Adversarial Search

(a.k.a. Game Playing)

Chapter
5

(Adapted from Stuart Russell, Dan Klein, and others. Thanks guys!)

Outline

- Games
- Perfect play: principles of adversarial search
  - minimax decisions
  - α–β pruning
  - Move ordering
- Imperfect play: dealing with resource limits
  - Cutting of search and approximate evaluation
- Stochastic games (games of chance)
- Partially Observable games
- Card Games

Games vs. search problems

- Search in Ch3&4: Single actor!
  - "single player" scenario or game, e.g., Boggle.
  - Brain teasers: one player against "the game".
  - Could be adversarial, but not directly *as part of game*
    - e.g. "I can find more words than you"

- Adversarial game: "Unpredictable" opponent shares control of state
  - solution is a strategy → specifying a move for every possible opponent response
  - Time limits ⇒ unlikely to find goal, must find optimal move with incomplete search
  - Major penalty for inefficiency (you get your clock cleaned)
  - Most commonly: "zero-sum" games. My gain is your loss = Adversarial

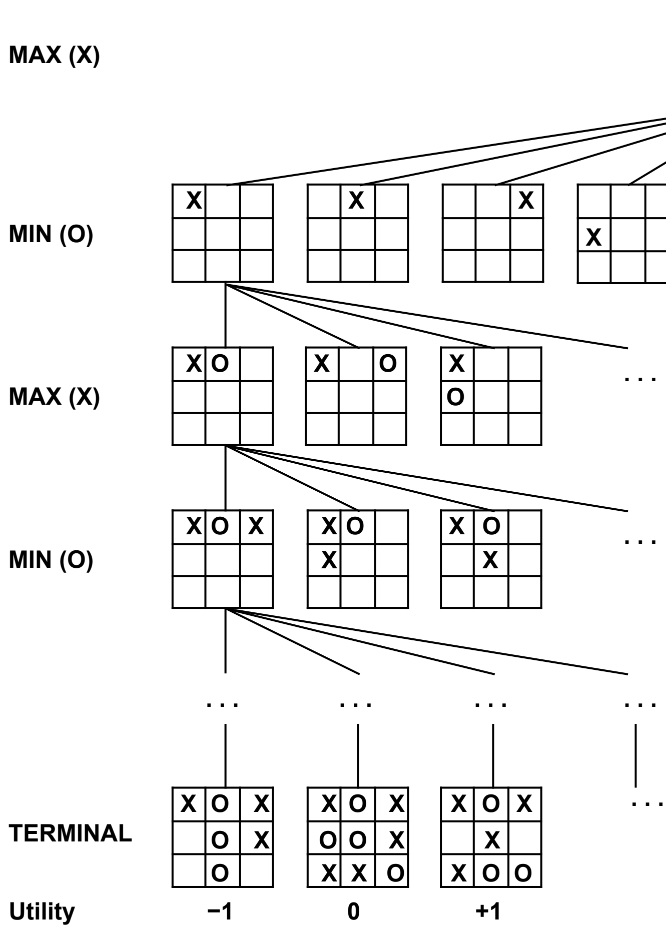
- Gaming has a deep history in computational thinking
  - Computer considers possible lines of play (Babbage, 1846)
  - Algorithm for perfect play (Zermelo, 1912; Von Neumann, 1944)
  - Finite horizon, approximate evaluation (Zuse, 1945; Wiener, 1948; Shannon, 1950)
  - First chess program (Turing, 1951)
  - Machine learning to improve evaluation accuracy (Samuel, 1952–57)
  - Pruning to allow deeper search (McCarthy, 1956)
  - Plus explosion of more modern results...

Types of Games

	deterministic	chance
perfect information	chess, checkers, go, othello, connect-4, tic-tac-toe	Backgammon, Monopoly, Chutes-n-ladders
imperfect information	Battleship, Blind tic-tac-toe, Kriegspiel	Bridge, Poker, Scrabble Nuclear war

- Access to Information
  - Perfect Info. Fully observable. Both player see whole board, all of the time
  - Imperfect Info. Not/partially-observable. Blind or partial knowledge of board.
- Determinism:
  - Deterministic: No element of chance. Players have 100% control over actions taken in game
  - Chance: Some element of chance: die rolls, cards dealing, etc.

Game tree (2-player, deterministic, turns)



Pondering Game Tree Size...

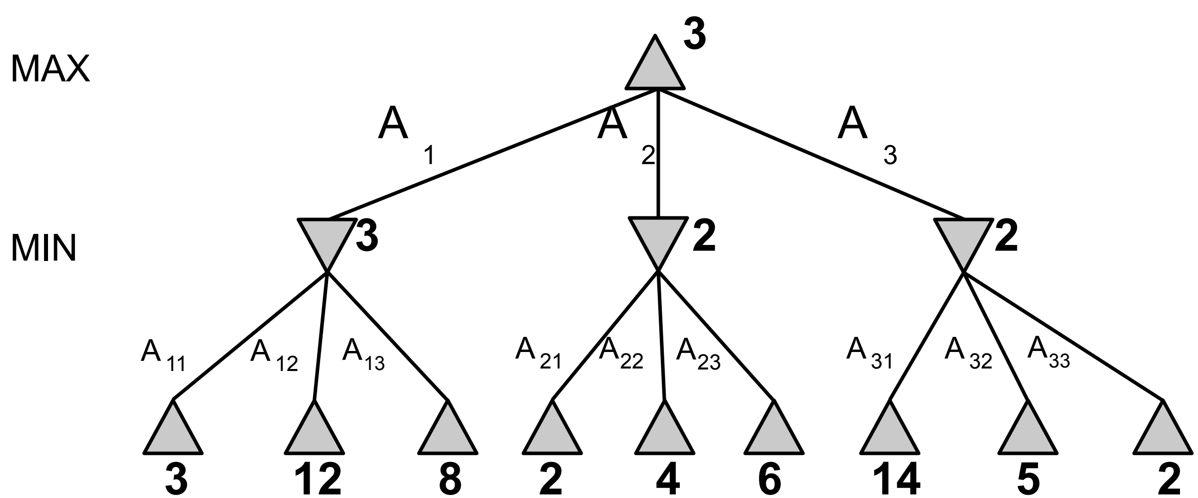
- Tic-tac-toe (3x3)
  - "Small" = 9! = 362,880 terminal nodes
- Chess
  - 1040 terminal nodes!
  - Never could generate whole tree!

Minimax Search

- Normal Search: Solution = seq. of actions leading to goal.
- Adversarial Search: Opponent interfering at every step!
  - Solution= Contingent plan of action
  - Finds optimal solution to goal, *assuming that opponent makes optimal counter-plays.*
  - Essentially an AND-OR tree (Ch4): opponent provides “non-determinism”

- Perfect play for deterministic, perfect-information games:
  - Idea: choose move to position with highest minimax value

- E.g., 2-ply game:



Minimax algorithm

```
function Minimax-Decision(state) returns an action
    inputs: state, current state in  game
    return the  a in Actions(state) maximizing Min-Value(Result(a, state))

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: Reflection

- Need to understand how minimax works!
- Recursive depth-first algorithm
  - Max-Value at one level...calls Min-Value at next...calls Max-Value at next.
  - Base case: Hits a **terminal** state = game is over → has known score (for max)
  - Scores "backed up" through the tree on recursive return
    - As each node fully explores its children, it can pass its value back
  - Score arriving back at root shows which move current player (max) should make
    - Makes move that maximizes outcome, *assuming optimal play by opponent*.

- Multi-player games?
  - Don't have just Max & Min. Have whole set of players A,B,C, etc.
  - Calculate **utility** *vector* of scores at each level/node
    - Contains node (board position) value for each player
  - Value of node = utility vector that maximizes benefit for player whose move it is

Properties of minimax search

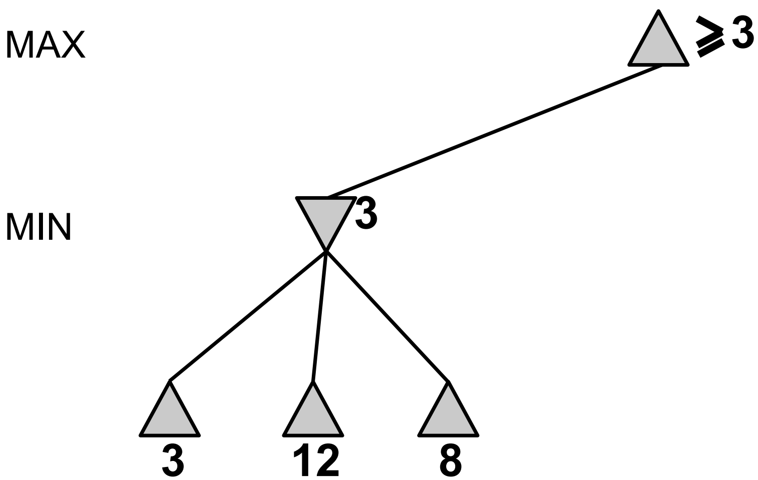
- Complete??
  - Yes, if tree is finite (chess has specific rules for this)
  - Minimax performs complete depth-first exploration of game tree
- Optimal??
  - Yes, against an optimal opponent. Otherwise??
- Time complexity??
  - O(bm)
- Space complexity??
  - O(bm) (depth-first exploration) (m is tree depth)
- Practical Analysis:
  - For chess, b ≈ 35, m ≈ 100 (moves) for "reasonable" games
    - Time cost gets out of range of "3 minute per move" standard fast!
    - ⇒ exact solution completely infeasible!

- Engage cleverness: do we really need to explore every path in tree?

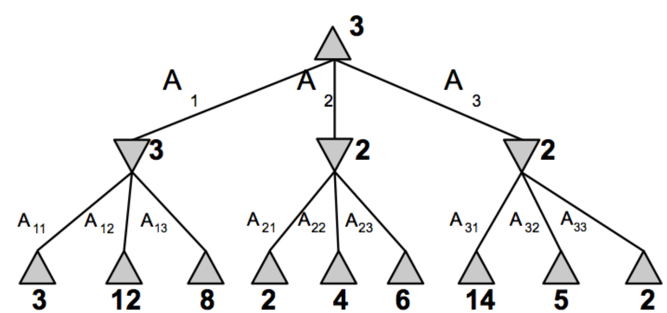
Alpha-Beta (α−β) pruning

- DFS plunges down tree to a terminal state fast!
  - Knows about one complete branch first…
  - Can we use this to *avoid* searching later branches?

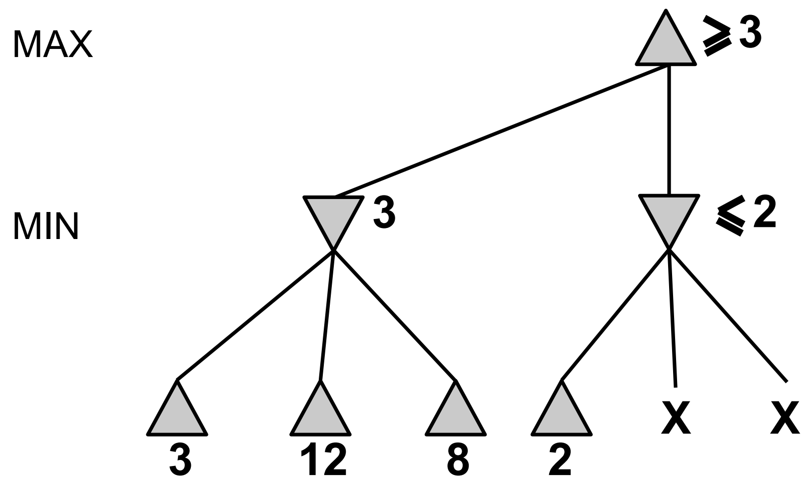
- Alpha-Beta pruning:



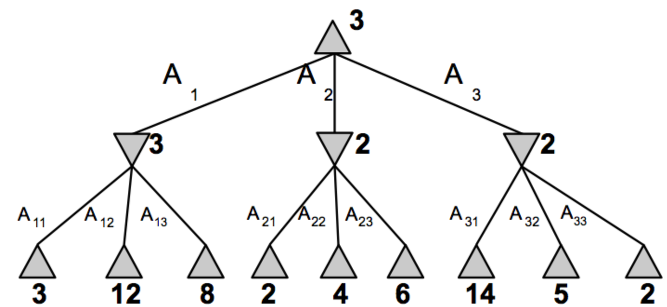
Reference: whole tree



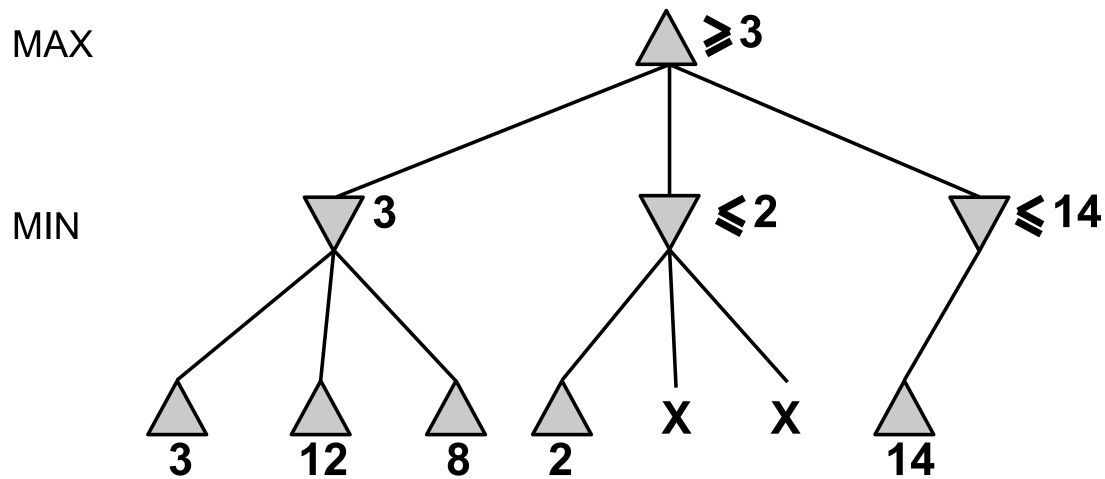
$\alpha-\beta$ pruning example



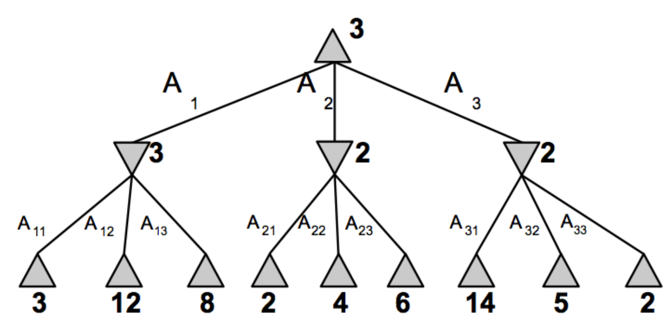
Reference: whole tree



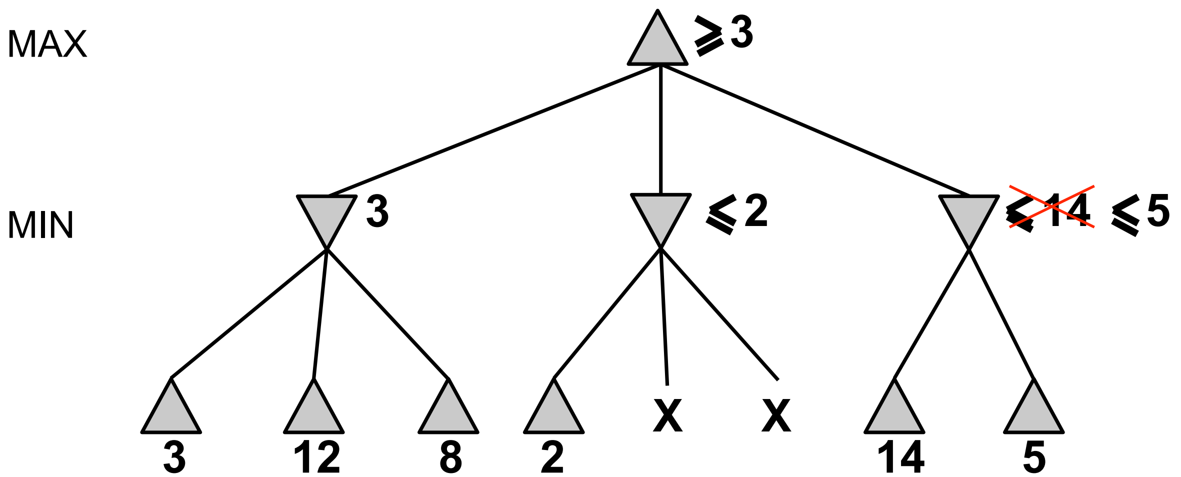
$\alpha-\beta$ pruning example



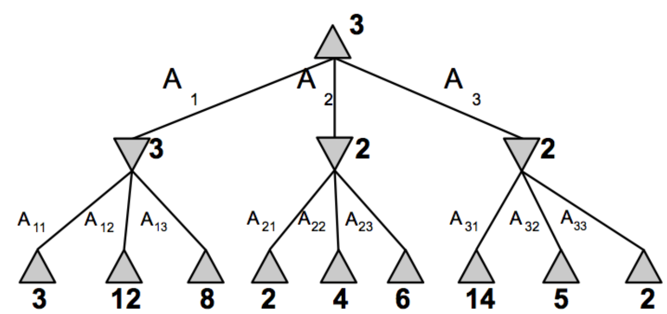
Reference: whole tree



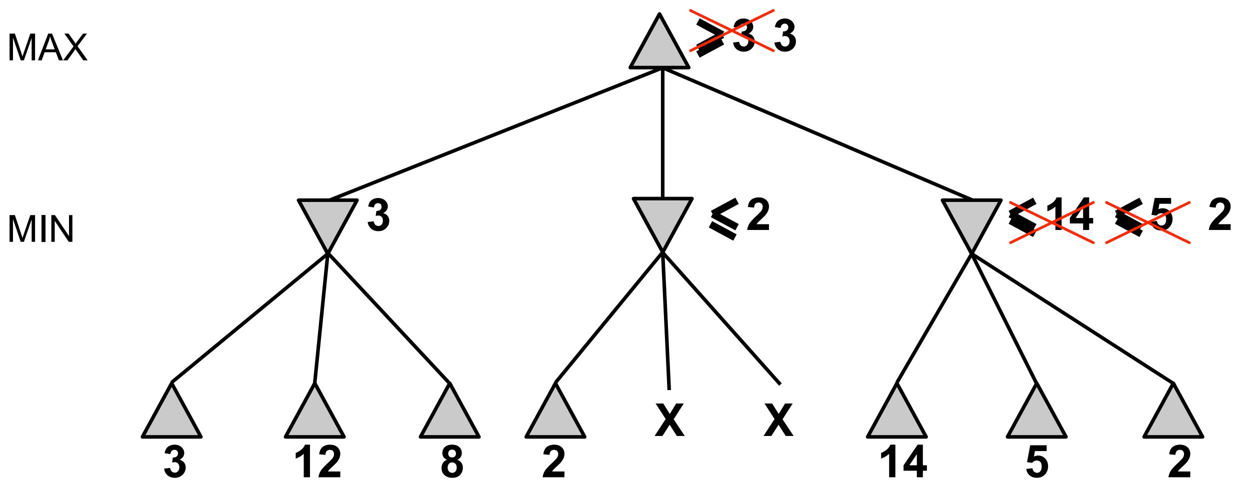
α–β pruning example



Reference: whole tree



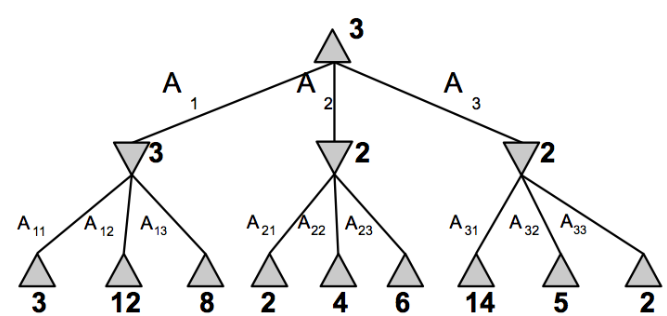
α–β pruning example



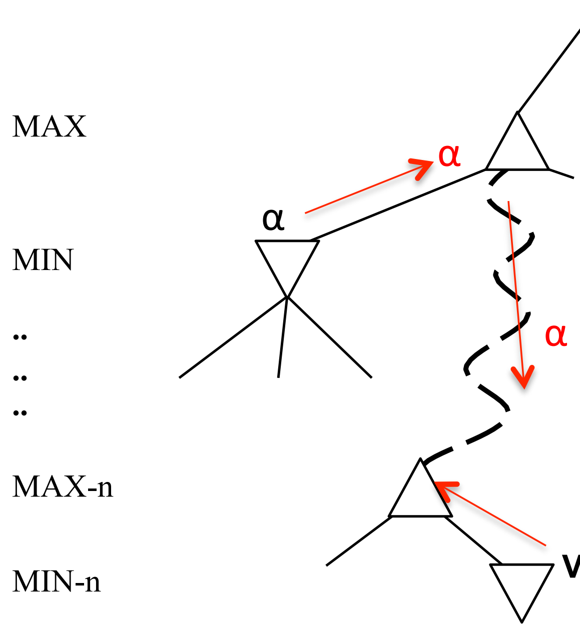
Observant Questions:

- What exactly is it that allowed pruning at <= 2 node?
- Why no pruning at sibling to right?
- More on this shortly...

Reference: whole tree



$\alpha-\beta$: Reflection on behavior



α is set/updated as first branch is explored…then sent down subsequent branches to prune with.

- α-β *maintains* two boundary values as it moves up/down tree
  - α is the best value (to max) found so far off the current path
  - β is the best value found so far at choice points for min

- Example: If V is worse than α, Max-n will avoid it
  - $\Rightarrow$ prune that branch
  - β works similarly for min

The α–β algorithm

```
function Alpha-Beta-Decision(state) returns an action
   return the a in Actions(state) maximizing Min-Value(Result(a, state))
```

```
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
```

```
function Min-Value(state, α, β) returns a utility value
   same as Max-Value but with roles of α, β reversed
```

Properties of α–β

- α–β observations:
  - Pruning is zero-loss.
    - Final outcome same as without pruning.

  - Great example of “meta-reasoning”= reasoning *about* computational process.
    - Here: reasoning about which computations could possibly be relevant (or not)
    - Key to high efficiency in AI programming.

  - Effectiveness depends hugely which path (moves) you examine first.
    - Slide 14: why prune in middle subtree…but not in rightmost one.
    - Middle subtree: examines highest value (for max) nodes first!

  - Analysis:
    - Chess has average branching factor around 35
    - Pruning removes branches (whole subtrees)
      - → effective branching factor = 28. Substantial reduction.

- Unfortunately, 28^{50} is still impossible to search in reasonable time!

Move ordering to improve α–β efficacy

- Plan: at any ply: examine higher value (to max) siblings *first*.
  - Sets the α value tightly → more likely to prune subsequent branches.
- Strategies:
  - Static: Prioritize higher value moves like captures, forward moves, etc.
  - Dynamic: prioritize moves that have been good in the past
    - Use IDS: searches to depth=n reveal high values moves for subsequent re-searches at depth > n.
- Stats:
  - Minimax search = $O(b^m)$
  - α–β with random ordering = about $O(b^{3m/4})$ → nice reduction
  - α–β with strong move ordering = about $O(b^{m/2})$
    - Effectively reduced b-factor from 35 to 6 in chess! Can ply *twice* as deep, same time!

- More power: transpositions
  - Some move chains are *transpositions* of each other. (a→b, then d→e) gives same board as (d→e, then b→a).
  - Identify and only compute **once**: can double reachable depth again!

Imperfect Game Play

- Reality check:
  - Thus far: minimax assumes we can search down to “bottom” of tree
  - Not realistic: minimax is $O(b^m)$
    - Chess = of 50 moves/game, b about 35
    - $O(35^{50})$….or, with theoretical best α–β move ordering: $O(6^{50})$. Huge!
  - Plan: Search as deep as time allows
    - Terminal-test() → Cutoff-test()
    - Cut-off-test(s) decides if we should stop searching at that state/level.
    - If true: apply *evaluation function* and return value of that board.

- When to cut off search?
  - Fred Flintstone static approach = just always cut off search at some depth d.
  - Problem: leaves valuable time on the table
    - Reachable depth within t-limit *varies* depending on board/# pieces/etc.
  - Solution: Use IDS.
    - Search until time is up → return result from latest completed search
    - Bonus: Use info from previous IDS runs to optimize α–β move ordering
  - Problem: *horizon effect* = something bad could happen *just beyond* search limit
  - Solution: Add *quiescence* metric. Never cut off search in middle of heavy action.

Advanced Techniques: when winning matters

- Idea 1: Find ways to search *deeper*.
  - Efficiency: efficient board representation, faster eval functions, etc.
  - Better pruning: maximize efficacy of move ordering subsystem
  - *Forward* pruning: cut off "un-interesting" branches of search tree early
    - **α–β** prunes nodes that are *provably* useless → loss-less
    - Forward pruning "guesses" → prunes nodes that are *probably* useless.
    - Danger: could prune away moves that ultimately lead to wins!
    - Strategy: shallow search gets rough node value. Stored info estimates *likely* utility

- Idea 2: More sophisticated evaluation function
  - Linear weighted function assume *independence* of features…statically
    - But often it's the *combo* of pieces that count…more at some points in game than others
    - E.g., pair of bishops > two bishops…but more so in the end-game
    - *Non-linear* weighted functions allow more subtle tuning
  - *Machine learning* can also be used to adjust weights from experience

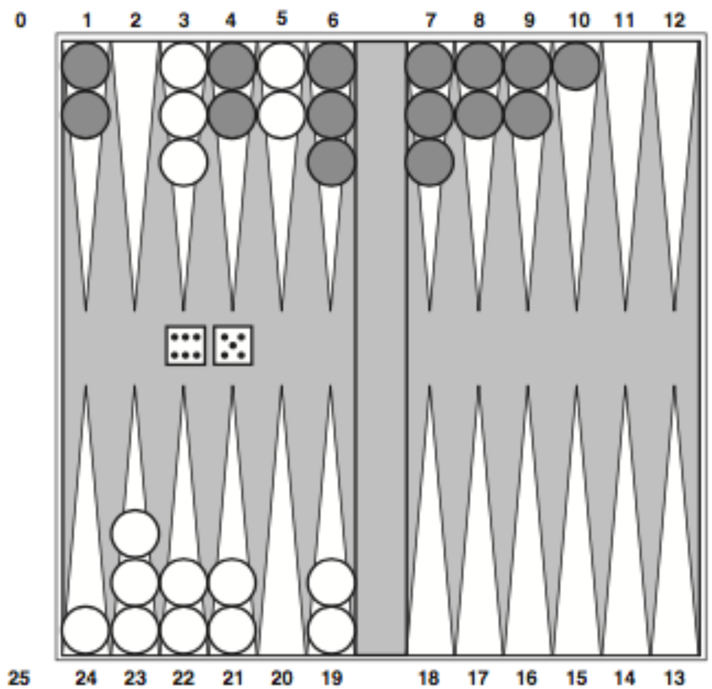
Advanced Techniques: when winning matters

- Idea 3: Avoid search completely when you can
  - In many games, there are certain *rote* phases
    - e.g. Chess: whole libraries of books about standard openings/end games
    - Why search down through billions of boards? Look it up!
  - Can just store and look-up moves for "standard" situations
    - Enter from books and other "human knowledge"
    - Calculate stats on DB of previously played games → which openings won most?
  - Computers can have advantage of humans here!
    - Human: has *general strategy* for certain endgames
      - King-rook-king (KRK) endgame, king-bishop-knight-king (KBNK), etc.
    - Computer: with so few pieces, can literally *compute* winning move sequence!
      - For *all possible* KRK endings, etc.
    - Computer recognizes a pre-computed sequence → plays perfect deterministic endgame!

History: Deterministic Games in practice…

- Checkers:
  - Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994.
  - Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.
- Chess:
  - Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997.
  - Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.
- Othello:
  - human champions refuse to compete against computers, who are too good.
- Go:
  - 2005: human champions refuse to compete against computers, who are too bad.
  - In go, b > 300, so most programs use pattern knowledge bases to suggest plausible moves.
  - 2017: IBM reveals it has been secretly entering its Go agent in online tournaments. And winning. Beats reigning Go champion four in a row…

Stochastic (non-deterministic games)



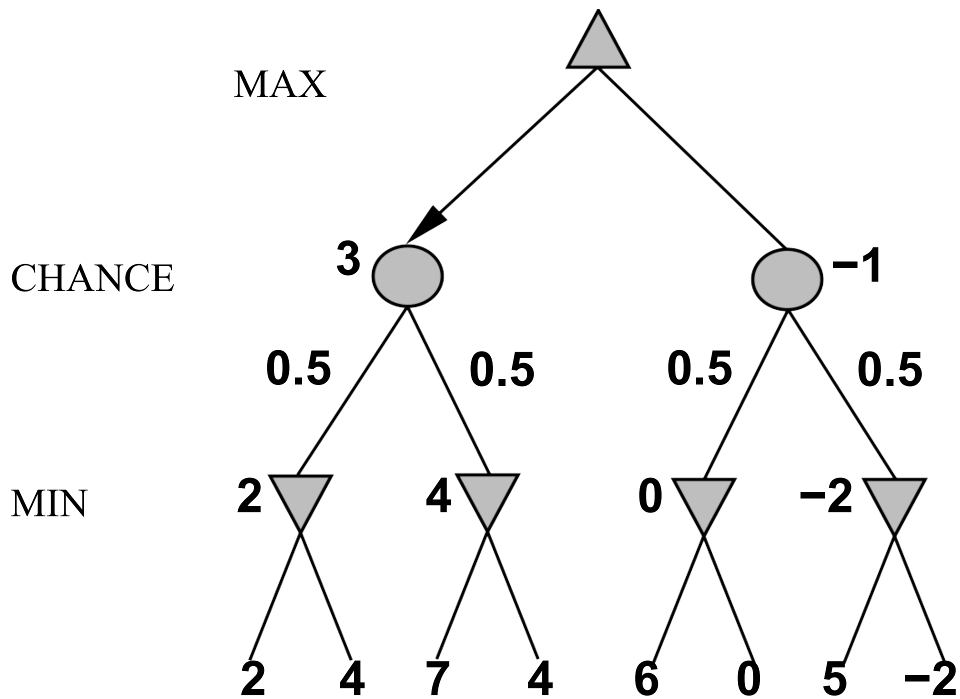
- Player-at-turn rolls dice:
- Can now move one piece 5 places, and another piece 6 places

- Combination of luck and skill
  - Strategy must account for roll of dice = random chance. **Plus** other player!
  - Backgammon: Dice determine possible moves
- Can't construct a standard game tree!

Non-deterministic Games

- Chance introduced by: dice, card-shuffling/dealing, drawing cards
- Minimax → Expectiminimax
  - Chance essentially acts as another "player"
  - Chance level= sum of *expected outcomes*, weighted by probability of happening.

- Simplified example with coin-flipping "move" inserted into some game:



Expectiminimax Algorithm

- Expectiminimax produces perfect play
  - Meaning: best possible play, given the stochastic probabilities involved.
- Just like Minimax, except we must also handle chance nodes:

```
. . .
If terminal-test(s)=true
        return Evaluation-fn(s)
if state is a Max node then
        return the highest ExpectiMinimax-Value of Successors(state)
if state is a Min node then
        return the lowest ExpectiMinimax-Value of Successors(state)
if state is a chance node then
        return SUM of probability-weighted(ExpectiMinimax-Value of Successors(state))
. . .
```

- Dice rolls increase *b*:
  - 21 possible rolls with 2 dice Backgammon $\approx$ 20 legal moves (can be 6,000 with 1-1 roll)
  - depth 4 = $20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$
  - Thus: As depth increases, probability of reaching a given node shrinks
    - $\Rightarrow$ value of lookahead is diminished
    - α–β pruning is much less effective (because chance makes pruning less common)
- TDGammon: uses depth-2 search + very good Eval$\approx$ world-champion level

Games of imperfect information

g.., card games, where opponent's initial cards are unknown

Typically we can calculate a probability for each possible deal

Seems just like having one big dice roll at the beginning of the game*Idea:

compute the minimax value of each action in each deal,
then choose the action with highest expected value over all deals*

Special case: if an action is optimal for all deals, it's optimal.*

GIB, current best bridge program, approximates this idea by
1) generating 100 deals consistent with bidding information
2) picking the action that wins most tricks on average

Partially Observable Games

- So far: Fully observable games
  - All player can see all functional pieces (state) of the game at all times
- Many games are fun because of *imperfect information*
  - Players see only none/part of opponents state.
  - E.g. Poker and similar card games, Battleship, etc.
- Example: Kriegspiel: Blind chess!
  - White and Black see only a board containing *their* pieces.
  - On turn: player *proposes* a move.
    - Referee announced: legal/illegal. If legal: “Capture on square X”, “Check by <direction>”, “checkmate” or “stalemate”.
  - Plan: Use belief states developed in Ch4!
    - Referee feedback = percepts that update/prune belief states.
    - All believe states NOT equally likely: can calculate probabilities on believe states based predicting optimum play by opponent.
    - Implication: Best to add some *randomness* to your play: be unpredictable!

Card Games

- *Stochastic* partial observability
  - Cards dealt randomly at the beginning of game. Deterministic after that.
  - Odds (probability) of possible hands easily calculated.
  - E.g. Bridge, Whist, Hearts, some forms of poker.

- Plan: Probabilistic weighted search
  - Generate all possible deals of the (missing) cards
  - Solve each one just like a fully observable games (Minimax)
  - Weight each outcome with probability of that hand being dealt
  - Chose move that has the best outcome, averaged over all possible deals.

- Reality check:
  - In Bridge there are 10+ million possible visible hands. Can't explore all!
  - Idea: Monte Carlo approach: solve random sample of deals
    - Choice of sample set is weighted to include more likely hands.
  - *Bidding* may add valuable info on hands → changes probabilities.

- GIB, leading bridge program: generates 100 deals consistent with bidding

Summary

- Games are just specialized search problems. Modifications:
  - Minimax (plus α–β pruning) to model opponent player
  - Stochastic “choice” layers in tree to model chance
  - Belief state management to model partial observability
  -
- Games illustrate several important points about AI
  - perfection is unattainable in reality ⇒ must approximate
  - good idea to think about what to think about
    - Meta-level analysis, as in considerations leading to α–β pruning
  - Uncertainty constrains the assignment of values to states
    - Increases effective branching factor, could make pruning less effective
- Optimal decisions depend on information state, not real state
  - As illustrated in partially observable games, when belief state is what matters
- Games are to AI as grand prix racing is to automobile design
  - Proving ground for hardware, data structures, algorithms…and cleverness


STOP