Artificial Intelligence for Games

Board Games, Part 1

Chapter 8

Game Theory

Game Theory

- Based on the seminal work by John von Neumann and Oskar Morgenstern, game theory is generally relatively un-useful in video game development.
- We begin with the “types of games”:
  - Number of players (nearly always two)
  - Plies (singular, *ply*, a half-turn), moves and turns
  - This has nearly always used chess as the canonical example

The Goal of the Game

- Zero-sum: one player's win is the other player's loss.
  - It doesn't matter if player A gains or player B has losses, the result is the same.
- Plus-sum: even the losers gain something, just not as much.
  - The stock market, ideally.
  - Wealth creation in general.
- Minus-sum: players at a casino.
  - The house always wins (the game is plus-sum for them).

Information

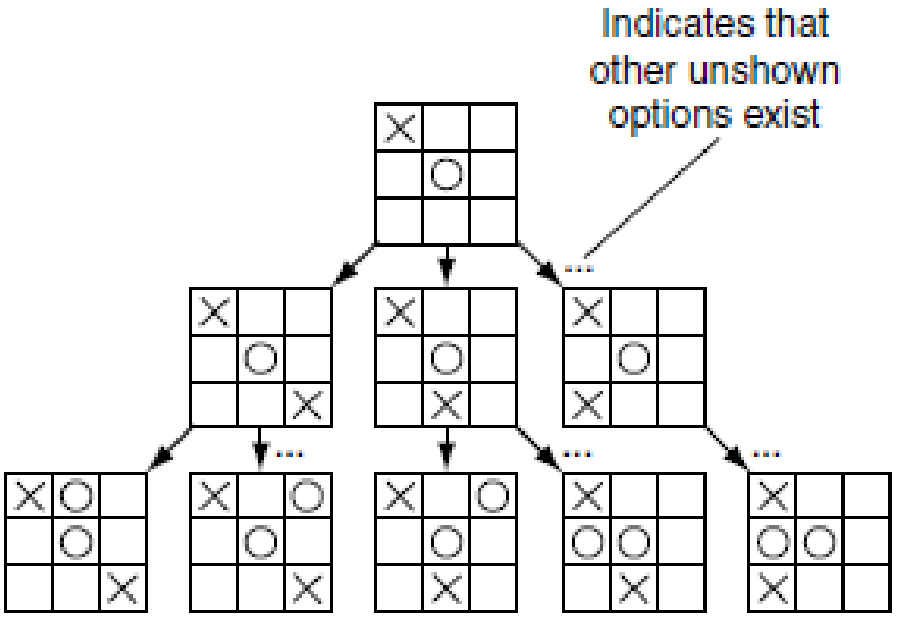
- The game can be one of *perfect information* (like Chess, Checkers, Reversi, Go), where both (or all) players know all there is to know about the state of the game.
- Or, the game can be one of *imperfect information*: each side possesses only partial (and different) information… typical of turn-based strategy games.
- The game can include randomness (Backgammon), meaning both sides lack perfect information.
- Strategy games also often include this.

Assumptions

- Most of the work in this area involves two players and perfect information.
- Going beyond that requires adapting the core algorithms.

The Game Tree

- Central to any turn-based game is the game tree, a model of how each ply (half-turn) of the game unfolds and what outcome it leads to.
- Tic-Tac-Toe is the canonical simple example here.



Plies and Turns

- Each player contributes a move at that player's "ply".
- The state of the game always changes as a result of that.
- With games like Tic-Tac-Toe, the number of available moves shortens as the game progresses: 9, 8, 7, etc.
- Some board positions don't have subsequent moves.
  - These are called *terminal positions*.
  - At every terminal position, we can assign a score to determine the winner and loser.
  - In zero-sum games, the two scores sum to zero.
  - In non-zero-sum games, someone came out ahead.

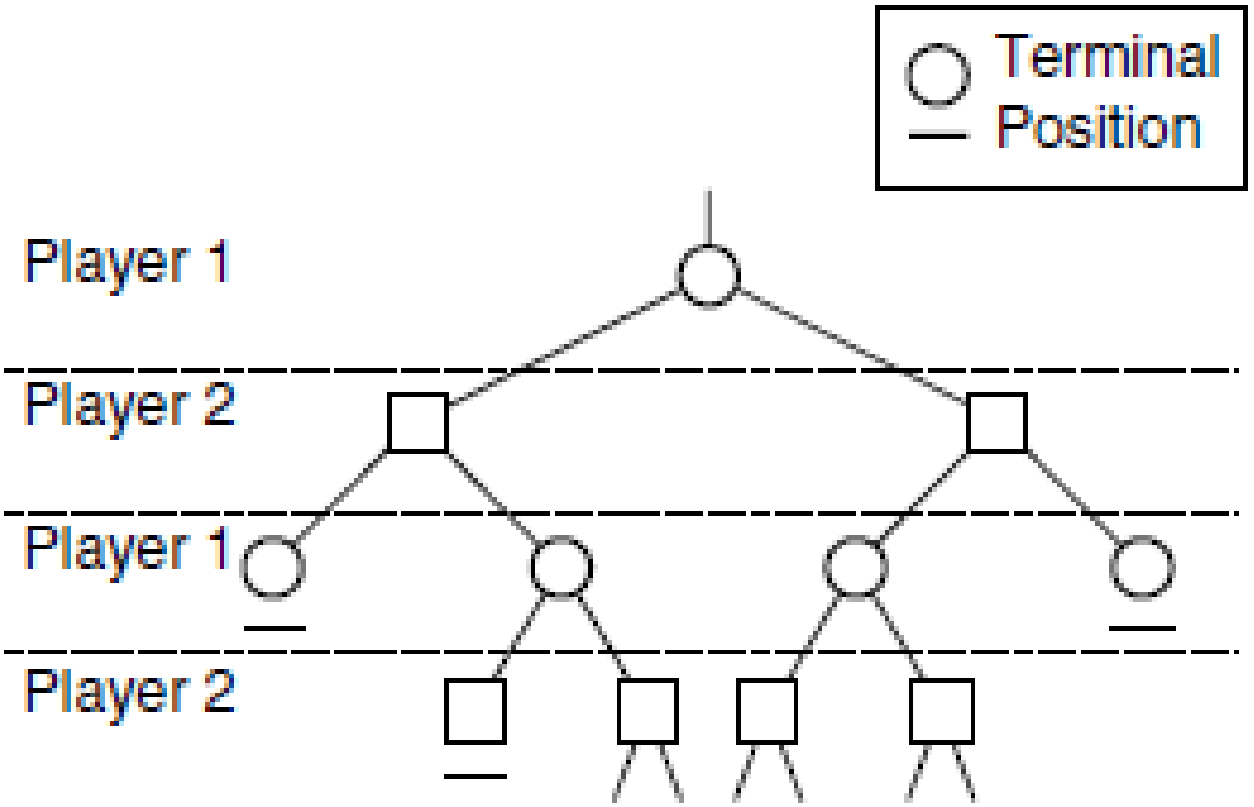
Representations

- Notice in the above Tic-Tac-Toe tree, each node in essence stored an entire snapshot of the board.
- In practice, we don't want to draw out the full state of the board at each node of the tree.
- In fact, we want the board state representation to be as fast and compact as possible.
- So a number of stratagems have been devised for ways to pack bits, store only delta information, etc.
- Consider doing this for Chess, for example….

Branching Factor and Depth

- The *branching factor* is the number of possible branches leading out of any node of the tree.
  - Typically, expressed as an average.
  - For Chess, there are an estimated 35 branches on average.
  - For Tic-Tac-Toe, the BF is 9, then 8, etc.
  - Backgammon is very “bushy”: 21 distinctly different 2d6 rolls times all the moves possible from any of those rolls.
  - This is generally in excess of 200.
- The *depth* is the number of turns forward to look in the tree.

Ply Notation



The Challenge of Scaling

- Taken together, it is clear that a tree for any reasonably complex game will quickly become very large and deep.
- Computers are generally better at playing such games when the tree is deep and with relatively low branching factors.

Transposition

- We are going to want to take every available shortcut to minimize both execution time and memory usage.
- One such trick is *transposition*.
  - Consider Tic-Tac-Toe again: there are really only three opening moves: center, corner, and side.
  - We can use rotation and reflection to take advantage of the game's natural symmetry.

Minimaxing

Minimaxing

- A computer plays any turn-based game by looking at the actions available to it on a turn and chasing down the consequences – as best it can – to decide the "best" move.
- We use a heuristic for this, called the *static evaluation function* (SEF).
  - This is a measure of how good a shape the computer player is in after making a possible move.
  - A naïve function for Chess: *pawn=1, bishop=knight=3, rook=5, queen=9.* Add them up and take the difference.

The Static Evaluation Function

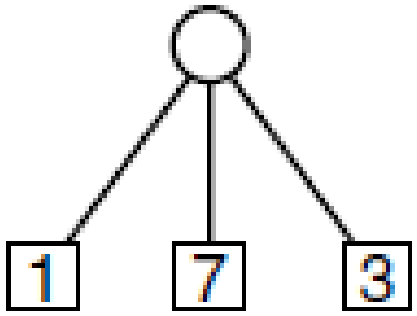
- If we arrive at a terminal node, this function's value is easy: the final score (+) for winning or the final score (–) for losing.
- As we move further back, this gets harder and harder to compute.
  - We have no choice but to search all possible branches from where we are to all the eventual terminal nodes.
  - We have to rely on our knowledge of how best to play.
  - In Reversi, it is often best to have fewer of your own color on the board during mid-game, to preserve initiative.
  - Thus, a good static evaluation function should favor such positions.

Range of the SEF

- In practice, we don't care about the "units," as long as the function reasonably returns good estimates.
  - Integer math is faster than float (we do care about this!)
  - The range doesn't matter. E.g., –100 to +100 is good, since it fits into a single byte (we care about this too!)
  - Use a really large number for win/lose (+1000 / –1000)
  - You need to do this so that the algorithm can reliably find it during the search.
  - You can also combine various measures, suitably weighted, to arrive at your SEF.
  - This resembles what we did with Tactical Analysis.

Using The SEF

- Using the SEF the computer simply evaluates the current state of the game, computing the score for each of the available moves and selecting the best one.
- If the SEF were perfect, that is all we would ever have to do.
- But the perfect SEF is “pure fantasy.”
- Getting a decent enough SEF is a major challenge.



Simple Move Choice

- So it is necessary for the computer to search the entire game tree, evaluating the highest score obtainable for any of the available moves, then selecting the best move from among them.
- This will prove to be a far more exhaustive – and exhausting – process than a single, top-level selection.
- This is, more or less, what human players do now in turn-based games.
- The algorithm we will be using is *minimax*, one of the most enduring algorithms in all of Game AI.

Minimaxing

- Humans generally have much better intuitions about the state of the game relative to any SEF, but even with a less-than-optimal SEF, the computer is able to search far more deeply than (most) humans.
- Computer SEFs are “narrow, limited, and poor.”

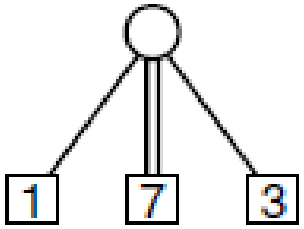
Minimaxing

- When we choose a move, we try to choose the best move for us… or the worst for our opponent.
- Similarly, our opponent tries to choose the best move for her… or the worst move for us.
- This is the basis for minimax: one side tries to maximize for oneself, minimize for the opponent, and the situation is exactly reversed for the opponent.
- When we search the game tree, we therefore alternate between looking for the best move (when it's our turn), and the worst move (when it's the opponent's turn).

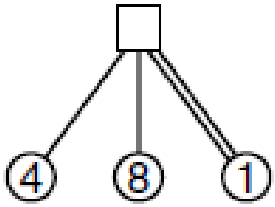
Minimaxing

- As the search proceeds we want to find the maximum score (eventually from the SEF), but we are mindful of the fact that the opponent is grinding out an equal and opposite agenda, trying to allow us only the minimum score (again, but from their SEF).

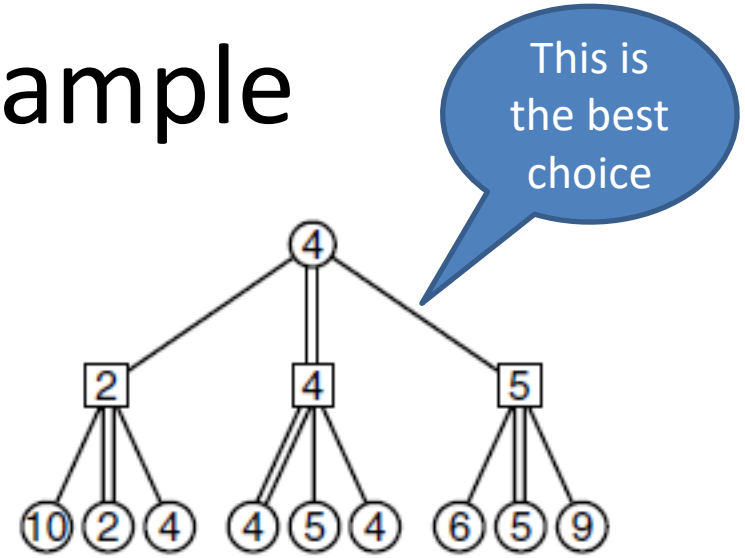
A Simple Example



Our move, first turn



Opponent's move, first turn



A full first turn, not quite done

- We basically bubble up the scores from the leaves (computed with the SEF) and attach them to the nodes, always alternating max and min according to whose turn it is.
- In the right-hand example above, we will eventually discover the best choice this way.

The Minimaxing Algorithm

- The algorithm is (not surprisingly) recursive.
- From any given board position it examines all possible available (legal) moves available to a player from that position.
  - It generates the new board position that would happen as a result of that move, and also tries to get its score.
  - … which it does by recursing from that point downward.
  - As it goes, it alternates between min and max:
    - Selecting the max when it's "our" turn (actually the AI's)
    - Selecting the min when it's the opponent's (human's) turn.

The Minimaxing Algorithm

- To prevent infinite search, we usually impose a depth limit.
- When the (remaining) search depth is zero, we are done with that branch. Apply the SEF and bubble the result upwards.
- If the algorithm is considering the move for the current player (or the AI) it returns the highest value found.
- For the opponent, it returns the lowest value found.
- From either POV, this represents the best move found.
- Lets look at some pseudocode (Code 8.2.3.py).

Pseudocode

- In the pseudocode we're assuming `minimax()` can return both the best move and its score. Fortunately, Python lets you do that, otherwise devise an appropriate `struct`.
- Note that we use `INFINITY` and `-INFINITY` to guarantee that it will always beat out any score computed by the SEF.
- Also note, no SEF. This depends greatly on the game.
- The call to `minimax()` was wrapped by `getBestMove()`, which simply does what it says.

More Than Two Players

- What if you want to model a game with more than two players?
- Simple, just have `currentPlayer` range over more than two values.
  - When running `minimax()` when "you" are the player, perform maximization.
  - Otherwise, when running `minimax()` for every other player, perform minimization.

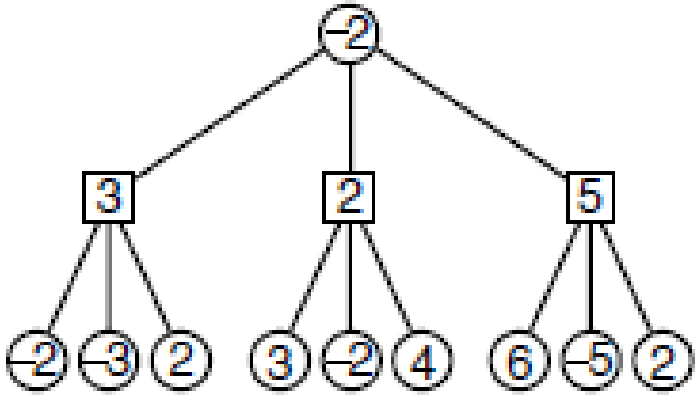
Performance

- The algorithm is O(*d*) in memory, where *d* is the maximum depth of the search (or maximum depth of the tree if that is smaller.
- O(*nd*) in time, where *n* is the number of possible moves.
- The wider and deeper this gets, the worse the algorithm performs.
- Nearly all the improvements made to basic minimax have been about fixing exactly this.

Negamaxing

Negamaxing

- It is very common for two-player games to be zero-sum, in which case one player's loss exactly equals the other player's gain.
- We can improve minimax slightly by simply alternately flipping the signs of the best score.
- This simplifies the bookkeeping somewhat.



Negamaxing

- This means the SEF no longer needs to know whose turn it is. It simply computes the best move for whichever player's turn it is.
- Time for more pseudocode. (Code 8.2.4.py).

Performance and Implementation

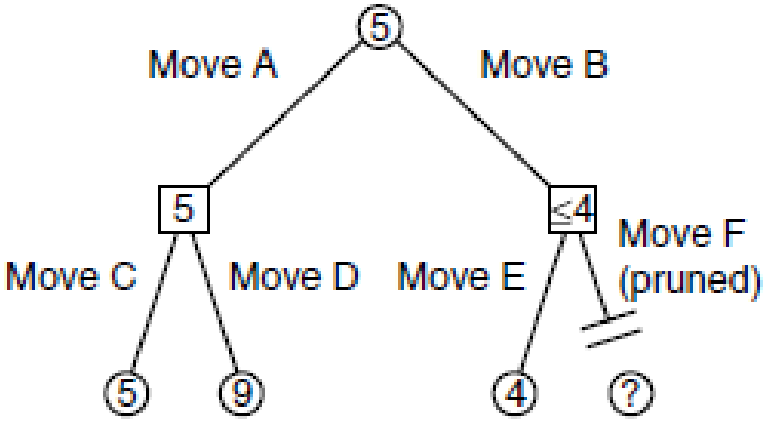
- Same big-O() notation as before.
- The per-search behavior is faster than before, but it still suffers the same scaling issue as minimax.
- Although what we are doing is often called “minimaxing” in practice, negamaxing is what most people use.
- Minimaxing is a generic term for all this.
- Next up, looking at a lot of powerful optimizations.

AB Pruning

AB Pruning

- The whole idea of pruning is to find ways to lop off entire sections of the search tree.
- We can do this at any point where we can determine that a given branch can't possibly have the best move.
- This is done with two kinds of pruning: *alpha* and *beta*.

Alpha Pruning



- If player 1 considers move A, player B makes move C and player 1's score is 5, which we bubble up.
- If player 1 then considers move B, player 2 makes move E and player 1's score is now 4.
- We conclude that the Move B branch can't possibly lead to a better outcome, so we prune F and abandon any further search there.

Alpha Pruning

- To do alpha pruning, we need to keep track of the best score we know we can achieve… so far.
- This is the lower limit of what we might actually get. We never have to accept any move lower than that.
- This lower limit is called *alpha*, and what we've done is alpha pruning.
- The enemy can't do anything worse to us because the enemy is never given the opportunity.

Beta Pruning

- Beta pruning is similar but now we track the upper limit on what we can hope to score.
- This value is called *beta*.
- We update beta whenever we find a sequence of moves the enemy can force us into.
- With beta we know there is no way to score better than that, so if we find a branch with a higher beta, we can prune it, since the enemy will not allow it.

The Alpha and Beta Window

- Together, alpha and beta provide a window or range of possible scores.
  - We will never choose a move with a score less than alpha.
  - Our opponent will not allow a move that lets us score more than beta.
  - The score we end up will lie between these two values.

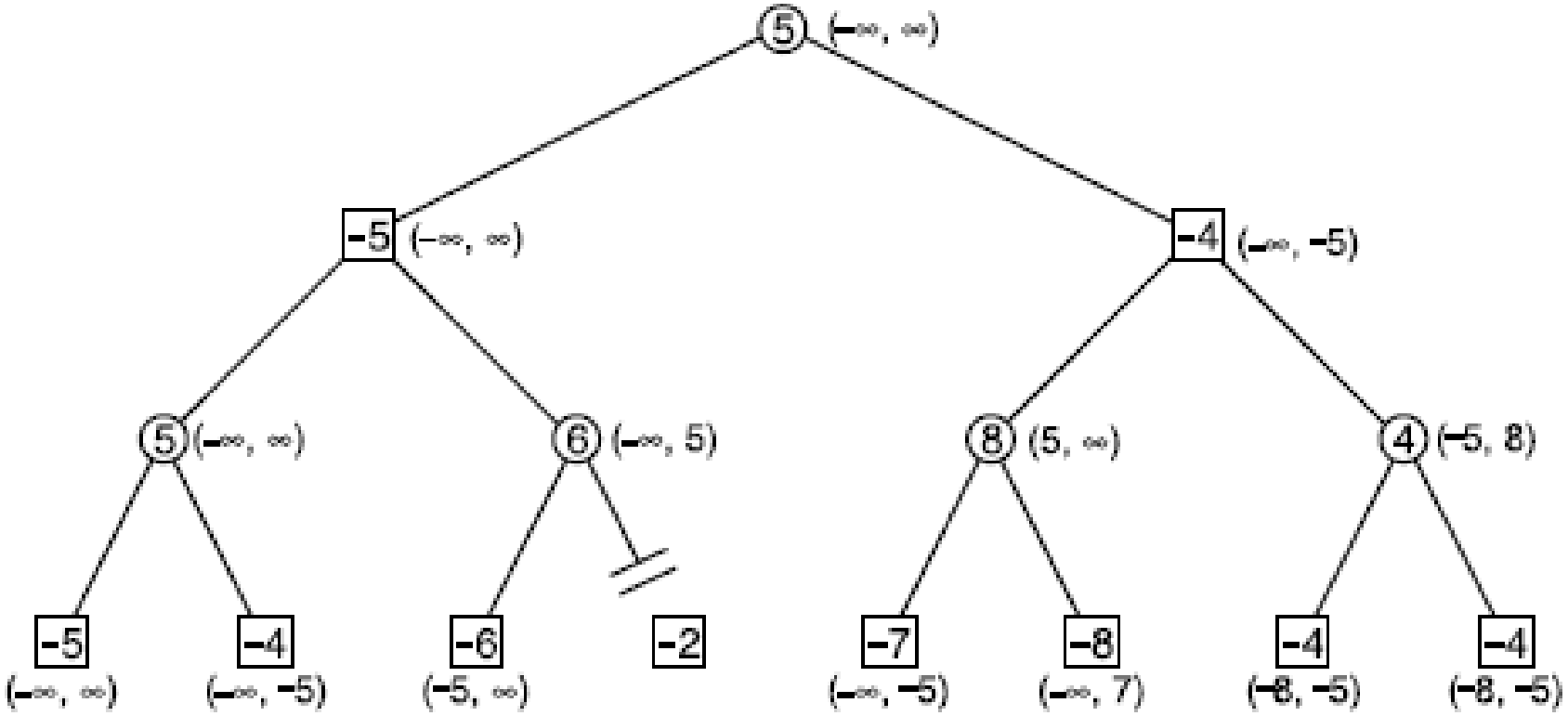
The Alpha and Beta Window

- As we search the tree, we continually update alpha and beta.
- A branch whose values fall outside the window can now be safely pruned.
  - When considering the opponent's turn to play, we perform score minimization, so we check this against alpha.
  - When considering our own turn to play, we perform score maximization, so we check this against beta.
- This leads to the next optimization, AB Negamax.

AB Negamax

- Just as we changed signs in negamax, in AB negamax we alternately swap alpha and beta.
- Only the beta needs to be examined for each ply.
- The AB Negamax solution is the first of the truly practical solution we've seen.
- All subsequent optimizations derive from it.

AB Negamax



AB Negamax

- If we search from left to right, we see that alpha and beta move closer to each other, narrowing the window.
- This means we get to be more and more aggressive in pruning sections we don’t need.
- Time for some more pseudocode. (Code 8.2.5.py).

Performance

- Same as before, O(*d*) in memory and O(*nd*) in time.
- What do we gain by making this change?
  - AB Negamax generally performs better because the AB Search Window continues to narrow, allowing more thorough pruning.
  - But if the left-to-right order is such that little pruning is possible, the algorithm will perform worse.
- If it occurs to you that the left-to-right order is important, you'd be onto something.

The AB Search Window

- The interval between alpha and beta is the search window.
- AB Negamax starts with `-INFINITY` and `+INFINITY` (because it has to).
- As it goes, the window narrows and we get to prune more.
- The more rapidly we can contract the window, the better the performance.

Move Order

- If the most likely moves are considered first, we can contract the window faster.
- But that's the trick: how do we know? That's the whole point of the AI.
- If we knew the best move, we wouldn't need the algorithm.
- But we can run the SEF on the top layer of the search, sort the resulting values, and search best to worst (or worst to best as the case may be).

Memory-enhanced Test Family

- An even more powerful optimization is to use the results of previous searches. Can be from:
  - iterative deepening (coming soon)
  - the results from minimax searches from previous turns.
- The so-called *memory-enhanced test* family of algorithms use this approach to order moves before they are considered.
- This can yield up to a 10x improvement, which generally means getting to search an extra turn or so.

Aspiration Search

- Another massive optimization is to deliberately set a narrow window from the start.
- This range is called the *aspiration*, because we aspire to get a great result quickly.
- But you don’t always: if the aspiration is set too narrow, you might not get a best move.
- What you do then is widen the window and try again.

Aspiration Search

- The “best” setting for the aspiration is often obtained by using the results of a previous search.
- So if a board got scored as +5 (from a previous search), then the window can be set to
  (5 – window size, 5 + window size).

Negascout

Negascout

- The extreme case is to have a search window of zero. This will aggressively prune the tree, throwing out all of the good branches along with the bad.
  - (This is called “Test”).
  - If you got a best move, it meant your guess was correct.
- We have been looking at the “fail-soft” version of AB negamax. If the search fails we get a best-move-so-far, which will be the alpha or beta value.
- When we use this fail data as a way of adjusting the window, we get better performance, leading to the Negascout algorithm.

Negascout

- Original scout algorithm combined minimax (with AB pruning) with calls to the zero-width test. It was quickly supplanted by Negascout.
- Negascout uses AB negamax and the aspiration to drive the test.
- It does a full examination of the first move from each board position. This is done with a wide search window so it doesn't fail.
- Successive moves are examined with a scout pass using a window based on the score from the first move.
  - If it fails, it is repeated with a full-width window (same as AB negamax).
- The initial wide search establishes a good approximation for the scout test.
- We avoid getting a lot of scout failures while doing a lot of pruning.

Aspiration Negascout Pseudocode

- Aspiration Negascout is extraordinarily powerful and used in many commercial two-person turn based games.
- Such games nowadays can routinely beat the best players.
- Time for some pseudocode. (Code 8.2.7.py)

Performance

- O(*d*) in memory; O(*nd*) in time.
- Although the order is the same, Negascout generally eliminates more branches than AB negamax, so Negascout generally dominates AB negamax.
- Until recently, Negascout was the undisputed winner in the algorithm wars.
- Some new variants, the *memory-enhanced test* (MT) approach can perform even better.

Move Ordering and Negascout

- Just as using sensible ordering of the moves to examine can speed up AB Negamax, it works even better for Negascout.
- This is because the initial wide-window pass will be very accurate, and the subsequent scout passes will fail less often.
- Add to that a memory system (next module) and it works even better.

Principal Variation Search

- Negascout is related to another algorithm called Principal Variation Search (PVS), which we won't cover here.
- The differences are minor, and in practice, most people use Negascout.

Next Up…

- Transposition Tables and Memory
- Memory-enhanced Test algorithms.
- Opening books and set plays
- Iterative Deepening
- Variable Depth approaches
- Turn-based strategy games.