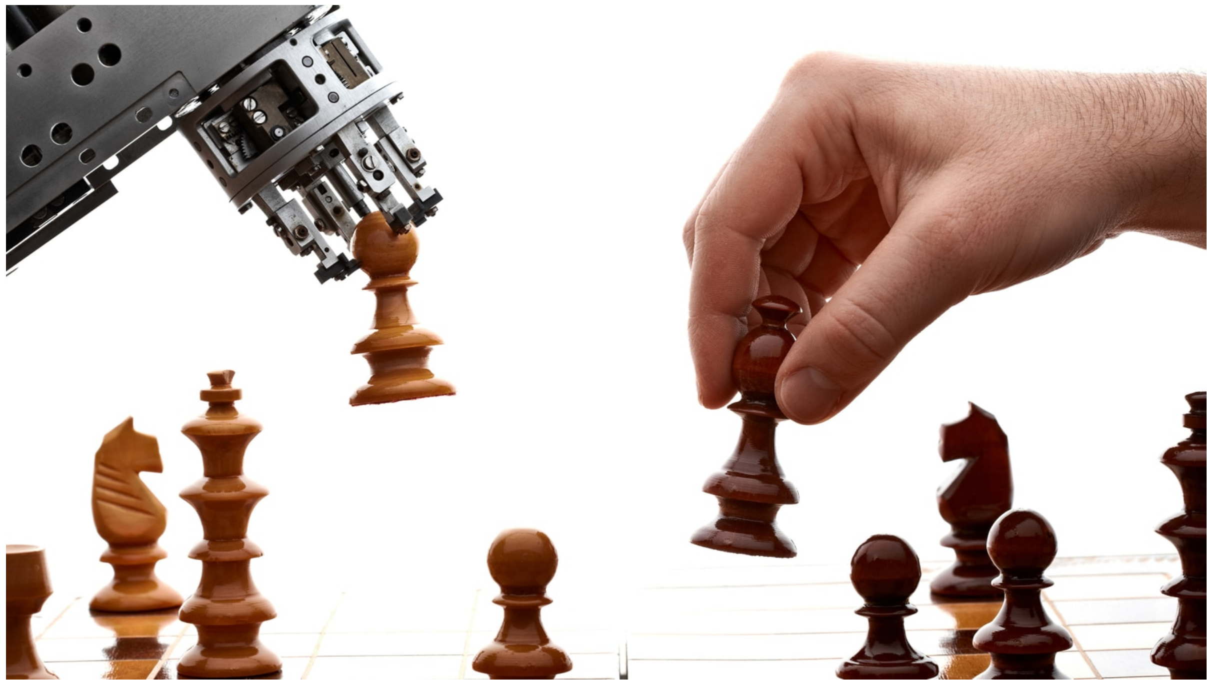
Fundamentals of Game AI



1. **Why use game AI?**

   In term projects, Game AI is used to create a computer player for specific types of games. These algorithms usually need to meet most or all of the criteria below:

   - Turn-based (these algorithms all require this)
   - 2-player
   - Zero-sum (i.e. actions that help player 1 hurt player 2)
   - Deterministic (i.e. no randomness, like dice rolls)
   - Perfect information (i.e. no information is hidden from either player)

   The main game AI algorithm is called **minimax** and only really works when all of those criteria are met. Modifications and/or similar algorithms also work well when some of those criteria are not met (i.e. *some* games of chance).

2. **Key Utilities**

The game AI algorithms usually require you to write several key utility functions. The ones listed below are the ones required for minimax, but the other ones are quite similar:

- You must have some consistent way of representing the current game state (i.e. the current "board", whose turn it is, etc.)
- A way to obtain all of the moves that the current player can make from the current state
- A way to obtain the new state caused by making a certain move (**this should almost always be nondestructive**)
- A way to know if the game is over, and if so, who won (or if there is a tie)
- A *heuristic function* that assigns a score to a state (more on this later)

You should try to keep app out of these functions, because they need to be able to work on arbitrary game states, not just the current game state of the animation.

Examples of each of these parts will be shown in the case studies in subsequent pages.

3. **Heuristics**

The problem is that no algorithm can figure out what move is *truly* the best without exploring all or almost all of the future game outcomes, potentially dozens or hundreds of moves into the future. This is computationally impractical, so most of these algorithms instead only look a few moves into the future and then use a **heuristic function** to assign a score to the state.

The heuristic function $h(s)$ should usually have the following properties:

- If s is a state where the player has won, $h(s) = +\infty$ (or is a really big positive number)
- If s is a state where the player has won, $h(s) = -\infty$ (or is a really big negative number)
- If s is a state where the game is tied, $h(s) = 0$
- If state s_1 is better for the player than state s_2, then $h(s_1)$ should ideally be $> h(s_2)$

The first 3 can be done by checking if the game is over (and if so, who won). As for the rest, that is often done by using several scoring metrics and taking a weighted average.

For example, if we are playing a game of Othello, the metrics could be:
$f_1(s) =$ the number of pieces the player has on the board
$f_2(s) =$ the number of moves the player can make
$f_3(s) =$ the number of corner pieces the player controls
And the following is one possible heuristic function:

$$h(s) = \begin{cases} +\infty & won \\ 0 & tied \\ -\infty & lost \\ f_1(s) + 3f_2(s) + 20f_3(s) & otherwise \end{cases}$$

Design of the heuristic function is often important to the speed and performance of the game AI. If it is too simple, the AI may not play well, and if it is too complex, then the AI may be too slow to be of use. For any given game, there could be hundreds of possible scoring metric functions you could write. Figuring these out is up to you. Research on the theory of your particular game may provide some insights.

Also, MVP definitions *often* require you to have at least one scoring metric in the heuristic that is *somewhat* complex.

4. **Algorithms**

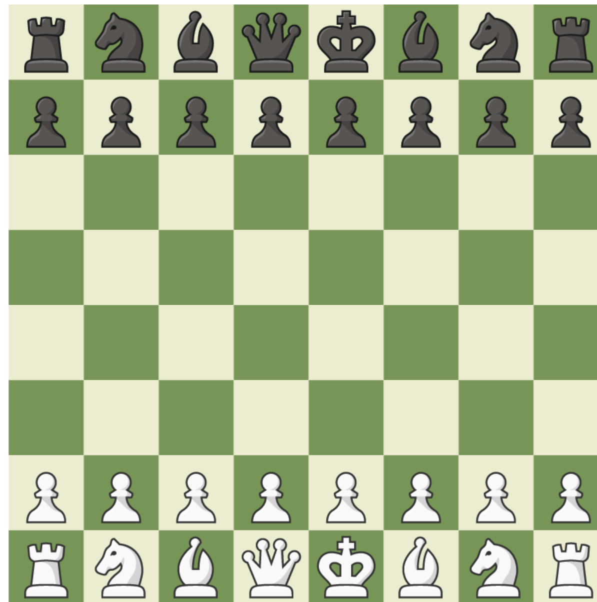
There are three major game AI algorithms worth considering (though there are other hybrids):

Minimax works best for games that meet the criteria mentioned earlier quite closely. The core idea of minimax is that it uses recursion to simulate future game states. In the base case (game over, or the target depth is reched), the heuristic function is called. In the recursive case, if it is your turn it picks the move/score that resulted in the highest score from the recursive call, and if its your opponent's turn it picks the one with the lowest score.

Expectimax is quite similar to minimax, but also allows for some elements of chance. In the recursive case, it has a third option instead of just your turn vs opponent's turn: a chance node. This is where the moves are all the possible outcomes of a random event (i.e. the roll of some dice). Instead of picking the best/worst recursive call outcome, you take a weighted average of all the recursive calls (weighted by the probability).

Monte Carlo is often used when there are too many moves and/or too much randomness for minimax/-expectimax to handle well. The Monte Carlo algorithm is based on making many random simulations and trying to make an informed decision based on those many simulations. Often it includes algorithmic tricks to maximize how much of the game tree is "explored", i.e. increasing the probability that a move will be randomly chosen if the resulting game state has not been simulated yet (these versions are called **Monte Carlo Tree Search**).

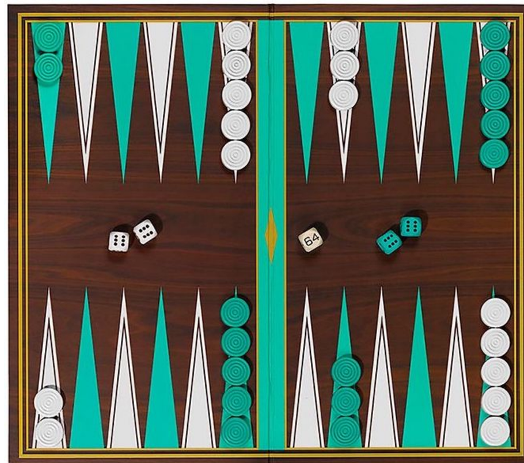
5. **Case Study: Chess**



Chess is the classical example of a game that can have a minimax AI. It meets all of the criteria (2 person, turn-based, deterministic, perfect information, zero-sum). The key utilities can be written as follows (there are other ways to write these):

(a) State: the board as a 2D list of strings, and a string saying whose turn it is (maybe also a list of captured pieces).
(b) Possibles moves: returns a list of (startRow, startCol, endRow, endCol) tuples indicating each move that each piece belonging to the current player can make.
(c) New state: nondestructively updates the board to reflect the moved piece and switches the turn.
(d) Game over: a function to detect checkmate
(e) Heuristic: could include metrics such as the number of pieces you have (weighted by type), whether you are in check, the number of "pins" you have, etc.

6. **Case Study: Backgammon**



Backgammon is a game that is more suited to expectimax. It meets all of the criteria for minimax, except that there is an element of chance; the legal moves for the player at each turn depend on the outcome of rolling two dice.

This can be handled by having a chance node before each player's turn, where the chance node's possible outcomes are all the possible dice rolls. The outcome of the dice roll is then stored in the state, allowing the actual player/opponent turn to make decisions based on the dice roll.

7. **Case Study: Battleship**

Battleship is an example of a game that is more suited to Monte Carlo approach (not necessarily the Monte carlo Tree Search). The criteria it violates is that it does **not** have perfect information; you do not know where your enemy's ships are and they do not know where yours are.

One approach to this would be to use some kind of recursive backtracker to simulate where the enemy's ships could be located based on the red/white pins pins, repeat several times, and then choose a location to fire on based on which location contained a ship in most of the simulations.

8. **Case Study: Risk**

Risk is an example of a game that is more suited to the Monte Carlo Tree Search. It has a great deal of chance and many possible moves at each turn. The likely aproach for writing an AI would be having it do many random simulations of the game and picking whichever of the current moves resulted in the best outcome on average.

The advantage to this approach is that its possible to write the random simulations in a way that they explore the potential *long term* outcomes of a move on the game by playing out the game perhaps dozens of moves into the future.

9. **Minimax Pseudocode**

Below is the pseudocode for minimax, which uses the utilities mentioned earlier. This version returns the best move and the associated score.

```
define minimax(state, depth, maximizing):
    If depth = 0 or the state is a game-over state:
        Return the score of the state
    If maximizing:
        Best score = −∞
        Best move = dummy value
        For each move from the current state:
            Obtain the new state caused by making the move
            Get the score of the new state by calling minimax:
                Use the new state as the state
                Decrease the depth by 1
                Maximizing boolean is now false
            If the score is better than the best score:
                Update the best score/move
        Return best score and best move
    Otherwise:
        Best score = +∞
        Best move = dummy value
        For each move from the current state:
            Obtain the new state caused by making the move
            Get the score of the new state by calling minimax:
                Use the new state as the state
                Decrease the depth by 1
                Maximizing boolean is now true
            If the score is worse than the best score:
                Update the best score/move
        Return best score and best move
```

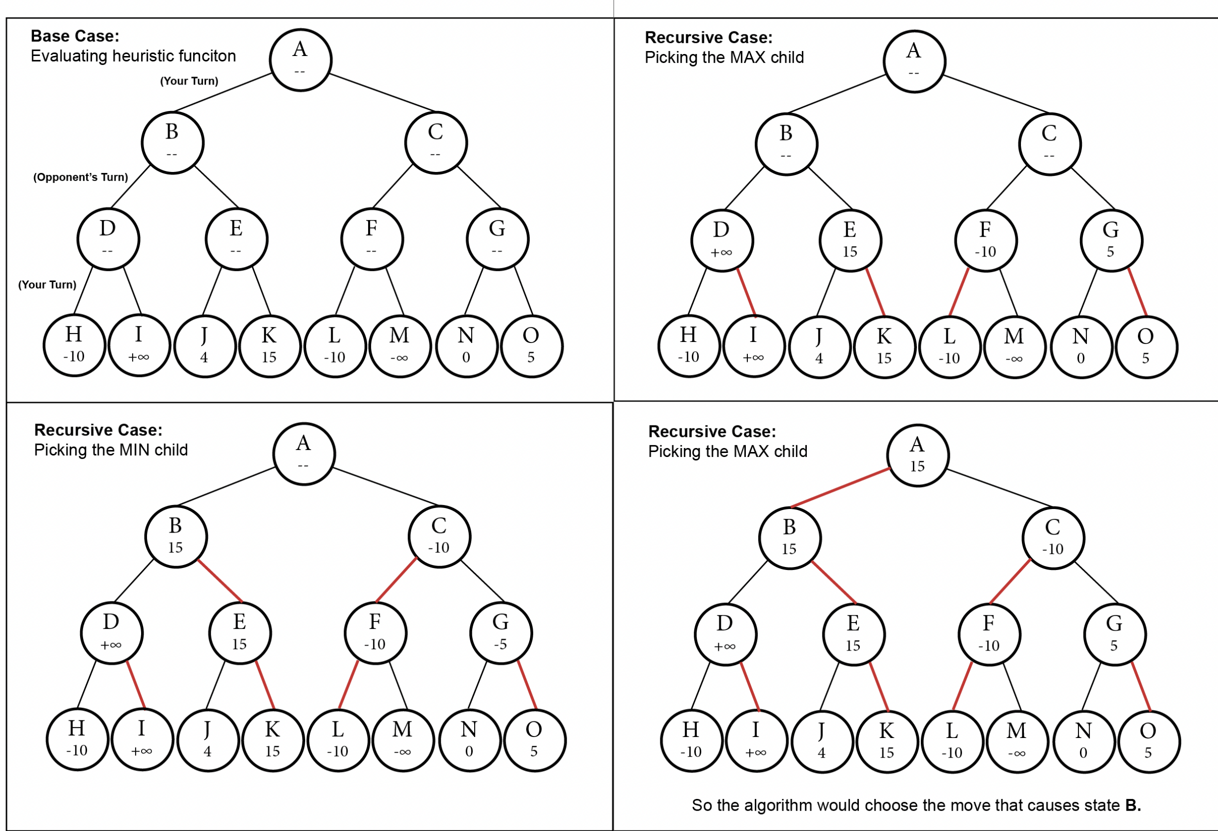
To modify for expectimax, before checking if its maximizing or not you would check if it is a chance node. If so, you would loop over all the moves, obtain hte new state and its score from the recursive call, then take the average of all those scores instead of the min or max.

10. Minimax Example

Below is an example of Minimax choosing the best move in a game. The game states are represented as a "tree" where the "root" (at the top) is the current game state and "leaves" (at the bottom) are the game states that we call the heuristic function on.

This example has 3 layers of recursive calls before reaching the base case, and thus goes through 2 turns of maximizing and 1 turns of minimizing the score.

For the purposes of this visual, recursive calls are evaluated simultaneously, and it starts by looking at the base case and working upwards. Also, each state is labeled with a name and the calculated score.



11. Note: Alpha Beta Pruning

After MVP, it is quite common for students who wrote minimax to add Alpha Beta Pruning (ABP for short) as an additional feature. ABP is a minor extension to the minimax algorithm that causes it to speed up its calculation (sometimes significantly) by "pruning" stupid parts of the game tree that it knows it won't have to explore.

12. **Where to start looking**

Gry to avoid looking at sources that actually show code. This includes the GeeksForGeeks articles on Minimax and Alpha Beta Pruning (the ones for Expectimax and Monte Carlo are less useful code, so looking at those pages is less problematic). If a page does have code, try to avoid looking at it too much.

You must cite all sources, even the ones below!

- The Wikipedia pages for the algorithms ([**Minimax**], [**Expectimax**], [**Monte Carlo Tree Search**], [**Alpha Beta Pruning**]) are useful, but some of them contain a lot of other information you don't need to worry about. Some of them also have some decent images / GIFs, which show you how the process works step-by-step.

- Some articles on Minimax / ABP: [**LevelUp Coding**] [**Baeldung**] [**Medium**] [**Educative**]
And some videos: [**Computerphile**] [**Berkeley CS188**]

- Some articles on Expectimax: [**Baeldung**] [**GeeksForGeeks**]

- Some articles on Monte Carlo Tree Search: [**GeeksForGeeks**] [**TowardsDataScience**] [**Baeldung**]
- If possible, avoid the GeeksForGeeks pages for Minimax or ABP, especially the parts with code or "pseudocode" (their minimax "pseudocode" section isn't very "pseudo").

- Plenty of universities (including CMU) have course notes you can find online explaining some or all of these algorithms. Some of these courses at CMU are 15-150 and 15-281.