COMP3308/COMP3608, Lecture 4

ARTIFICIAL INTELLIGENCE

Game Playing

Reference: Russell and Norvig, ch. 5

Outline

- Games
  - Characteristics
  - Games as search
- Games of perfect information
  - Minimax
  - Alpha-beta pruning
- Games of imperfect information
- Games of chance

Why Study Games?

- **Why are games attractive to humans? They:**
  - **are fun!**
  - **offer competition – the goal is to win**
  - **have simple rules**
  - **challenge our ability to think – test our "intelligence"**

- **Why are games appealing target for AI research?**

1. **Games are *too hard to solve***
   - **Very large search space for common games**

     Chess: b=35, m=100 (50 moves by each player) => 35^{100} nodes in the search tree (legal positions "only" 10^{40})

     tic-tac-toe: b=5, m=9 => 5^9=125 nodes
   - **Require decisions in limited time**

     No time to calculate the exact consequences of each move; need to make best guess based on previous experience, heuristics

Games are Attractive for AI (cont.)

2. **More *like the real world* than the toy search problems**
    - **Optimal decision is infeasible, yet some decision is required**
    - **There is an "unpredictable" opponent to be considered**

3. ***Easy to represent as search problems***
    - **A state is a board configuration**
    - **Moves: typically a small number and well defined**
    - **Clear criterion for success**

Characteristics of Games

- **Deterministic vs chance**
  - **Deterministic: no chance element (no die rolls, no coin flips)**
- **Perfect vs imperfect information**
  - **Perfect: no hidden information; fully observable game, each player can see the complete game**
  - **Imperfect: some information is hidden, e.g. in card games the player's cards are hidden from the other players**
- **Zero-sum vs non zero-sum**
  - **Zero-sum (adversarial): one player's gain is the other player's loss**

Types of Games

	deterministic	chance
perfect information		
imperfect information		

- Chess=? Checkers=? Backgammon? Bridge=? Poker=? Monopoly-=? Connect-4=? Scrabble?

Types of Games

	deterministic	chance
perfect information	chess, checkers, connect-4	backgammon, monopoly
imperfect information		bridge, poker, scrabble

Specific Setting

- **2 players - MAX and MIN**
- **Players take turns**
- **No chance involved (e.g. no dice)**
- **Perfect information (fully observable) game**
- **Zero-sum game – a win for MAX is a loss for MIN and vice versa, i.e. MAX wants MIN to lose and vice versa**
- **Examples: chess, checkers, tic-tac-toe, connect-4, Go, Othello**

A Possible Way to Play

- Consider all legal moves you can make from the current position
- Compute all new positions resulting from these moves
- Evaluate each new position and determine the best move
- Make that move
- Wait for your opponent to move and then repeat

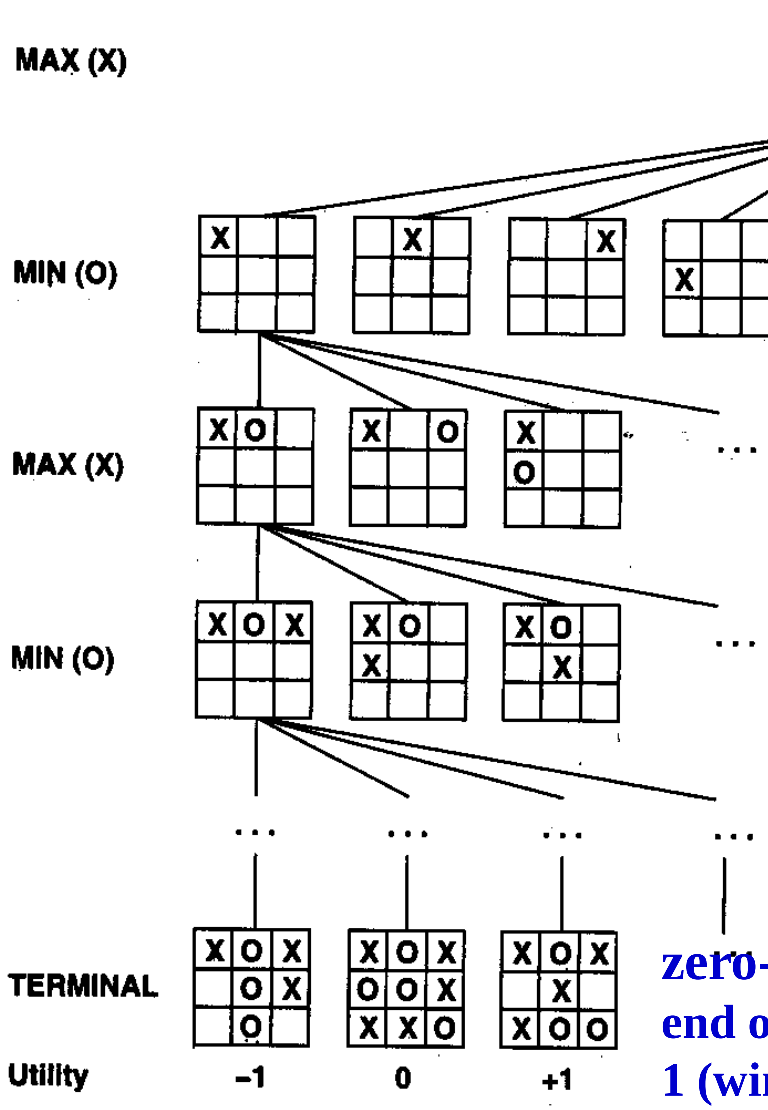
- Key steps:
  - Representing a position
  - Evaluating a position
  - Generating all legal moves from a given position

- This can be represented as a search problem

Game Playing as Search

- **Computers play games by searching a game tree**
- ***State*** **- board configuration**
- ***Initial state*** **– initial board configuration + who goes first**
- ***Terminal states*** **– board configurations where the game is over**
  - ***Terminal test*** **– when is the game over?**
- ***Operators*** **– legal moves a player can make**
- ***Utility function*** **(=payoff function) – numeric value associated with <u>terminal states</u> representing the <u>game outcome, e.g.</u>**
  - **e.g. a positive value - win for MAX, negative - loss for MAX, 0 – draw (e.g. 1, -1 and 0 can be used)**
  - **the higher the value = the better the outcome for MAX**
- ***Evaluation function*** **– numeric value associated with <u>non-terminal states</u>; shows how good the state is, e.g. how close it is to a win**
- ***Game tree*** **– represents all possible game scenarios**

Partial Game Tree for Tic-Tac-Toe



zero-sum game: the utility values at the end of the game are equal and opposite, e.g. 1 (win) and -1 (loss) from the first player's perspective

Evaluation Function

- **A numeric value *e(s)* associated with each non-terminal state *s***
- **Similar to the heuristic functions we studied before (e.g. an estimate of the distance to the goal)**
- **It gives the expected outcome of the game from the current position, for a given player, e.g. MAX:**

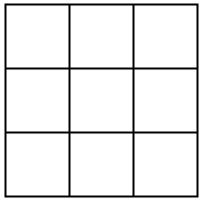
- **state *s* → number *e(s)***
  - ***e(s)* is a *heuristics* that estimates how favorable *s* is for MAX**
  - ***e(s)* > 0 means that *s* is favorable for MAX (the larger the better)**
  - ***e(s)* < 0 means that *s* is favorable for MIN**
  - ***e(s)* = 0 means that *s* is neutral**
- **Or more generally: the higher the value *e(s)*, the more favorable the position *s* is for MAX**

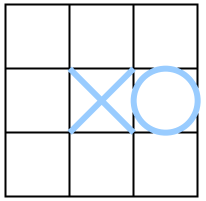
http://ai.stanford.edu/~latombe/cs121/2011/slides/L-adversarial.ppt

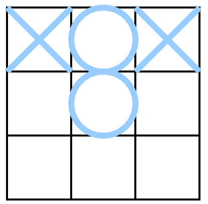
Evaluation Function – Example for Tic-Tac-Toe

MAX: cross, MIN: circle

***e(s)* = number of rows, columns, and diagonals open for MAX - number of rows, columns, and diagonals open for MIN**


8–8 = 0


6–4 = 2


3–3 = 0

Game Playing as Normal Search

- **Consider the following:**
  - **Player 1 (MAX) is the first player**
  - **He/she searches for a sequence of moves leading to a terminal state that is a winner (according to the utility function)**
- **Let's apply a normal search strategy, e.g. *greedy search***
  - **Expand each path to a terminal state, i.e. build the game tree**
  - **Choose the move with the best heuristic value for each player**

Game Playing Using Greedy Search

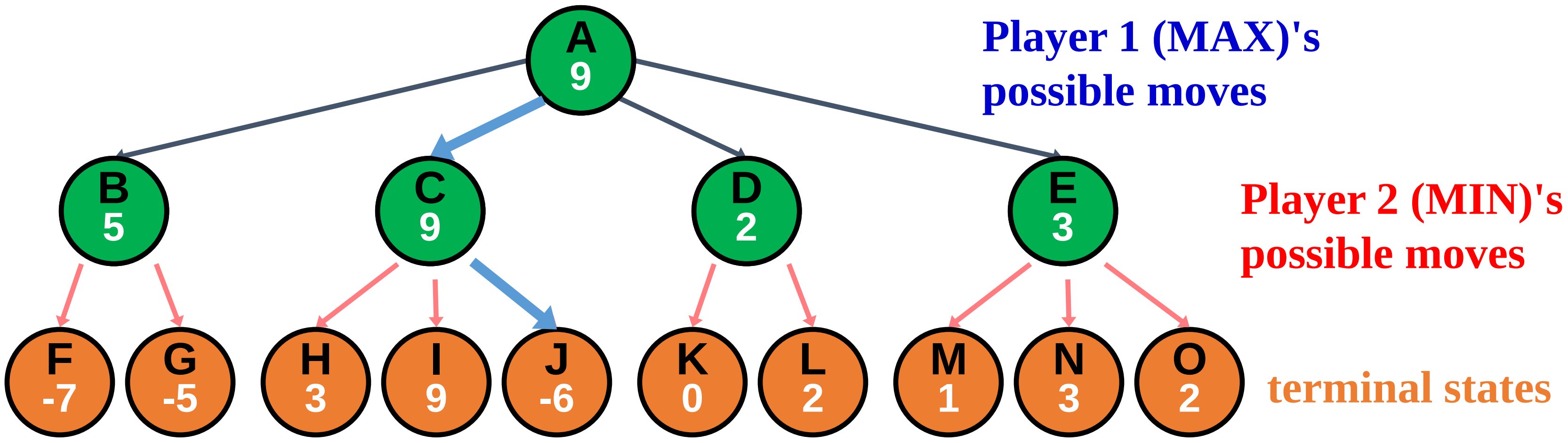
Example from http://pages.cs.wisc.edu/~skrentny/cs540/slides/7_gamePlaying.ppt

1) Player 1 (MAX) chooses C (the max value)

2) Player 2 (MIN) chooses J (the min value)

- **J is terminal node, the game ends**
- **J's value is negative => player 2 wins and player 1 loses**

- **What is the problem with greedy search? Does it <u>guarantee</u> anything (e.g. a win) for player 1 or player 2?**



Minimax Algorithm - Idea

- **Given: a 2-player, deterministic, perfect information game**
- **Minimax gives the *perfect (best, optimal) move* for a player, assuming that its opponent plays optimally**
  - **i.e. assuming the worst case scenario - optimal play by the opponent**

- **A player *plays optimally* if he/she always selects the best move based on the evaluation function**
  - **MAX – always select the node with the max evaluation value**
  - **MIN – always selects the node with the min evaluation value**

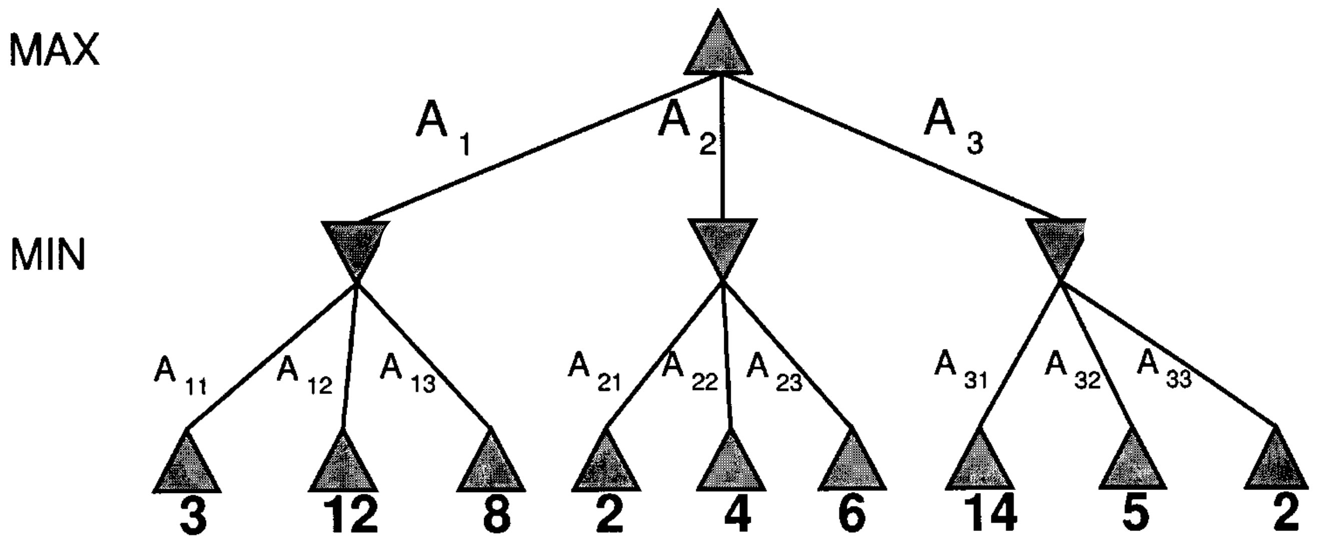
Minimax Algorithm – Idea (2)

- Minimax computes a value for each non-terminal node, called a *minimax value*
- It then chooses the move to the position with the best *minimax value* for the current player (the highest values for MAX or the lowest value for MIN)
- This position has the best achievable outcome for the current player, given that the other player selects the best move, i.e. plays optimally

Minimax Algorithm

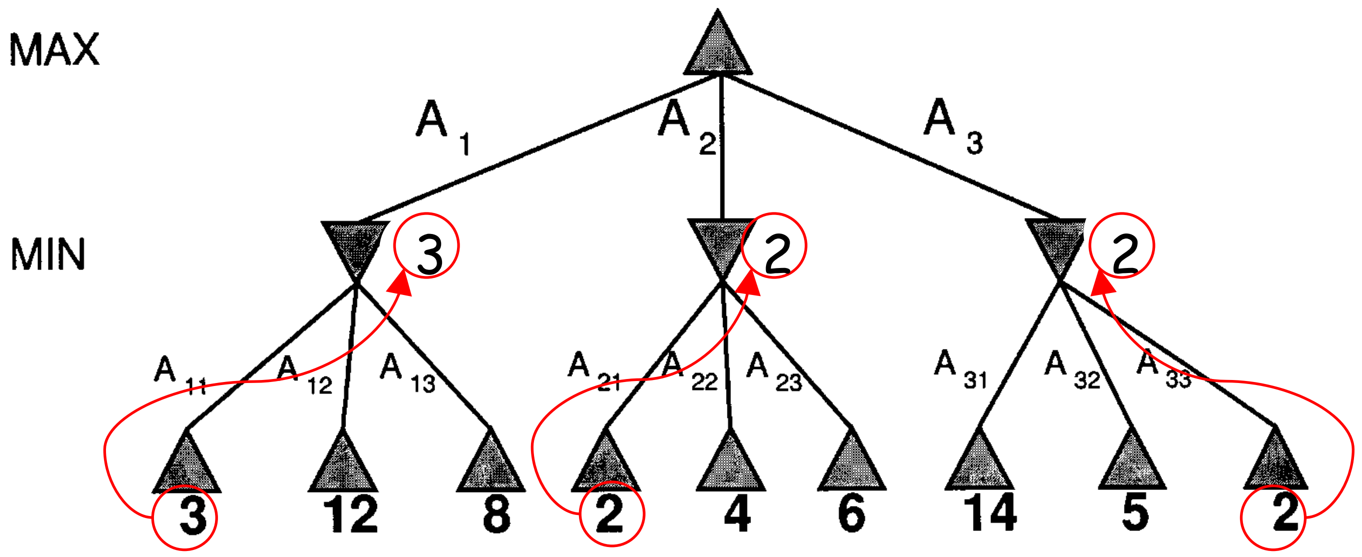
- Generate the game tree
    - Create start node as a MAX node (MAX's turn to move) with the current board configuration
    - Expand nodes down to terminal states [or to some depth (look ahead) if there is resource limitation – time or memory]
- Evaluate the utility of the terminal states (leaf nodes)
- Starting at the leaf nodes and going back to the root of the tree, compute recursively the minimax-value of each node:
    - at MIN nodes (i.e. in MIN's turn) – take the min of the children's values
    - at MAX nodes (i.e. in MAX's turn) – take the max of the children's values
- When the root node is reached, select the best move for MAX
    - i.e. the move which determined the value at the root (= the max of the minimax values of children)

Example - Game Tree for a 2-ply game



- **MAX wants the highest score**
- **MIN wants the lowest score**

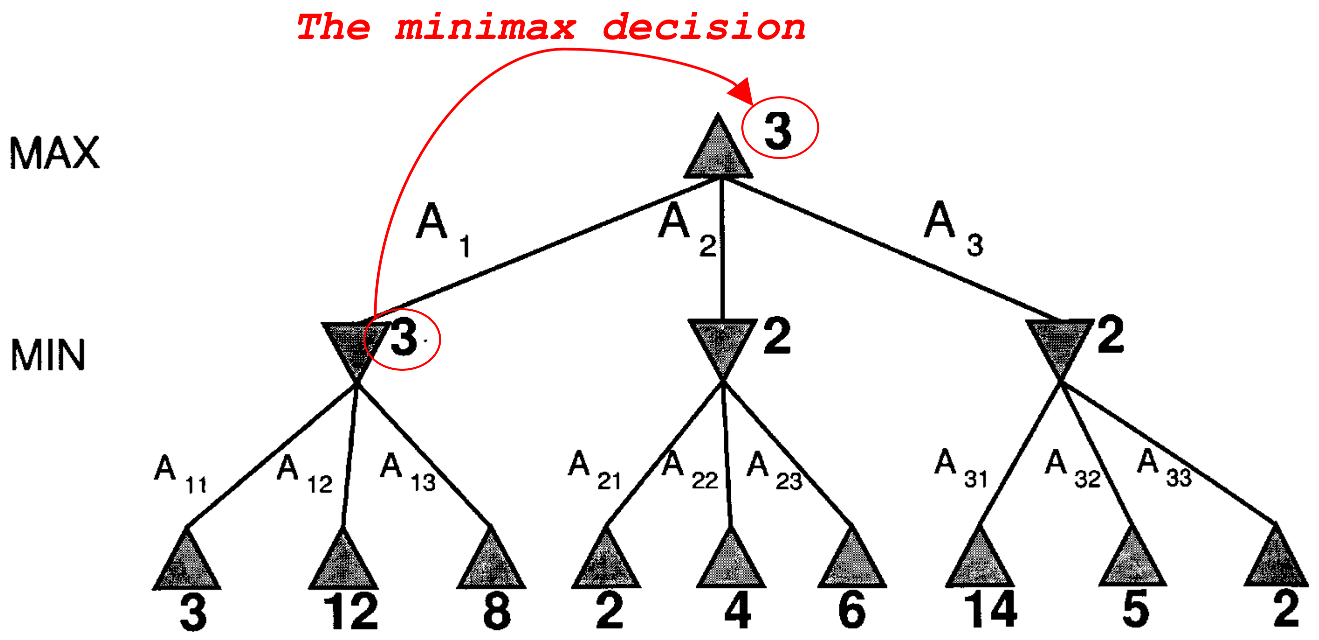
Example - Game Tree for a 2-ply game (2)



- **MAX wants the highest score**
- **MIN wants the lowest score**

At MIN nodes – take the min of the children's values

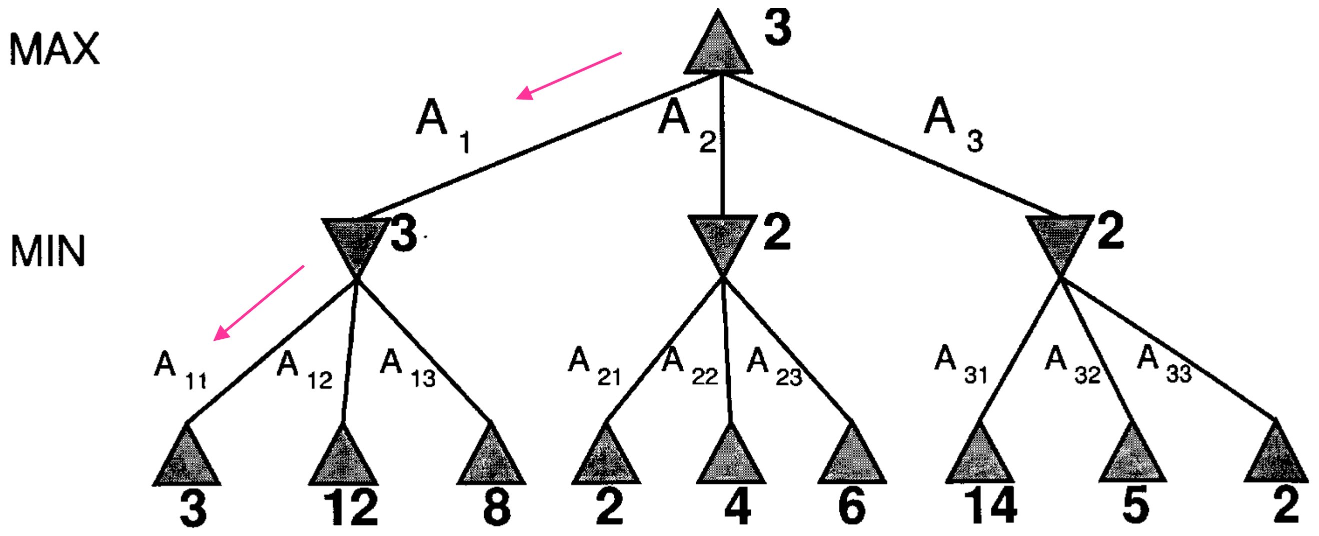
Example - Game Tree for a 2-ply game (3)



- **MAX wants the highest score**
- **MIN wants the lowest score**

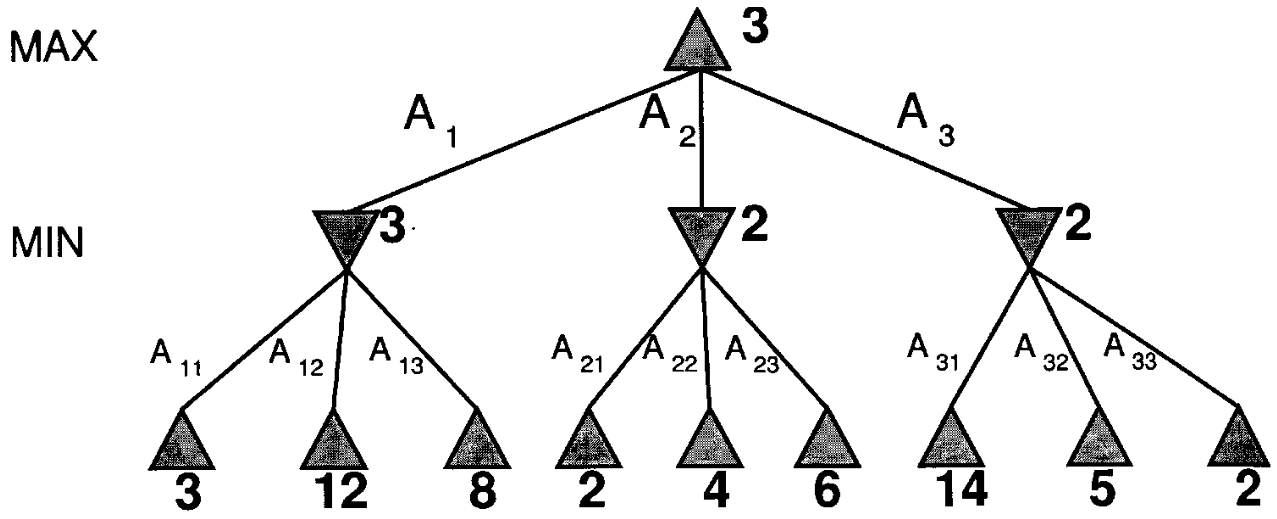
At MAX nodes – take the max of the children's values

Example - Game Tree for a 2-ply game (3)



- **Which move should MAX choose? The one with the highest evaluation function => the left most (3)**
- **If MIN plays optimally, what will be the result of the game?**
  - **3 - MIN will choose the lowest value (the left branch)**

Check that Mimimax Achieved the Best Possible Outcome for MAX Assuming MIN Played Optimally



3 options for MAX:

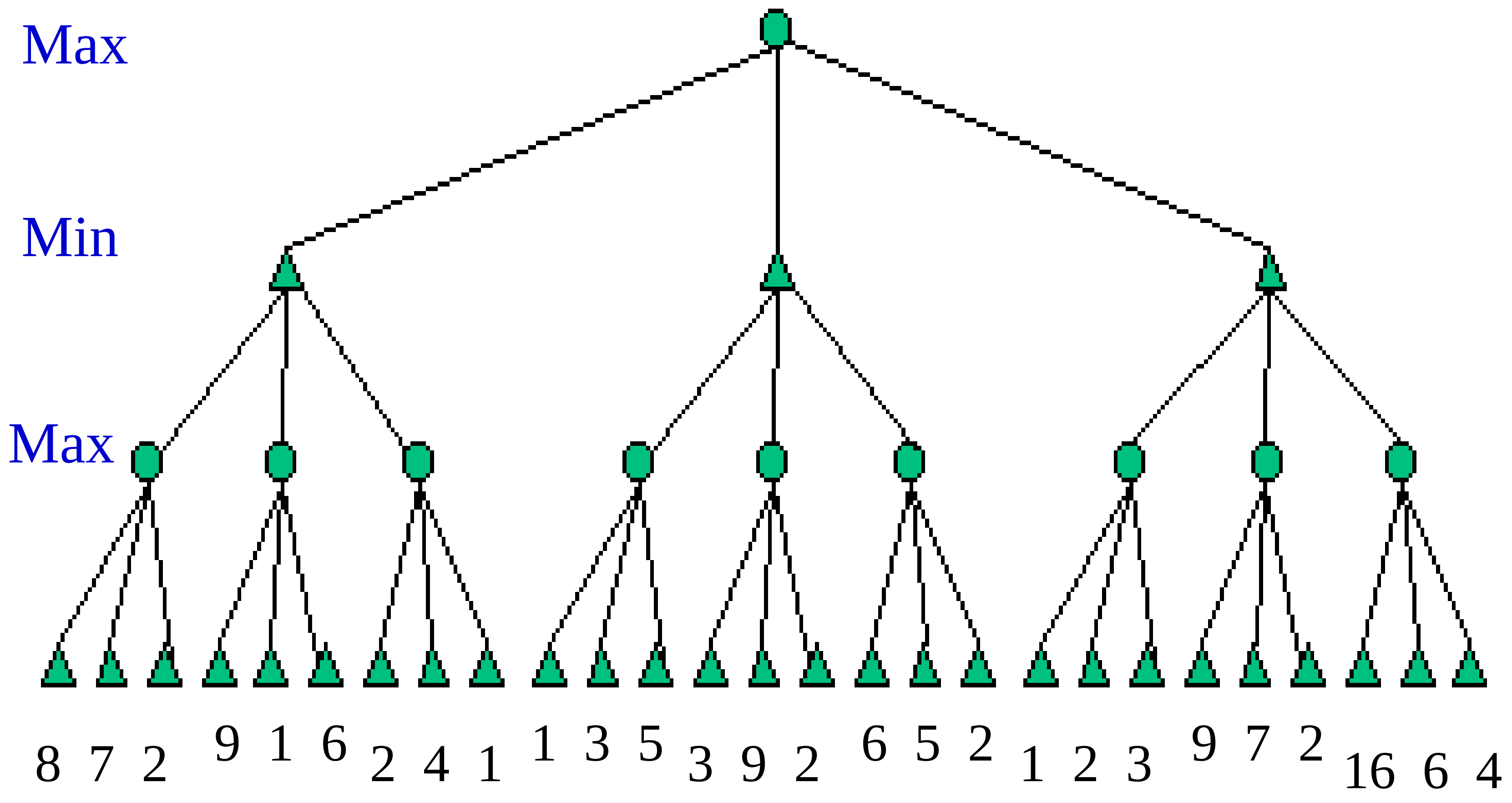
1) If MAX follows left branch and MIN plays optimally: utility =3

2) If MAX follows middle branch and MIN plays optimally: utility=2

3) If MAX follows right branch and MIN plays optimally: utility=2

- **=> yes, 1) is the best option for MAX (highest utility) = optimal move**

Another Example



Minimax Algorithm - Pseudocode

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v←MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← - ∞
  for a,s in SUCCESSORS(state) do
    v ← MAX(v,MIN-VALUE(s))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MIN(v,MAX-VALUE(s))
  return v
```

Minimax Algorithm – Typical Implementation

Example from http://pages.cs.wisc.edu/~skrentny/cs540/slides/7_gamePlaying.ppt

For each move by MAX (first player):

1. **Perform depth-first search to a terminal state**
2. **Evaluate each terminal state**
3. **Propagate upwards the minimax values**

   if MIN's move, minimum value of children backed up

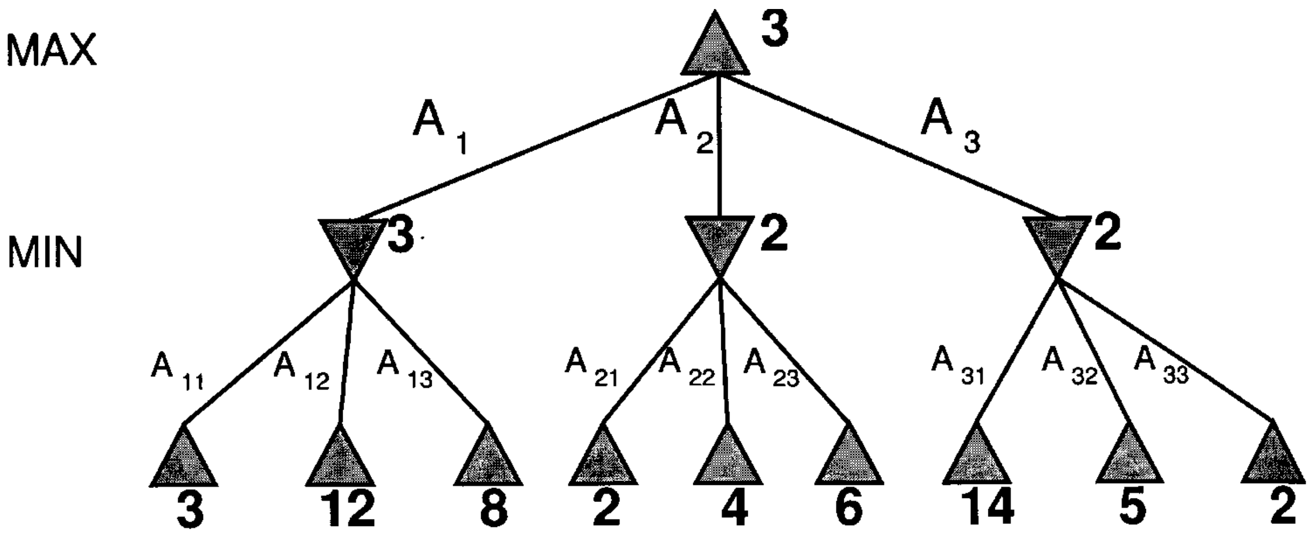
   if MAX's move, maximum value of children backed up
4. **Choose move with the maximum of minimax values of children**

Note:

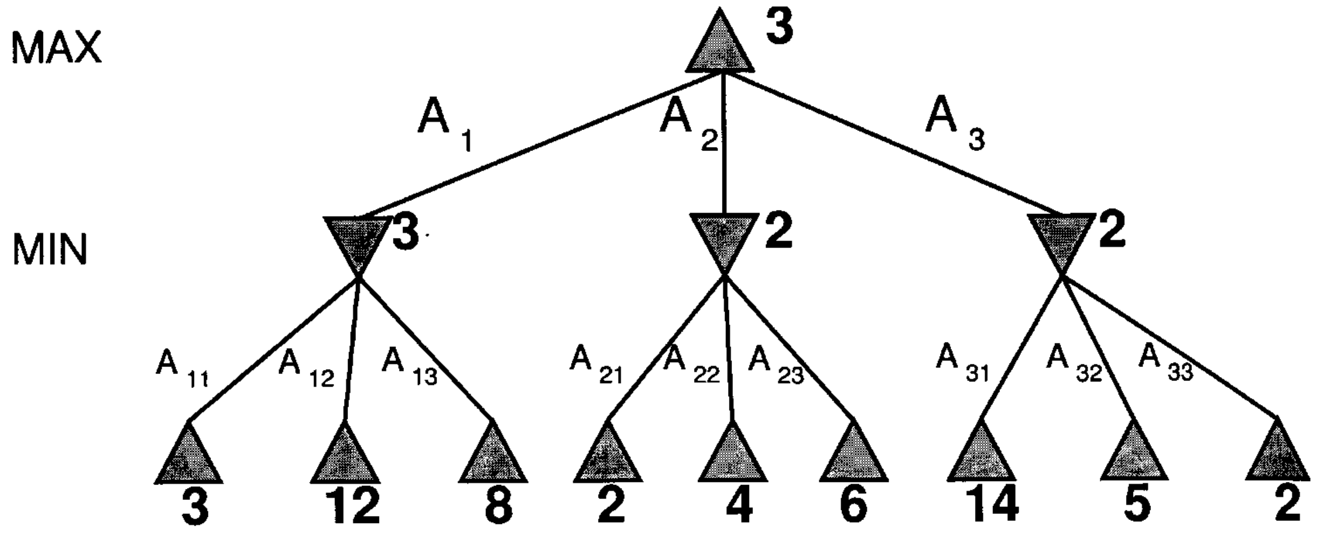
- **minimax values gradually propagate upwards as DFS proceeds: i.e., minimax values propagate up in left-to-right fashion**
- **minimax values for sub-tree backed up as-we-go,**
  so only *O(bd)* nodes need to be kept in memory at any time

What if MIN Does Not Play Optimally?

- **In Minimax MAX assumes that MIN plays optimally: maximizes worst-case outcome for MAX**
- **But if MIN does not play optimally, MAX will do even better! (proven)**



Checking that MAX will do Even Better if MIN does not Play Optimally



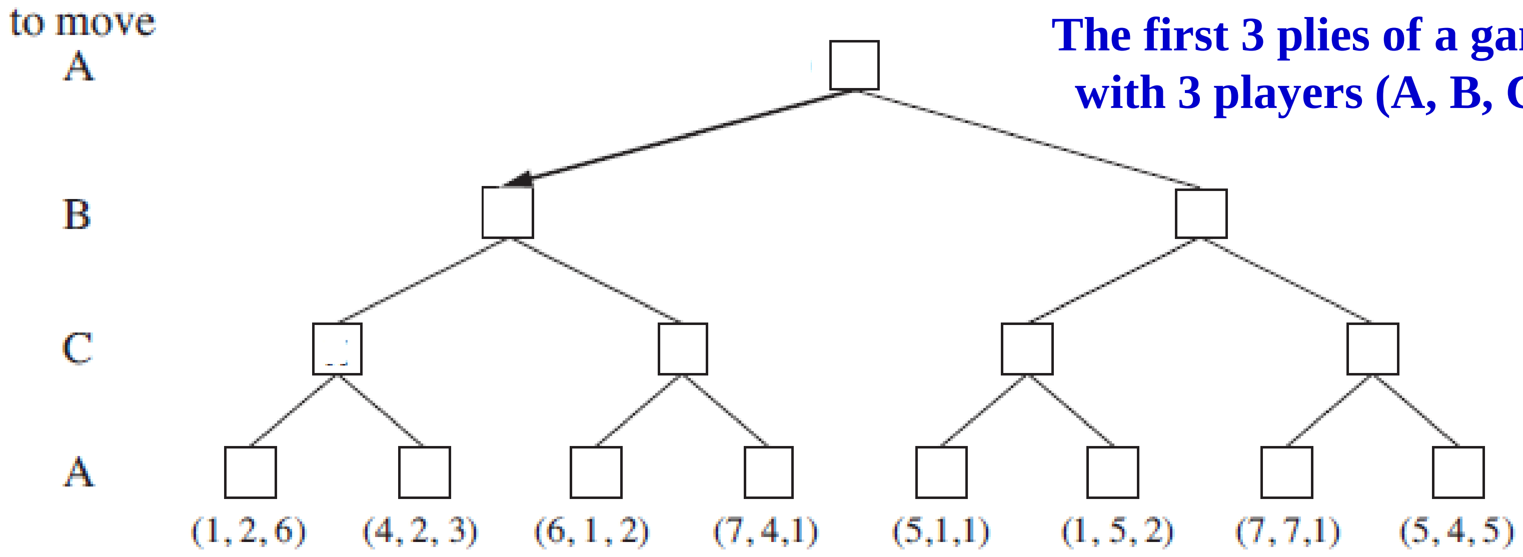
1) If MAX follows left branch and MIN does <u>not</u> play optimally: utility = 12 or 8, better result for MAX than utility=3 if MIN plays optimally

2) If MAX follows middle branch and MIN does <u>not</u> play optimally: utility = 4 or 6, better result for MAX than utility=2 if MIN plays optimally

3) If MAX follows right branch and MIN dies <u>not</u> play optimally: utility = 14 or 5, better than utility=2 if MIN plays optimally

Games with More Than Two Players

- **Many games allow more than two players**
- **Single minimax values become vectors with a value for each player**
- **Each player is maximizing its value**
- **The whole vector is backed up, not a single value**
- **Example: a three player game**

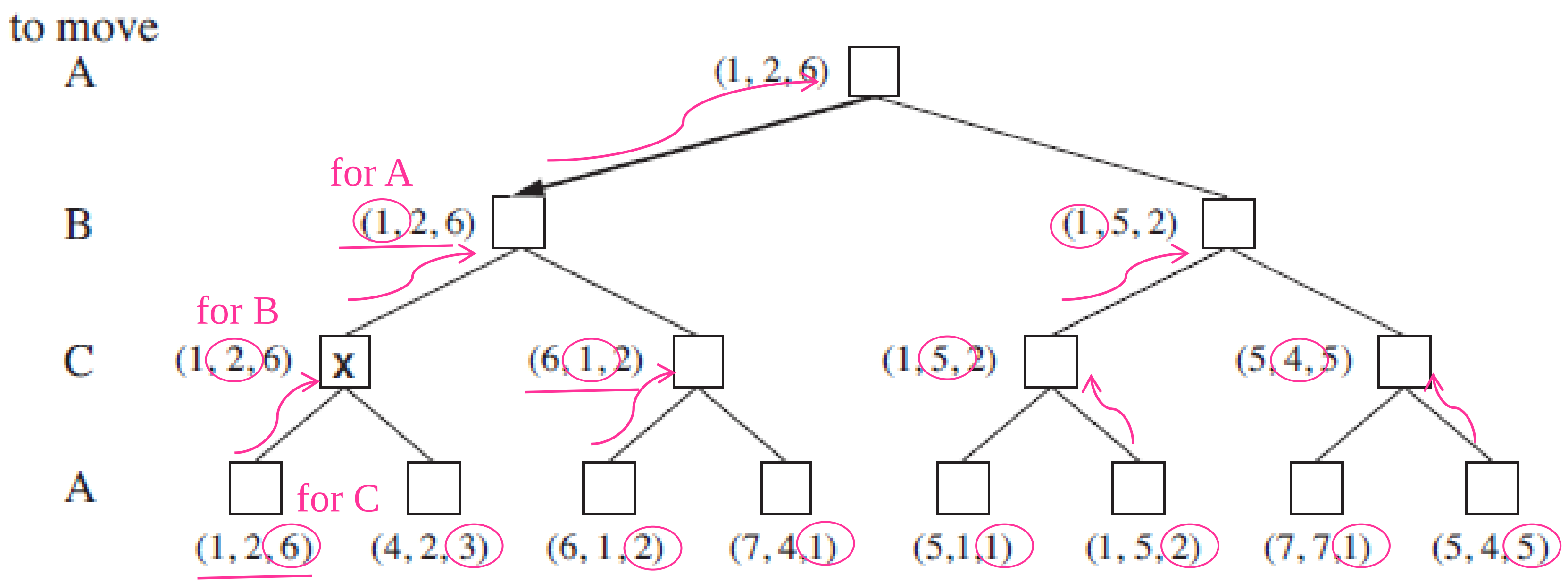


The first 3 plies of a game with 3 players (A, B, C)

Each node is labelled with utility values from the viewpoint of each player

Multiplayer Games (2)

- Backed-up value of a node ***n*** is the utility vector of whichever child has the highest value for the player choosing at ***n***
- Example: player B chooses a move
  - Compare the second element of (1,2,6) and (6,1,2), 2>1 => B chooses the node (1,2,6)



Properties of Minimax

- **Implemented as DFS**
- **Assumptions: branching factor *b*, all terminal nodes at depth *d***
- **Optimal? – will the optimal score be reached?**
  - **"optimal score"- the score of the terminal node that will be reached if both players play optimally**
  - **Yes, against an optimal opponent (i.e. MAX and MIN play optimally)**
  - **Against a suboptimal player? I.e. MAX plays optimally but MIN plays sub-optimally: the score of the terminal node reached will never be lower than the optimal score (see slide 28 and the tutorial exercises)**
- **Time complexity? $O(b^m)$ as in DFS**
- **Space complexity? $O(bm)$ as in DFS**
- **Time is the major problem – moves need to be chosen in a limited time**

Resource Limits

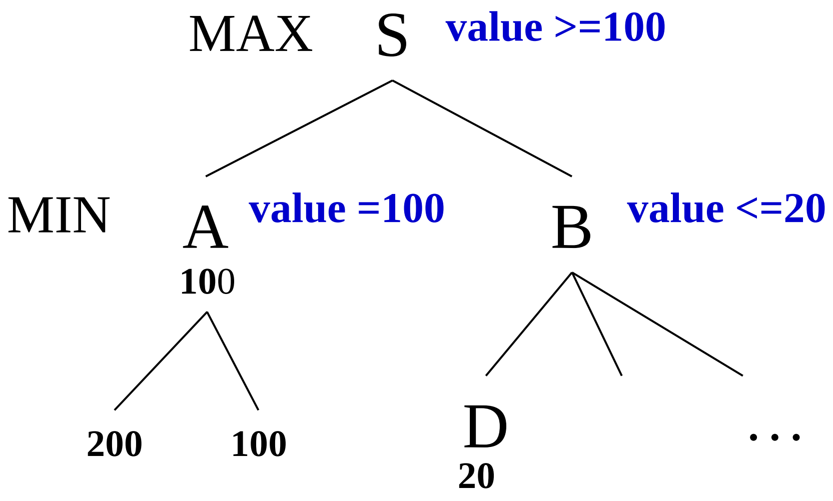
- Minimax assumes that the program has time to search all the way to terminal states, which is not always possible
- In most cases we have limited resources, especially time
  - How far ahead can we look? Consider chess and suppose that:
    - we have 100 seconds per move
    - in 1 second we can explore 10 000 nodes

    => i.e. at each move we can explore 10^6 nodes:

    => $b^m=10^6$, for b=35 => m=4 ply ahead
  - But this is not good enough
    - 4 ply = human novice
    - 8 ply = human master
    - 12 ply = Kasparov and Deep Blue
- Is there any algorithm that is guaranteed to produce the same result as minimax but will generate less nodes? Yes!

Not All Nodes Are Worth Exploring

- **There is no need to evaluate the other children of B as we already know that MAX will not choose B, it will choose A**



"If you have an idea that is surely bad, don't waste time to see how truly awful it is." **Patrick Winston**

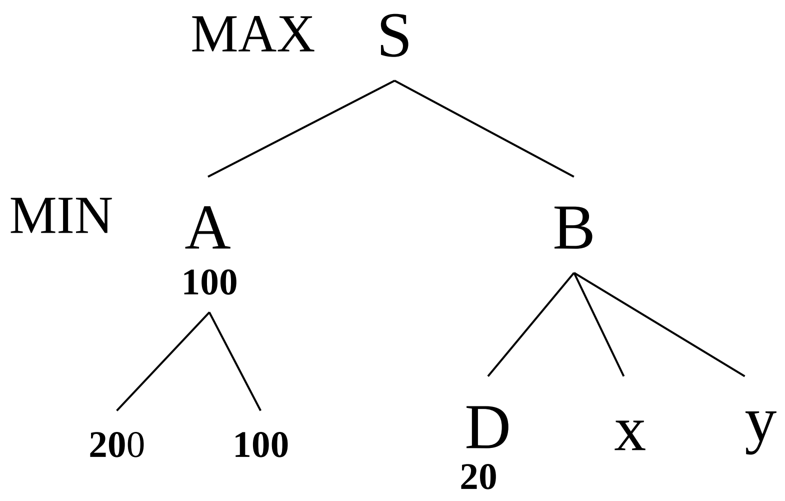
Why Not All Nodes Are Worth Exploring

Let the values of the 2 non-evaluated children of B are x and y

minimax(S) = max(A,B) = max(min(200,100),min(20,x,y)) =

= max(100, ≤ 20) =100

=> The value of the root is independent of the values of the leaves x and y => we can prune x and y – there is no need to generate and evaluate them



Alpha-Beta Pruning - Idea

- No need to examine every node; prune the nodes that do not influence the decision

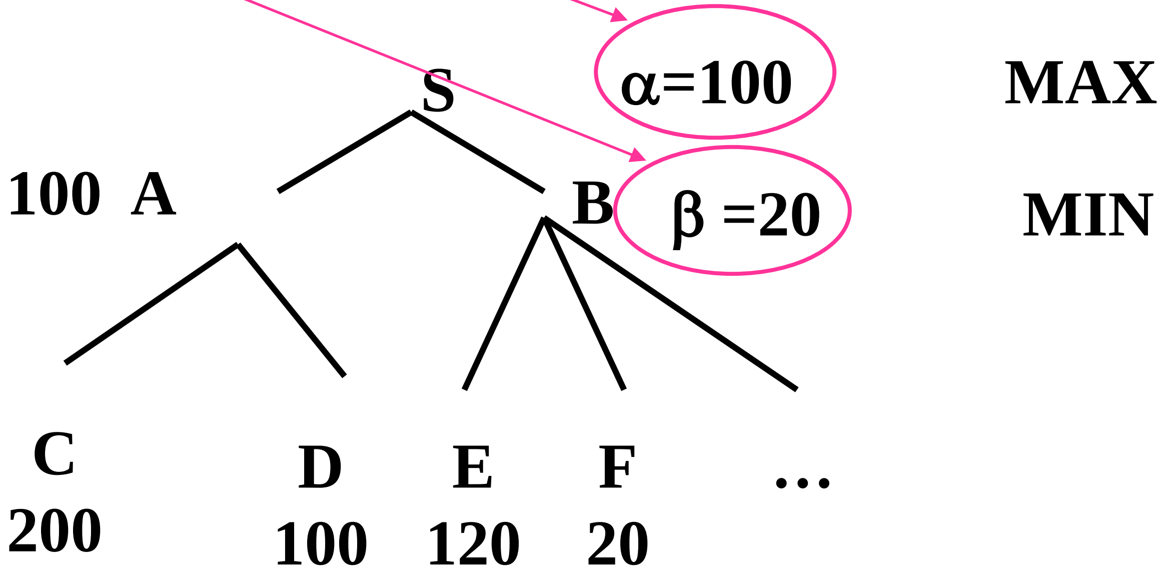
- If alpha-beta pruning is applied to a standard minimax tree, it returns the same move as minimax, but prunes away brunches that cannot influence the final decision

Alpha-Beta Pruning

- While doing DFS of game tree, along the path, keep 2 values:
  - Alpha value = the best value for MAX (the highest) so far along the path (initialise to –∞)
  - Beta value = the best value for MIN (the lowest) so far along the path (initialise to +∞)
- If a MAX node exceeds beta, prune the sub-tree below
- IF a MIN node is below alpha, prune the sub-tree below

Closer Look at the Pruning – Alpha Cut-off

- **IF a MIN node is below alpha, prune it *(alpha cut-off)***
  - **i.e. if child's beta <= parent's alpha**
  - **No need to expand B further as MAX can make a better move**



Alpha cutoff

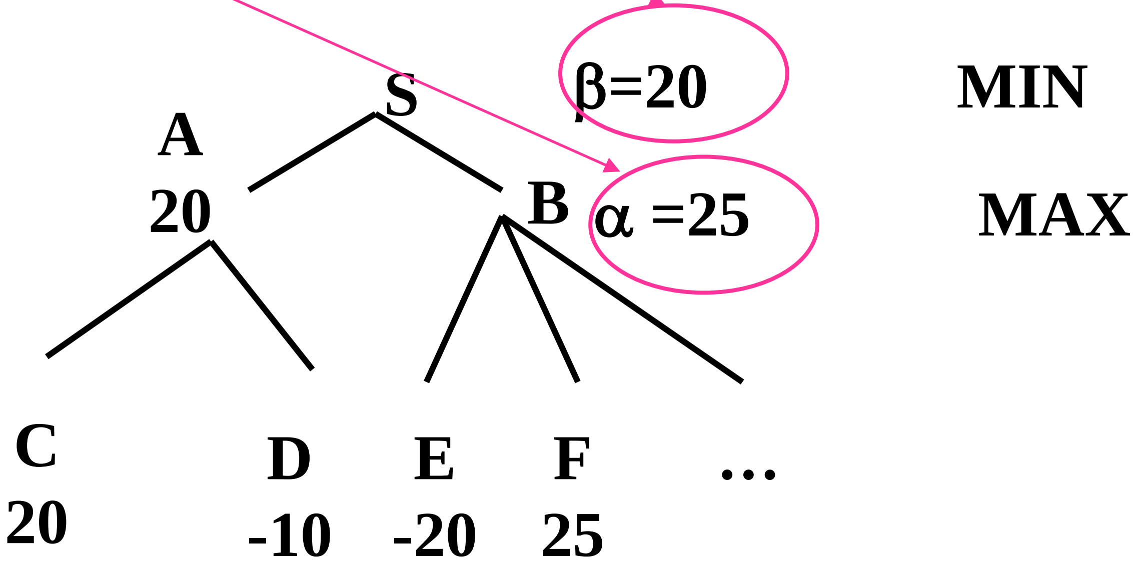
$\beta(B)=20<\alpha(S)=100$

Closer Look at the Pruning – Beta Cut-off

- **If a MAX node exceeds beta, prune it *(beta cut-off)***
  - **child's alpha >= parent's beta**
  - **No need to expand B further as MIN will not allow MAX to take the move**

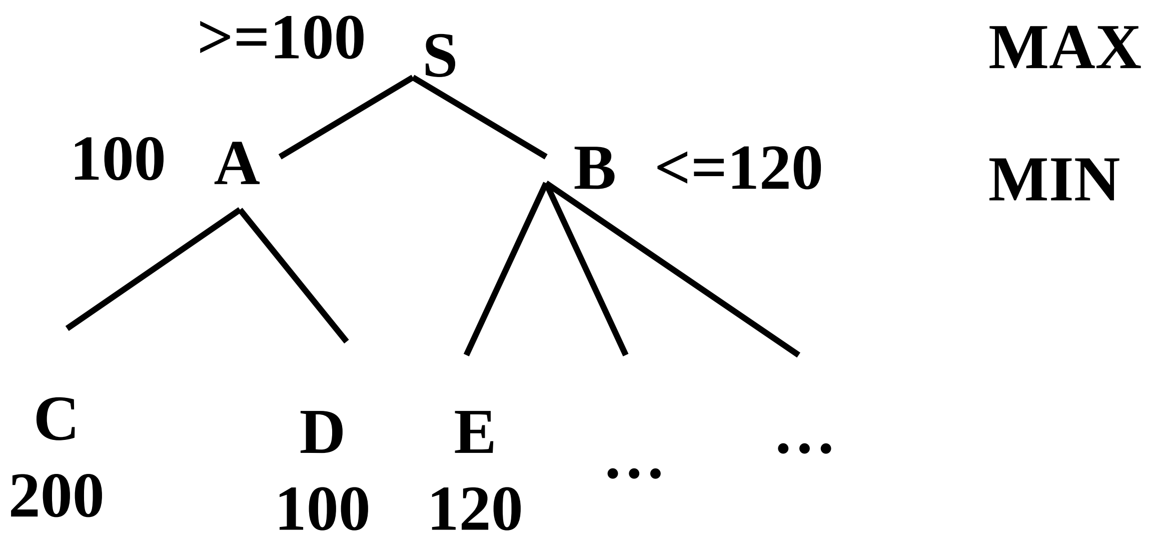
Beta cutoff

$\alpha(B)=25>\beta(S)=20$



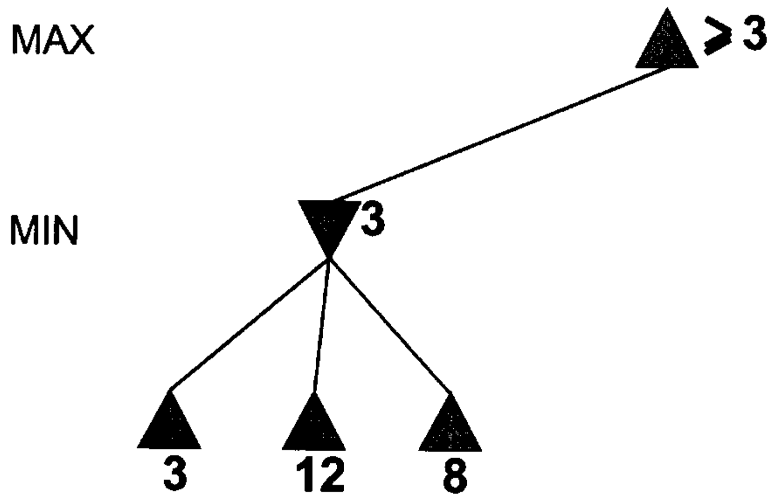
Another Way to Test if it's Possible to Prune

- **Keep alpha and beta bounds, not values**
- **If the intervals they define overlap, pruning is not possible**
- **If the intervals they define do not overlap, pruning is possible**



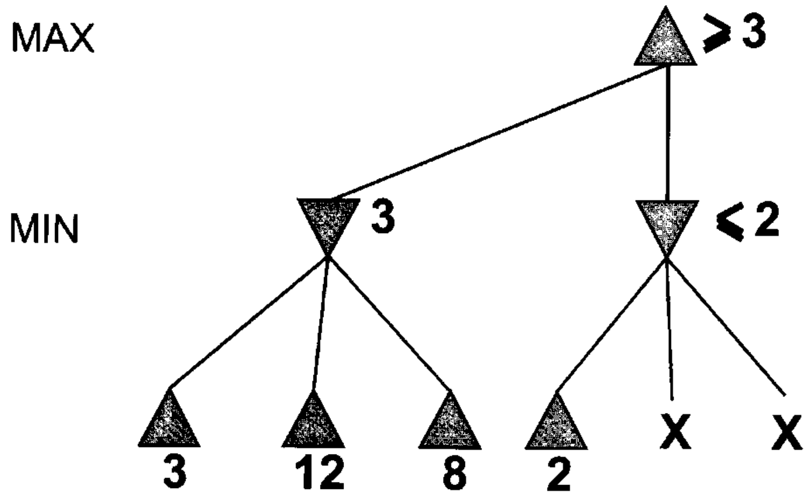
- **Do we need to evaluate the other children of B or we can prune them?**
- **Need to evaluate them as <=120 and >=100 overlap**

Alpha-Beta Pruning – Example



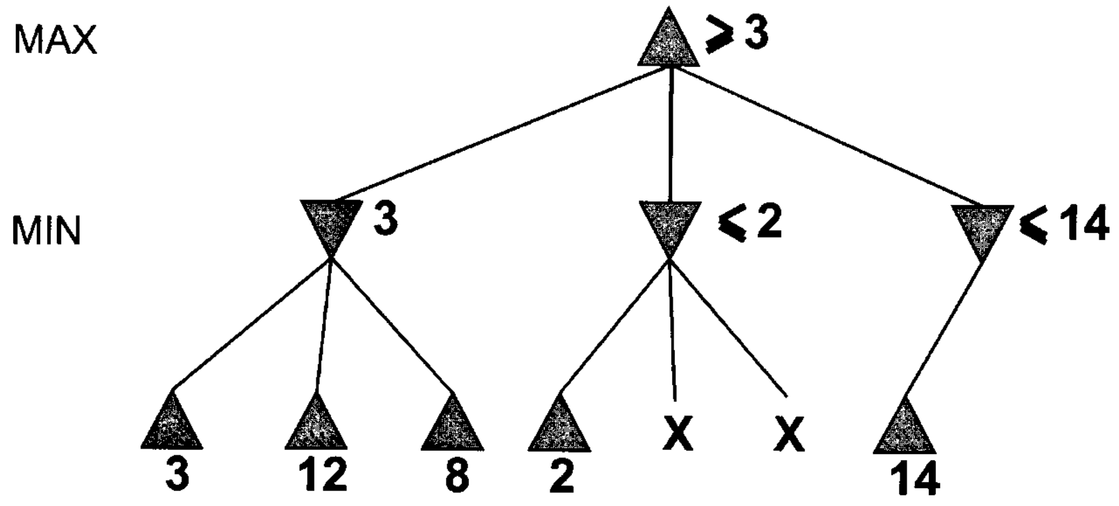
- **We always need to evaluate all children of the first branch – it is not possible to prune them**

Alpha-Beta Pruning – Example



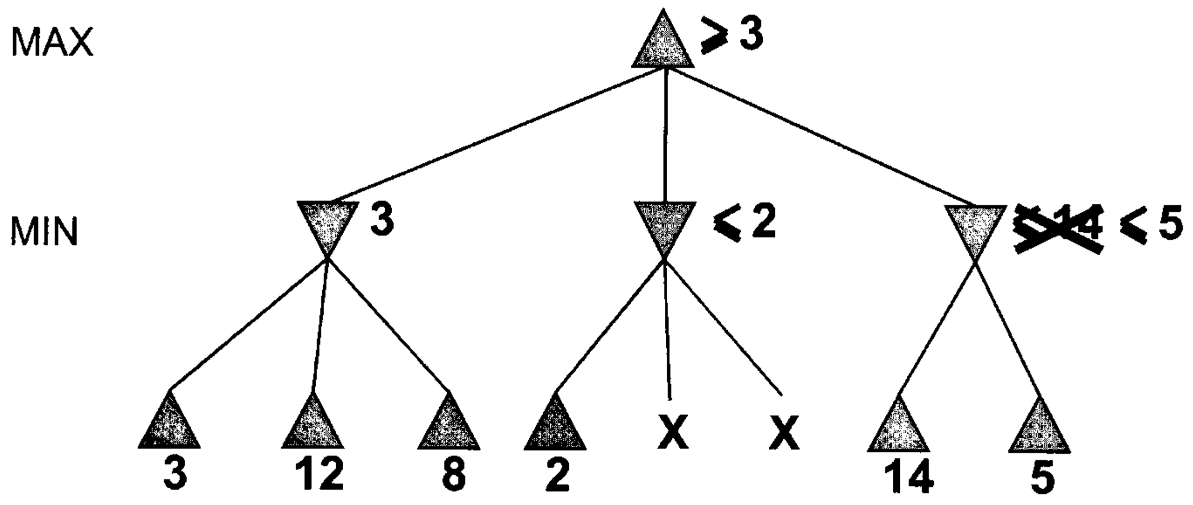
- **The intervals <=2 and >=3 do not overlap, so no need to evaluate the 2 children**

Alpha-Beta Pruning – Example



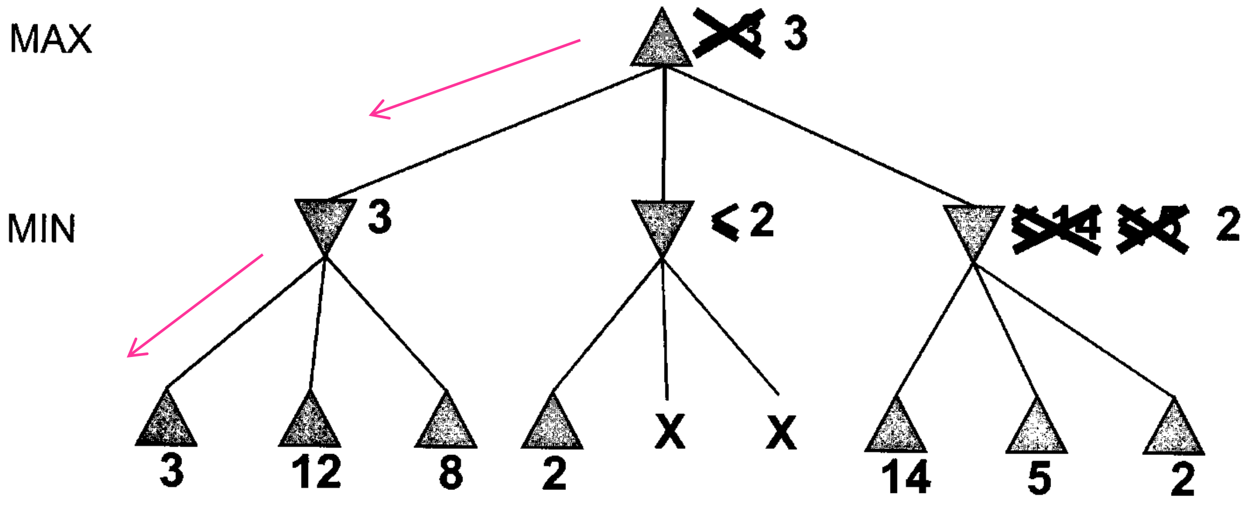
- **The intervals <=14 and >=3 overlap, so the evaluation of children should continue**

Alpha-Beta Pruning – Example



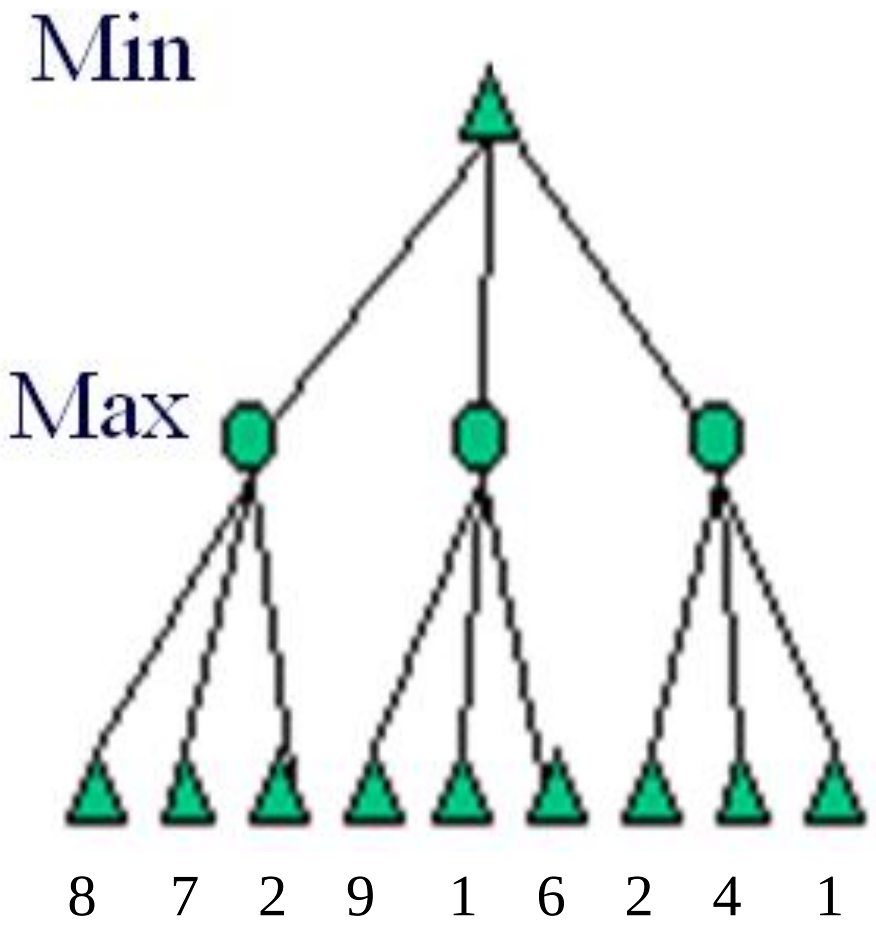
- **The intervals <=5 and >=3 overlap, so we should evaluate the remaining children (no pruning is possible)**

Alpha-Beta Pruning – Example



- **MAX should choose the left most path (value =3)**
- **MIN also should choose the left most path (value =3)**

Another Example



Alpha-Beta Algorithm

- **Traverse the search tree in depth-first order**
- **Apply utility function at each leaf node that is generated**
- **Compute values backwards**
- **At each non-leaf node store a value indicating the best backed up value so far**
  - **MAX nodes: alpha value = best (max) value found so far**
  - **MIN nodes: beta value = best (min) value found so far**
- **Given a node *n*, cutoff the search below *n* if:**
  - ***n* is a MAX node and *alpha(n)* $\geq$ *beta (i)* for some MIN node *i* that is ancestor of *n* (*beta cutoff*)**
  - ***n* is a MIN node and *beta(n)* $\leq$ *alpha(i)* for some MAX node *i* that is ancestor of *n* (*alpha cutoff*)**

Alpha-Beta Algorithm - Pseudocode

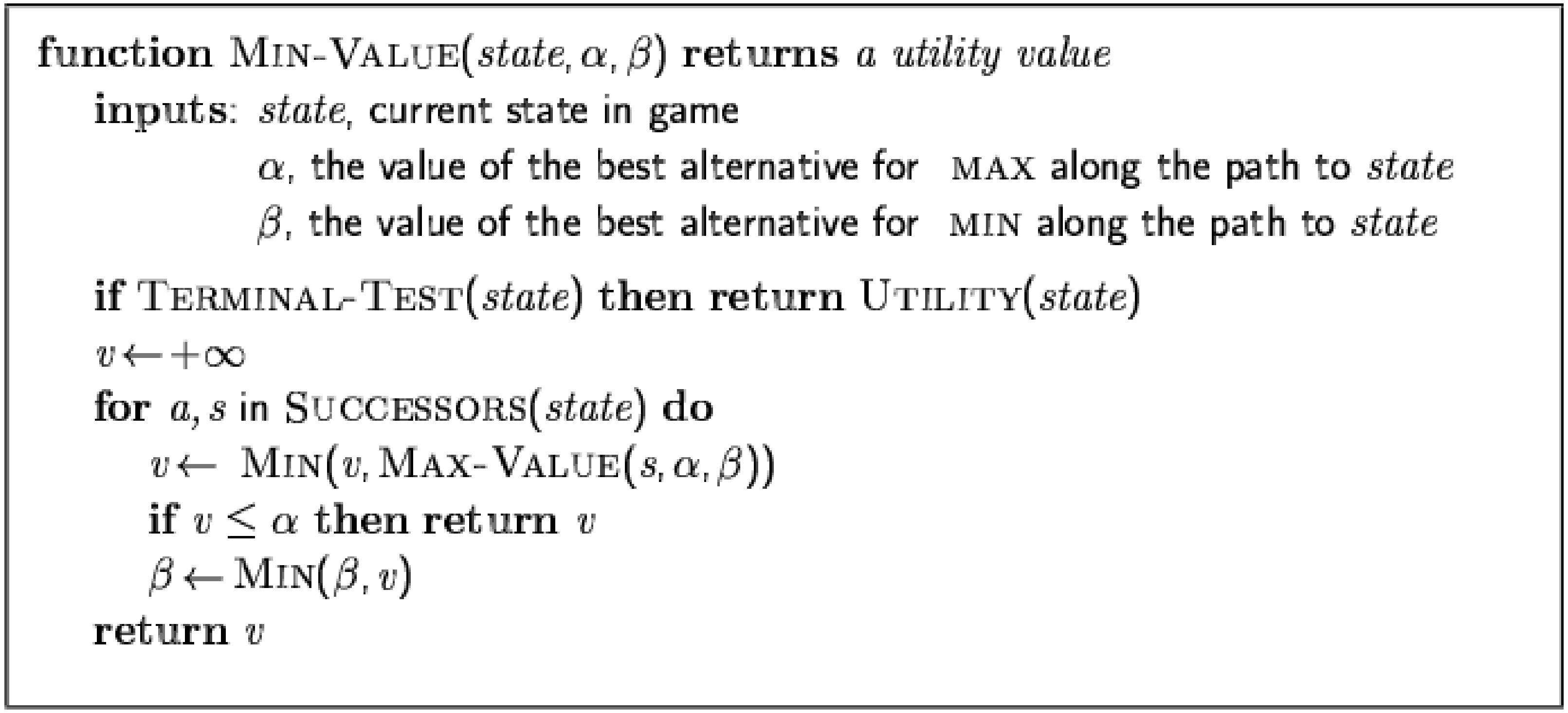
```
function ALPHA-BETA-SEARCH($state$) returns $an\ action$
    inputs: $state$, current state in game

    $v \leftarrow$ MAX-VALUE($state$, $-\infty$, $+\infty$)
    return the $action$ in SUCCESSORS($state$) with value $v$
```

```
function MAX-VALUE($state$, $\alpha$, $\beta$) returns $a\ utility\ value$
    inputs: $state$, current state in game
            $\alpha$, the value of the best alternative for MAX along the path to $state$
            $\beta$, the value of the best alternative for MIN along the path to $state$

    if TERMINAL-TEST($state$) then return UTILITY($state$)
    $v \leftarrow -\infty$
    for $a, s$ in SUCCESSORS($state$) do
        $v \leftarrow$ MAX($v$, MIN-VALUE($s$, $\alpha$, $\beta$))
        if $v \geq \beta$ then return $v$
        $\alpha \leftarrow$ MAX($\alpha$, $v$)
    return $v$
```

Alpha-Beta Algorithm – Pseudocode (cont.)



```
function MIN-VALUE($state, \alpha, \beta$) returns a utility value
    inputs: $state$, current state in game
            $\alpha$, the value of the best alternative for MAX along the path to $state$
            $\beta$, the value of the best alternative for MIN along the path to $state$

    if TERMINAL-TEST($state$) then return UTILITY($state$)
    $v \leftarrow +\infty$
    for $a, s$ in SUCCESSORS($state$) do
        $v \leftarrow$ MIN($v$, MAX-VALUE($s, \alpha, \beta$))
        if $v \le \alpha$ then return $v$
        $\beta \leftarrow$ MIN($\beta, v$)
    return $v$
```

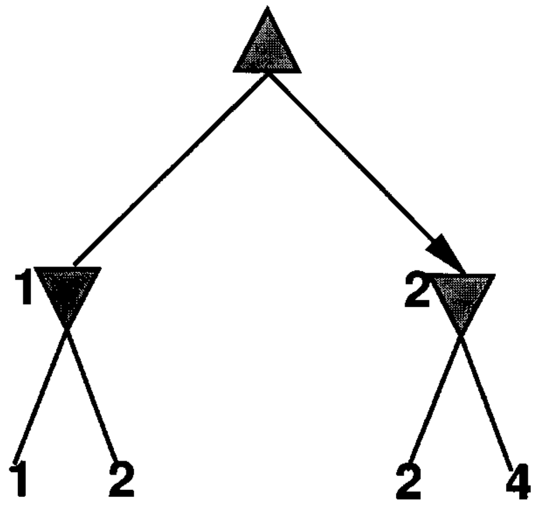
Properties of Alpha-Beta Pruning

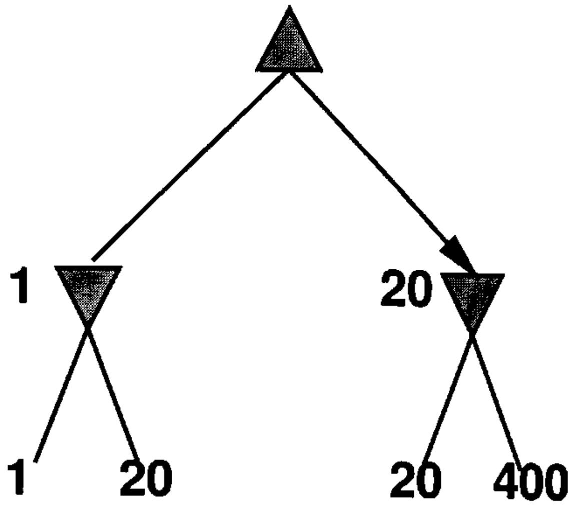
- **Pruning does not affect the final result**
  - **Alpha-beta is guaranteed to compute the same value for the root node as computed by minimax, with less or equal number of evaluated nodes**

- **Good move ordering improves effectiveness of pruning**
  - **Worst case: no pruning, examine b^d nodes (=minimax)**
  - **Best case ("perfect ordering") - examine only $b^{d/2}$ nodes**
    - **Meaning – double depth of look-ahead search; for chess – we can easily reach depth 8 and play good chess**

Exact Values Don't Matter Only the Relative Values Matter

- In deterministic games, the evaluation functions need not to be precise as long as the comparison of values is correct
  - the relative values are important not the actual values
- Behavior is preserved under any monotonic transformation of the evaluation function



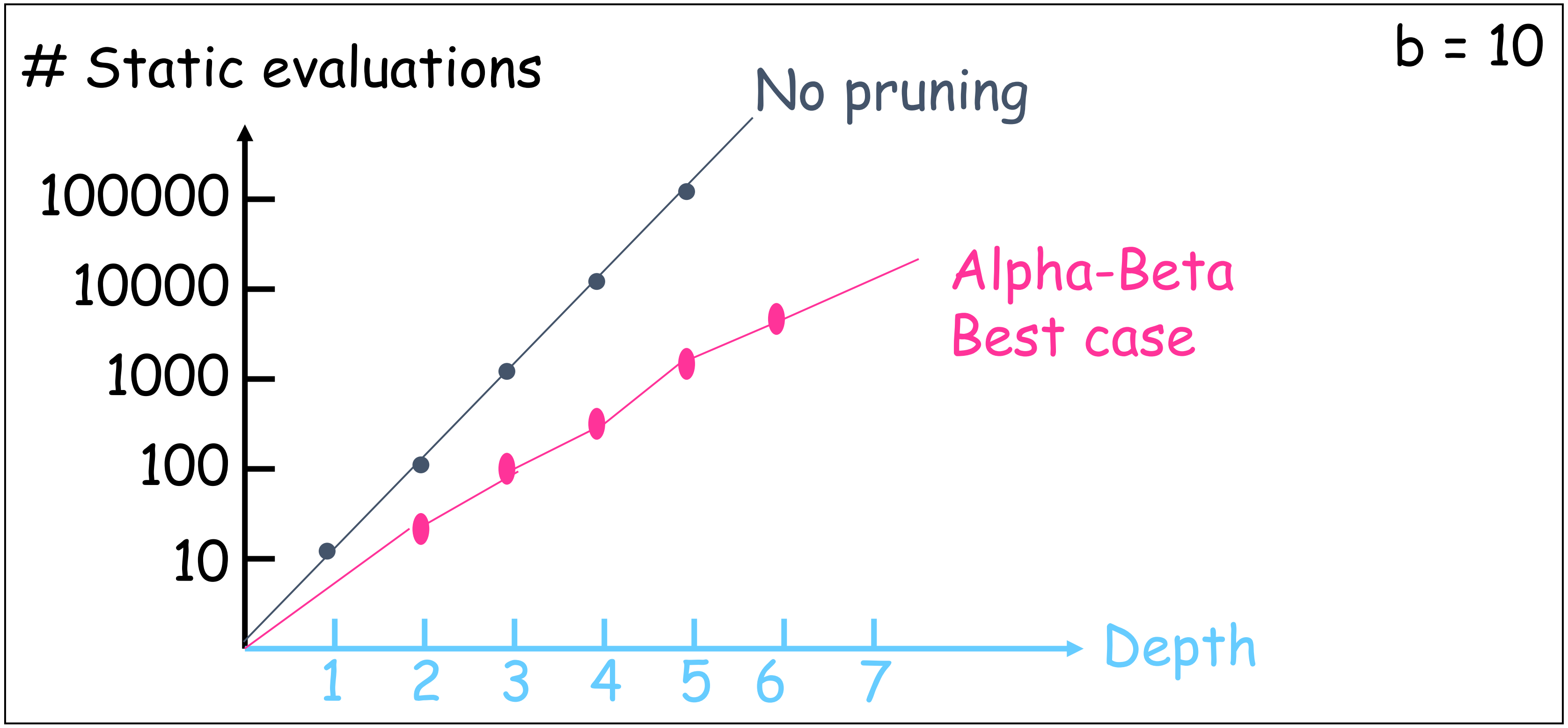


Imperfect, Real-Time Decisions

- **Both minimax and alpha-beta require too many evaluations**
  - **Minimax - the full game tree must be generated to maximum depth**
  - **Alpha-beta – at least some parts of the game tree must be generated to maximum depth (the others may be pruned)**
  - **Example: chess – impossible to generate the complete game tree:**
    - **complete game tree has 10^{40} nodes**
    - **it would take 10^{22} centuries to generate it even assuming that a child node could be generated in 1/3 nano seconds**
    - **compare: the universe is 10^{8} centuries old**
  - **See the graph on next slide**
- **Both algorithms may be impractical**
  - **a move needs to be made in minutes at most**
- **Idea: cut off the search earlier, before reaching the terminal nodes**

www.cs.kuleuven.ac.be/~dannyd/Games.ppt

Alpha-Beta Pruning – Best Case



- **Still exponential growth**

Cutting Off Search Earlier

- Expand the tree only to a given depth (as opposed to maximum depth)
  - The fixed depth is selected so that the amount of time will not exceed what the rules of the game allow
- Apply heuristic evaluation function to the leaf nodes, i.e. treat them as terminal nodes
- Return values to the parents of the leaf nodes as in minimax and alpha-beta

Cutting Off Search Earlier (2)

- This means changing minimax or alpha-beta in 2 ways:
  - Replace the *terminal test* by a *cutoff test*
  - Replace *utility function* by a *heuristic evaluation function* (=estimated desirability of a position)

- Pseudocode change in `minimax` and `alpha-beta-search`:

if TERMINAL-TEST(*state*) then return UTILITY(*state*) becomes

if CUTOFF-TEST(*state*, *depth*) then return EVAL(*state*)

when the cut-off depth is reached call the heuristic evaluation function

- Instead of fixed depth search (alpha-beta uses depth first search), iterative deepening (depth-first) search can be used
  - When the time runs out, the program returns the move selected by the deepest completed search

More on Evaluation Functions - Evaluation Functions for Chess

- **Typical evaluation function - weighed linear sum of features:**

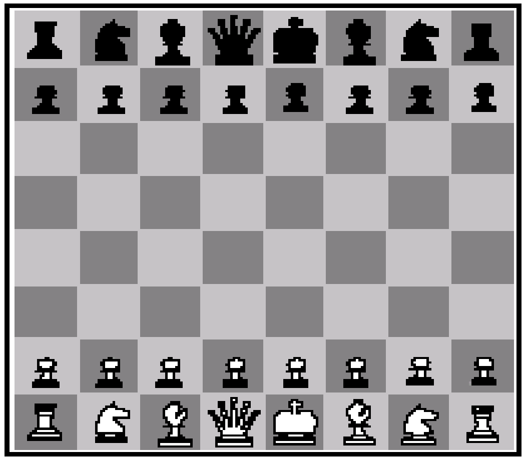
  $EVAL(s)=w_1f_1(s)+w_2f_2(s)+\ldots+w_nf_n(s)$,

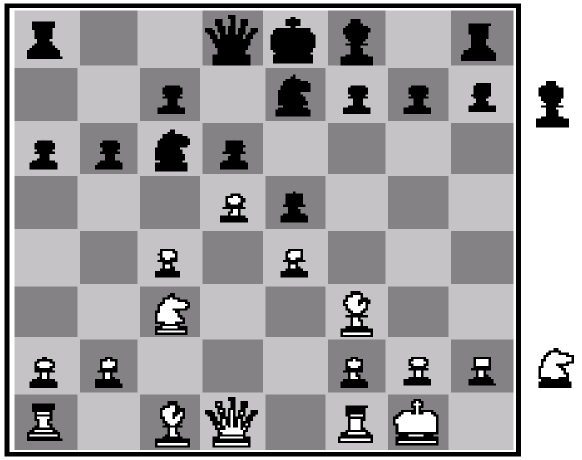
  where f_i – feature i, w_i – weight of feature i
- **Example**
  - **f = number of different pieces on the board, e.g. f1 – number of pawns, f2 – number of bishops**
  - **w = material value of the piece, e.g. w=1 for pawn, w=3 for bishop, w=5 for rook, w=9 for queen**
- **Other features**
  - **combinations of number of pieces, e.g. # white queens - # black queens**
  - **“good pawn structure”, “king safety”, etc.**

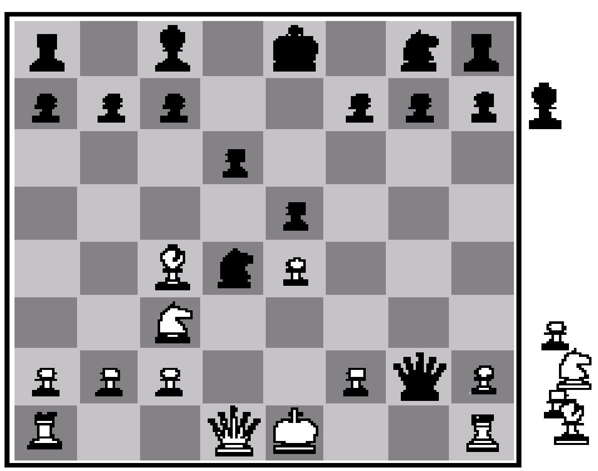
Some Problems with These Evaluation Functions

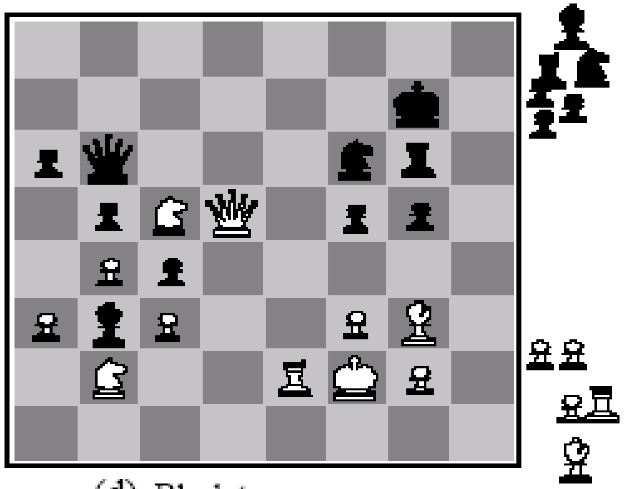
- **A linear combination assumes that the pieces are independent from each other and have the same value during the game – not true, e.g.**
  - **A pawn is more powerful when there are many other pawns on the board (good pawn structure); the higher the number, the higher the value of the pawn**
  - **Bishops are more powerful at the end of the game when they have a lot of space to move than at the beginning**
    - **=> the bishop value correlates with the number of other pieces**
    - **=> the bishop value does not have a constant value of 3 during the game**

Some Chess Positions and Their Evaluation


(a) White to move
Fairly even

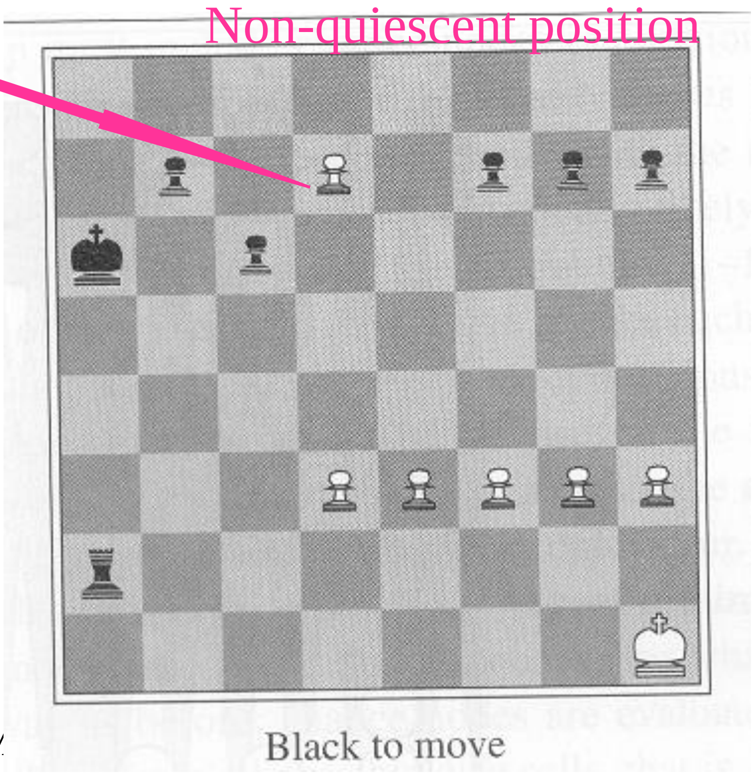

(b) Black to move
White slightly better


(c) White to move
Black winning


(d) Black to move
White about to lose Why?

Horizon Effect

- Hidden pitfall not captured by the heuristic = sudden damaging change just outside the search window that we can't see
    - Example: Black has material advantage but if White can advance his/her pawn, it becomes a queen and White will win
    - We should not stop the search here, as we will not be able to see this sudden change

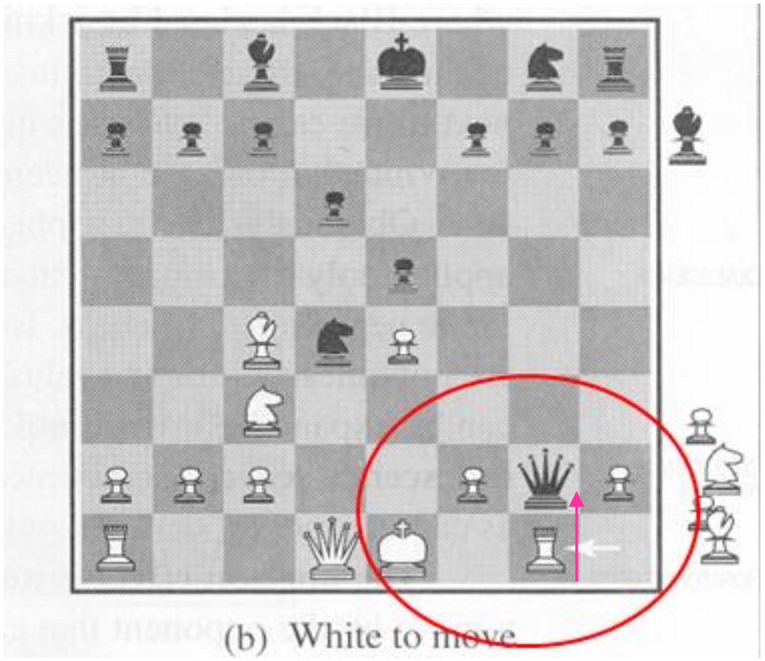


- One of the main difficulties in game paying programs

Horizon Effect – Another Example

- We have a heuristic that counts material advantage, e.g. pieces won, and Black has an advantage
- But White is to move and he/she will capture the queen and the heuristic value will drastically change and Black may even lose!
- We can only see this if we look 1 more move ahead => i.e. we should not stop the search at this position as it is not a ***quiescent position***

Non-quiescent position



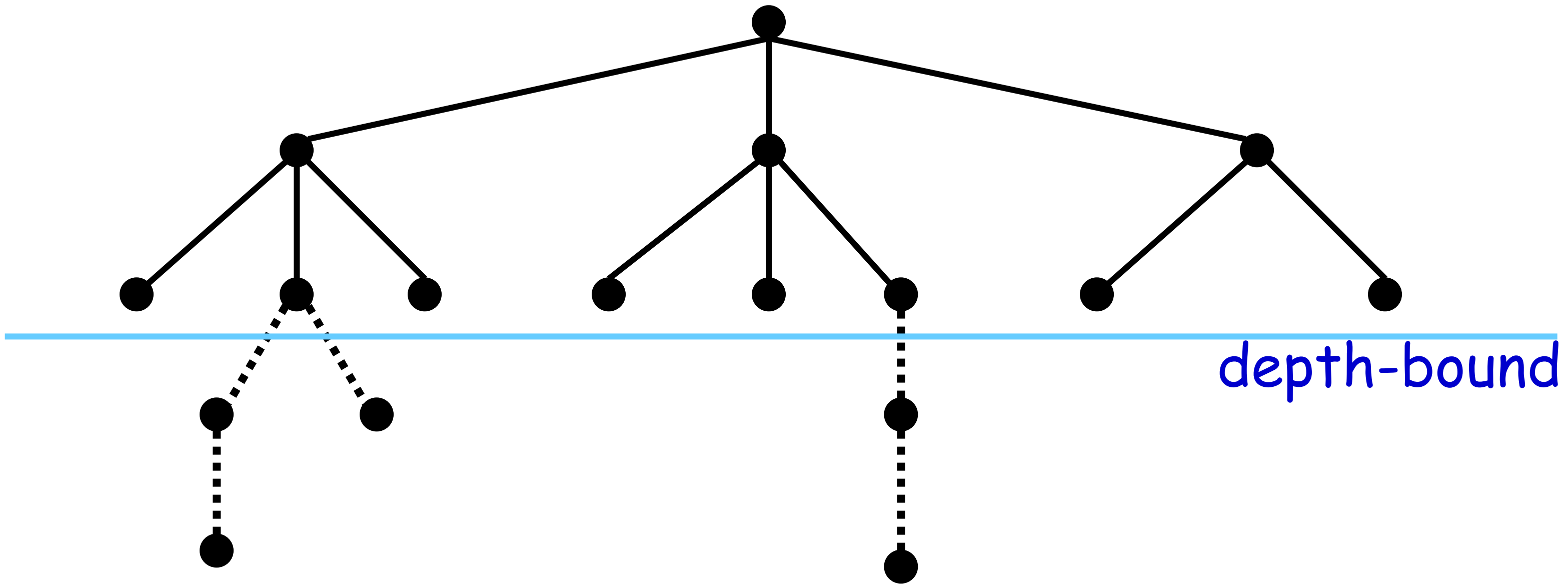
Horizon Effect - Solution

- A better cut-off test is needed!
- The evaluation function should be applied only to positions which are *quiescent*, i.e. unlikely to change extremely in near future
  - examples of non-quiescent positions:
    - a capture can be made after it
    - a pawn become queen after it
- Solution: *secondary* search extends the search for the selected move to make sure that there is no hidden pitfall
  - i.e. non-quiescent positions need to be expanded further until reaching a quiescent position
  - this search is typically restricted to certain moves, e.g. captures

www.cs.kuleuven.ac.be/~dannyd/Games.ppt

Secondary (Extended) Search

- **In general, and especially in strategically crucial situations:**
  - **select the most interesting nodes and extend search for them beyond the depth limit**
  - **check if king in danger, pawn to become queen, piece to be captured, etc.**



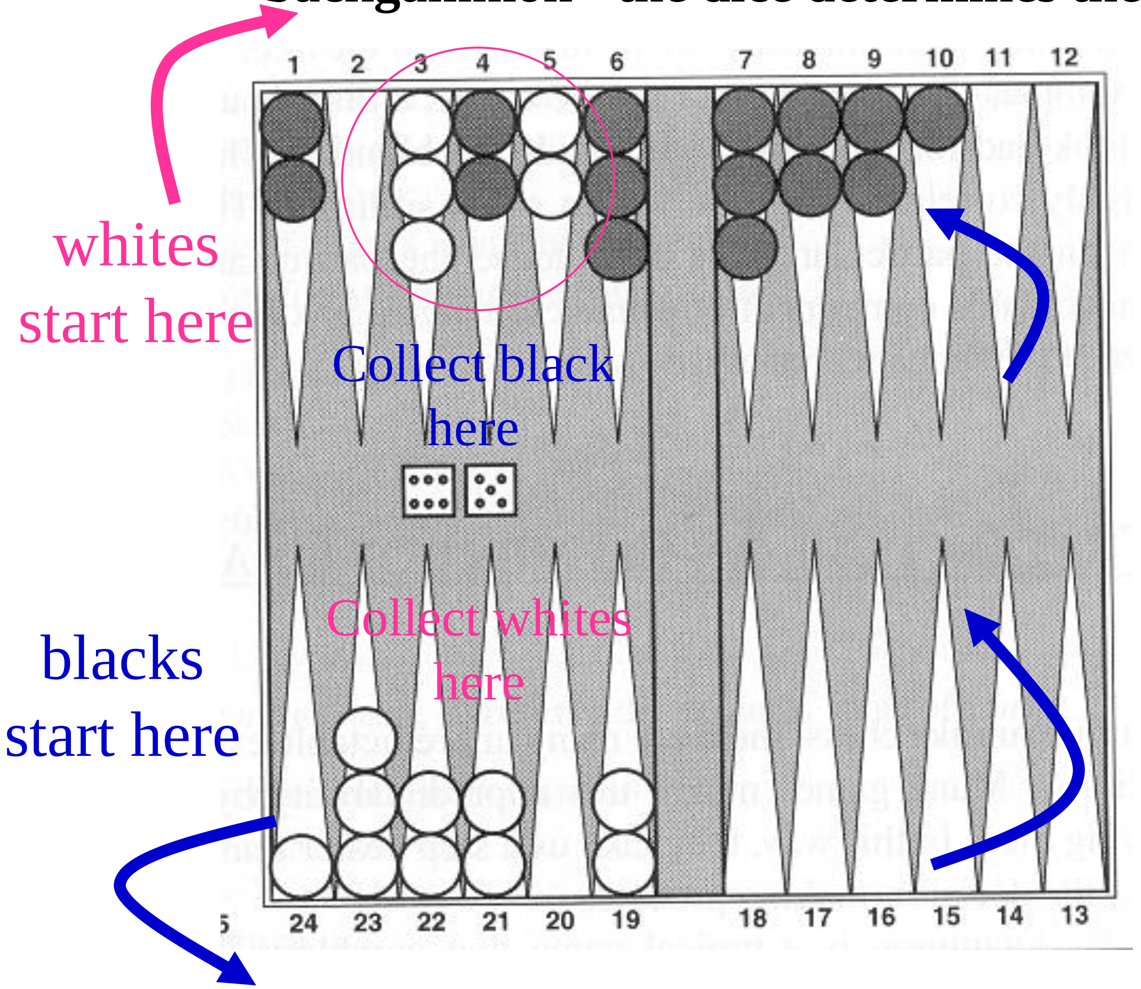
Other Refinements

- **Iterative deepening search: Start depth 1, 2, 3, and so on. When time is up, returns the best solution found so far**
- **Book moves: catalogue of pre-computed sequences of moves for a particular board configuration.**
  - **e.g. chess opening sequence and endgame strategies, checkers endgame strategies**
- **Heuristic pruning to reduce branching factor**
  - **Ordering of the search tree based on how plausible the moves are – more plausible first**
- **Alternative to minimax**
  - **Risking a bad move may lead to a much better board configuration**

Non-Deterministic Games (Games of Chance)

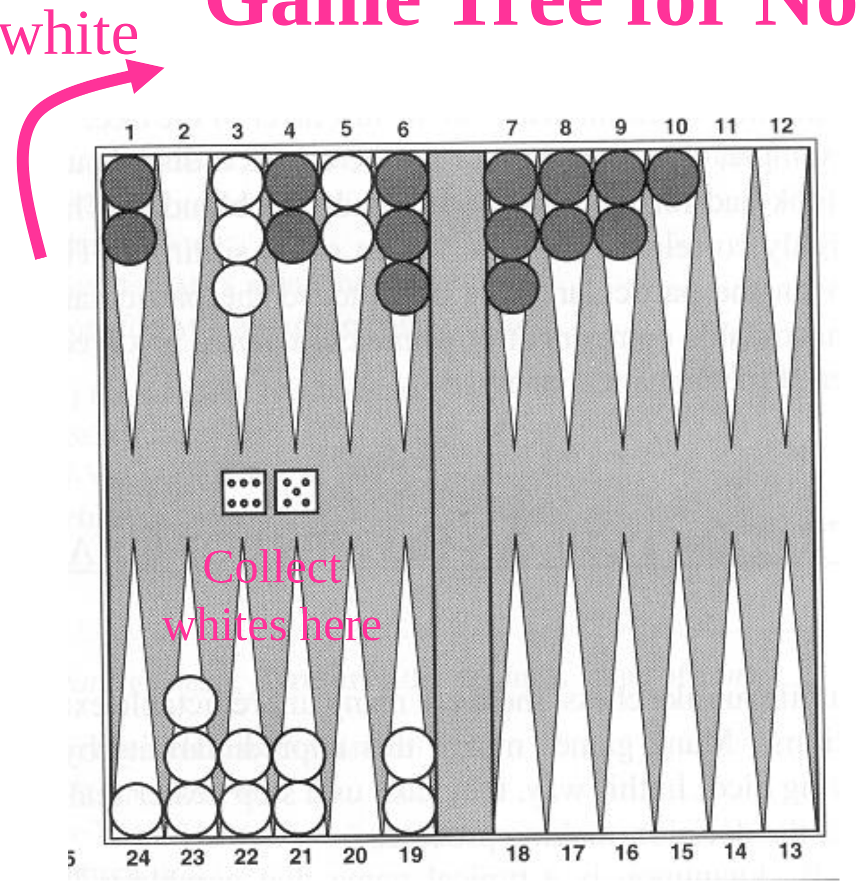
Non-Deterministic Games

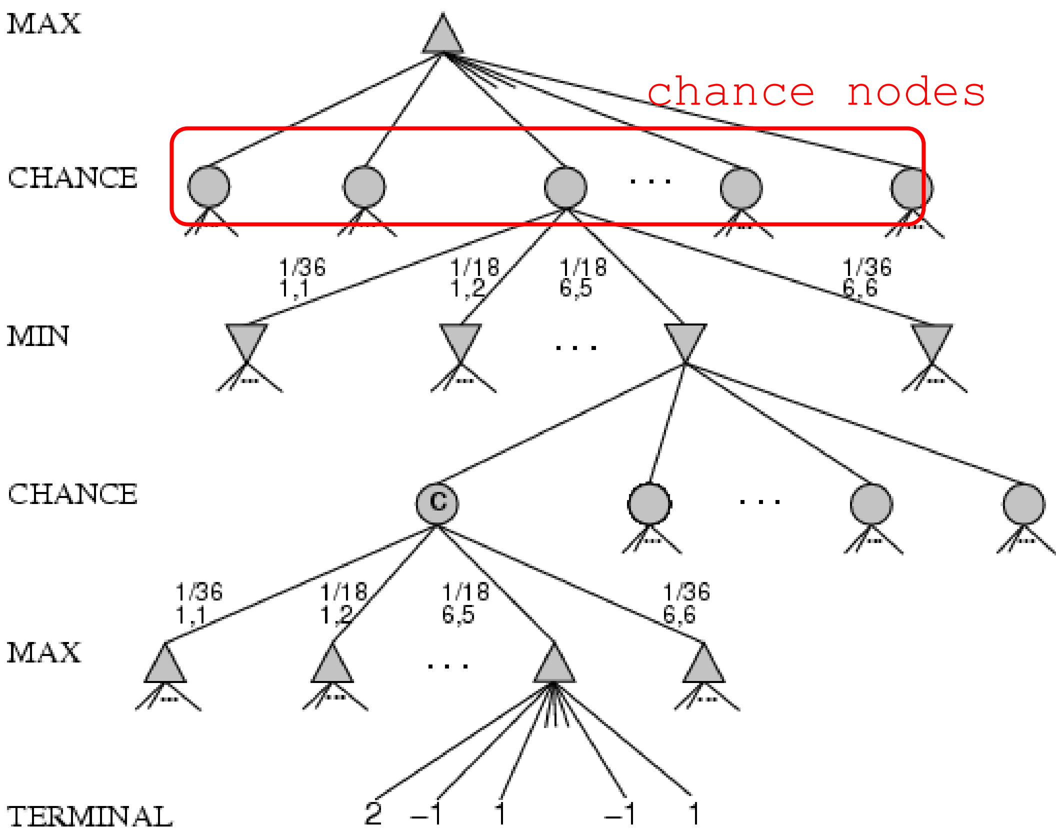
- **Can we apply minimax and alpha-beta in games of chance?**
  - **backgammon - the dice determines the legal move**



- **White knows what his/her legal moves are**
- **But White doesn't know what Black will roll and, thus, what Black's legal moves will be**
- **=> White cannot construct a standard game tree**

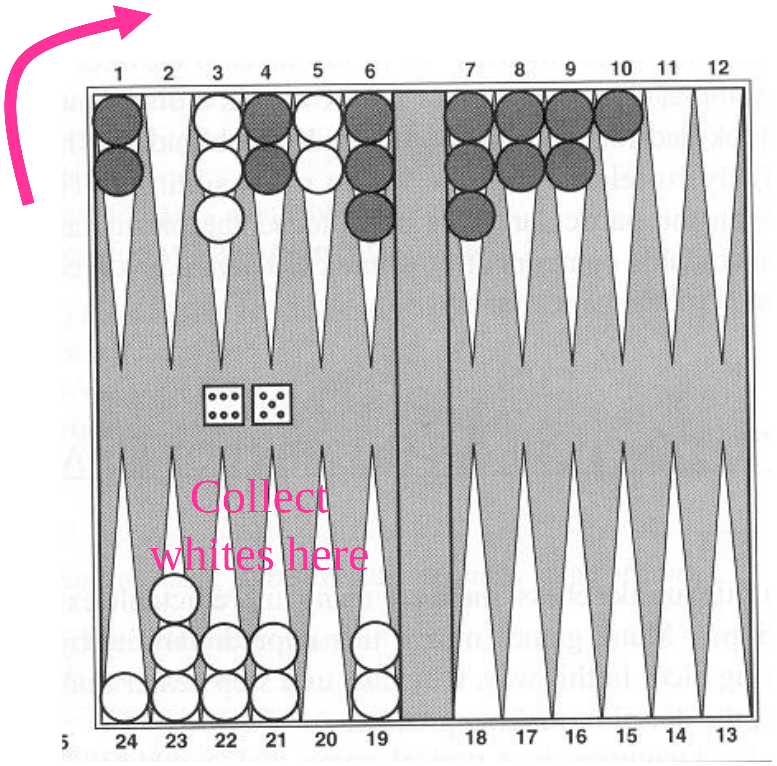
Game Tree for Non-Deterministic Games

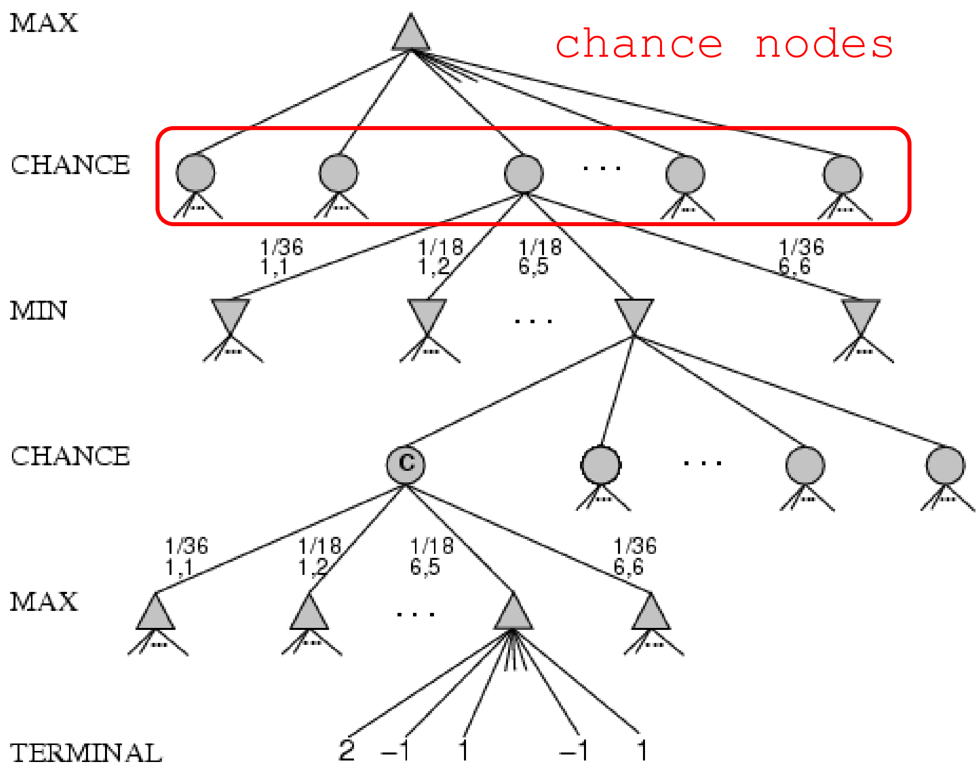




- The game tree includes chance nodes, corresponding to the dice values
- Because of the chance nodes, we cannot calculate definite minimax values, only expected value over all dice rolls

A Closer Look at the Chance Nodes – Determining the Probabilities





- **Dice is 6,5; possible moves for white:**
  (5-10, 5-11), (5-11, 19-24),(5-10, 10-16) & (5-11, 11-16)
- **When rolling 2 dice, how many possible outcomes are there? Are they equally likely or some are more likely?**
- **How many distinct ways are there? Probability for each of them?**
  - **[1,1]- [6,6] chance 1/36, all other chance 2/36=1/18**

Expectiminimax Algorithm for Non-Deterministic Games

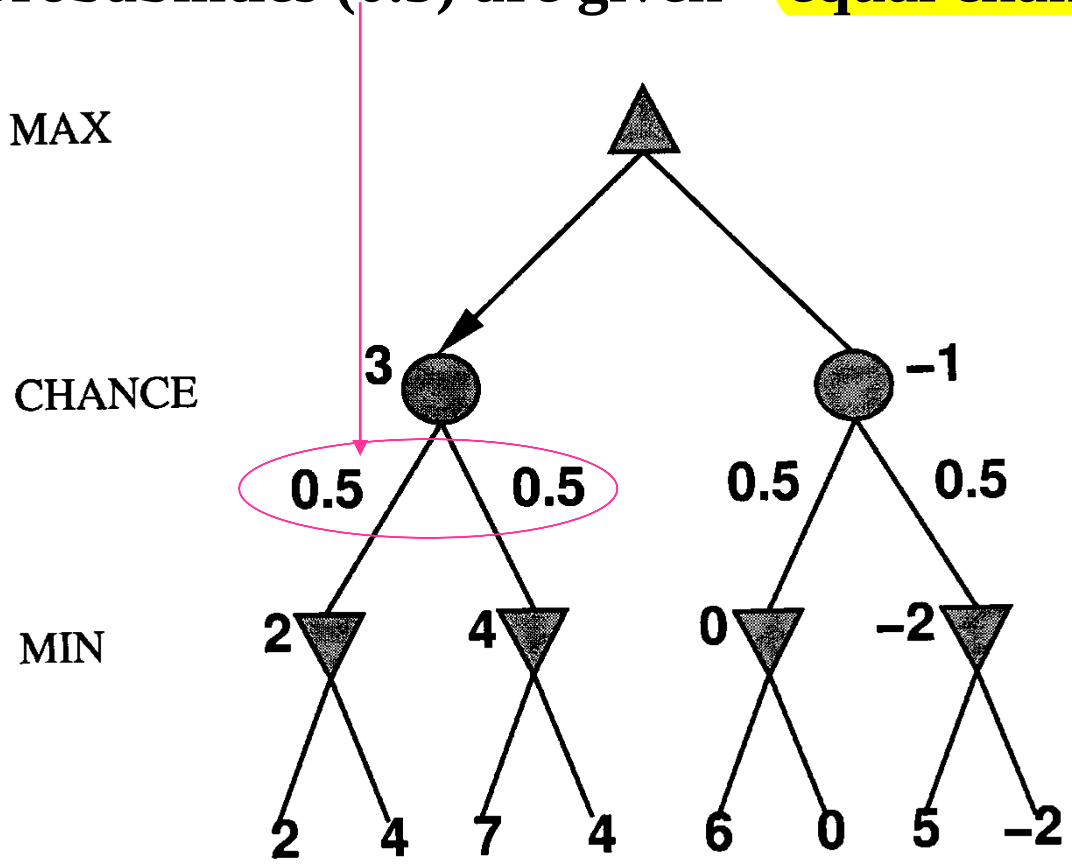
- An an extension of minimax for non-deterministic games
- Gives the perfect play for non-deterministic games
- Just like MINIMAX, except we must also handle chance nodes
  - At terminal state nodes - utility function
  - At MIN nodes – minimum of the children node values
  - At MAX nodes – maximum of the children node values
  - At CHANCE nodes - weighted average of EXPECTIMINIMAX values resulting from all possible dice rolls
- A version of alpha-beta pruning is possible but only if the leaf values are bounded

minimax

- Time complexity: $O(\overset{\text{minimax}}{b^m} n^m)$, n being the number of distinct dice rolls
  - For backgammon: b=20, n=21 => 3 plies max

Non-Deterministic Games – Search Tree

- **Simplified example with coin-flipping instead of dice-rolling**
- **The probabilities (0.5) are given – equal chance for heads and tails**

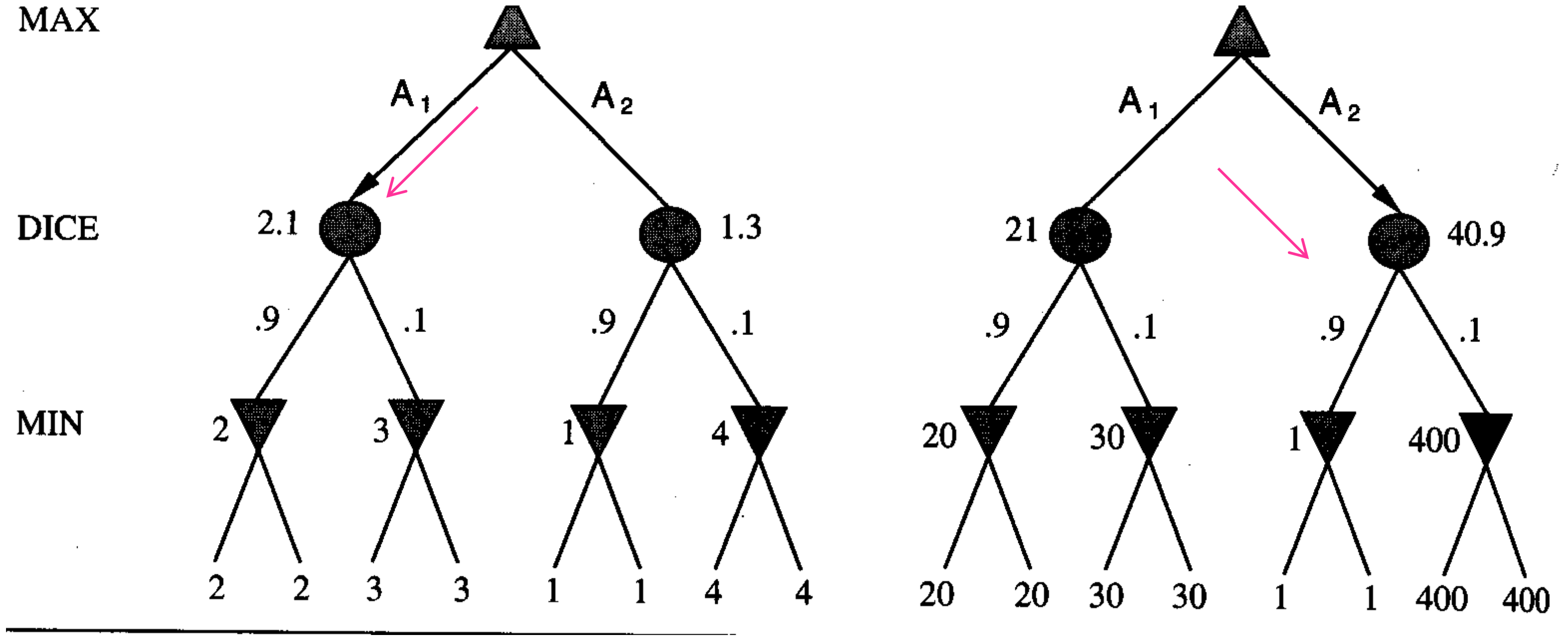


Expectiminimax evaluates chance nodes by taking the average utility of all children, weighted by the probability of each child

3=2*0.5+4*0.5

Exact Values Do Matter

- Evaluation functions for games with chance should be carefully constructed



- It is possible to avoid this – the evaluation function needs to satisfy a certain condition
  - be a positive linear transformation of the probability of winning from a position

Learning Evaluation Functions

- TD-Gammon (Tesauro 1992-1995) learns evaluation functions for backgammon from examples
  - Learner (classifier): neural network trained with the backpropagation algorithm
    - Maps input to output, and this mapping is learned from previous examples
    - Once trained, predicts the output for a new input (i.e. predicts the evaluation function for a new position)
  - Input: backgammon position
    - 198 dimensional input vector
  - Output: value of the position
    - 4 dimensional vector p
    - values are then combined, e.g. $v=p_1+2p_2-p_3-2p_4$
  - Training during actual play
  - Near championship level

Why are Games Fun?

- **Because they challenge our *ability to think***
  - **Even simple games like Tic-Tac-Toe or puzzles like 8-puzzle are challenging to children**
  - **More complex games like checkers, chess, bridge, and Go can require years of gifted adults to master them**
  - **Nearly all games require *recognizing patterns, making plans, searching combinations, judging alternative moves, and learning from experience* – skills which are also involved in many daily tasks**
- **It's no surprise that Shannon and Turing proposed playing chess as a good task for studying computers' ability to reason**
- ***Games have provided simple proving grounds for many powerful ideas in AI***

Games – State of the Art

Chess

- 1997: IBM's Deep Blue program defeated Kasparov in a 6-game match (2 wins for Deep Blue, 1 win for Kasparov, 3 draws)
- Deep Blue ran on a parallel computer and used alpha-beta search; searched 200 million positions per second, used very sophisticated evaluation function and undisclosed methods for extending some lines of search of up to 40 ply
    - Evaluation function with 8000 features
    - Opening book of 4000 positions
    - Database of 700 000 grandmaster games from which consensus recommendations were extracted
    - Large endgame database of solved positions containing all positions with 5 and many with 6 pieces; had the effect of extending the search depth

Image from http://www.thechessdrum.net/blog/2008/02/16/kasparov/

Read more: http://blogs.gartner.com/andrew_white/2014/03/12/the-chess-master-and-the-machine-the-truth-behind-kasparov-versus-deep-blue

Chess (2)

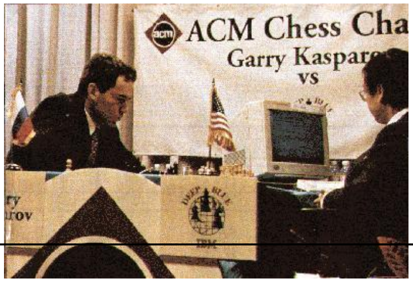
- Feb. 2003: Gary Kasparov vs. Deep Junior: draw
- Nov. 2003: Kasparov vs. X3D Fritz: draw (http://www.x3dchess.com/)
- Current computer players: Houdini and HIARCIS – can beat the top players
  - Interesting fact: Chess playing programs are so good that cheating has become an issue – getting help from computers during breaks

- Gary Kasparov is one of the greatest ever chess players
- It is a bit sad that he was the world champion when the AI algorithms and computer power reached the point when computers were able to beat humans!

http://ai.stanford.edu/~latombe/cs121/2011/slides/L-adversarial.ppt

http://ai.stanford.edu/~latombe/cs121/2007/slides/L-adversarial.ppt

Chess: Kasparov vs Deep Blue



Kasparov		Deep Blue
1.78 m	Height	1.96 m
80 kg	Weight	1089 kg
34 years	Age	4 years
50 billion neurons	Computers	32 RISC processors + 256 VLSI chess engines
2 pos/sec	Speed	200,000,000 pos/sec
Extensive	Knowledge	Primitive
Electrical/chemical	Power Source	Electrical
Enormous	Ego	None

1997: 6 games: 2 wins for Deep Blue, 1 win for Kasparov and 3 draws

Kasparov vs Deep Blue: a Contrast in Styles

- Top 10 dissimilarities (http://www.research.ibm.com/deepblue)

1. Deep Blue can examine and evaluate up to 200,000,000 chess positions per second; Garry Kasparov: up to 3.
2. Deep Blue has a small amount of chess knowledge and an enormous amount of calculation ability.
   Garry Kasparov has a large amount of chess knowledge and a much smaller amount of calculation ability.
3. Garry Kasparov uses his tremendous sense of feeling and intuition to play world champion-caliber chess.
   Deep Blue is a machine that is incapable of feeling or intuition.
4. Deep Blue has benefited from the guidance of 5 IBM researchers and 1 international grandmaster.
   Garry Kasparov is guided by his coach Yuri Dokhoian and by his own driving passion to play the finest chess in the world.

Kasparov vs Deep Blue: a Contrast in Styles

5. **Garry Kasparov is able to learn and adapt very quickly from his own successes and mistakes.**
   Deep Blue, as it stands today, is not a "learning system." It is therefore not capable of utilizing AI to either learn from its opponent or "think" about the current position of the chessboard.

6. **Deep Blue can never forget, be distracted or feel intimidated by external forces (such as Kasparov's infamous "stare").**
   Garry Kasparov is an intense competitor, but he is still susceptible to human frailties such as fatigue, boredom and loss of concentration.

7. **Deep Blue is stunningly effective at solving chess problems, but it is less "intelligent" than even the stupidest human.**
   Garry Kasparov is highly intelligent. He has authored three books, speaks several languages, is active politically and is a regular guest speaker at international conferences.

Kasparov vs Deep Blue: a Contrast in Styles

8. **Any changes in the way Deep Blue plays chess must be performed by the members of the development team between games.**
   Garry Kasparov can alter the way he plays at any time before, during, and/or after each game.
9. **Garry Kasparov is skilled at evaluating his opponent, sensing their weaknesses, then taking advantage of those weaknesses.**
   While Deep Blue is quite adept at evaluating chess positions, it cannot evaluate its opponent's weaknesses.
10. **Garry Kasparov is able to determine his next move by selectively searching through the possible positions.**
    Deep Blue must conduct a very thorough search into the possible positions to determine the optimal move (which isn't so bad when you can search up to 200 million positions per second).

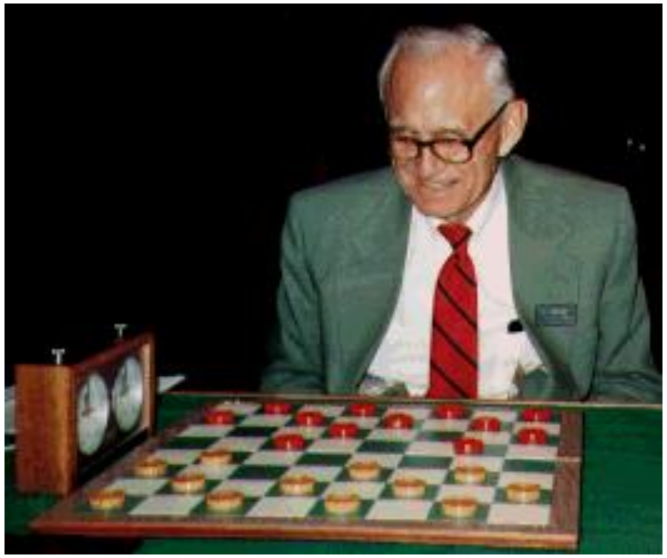
http://ai.stanford.edu/~latombe/cs121/2007/slides/L-adversarial.ppt

Checkers: Tinsley vs Chinook

- **1994: Chinook (developed by Jonathan Shaeffer) ended the 40-year-reign of human world champion Marion Tinsley**
- **Uses alpha-beta search + very large endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.**

Read more: http://www.cs.ualberta.ca/~chinook/
http://webdocs.cs.ualberta.ca/~jonathan/
http://science.sciencemag.org/content/317/5836/308.1.full

Checkers: Tinsley vs. Chinook (2)



Name:	**Marion Tinsley**
Profession:	**Mathematics teacher**
Hobby:	**Checkers**
Record:	**Over 42 years – lost only 3 games of checkers**

World champion for over 40 years

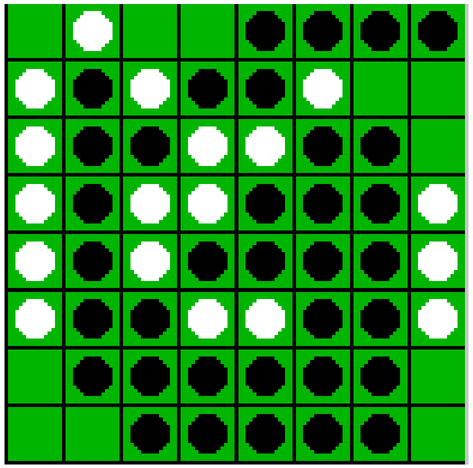
- **1st match (1990) - Tinsley lost some games but won the match**
- **2nd match (1994) – 6 draws, then Tinsley retired for medical reasons, and Chinook was declared a winner**

- **2007 – Chinook used a computer network to explore 500 billion billion possible positions and it was proven mathematically that it could play perfectly and never lose => no doubt about the superiority of computers in checkers!**

Othello

- **Othello is more popular as a computer game than as a board game**
- **It has a much *smaller search space than chess*; 5-15 legal moves**
- ***=> Computers are too good* - human champions refuse to compete against computers as they are too good!**

Murakami vs. Logistello



Takeshi Murakami
World Othello Champion

1997: The Logistello program won vs Murakami by 6 games to 0

State of the Art - Go

- ~~**Board 19 x 19 = > the branching factor is too big (b>300) for existing search techniques => computers don't perform well**~~
- ~~***Humans are too good*** **- human champions refuse to compete against computers as they are too bad!**~~
- ~~**Best existing program: Gomate – can be easily beaten by humans**~~

Not true any more!

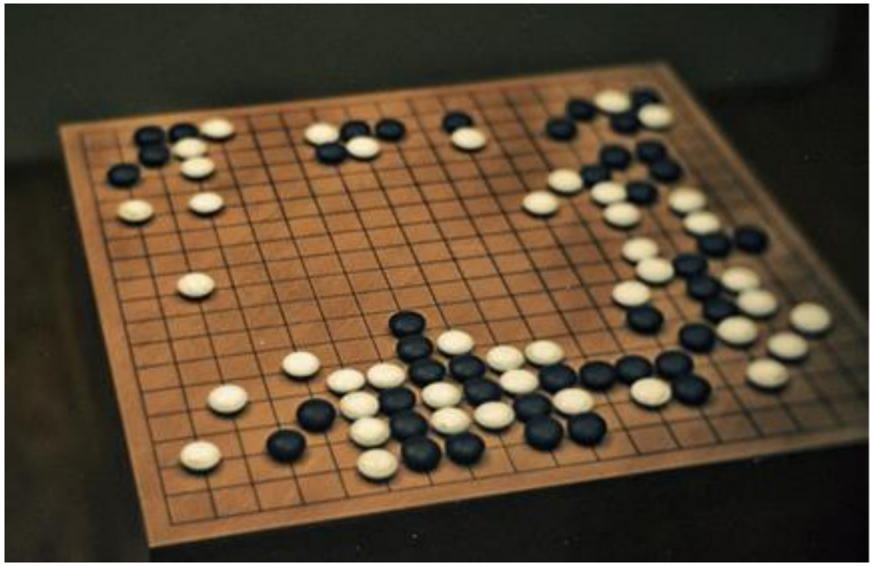


Image from Linh Nguyen/Flickr, CC BY-NC-ND

- **Ancient Chinese game, played by 40 million people**
- **Simple rules, big complexity**
- **2 players; 19x19 board; black and white stones**
- **Goal: capture the opponent's stones or surround empty space to make points of territory**

Number of States in Go

- **1,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,00
0,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000**

- **This is more than the number of atoms in the universe!**

- **Tree search algorithms such as minimax and alpha-beta pruning don’t have a chance**

AlphaGo

- A system developed by Google
- Uses a classifier - a convolutional neural network (deep learning neural network) to predict the next move to play
- The neural network is trained using data from previous games played by human experts (30 million moves) – accuracy = 57%
- AlphaGo can also discover new strategies by playing the game alone
- It uses not the traditional search tree algorithms but a Monte Carlo search (https://en.wikipedia.org/wiki/Monte_Carlo_tree_search) - only some random parts of the tree are expanded till the end and then evaluated
- It is not the effort of 1 person + 1 computer but a team of Google's engineers + a lot of computing power (Google's server farms)

AlphaGo vs Lee Sedol

- **2016: AlphaGo defeated Lee Sedol (4:1 games)**
- **https://en.wikipedia.org/wiki/AlphaGo_versus_Lee_Sedol**
- **Lee Sedol is South Korean professional Go player, was ranked 2nd in the world in 2016**
- **The match includes 5 games**
  - **AlphaGo won the first 3 games and 5th game**
  - **Lee Sedol won the 4th game – he played an “unexpected” move that confused AlphaGo**
- **End end of the human dominance in Go**
- **A lot of publicity for AI and deep learning:**

AI has beaten us at Go. So what next for humanity? The Conversation, 2016

To Beat Go Champion, Google's Program Needed a Human Army (New York Times, 2016)

State of the Art – Bridge and Backgammon

- **Bridge:** **Expert-level computer players exist but no world champions yet!**
- **Poker (a simpler version):** **computer better than human**
  - **http://www.sciencemag.org/news/2015/01/texas-hold-em-poker-solved-computer**
- **Backgammon:** **Tesauro combined Samuel's learning method with neural networks in 1992 => program ranked among the best 3 players in the world**

http://ai.stanford.edu/~latombe/cs121/2007/slides/L-adversarial.ppt

Secrets

- **Many game programs are based on alpha-beta + iterative deepening + extended + transposition tables (store evaluation of repeated positions) + huge databases + ...**
- **For instance, Chinook searched all checkers configurations with 8 pieces or less and created an endgame database of 444 billion board configurations**
- **Other methods (Go): deep learning (huge training set of previous games) + Monte Carlo search**
- **The methods are general, but their implementation is dramatically improved by many specifically tuned-up enhancements (e.g. the evaluation functions)**

Perspectives on Games

Saying Deep Blue doesn't really think about chess is like saying an airplane doesn't really fly because it doesn't flap its wings.

Drew McDermott

Artificial Intelligence is creating the illusion of intelligence. Does a simple program searching thirteen plies ahead play a strong game of checkers? Yes. Are people impressed with this? Yes. Therefore a checkers program like Chinook is artificially intelligent. The techniques used are irrelevant. Again, just because a bird flies by flapping its wings doesn't mean that we should build airplanes that use the same approach"

Jonathan Shaeffer, 1997

- **What do you think? Are Deep Blue, Chinook and AlphaGo intelligent?**
- **Does their success tell us anything about the nature of human intelligence?**

Summary

- In 2-player, 0-sum games, with perfect information:
  - minimax can select the optimal move using depth first traversal of the game tree
  - alpha-beta does the same but it is typically more efficient as it prunes unnecessary sub-trees
- It is not visible to consider the whole game tree, so we need to cut the search off and apply evaluation function which estimates the goodness of a state
- Games by chance can be handled by extended minimax (called expectiminimax) that evaluates chance nodes by taking the average utility of all children, weighted by the probability of each child

Summary (2)

- **Games are fun to work on!**
- **They illustrate several important points about AI**
- **Perfection is unattainable => must approximate**

- ***They are the "mice labs" for AI***
  - **Games are highly specialized tasks but a lot of cutting-edge AI concepts and ideas came out of them!**