9 Adversarial Search

In ordinary state-space search, you are the only agent involved: you are in a state, you perform operations, and an operation transitions you to a new state. In contrast, in **adversarial search** (or **game-tree search**), you are not the only agent. In addition to yourself, there is an **adversary** (an opponent) acting to change the state as well.

Adversarial search is usually cast into the form of a **game** between you and an opponent (that is, a **two-agent game**). For example, perhaps a pesky human is trying to play Chess or Tic-Tac-Toe with you. But not just any game: adversarial search can only be applied to certain kinds of games. Here are the qualities we're looking for in a gem:

- **Two-Player, Turn-Based Games** The game must involve exactly two players, and the players do not act simultaneously: they take turns performing moves (though it's okay if one player does several moves before relinquishing control to the other player). For example, Chinese Chess, Monopoly, and Scrabble aren't necessarily Two-Player games. And Diplomacy and Poker aren't Turn-Based games because each player is plotting his action simultaneously.

- **Constant-Sum Games** The final scores of the two players must add to a fixed constant value. For example, if the constant is 10, then if you won 8 points, your opponent must have won 2 points. It's reasonable for a player to have scored negative points. f the constant is 12, for example, and you won 16 points, then your opponent won -4 points.

  The most common kinds of constant-sum games are **zero-sum games**, where the constant is 0. Thus if you won 4 points, your opponent must have won -4 points. If you won 0 points, your opponent did as well (a tie). You can always convert a constant-sum game into a zero-sum game: just subtract half of the constant from each person's score. A simple kind of zero-sum game is a **win-lose-draw game**, where there are only three outcomes: you won, you lost, or you had a draw. We can write this as the ending score being 1, -1, or 0.

  The nice thing about constant sum games is that we can simplify how we normally think about them. When we think about games, we usually see them as contests where each person is trying to maximize his own score. But in zero-sum games, we can think of them as *you* trying to maximize your score, and your opponent trying to *minimize your score* (which thus maximizes his). This simplifies the game to being just about one score: yours.[70]

  In some games (like Othello or Go) the objective is not just to win, but to win by the highest score, and your score plus your opponent's doesn't sum to a constant. Even in this case, you can cast the situation into a zero-sum game. Let x be your current score and y be your opponent's current score. Your "zero-sum" score is $\frac{x-y}{2}$ and your opponent's "zero-sum" score is $\frac{y-x}{2}$. For example, if you have 20 points and your opponent has 10 points, then your "zero-sum" score is 5 and your opponent's "zero-sum" score is -5.

- **Perfect Information Games** In these games, sometimes called **total information games**, both you and your opponent know everything about what's going on at all times. For example, Scrabble, Battleship, and Stratego aren't perfect information games because each of you are hiding something from the other player.

[70] Remember, by "you" we mean "you, the computer algorithm'.

So what games are left? Here are a few: Tic-Tac-Toe, Nine Men's Morris, Gomoku (Pente), Checkers (both French and English/Draughts), Connect Four, Chess (all variants), Go, Othello, Hex, Y, and Mancala are all examples of total-information, two-player, turn-based constant sum games. Backgammon is as well, though it also has an element of **stochasticity**: chance is involved.

A Game-Playing Agent We're going to discuss how to make an agent which plays these kinds of games. In general the agent does the following each time his turn is up:

1. Determine every single possible move the agent can make.
2. For each such move, determine the new game state that would result from making that move.
3. For each such new game state, call a **scoring function** to assess how good it would be for the agent to be in that new state.
4. Perform the move which led to the new game state with the highest score.

Notice that the agent *does not* bother scoring his *current* state: it's irrelevant. All that matters is which *new* state would be the best one to be in. Most such scoring functions are variants of a search algorithm called **Min-Max**.

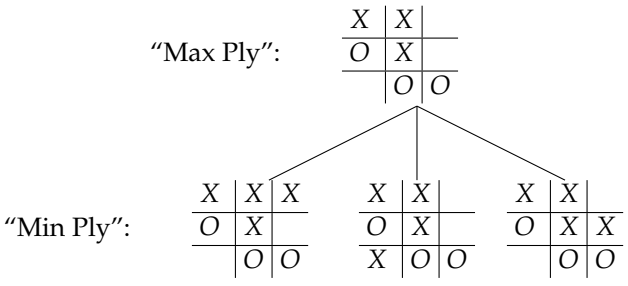
9.1 The Min-Max Algorithm

Let's cast this problem into a kind of state-space search. We define a state as a current game configuration: what the board looks like and whose turn it is. Transition edges in the space represent possible moves that a player could make in that game configuration. The initial state in our state space is the *current game state* (by which we mean, one of the "new game states" computed in step #2 above). Some states have no outgoing edges: these **terminal states** represent the end of a game. A goal state would be a terminal state where you have defeated your opponent, and ideally by the highest possible amount. On the other hand, our opponent has a different goal: to reach a terminal state in which you have *lost*, and ideally by as much as possible. Thus you are trying to maximize your score, and your opponent is trying to minimize *your score*. We will give each of you names. You will be called **Max**, and your opponent will be called **Min**.

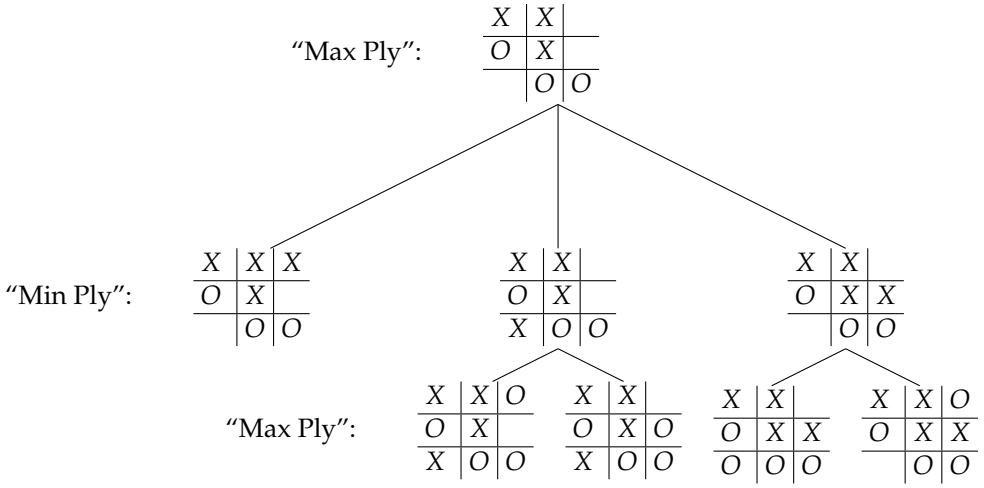
Game search is usually thought of in a search tree fashion. Consider the following Tic-Tac-Toe situation, where you (Max) are X, and it is your turn:

X	X	
O	X	
	O	O

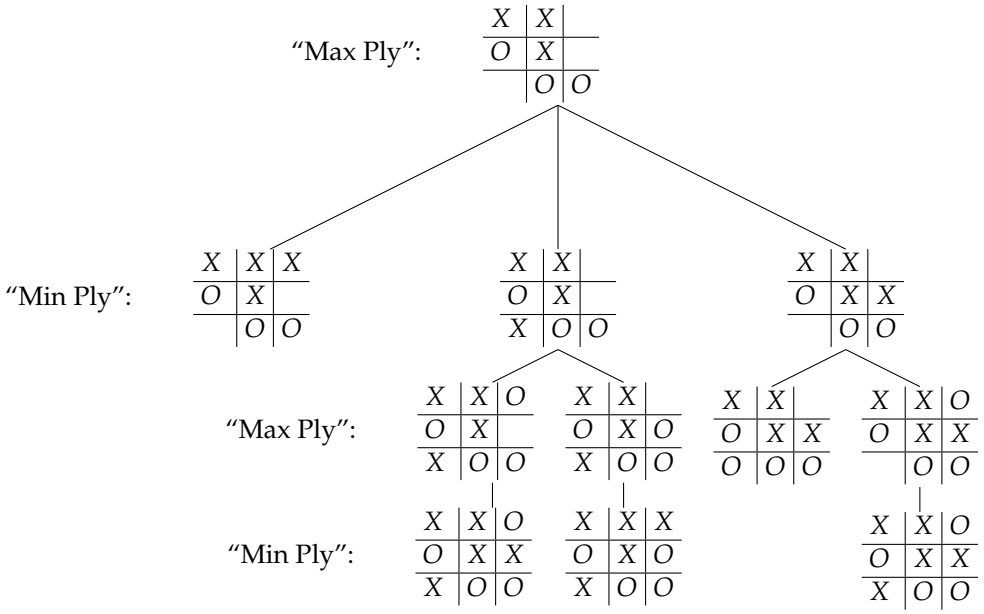
Here you have three possible moves, and so the state has three children, each the state resulting from one of those moves:



In every state in a "Max Ply" (layer) in the search tree, it's Max's turn. Likewise, it's Min's turn in every state in a "Min Ply". In the leftmost situation in the previous example, you have won: that's a terminal state. In the others, your opponent (Min) still now make a move. So the tree expands to:

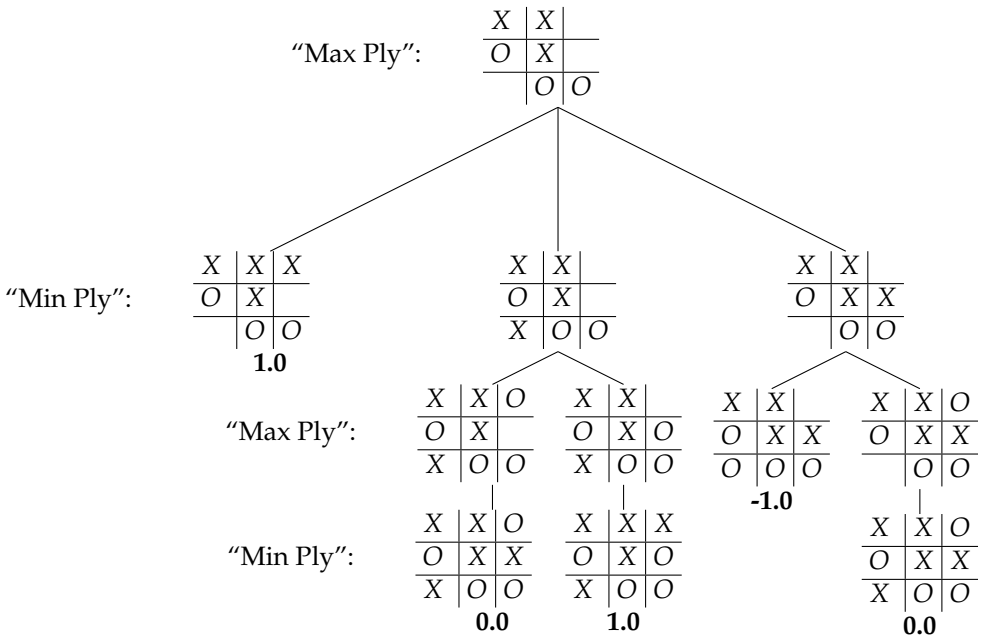


The third child is a terminal state (O won). The others can still be expanded by you (Max) making a move:

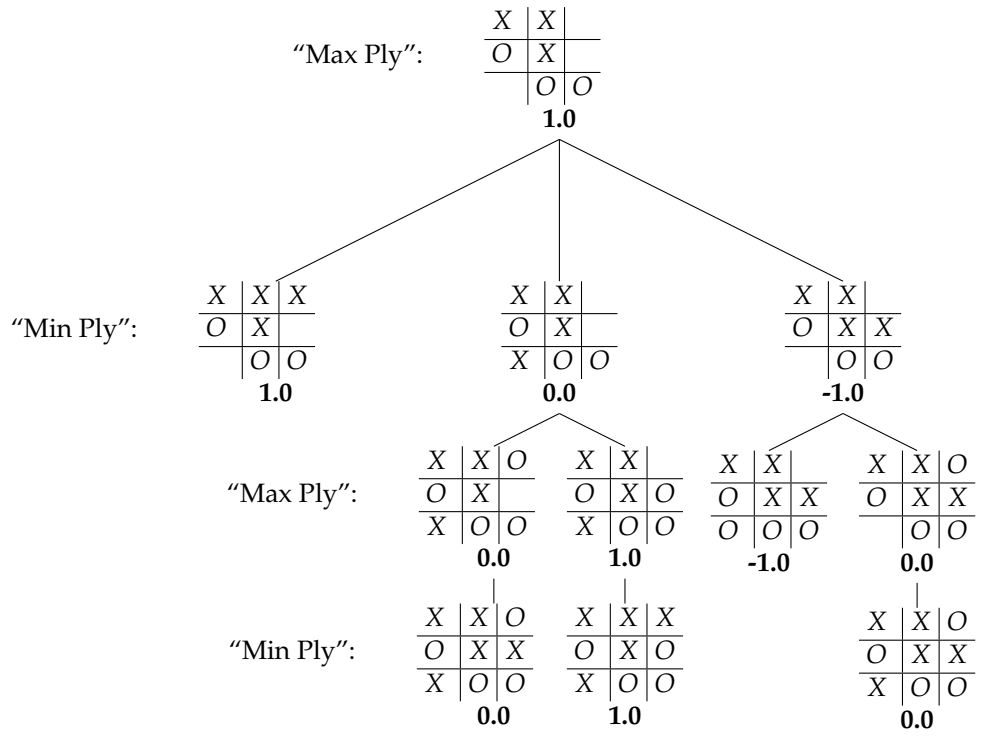


All these children are of course terminal. We call this construction a **game tree**. A game tree, once expanded, can help us determine how good it would be to be in the situation represented by the initial (root) state. To compute this, we're going to assume that both Min and Max are **intelligent players** and won't make stupid moves.

We begin by labeling each terminal (leaf node) state with the score for Max in that state. In Tic-Tac-Toe, the only possible scores are 1 (Max won), 0 (tie), and -1 (Min won).



Now starting bottom-up, we label the remaining nodes in the Min plies with the **minimum score** of their children. And label the remaining nodes in the Max plies with the **maximum score** of their children:



This is a **Min-Max game tree**. As we can see, the situation is good for Max in the root node: it's been scored a 1.0 because Max has a way to win no matter what Min does.

Heuristic Board Evaluation In Tic-Tac-Toe, it's feasible to compute a Min-Max tree clear out to all possible terminal states, then bubble up the scores and compute the right move to make. But in other games, like chess, this just isn't possible! The branching factor for Tic-Tac-Toe is no more than 9 and the trees are no more than 9 moves deep. As a loose upper bound there are only 9! (45360) Tic-Tac-Toe games. That's tiny. On the other hand, the *average* branching factor for chess is around 35, and games might be up to, oh, 100 plies deep (50 moves on each side) or more. Trust me when I tell you that 35^{100} is a very large number.

Thus in Chess we cannot possibly create a full game tree, except perhaps very near the end of the game. So what do you do in this situation? At some point the tree expands to so many children that you have to cut off the search. At this point you're left with a bunch of child states which aren't terminal states. Since they're going to be essentially leaf nodes in your tree, you *have* to give them a score in order to perform the Min-Max computation.

The way this is handled is by handing these states to a **heuristic state-evaluation function**. This function is responsible for evaluating the states and giving them approximate scores suggesting how good or bad Max is doing. If for example your scoring system assigns 1 for a total win for Max, -1 for a total loss, and 0 for a draw, then your state-evaluation function might assign a 0.62 to a state where Max is mostly winning, and -0.23 to a state where Max is a little behind.

It's up to you to build this heuristic, and it is quite the black art. One approach, for example, is to make the heuristic be the weighted sum of various factors, for example (in chess):

- What is the difference of pieces between Max and Min?
- What is the difference in "material" between Max and Min (in chess, some pieces are considered to be worth more than others)
- How many of Max's rooks are in control of a column with no pawns? How about Min?
- How many of Max's pawns are advanced far down the board? How about Min?
- Is Max castled? How about Min?
- Does Max have both his bishops? How about Min?
- Is Max in check? How about Min?

... etc. The heuristic function might compute values for each of these questions, weight them according to importance, and then add them up to determine how good a situation Max is in.

The Min-Max Algorithm To perform the Min-Max algorithm, you'll need the following pieces:

- A heuristic state evaluation function
- A function which tells you if it's Max's turn in a given state
- A function which tells you if a state is a terminal (end game) state
- A function which computes the child states from a given state as a result of making one move
- A desired maximum depth to search

We will also assume that all possible scores, including those returned by the heuristic state evaluation function, range from MINSCORE (the worst possible score for Max) to MAXSCORE (the best possible score for Max), where 0 is effectively a draw. Here's the algorithm:

Algorithm 26 *Min-Max*

```
1: S ← current game state
2: depth ← current depth                                          ▷ Initially 0
3: maxdepth ← maximum desired depth
4: ismaxsturn(...) ← returns true if it's Max's turn in a given state
5: terminal(...) ← returns true if the game is over in a given state
6: expand(...) ← expands a state into all possible successor states
7: evaluate(...) ← returns a heuristic evaluation of the given state

8: if terminal(S) or depth ≥ maxdepth then
9:     return evaluate(S)
10: else if ismaxsturn(S) then
11:     score ← MINSCORE
12:     for each child C ∈ expand(S) do
13:         mm ← Min-Max with C, depth +1, maxdepth, ismaxsturn, expand, terminal, evaluate
14:         score ← max(score, mm)
15:     return score
16: else
17:     score ← MAXSCORE
18:     for each child C ∈ expand(S) do
19:         mm ← Min-Max with C, depth +1, maxdepth, ismaxsturn, expand, terminal, evaluate
20:         score ← min(score, mm)
21:     return score
```

You start the MIN-MAX function by passing in the current board state, and 0 for the current depth. Typically you'll have some maximum allotted time to do analysis. The branching factor of your game, your chosen tree depth, and the cost of performing a heuristic evaluation both contribute to how many states you can evaluate in that time, given your computer's speed. The branching factor is fixed, as is the heuristic evaluation function once you've created it. Thus the only variable you can play with to maximize your search is the maxdepth.

Note that you can make Min-Max a bit faster by cutting off analysis as soon as you find a sure-fire win or loss. For example, if it's Max's turn, and you find a child whose score is MAXSCORE, then obviously that's going to be Max's score, so there's no reason to continue evaluating the other children. (Likewise if it's Min's turn and you find a child whose value is MINSCORE, no need to continue evaluating the other children). We'll see an even smarter extension of that in just a minute.

Also note that there's a trade-off involved in developing your heuristic evaluation function. If you have an inaccurate function, you'll need to search deeper to compensate for it (expensive). But if an accurate function is slow, then even though you don't have to search as deep, each state evaluation is more costly. You want a function which is both fast and accurate, but figuring out the best trade-off between the two isn't always easy.

9.2 A Game-Playing Agent

Min-Max just tells you the score of a given game state. Now what you need to do is use it to play a game. The procedure is simple: take your current game state and expand it into its children. Run Min-Max on each of those children to get their scores. Pick the child with the highest score. That's the move you should make.

Algorithm 27 *Make a Move*

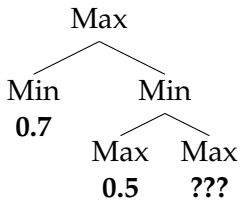
```
S ← current game state
maxdepth ← maximum desired depth
ismaxsturn(...) ← returns true if it's Max's turn in a given state
terminal(...) ← returns true if the game is over in a given state
expand(...) ← expands a state into all possible successor states
evaluate(...) ← returns a heuristic evaluation of the given state

best ← □
bestscore ← MINSCORE
for each child C ∈ expand(S) do
    score ← Min-Max with C, 0, maxdepth, ismaxsturn, expand, terminal, evaluate
    if score > bestscore then
        best ← C
        bestscore ← score
make the move which leads to best
```

9.3 α-β Pruning

The trouble with MIN-MAX is that it searches more subtrees than it has to. As it turns out, you can "prune out" a lot of subtrees with the following heuristic assumption: your opponent isn't stupid.

Here's the situation. Imagine the following game tree:



You have already evaluated Min:0.7, and it scored, well, a 0.7. And you've evaluated Min:0.5 and it scored 0.5. Now here's the question: should you bother evaluating Max:??? and its children?

Nope. Consider: when your opponent decides to make a move, he will choose between Max:0.5 and Max:???. He'll pick the minimum of the two, thus whatever he picks, it'll have a score less than or equal to 0.5. This means that YOU will get to pick between Min:0.7 something that is ≤ 0.5. Obviously you will pick the higher, which is guaranteed to be 0.7. Thus *no matter what Max:??? is, you will always pick Min:0.7*. So if it doesn't matter, why bother computing it?

In general, α-β pruning avoids searching subtrees where it knows that your opponent will pick something equal to or worse than that, and then YOU will pick something better than the two

(or vice versa: you will pick something better or equal to that, and your opponent will then pick something worse than either of them).

The α-β pruning algorithm is simple, but it takes some thought to understand what's going on. The general idea behind α-β pruning is that at all times the algorithm keeps track of the best option that YOU have at any point in time (α) and the best option that your opponent has at any point in time (β) from the initial state to the current state. If it's your turn, and β is less than α, then your opponent is going to pick β (or worse), and you in turn will ultimately pick α, so it doesn't make any sense to continue looking. And vice versa. Now that you're sufficiently confused, here's the algorithm.

Algorithm 28 *Min-Max with α-β Pruning*

$S \leftarrow$ current game state
depth $\leftarrow$ current depth ▷ Initially 0
maxdepth $\leftarrow$ maximum desired depth
ismaxsturn(...) $\leftarrow$ returns true if it's Max's turn in a given state
terminal(...) $\leftarrow$ returns true if the game is over in a given state
expand(...) $\leftarrow$ expands a state into all possible successor states
evaluate(...) $\leftarrow$ returns a heuristic evaluation of the given state
$\alpha \leftarrow$ Max's best score so far along the path from the initial state ▷ Initially MINSCORE
$\beta \leftarrow$ Max's worst score so far along the path from the initial state ▷ Initially MAXSCORE

if terminal(S) or *depth* $\geq$ *maxdepth* **then**
    return evaluate(S)
else if ismaxsturn(S) **then**
    for each child $C \in$ expand(S) **do**
        mm $\leftarrow$ Min-Max with α-β Pruning with C, *depth* $+1$, *maxdepth*, *ismaxsturn*, *expand*, *terminal*, *evaluate*, α, β
        $\alpha \leftarrow \max(\alpha, mm)$
        if $\alpha \geq \beta$ **then** ▷ Cut off: don't bother searching any more
            return β
    return α
else
    for each child $C \in$ expand(S) **do**
        mm $\leftarrow$ Min-Max with α-β Pruning with C, *depth* $+1$, *maxdepth*, *ismaxsturn*, *expand*, *terminal*, *evaluate*, α, β
        $\beta \leftarrow \min(\beta, mm)$
        if $\alpha \geq \beta$ **then** ▷ Cut off: don't bother searching any more
            return α
    return β

You start the α-β pruning algorithm by passing in 0 for the current depth, MINSCORE for α, and MAXSCORE for β. The effectiveness of α-β pruning is strongly dependent on the order that you look at the children. In a perfect world, α-β pruning would reduce the branching factor from b to effectively $\sqrt{b}$. That's great! But this assumes you pick exactly the right kids to look at first so you build up those α and β values as soon as possible. In the worst case, you gain nothing.

One reasonable way to order the the children is to evaluate them with your heuristic evaluator function, then sort them by their scores. If it's Max's turn, you'd sort them so you looked at the highest scores first. If it's Min's turn, you'd sort them so you looked at the lowest scores first.

9.4 Dealing with Extra Turns, Cycles, and Chance

Repeated Moves Some games, like Mancala, let a player take extra turns, under certain circumstances, before relinquishing control to the other player. Thus the children of a Max game state might be a Min game state, or they *might be another Max game state*.

This is in fact not an issue at all. Though all the examples we've seen so far assume that all states at a given ply are all Max nodes or all Min nodes, this doesn't have to be the case. Min-Max, with or without α-β, will work just fine in this situation.

Cycles Some games cannot get caught in cycles: you can't return to a previous game or board configuration that you'd seen earlier in the game. For example, Tic-Tac-Toe can't get caught in a cycle because new things are constantly being added to it. The same goes for games like Connect Four, Gomoku, or Nim.

But many games can get caught in cycles. For example, in Chess, if the board is repeated some n times, the game is a stalemate (a draw). In checkers there are no such rules: players can go on and on forever repeating the same positions.[71] In Go the rule is simple: a game may not repeat the same configuration: it is an illegal move to do so.

How you'd handle cycles largely depends on the rules of the game. One way to do it is to keep a previous history of all the game configurations seen so far as part of a game state. When evaluating a state, you might consider these previous configurations as part of the scoring. For example, if you're playing Chess, and the current configuration appeared n times in the past, the configuration is immediately evaluated to be a draw (score 0).

If you're playing Go, it's not even *permitted* to repeat a game configuration. In this case, when you call the `expand(...)` function, it only expands to those children which have not appeared earlier in the game.

In other games, like Checkers, it's permissible to go on and on in cycles. However this will cause Min-Max or α-β to go round and round until they have to give up and use the heuristic evaluation function. You can do better than this by recognizing such situations and scoring them as a 0. That is, just follow the same procedure as described for Chess above, with n set to 1.

Games of Chance Finally, some games involve elements of chance interspersed with moves by each of the players. This adds a lot of complexity, but in some limited cases we can modify α-β to handle this. For example, in the game of Backgammon, Max throws some dice which constrain his available moves. He then chooses a move. Then Min throws some dies which constrain his available moves. He then chooses a move. And so on.

In this example, we can augment Min-Max or α-β with **chance nodes** in addition to Min nodes and Max nodes. A chance node reflects a random event like rolling dice: and the children to a chance node are all possible outcomes which could result from that event. Recall that a Min node returns the minimum of its children, and a Max node returns the maximum of its children. A chance

[71] Though the American Checker Foundation requires that games show progress or else they're eventually declared a draw.

node returns the *average score* of its children (weighted by the probability that each would happen. For example, if child 1 to a chance node happened half of the time and resulted in a score of 1, child 2 happened one third of the time and resulted in a score of 0, and child 3 happened one sixth of the time and resulted in a score of $\frac{1}{2}$, then the chance node's score would be $\frac{1}{2} \times 1 + \frac{1}{3} \times 0 + \frac{1}{6} \times \frac{1}{2} = \frac{7}{12}$.

Here's a variant of Min-Max with Chance Nodes thrown in.

Algorithm 29 *Min-Max with α-β Pruning and Chance Nodes*

```
1: S ← current game state
2: depth ← current depth                                              ▷ Initially 0
3: maxdepth ← maximum desired depth
4: situation(...) ← returns 1 if it's Max's turn in a given state, -1 if it's Min's turnm,
                                                  and 0 if it's time for a chance event
5: terminal(...) ← returns true if the game is over in a given state
6: expand(...) ← expands a state into all possible successor states
7: evaluate(...) ← returns a heuristic evaluation of the given state
8: α ← Max's best score so far along the path from the initial state          ▷ Initially MINSCORE
9: β ← Max's worst score so far along the path from the initial state         ▷ Initially MAXSCORE

10: if terminal(S) or depth ≥ maxdepth then
11:     return evaluate(S)
12: else if situation(S) = 0 then                          ▷ It's time for a chance event
13:     averagescore ← 0
14:     for each child C ∈ expand(S) with associated probability P(C) do
15:         mm ← Min-Max with α-β Pruning and Chance Nodes with C, depth +1, maxdepth, situation,
                                                  expand, terminal, evaluate, α, β
16:         averagescore ← averagescore + P(C)× mm
17:     return averagescore
18: else if situation(S) = 1 then
19:     for each child C ∈ expand(S) do
20:         mm ← Min-Max with α-β Pruning and Chance Nodes with C, depth +1, maxdepth, situation,
                                                  expand, terminal, evaluate, α, β
21:         α ← max(α, mm)
22:         if α ≥ β then                          ▷ Cut off: don't bother searching any more
23:             return β
24:     return α
25: else
26:     for each child C ∈ expand(S) do
27:         mm ← Min-Max with α-β Pruning and Chance Nodes with C, depth +1, maxdepth, situation,
                                                  expand, terminal, evaluate, α, β
28:         β ← min(β, mm)
29:         if α ≥ β then                          ▷ Cut off: don't bother searching any more
30:             return α
31:     return β
```

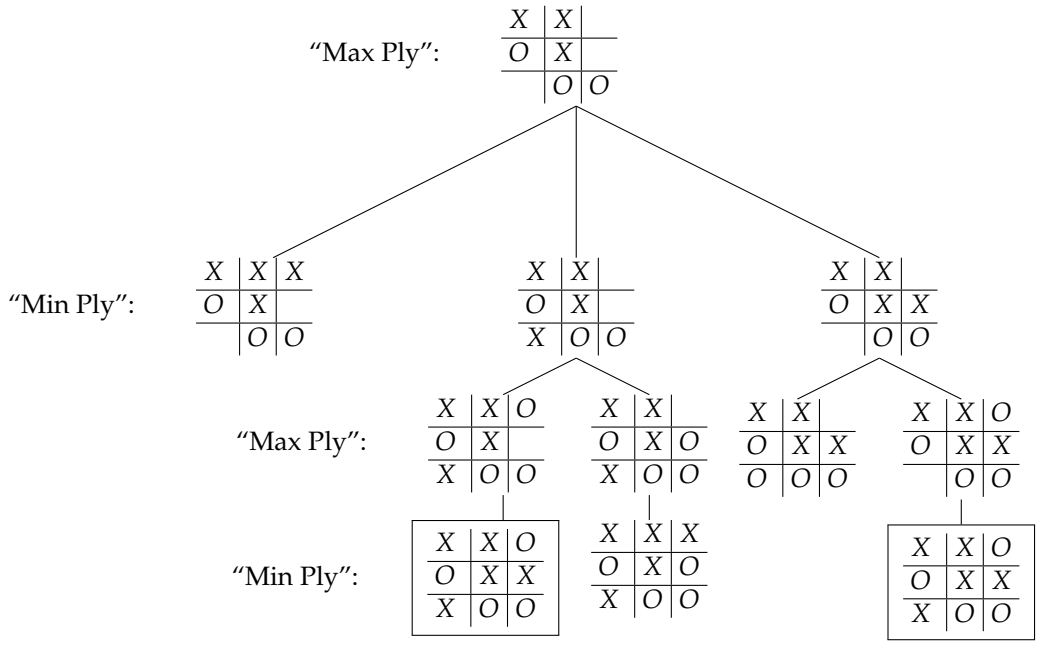
9.5 Search or Optimization?

A quick side note. I define search algorithms as ones in which you **expect to find an answer**, that is, you are *looking for something*. In contrast, optimization algorithms are ones in which you are trying to come up with the **best answer you can** in the time allotted. Is α-β a search algorithm or an optimization algorithm? If it searches clear to the terminal states of the game, we might reasonably say it's a search algorithm. But if we're cutting bait and using a heuristic evaluation function, it's more like an optimization algorithm in reality.

9.6 Memoization and Dynamic Programming

As mentioned earlier, some games have cycles in their game graphs (you can repeat game situations), while others (like Tic-Tac-Toe, Gomoku, or Nim) only progress forward and can never revisit previous situations. That is, the game state graph of these games is a DAG or a tree.

It's rare to see a tree: much more commonly we see DAGs. That is, the same state can be reached by different combinations of player moves. For example, did you notice that the Tic-Tac-Toe game tree example above had the same state in two different leaf nodes? Here they are framed for you.



Thus this tree isn't *really* a tree: it's better thought of as a DAG. We can take advantage of this to avoid unnecessary reevaluation of subtrees if we've already evaluated them before, using a procedure known as **memoization**.

A memoized version of a function $f(x)$ is another function $f'(x)$ which works like this. $f'(x)$ has a hash table which stores solutions keyed by various x values. When $f'(x)$ is called, it first checks to see if the answer to $f(x)$ is in the hash table. If it is, it just returns the answer. Otherwise, it calls $f(x)$, gets the solution, stores it in the hash table keyed by x, and returns it.

We'll use it like this: as we're doing the depth-first traversal of the game tree (er, DAG), we store in a history (a hash table) all the game situations we've seen so far and what their scores were. If we come across a situation which is already in the history, we simply return its score instead

of recomputing it. To do this, we just make a memoized cover function for our Min-Max (or α-β) function. The memoized version calls the original function. Since the original function is recursive, instead of calling itself, we'll have it call the memoized function.

Here's our two-part memoized function. See if it makes sense to you. We start with the subsidiary Min-Max Function. Don't call this directly, instead, call the next function after it:

Algorithm 30 *Subsidiary Min-Max with α-β Pruning*

$S \leftarrow$ current game state
depth $\leftarrow$ current depth ▷ Initially 0
maxdepth $\leftarrow$ maximum desired depth
ismaxsturn(...) $\leftarrow$ returns true if it's Max's turn in a given state
terminal(...) $\leftarrow$ returns true if the game is over in a given state
expand(...) $\leftarrow$ expands a state into all possible successor states
evaluate(...) $\leftarrow$ returns a heuristic evaluation of the given state
$\alpha \leftarrow$ Max's best score so far along the path from the initial state ▷ Initially MINSCORE
$\beta \leftarrow$ Max's worst score so far along the path from the initial state ▷ Initially MAXSCORE
$H \leftarrow$ History ▷ Initially $H \leftarrow \{\}$. Don't copy H: it's shared throughout the search.

if terminal(S) or *depth* $\geq$ *maxdepth* **then**
    return evaluate(S)
else if ismaxsturn(S) **then**
    for each child $C \in$ expand(S) **do**
        mm $\leftarrow$ Memoized Min-Max with α-β Pruning with C, *depth* $+1$, *maxdepth*, *ismaxsturn*, *expand*, *terminal*, *evaluate*, α, β, H
        $\alpha \leftarrow \max(\alpha, mm)$
        if $\alpha \geq \beta$ **then** ▷ Cut off: don't bother searching any more
            return β
    return α
else
    for each child $C \in$ expand(S) **do**
        mm $\leftarrow$ Memoized Min-Max with α-β Pruning with C, *depth* $+1$, *maxdepth*, *ismaxsturn*, *expand*, *terminal*, *evaluate*, α, β, H
        $\beta \leftarrow \min(\beta, mm)$
        if $\alpha \geq \beta$ **then** ▷ Cut off: don't bother searching any more
            return α
    return β

This function is called by, and calls, our memoizer function. The memoizer function is the top-level (it's what you'd call to get things going):

Algorithm 31 *Memoized Min-Max with α-β Pruning*

```
S ← current game state
depth ← current depth                                                    ▷ Initially 0
maxdepth ← maximum desired depth
ismaxsturn(...) ← returns true if it's Max's turn in a given state
terminal(...) ← returns true if the game is over in a given state
expand(...) ← expands a state into all possible successor states
evaluate(...) ← returns a heuristic evaluation of the given state
α ← Max's best score so far along the path from the initial state          ▷ Initially MINSCORE
β ← Max's worst score so far along the path from the initial state         ▷ Initially MAXSCORE
H ← History          ▷ Initially H ← {}. Don't copy H: it's shared throughout the search.

if S is a key in H then
    return the value stored in H keyed by S
else
    score ← Subsidiary Min-Max with α-β Pruning with S, depth, maxdepth, ismaxsturn,
                                                    expand, terminal, evaluate, α, β, H
    store score in H keyed by S
    return score
```

Nim and Dynamic Programming Anything you solve with memoization can also be solved with dynamic programming as well. Recall that the idea in dynamic programming is to first solve all the simple problems (let's call them "level 1"), then store their solutions, then solve more complex problems which rely on them ("level 2"), then store *their* solutions, then solve more complex problems ("level 3"), and so on. This approach works if the complex problems have a high degree of interdependency on the earlier simpler problems (complex problems share and reuse the same simple problems again and again). Just like memoization, the idea is to avoid re-solving those simple problems over and over.

Dynamic programming is a **bottom-up** approach, while memoization is a **top-down** approach. The two methods have the same computational complexity but not necessarily the same memory usage. In the worst case, memoization will have stored every single possible solution. But in some occasional tasks, to solve the "level n" problems you only need to know the answers to the "level $n - m$" problems, for some $0 < m \leq a$, and not any earlier ones. This allows you to throw away earlier levels you're no longer using. If this is the case for your task, then dynamic programming can achieve significant memory savings by carefully throwing away that data.

Tic-Tac-Toe is an example of one such task. Another example is a game called **One-Heap Nim**. Nim is a family of two-player games where each player takes turns removing stones from one or more *heaps* (piles of stones). Usually, the person to remove the very last stone is the winner. In some versions of Nim, known as **misère** versions, the person to remove the very last stone is the *loser*.

The most common versions of Nim are One-Heap and **Three-Heap Nim**.[72] In Three-Heap Nim there are three piles of stones. When it's your turn, you must remove at least one stone from exactly

[72]Three-Heap Nim doesn't have the property that gives dynamic programming a memory advantage over memoization.

one heap. You can remove as many stones as you like, as long as they're from a single heap. If a heap is empty, of course, you can no longer remove stones from it. In contrast, in One-Heap Nim, there is a single heap, but you're only allowed to remove one, two, or three stones from it at a time.

Figure 51 shows the game graph for One-Heap Nim, starting with a heap of size six, with either player possibly going first. As you can see, Nim has the properties we're looking for: a small set of game states, no cycles, and a rich set of edges and sharing of earlier states among the later states. And to distinguish it from memoization: to determine the game score when there are n stones in the heap, one only needs to know the game scores when there are $n - m$ stones in the heap, where $0 < m \leq 3$, since at most three stones can be removed from a heap at a time.

To solve this with dynamic programming, we first figure out the scores for games with 0 stones in the heap, then games with 1 stone in the heap, then games with 2 stones in the heap, and so on.

Algorithm 32 *Dynamic Programming Min-Max*

```
1: F ← current state                          ▷ We'll work our way backwards to the current state, then quit
2: S ← {S_1, ...} ← current game states, initially all terminal game states
3: ismaxsturn(...) ← returns true if it's Max's turn in a given state
4: parents(...) ← expands a state into all possible predecessor states
5: expand(...) ← expands a state into all possible successor states
6: evaluate(...) ← returns an evaluation of the game state                    ▷ Not heuristic

7: H ← History                          ▷ A map of some sort (hash table?). Initially H ← {}
8: for each S_i ∈ S do                                          ▷ evaluate the terminal states
9:     score ← evaluate(S_i)
10:    add score to H, keyed by S_i
11: loop                                                        ▷ evaluate the next level states
12:    Optionally remove from H all states (keys) which are "too old" to use in finding newer solutions
13:    S ← parents(S)
14:    Remove from S all states which are keys in H                    ▷ Already computed those
15:    if F ∈ S then                                                          ▷ we're done!
16:        return H
17:    else
18:        for each S_i ∈ S do
19:            if ismaxsturn(S_i) then
20:                score ← MINSCORE
21:                for each child C ∈ expand(S_i) do
22:                    mm ← score in H keyed by C
23:                    score ← max(score, mm)
24:                add score to H, keyed by S_i
25:            else
26:                score ← MAXSCORE
27:                for each child C ∈ expand(S_i) do
28:                    mm ← score in H keyed by C
29:                    score ← min(score, mm)
30:                add score to H, keyed by S_i
```

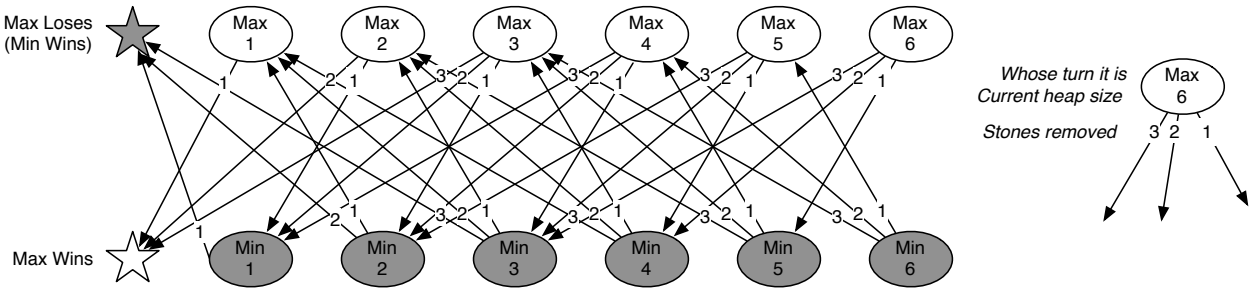


Figure 51 Full game graph of One-Heap Nim, "normal" play, with an initial heap of six stones. Game play can start with either Min or Max. In "normal" play, whoever removes the final stone wins. In contrast, in "misère" play, whoever removes the final stone loses. The misère graph is identical except that the two end states (the stars) are swapped.

Note that this isn't a heuristic function any more: since we're starting at the endgame and working our way back to the current state, there's no need for a heuristic. There's also no depth or maxdepth. But we did need to add a new function, called parents(...), which is more or less the opposite of expand(...). Rather than produce the children of a state S, it produces the *parents* of a state S. This is what lets us work our way backwards.

This version of Min-Max doesn't return a score: it returns H, which contains *scores for all game states up to but not including* H. So we'd use it like this:

Algorithm 33 *Make a Move (Dynamic Programming)*

```
1: F ← current game state
2: ismaxsturn(...) ← returns true if it's Max's turn in a given state
3: terminal(...) ← returns true if the game is over in a given state
4: parents(...) ← expands a state into all possible predecessor states
5: expand(...) ← expands a state into all possible successor states
6: evaluate(...) ← returns an evaluation of the game state                ▷ Not heuristic

7: best ← □
8: bestscore ← MINSCORE
9: H ← Dynamic Programming Min-Max with C, ismaxsturn, expand, terminal, evaluate
10: for each child C ∈ expand(S) do
11:     score ← value in H keyed by C
12:     if score > bestscore then
13:         best ← C
14:         bestscore ← score
15: make the move which leads to best
```

Removing old elements from H can be challenging. You don't want to have to go through *all* of H every iteration: this could be pretty expensive, and in this case it'd make more sense, probably, to go with memoization. Usually dynamic programming is applied to problems where the elements in H at any time are fixed in number: this is certainly the case for One-Heap Nim, where H will need no more than six previous game states. Then you can just implement H as an array or queue and remove the last n elements from it each time around. Dynamic programming also only makes sense if you can reasonably process **every single game state** from the terminal states clear to your current game state situation. Only for a very few games is this really true.

So there you have it: two ways of exploiting shared child states. In truth, if you have the memory for it, I'd do memoization.

9.7 Why Go is Hard

As far as AI is concerned, what would it mean for a game to be "solved"? In one sense, a "solved" game is one for which we know how to program a computer to guarantee that it always wins if it goes first, probably because it searches clear to the end of the game and figures out the winning strategy. But there's a looser notion of "solved": a game is solved if computers can beat any human alive at the game.

Using this second sense of "solved", there is only one perfect-information, two-agent, zero-sum board game that hasn't been solved. **That game is Go.**

Here is not the place to teach how to play Go. It's an ancient and fascinating game, that's deeper than chess but has relatively simple rules. Look it up! But Go presents a special challenge for AI researchers. The reason for this is mired in history. Very early in AI, back when AI meant "reasoning" or "search", researchers thought that chess would be a good benchmark problem. After all, chess represents an achievement in man's ability to reason. But it turns out that computers are *really good* at reasoning, using heuristic techniques such as α-β pruning or A*. They're much better than *we* are, in fact, and it's easy to program them that way. What's worse, it turns out that the α-β algorithm doesn't figure out chess moves like chess masters do. We naturally *imagine* that chess masters work through all the possibilities. But they don't. Instead, a chess master has a professional "batting eye" which he uses to whittle down the moves under consideration to just a very few. He then works out the consequences of each of those moves.

This batting eye is a result of how humans *really* play games like Chess: using the sophisticated visual pattern recognition systems in our brains, seeing connections and interrelationships. The problem is, we don't have a good idea of how to get a computer to do play in the same fashion. Thus while computers can beat humans in chess, the chess problem didn't really teach AI researchers all that much about how to program computers to think like humans. For the most part, chess is no longer considered an interesting problem. It was only a matter of time before Deep Blue would beat Kasparov.

That's why some hope is held out for Go. Humans use powerful pattern-recognition capabilities to play Go. But more importantly, you can't easily apply heuristic search algorithms like α-β pruning to Go. That's because while chess has an average branching factor of about 35, Go (played on a 19x19 board) has a branching factor of under 361. In chess, a decent system nowadays can search about 17 plies deep before it runs out of time, if it uses clever heuristics. But to search 17 plies deep in Go, the number of leaf-node states is astronomical, even with good heuristics to whittle down the stupid moves. And we don't have any good idea yet of what constitutes a "stupid" move, so ordering the children is problematic as well. To make matters worse, in Go, you really need to search a lot deeper before you get any sense of who's ahead or behind.

The critical factors for α-β pruning are (1) the branching factor and (2) how deep you have to search before your evaluation function is any good. Go hurts the computer in both of these categories. Having a good evaluation function is absolutely critical as well, and in Go it's still not obvious what good state evaluation functions look like.

But this is a good thing! Because *humans* can still play Go pretty well. Researchers hold out the hope that Go will force us, once and for all, to get computers to "play the game" more like humans do.