Unit 3
Adversarial Games

Context:

An important category of AI techniques is dedicated to the study of ***adversarial games***. These encompass well known problems such as playing chess and Go, but beyond games, the general problem of finding a winning strategy in a competitive task has wide applicability.

Problem definition

We will focus for the time being on 2-player games (the methods we cover can be generalized to multi-player games).

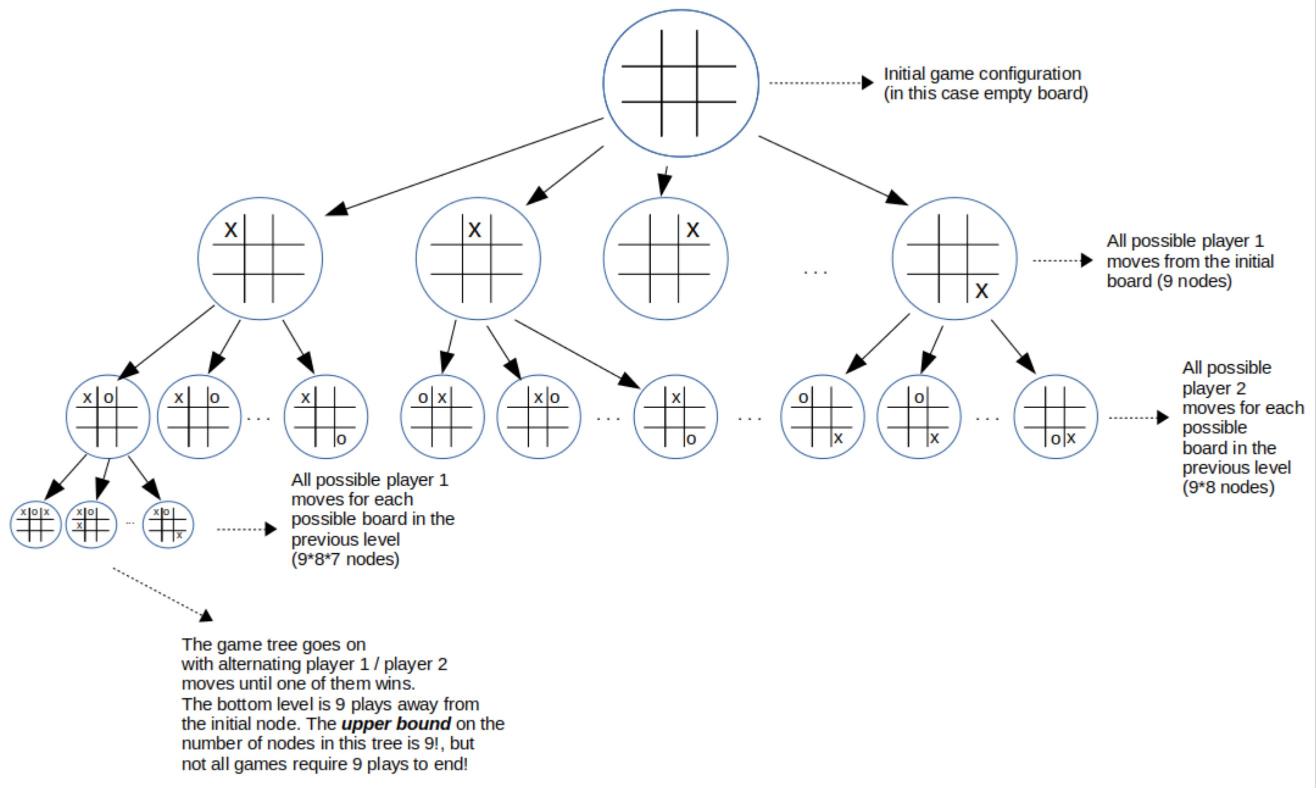
- Both players are trying to ‘win’ at some pre-defined task with a well defined set of rules
- Players take turns making ‘moves’, that is, they decide upon, and carry out some action during their turn that is intended to help them win the game.
- The game is competitive, if one player wins, the other one must lose

Classic algorithms for adversarial games formulate the above as a special type of search problem.

Each state in this search problem is a ***possible configuration for the game,*** and the game itself forms a ***tree*** with the ***initial configuration*** at the top, and each successive level shows possible configurations that can be reached via different combinations of moves of the two players.

As an example, the tree below illustrates the game tree for the very simple game tic-tac-toe:

(c) 2020 – F. Estrada



Suppose ***we are player 1***, the problem we are faced with is finding ***which move to make for the first play of the game*** so as to ***maximize our chance of winning***.

In order to do this, we need to look ***down the tree*** and find ***paths*** that lead to configurations in which we win the game. If possible, we would like to bring the game to a point from which there is no way for player 2 to win! In tic-tac-toe, if your opponent plays well, this is impossible to do, but for other games, with a suitable strategy, it may be possible for a player to know they will win the game several moves before this actually happens (or even from the start of the game!).

Of course, life is not so easy – while player 1 is busily trying to build a path toward a configuration in which player 1 wins, player 2 is attempting to do exactly the opposite: build a path toward a configuration in which player 2 wins.

This allows us to formalize our problem.

- We define a ***utility function*** that evaluates game configurations in terms of how good they are for each player. A very simple one for the tic-tac-toe game above could be:

    +1 for a configuration in which player 1 wins
    -1 for a configuration in which player 2 wins
    0 for any configuration that is not a win for either player

this may be enough for a simple game like tic-tac-toe, but for more complex games the utility function should be able to evaluate how ***close*** a partial game configuration is to a win by either of the players, giving positive values to configurations favourable for player 1, and negative values to configurations that are good for player 2.

- The search process is called ***MiniMax*** – when it's player 1's turn to play, player 1 will choose a move that ***maximizes*** the value of the utility function. When it's player 2's turn, they will choose a move that ***minimizes*** the value of the utility function.

- Because players alternate turns, the maximization and minimization steps are also alternated, thus ***minimax***.

But what does it mean to choose a move that maximizes (or minimizes) utility for a specific move?

Naive Strategy:

- During its turn, each player evaluates ***all possible moves*** it can make using the ***utility function*** and chooses the move that maximizes (or minimizes) this value according to its goal.

It's a trivial process, and relies entirely on the utility function doing a good job of evaluating partial configurations. It, sadly, won't do very well for complex games like chess (can you think of a utility function that can tell you reliably what is the best possible opening move across all possible games that could be played?) .

The utility function, however good, can not capture the complexity of a game or account for the actions of the opposing player. Therefore, we need a stronger method for developing a winning strategy.

MiniMax process

Played 1 needs to determine the move that ***maximizes its utility***, but the utility of the moves that are open to player 1 ***depends on what player 2 will do***.

So here's what the process looks like:

(A) Player 1 turn:

Look at every possible move it can make

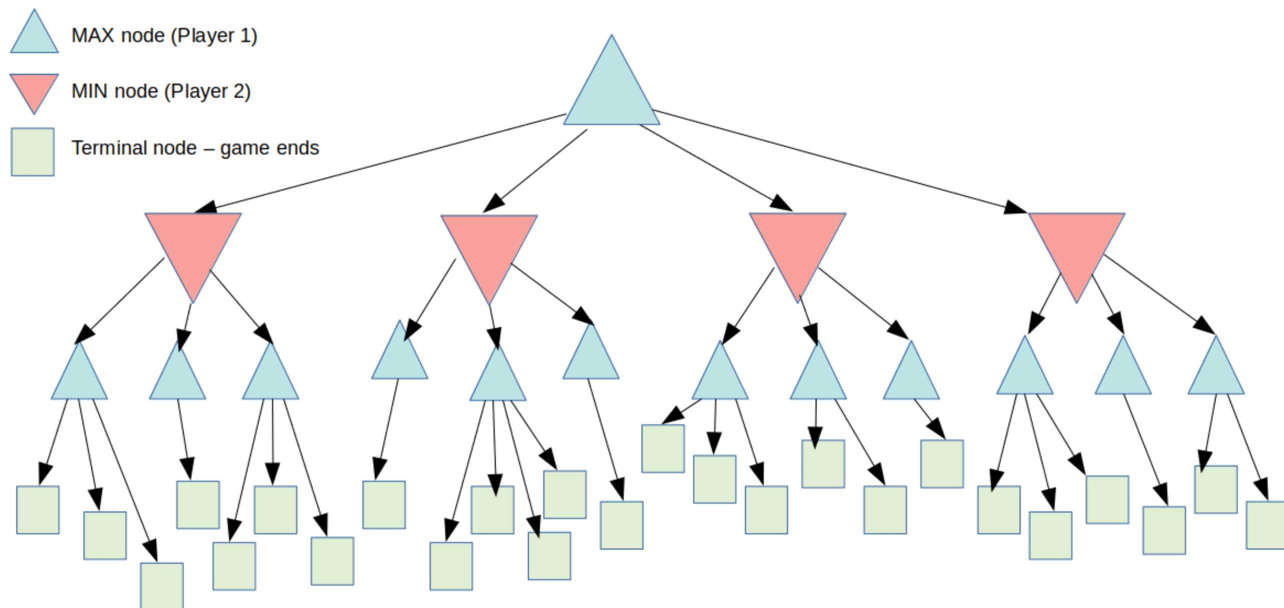
For each move, figures out ***what player 2 would do*** to win the game if player 1 makes that move.

This means looking at ***all possible ways player 2 could move*** and assuming player 2 will choose the one that ***minimizes utility***.

To do this, ***player 2*** needs to figure out ***what player 1 would do*** to win if player 2 chooses a particular move... and we're back to (A)!

(c) 2020 – F. Estrada

This leads to a ***MiniMax game tree*** that looks a lot like the tree shown above for tic-tac-toe, but with the addition of utility values based on the minimax process described above. Let's see how that works with an example:



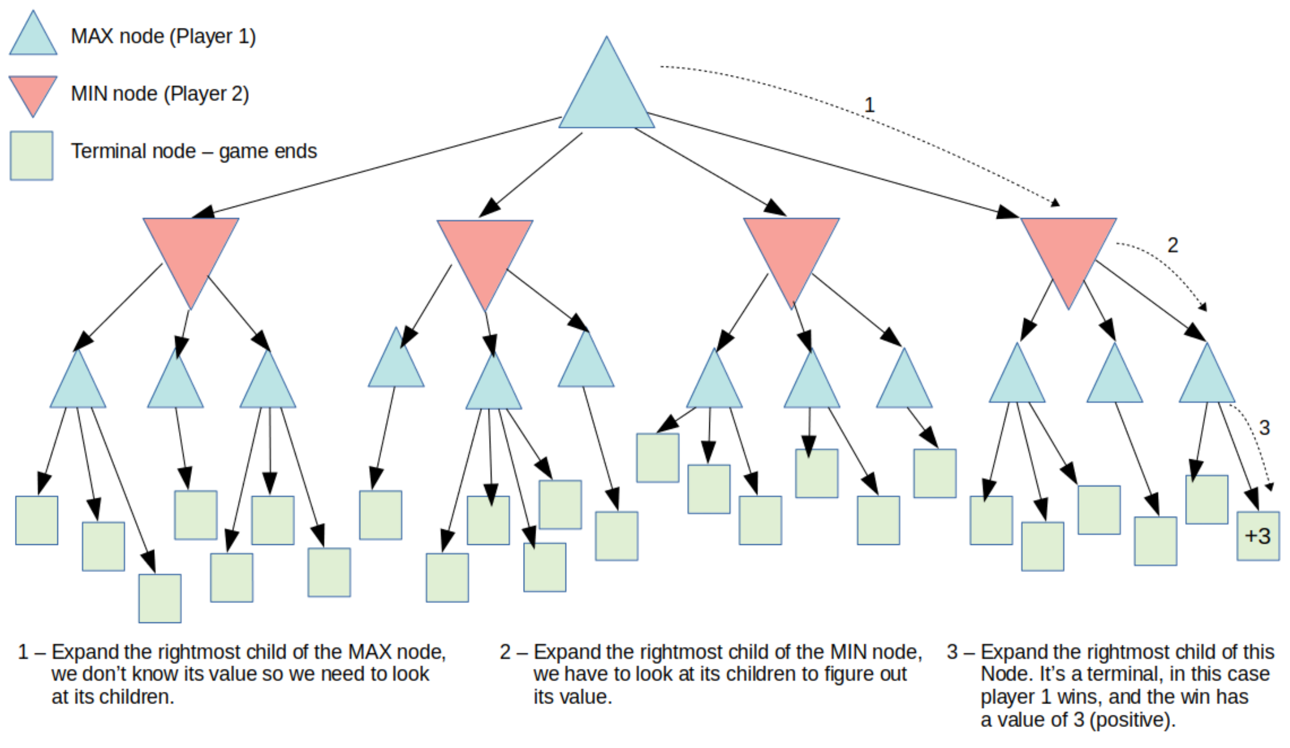
This corresponds to a very short game! Only three moves, one by player 1, and one by player 2, and one more for player 1 which results in the game ending. Most games we care about will have a lot more moves! But this is just an example so you can see how things work.

Note that each level corresponds to one of the players. Level 1 is made up exclusively of ***MAX*** nodes for player 1, level 2 has exclusively ***MIN*** nodes for player 2. Level 3 has all ***MAX*** nodes again, and if there were more levels they would keep alternating.

Player 1 needs to choose the move that maximizes its chance to win among its four possible opening moves. However, the value of each possible move depends on what player 2 will do.

Initially, player 1 has no real way to choose. But ***player 1 can do a search in the minimax tree***. The search will ***expand every node in the tree up to either the bottom of the tree, or some maximum pre-specified depth***. In this case, this is a shallow tree, so player 1 can look all the way to the bottom of the tree.

Once the search reaches a terminal node, we can evaluate which player won, by how much, and assign a value to the terminal node. Let's see how the search process works for the tree above:



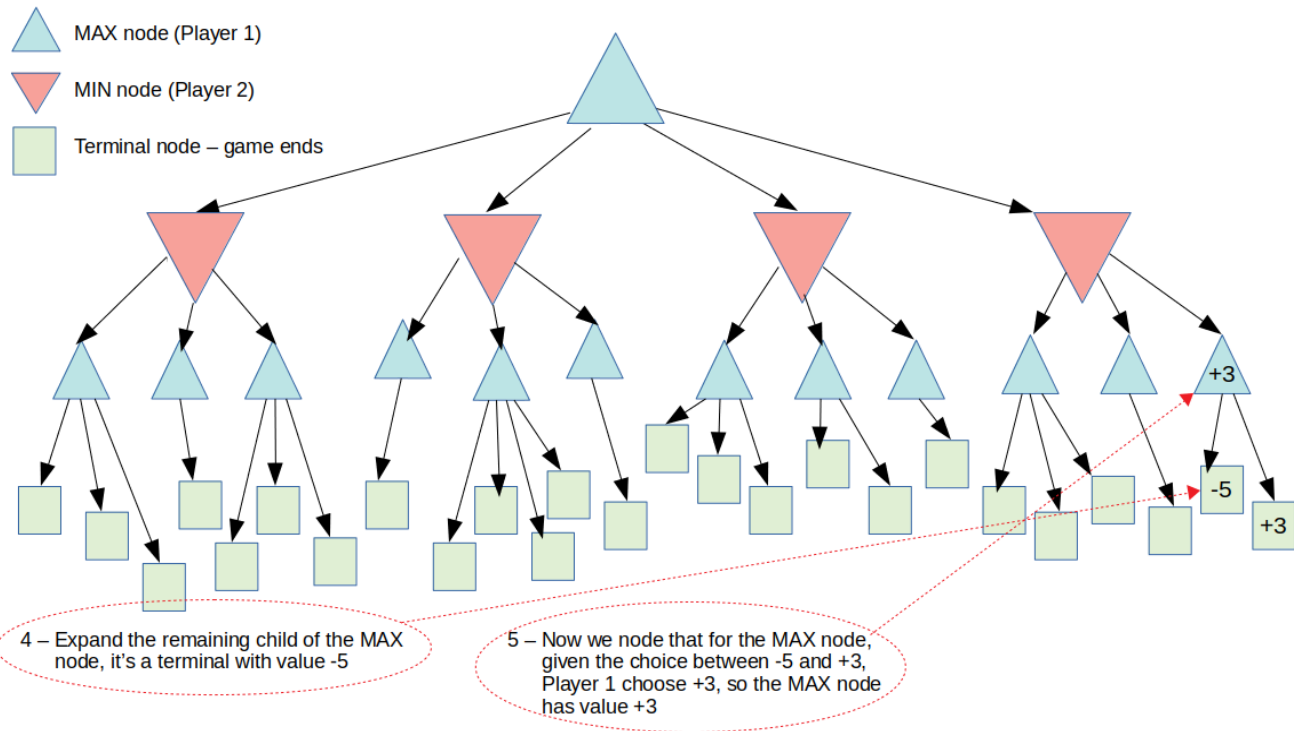
Let's look at Player 1's first move, and let's look at its choices from right to left. The rightmost node is a ***MIN*** node, but we don't know how much it's worth (because we don't yet know what Player 2 would choose at that level). So we have to expand the children of this node and find out.

We expand the rightmost possible move for this ***MIN*** node, it's another ***MAX*** node with no assigned value (yet). We need to look at its children!

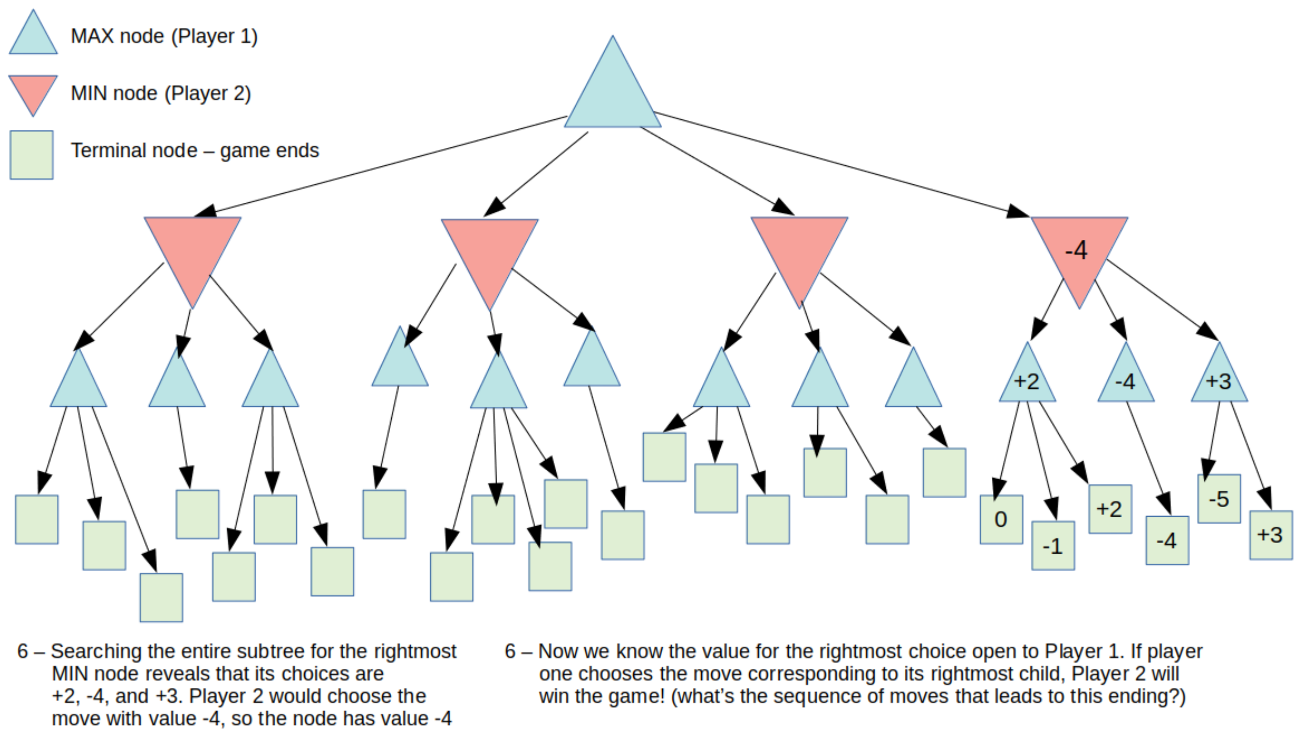
The first node we expand from there is a terminal node, we can evaluate who won, and by how much. For this example, let's assume it's a win for Player 1, with a value of 3 (Positive since player 1 wins).

We still can't assign a value to the ***MAX*** node, we have to look at its other children. There's one more, it's a terminal node, and let's imagine that in this ending, Player 2 wins, and the value of the node is -5.

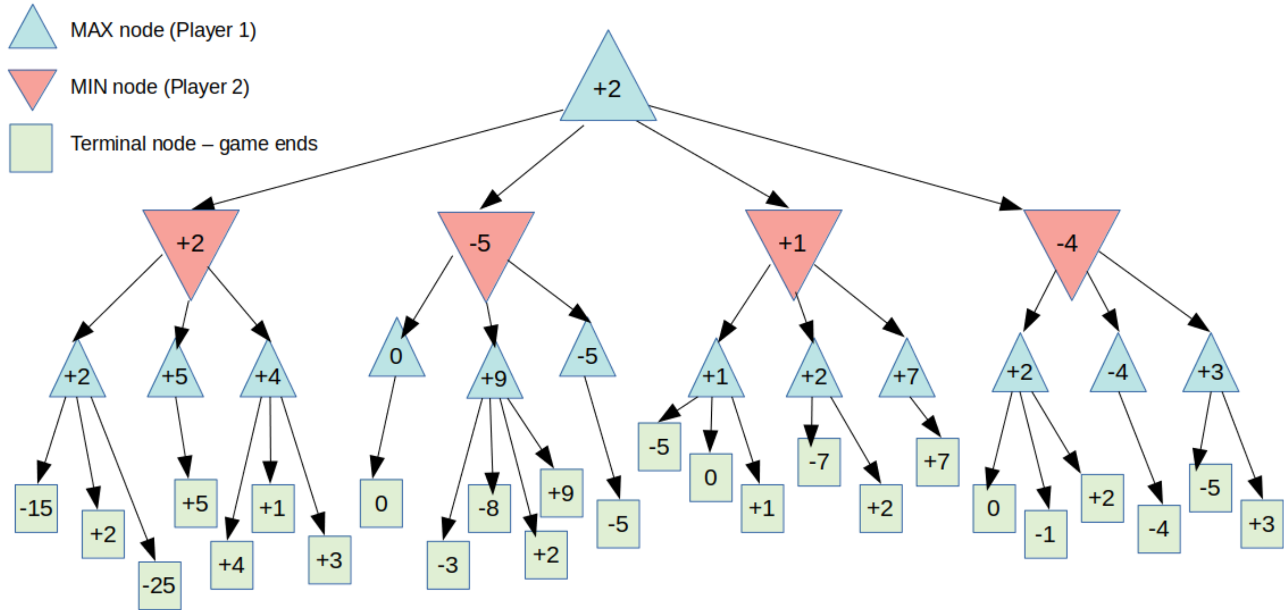
We now have values for ***all the children of the MAX node***. So we can assign a value to the ***MAX*** node itself. Which value should this node take? Given the options available to them, Player 1 will always choose the node with the ***highest value***. So given the choice between -5 and +3, Player 1 chooses +3. Therefore, the value for this ***MAX*** node is also +3: If we ever end up in this configuration, we know Player 1 will choose the move corresponding to its right child, and win the game.



We still can't assign a vale to the rightmost ***MIN*** node, we need to keep expanding its children. Let's see what happens once we have expanded all of the children of the rightmost ***MIN*** node:



We now know how the game ends if Player 1 chooses its rightmost move. Player 1 needs to repeat this process with all its children to determine the best move. If we carry the search process for every possible move for player 1 we will find:



Complete MINI-MAX tree, utility values propagate from the terminal nodes up the tree, at each level the corresponding player chooses either the MIN or the MAX of the choices open to them, and at the top (root), Player 1 knows it should choose the move corresponding to its leftmost child, which leads to Player 1 winning, in a configuration with a value of +2.

Notice that the actual process is a variation of DFS – we expand nodes downward until we find a terminal node. Then the information propagates back up the tree with each player choosing according to its preference (MIN or MAX) until we have enough information at the top of the tree to make a move.

Note that once we determine the value for the root node (+2 in the case above), we ***know even before the first move is made*** that Player 1 wins this game! Even if Player 2 plays the best possible game! (to see this, go back to the fully expanded tree above, and see what happens if Player 2 makes a mistake during its turn – things can only get better than +2 for Player 1!).

That seems boring right? Knowing the whole outcome of the game before we start? Does this search process seem ***intelligent*** to you?

Regardless of our opinion, for a game where you can expand the MINIMAX tree all the way to the terminal nodes, it is possible to determine the optimal move for either player, at any point in the game. A properly implemented MINIMAX search for such a game will play optimally.

Of course, life is not so easy – interesting games have a search tree that is too big to expand completely during a reasonable amount of time allowed to decide a move. So in practice, we often find we reach the ***maximum search depth*** before hitting terminal nodes that have a definite score.

(c) 2020 – F. Estrada

What happens if we reach the maximum expansion depth but the node there is ***not a terminal?*** This is where we need ***a good utility (scoring) function*** : We use this function to evaluate ***how good a partial game configuration is for either player, and assign a value to the node right there***.

The strength of a MINIMAX playing algorithm depends then on two factors:

- The available computing power (more computing power means we can expand more levels of the tree in the same amount of time) – each aditional level corresponds to looking one more move ahead to see what happens.

- The scoring function used to evaluate partial (non-terminal) configurations.

Even with a powerful computer, and a good scoring function, some games are just too complex (the possible outcomes and moves make the tree grow too fast) to be played well by a simple MINIMAX search.

In a moment we will see how we can use a bit of cleverness to reduce the amount of computation required by MINIMAX, thus allowing us to search deeper into the tree. But first, here’s a concise pseudocode description of the standard MINIMAX procedure.

```
MiniMax(C)        // Returns the MINIMAX value of node C

    // Check if we have to terminate the search and return a value
    // eval(C) is the utility function that return a value for a game
    // configuration, either partial, or terminal.
    if C is a terminal node, or max-depth reached, return eval(C)

    // Get the value of node C by checking all its children
    for each child i of C

        v[i]=MiniMax(i)          // Get MiniMax value for each child node

    if  C  is a MAX node, return max(v[i])
    else  return  min(v[i])
```

Alpha-Beta Pruning

The standard technique for reducing the amount of work that needs to be done to decide on a good move is called ***alpha-beta pruning***. It is part of a class of methods known as ***branch and bound***. Alpha-beta pruning is based on the following idea:

Suppose I am trying to establish the utility value of a ***max*** node.

The node has 5 children, and each of them is a ***min*** node.

Explore the subtree around child #1, that subtree returns a value of ***5***
(this already tells me that my parent ***max*** node will have a utility of ***at least 5***).

(c) 2020 – F. Estrada

Explore the subtree around child #2

But now we have information we didn't have before! We know the parent node has the option to choose a move with utility 5 (from child #1).

Child #2 is a ***min*** node – to evaluate its utility we start exploring its children.

The moment any of the children return a value <= 5 we can stop!

That is because since this is a ***min*** node, whatever else is returned by other children, this node can and will choose a value that is at most 5. Since the parent node already found a move with utility 5 – ***there's no chance this subtree will return a move that is better than what the parent already has.***

Therefore – no need to explore this subtree any further, we can return early!

The same reasoning applies if the parent node is a ***min*** node, with children that are ***max*** nodes, except now we stop searching when the children find at least one path with utility >= to what the parent has already found from other child nodes.

To formalize this idea, each node will now have two values attached to them:

alpha and ***beta*** – which are ***passed on by the parent node*** when we start exploring their subtree.

For the root node, ***alpha has an initial value that is a very large negative number, beta has an initial value that is a very large positive number***.

As we carry out the MiniMax search process:

If the node we're expanding is a ***MAX*** node

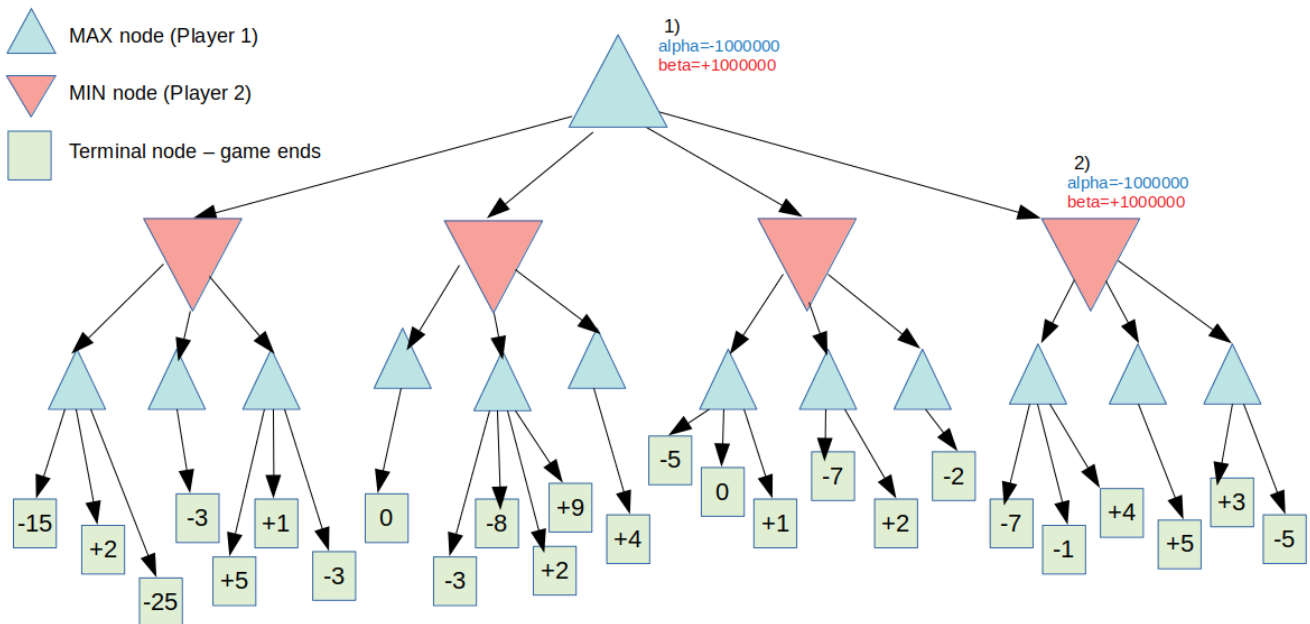
- When a child node returns a utility value > current ***alpha*** value, update the current ***alpha*** value
- if ***alpha >= beta, return alpha*** (we stop search right now!)

If, on the other hand, the node we're expanding is a ***MIN*** node

- When a child node returns a utility value < current ***beta*** value, update the current ***beta*** value
- if ***beta <= alpha, return beta*** (we stop search right now!)

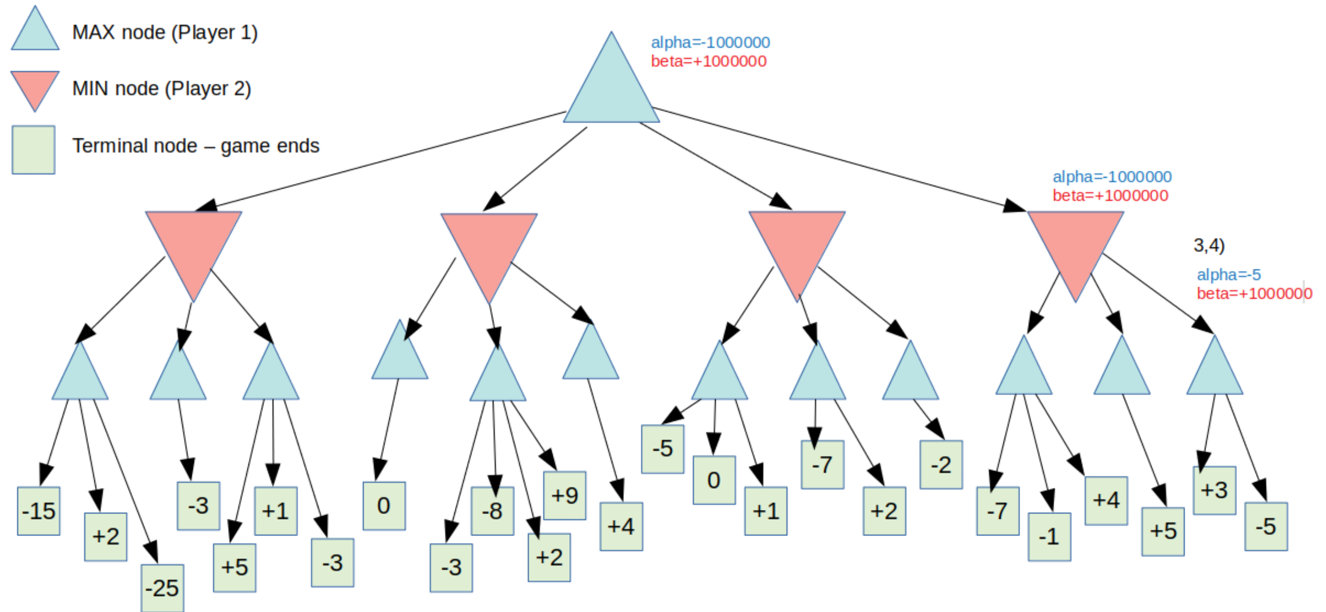
Let's see how this works with an example:

(c) 2020 – F. Estrada



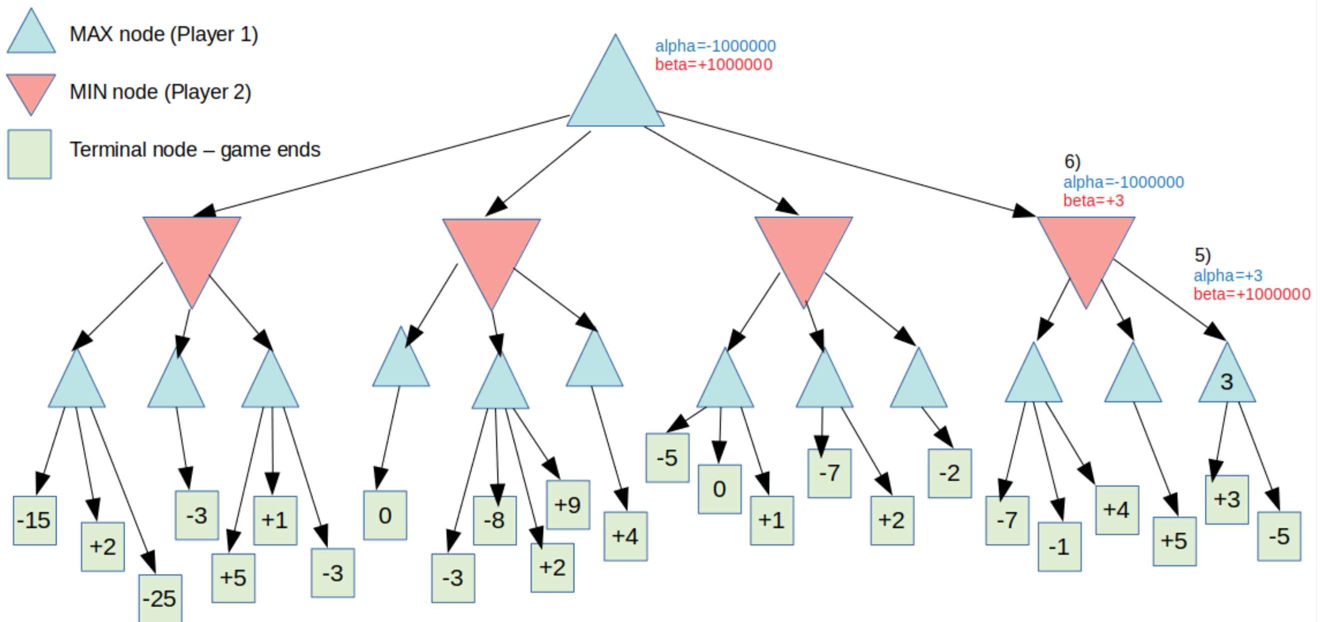
1) Initial step – we're going to perform MiniMax for the ***max*** node at the top – initial alpha and beta set to large negative and positive values respectively, and we will start expanding children from ***right*** to ***left***.

2) Rightmost child node inherits alpha and beta from parent.



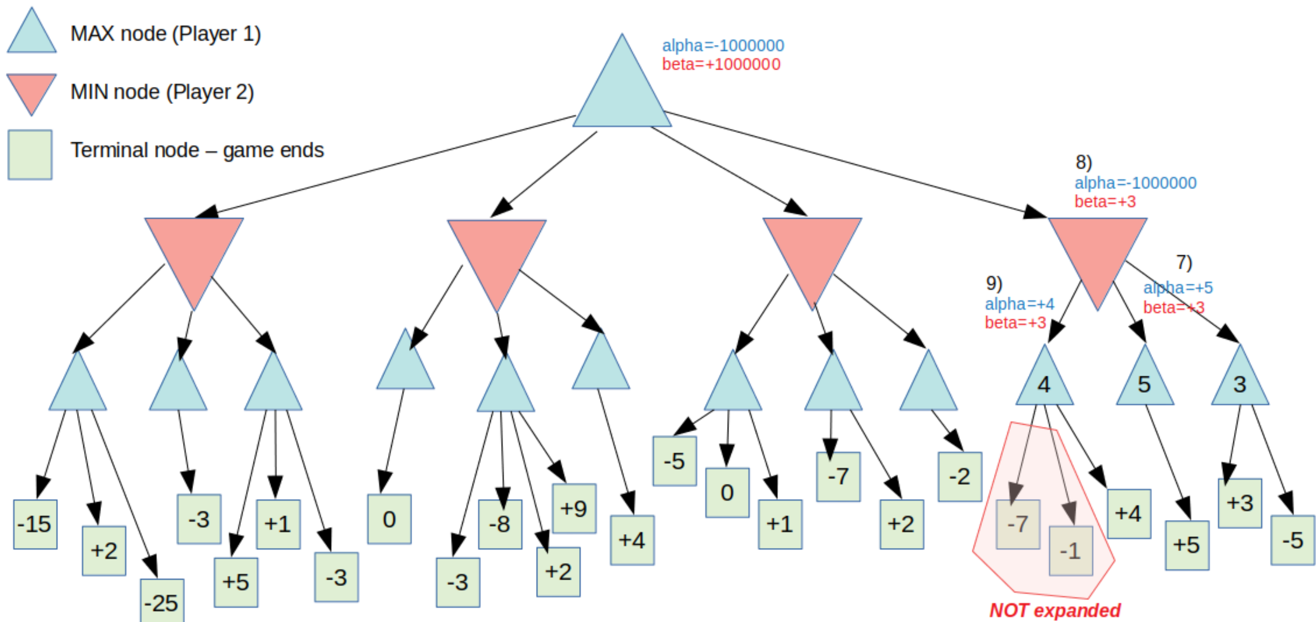
3) Expand the rightmost child of the ***min*** node – it inherits the alpha/beta from the parent

4) Expand the rightmost child of this node, it's a terminal with value -5 : This is a ***max*** node, whenever we find a value ***greated than alpha*** we have to update alpha, so alpha becomes -5



5) Expand the next child of this node, it is a terminal with value +3, which is higher than the current alpha (-5) – so we update alpha to +3

6) This node now returns its utility (+3) to its parent, the parent is a ***min*** node, whenever this node receives a value ***less than the current beta,*** it updates beta, so now beta=+3



7) Expand the next child of the ***min*** node, it inherits ***alpha=-1000000, beta=+3*** from its parent, its only child returns +5, because it's a ***max*** node, it updates its alpha to +5 and returns a +5 to the ***min*** node.

8) The ***min*** node receives a (+5) from its middle child, this is greater than the current ***beta*** so we don't update beta.

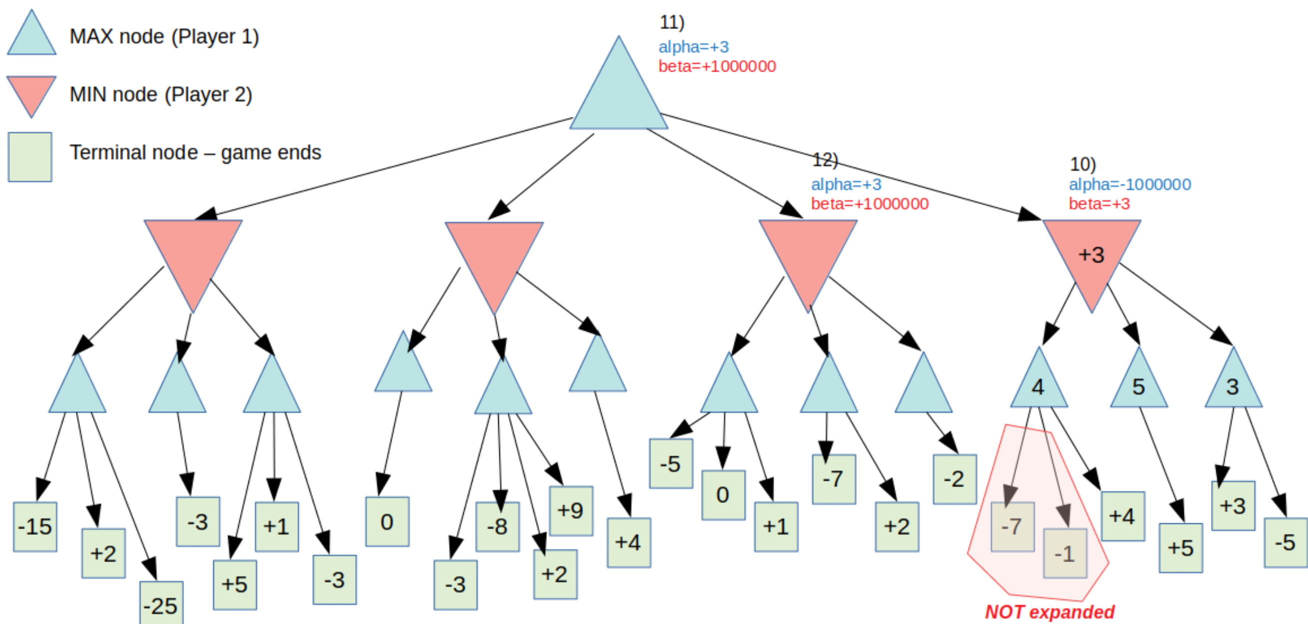
9) Expand the leftmost child – it inherits ***alpha=-1000000, beta=+3*** from its parent. Its rightmost child returns (+4) which updates ***alpha=+4***. ***Note that at this point alpha>=beta, so we stop and return alpha!*** (the remaining children are never expanded!)

It’s important to stop for a second at this point and think about what happened. ***Why*** does it make sense to stop here? there’s a -1 and a -7 yet to be explored, and those are better options for Player 2!

Yes, but their parent is a ***MAX*** node. It will never pick -1 or -7 since it already found a +4. Meanwhile, its parent, the ***MIN*** node ***already knows a move with utility +3*** which is better than the current +4 its leftmost child just discovered – ***therefore we already know that the min node will not select the leftmost child as its optimal move.*** We can stop!

Doesn’t look like we saved a lot – ***but remember this is just a little example, in a real MiniMax tree, the subtrees under the -1 and -7 could be huge!*** In the general case, not expanding a subtree will be a significant amount of work saved!

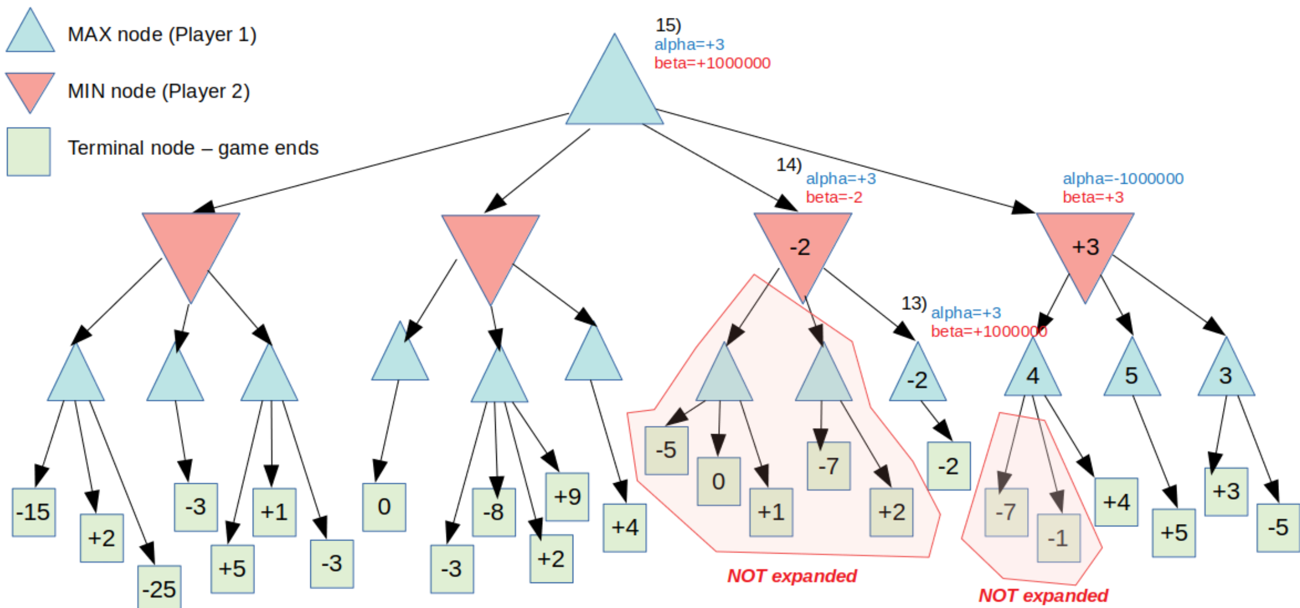
Let’s keep going...



10) The ***min*** node now knows its utility is +3, and returns that to its parent.

11) The ***max*** node receives a (+3) from its rightmost child, this is ***greater than the current alpha*** so it updates alpha to +3.

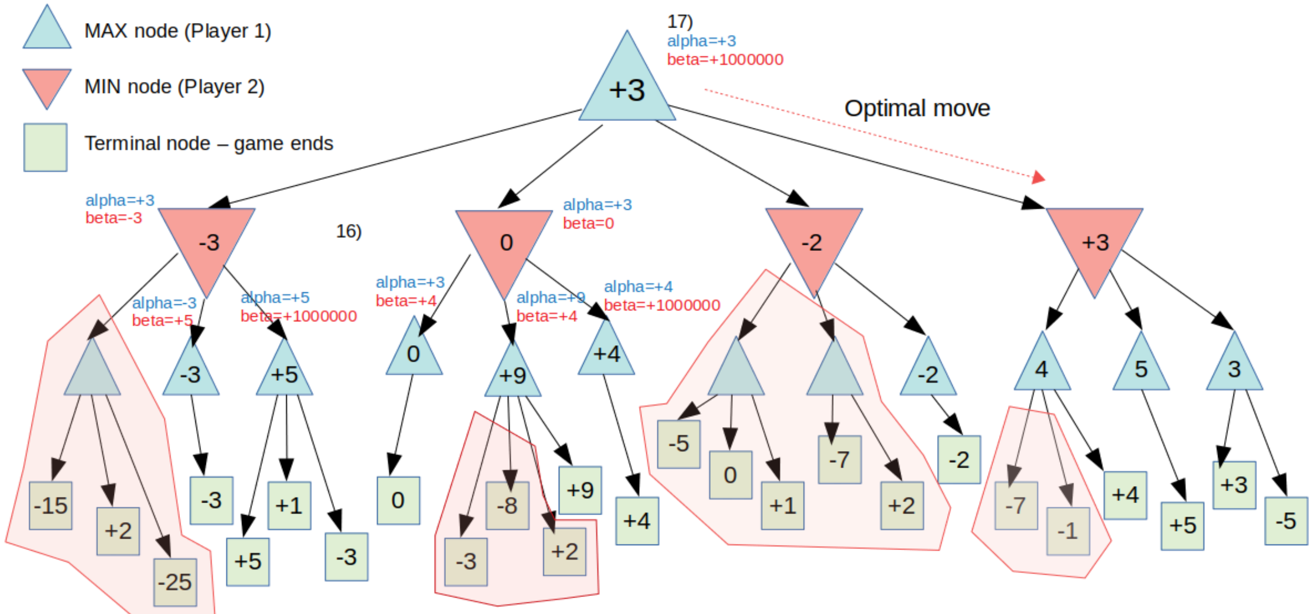
12) We now have to look at the next child node (a ***min*** node) which inherits alpha=+3, beta=1000000 from its parent



13) The rightmost child of the ***min*** node has a utility of (-2) – which updates its ***beta*** to (-2) and it returns a (-2) to its parent.

14) The ***min*** node receives a (-2) from its rightmost child. It's ***less than beta*** so we update ***beta=-2***. At this point, ***beta<=alpha so we stop searching and return beta=-2***. The remaining children of this ***min*** node are never expanded.

15) The ***max*** node at the top receives a (-2) from its child, it's less than the current alpha, so we do not update alpha.



16) We continue this process with the remaining children of the ***max*** node at the top. Make sure you understand why some nodes were expanded and some weren't – this will help you check that you have fully understood the pruning process. Only the final values of alpha and beta are shown.

17) The final value for ***alpha*** at the top is also the final ***utility value for the max node*** and indicates which choice Player 1 must make. In this case, the optimal choice has a value of (+3) and corresponds to the rightmost child.

(c) 2020 – F. Estrada

The end result of the MiniMax search with alpha-beta pruning is that we discover the ***same optimal choice*** we would find without the alpha-beta pruning, but with far less search. This allows us to look further down into the tree (i.e. look forward a larger number of moves) before deciding which move to make, compared with regular MiniMax.

What about games with chance?

Often, we may find adversarial games where there is an element of randomness or chance – for example, any game where after making a choice about what they want to do, players have to roll dice to figure out the result of their actions.

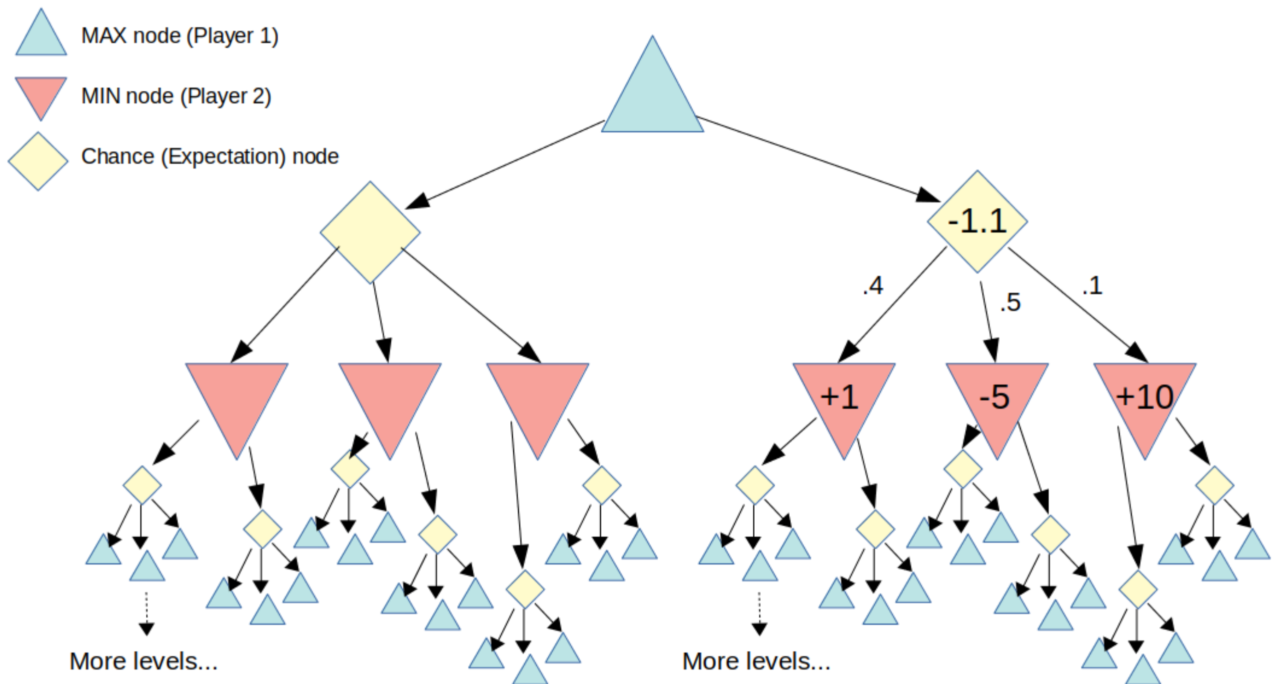
We can apply MiniMax to these types of games as well, though we have to be careful how we set up the utility evaluation process – it depends on chance. The critical insight here is that we can perform exactly the same type of MiniMax search if we replace the ***utility*** of our standard formulation with ***expected utility*** – that means, the ***weighted average*** of the utilities for ***all the possible outcomes*** of a given choice – weighted by the probability they occur.

For example, suppose I choose a specific move which ***could result in 3 different outcomes:***

a) Utility = +10, p=.1 (10% of the time we obtain a utility of +10)
b) Utility = -5, p=.5 (50% of the time we obtain a utility of -5)
c) Utility = +1, p=.4 (40% of the time we obtain a utility of +1)

The ***expected utility*** for this move is = (10*.1)+(-5*.5)+(1*.4)= -1.1

This process would be applied for each possible move, at each level of the tree. It makes for a somewhat bushy tree (each move now has several possible outcomes), but we can apply exactly the same process to this tree as we did above for deterministic games. Because we're using ***expected utility***, we call this process ***Expecti-MiniMax***.



Example of an ***Expecti-MiniMax*** tree, assume we have explored the lower levels of the tree and found the ***expected utility*** for the three ***min*** nodes on the right. Given these three values and the probabilities of each outcome, we can determine the expected utility for this move (represented with the yellow chance node) to be 1.1 – Note that the ***expected utility*** for each of the ***min*** nodes would be the result of computing ***expected utilities*** for each of their children, and choosing the smallest one.

MiniMax is not the only strategy for playing adversarial games well – many games proved too difficult to handle with MiniMax, and have required entirely different techniques. Reinforcement Learning is one domain of AI that has been applied to game playing, and that is where we are heading next.