparatively, requires many more nodes, too many, in fact, to illustrate in a diagram. On average, any given board state has 31 branches, or chess has an average branching factor of 35.

The branching factor affects the complexity as the number of nodes within the tree of depth d is b^d, where b is the branching factor and d is the depth. Consider the below table showing the number of nodes per branching factor and depth:

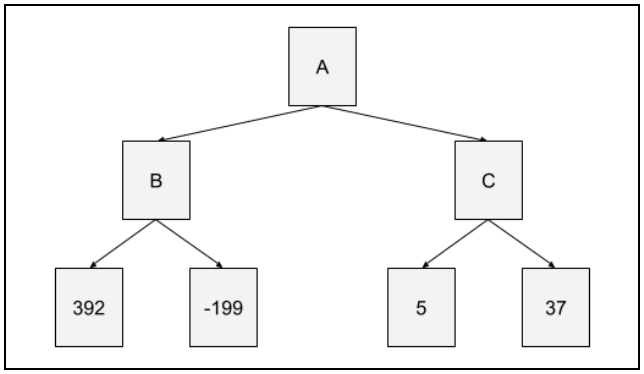
B. Factor	**d = 1**	**d = 2**	**d = 3**	**d = 4**	**d = 5**
2	2	4	8	16	32
3	3	9	27	81	243
7	7	49	343	2401	$>16 \times 10^3$
12	12	144	1728	$>20 \times 10^3$	$>24 \times 10^5$
31	31	961	$>29 \times 10^3$	$>92 \times 10^4$	$>28 \times 10^6$

As evidenced, a larger branching factor has exponential growth. Chess is an extremely computationally taxing game because the branching factor is so large.

1.1. Negamax and Minimax

The Minimax algorithm is a technique of tree traversal utilizing the board evaluation function. Each terminal node of the tree is given a value, and the purpose of this algorithm is to maximize the player's score.

Consider a tree of depth 2:



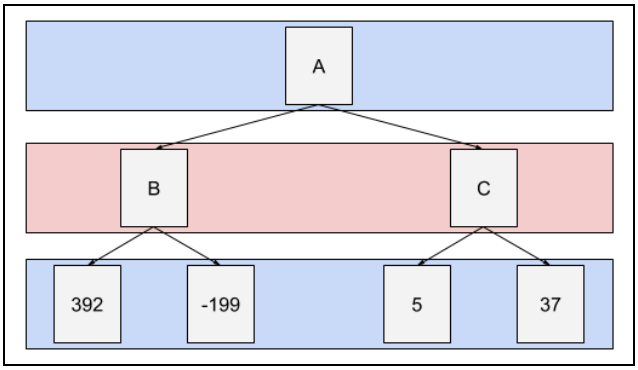
Each terminal node was assigned a score as defined by the board evaluation function. But how is the "home" node value determined?

This algorithm allows a higher order node score to be evaluated in such a way that maximizes the value of the home node based on the contents of the branch nodes as well as predictable behavior of each of the two players. In practice, this algorithm allows the AI to "see ahead" in future game states and choose a move that increases its score.

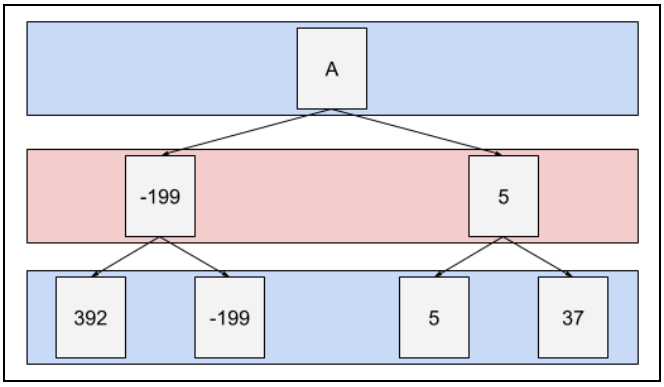
Pseudocode for the algorithm is below:

```
function minimax(node, depth, player):
  if depth = 0:
    return eval(node)
  if player = minimizingplayer:
    v := -inf
    foreach branch of node:
      v := max(v, minimax(branch, depth-1, minimizingplayer)
  if player = minimizingplayer:
    v := +inf
    foreach branch of node:
      v := min(v, minimax(branch, depth-1, maximizingplayer)
  return v
```

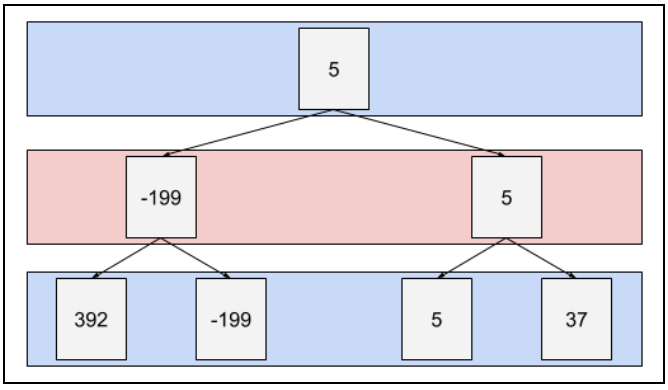
The goal of the algorithm is to choose nodes such that the minimizing player has the lowest score while the maximizing player has the highest. Using the same example as before:



The nodes colored blue will be maximized between each pair and the nodes colored red will be minimized between each pair. As the terminal nodes already have scores, the algorithm begins at the next node upwards. In this case, the B node will choose the minimum between each branch which it determines is -199. Likewise, node C will choose 5. Thus the values of nodes B and C are found:



Now, node A's value can be determined by choosing a maximum between branch nodes. In this case, 5 is chosen:



Using the Minimax algorithm, all nodes now have values ascribed to them. For this example, the maximizing player would choose node C as their move, as it ensures the outcome at a terminal depth is highest.

Were it to choose B, whose branch nodes contain the highest possible move for the player, since the algorithm presupposes each player makes optimal moves, the opponent would choose a move which is worse overall.

Recall the pseudocode for the Minimax algorithm from before: this can be reduced into a Negamax algorithm. Negamax has equivalent complexity as Minimax although rather than requiring a check for two players, considers only one player. Negamax relies on the below property:

$$max(a,\ b)\ =\ -min(-a,\ -b)$$

Relying on this maxim, the Minimax algorithm can be rewritten as Negamax:

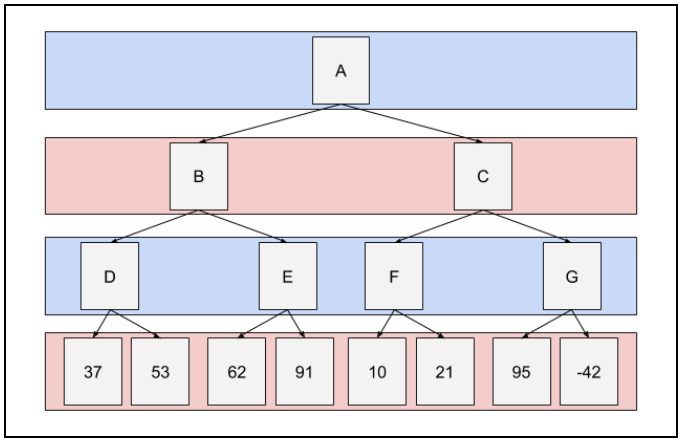
```
function negamax(node, depth, player):
  if depth = 0:
    return player * eval(node)
    v := -inf
    foreach branch of node:
      v := max(v, -negamax(branch, depth-1, -player)
  return v
```

This algorithm is player agnostic in the sense that it is assumed the caller of the function is the maximizing player from before. Since each recursive step alternates between a positive and negative evaluation, the algorithm is much more concise yet equivalent to the Minimax implementation. There may be merit in using Negamax over Minimax, however, as fewer translation units are needed for a compiled object file.

The crux of this algorithm is to recursively explore game states and make decisions based on terminal game conditions. The algorithm effectively choose optimal moves for the AI based on this information. However, this method is exhaustive and relatively computationally expensive; there are methods to reduce the complexity of the Negamax algorithm.

1.1.1. Alpha-Beta Pruning

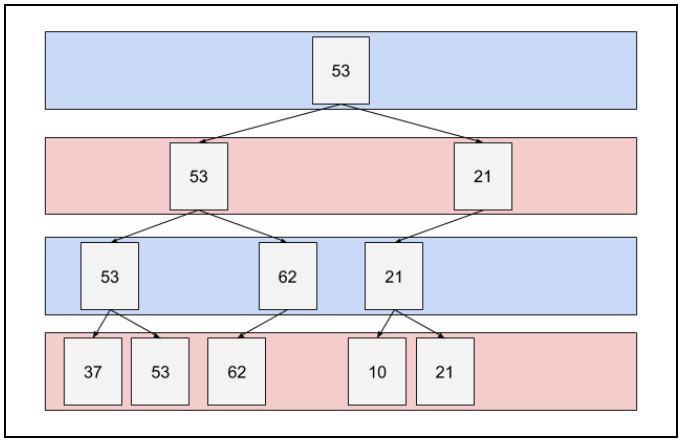
Alpha-Beta pruning is an alteration of the tree search and evaluation previously discussed: it allows for some branches to be “pruned” as they may be deemed redundant in the search. For example, consider the below tree:



Many things can be implicitly conferred about this tree: some branches do not affect the decision of a maximizing or minimizing player as nodes already explored provide a value that is assuredly better than an unvisited node.

In this example, node D would have a maximized value of 53, and upon searching node E’s first branch node, 62 is found. Since the minimizing player would have chosen the minimized value of its branch nodes, it is assured that node D is an optimal choice over either of node E’s branch nodes. Therefore node E’s second branch does not need to be explored and is culled.

Similarly, for node C, the same occurs, however the culled branch is much larger. The resultant tree structure would appear as below:



This operation ensures at the very least the tree search is equivalent versus the vanilla algorithm but is oftentimes better. Consider a best case scenario as above: of fourteen branch nodes, four were culled for a reduction of 29%.

For alpha-beta pruning to be implemented, the search algorithm needs to be adjusted; for the Minimax algorithm, pseudocode for an alpha-beta implementation is below:

```
function minimax(node, depth, α, β, player):
  if depth = 0:
    return eval(node)
  if player = minimizingplayer:
    v := -inf
    foreach branch of node:
      v := max(v, minimax(branch, depth-1, α, β, minimizingplayer)
      α := max(α, v)
      if α >= β: break
  if player = minimizingplayer:
    v := +inf
    foreach branch of node:
      v := min(v, minimax(branch, depth-1, α, β, maximizingplayer)
      β := min(β, v)
      if α >= β: break
  return v
```

Where both alpha and beta are assured best moves for each of the maximizing and minimizing players respectively. This new algorithm keeps track of best values as found in prior nodes and were a value to be worse than a previous higher node, the algorithm leaves its recursive state for that stack frame and will ignore further branch nodes.

As before, the algorithm can be rewritten based on the Negamax maxim as discussed earlier:

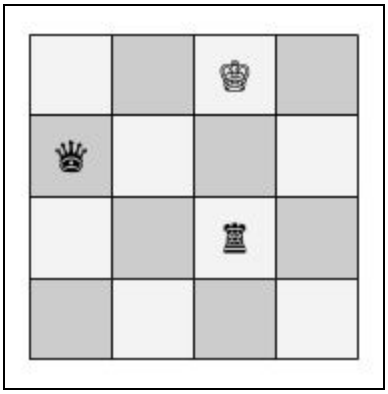
```
function negamax(node, depth, α, β, player):
  if depth = 0:
    return player * eval(node)
    v := -inf
    foreach branch of node:
      v := max(v, -negamax(branch, depth-1, -β, -α, -player))
      α := max(α, v)
      if α >= β: break
  return v
```

This again simplifies the algorithm while maintaining the computational cost and outcome.

1.1.2. Other Pruning

In addition to alpha-beta pruning as described in the previous section, another pruning technique is employed, although with not as much of an impact. In remaining within the rules of chess, it is not possible to perform move reduplication. Not only is it unsportsmanlike but also may result in a forced draw were move reduplication be the only move a player may make.

Consider a board state as the below:

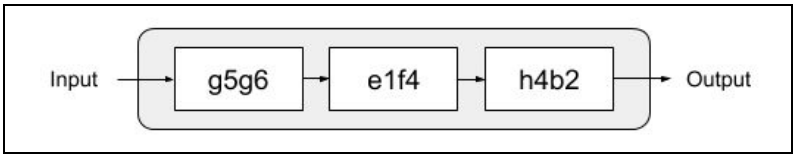


The Negamax algorithm will recognize that white has precisely one move which it can perform. However, were white to move king east one tile, black has many opportunities to force white king to return to its original position. Likewise, black can simply move a piece to return to the original board configuration.

For Negamax, this poses a problem as if white has only one possible move and black were to minimize white's value on the board, it becomes a cycle of move reduplications with no end. As white tries to leave check, black tries to force a check, which makes white try to leave check ad infinitum.

In situations as these, a move buffer is necessary to keep the history of moves within the AI algorithm. A buffer solves two challenges: preventing threefold repetition (where a draw is concluded if each player performs the same three moves in sequence) and infinite cycles where the only available move leads to a repetition of move reduplications.

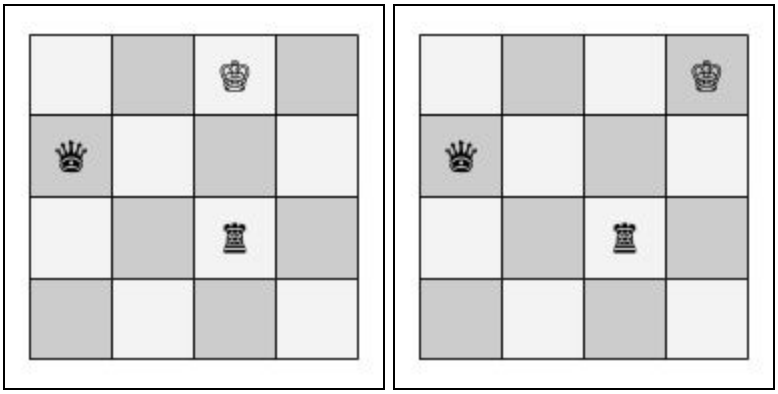
A buffer can be visualized as a fixed size queue of a specified length:



In the prior figure, a buffer of length three is shown. When the AI algorithm devises a new move, it is compared against the buffer and determines whether it is a valid move or not.

Initially the buffer is empty, but is populated by moves as the AI generates them. If the buffer is full, no new moves can be inserted until that move is compared against each existing move in the buffer. If a new move exists within the buffer, it is an invalid move and another move, if possible, is chosen instead. If the proposed move is not within the buffer already, the buffer is evacuated in a first-in-first-out fashion to make room for the newest move.

Consider a sequence of board states as below:



This move is stored in the buffer and using the example buffer of length three, this same move is not possible until two more white moves has occurred. This presupposes the buffer is full, of course, as the check for move reduplication is only for move sequences of three, which is every board state except the first three.

Using the ongoing example, if the only available move were to be a duplicated of one recent, the game ends in a stalemate. This method prevents threefold repetition and infinite cycles where a lone move (or few moves) are duplicated as they are the only possible moves a player may make.

Ultimately, this type of pruning has a small effect on overall performance and may have negligible impact on earlier board configurations but it is not trivial. The utility in this operation is worth more than the cost saving.

2. Board Evaluation

Paramount to effective chess artificial intelligence is the heuristic function employed to evaluate a game's board configurations. While there are many techniques to evaluate a game board, heuristics must be chosen such that the below are considered:

- Material Value
  - "Will I lose a piece if I make this move? Will I capture an opponent's piece? Does this move leave my pieces open to attack?"
  - dependent on total pieces on board and their values
- Mobility Value
  - "Will this move offer my other pieces more freedom of movement? Will it restrict my opponent's movement?"
  - dependent on total tiles available to be moved to in successive turns
- Pawn Value
  - "Will this move progress my pawns closer to promotion? Will it prevent my opponent from doing the same?"
  - dependent on distance pawns are from initial conditions

In effect, three heuristic evaluation functions need to be employed and weighed such that a balance is found between them. This application uses the general equation below:

$$h = c_1 * v + c_2 * m + c_3 * p$$

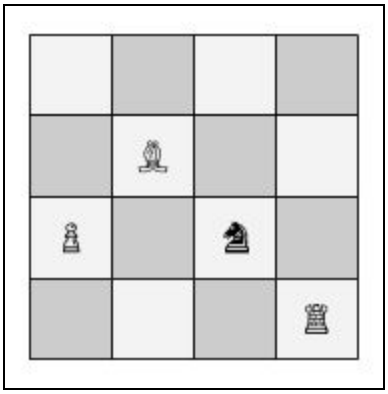
Where h is the heuristic score, v is the material value of the board, m is the mobility value of the board, and p is the pawn value of the board. Each of these variables has multiplicand c which is some coefficient found empirically.

As stated, material is the value of a given piece and remains constant throughout the piece's lifespan. A table of pieces and their values is below:

Piece	Value
Pawn	1
Knight	3
Bishop	3
Rook	5
Queen	9
King	100

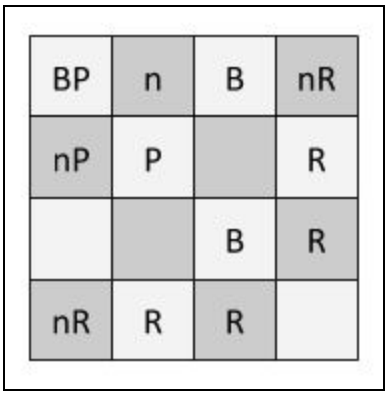
For each piece, from the view of a given player, the opposing player piece values is seen as negative. Therefore capturing a piece has a positive impact on the board's evaluation while losing a piece has the reverse effect.

Consider the below board state:



In tabulating the piece values of each piece, the material value of this board state is found to be 10 from the perspective of white.

Mobility concerns itself with the number of moves available to all pieces. Using the same board as above, the moves can be identified:



Where an uppercase letter corresponds to a white move and a lowercase to a black move, "B" is bishop, "N" is knight, "R" is rook, and "P" is pawn. In tabulating the score of this board's mobility from white's perspective, a summation of all white moves less a summation of all black moves is performed: the mobility value of this board is 7.

The board state's pawn value is simply the rank of the pawn. A rank is whichever row that pawn is respective to the row closest to that player. For example, a white pawn on **a6** has a rank of six, whereas a black pawn on **b2** has a rank of seven. In

the ongoing example, assuming this is the complete board, the board's pawn value evaluates to two.

The behavior of the artificial intelligence will depend entirely on the ratio of these values, hence why the coefficients are necessary. Consider some conditions below:

c_1	c_2	c_3	**Behavior**
10	0	0	Will prioritize keeping and capturing pieces
0	10	0	Will prioritize maintaining board control (maximizing movement of pieces)
0	0	10	Will prioritize moving pawns, not necessarily promoting them
0	5	5	Will try to move pawns out of the way of other pieces, or block opponent
5	0	5	Will prioritize moving pawns and promoting them

There is no way to determine idealized coefficients other than through experimentation and deduction based on expected behavior.

There are also coefficient choices which are objectively poor. Consider a negative coefficient for material value: an agent using this heuristic will actively try to lose pieces. And a negative pawn value coefficient may prevent moving pawns at all, forcing the agent to use only their knights.

Through empirical means, the below heuristic is found to be a good contender for a competent artificial intelligence:

$$h = 15 * v + 3 * m + 6 * p$$

In testing, the material coefficient should be nonzero and largest among coefficients, mobility coefficient should be much smaller in comparison, and the pawn value coefficient should be larger than the mobility coefficient.

The heuristic equation is also reducible as it is a ratio. The same heuristic would be found using values divisible by another for each coefficient.

3. Chess Implementation

In designing an effective chess engine, outside of the scope of AI, the rules of chess needs to be implemented strongly: in other words, no illegal board state can be achieved. To accomplish this, a few things need to be considered:

- designing an effect chessboard
    - ensuring constraints to board space are upheld at all times
    - ensuring out of bound instances are impossible
    - effective use of containers and object orientation to put piece to board efficiently
- handling movement correctly
    - preventing illegal moves
    - performing piece moves as described in the rules of chess
    - handling special moves (castling, promotion, etc)
- handling terminal board states
    - checkmate, stalemate, etc

This section will deal only with the physical aspects of the chess game rather than the artificial intelligence component and the methods employed within it.

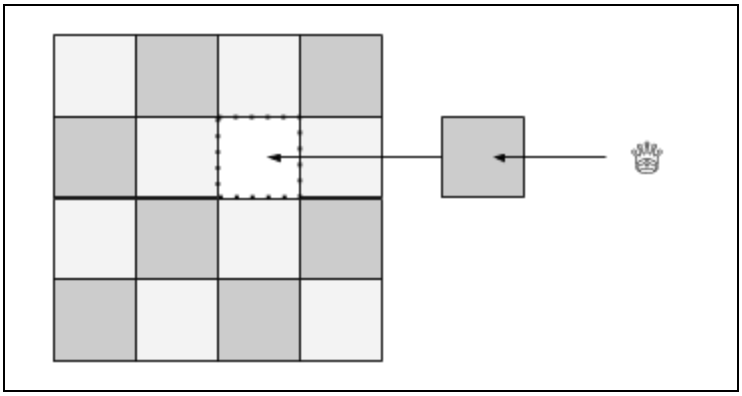
3.1. Board Representation

A chessboard is an 8x8 grid of tiles in which pieces are placed. A tile can either contain a piece or no piece. It is useful to compartmentalize the board into separate aspects as it makes for easier description from a pragmatic standpoint.

For this chess engine, the `Board` is a collection of `Tiles`. A board strictly interacts with the tiles within it but may access the contents of a `Tile` which may contain other aspects of the application (as an example, a `Piece`).

Initially, the game board is a series of blank tiles pointing to nothing, but as a `Board` is instantiated, `Tiles` become occupied by `Pieces` as defined by the type of chess game. This application allows a default standard `Board` as well as custom `Boards` as user defined.

Consider the below figure:



In this, an empty tile on a board is instantiated with a `Tile` which is then instantiated with a `Piece` within it. A `Tile` is effectively just a container for `Piece` although it also contains logic specific to handling the `Piece`.

For example, at multiple points in the chess engine, a check for occupancy is required (for example, capturing and moving pieces). This is the crux of defining a legal move versus an illegal one. Most of the checks against illegal moves or board states boil down to whether or not a piece is in valid position is paramount

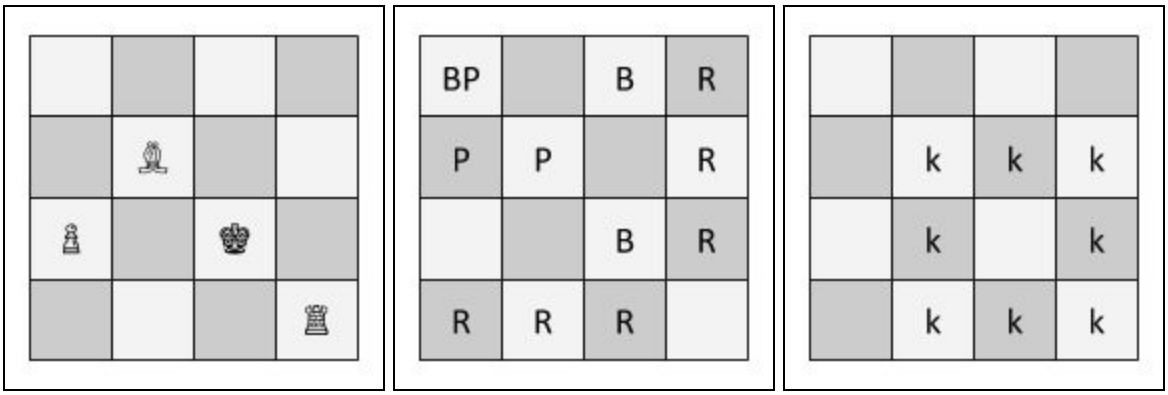
3.2. Movement

Movement in chess is defined by piece. Each piece may move in any direction within its own move constraints and provided that move is an illegal move. Every piece has varying movement behavior. Below is a table summarizing them in terms of *x* and *y* co-ordinates:

Piece	***(x, y)***
Pawn	(0, +1) (0, +2) (first move) (±1, +1) (capturing)
Knight	(±2, ±1) (±1, ±2)
Bishop	(±n, ±n) (diagonal)
Rook	(±n, 0) (west/east) (0, ±n) (south/north)
Queen	(±n, ±n) (diagonal) (±n, 0) (west/east) (0, ±n) (south/north)
King	(±1, ±1) (diagonal) (±1, 0) (west/east) (0, ±1) (south/north)

Knowing these move constraints, legal moves can be defined as a measure of distance in both row and columnar form. Provided a move is within these ranges and the `Board` agrees with the move, it is a viable move with the exception of the king piece.

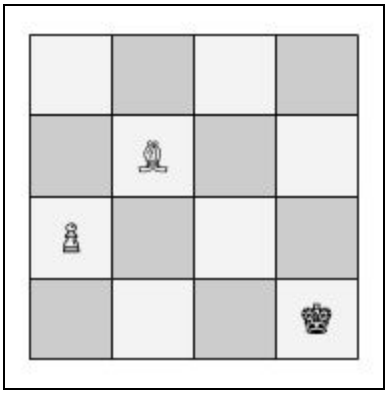
Consider the game state below and the moves every piece can make:



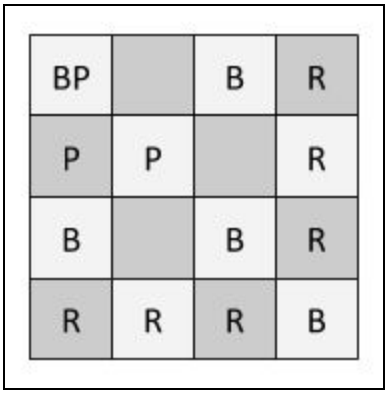
In the prior diagrams, white moves are denoted by an uppercase letter and black moves by lowercase letters. “P” for pawn, “B” for bishop, “K” for king, and “R” for rook.

For this example, wherever there is overlap between white and black moves is move the king can make, putting itself into check. For this reason, these moves are removed from the viable moves that king can make.

However, some consideration needs to be given for the *paths* a piece could make were it not obstructed. Consider in the prior example were the king to capture the rook:



Despite the king’s new position not being a viable move for the bishop, the resultant board state is illegal: a king may not move into check. Rather than determine a valid move by other piece’s moves, their paths must also be taken into consideration. Here is the paths all white pieces could make:



This reduces the king’s viable moves and best illustrates legal moves for that king.

3.3. Check, Checkmate, Stalemate

Special consideration needs to be made for board states which result in a check, checkmate, or stalemate, as the rules for typical movement behavior changes.

When in check, the king is in threat of being captured. Capturing a king is not a legal chess movement and it is convention to remove the king from check, either by blocking the offending piece from capturing the king or moving the king out of check.

The chess engine handles this by investigating the board state in the future after a proposed move was made. Examining the moves and paths of other pieces, a viable move list can be generated.

Likewise similar methods are used for stalemate and checkmate: in a checkmated game board, the total number of possible moves between the king and other pieces is examined. Provided no moves exist to leave check for a king, the game board is determined to be in checkmate and the game ends.

Stalemate is not dissimilar, however it differs in that the king is not in check: if the total number of moves a king may make as well as other pieces may make is zero, yet there is no check condition, the game ends in a draw as stalemate dictates.

4. References

Russell, Stuart J.; Norvig, Peter (2010). *Artificial Intelligence: A Modern Approach* (3rd ed.). Upper Saddle River, New Jersey: Pearson Education, Inc. p. 167.

Edwards, D.J.; Hart, T.P. (4 December 1961). "The Alpha–beta Heuristic (AIM-030)". Massachusetts Institute of Technology.

George T. Heineman; Gary Pollice & Stanley Selkow (2008). "Chapter 7: Path Finding in AI". *Algorithms in a Nutshell*. Oreilly Media. pp. 213–217.

Omid David, Jaap van den Herik, Moshe Koppel, Nathan S. Netanyahu (2009). *Simulating Human Grandmasters: Evolution and Coevolution of Evaluation Functions*. ACM Genetic and Evolutionary Computation Conference (GECCO '09), pp. 1483 - 1489.

Peter W. Frey (1985). *An Empirical Technique for Developing Evaluation Functions*. ICCA Journal, Vol. 8, No. 1.

David Slate, Larry Atkin (1977). *CHESS 4.5 - The Northwestern University Chess Program.* Chess Skill in Man and Machine (ed. Peter W. Frey), pp. 82-118. Springer-Verlag, New York, N.Y. 2nd ed. 1983.

Alan Turing (1953). *Chess*. part of the collection *Digital Computers Applied to Games*, Bertram Vivian Bowden, reprinted 1988.