Search Algorithms II

3

3 Search Algorithms II

3.1 Local search
- Hill-climbing
- Simulated annealing
- Genetic algorithms*

3.2 Adversarial search
- minimax decisions
- α–β pruning
- Monte Carlo tree search

3.3 Online search+

3.4 Metaheuristic search*

Local Search

Local search algorithms operate using a single current (rather than multiple paths) and generally move only to neighbors of that node

Local search vs. global search

– global search, including informed or uninformed search, systematically explores paths from an initial state

– global search problems: observable, deterministic, known environments

– local search uses very little memory and finds reasonable solutions in larger or infinite (continuous) state spaces for which global search is unsuitable

Local Search

Local search is useful for solving **optimization problems**

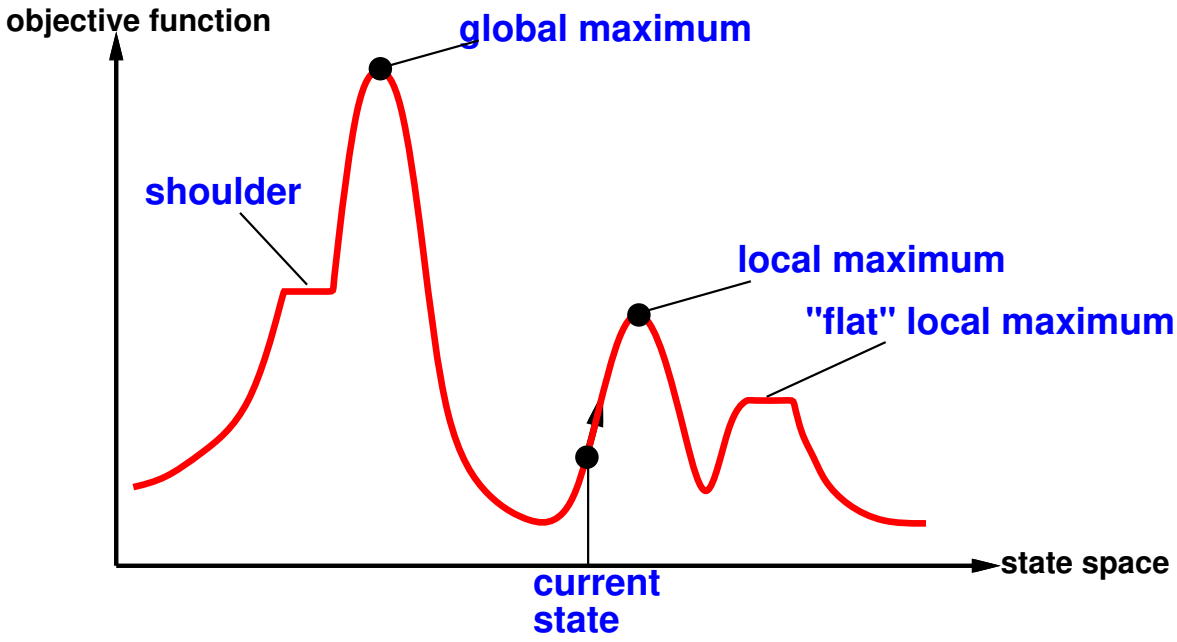
— the best estimate of "objective function", e.g., reproductive fitness in nature by Darwinian evolution

Local search algorithms

- Hill-climbing (greedy local search)
- Simulated annealing
- Genetic algorithms

Hill-climbing

Useful to consider state space landscape, gradient descent



Random-restart hill climbing overcomes local maxima — trivially complete (with probability approaching 1)

Random sideways moves: escape from shoulders, but loop on flat maxima

Hill-climbing

Like climbing a hill with amnesia (or gradient ascent/descent)

```
def HILL-CLIMBING( problem)
    current ← problem.INITIAL
    while true do
        neighbor ← a highest-valued successor of current
        if VALUE(neighbor) ≤ VALUE(current) then return current  // a local max
          current ← neighbor
    return a state that is a local maximum
```

The algorithm halts if it reaches a plateau where the best successor has the same value as the current state

Simulated annealing

Idea: escape local maxima by allowing some "bad" moves
but gradually decrease their size and frequency

```
def SIMULATED-ANNEALING( problem, schedule)
    current ← problem.INITIAL
    for t=1 to ∞ do
        T ← schedule[t]
        if T=0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE(next) – VALUE(current)
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(ΔE/T)
```

Properties of simulated annealing#

At fixed "temperature" T, state occupation probability reaches Boltzman distribution (see later in probabilistic distribution)

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

T decreased slowly enough $\Longrightarrow$ always reach best state x^*
because $e^{\frac{E(x^*)}{kT}} / e^{\frac{E(x)}{kT}} = e^{\frac{E(x^*)-E(x)}{kT}} \gg 1$ for small T
find a global optimum with probability approaching 1

Devised (Metropolis et al., 1953) for physical process modeling

Simulated annealing is a field in itself, widely used in VLSI layout, airline scheduling, etc.

Local beam search

Beam search by local search to choose top k of all their successors

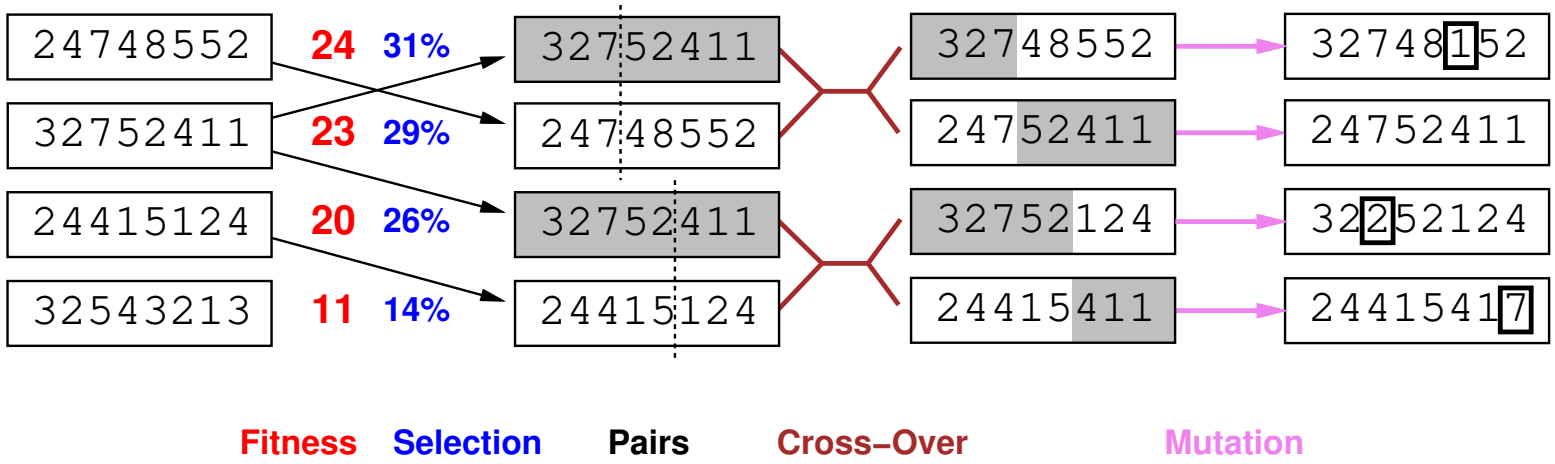
Problem: quite often, all k states end up on same local hill

Idea: choose k successors randomly (stochastic beam search), biased towards good ones

Observe the close analogy to natural selection

Genetic algorithms*

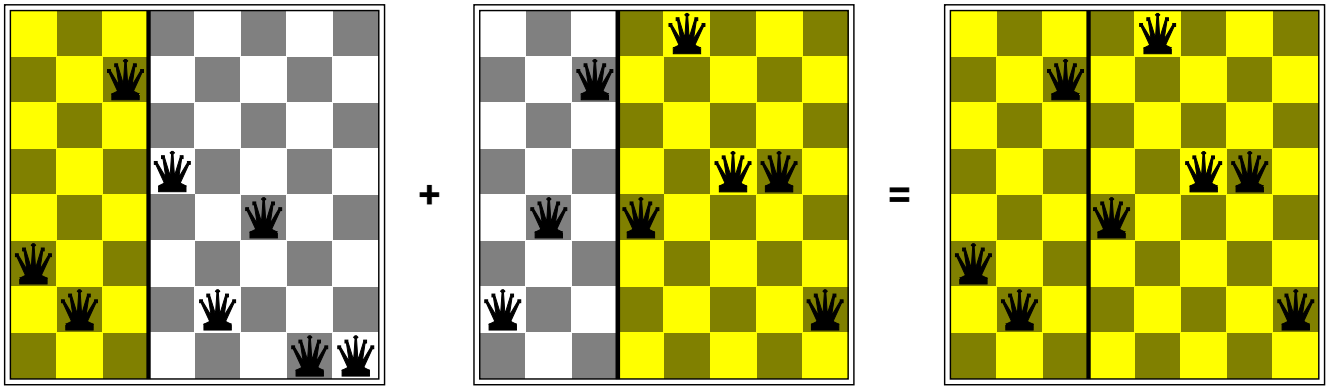
GA= stochastic local beam search + generate successors from **pairs** of states



Genetic algorithms

GAs require states encoded as strings (GPs use programs)

Crossover helps **iff substrings are meaningful components**



GAs $\neq$ evolution: e.g., real genes encode replication machinery

A.k.a, evolutionary algorithms

Genetic programming (GP) is closely related to GAs

Artificial Life (AL) moves one step further

Local search in continuous state spaces*

Suppose we want to site three airports in Romania

– 6-D state space defined by (x_1, y_2), (x_2, y_2), (x_3, y_3)

– objective function $f(x_1, y_2, x_2, y_2, x_3, y_3) =$

sum of squared distances from each city to the nearest airport

Discretization methods turn continuous space into discrete space,
e.g., empirical gradient considers $\pm\delta$ change in each coordinate
Gradient methods compute

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3}\right)$$

to increase/reduce f, e.g., by $\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$
Sometimes can solve for $\nabla f(\mathbf{x}) = 0$ exactly (e.g., with one city).
Newton–Raphson (1664, 1690) iterates $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x}) \nabla f(\mathbf{x})$
to solve $\nabla f(\mathbf{x}) = 0$, where $\mathbf{H}_{ij} = \partial^2 f / \partial x_i \partial x_j$

Hint: Newton-Raphson method is an efficient local search

Adversarial search

- Games
- Perfect play
  - minimax decisions
  - α–β pruning
- Imperfect play
  - Monte Carlo tree search

Games

Game as adversarial search

"Unpredictable" opponent $\Rightarrow$ solution is a strategy
specifying a move for every possible opponent reply
Time limits $\Rightarrow$ unlikely to find goal must approximate
Plan of attack

- 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)

Types of games#

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

Computer game

Single game playing: a program to play one game

General Game Playing (GGP): program to play more than one game

Perfect play

Perfect information: deterministic to each player, zero-sum games

Games of chess

Checkers $\rightarrow$ Othello $\rightarrow$ Chess(/Chinese Chess/Shogi) $\rightarrow$ Go

Zermelo theorem: if the game cannot end in a draw, then one of the two players must have a winning strategy (i.e. force a win)

Search state spaces are vast for Go/Chess

– each state is a point of decision-making to move

Go: Legal position (Tromp and Farnebäck 2007)

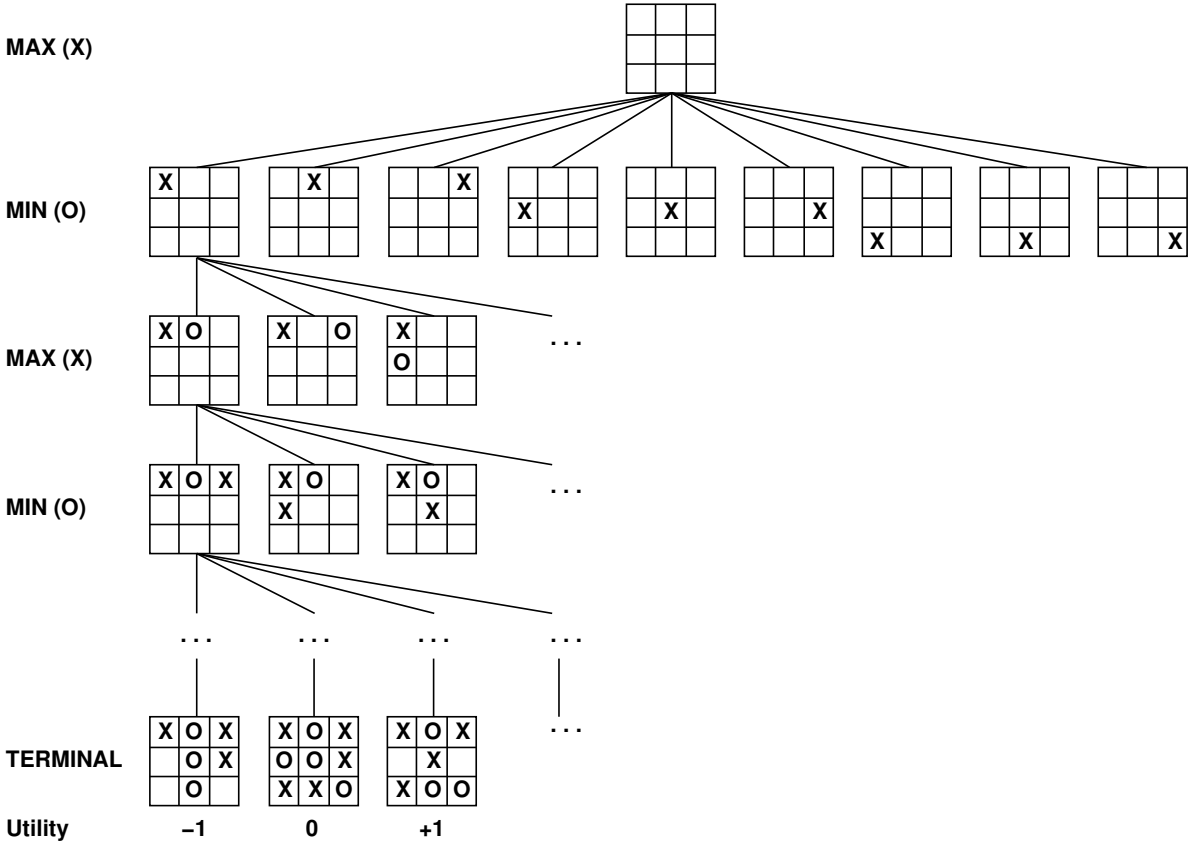
3^{361} (empty/black/white, 19×19 board), about 1.2% legal rate

$3^{361} \times 0.01196\cdots = 2.08168199382\cdots \times 10^{170}$

– the observable universe contains around 10^{80} atoms

Possible to reduce the space as small enough as likely exhaustively search??

Game tree (2-player, tic-tac-toe)



Small state space ⇒ First win (cf. Zermelo theorem)

Go: a high branching factor ($b \approx 250$), deep ($d \approx 150$) tree

Minimax

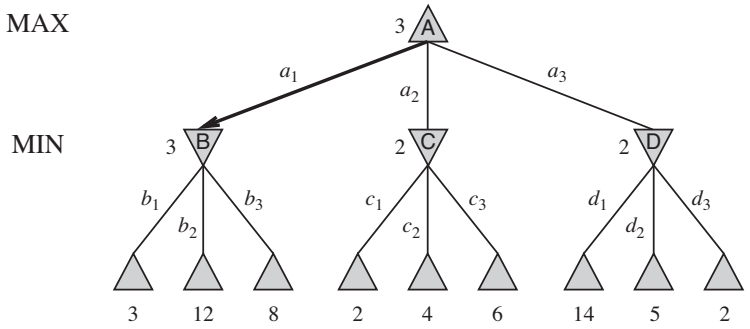
Perfect play for deterministic, perfect-information games

Idea: choose move to position with the best minimax value

= best achievable payoff, computing by the utility

(assuming that both players play optimally to the end of the game, the minimax value of a terminal state is just its utility)

E.g., a 2-ply game



MAX – the highest minimax; MIN – the lowest minimax

$argmax_{a \in S} f(a)$: computes the element a of set S that has the maximum value of $f(a)$ ($argmin_{a \in S} f(a)$ for the minimum)

Minimax algorithm

```
def MINIMAX-SEARCH(game,state)
    player ← game.TO-MOVE(state)  //The player's turn is to move
    value,move ← MAX-VALUE(game,state)
    return move //an action
def MAX-VALUE(game,state)
    if game.IS-TERMINAL(state) then return game.UTILITY(state,palyer),null
      // a utility function defines the final numeric value to player
    v ← −∞
    for each a in game.ACTIONS(state) do
        v2,a2 ← MIN-VALUE(game,game.RESULT(state, a))
        if v2 > v then
        v,move ← v2,a
    return v,move //a (utility,move) pair
```

Minimax algorithm

```
def MIN-VALUE(game,state)
    if game.IS-TERMINAL(state) then return game.UTILITY(state,palyer),null
    v ← +∞
    for each a in game.ACTIONS(state) do
        v2,a2 ← MAX-VALUE(game,game.RESULT(state, a))
        if v2 < v then
        v,move ← v2,a
    return v,move
```

Properties of minimax#

Complete??

Properties of minimax#

Complete?? Only if a tree is finite (chess has specific rules for this)
(A finite strategy can exist even in an infinite tree)

Optimal??

Properties of minimax#

Complete?? Yes, if a tree is finite (chess has specific rules for this)

Optimal?? Yes, against an optimal opponent

Time complexity??

Properties of minimax#

Complete?? Yes, if a tree is finite (chess has specific rules for this)

Optimal?? Yes, against an optimal opponent. Otherwise??

Time complexity?? $O(b^m)$

Space complexity??

Properties of minimax#

Complete?? Yes, if a tree is finite (chess has specific rules for this)

Optimal?? Yes, against an optimal opponent. Otherwise??

Time complexity?? $O(b^m)$

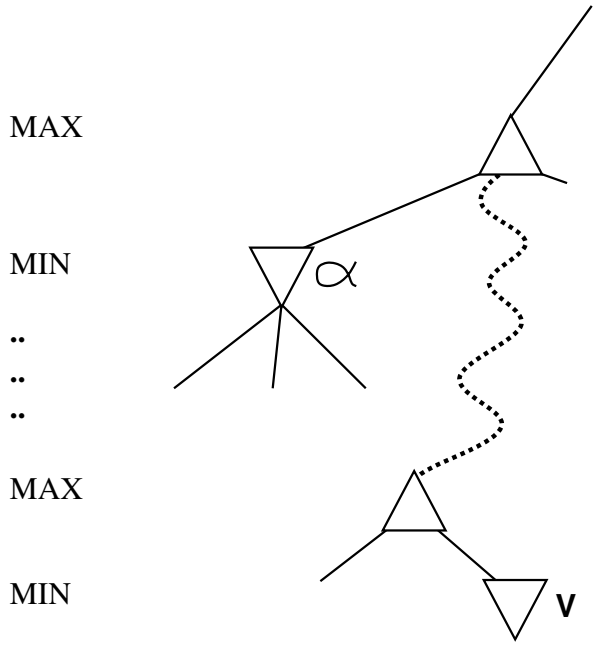
Space complexity?? $O(bm)$ (depth-first exploration)

For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
$\Rightarrow$ exhaustive search is infeasible

For Go, $b \approx 250$, $m \approx 150$

But do we need to explore every path?

α–β pruning

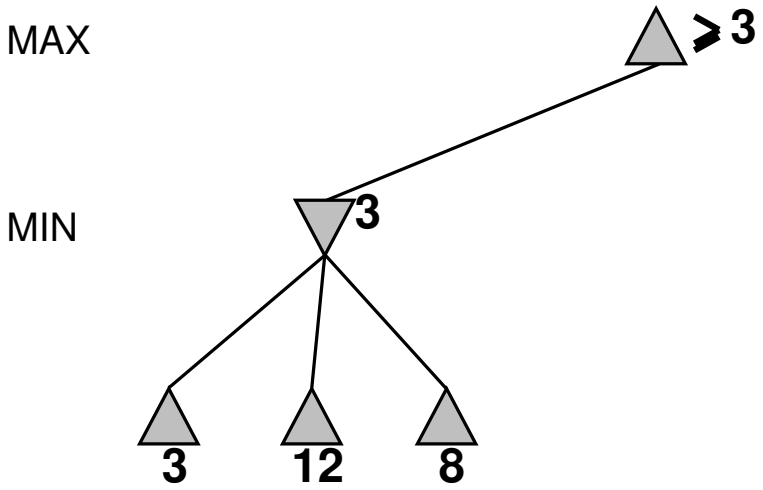


α is the best value (to MAX) found so far off the current path

If V is worse than α, MAX will avoid it $\Rightarrow$ prune that branch

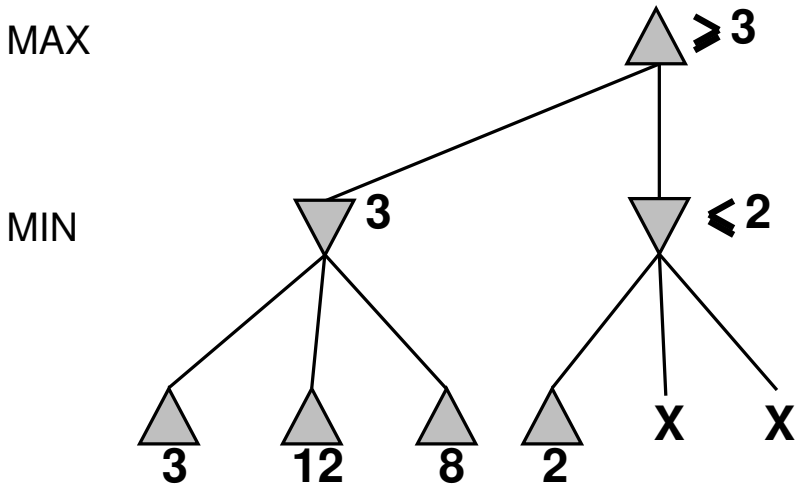
Define β similarly for MIN

Example: α–β pruning



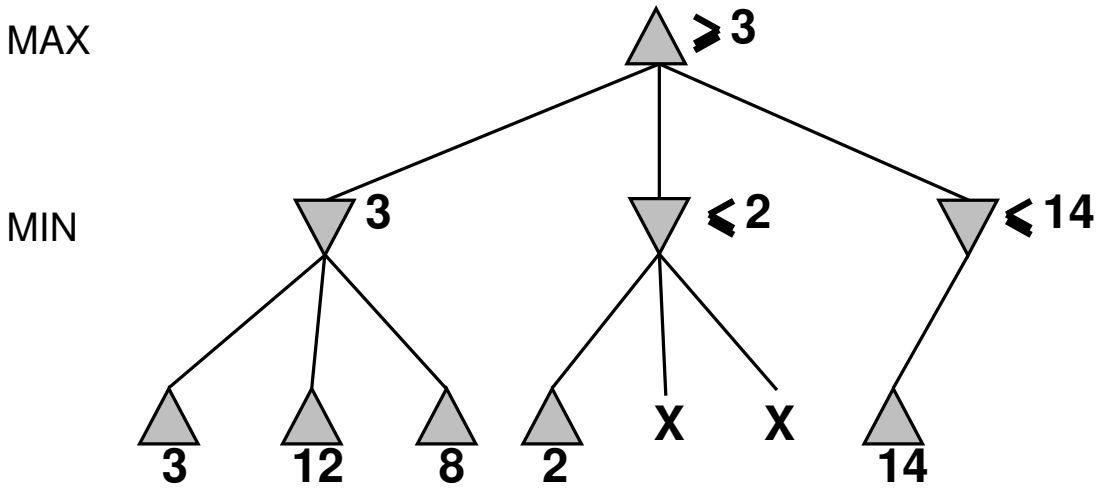
The first leaf below MIN node has a value at most 3

Example: α–β pruning



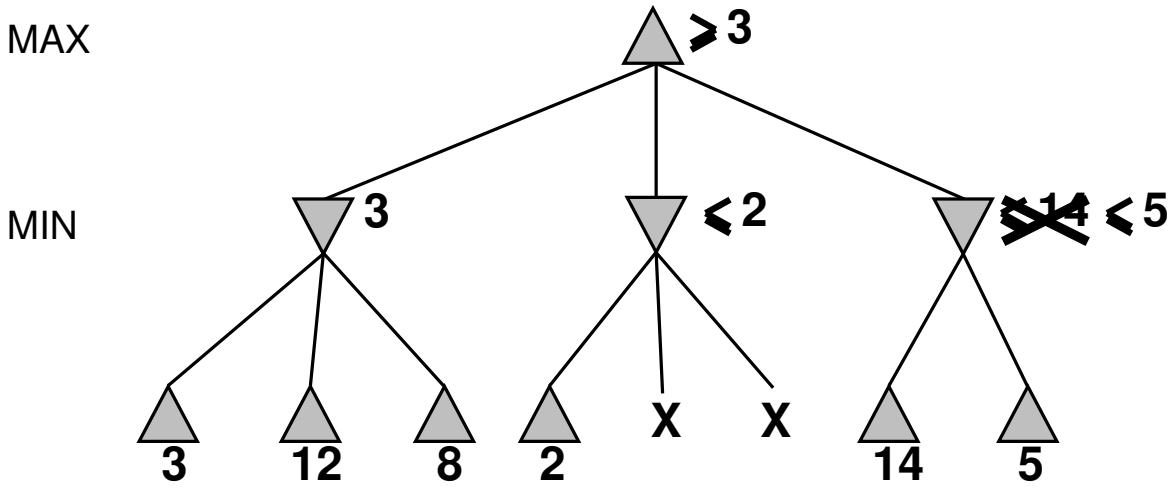
The second leaf has a value of 12, MIN would avoid, and so is still at most 3

Example: α–β pruning



The third leaf has a value of 8, the value of MIN is exactly 3 with all the successor states

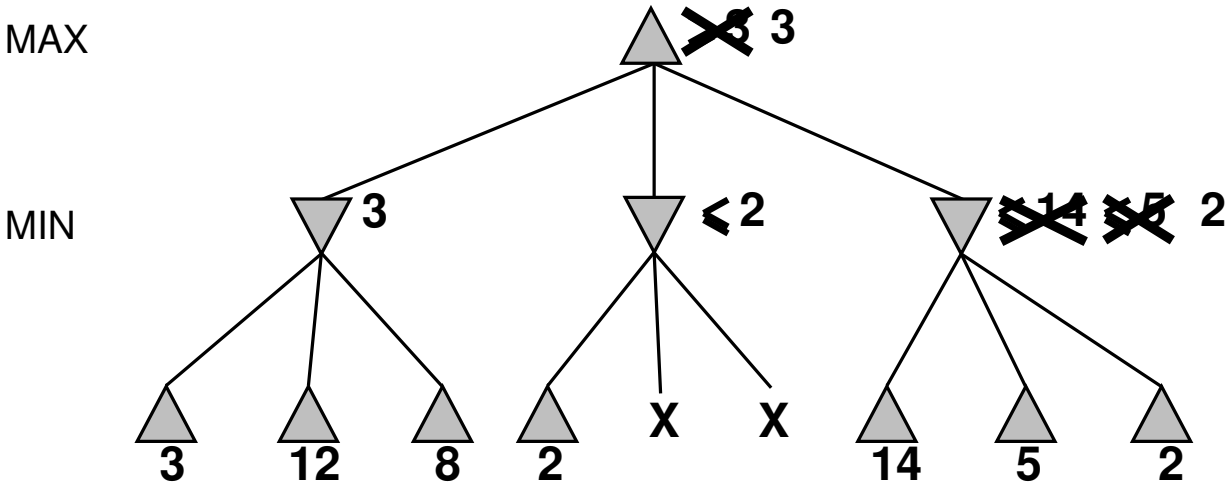
Example: α–β pruning



The first leaf below the second MIN node has a value at most 2, but the first MIN node is worth 3, so MAX would never choose it and look at other successor states

— pruning

Example: α–β pruning



— pruning

The α–β algorithm#

```
def ALPHA-BETA-PRUNING(game,state)
    player ← game.TO-MOVE(state)
    value,move ← MAX-VALUE(game,state, −∞, +∞)
    return move
def MAX-VALUE(game,state,α,β)
    if game.IS-TERMINAL(state) then return game.UTILITY(state,palyer),null
    v ← −∞
    for each a in game.ACTIONS(state) do
        v2,a2 ← MIN-VALUE(game,game.RESULT(state, a),α,β)
        if v2 > v then
        v,move ← v2,a
        β ← MAX(α,v)
        if v ≥ β then return v,move
    return v,move
```

The α–β algorithm

```
def MIN-VALUE(game,state,α,β)
    if game.IS-TERMINAL(state) then return game.UTILITY(state,palyer),null
    v ← +∞
    for each a in game.ACTIONS(state) do
        v2,a2 ← MAX-VALUE(game,game.RESULT(state, a),α,β)
        if v2 < v then
        v,move ← v2,a
        α ← MAX(β,v)
        if v ≤ α then return v,move
    return v,move
```

Note: The same as the MINIMAX-SEARCH except for α, β to prune

Properties of α–β

Pruning **does not** affect the final result

Good move ordering improves the effectiveness of pruning

With "perfect ordering," time complexity $= O(b^{m/2})$
$\Rightarrow$ **doubles** solvable depth

A simple example of the value of reasoning about which computations are relevant (a form of metareasoning)

Unfortunately, 35^{50} is still impossible (for chess)

Depth-first minimax search with α-β pruning achieved super-human performance in chess, checkers and othello, but not effective in Go

Imperfect play

Resource limits

– deterministic game may have imperfect information in real-time

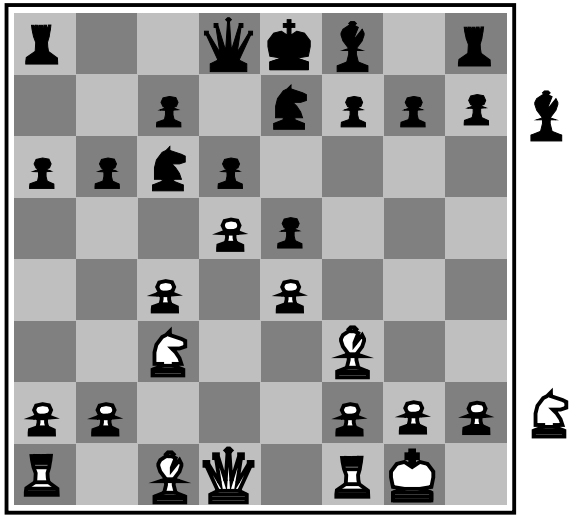
- Use CUTOFF-TEST instead of IS-TERMINAL
  e.g., depth limit
- Use EVAL instead of UTILITY
  i.e., eval. function that estimates the desirability of the position

Suppose we have 100 seconds, explore 10^4 nodes/second

$\Rightarrow 10^6$ nodes per move $\approx 35^{8/2}$

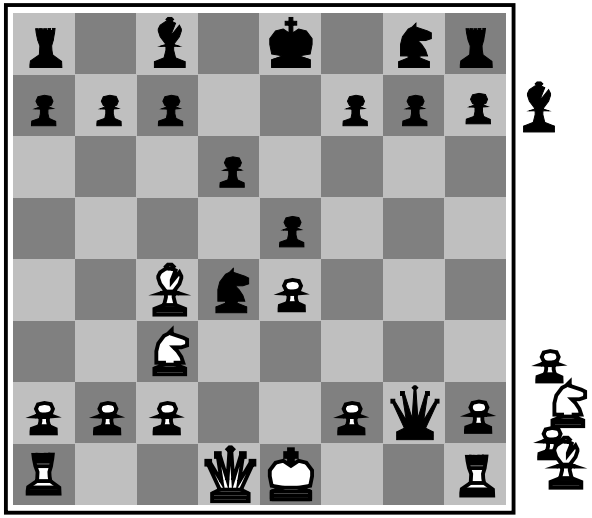
$\Rightarrow \alpha$–β reaches depth $8 \Rightarrow$ pretty good chess program

Evaluation functions



Black to move

White slightly better



White to move

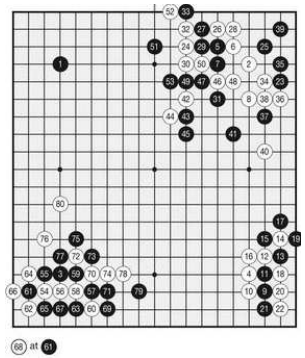
Black winning

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., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) – (number of black queens), etc.

Evaluation functions



For Go, the simply linear weighted sum of features

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

e.g., for some state s, $w_1 = 9$ with
$f_1(s) =$ (number of Black good) – (number of White good), etc.

Evaluation functions need human knowledge and are hard to design

Go lacks any known reliable heuristic function
– difficulty than (Chinese) Chess

Deterministic (perfect information) games in practice

Checkers: Chinook ended the 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

Othello: human champions refuse to compete against computers, who are too good

Chess: IBM Deep Blue defeated human world champion Gary Kasparov in 1997. Deep Blue uses very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply

Deterministic games in practice

AlphaGo

– Defeated human world champion in 2016 and in 2017

– AlphaGo Zero defeated AlphaGo in 2017

– AlphaZero: a GGP program, achieved within 24h a superhuman level of play in the games of Chess/Shogi/Go (defeated AlphaGo Zero) in Dec. 2017

MuZero: extension of AlphaZero including Atari in 2019

– outperform AlphaZero

– without any knowledge of the game rules

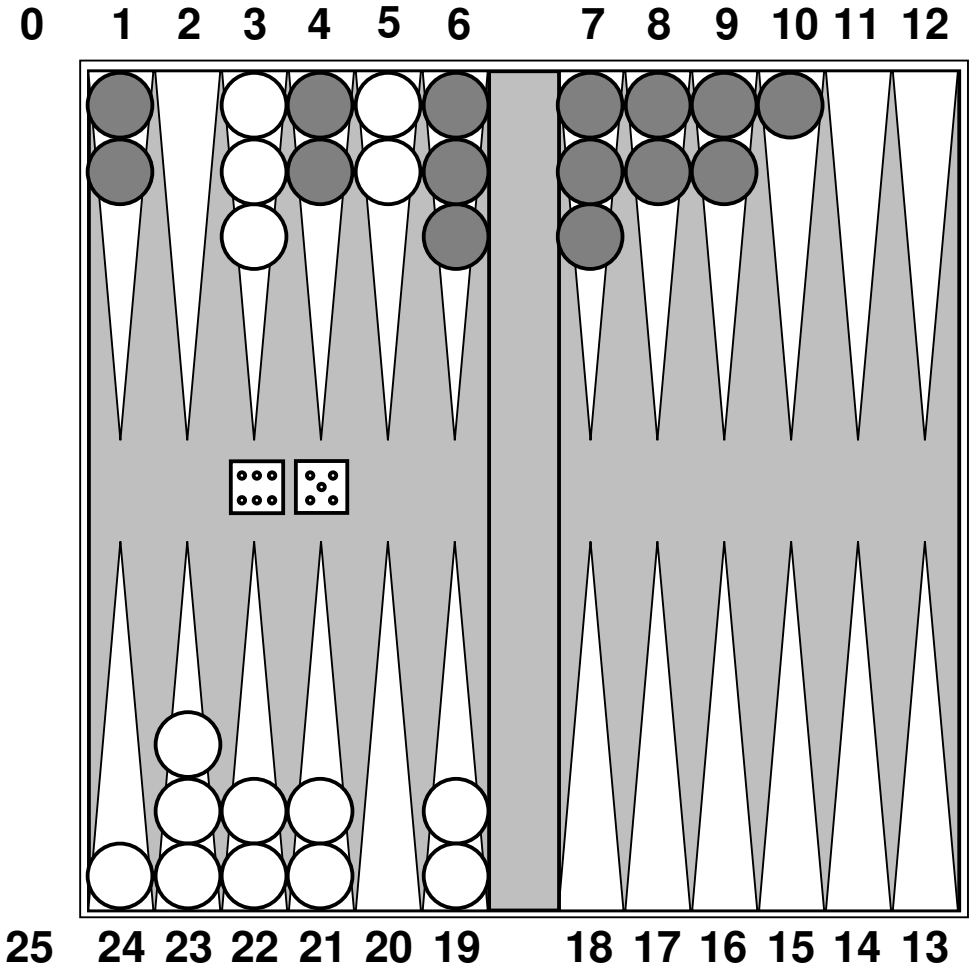
Achievements:

- "the God of the chess" of superhuman
- self-learning without prior human knowledge

Chinese Chess: the algorithm of AlphaZero/MuZero can be directly used for Chess and similar deterministic games

Almost all deterministic games have been well defeated by AI

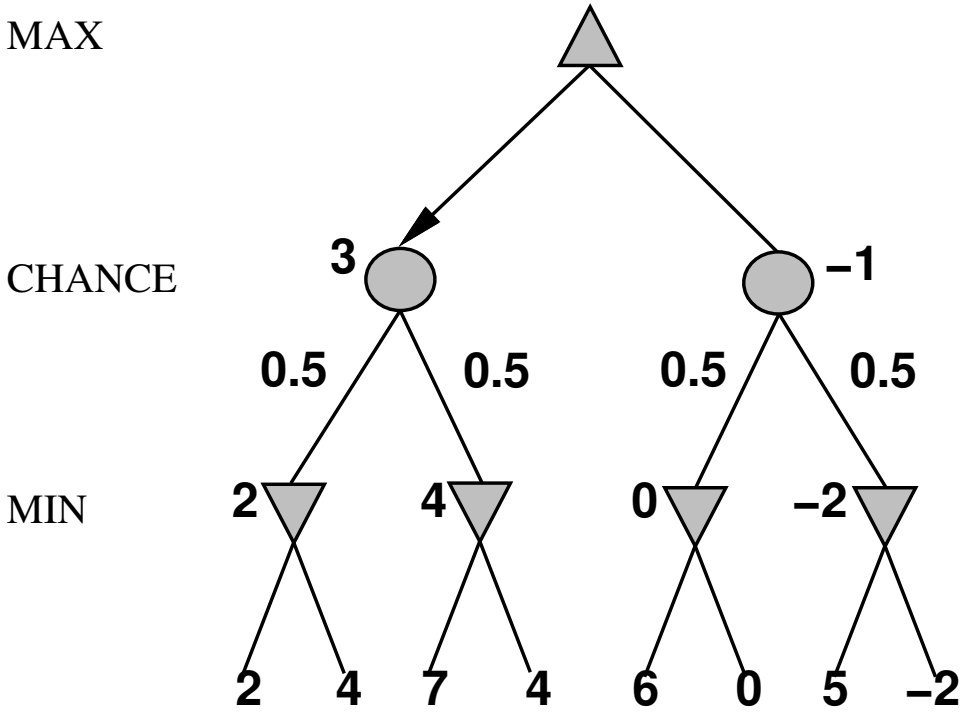
Nondeterministic games: backgammon



Nondeterministic games in general

In nondeterministic games, chance introduced by dice, card-shuffling

Simplified example with coin-flipping:



Algorithm for nondeterministic games

EXPECTIMINIMAX gives perfect play

Just like MINIMAX, except we must also handle chance nodes

```
...
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 average of EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
...
```

Nondeterministic (perfect information) games in practice

Dice rolls increase b: 21 possible rolls with 2 dice
Backgammon $\approx$ 20 legal moves (can be 6,000 with 1-1 roll)

$$\text{depth } 4 = 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$$

As depth increases, the probability of reaching a given node shrinks
$\Rightarrow$ value of lookahead is diminished

α–β pruning is much less effective

TDGammon uses depth-2 search + very good Eval
$\approx$ world-champion level

Games of imperfect information

E.g., card games, where the 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 the highest expected value over all deals

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

Monte Carlo tree search

MCTS

– a heuristic, expanding the tree based on a random sampling of the state space

– like as depth-limited minimax (with α-β pruning)

– interest due to its success in computer Go since 2006

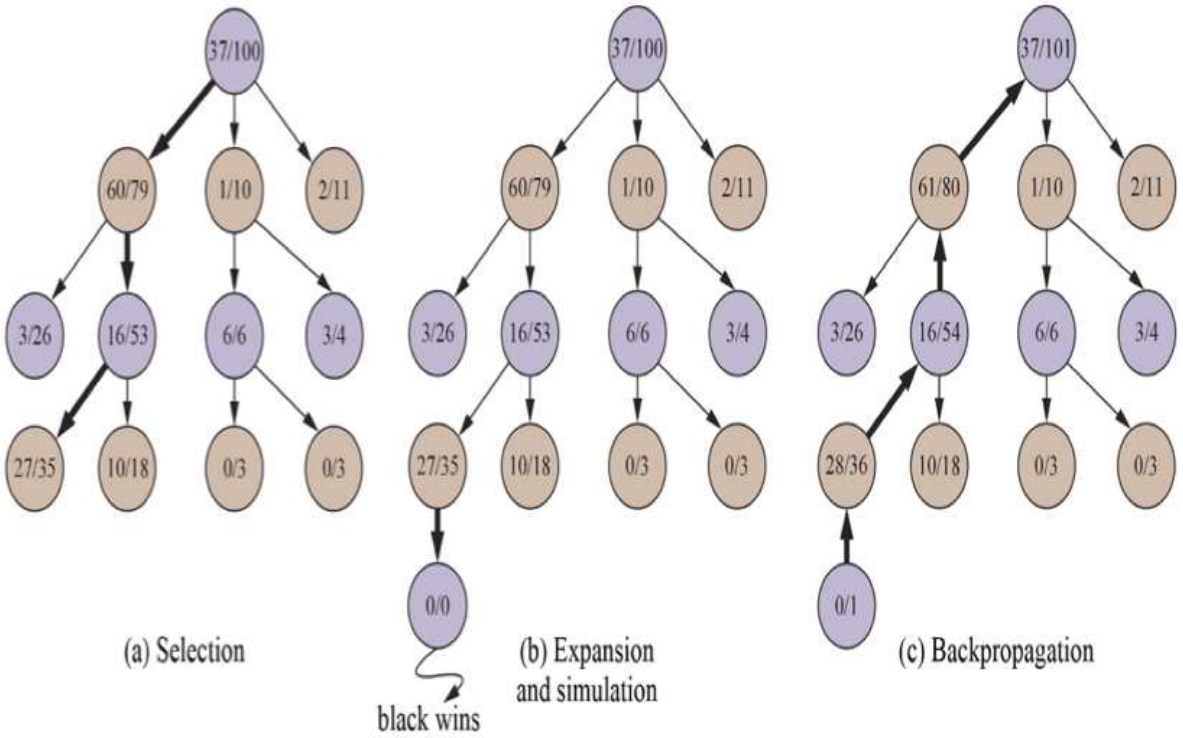
(Kocsis L et al. UCT, Gelly S et al. MoGo, tech. rep., Coulom R coining term MCTS, 2006)

Motivation: evaluation function $\Leftarrow$ **stochastic simulation**

A simulation (playout or rollout) chooses moves first for one player, then for the other, repeating until a terminal position is reached

Note: Named after the Casino de Monte-Carlo in Monaco

MCTS



After 100 iterations (a) select moves for 27 wins for blackout of 35 playouts; (b) expand the selected node and do a playout ended in a win for black; (c) the results of the playout are back-propagated up the tree (incremented 1)

MCTS

a. Selection: starting at the root, a child is recursively selected to descend through the tree until the most expandable node is reached
b1. Expansion: one (or more) child nodes are added to expand the tree, according to the available actions
b2. Simulation: a simulation is run from the new node(s) according to the default policy to produce an outcome (random playout)
c. Backpropagation: the simulation result is "backed up" through the selected nodes to update their statistics

MCTS policy

- Tree Policy: selects a leaf node from the nodes already contained within the search tree (selection and expansion)

  – focuses on the important parts of the tree, attempting to balance

  – – exploration: looking in states that have not been well sampled yet (that have had few playouts)

  – – exploitation: looking in states which appear to be promising (that have done well in past playouts)

- Default (Value) Policy: play out the domain from a given non-terminal state to produce a value estimate (simulation and evaluation) — actions chosen after the tree policy steps have been completed

  – in the simplest case, to make uniform random moves

  – values of intermediate states don't have to be evaluated, as for depth-limited minimax

Exploitation-exploration

Exploitation-exploration dilemma

– one needs to balance the exploitation of the action currently believed to be optimal with the exploration of other actions that currently appear suboptimal but may turn out to be superior in the long run

There are several variations of tree policy, says UCT (see the next page),

Theorem: UCT allows MCTS to converge to the minimax tree and is thus optimal, given enough time (and memory)

Upper confidence bounds for trees#

UCT: say tree policy UCB1 — an upper confidence bound formula for ranking each possible move

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N\left(\text{PARENT}(n)\right)}{N(n)}}$$

$U(n)$ — the total utility of all playouts through node n
$N(n)$ — the number of playouts through n
$\text{PARENT}(n)$ — the parent node of n in the tree

Exploitation $\frac{U(n)}{N(n)}$: the average utility of n

Exploration $\sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$: the playouts are given to the node with highest average utility

C ($\sqrt{2}$) — a constant that balances exploitation and exploration

MCTS algorithm

```
def MCTS(state)
    tree ← NOTE(state)
    while IS-TIME-REMAINING() do // within computational budget
        leaf ← SELECT(tree)  // say UCB1
        child ← EXPAND(leaf)
        result ← SIMULATE(child) // self-play
        BACK-PROPAGATE(result, child)
    return the move in ACTION(state) whose node has highest number of playouts
```

Properties of MCTS#

- *Aheuristic*: don't need domain-specific knowledge (heuristic), applicable to any domain modeled by a tree

- *Anytime*: all values up-to-date by backed up immediately allow to return an action from the root at any moment in time

- *Asymmetric*: The selection allows to favor of more promising nodes (without allowing the selection probability of the other nodes to converge to zero), leading to an asymmetric tree over time

Example: Alpha0$^+$

Go

– MCTS had a dramatic effect on narrowing this gap but is competitive only on small boards (say, 9×9), or weak amateur level players on the standard 19×19 board

– Pachi: open-source Go program, using MCTS, ranked at amateur 2 *dan* on KGS, that executes 100,000 simulations per move

Ref. Rimmel. A *et al.*, *Current Frontiers in Computer Go*, IEEE Trans. Comp. Intell. AI Games, vol. 2, no. 4, 2010

Example: Alpha0

Alpha0 algorithm design

1. combine deep learning in an MCTS algorithm
   – a single DNN (deep neural network) for both
     police for breadth pruning, and
     value for depth pruning
2. in each position, an MCTS search is executed guided by the DNN
   with data by self-play reinforcement learning
   without human knowledge w/o the game rules (prior know.)
3. asynchronous multi-threaded search that executes simulations
   on CPUs, and computes DNN in parallel on GPUs

Motivation: evaluation function $\Leftarrow$ stochastic simulation $\Leftarrow$ **deep learning**
(see later in machine learning)

Example: Alpha0

Implementation

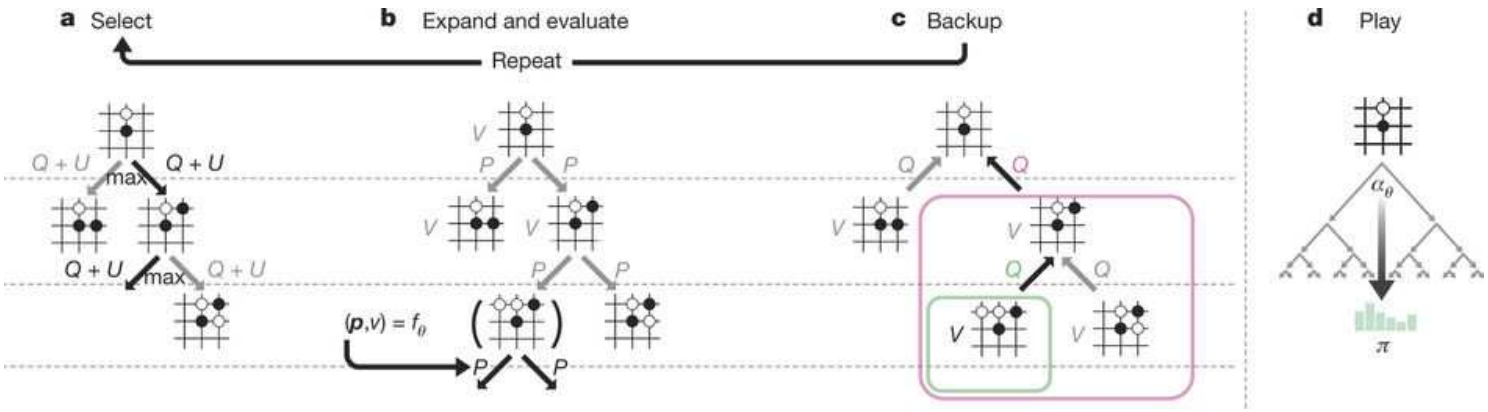
Raw board representation: $19 \times 19 \times 17$

historic position $s_t = [X_t, Y_t, X_{t-1}, Y_{t-1}, \cdots, X_{t-7}, Y_{t-7}, C]$

Reading Silver D, et. al., *Mastering the game of Go without human knowledge*, Nature 550, 354-359, 2017; or
(Silver, D, et. al., *A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play*, Science 07 Dec 2018: Vol. 362, Issue 6419, pp. 1140-1144
Schrittwieser J et al., Mastering atari, go, chess and shogi by planning with a learned model, Nature 588, 604-612, 2020)

Alpha0 algorithm: MCTS+

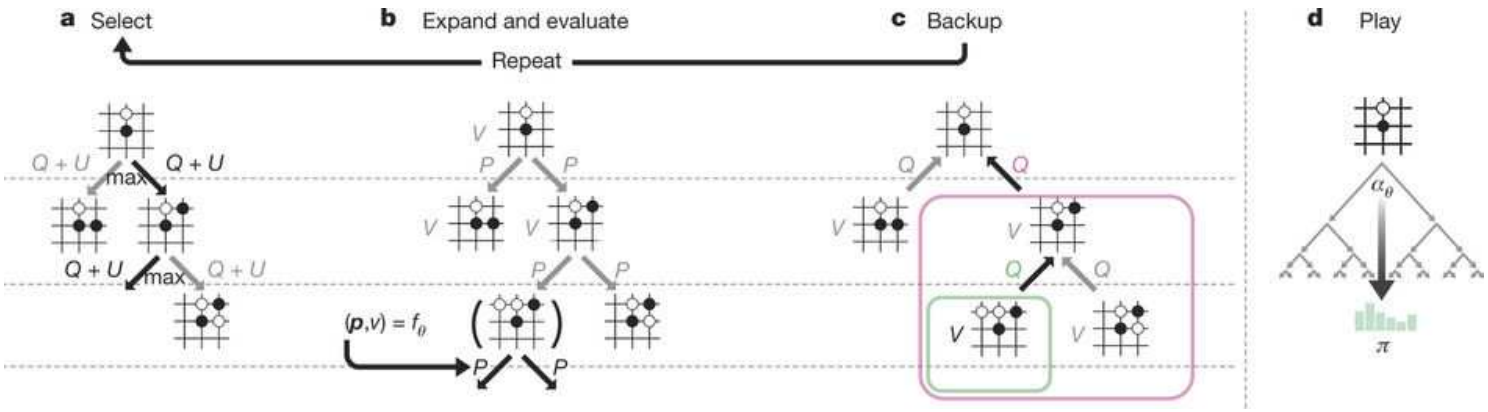
Improvement MCTS: using a DNN and self-play



a. Selecting s with maximum value $Q = \frac{1}{N(s,a)} \Sigma_{s\prime|s,a\to s\prime} V(s\prime)$ + an upper confidence bound $U \propto \frac{P(s,a)}{1+N(s,a)}$ (stored prior probability P and visit count N)
(each simulation traverses the tree; don't need rollout)

Alpha0 algorithm: MCTS$^+$

Improvement MCTS: using a DNN and self-play



b. Expanding leaf and evaluating s by DNN $(P(s,\cdot), V(s)) = f_\theta(s)$

c. Updating Q to track all V in the subtree

d. Once completed, search probabilities $\pi \propto N(s,a)^{1/\tau}$ are returned

(τ is a hyperparameter)

Alpha0 pseudocode$^+$

```
def ALPHA0(state)
    inputs: rules, the game rules // maybe omit
            scores, the game scores
            board, the board representation
            // historical data and color for players
    create root node with state s_0, initially random play

    while within computational budget do
        α_θ ← MCTS(s(f_θ))
        a ← MOVE(s, α_θ)
    return a(BestMove(α_θ))
```

Alpha0 pseudocode

def MCTS($state$, f_θ)
  inputs: $tree$, the search tree /*First In Last Out*/
  $P(s, a)$: prior probability, each edge $(s,\ a)$ in tree
  $N(s, a)$: visit count
  $Q(s, a)$: action value

  while within computational budget **do**
    $f_\theta \leftarrow \text{DNN}(s_t,\pi_t,z_t)$
    $(P(s',\ \cdot), V(s')) \leftarrow f_\theta$
    $U(s,\ a) \leftarrow \text{POLICY}(P(s',\cdot),s_0)$
    $Q(s,\ a) \leftarrow \text{VALUE}(\ V(s'),s_0)$
    $s' \leftarrow \text{MAX}(U(s, a) + Q(s, a))$
    $\text{BACKUP}(s', Q)$
  return $\alpha_\theta(\text{BESTCHILD}(s_0))$

Some source codes can be found on Github, say LeelaZero

Complexity of Alpha0 algorithm$^+$

Go/Chess are NP-hard ("almost" in PSPACE) problems

Alpha0 does not reduce the complexity of Go/Chess, but
- outperforms humans in the complexity
- – practical approach to handle NP-hard problems
- obeys the complexity of MCTS and machine learning
- – performance improvements from deep learning by bigdata and computational power

Alpha0 toward N-steps optimization??
- if so, a draw on Alpha0 vs. Alpha0 for Go/Chess

Alpha0 vs. deep blue#

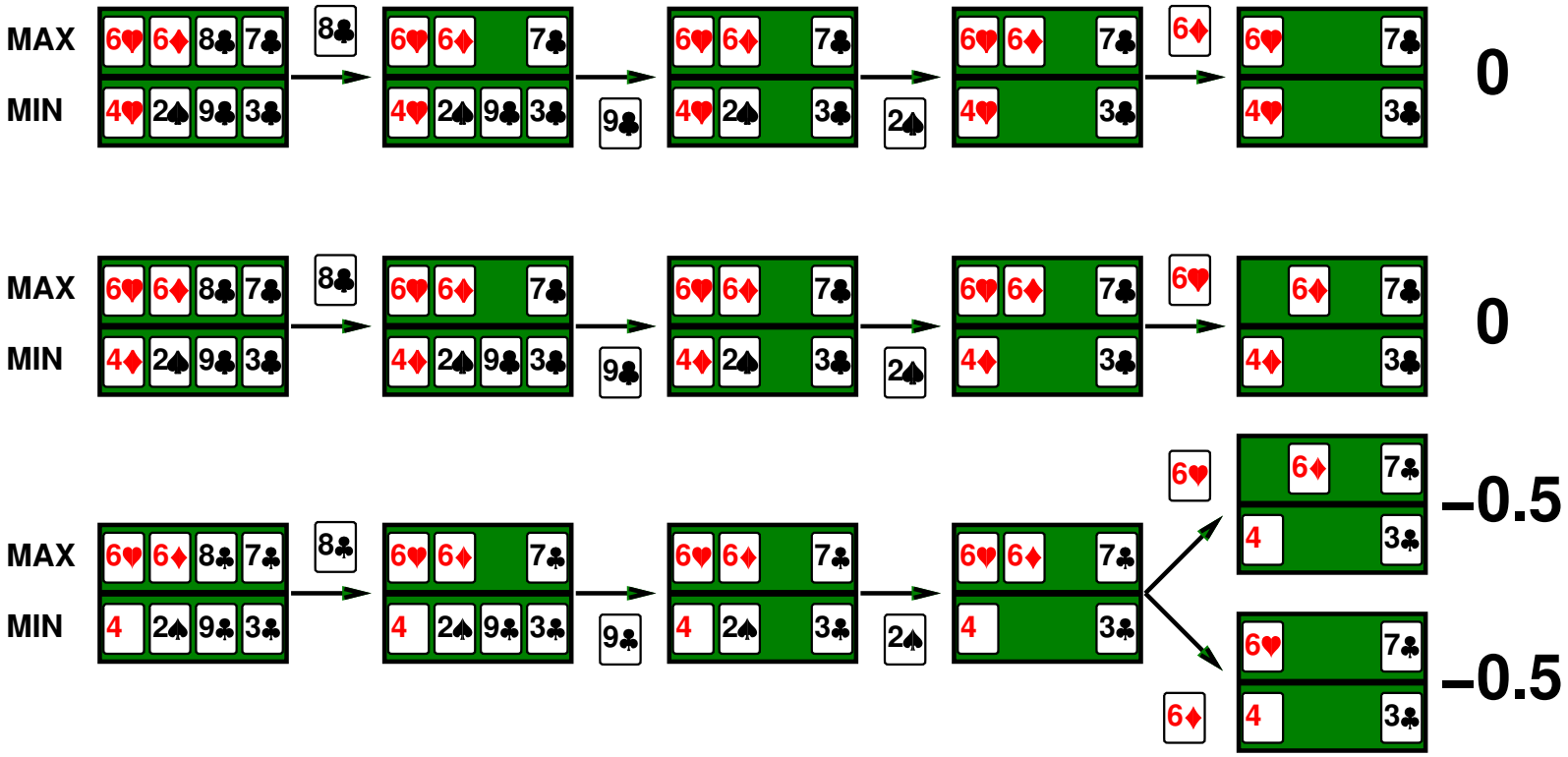
Alpha0Go/Chess exceeded the performance of all other Go/Chess programs, demonstrating that DNN provides a viable alternative to Monte Carlo simulation

Evaluated thousands of times fewer positions than Deep Blue did in match

– while Deep Blue relied on a handcrafted evaluation function, Alpha0's neural network is trained purely through self-play reinforcement learning

Example: Card*

Four-card bridge/whist/hearts hand, MAX to play first



Imperfect information games in practice

Poker: surpass human experts in the game of heads-up no-limit Texas hold'em, which has over 10^{160} decision points

– DeepStack: beat top poker pros in limit Texas hold'em in 2008, and defeated a collection of poker pros in heads-up no-limit in 2016

– Libratus/Pluribus: two-time champion of the Annual Computer Poker Competition in heads-up no-limit, defeated a team of top heads-up no-limit specialist pros in 2017

– ReBeI: achieved superhuman performance in heads-up no-limit in 2020, extended AlphaZero to imperfect information game by Nash equilibrium (with knowledge)

Bridge: more difficult than Poker

– NooK (startup NukkAI) won 67 or 83% of the 80 sets with 8 world champions (no deception) in 2022

a hybrid of rules-based and deep learning systems ("white box" or "neurosymbolic", explainability)

Imperfect information games in practice

StarCraft II (real-time strategy games): Deep Mind AlphaStar 5-0 defeated a top professional player in 2018

Mahjong: Suphx (Japanese Mahjong) – rated above 99.99% of top human players in the Tenhou platform in 2020

Stratego: the game tree on the order of 10^{535} nodes (10175 times larger than that of Go)
DeepNash got up to a human expert level from scratch in 2022

Imperfect information games involve obstacles not present in classic board games like go, but

which are present in many real-world applications, such as negotiation, auctions, security, weather prediction and climate modeling etc.

Online search$^+$

Offline search algorithms compute a complete solution before exec.

vs.

online search ones interleave computation and action (processing input data as they are received)

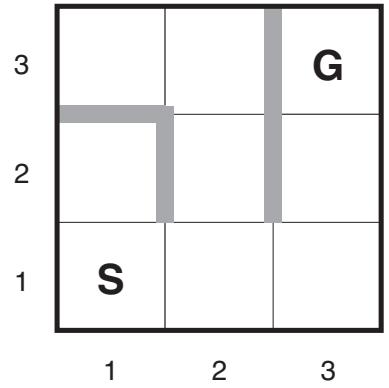
– necessary for unknown environment

(dynamic or semidynamic, and nonderterministic domains)

$\Leftarrow$ exploration problem

Example: maze problem

An online search agent solves problems by executing actions, rather than by pure computation (offline)



The competitive ratio – the total cost of the path that the agent actually travels (online cost) / that the agent would follow if it knew the search space in advance (offline cost) $\Leftarrow$ as small as possible

Online search expands nodes in local order, say, DepthFirst and HillClimbing have exactly this property

Online search agents#

def ONLINE-DFS-AGENT($problem$,s')
  persistent: s',a, the previous state and action,initially null
    $result$, a table mapping (s, a) to s',initially empty
    $untried$, a table mapping s to a list of untried actions
    $unbacktracked$, a table mapping s to a list of states never backtracked to
  if $problem$.IS-GOAL(s') **then reture** $stop$
  if s' is a new state (not in $untried$) **then** $untried[s'] \leftarrow problem$.ACTION($s'$)
  if s is not null **then**
    $result[s, a] \leftarrow s'$
    add s to the front of $unbacktraked[s']$
  if $untried[s']$ is empty **then**
    if $unbacktracked[s']$ is empth **then return** $stop$
    else $a \leftarrow$ an action b s.t. $result[s', b]$=POP($unbacktracked[s']$)
  else $a \leftarrow$ POP($untried[s']$)
  $s \leftarrow s'$
  return a

Metaheuristic search*

Metaheuristic: higher-level procedure or heuristic

to find a heuristic for optimization $\Leftarrow$ local seach

e.g., simulated annealing

Metalevel vs. object-level state space

Each state in a metalevel state space captures the internal state of a program that is searching in an object-level state space

An agent can **learn** how to search better

– metalevel learning algorithm can learn from experiences to avoid exploring unpromising subtrees

– learning is to minimize the total cost of problem solving

especially, learning admissible heuristics from examples, e.g., 8-puzzle

Tabu search

Tabu search is a metaheuristic employing local search

resolving stuck in local minimum or plateaus

e.g., HILL-CLIMBING

Tabu (forbidden) uses memory structures (*tabulist*) that describe the visited solutions or user-provided rules

– to discourage the search from coming back to previously-visited solutions

– depended on certain short-term periods or violated a rule (marked as "tab") to avoid a repeat

Tabu search algorithm#

```
def TABU-SEARCH( solution)
    persistent: tabulist, a memory structure for states visited, initially empty
    solution-best ← solution
    while not STOPPING-CONDITION
        CANDIDATE ← null
        best.CANDIDATE-LIST ← null
        for solution.CANDIDATE in solution.NEIGHBORHOOD
          if not tabulist.CONTAINS(solution.CANDIDATE) and
              FITNESS(solution.CANDIDATE)> FITNESS(best.CANDIDATE)
              best.CANDIDATE ← solution.CANDIDATE
        solution ← best.CANDIDATE
        if FITNESS(best.CANDIDATE) > FITNESS(solution-best)
            solution-best ← best.CANDIDATE
        tabuist.PUSH(best.CANDIDATE)
        if tabulist.SIZE > MaxTabuSize
            tabulist.REMOVEFIRST()
    return solution-best
```

Searching search*

Researchers have taken inspiration for search (and optimization) algorithms from a wide variety of fields

- metallurgy (simulated annealing)
- biology (genetic algorithms)
- economics (market-based algorithms)
- entomology (ant colony)
- neurology (neural networks)
- animal behavior (reinforcement learning)
- mountaineering (hill climbing)
- politics (struggle forms), and others

Is there a general problem solver to generality of intelligence?? – NO