CS3191 Section 3

Medium Games

Andrea Schalk

Department of Computer Science, University of Manchester

Medium games

The topic of Section 3 are medium games.

For a game to be medium-sized it has to be larger than small—in other words, it should be so large that it is not feasible to describe it *via* the Players' strategies.

It will become clear in the course of this section when a game is, in fact large.

The idea for this section is to employ computer power to help us find good strategies for a game. To some extent we will be building on our theory of games developed in Sections 1 and 2, since the notion of game tree will still be present.

Algorithmic point of view

Algorithms

In order to solve a small game, the following tasks have to be performed:

- generate all the strategies for all the players; ✓
- calculate the pay-offs when playing the various strategies against each other; ✓
- find equilibrium points. X

Beyond small games

How do games grow?

Question. How does the number of positions grow with the size of the game tree? How does the number of strategies (added up over all players) grow with the size of the game tree?

If the tree branches at least into two at most decision points then for a tree of height m there are about

$$2^{m+1} - 1$$

decision points, that is their number grows exponentially. To count the number of strategies, we need to know how many decision points each player has (nodes when it is his turn), and how they relate.

How do games grow?

We look at an example: Let us assume we have a complete binary tree where two players move alternatingly.

height	no pos.	strats for P1	strats for P2	all strats
0	1	1	1	2
1	3	2	1	3
2	7	2	4	6
3	15	8	4	12
4	31	8	64	72
5	63	128	64	192
6	127	128	16384	16512
7	255	32768	16384	49152
8	511	32768	1073741824	1073774592

What to do?

The fact that the number of strategies in a game typically grows exponentially makes it quite clear that we soon move out of the realms of games that can be solved with the methods from Section 2.

The aim for this section is to find good strategies without creating all strategies first.

For that we assume that the game tree is given in a computer.

However, our algorithms will not require that the game is present in memory at all times—it is sufficient if it can be created on demand.

The minimax algorithm

Values

For a game of complete information we define the value of a position p for Player X to be the pay-off he can guarantee for himself at the very least. (This harkens back to Scottie's and Amelia's original analysis, which we will take to its natural conclusion in this section.)

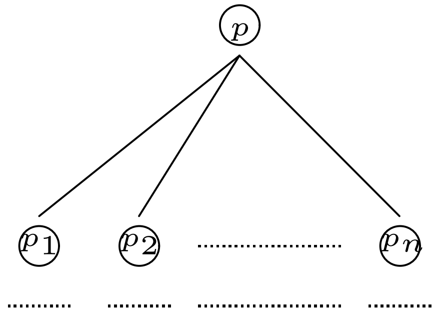
Note that the player has to assume that the others will do their worst in order for the guarantee to work.

Question. Why do we insist on having a game of complete information here?

We can deal with elements of chance if we stick to the expected pay-off as before.

Calculating the value

The following proposition shows us how to calculate the value of a position for a given player.



Proposition 3.1 *Let p be a position with $p_1, p_2, \ldots, p_m$ the positions which can be reached from p with one move (see the figure). Then the value for player X of p, $v_X(p)$, is*

$$v_X(p) = \begin{cases} p_X(p) & p \text{ is final} \\ \max_{1 \leq i \leq m} v_X(p_i) & X \text{ to move at } p \\ \min_{1 \leq i \leq m} v_X(p_i) & Y \neq X \text{ to move at } p \\ q_1 v_X(p_1) + q_2 v_X(p_2) + \cdots + q_n v_X(p_n) & \text{chance move at } p; \\ & q_i\text{: probability for } p_i \end{cases}$$

Proof

$$v_X(p) = \begin{cases} p_X(p) & p \text{ is final} \\ \max_{1 \leq i \leq m} v_X(p_i) & X \text{ to move at } p \\ \min_{1 \leq i \leq m} v_X(p_i) & Y \neq X \text{ to move at } p \\ q_1 v_X(p_1) + q_2 v_X(p_2) + \cdots + q_n v_X(p_n) & \text{chance move at } p; \\ & q_i\text{: probability for } p_i \end{cases}$$

This is really obvious from the definition:

If it is Player X's turn at p then he can choose the position with the maximum value and from there use a strategy to achieve that value.

If it is some other player's turn he cannot do better than guarantee the value that he might get in the worst case, which is the minimum of the values that can be reached from p with one move.

If it is a chance move then the minimal expected pay-off from p is

$$q_1 v_X(p_1) + q_2 v_X(p_2) + \cdots + q_m v_X(p_m).$$

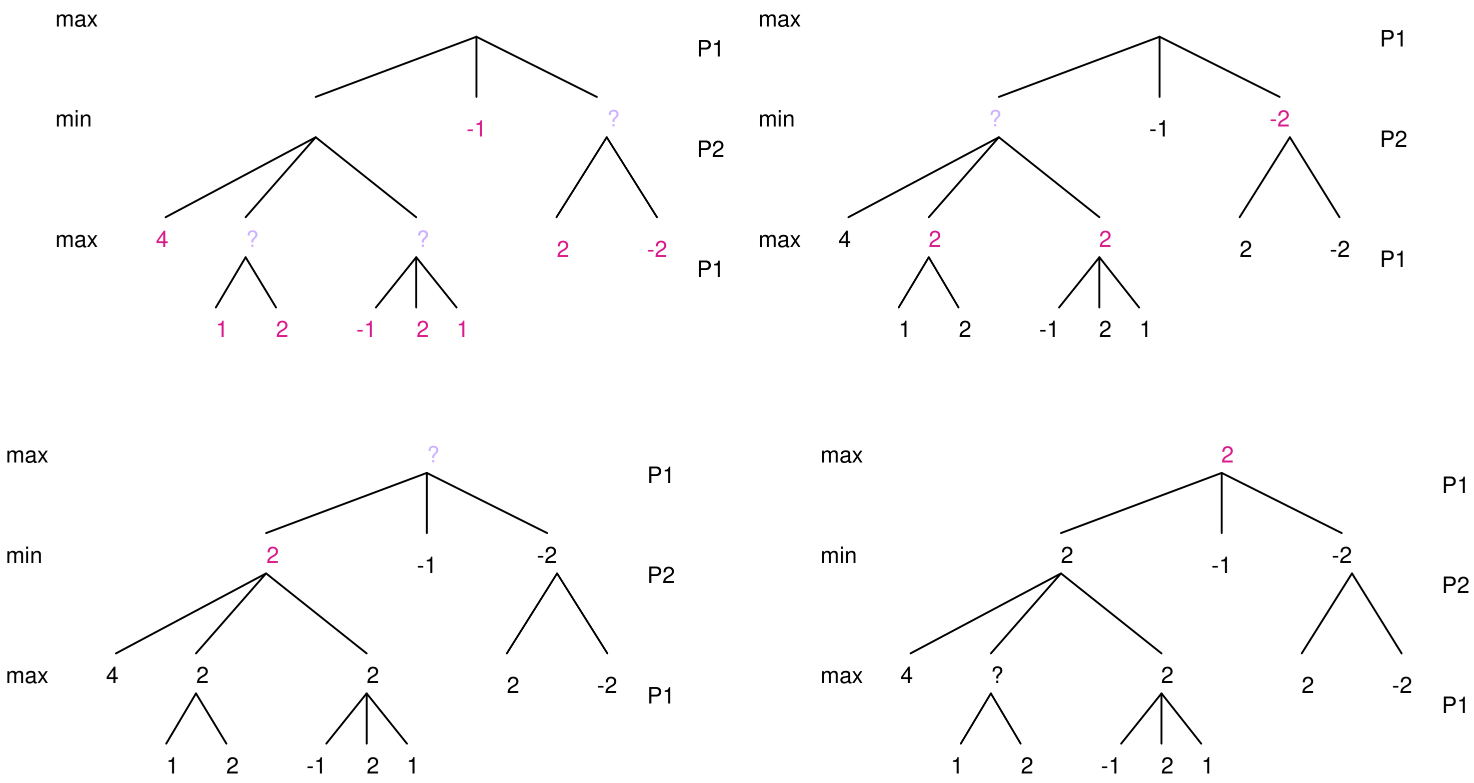
The values of a game

Note that unless we have a 2-person zero-sum game there will not be a unique value which we can take as relevant for all the players.

The various values for the root for each player give the pay-off they can guarantee for themselves in the worst case. Of course, they might do better when the game is played!

We have given a recursive algorithm—in order to find the value (for a player) of some node we first have to know the value (for that player) of all its children. To find their values we have to have the values of all their children, and so forth until we reach a leaf.

How to find the value–bottom-up



Alternatives?

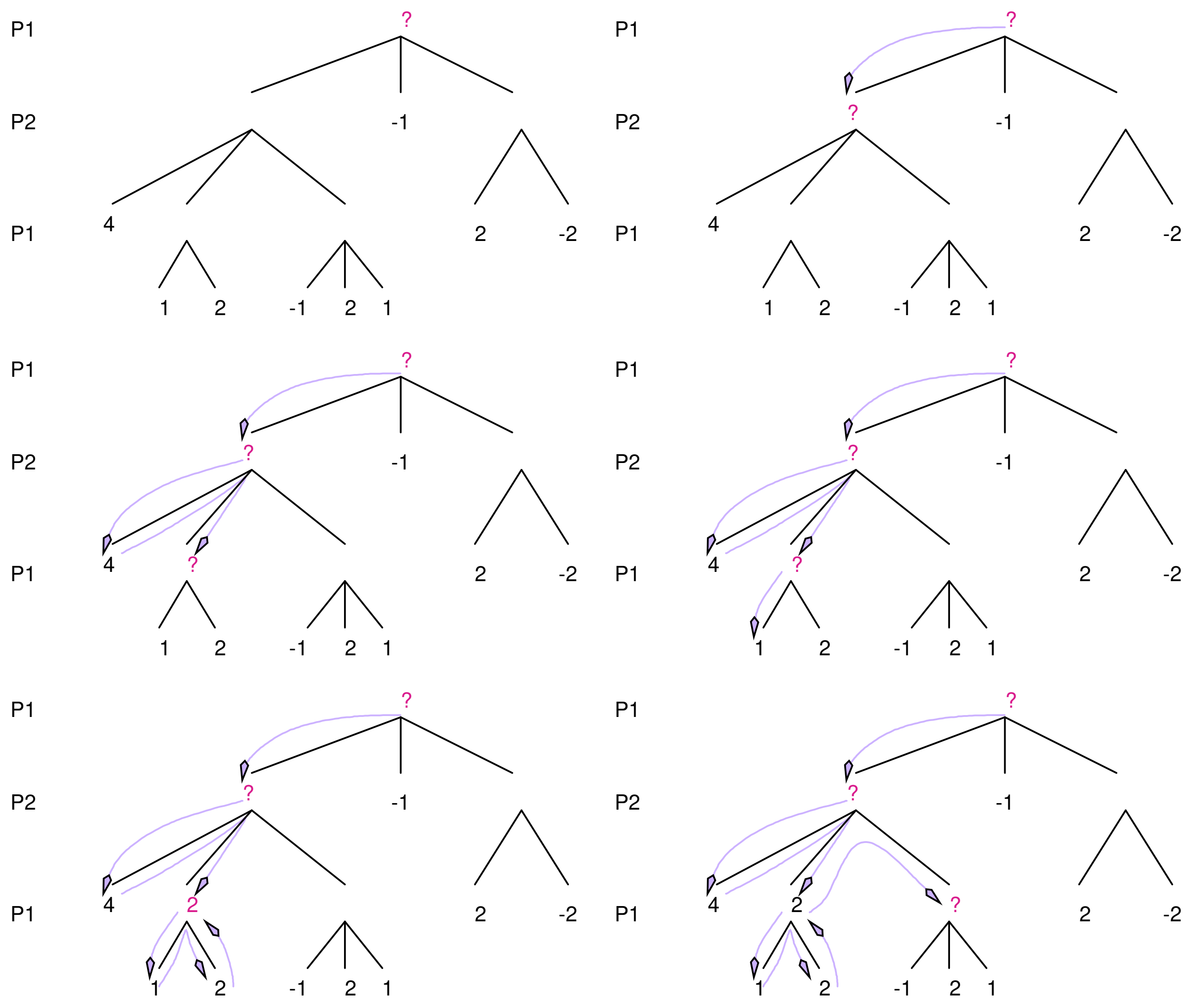
Question. How does a program implementing this algorithm traverse the tree?

This would have to be implemented using breadth-first traversal. This is usually not a very efficient way of traversing a tree, in particular one that is being created as it is being searched!

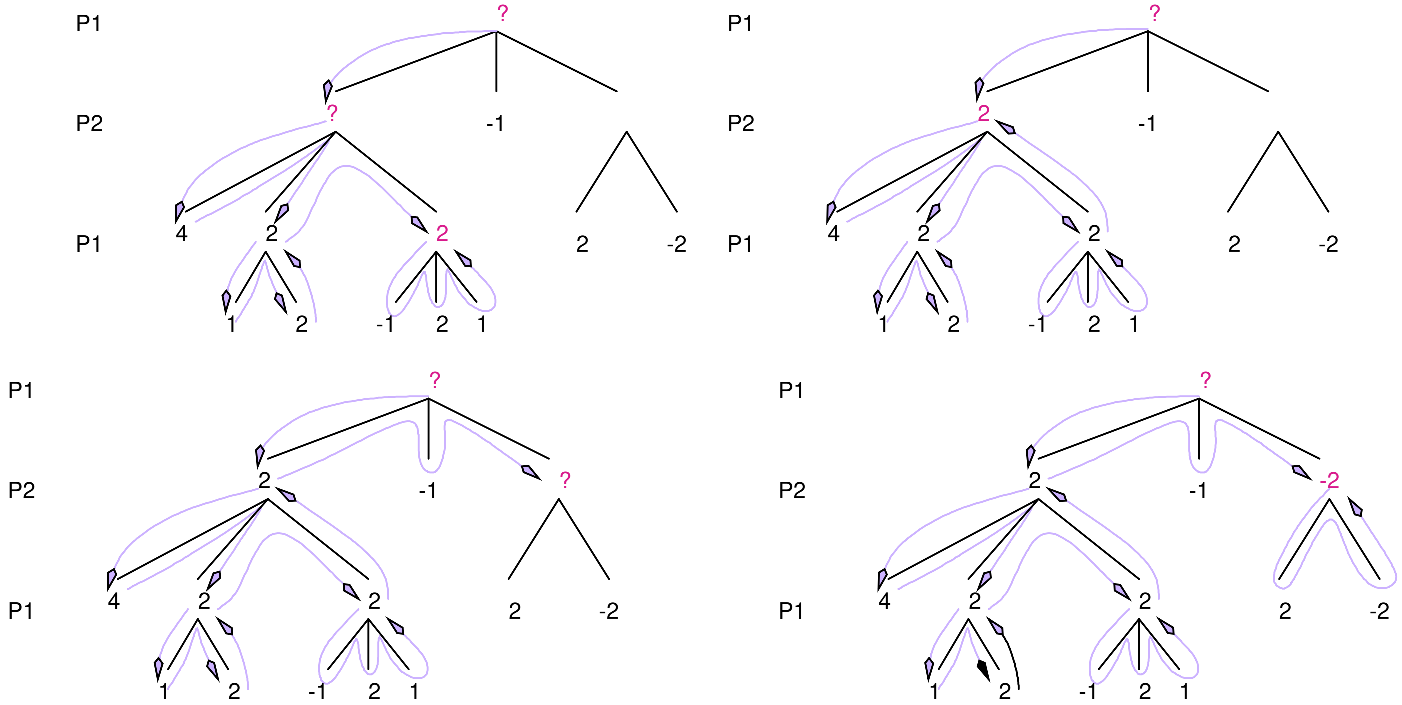
Question. If you had to write a program for finding the value of a game, would you do it in this way? Can you think of alternatives?

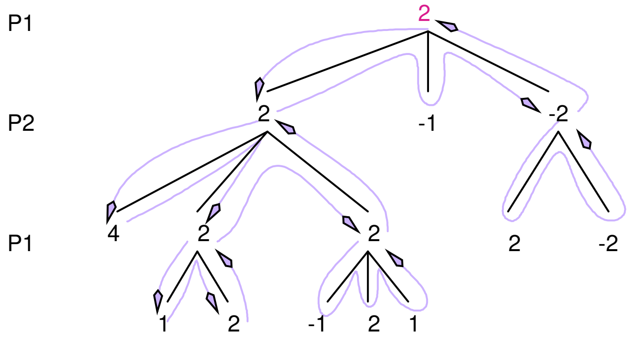
It is much better to traverse the tree depth-first. We will study this idea next.

Depth-first algorithm



Depth-first algorithm





This is known as the minimax-algorithm.

Minimax on 2-person zero-sum games

If we apply the minimax algorithm to a 2-person zero-sum game of complete information, then

- the value calculated for Player 1 is the value of the game in the sense of Section 2;
- the value calculated for Player 2 is the negative of that calculated for Player 1.

Minimax–what does it do?

Clearly, the minimax algorithm cannot help at all in dectecting mixed-strategy equilibrium points.

It will only calculate the actual value of a 2-person zero-sum game if a pure strategy equilibrium point exists; we can tell that this is the case when the values for the two players are the negative of each other.

If the algorithm remembers how to achieve the calculated value it calculates a strategy for guaranteeing it.

Minimax on non zero-sum games

What does the value for a player mean when the game is not zero-sum?

It is a minimal pay-off that each player can guarantee for himself. This makes the assumption that the other players do their worst from our player's point of view—clearly that is not realistic.

Note that the minimax algorithm cannot work on games of incomplete information.

Once again, we get a less than desirable solution to our problem, but it is still the best we can do.

The problem with the idea of trying to take into account what other players might really do is that when this is done by all the players then this results in a process which usually goes around in circles.

More improvements?

The minimax algorithm visits each leaf once. It visits other nodes slightly more often (depending on how many leaves they have).

One could write a minimax algorithm that visits every node at most twice if the tree data structure used knows about siblings. However, the improvement in effiency usually isn't worth worrying about.

Note that once the program has found the value of a node, it is fine if it forgets everything below that node.

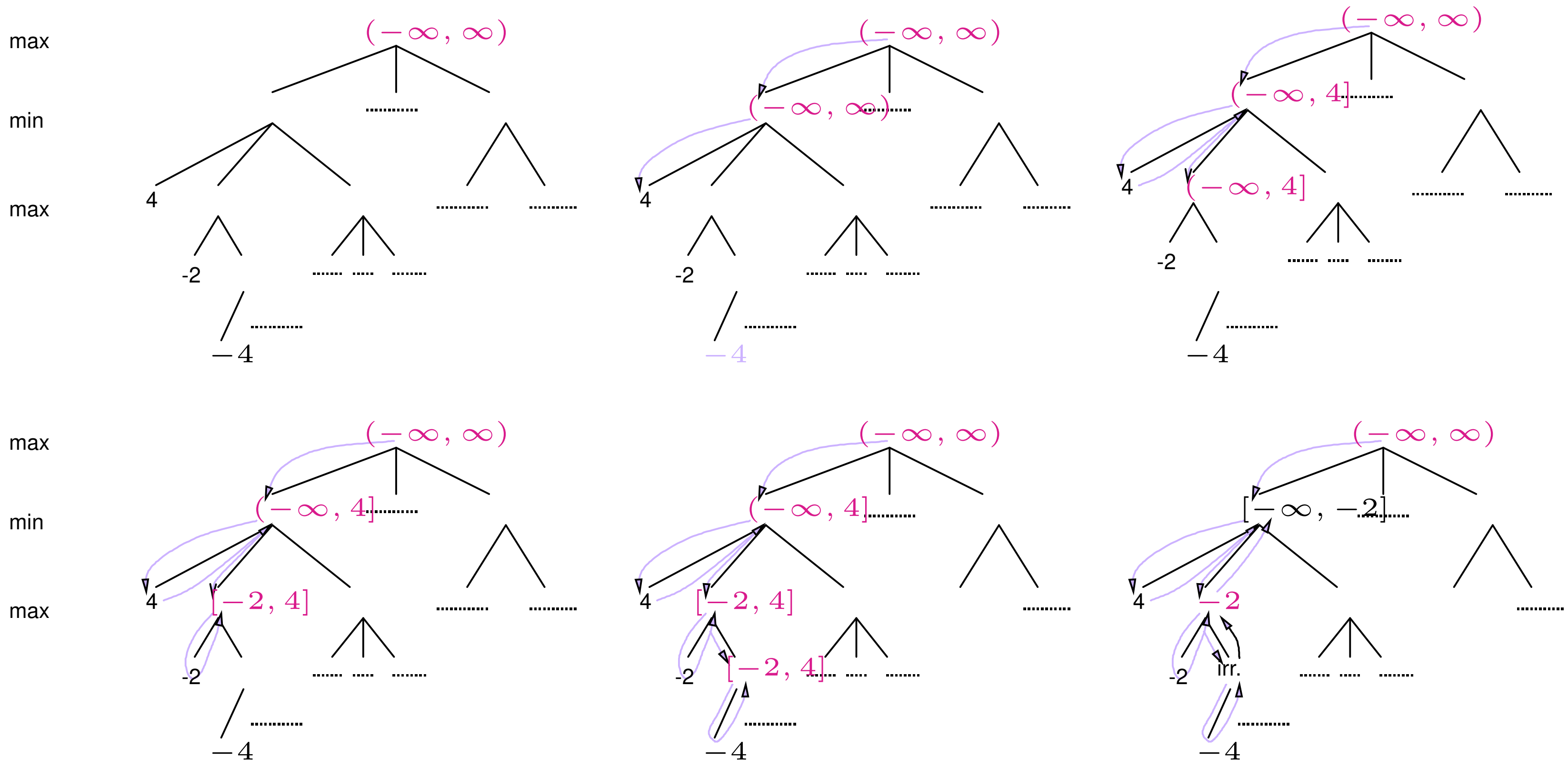
It seems that visiting every node just once (or maybe twice) is the best we can possibly do to determine the value of a node in this way. This turns out to be

wrong.

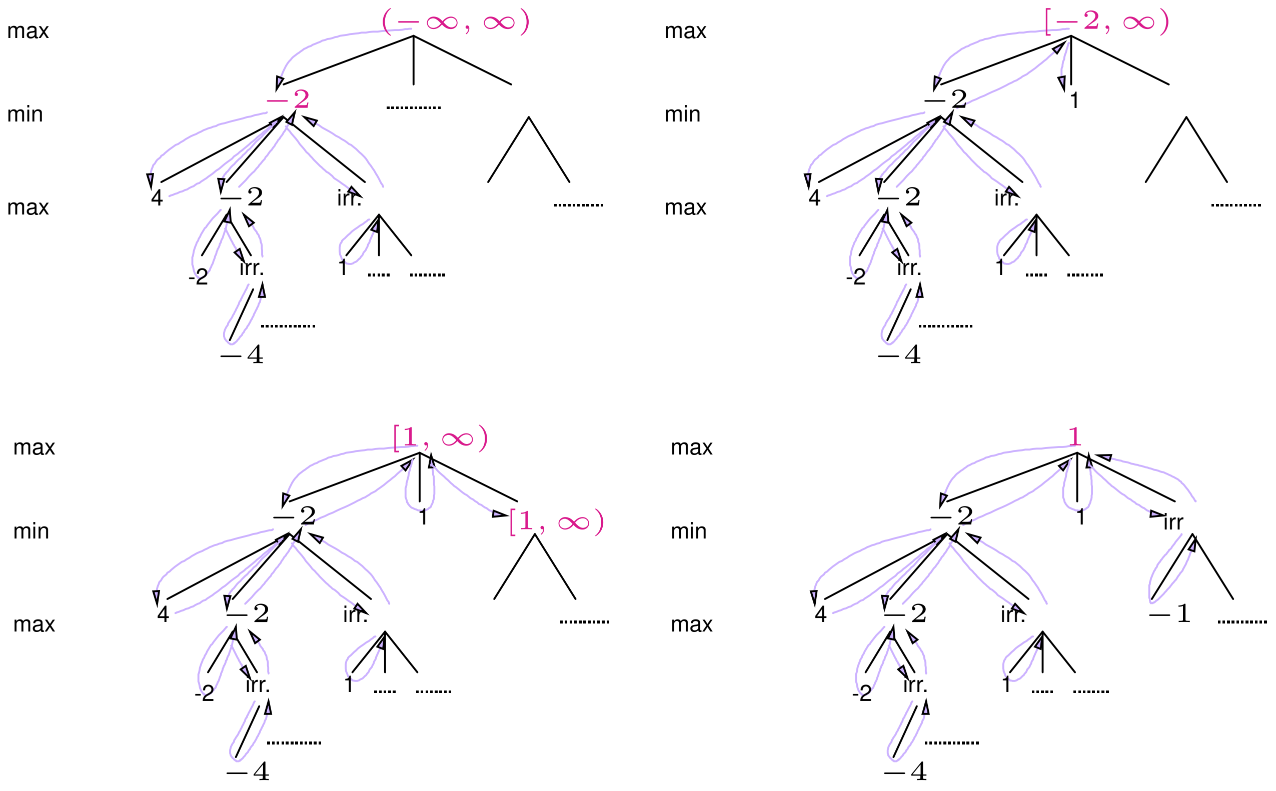
Alpha-beta pruning

Alpha-beta pruning

The following algorithm works by passing along information about what is known so far.



Alpha-beta pruning



There are parts of the tree we have not searched at all!

How does this work?

On each recursive call of the procedure that determines the value, two parameters (the 'alpha' and 'beta' in 'alpha-beta pruning', or 'alpha-beta search') are passed along to indicate the relevant range of values. At the start, these are set to $-\infty$ and ∞ respectively.

Node is of type	max	min
Increase lower bound α when value found is greater	$\checkmark$	
decrease upper bound β when value found is smaller		$\checkmark$
Stop search and return to parent when value found is	smaller than α	greater than β

Some thoughts

The better the first move searched in an alpha-beta search, the more of the tree can be pruned.

This is because the true value of a node will be determined by the best move for the Player whose turn it is.

This is going to cut down the range of possible values the most, and therefore will allow us to discard more values as irrelevant.

Deciding which available moves are likely to be good for a game like Chess, Go, Othello, or the like, is hard!

Summary of Section 3

- It is possible to determine the value for a game for each player using a recursive algorithm, the minimax algorithm. It also determines a strategy for reaching that value.
- This algorithm does not require the whole game tree to be in memory as it runs, it merely has to build it locally in order to visit all nodes in a depth-first search.
- For non-zero sum games, the outcome of this procedure is to be viewed with a grain of salt.
- This algorithm only works for games of complete information.
- We can improve on this algorithm to get one which does not have to visit all nodes in the tree; this is known as alpha-beta pruning or alpha-beta search. Again we can use this algorithm to determine the best move, namely one that leads to the value.