Adversarial Search

Chapter 5

Outline

- Optimal decisions in games
  - **Which strategy leads to success?**
- Perfect play
  - minimax decisions
- α-β pruning
- Resource limits and approximation evaluation
- Games of imperfect information
- Games that include an element of chance

Games

- **Games are a form of *multi-agent environment***
  - What do other agents do and how do they affect our success?
  - Cooperative vs. competitive multi-agent environments.
  - Competitive multi-agent environments give rise to adversarial problems a.k.a. *games*
- **Why study games?**
  - Interesting subject of study because they are hard
  - Easy to represent and agents restricted to small number of actions

Games vs. Search problems

- **Search – no adversary**
  - Solution is (heuristic) method for finding goal
  - Heuristics and CSP techniques (ch.6) can find *optimal* solution
  - **Evaluation function**: estimate of cost from start to goal through given node
  - Examples: path planning, scheduling activities
- **Games – adversary**
  - Unpredictable" opponent solution ⇒ solution is a strategy (contingency plan)
  - Time limits → unlikely to find goal, must approximate **plan of attack**
  - **Evaluation function**: evaluate "goodness" of game position

Games Vs. search problems

- Iterative methods apply here since search space is too large, thus search will be done before each move in order to select best move to be made.

- Adversarial search algorithms: designed to return optimal paths, or winning strategies, through game trees, assuming that the players are adversaries (rational and self-interested): they play to win.

- Evaluation function(static evaluation function) : Unlike in heuristic search where the evaluation function was a non-negative estimate of the cost from the start node to a goal and passing through the given node, here the evaluation function, can be positive for a winning or negative for losing.

Games search

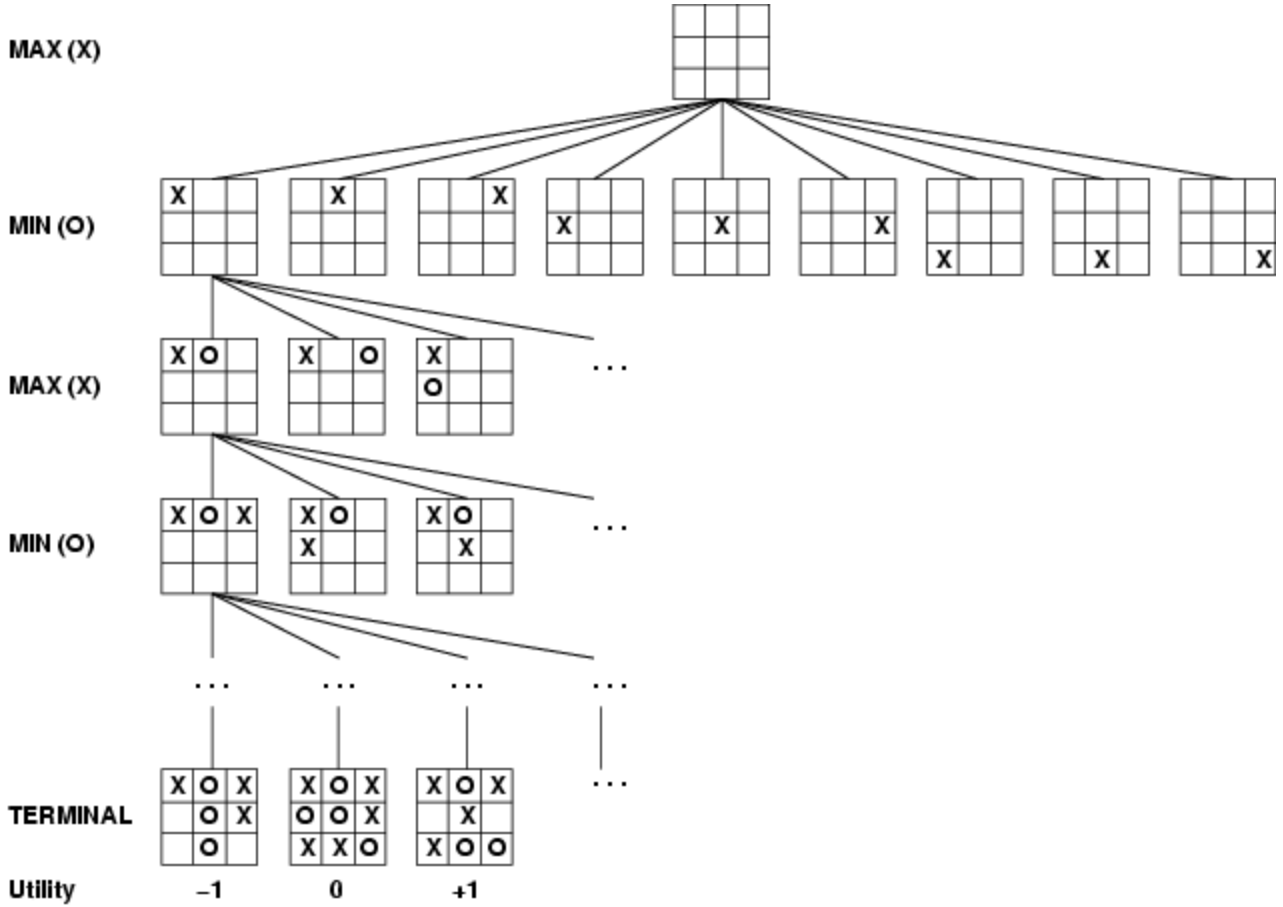
- aka "adversarial search": 2+ opponents working against each other
- game tree: a tree in which nodes denote board configurations, and branches are board transitions
  - a state-based tree: configuration = state
- Most 2-player game require players taking turns
  - then levels of tree denote moves by one player, followed by the other, and so on
  - each transition is therefore a move
- Ply: total number of levels in tree, including the root
  - ply = tree depth + 1
- Note: most nontrivial game situations do not permit:
  - exhaustive search --> trees are too large
  - pure goal reduction --> situations defy simple decomposition
    - i.e., difficult to convert board position into simple sequence of steps
- thus search + heuristics needed

Game Playing

	deterministic	chance
perfect information	chess, checkers go, othello	backgammon monopoly
imperfect information	battle ships Blind tictactoe	bridge, poker, scrabble nuclear war

- consider a 2-player, zero-sum, perfect information (i.e., both players have access to complete information about the state of the game) in which the player moves are sequential.

Partial Game tree for Tic-Tac-Toe (2-player, deterministic, turns)



Game setup

- Two players: MAX and MIN
- MAX moves first and they take turns until the game is over. Winner gets award, looser gets penalty.
- Games as search:
  - Initial state: e.g. board configuration of chess
  - Successor function: list of (move,state) pairs specifying legal moves.
  - Terminal test: Is the game finished?
  - Utility function (a.k.a **payoff function**): : Gives numerical value of terminal states. E.g. win (+1), loose (-1) and draw (0) in tic-tac-toe
- MAX uses search tree to determine next move.

Minimax procedure

- Game playing involves competition:
  - two players are working towards opposing goals
  - thus the search tree differs from previous examples in that transitions representing game turns are done towards opposite goals
    - there isn't one search for a single goal!
- static evaluation: a numeric value that represents board quality
  - done by a static evaluator
  - basically a heuristic score (as used in informed search)
- utility function: maps an end-game state to a score
  - essentially same as the static evaluator

Minimax procedure

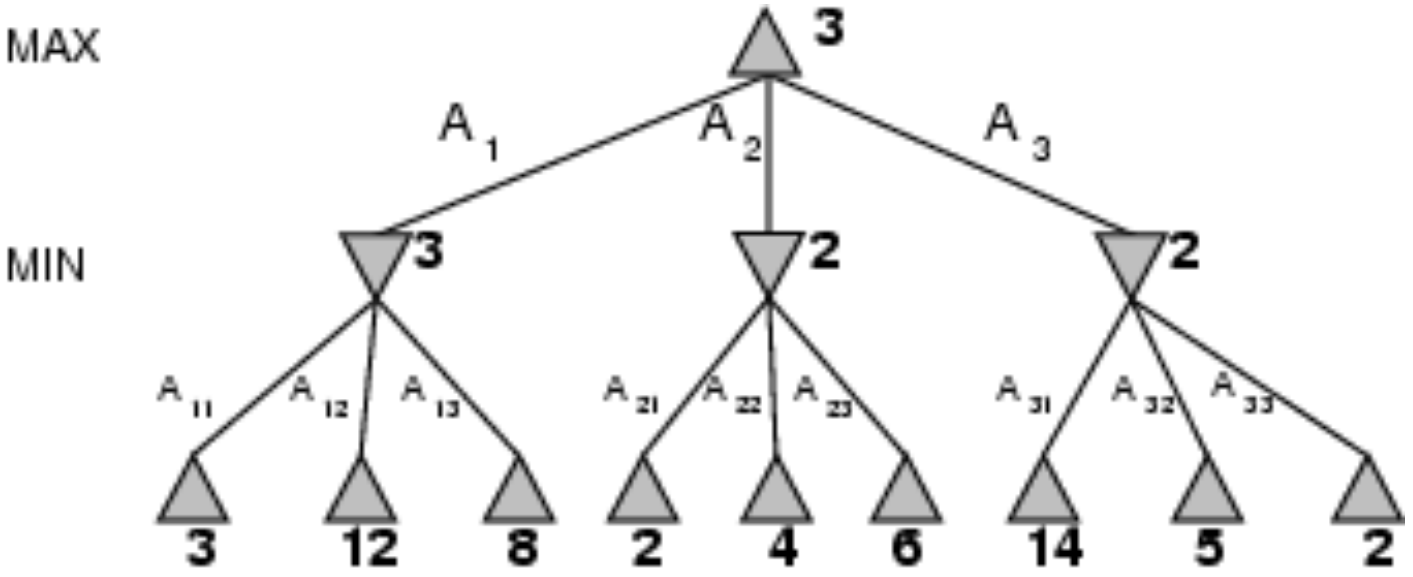
- Maximizer: player hoping for high/positive static evaluation scores
- Minimizer: the other player wants low/negative values
- Thus the game tree consists of alternate maximizing and minimizing layers
  - each layer presumes that player desires the evaluation score most advantageous to them

Minimax

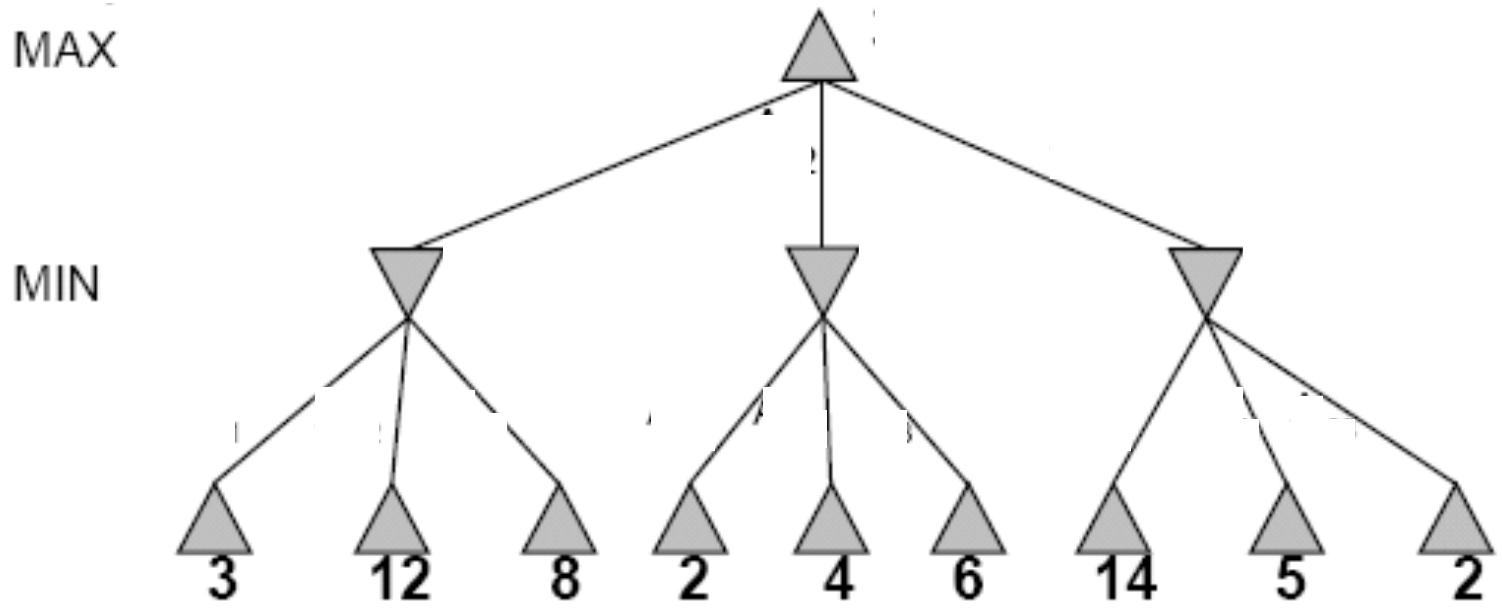
- **Presume that we - the computer entity - are always the MAX**
- **When examining a game tree , the MAX wants to obtain the highest static eval. score at each level**
  - **–but the MAX presume that the opponent is intelligent, and has access to the same evaluation scores**
  - **–hence must presume that opponent will try to prevent you from obtaining best score... and vice versa!**
- **Minimax procedure: a search strategy for game trees in which:**
  - **a) a finite search ply level p is used: tree expanded p deep**
  - **b) static evaluation done on all expanded leaf configurations**
  - **c) presumption that opponent will force you to make least desirable move for yourself, and best for herself/himself.**

Minimax

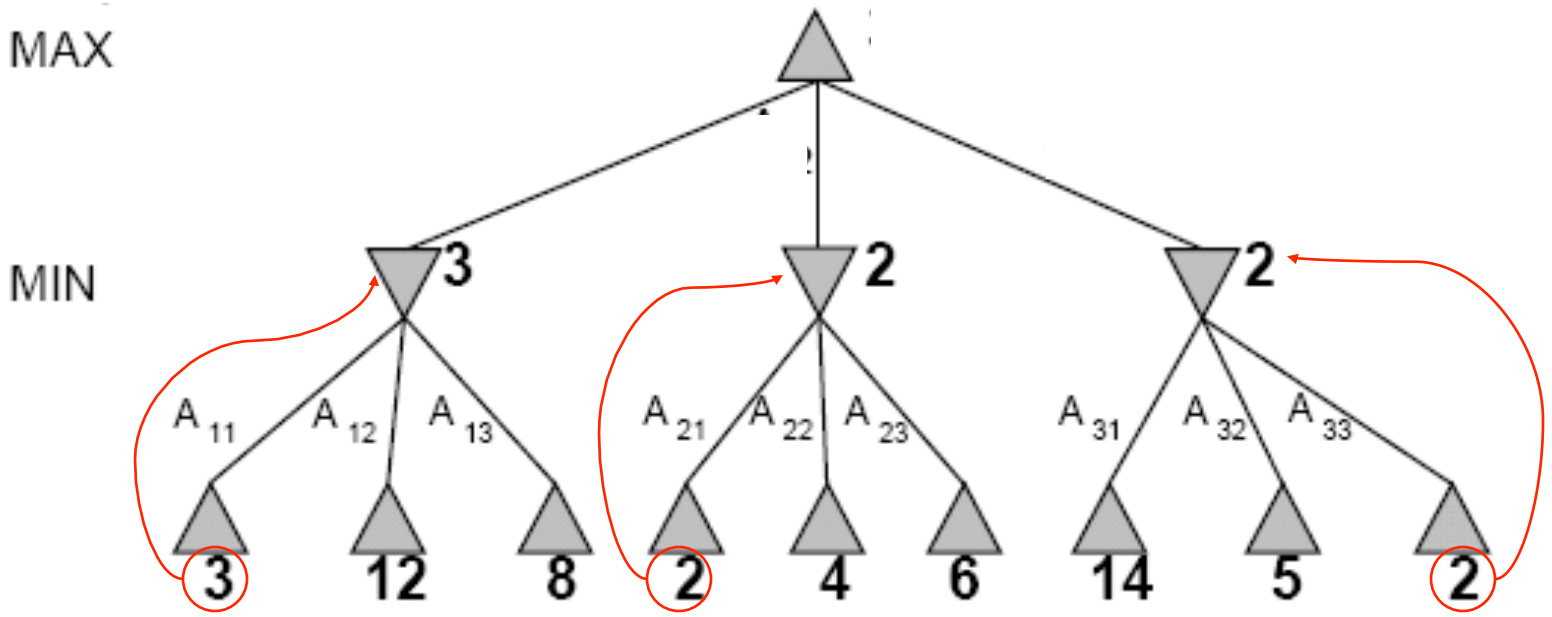
- Perfect play for deterministic games
- Idea: choose move to position with highest minimax value
  = best achievable payoff against best play
- E.g., 2-ply game:



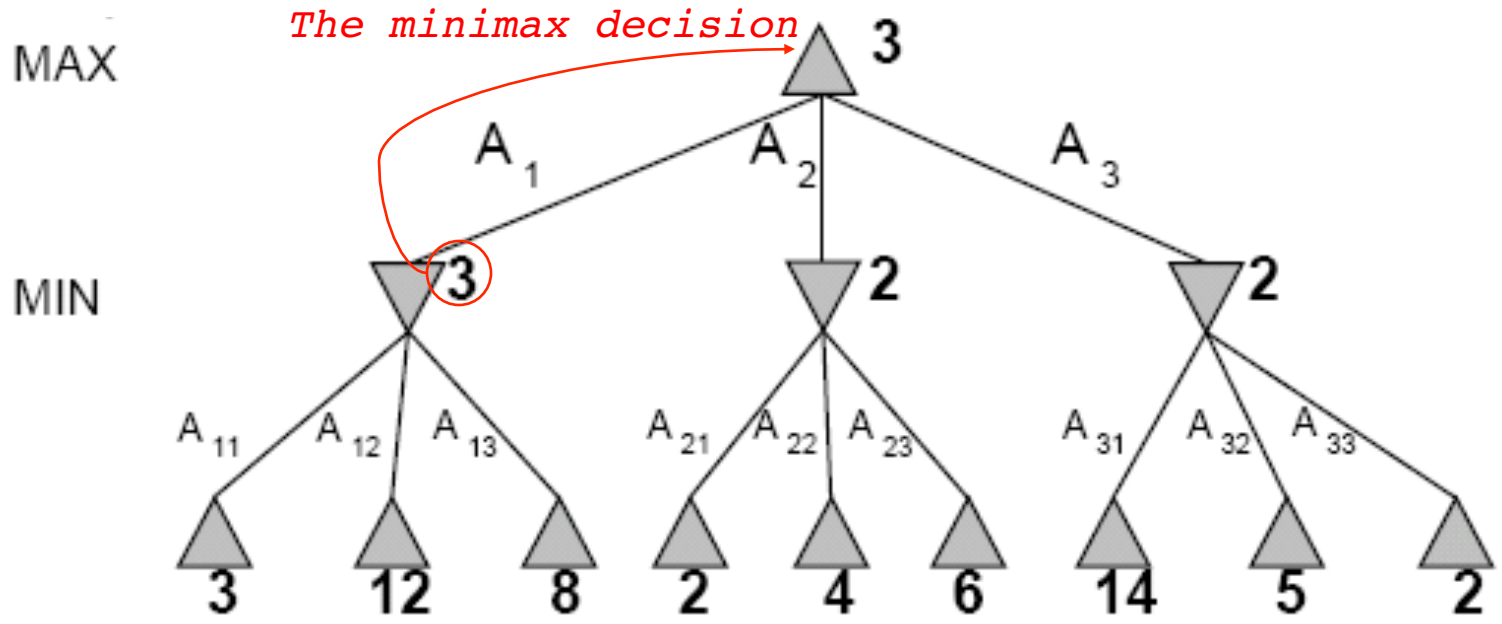
Minimax



Minimax



Minimax



Minimax maximizes the worst-case outcome for max.

Minimax

Steps used in picking the next move:

1. Create start node as a MAX node (since it's my turn to move) with current board configuration
2. Expand nodes down to some depth (i.e., ply) of lookahead in the game
3. Apply the evaluation function at each of the leaf nodes
4. "Back up" values for each of the non-leaf nodes until a value is computed for the root node. At MIN nodes, the backed up value is the minimum of the values associated with its children. At MAX nodes, the backed up value is the maximum of the values associated with its children.
5. Pick the operator associated with the child node whose backed up value determined the value a the root

What if MIN does not play optimally?

- Definition of optimal play for MAX assumes MIN plays optimally: maximizes worst-case outcome for MAX.
- But if MIN does not play optimally, MAX will do even better.

Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
    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
```

Properties of minimax

- Complete? Yes, if tree is finite (chess has specific rules for this)
- Optimal? Yes, against an optimal opponent. Otherwise??
- Time complexity? $O(b^m)$
- Space complexity? O(bm) (depth-first exploration)

- For chess, b ≈ 35, m ≈100 for "reasonable" games
  → exact solution completely infeasible

But do we need to explore every path?

Minimax

- **Strengths:**
  - –**presumption that opponent is at least as intelligent as you are**
  - –**ply parameter can be played with**
  - –**practical: search can continue while opponent is thinking**
- **Short-comings:**
  - –**single static evaluation score is descriptively poor**
    - **convenient for analytical and search purposes**
    - **but it's a "lossy" compression scheme - you lose lots of important information**
    - **about configuration (this applies to any single-value heuristic or state descriptor that compresses info)**
  - –**requires entire subtree of ply depth p to be generated**
    - **may be expensive, especially in computing moves and static evaluation scores**

Problem of minimax search

- Number of games states is exponential to the number of moves.
  - Solution: Do not examine every node
  - ==> Alpha-beta pruning
    - Alpha = value of best choice found so far at any choice point along the MAX path
    - Beta = value of best choice found so far at any choice point along the MIN path

Minimax

•In an implementation, expansion, evaluation and search are interwoven together.

– No point saving all expanded nodes either. Only highest tree level's next move must be saved, in order to do it.

– intermediate scores as found by minimax are returned in the routine.

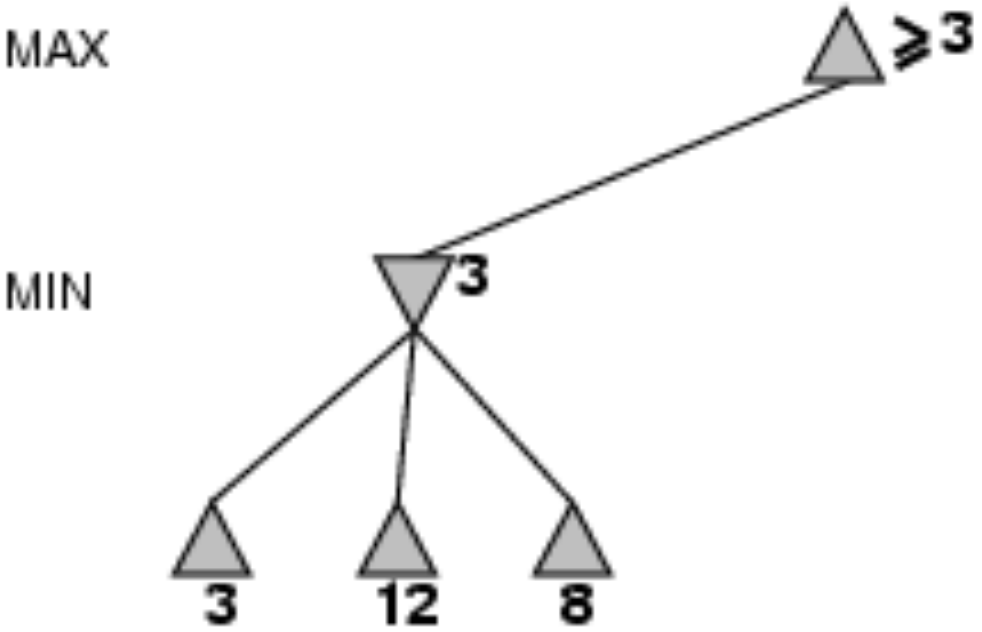
• Note that intermediate nodes are not evaluated in minimax!

- Decisions at higher levels of tree depend only on leaf evaluations in descendents.
- "Look ahead" logic!
- enhancements to minimax may evaluate intermediates under certain conditions (will discuss later)

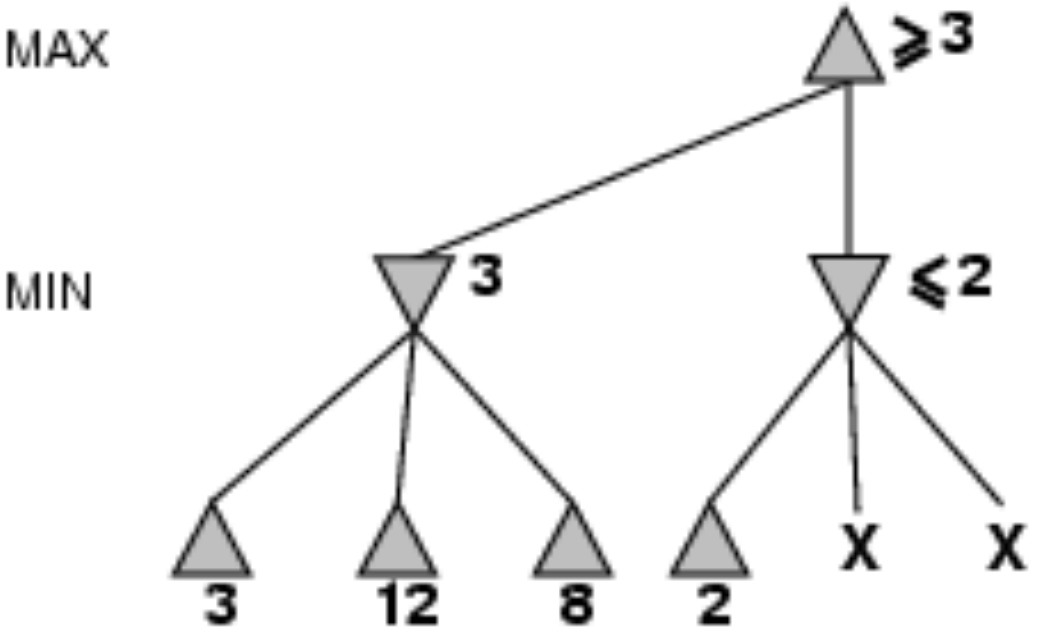
Pruning game trees: Alpha-Beta Procedure

- pruning: the deletion of unproductive searching in a search tree
  - normally involves ignoring whole branches
  - Wrt search strategies, a qualitative decision about suitability in searching a particular branch is made
  - Prolog: the “cut” operator is a pruning operator (but done syntactically - no decision making)
- Alpha-beta pruning procedure:
  - in concert with the minimax procedure, alpha-beta pruning prevents the unnecessary evaluation of whole branches of the search tree
  - decision to ignore a branch is based on knowing what your opponent will do if a clearly good move is available to him/her
- best case: alpha-beta cuts exponential growth rate by 1/2

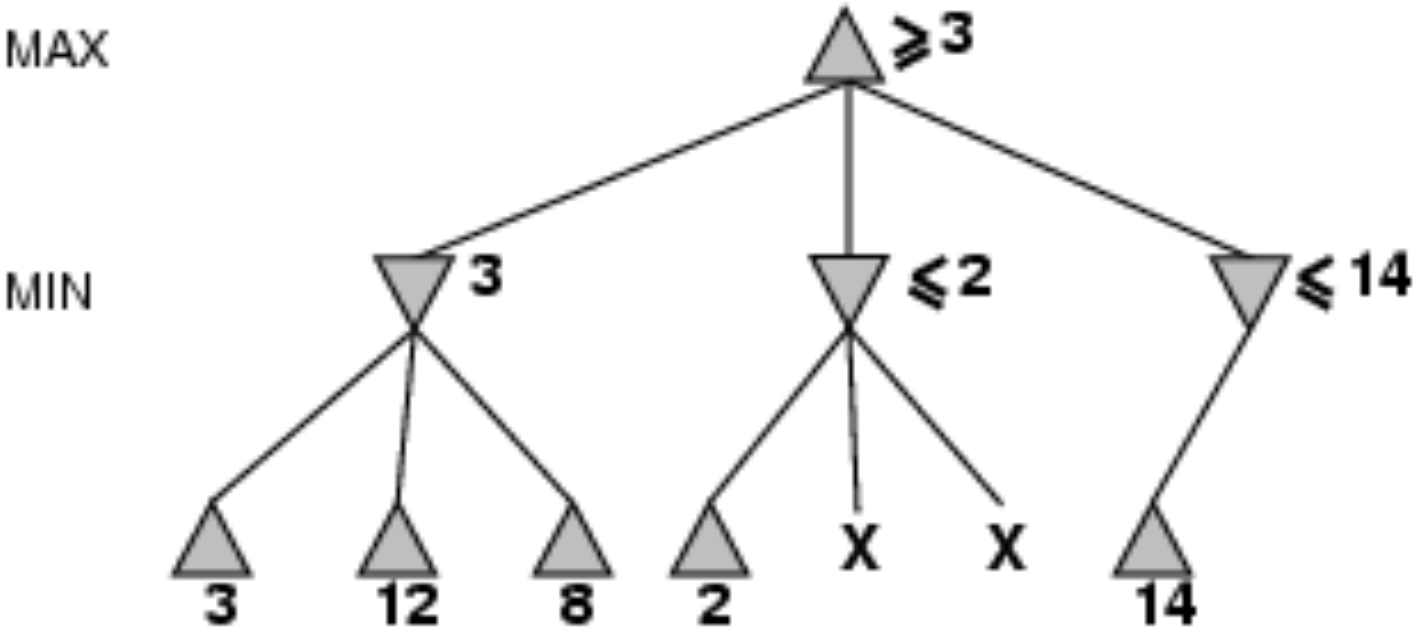
α-β pruning example



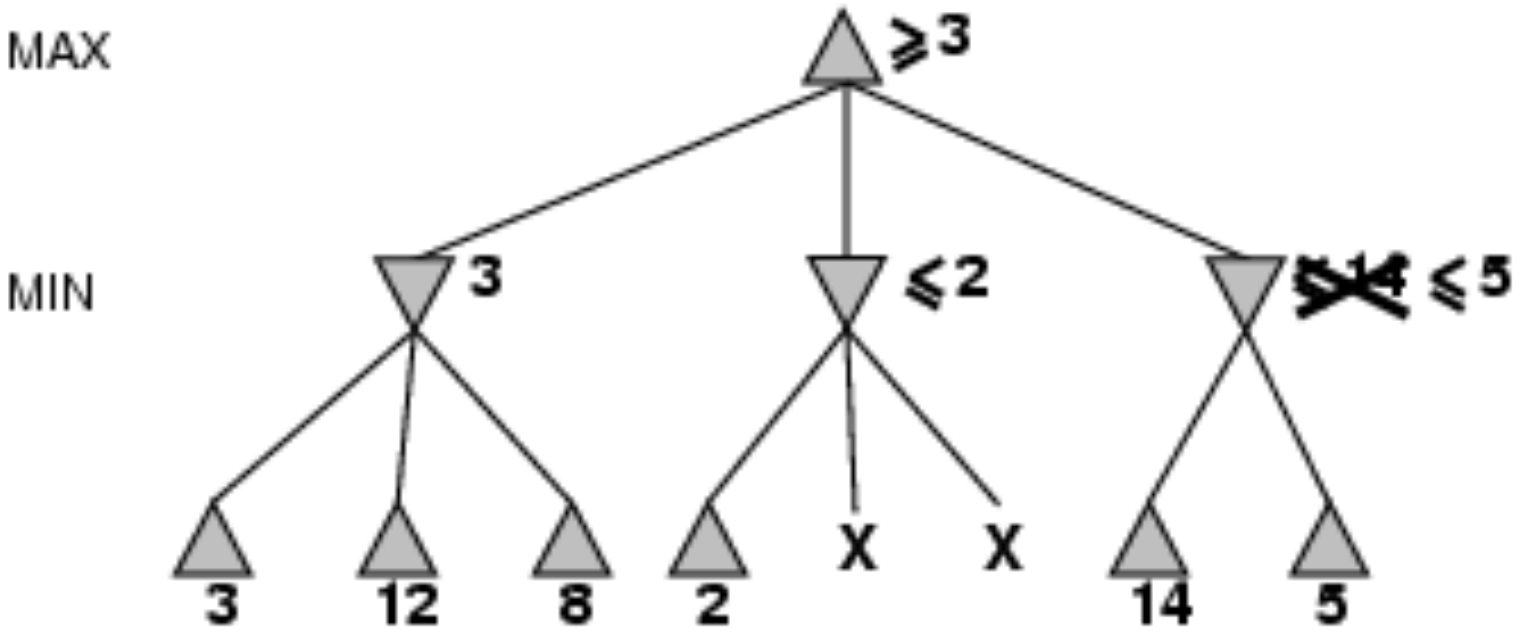
α-β pruning example



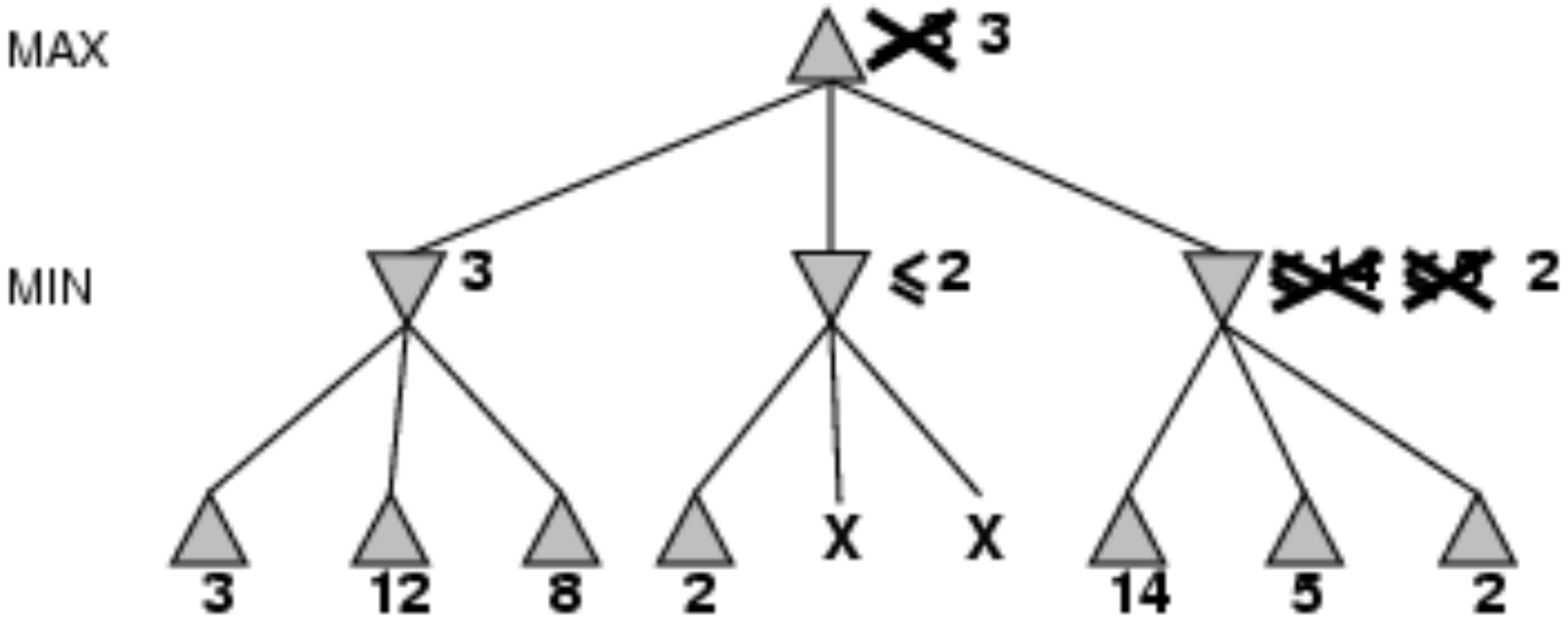
α-β pruning example



α-β pruning example



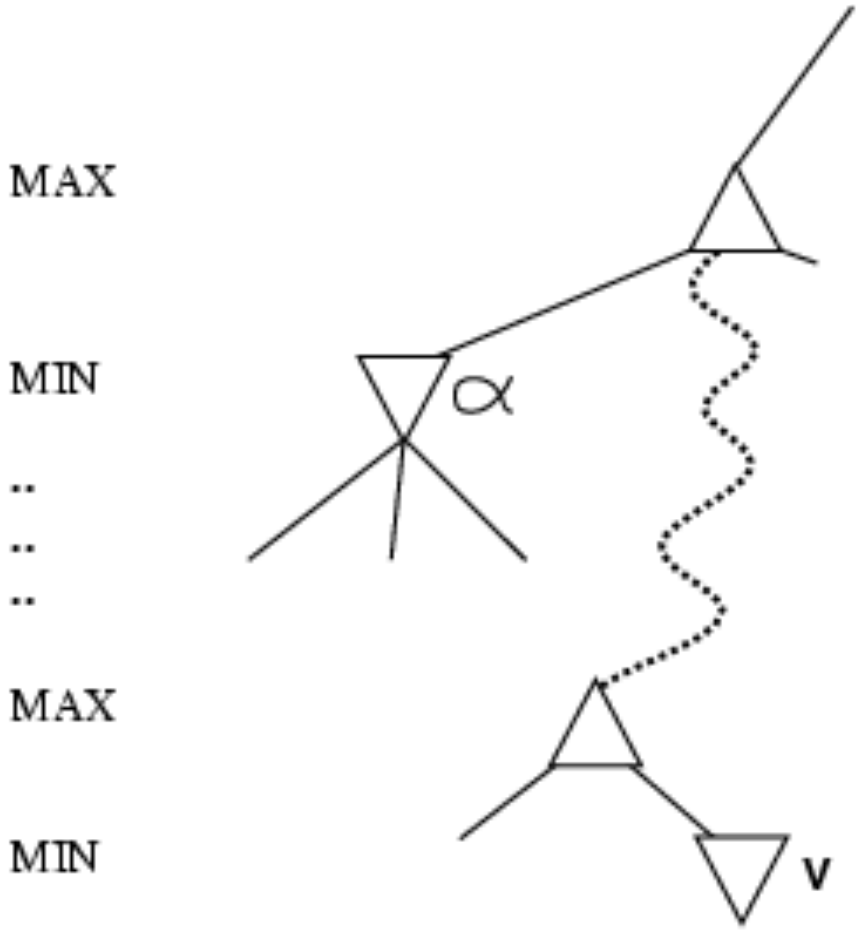
α-β pruning example



Why is it called α-β?

- α is the value of the best (i.e., highest-value) choice found so far at any choice point along the path for *max*
- If *v* is worse than α, *max* will avoid it

  → prune that branch
- Define β similarly for *min*



The α-β algorithm

```
function ALPHA-BETA-SEARCH(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, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state

    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s, α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

The α-β algorithm

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

    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s, α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

Properties of α-β

- Pruning does not affect final results
- Entire subtrees can be pruned.
- Good move *ordering* improves effectiveness of pruning
- With "**perfect ordering**," time complexity is $O(b^{m/2})$
  - Alpha-beta pruning can look twice as far as minimax in the same amount of time
  - => doubles solvable path
- A simple example of the value of reasoning about which computations are relevant(a form of metareasoning)
-

Alpha-Beta

- Advantages:
  - an efficient means to determine when particular branches are not worth further consideration
  - again, presumes opponent has same heuristic info as you
- Disadvantages:
  - not really any, other than it doesn't reduce exponentiality of trees

- It does seem paradoxical that alpha-beta can force the ignoring of a whole branch!
  - that branch might have your most brilliant strategic move... why on earth should we ignore it?!?!
  - reason: (again) we must presume the opponent is as intelligent as we are, and will (a) make moves that strengthen his/her position, and (b) make moves that weaken us
  - --> *a smart opponent will foil our brilliant moves!* (alpha-beta knows this)

Resource limits

- Minimax and alpha-beta pruning require too much leaf-node evaluations.
- May be impractical within a reasonable amount of time.
- Standard approach:
  - **Cut off search** earlier (replace **TERMINAL-TEST** by **CUTOFF-TEST**)
    e.g., depth limit
  - **Apply heuristic evaluation function** (use **EVAL** instead of **UTILITY**)
    =estimated desirability of position

Cutting off search

MinimaxCutoff is identical to *MinimaxValue* except

1. *Terminal?* is replaced by *Cutoff?*
2. *Utility* is replaced by *Eval*

- Introduces a fixed-depth limit *depth*
  - **Is selected so that the amount of time will not exceed what the rules of the game allow.**
- When cuttoff occurs, the evaluation is performed.

4-ply lookahead is a hopeless chess player!

- 4-ply ≈ human novice
- 8-ply ≈ typical PC, human master
- 12-ply ≈ Deep Blue, Kasparov

Heuristic EVAL

- Idea: produce an estimate of the expected utility of the game from a given position.
- Performance depends on quality of EVAL.
- Requirements:
  - EVAL should order terminal-nodes in the same way as UTILITY.
  - Computation may not take too long.
  - For non-terminal states the EVAL should be strongly correlated with the actual chance of winning.

Heuristics and game trees

- Game search programs are usually real-time and have time restrictions
  - –ply depth is critical: too small = weak analysis; too large = can't search properly (slow)
  - –difficult to know what to use, given dynamic nature of computations, hardware, etc
- iterative deepening (ch.3): analyze 1 level deep, then 2, then 3, until time expires
  - –but isn't this a waste? Why analyze p=k when p=k+1 will be tried next'
  - –good when tree size, speed, resource usage are unknown
  - –surprisingly, for a branching factor of b, all the analysis for all levels before leafs take only a fraction of the total time for level b (for large b)
  - – these earlier nodes should therefore be analyzed first... little waste of time!
  - - the larger of b, the greater the savings

Heuristics and game trees

- horizon effect: when using a finite, fixed search depth, you cannot see any deeper
  - decisions are necessarily based on limited information
  - an effect of this is that decisions can be unduly influenced by what look like outstanding moves... but if the search progressed a little further, these moves might be bad after all
  - another effect: delaying the inevitable (move bad situations beyond the “horizon”)

- singular-extension heuristic: if one move appears considerably better than others, search its subtree a little deeper
  - this tries to discount superficially good moves that are really bad!

Heuristics and game trees

- search-until-quiescent heuristic: don't let captures (or key game configuration changes) unduly influence choices
  - similar to singular-extension idea
- tapered search: vary depth factor among nodes
  - can vary the depth factor with heuristic ranking of children, ie. better configurations are expanded more
- can also use rule-based decisions in controlling search
  - high-level database of domain knowledge ("knowledge base") can influence search decisions

Evaluation functions

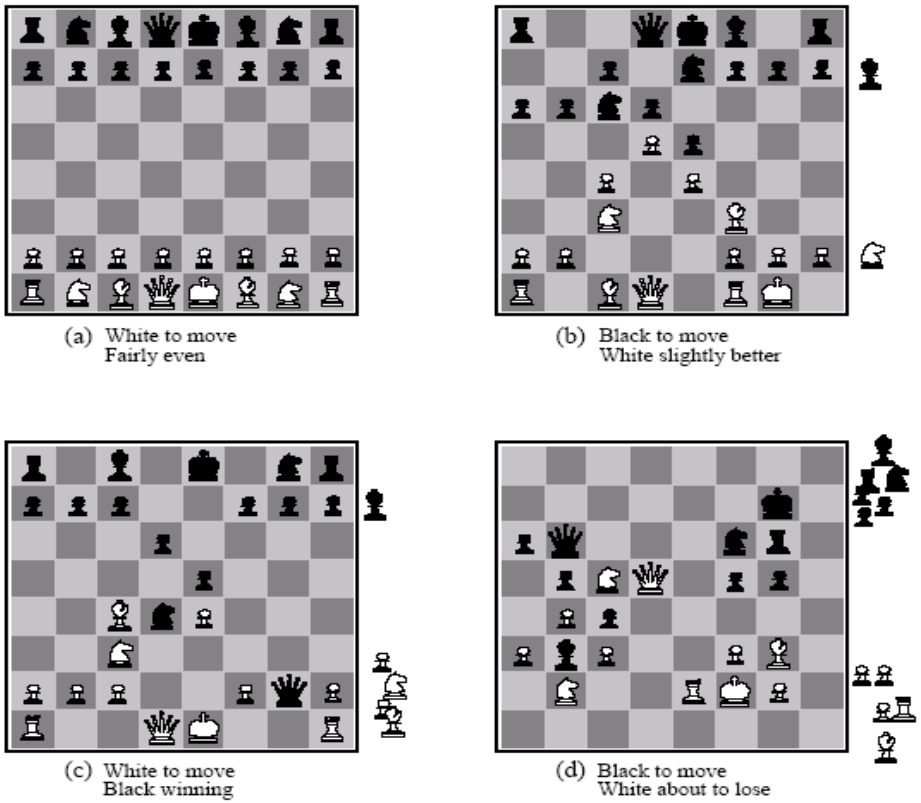
- For chess, typically linear weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- e.g.,

  $f_1(s)$ = (number of white queens) – (number of black queens), etc.

Heuristic EVAL example



(a) White to move
Fairly even

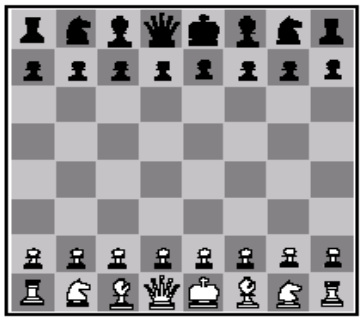
(b) Black to move
White slightly better

(c) White to move
Black winning

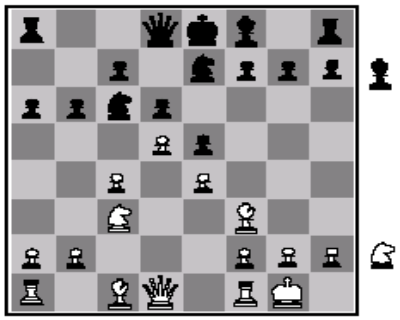
(d) Black to move
White about to lose

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

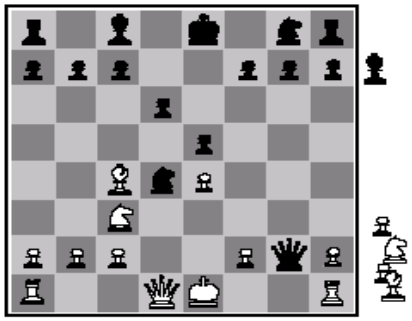
Heuristic EVAL example



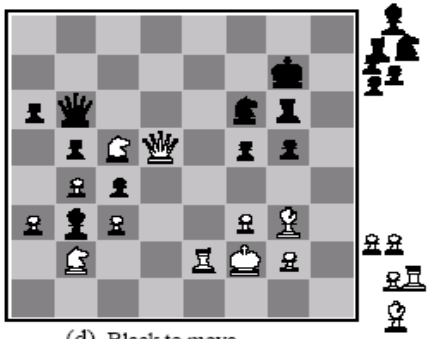
(a) White to move
Fairly even



(b) Black to move
White slightly better



(c) White to move
Black winning



(d) Black to move
White about to lose

Addition assumes independence

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

Games with chance

- **unpredictability adds nondeterminism to tree transitions: unlike (e.g) chess, next move is not entirely determined by either player**
  - **–e.g. any game with dice; some with cards**
- **chance requires chance nodes in tree**
  - **–these nodes denote nondeterministic outcomes**
  - **–for dice, they would have dice totals**
  - **–can use the average outcome of the chance node children in order to determine an overall result (which is needed for minimax)**
  - **–can also use probabilities for dice outcomes**
- **expectiminimax: minimax with chance**
  - **–search complexity of chance games is considerably higher; ply is limited to approx. 2**

Games with chance, or imperfect information

State of art in computer gaming

- Chess: the traditional AI domain for game playing
  - have finally beat world champions in tournaments, (esp. at speed playing)
  - computers ranked in the top 100 in tournament level
  - 1970's: used search tricks; alpha-beta pruning
  - 1982: Belle chess computer - special circuits for move generation and evaluation
    - rated 2250 (beginners = 1000, world champion = 2750)
  - 1987: Hitech system beat world champion
    - computes 10 million positions per move

  (90s, next page)

State of art in computer gaming

- **Chess:** Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997.
  - **evaluates 200 million positions / sec (100-200 bill/ move)**
  - **uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.**
  - **Uses very sophisticated evaluation**
  - **Lines of search up to 40 ply**

Checkers

- Other games: Checkers
  - 1952: first checkers computer player by Arthur Samuel
    - learned it's own evaluation function by playing itself!
    - system only had 10K mem, 1 millisecond CPU!
- Chinook program is the world champion (as of 1994)
  - Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a pre-computed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.

Othello & Backgammon

- Othello better than humans (human experts decline to play the computers in tournaments)

- Backgammon: (1992) a Samuel-like program with neural nets is ranked in the top 3

Go

- **Popular game in east Asia**
  - Japan “Go”, Korea “Baduk”, China “Wei chi”
- **defies minimax search approaches:**
  - 1. avg branching factor is b > 300 (chess = 25)
  - 2. cannot define a static evaluation function as readily as chess
    - all pieces worth as much; trick is determining what territory is owned
- **Many believe that a smart Go program will be a true test of AI techniques**
  - Chess advances primarily due to fast CPU’s, brute-force search
  - Go will probably require: pattern recognition, knowledge bases, learning. For more on Go, see

http://www.computer-go.jp/index.html

Summary

- Games are fun (and dangerous)
- They illustrate several important points about AI
  - Perfection is unattainable -> must approximation
  - Good good idea to think about what to think about
  - Uncertainty constrains the assignment of values to states
- Games are to AI as grand prix racing is to automobile design.
- optimaldecisionsdependoninformationstate,notrealstate