Game tree search

- Game tree
- Minimax strategy
- Alpha beta pruning
- Evaluation functions

Next: CSP

Readings: Chap 6.1, 6.2, 6.3

*Slides based on those of Sheila McIlraith

Generalizing search problems

- So far: our search problems have assumed agent has complete control of environment
  - state does not change unless the agent (robot) changes it
- Assumption not always reasonable
  - other agents whose interests conflict with yours
- In these cases, we need to generalize our view of search to handle state changes that are not in the control of the agent

What are key features of a game

- Players have their own interests
- Each player tries to alter the world so as to best benefit itself
- They are hard because: How you should play depends on how you think the other person will play; but how they play depends on how they think you will play

Game properties

- Two-player
- Discrete: Game states or decisions can be mapped on discrete values.
- Finite: There are only a finite number of states and possible decisions that can be made.

Game properties

- Zero-sum (零和): Fully competitive
  - if one player wins, the other loses an equal amount
  - note that some games don't have this property
- Deterministic: no chance involved
  - no dice, or random deals of cards, or coin flips, etc.
- Perfect information: all aspects of the state are fully observable
  - *e.g.*, no hidden cards.

An example: Rock, Paper, Scissors

- Scissors cut paper, paper covers rock, rock smashes scissors
- Represented as a matrix: Player I chooses a row, Player II chooses a column
- Payoff to each player in each cell (Pl.I / Pl.II)
- 1: win, 0: tie, -1: loss

  so it's zero-sum

Player I \ Player II	R	P	S
R	0/0	-1/1	1/-1
P	1/-1	0/0	-1/1
S	-1/1	1/-1	0/0

Extensive Form Two-Player Zero-Sum Games

- But R,P,S is a simple "one shot" (一次性) game
  - single move each
  - in game theory: a strategic or normal form game (策略或范式博弈)
- Many games extend over multiple moves
  - turn-taking: players act alternatively
  - *e.g.*, chess, checkers, etc.
  - in game theory: extensive form games (扩展形式博弈)
- We'll focus on the extensive form
  - that's where the computational questions emerge

Two-Player Zero-Sum Game – Definition

- Two players A (Max) and B (Min)
- Set of states S (a finite set of states of the game)
- An initial state $I \in S$ (where game begins)
- Terminal positions $T \subseteq S$ (Terminal states of the game: states where the game is over)
- Successors (or Succs - a function that takes a state as input and returns a set of possible next states to whomever is due to move)
- Utility (效益) or payoff (收益) function $V : T \to \mathbf{R}$. (a mapping from terminal states to real numbers that show how good is each terminal state for player A and bad for player B.)

Two-Player Zero-Sum Game – Intuition

Players alternate moves (starting with A, or Max)

- Game ends when some terminal $t \in T$ is reached

A game state: a state-player pair

- Tells us what state we're in and whose move it is

Utility function and terminals replace goals

- A, or Max, wants to maximize the terminal payoff
- B, or Min, wants to minimize the terminal payoff

Think of it as:

- A, or Max, gets V(t) and B, or Min, gets –V(t) for terminal node t
- This is why it's called zero (or constant) sum

Nim: informal description

1. We begin with a number of piles of matches.
2. In one's turn one may remove any number of matches from one pile.
3. The last person to remove a match loses.

In *II-Nim*, one begins with two piles, each with two matches…

S =			
	(_ , _)-A	(_ , i)-A	(_ , ii)-A
	(i , _)-A	(i , i)-A	(i , ii)-A
	(ii , _)-A	(ii , i)-A	(ii , ii)-A
	(_ , _)-B	(_ , i)-B	(_ , ii)-B
	(i , _)-B	(i , i)-B	(i , ii)-B
	(ii , _)-B	(ii , i)-B	(ii , ii)-B

Nim: informal descripti

1. We begin with a numbe
2. In one's tu
3.

A common trick: By symmetry, some of the states are trivially equivalent (e.g. (_,ii)-A and (ii,_)-A). Make them one state by some canonical description (e.g. left pile never larger than right).

two matches, each with two piles…

S =

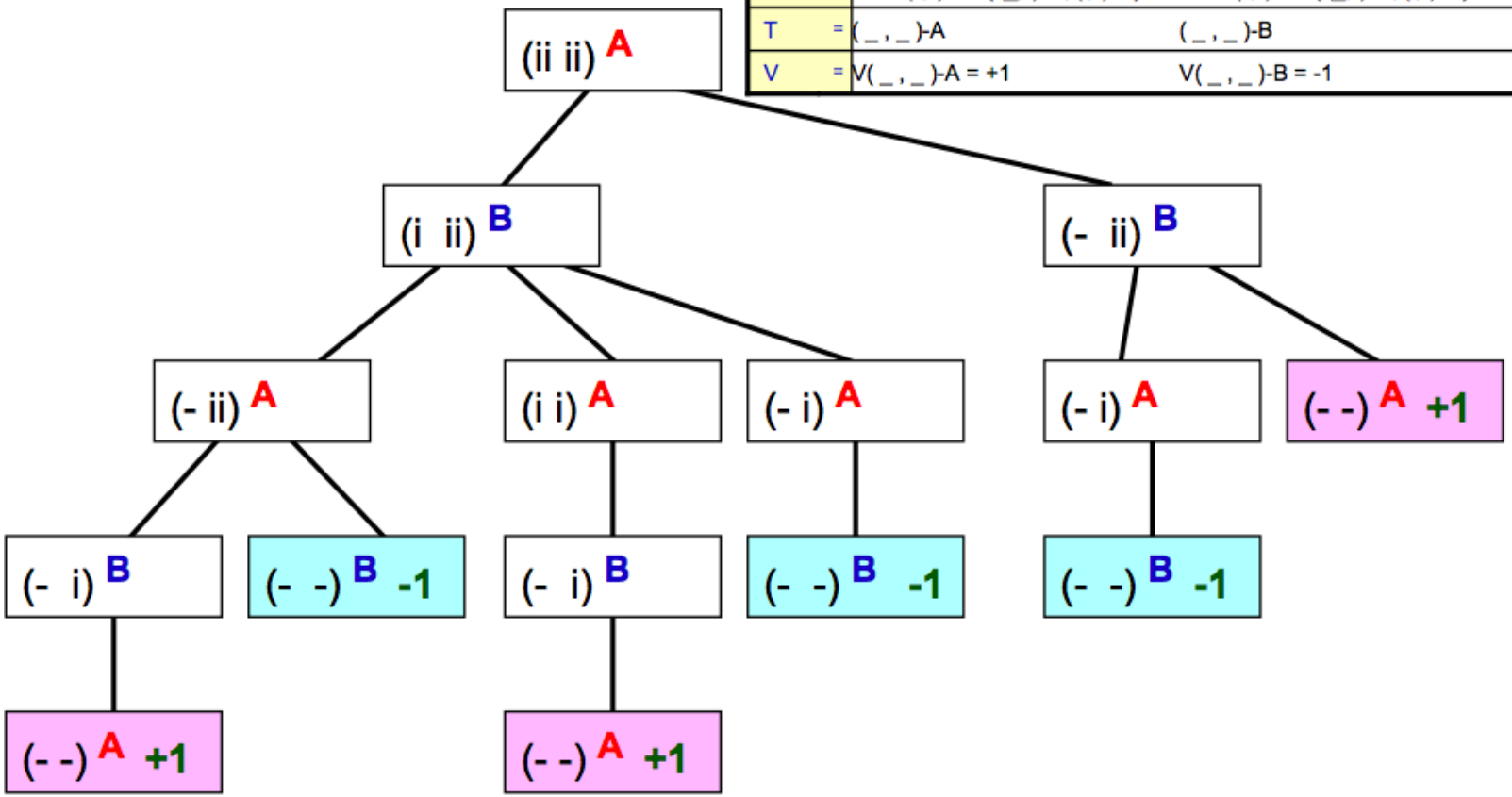
(_ , _)-A	(_ , i)-A	(_ , ii)-A
	(i , i)-A	(i , ii)-A
		(ii , ii)-A
(_ , _)-B	(_ , i)-B	(_ , ii)-B
	(i , i)-B	(i , ii)-B
		(ii , ii)-B

II-Nim

S =	a finite set of states (note: state includes information sufficient to deduce who is due to move)	(_ , _)-A (_ , i)-A (_ , ii)-A (i , i)-A (i , ii)-A (ii , ii)-A (_ , _)-B (_ , i)-B (_ , ii)-B (i , i)-B (i , ii)-B (ii , ii)-B	
I =	the initial state	(ii , ii)-A	
Succs =	a function which takes a state as input and returns a set of possible next states available to whoever is due to move	Succs(_,i)-A = { (_,_)-B } Succs(_,ii)-A = { (_,_)-B , (_,i)-B } Succs(i,i)-A = { (_,i)-B } Succs(i,ii)-A = { (_,i)-B (_,ii)-B (i,i)-B} Succs(ii,ii)-A = { (_,ii)-B , (i,ii)-B }	Succs(_,i)-B = { (_,_)-A } Succs(_,ii)-B = { (_,_)-A , (_,i)-A } Succs(i,i)-B = { (_,i)-A } Succs(i,ii)-B = { (_,i)-A , (_,ii)-A (i,i)-A } Succs(ii,ii)-B = { (_,ii)-A , (i,ii)-A }
T =	a subset of S. It is the terminal states	(_ , _)-A	(_ , _)-B
V =	Maps from terminal states to real numbers. It is the amount that A wins from B.	V(_ , _)-A = +1	V(_ , _)-B = -1

II-Nim Game Tree

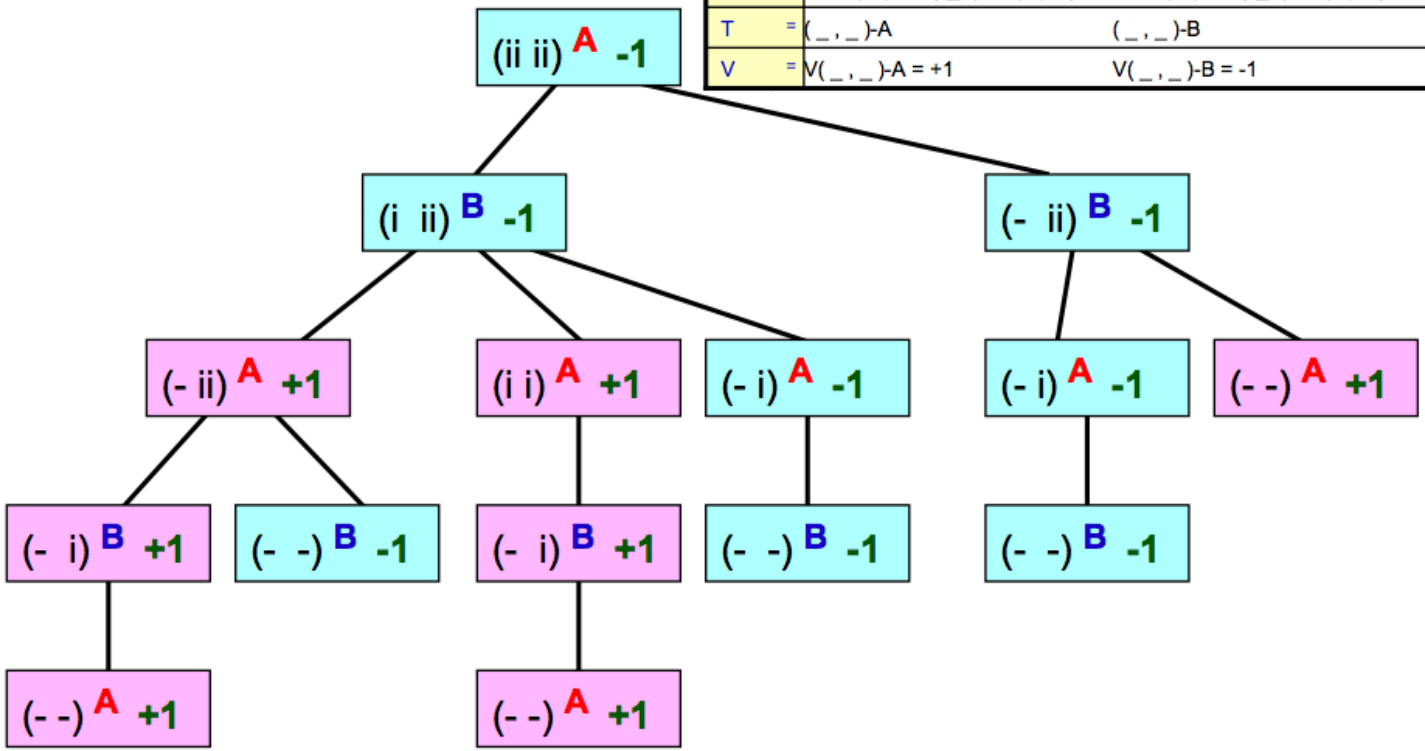
S	=	(_ , _)-A (_ , i)-A (_ , ii)-A (i , i)-A (i , ii)-A (ii , ii)-A (_ , _)-B (_ , i)-B (_ , ii)-B (i , i)-B (i , ii)-B (ii , ii)-B	
I	=	(ii , ii)-A	
Succs	=	Succs(_,i)-A = { (_,_)-B }	Succs(_,i)-B = { (_,_)-A }
		Succs(_,ii)-A = { (_,_)-B , (_,i)-B }	Succs(_,ii)-B = { (_,_)-A , (_,i)-A }
		Succs(i,i)-A = { (_,i)-B }	Succs(i,i)-B = { (_,i)-A }
		Succs(i,ii)-A = { (_,i)-B (_,ii)-B (i,i)-B}	Succs(i,ii)-B = { (_,i)-A , (_,ii)-A (i,i)-A }
		Succs(ii,ii)-A = { (_,ii)-B , (i,ii)-B }	Succs(ii,ii)-B = { (_,ii)-A , (i,ii)-A }
T	=	(_ , _)-A	(_ , _)-B
V	=	V(_ , _)-A = +1	V(_ , _)-B = -1



Back values up the tree

II-Nim Game Tree

S	=	(_ , _)-A (_ , i)-A (_ , ii)-A (i , i)-A (i , ii)-A (ii , ii)-A (_ , _)-B (_ , i)-B (_ , ii)-B (i , i)-B (i , ii)-B (ii , ii)-B	
I	=	(ii , ii)-A	
Succs	=	Succs(_,i)-A = { (_,_)-B }	Succs(_,i)-B = { (_,_)-A }
		Succs(_,ii)-A = { (_,_)-B , (_,i)-B }	Succs(_,ii)-B = { (_,_)-A , (_,i)-A }
		Succs(i,i)-A = { (_,i)-B }	Succs(i,i)-B = { (_,i)-A }
		Succs(i,ii)-A = { (_,i)-B (_,ii)-B (i,i)-B}	Succs(i,ii)-B = { (_,i)-A , (_,ii)-A (i,i)-A }
		Succs(ii,ii)-A = { (_,ii)-B , (i,ii)-B }	Succs(ii,ii)-B = { (_,ii)-A , (i,ii)-A }
T	=	(_ , _)-A	(_ , _)-B
V	=	V(_ , _)-A = +1	V(_ , _)-B = -1



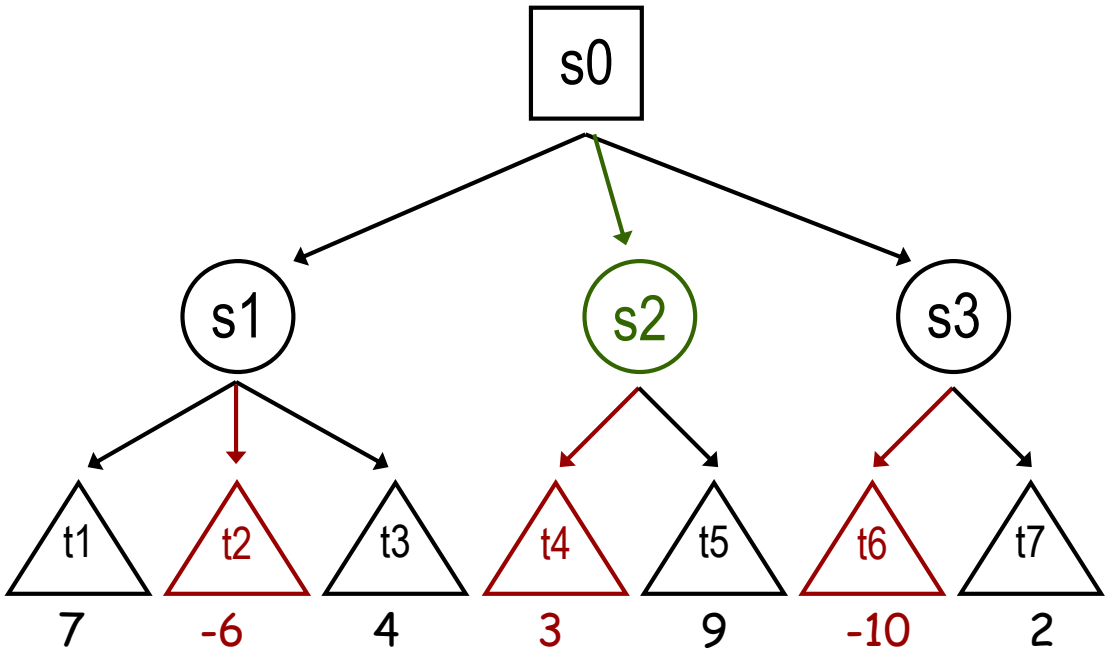
The MiniMax Strategy

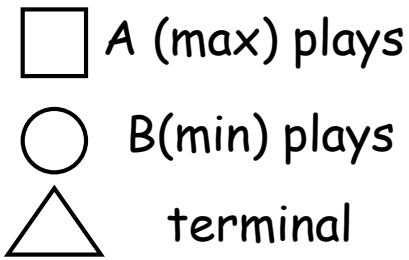
Assume that the other player will always play their best move

- you always play a move that will minimize the payoff that could be gained by the other player.
- By minimizing the other player's payoff, you maximize your own.

Note that if you know that Min will play poorly in some circumstances, there might be a better strategy than MiniMax (i.e., a strategy that gives you a better payoff).

MiniMax Strategy payoffs



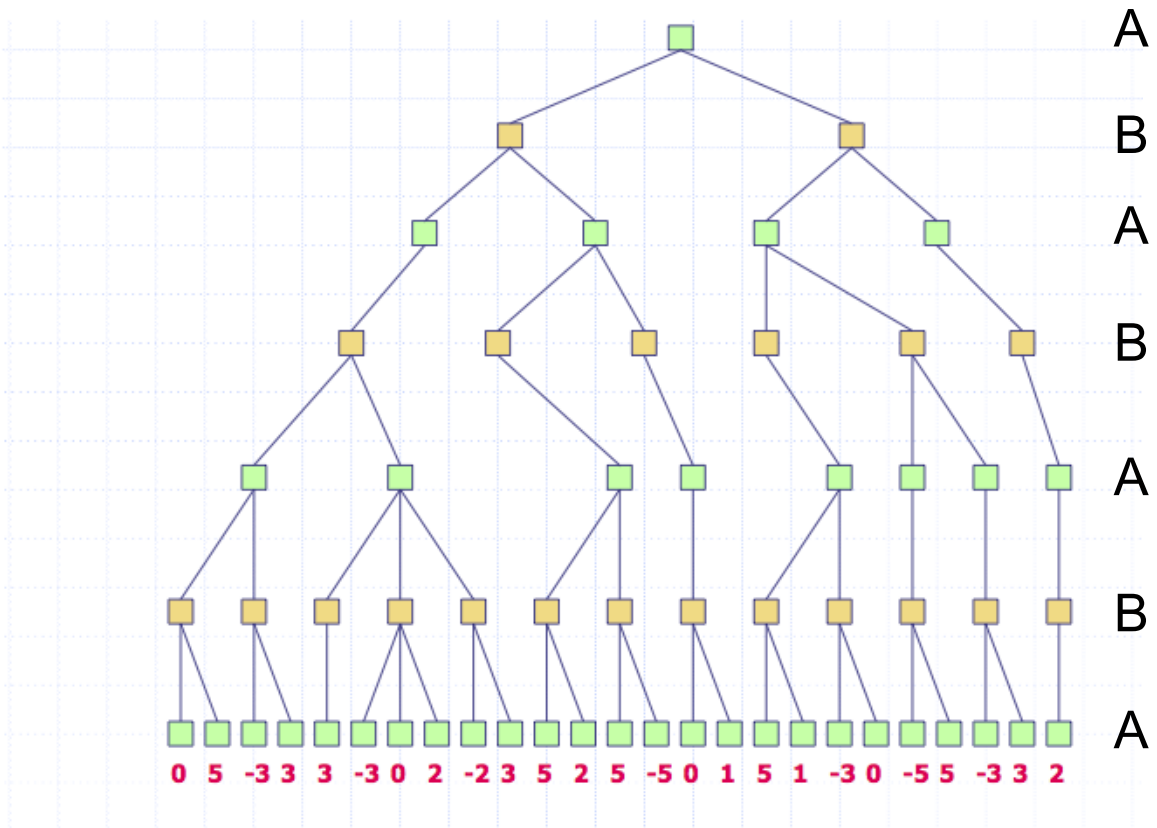


The terminal nodes have a utility value (V).
We can compute a "utility" for the non-terminal states by assuming both players always play their best move.

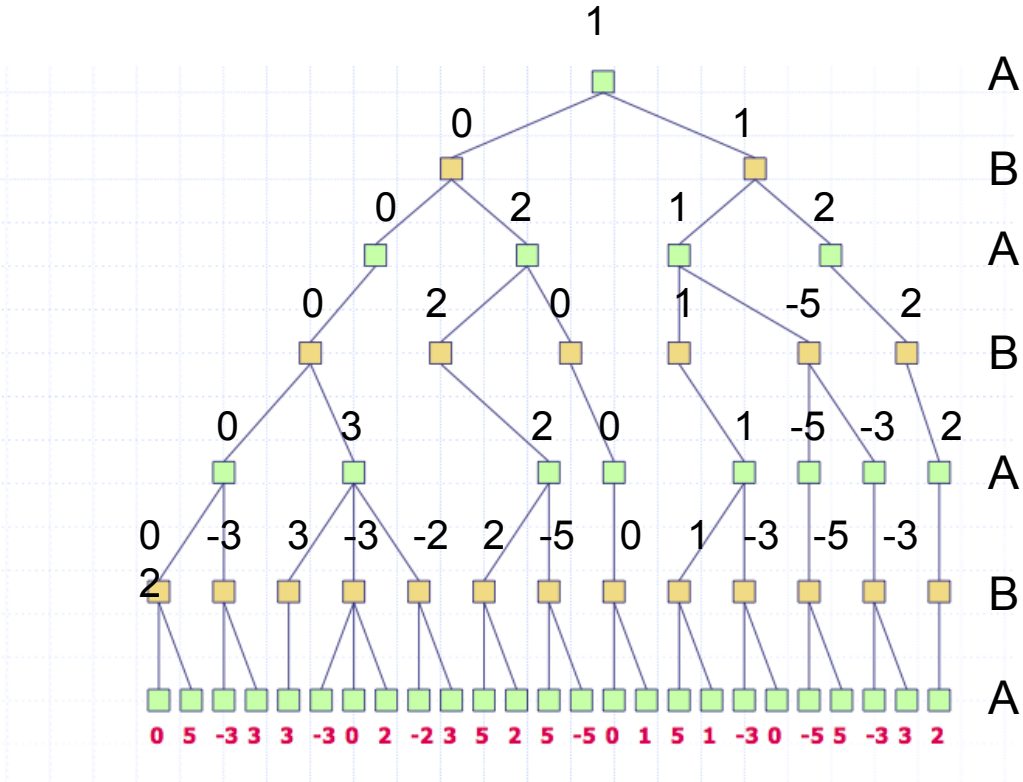
MiniMax Strategy

- Build full game tree (all leaves are terminals)
  - Root is start state, edges are possible moves, etc.
  - Label terminal nodes with utilities

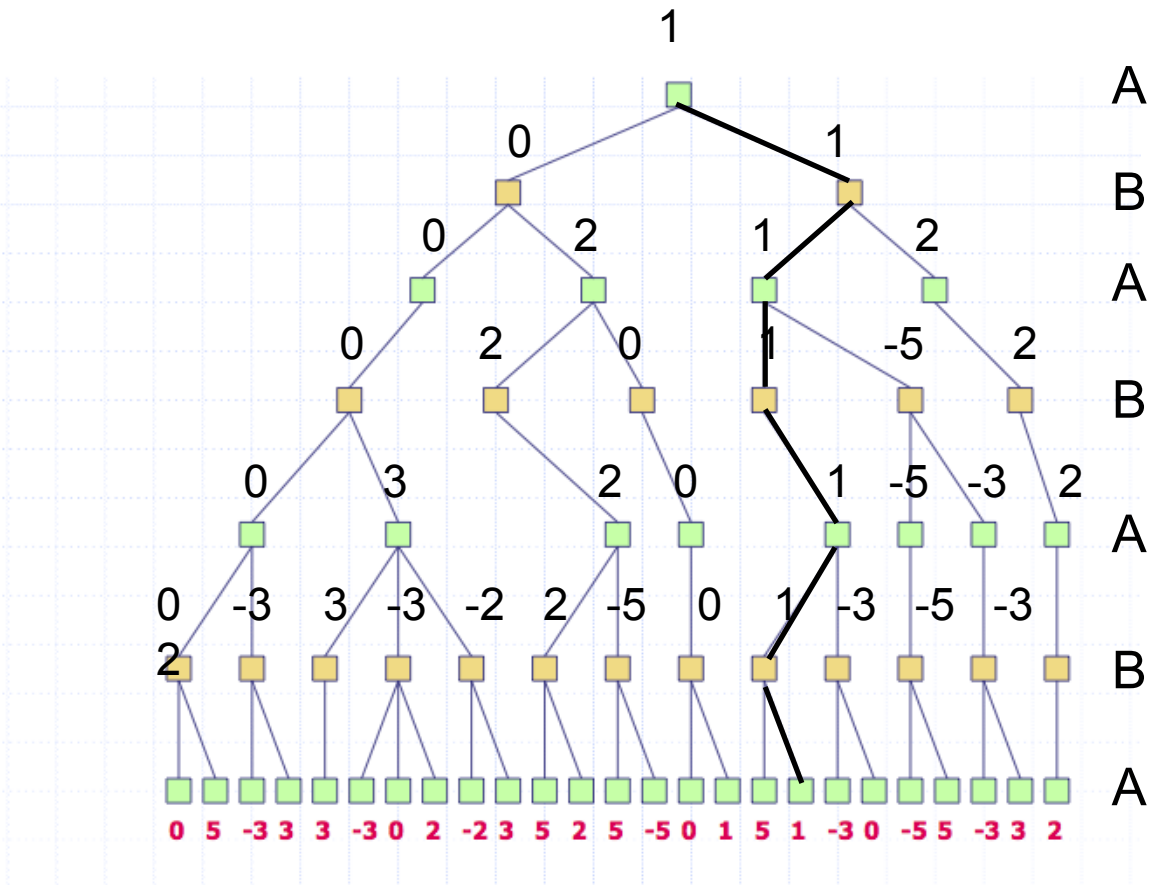
- Back values *up* the tree
  - $U(t)$ is defined for all terminals (part of input)
  - $U(n) = \min \{U(c) : c \textit{ is a child of } n\}$ if n is a Min node
  - $U(n) = \max \{U(c) : c \text{ is a child of } n\}$ if n is a Max node



Let's practice by computing all the game theoretic values for nodes in this tree.



Question: if both players play rationally, what path will be followed through this tree?


1
A
0
1
B
0
2
1
2
A
0
2
0
1
-5
2
B
0
3
2
0
1
-5
-3
2
A
0
-3
3
-3
-2
2
-5
0
1
-3
-5
-3
2
B
A
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

Depth-First Implementation of MiniMax

- Building the entire game tree and backing up values gives each player their strategy.
- However, the game tree is exponential in size.
- Furthermore, as we will see later it is not necessary to know all of the tree.
- To solve these problems we find a **depth-first** implementation of minimax.
- We run the depth-first search after each move to compute what is the next move for the MAX player. (We could do the same for the MIN player).
- This avoids explicitly representing the exponentially sized game tree: we just compute each move as it is needed.

Depth-First Implementation of MiniMax

```
DFMiniMax(n, Player) //return Utility of state n given that
                     //Player is MIN or MAX

If n is TERMINAL
Return V(n) //Return terminal states utility
            //(V is specified as part of game)

//Apply Player's moves to get successor states.
ChildList = n.Successors(Player)
If Player == MIN
    return minimum of DFMiniMax(c, MAX) over c ∈ ChildList
Else //Player is MAX
    return maximum of DFMiniMax(c, MIN) over c ∈ ChildList
```

- Note: the game tree has to have finite depth for this to work
- Advantage of DF implementation: space efficient

Pruning

It is not necessary to examine entire tree to make correct MiniMax decision

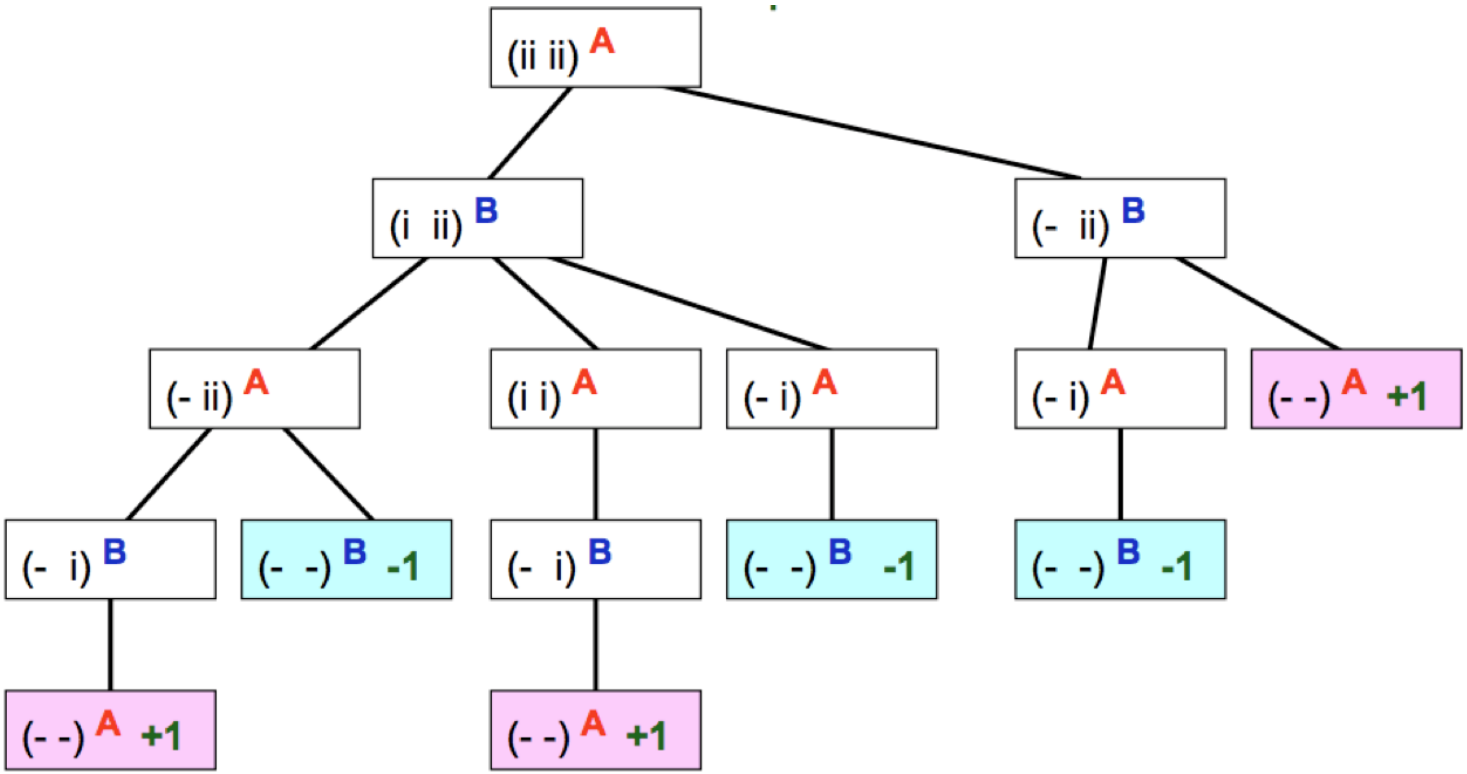
Assume depth-first generation of tree

- After generating value for only *some* of *n*'s children we can prove that we'll never reach *n* in a MiniMax strategy.
- So we needn't generate or evaluate any further children of *n*!

Two types of pruning (cuts):

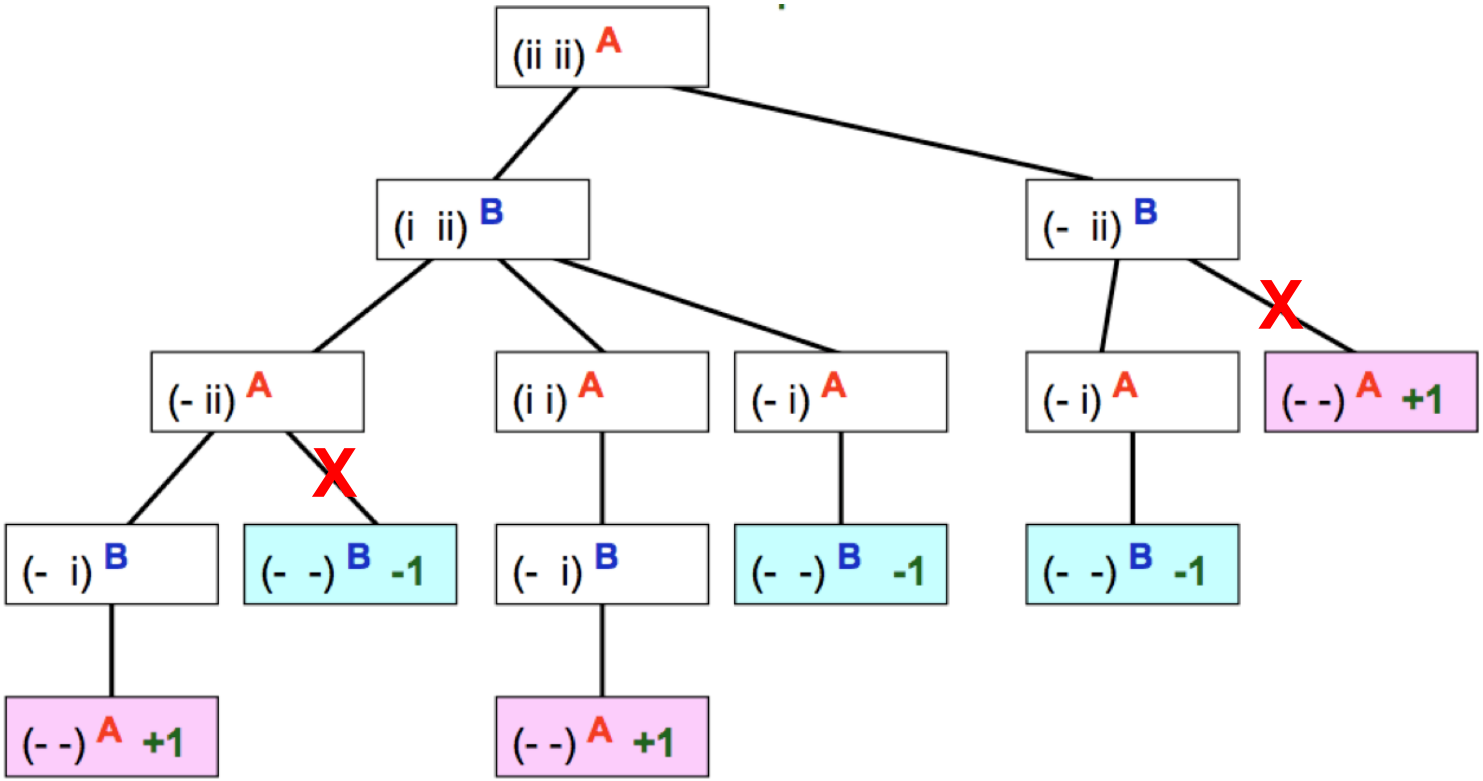
- pruning of max nodes (α-cuts)
- pruning of min nodes (β-cuts)

Pruning



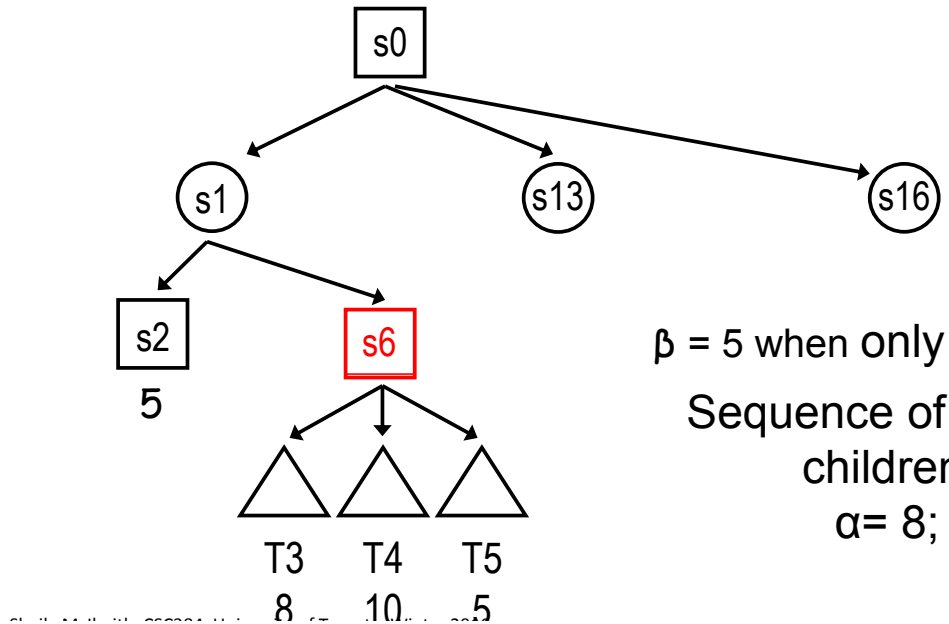
Assume the only values of terminals are -1 and 1 and we're running a DFS implementation of MiniMax. Where can we prune our tree?

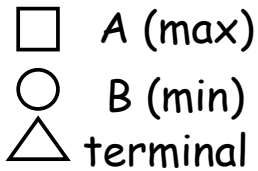
Pruning



Cutting Max Nodes (Alpha Cuts)

- At a Max node *n*:
  - Let β be the lowest value of *n*'s siblings examined so far (siblings to the left of *n* that have already been searched)
  - Let α be the highest value of *n*'s children examined so far (changes as children examined)



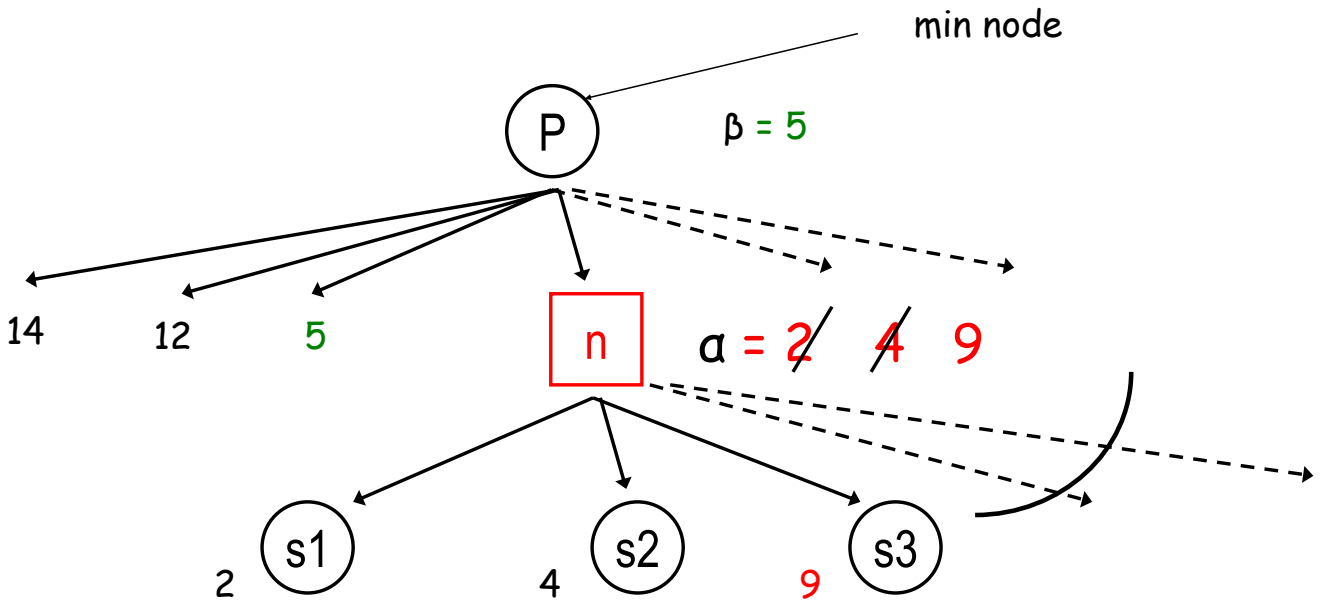


β = 5 when only one sibling value is known

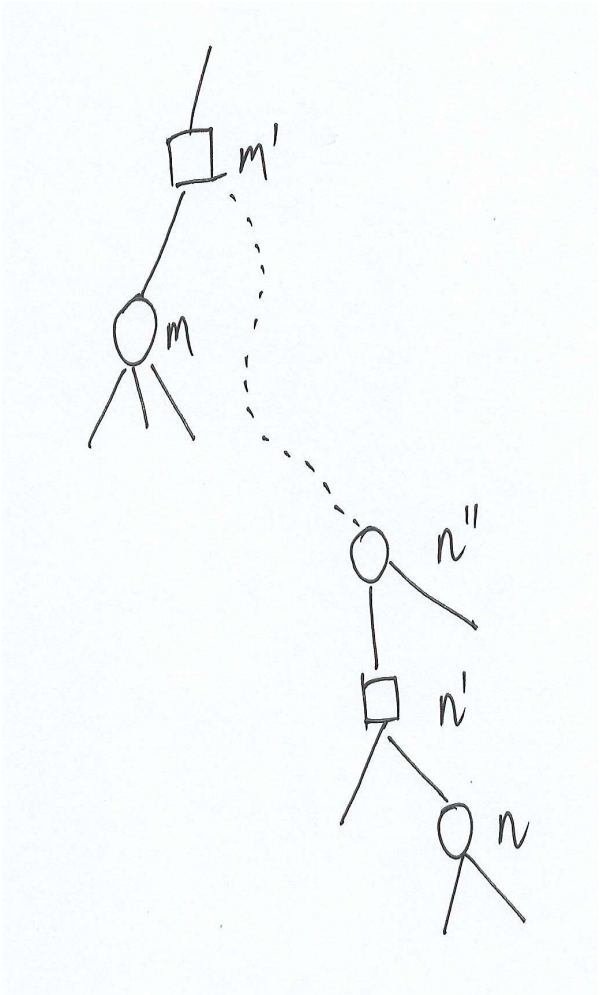
Sequence of values for α as s6's children are explored:
α= 8; α= 10; α= 10

Cutting Max Nodes (Alpha Cuts)

- While at a Max node *n*, if α becomes ≥ β we can stop expanding the children of *n*
  - Min will never choose to move from *n*'s parent to *n* since it would choose one of *n's* lower valued siblings first.



Alpha-beta cuts in general

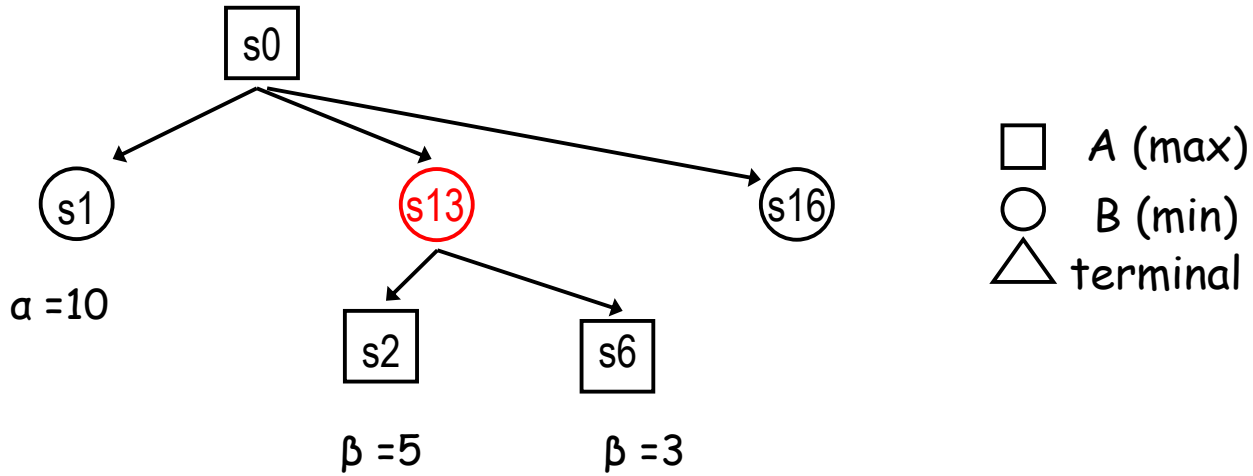


If $U(m) \geq U(n)$, then n can be pruned.

- We prove by induction on the distance d between m' and n'.
- Base case: $m' = n'$. Obviously, n can be pruned.
- We now prove the inductive step.
- Case 1: $U(n') > U(n)$. Then n can be pruned.
- Case 2: $U(n') = U(n)$. Then $U(m) \geq U(n) = U(n') \geq U(n'')$. By induction, n'' can be pruned, and hence n can be pruned.

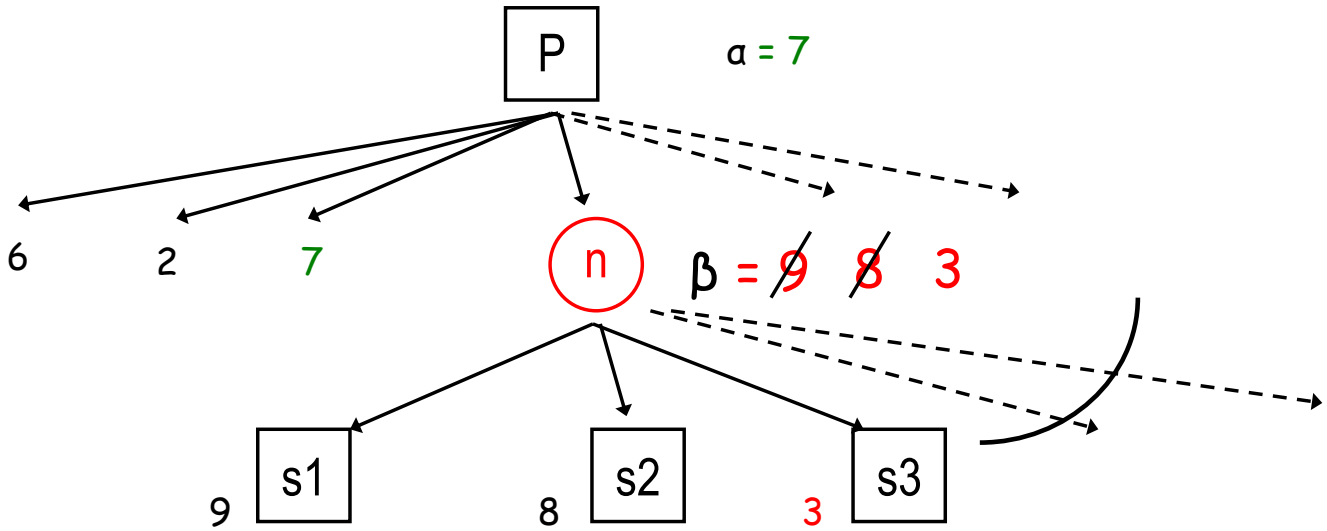
Cutting Min Nodes (Beta Cuts)

- At a Min node *n*:
  - Let α be the highest value of *n*'s sibling's examined so far (fixed when evaluating *n*)
  - Let β be the lowest value of *n*'s children examined so far (changes as children examined)



Cutting Min Nodes (Beta Cuts)

- If β becomes ≤ α we can stop expanding the children of *n.*
  - Max will never choose to move from *n*'s parent to *n* since it would choose one of *n's* higher value siblings first.



In general, at a Min node n, if β becomes $\leq \alpha$ value of an ancestor Max node, then we can stop expanding n

Implementing Alpha-Beta Pruning

```
AlphaBeta(n,Player,alpha,beta) //return Utility of state
If n is TERMINAL
    return V(n) //Return terminal states utility
ChildList = n.Successors(Player)
If Player == MAX
 for c in ChildList
    alpha = max(alpha, AlphaBeta(c,MIN,alpha,beta))
    If beta <= alpha
        break
 return alpha
Else //Player == MIN
  for c in ChildList
    beta = min(beta, AlphaBeta(c,MAX,alpha,beta))
    If beta <= alpha
        break
 return beta
```

When AlphaBeta(n,Player,alpha,beta) is called, alpha is the maximum alpha value of n's ancestor Max nodes, and beta is the minimum beta value of n's ancestor Min nodes

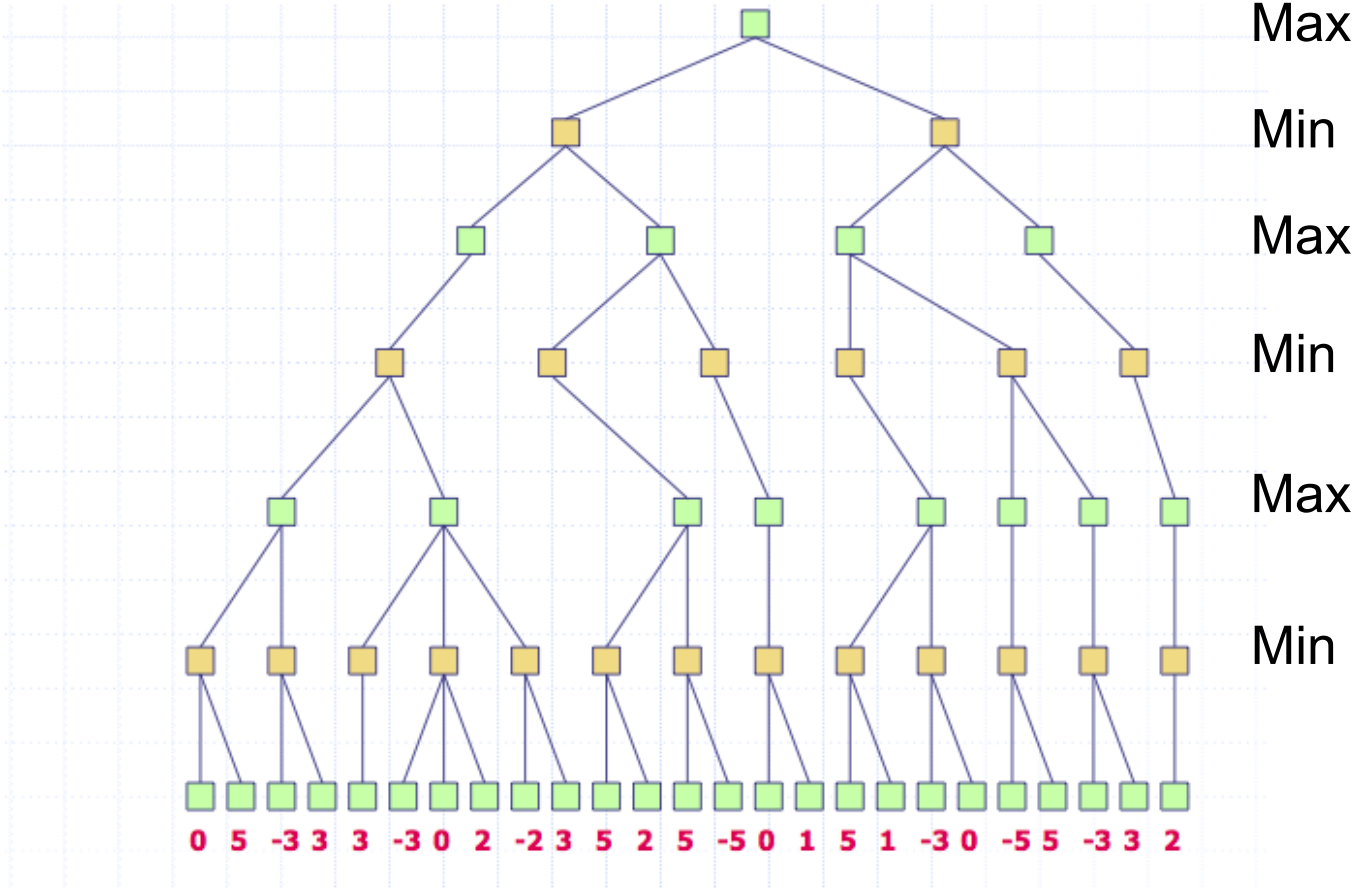
Initial call: AlphaBeta(START-NODE,Player,-infinity,+infinity)

Tracing alpha beta pruning

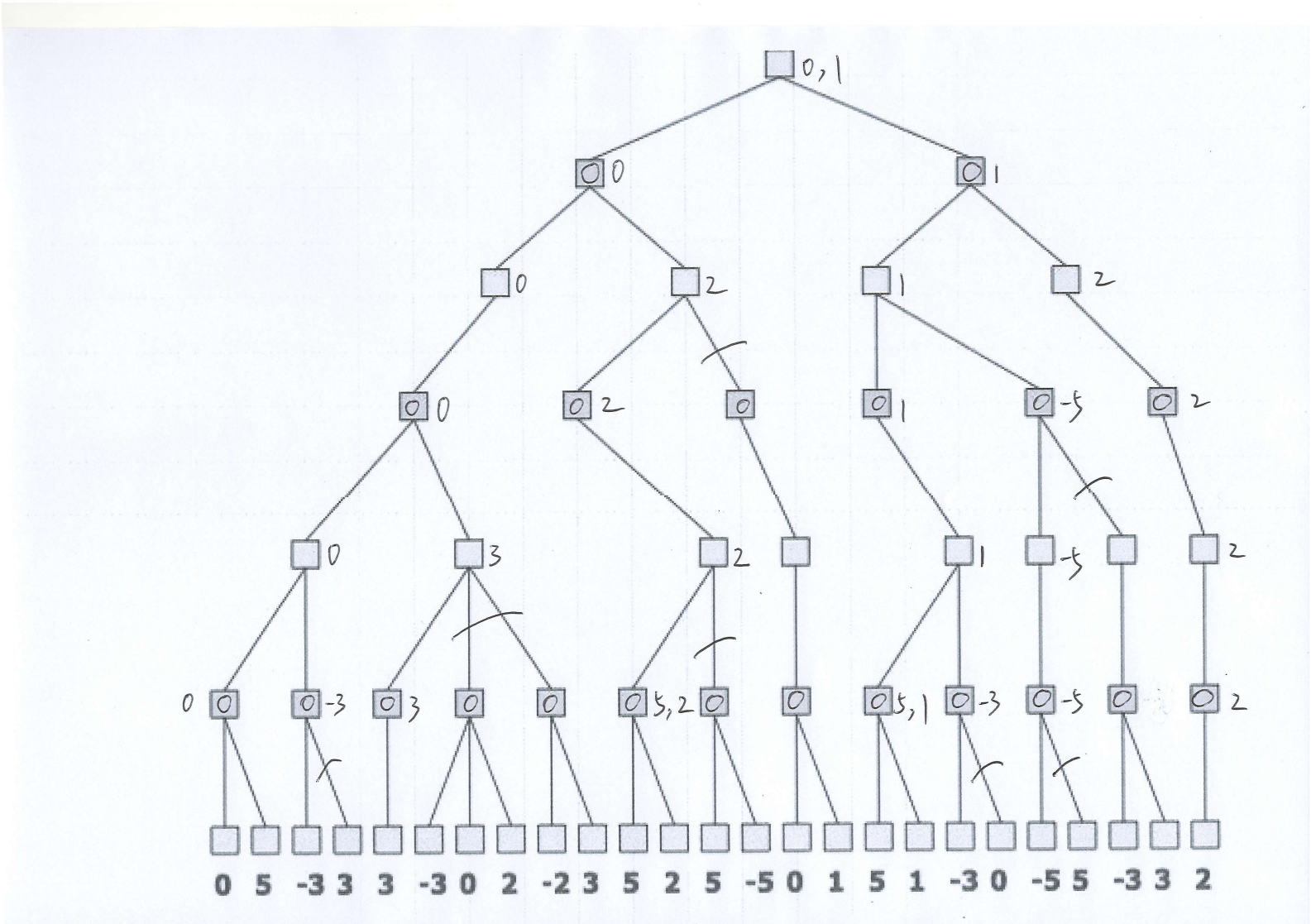
- Mark Max nodes with the change of alpha values, and Min nodes with the change of beta values
- Do alpha cut on a Max node whenever the current value $\geq$ the value of an ancestor Min node
- Do beta cut on a Min node whenever the current value $\leq$ the value of an ancestor Max node

Example

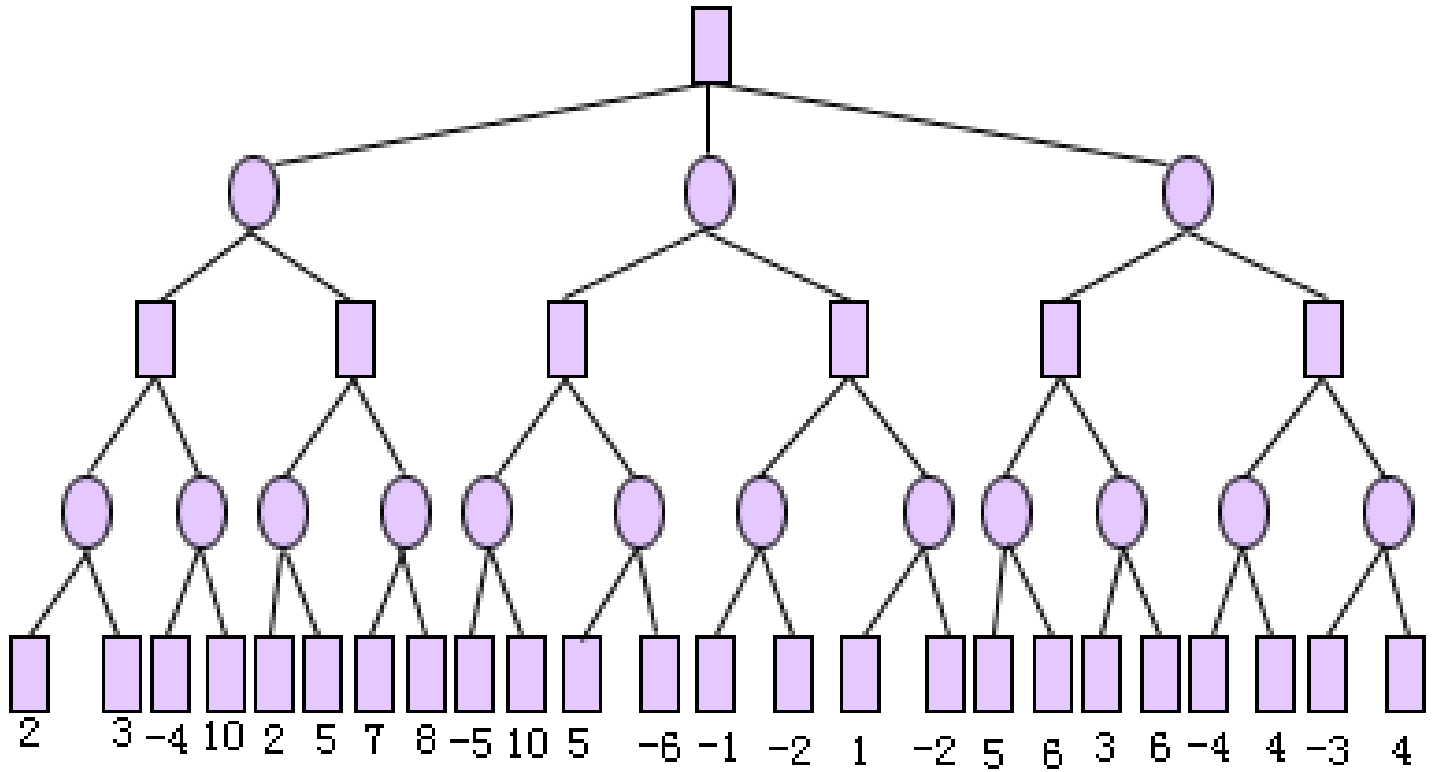
Which computations can we avoid here? Assuming we expand nodes left to right?



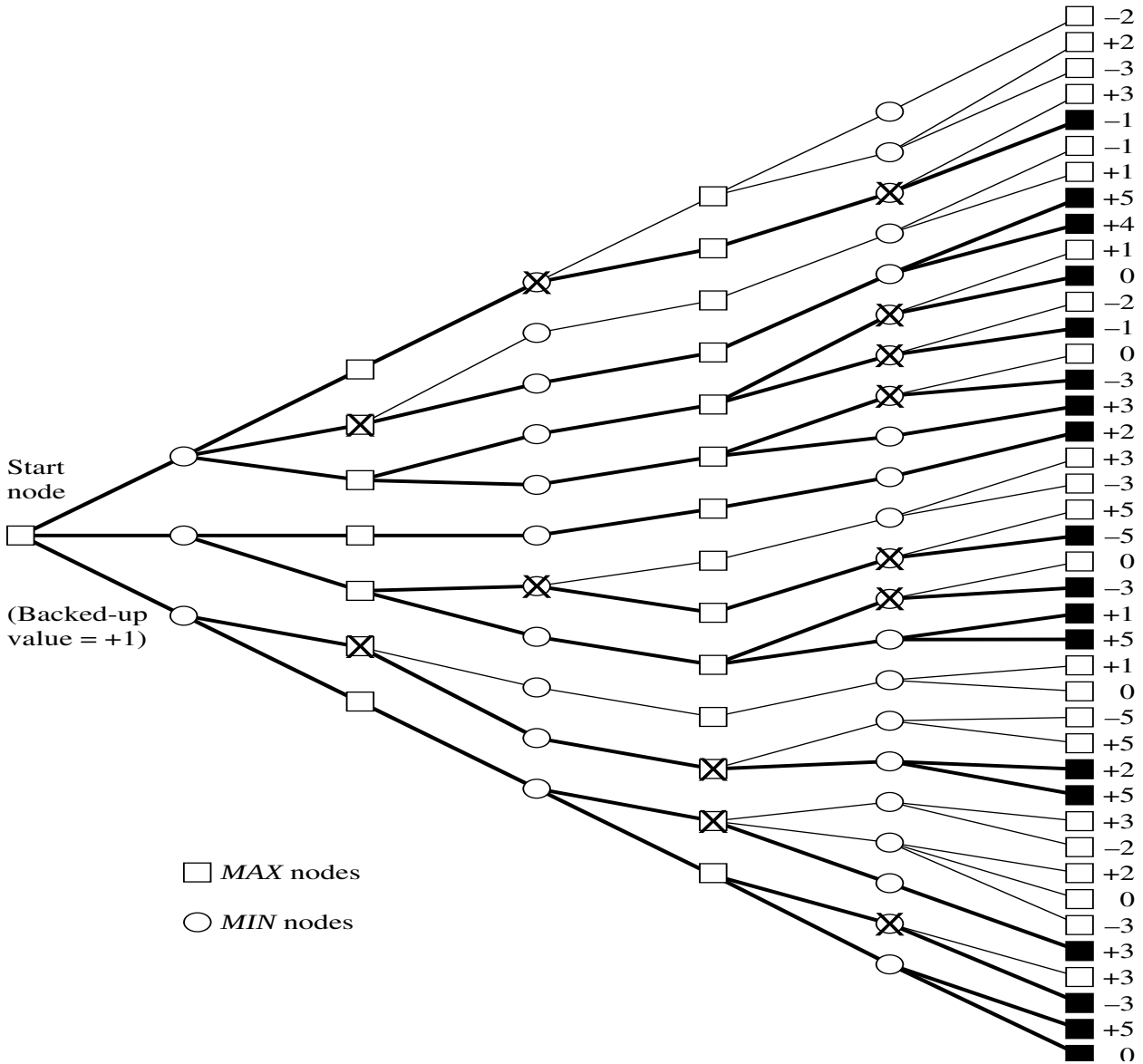
Example


0, 1
0
1
0
2
1
2
0
2
0
1
-5
2
0
3
2
1
-5
2
0
-3
3
5, 2
5, 1
-3
-5
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

Another example



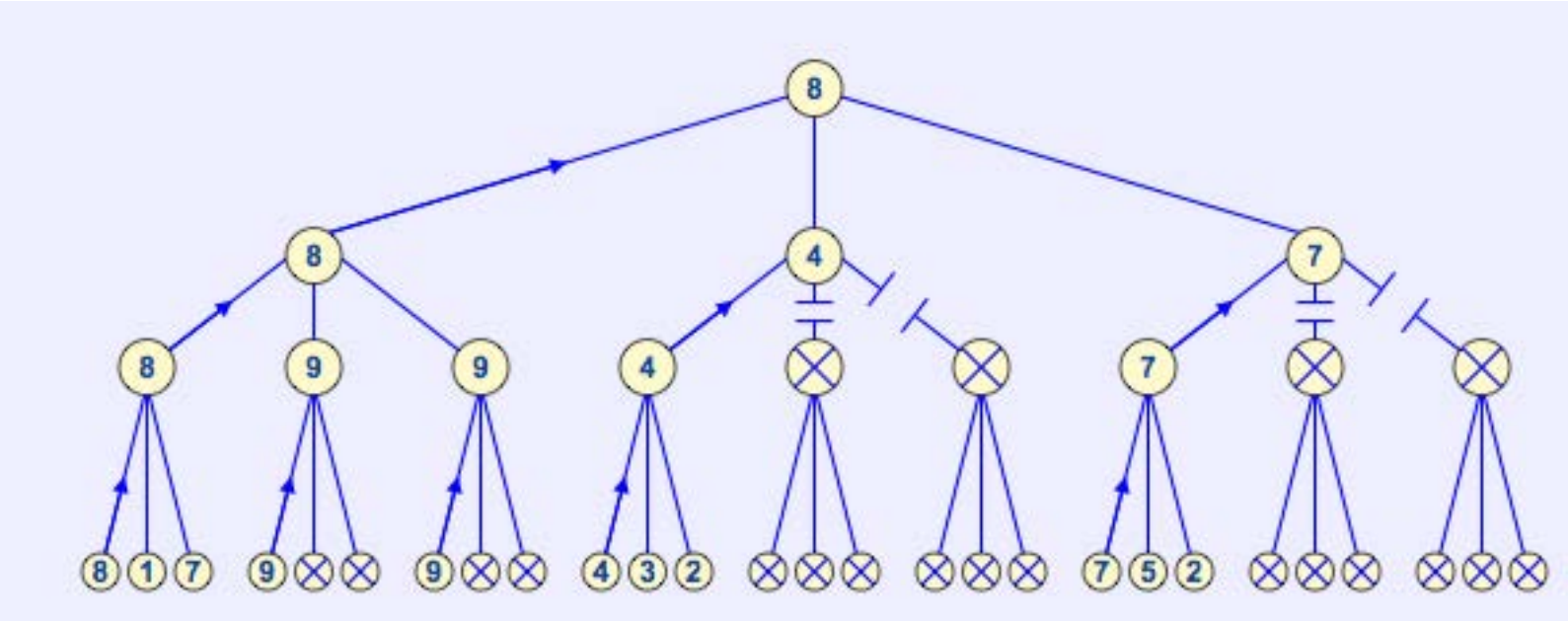
One more example



Effectiveness of alpha beta pruning

- With no pruning, you have to explore $O(b^D)$ nodes, the same as plain MiniMax.
- However, if the move ordering for the search is optimal (meaning the best moves are searched first), the number of nodes we need to search using alpha beta pruning is $O(b^{D/2})$.
- This means we can, in theory, search twice as deep!
- In Deep Blue, alpha beta pruning meant the average branching factor at each node was about 6 instead of 35.

An example of the best case scenario



The effective branching factor of the first layer is B. The effective branching of the second is 1. The effective layer of the third is B. And so on ....

Practical Matters

"Real" games are too large to enumerate tree

- *e.g.*, chess branching factor is roughly 35
- Depth 10 tree: 2,700,000,000,000,000 nodes
- Even alpha-beta pruning won't help here!

We must limit depth of search tree

- Can't expand all the way to terminal nodes
- We must make heuristic estimates about the values of the (nonterminal) states at the leaves of the tree
- These heuristics are often called evaluation function

Evaluation functions: basic requirements

- Should order the terminal states in the same way as the true utility function.
- The computation must not take too long!
- For nonterminal states, the evaluation function should be strongly correlated with the actual chances of winning.

How to design evaluation functions?

- Features of the states, *e.g.*, in chess, the number of white pawns(卒), black pawns, white queens, etc.
- The features, taken together, define various categories or equivalence classes of states: the states in each category have the same values for all the features.
- Any given category will contain some states that lead to wins, some that lead to draws, and some that lead to losses.
- *e.g.*, suppose our experience suggests that 72% of the states in a category lead to a win, 20% to a loss, and 8% to a draw.
- Then a reasonable evaluation for states in the category is the expected utility value: $0.72 \cdot 1 + 0.20 \cdot (-1) + 0.08 \cdot 0 = 0.52$.
- However, there are too many categories

How to design evaluation functions?

- Most evaluation functions compute separate numerical contributions from each feature and then combine them
- *e.g.*, each pawn is worth 1, a knight(马) or bishop(象) is worth 3, a rook(车) 5, and the queen 9
- Mathematically, a weighted linear function $Eval(s) = w_1 \cdot f_1(s) + \ldots + w_n \cdot f_n(s) = \sum_{i=1}^{n} w_i \cdot f_i(s)$
- Deep Blue used over 8000 features
- This involves a strong assumption: the contribution of each feature is independent of the values of the other features.
- The assumption may not hold, hence nonlinear combinations are also used

How to design evaluation functions?

- The features and weights are not part of the rules of chess!
- They come from centuries of human chess-playing experience.
- In case this kind of experience is not available, the weights of the evaluation function can be estimated by machine learning techniques.

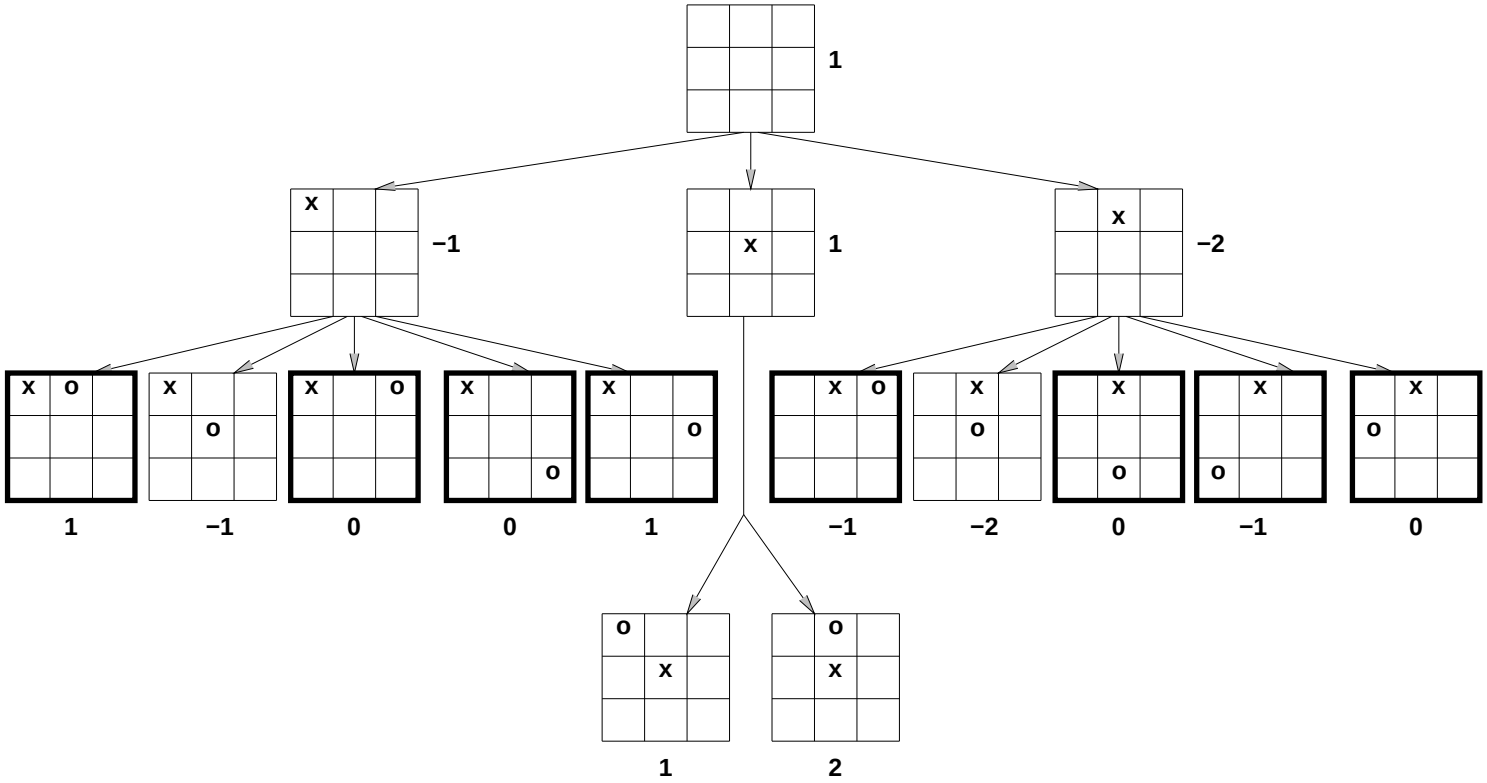
Alpha beta pruning: tic-tac-toe

- We define X_n as the number of rows, columns, or diagonals with exactly n X's and no O's.
- Similarly, O_n is the number of rows, columns, or diagonals with just n O's.
- The utility function assigns $+1$ to any position with $X_3 = 1$ and -1 to any position with $O_3 = 1$.
- All other terminal positions have utility 0.
- For nonterminal positions, we use a linear evaluation function defined as $Eval(s) = 3X_2(s) + X_1(s) - (3O_2(s) + O_1(s))$.

Alpha beta pruning: tic-tac-toe

- Show the whole game tree starting from an empty board down to depth 2, taking symmetry into account.
- Mark on your tree the evaluations of all positions at depth 2.
- Using the minimax algorithm, mark on your tree the backed-up values for the positions at depths 1 and 0
- Circle the nodes at depth 2 that would not be evaluated if alpha - beta pruning were applied, assuming the nodes are generated in the optimal order for alpha - beta pruning.

Alpha beta pruning: tic-tac-toe

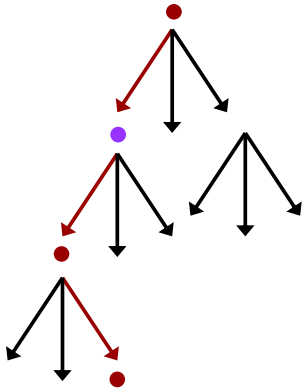


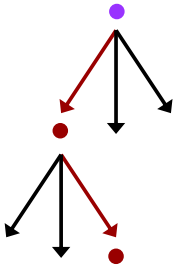
An Aside on Large Search Problems

- Issue: inability to expand tree to terminal nodes is relevant even in standard search
  - Often we can't expect A* to reach a goal by expanding full frontier
  - So we often limit our look-ahead, and make moves before we actually know the true path to the goal
  - Sometimes called *online* or *real-time* search

- In this case, we use the heuristic function not just to guide our search, but also to commit to moves we actually make
  - In general, guarantees of optimality are lost, but we reduce computational/memory expense dramatically

Real-time Search





1. We run A* (or our favorite search algorithm) until we are forced to make a move or run out of memory. Note: no leaves are goals yet.

2. We use evaluation function f(n) to decide which path *looks* best (let's say it is the red one).

3. We take the first step along the best path (red), by actually *making that move*.

4. We restart search at the node we reach by making that move. (We may actually cache the results of the relevant part of first search tree if it's hanging around, as it would with A*).