Artificial Intelligence

Lecturer 3 - Advanced search methods

Brigitte Jaumard
Dept of Computer Science and Software Engineering
Concordia University
Montreal (Quebec) Canada

Outline

- Memory-bounded heuristic search
- Hill-climbing search
- Simulated annealing search
- Local beam search
- Game and search
- Alpha-beta pruning

2

MEMORY-BOUNDED HEURISTIC SEARCH

Memory-bounded heuristic search

- Some solutions to A* space problems (maintain completeness and optimality)
  - Iterative-deepening A* (IDA*)
    - Here cutoff information is the *f*-cost (*g*+*h*) instead of depth
  - Recursive best-first search(RBFS)
    - Recursive algorithm that attempts to mimic standard best-first search with linear space.
  - (simple) Memory-bounded A* ((S)MA*)
    - Drop the worst-leaf node when memory is full

Iterative Deepening A*

- Iterative Deepening version of A*
  - use threshold as depth bound
    - To find solution under the threshold of *f*(.)
  - increase threshold as minimum of *f*(.) of
    - previous cycle
- Still admissible
- Same order of node expansion
- Storage Efficient – practical
  - but suffers for the real-valued *f*(.)
  - large number of iterations

Depth-Limited Tree Search

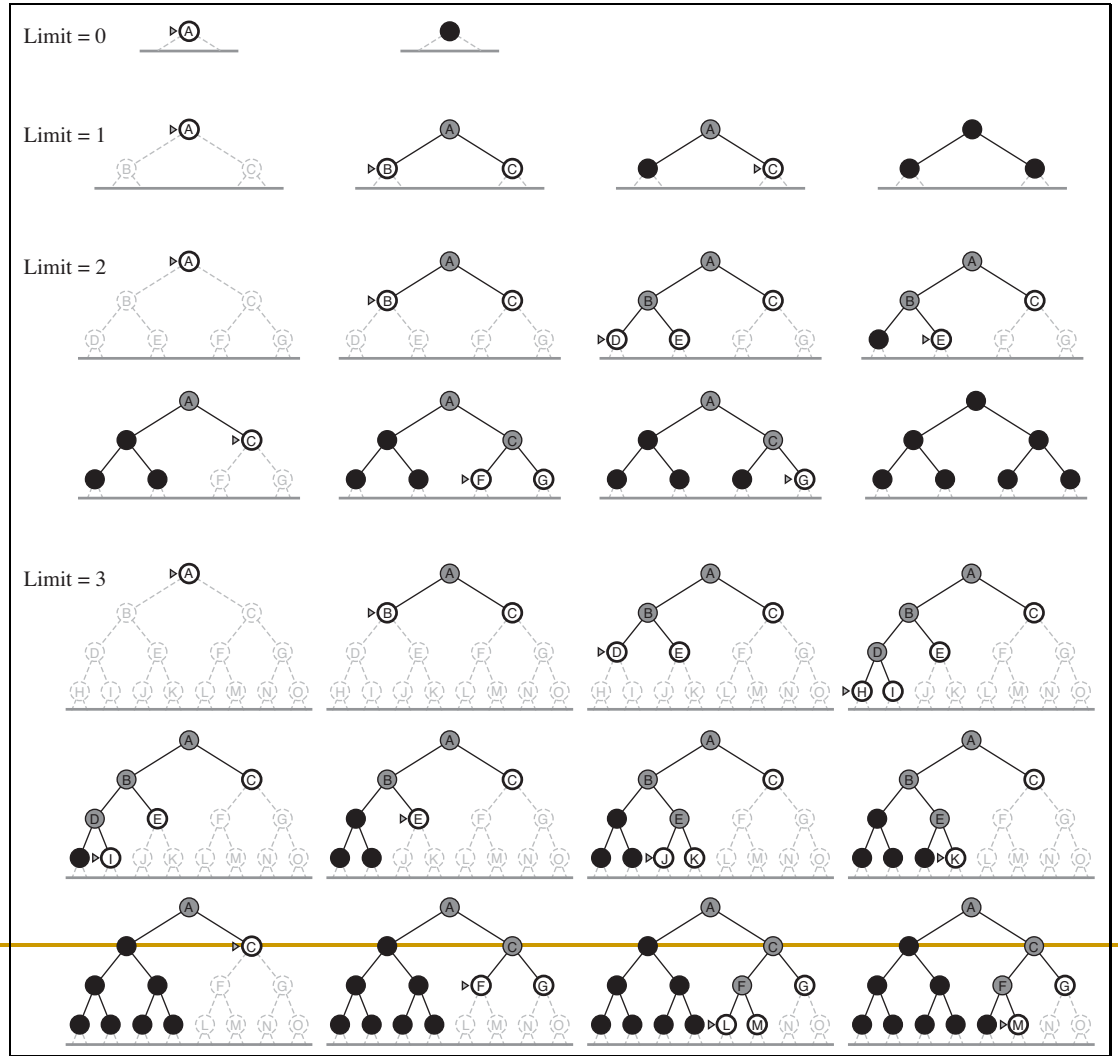
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Iterative Deepening Algorithm

- Repeatedly applies depth-limited search with increasing limits
- Terminates when a solution is found or if the depth-limited search returns *failure*, meaning no solution exists.

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  for *depth* = 0 **to** ∞ **do**
    result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ cutoff **then return** *result*

Example: four iterations of iterative deepening search on a binary tree



Recursive Best-First Search (RBFS)

- A variation of Depth-First Search (DFS)
- Keep track of *f*-value of the best alternative path
- Unwind if *f*-value of all children exceed its best alternative
- When unwind, store *f*-value of best child as its *f*-value
- When needed, the parent regenerate its children again.

Recursive Best-First Search (RBFS)

function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **return** a solution or failure
    return RBFS(*problem*,MAKE-NODE(INITIAL-STATE[*problem*]),∞)

function RBFS (*problem, node, f_limit*) **return** a solution or failure and a new *f-cost* limit
    if GOAL-TEST[*problem*](STATE[*node*]) **then return** *node*
    successors ← EXPAND(*node, problem*)
    if *successors* is empty then return failure, ∞
    for each *s* **in** *successors* **do**
        f [*s*] ← max(*g(s)* + *h(s)*, *f* [*node*])
    repeat
        best ← the lowest *f*-value node in *successors*
        if *f* [*best*] > *f_limit* **then return** failure, *f* [*best*]
        alternative ← the second lowest *f*-value among *successors*
        result, *f* [*best*] ← RBFS*(problem, best,* min(*f_limit, alternative*))
        if *result* ≠ failure **then return** *result*

Recursive Best-First Search (RBFS)

- Keeps track of the *f*-value of the best-alternative path available.
  - If current *f*-values exceeds this alternative *f*-value then backtrack to alternative path.
  - Upon backtracking change *f*-value to best *f*-value of its children.
  - Re-expansion of this result is thus still possible.

RBFS evaluation

- RBFS is a bit more efficient than IDA*
  - Still excessive node generation (mind changes)
- Like A*, **optimal if** *h(n)* is admissible
- **Space complexity** is *O(bd)*.
  - IDA* retains only one single number (the current *f*-cost limit)
- **Time complexity** difficult to characterize
  - Depends on accuracy if *h(n)* and how often best path changes.
- IDA* and RBFS suffer from ***too little*** memory.

(Simplified) Memory-bounded A* (SMA*)

- Use all available memory.
  - I.e. expand best leafs until available memory is full
  - When full, SMA* drops worst leaf node (highest *f*-value)
  - Like RBFS, we remember the best descendant in the branch we delete
- What if all leafs have the same *f*-value?
  - Same node could be selected for expansion and deletion.
  - SMA* solves this by expanding *newest* best leaf and deleting *oldest* worst leaf.
- The deleted node is regenerated when all other candidates look worse than the node.
- SMA* is complete if solution is reachable, optimal if optimal solution is reachable.
- Time can still be exponential.

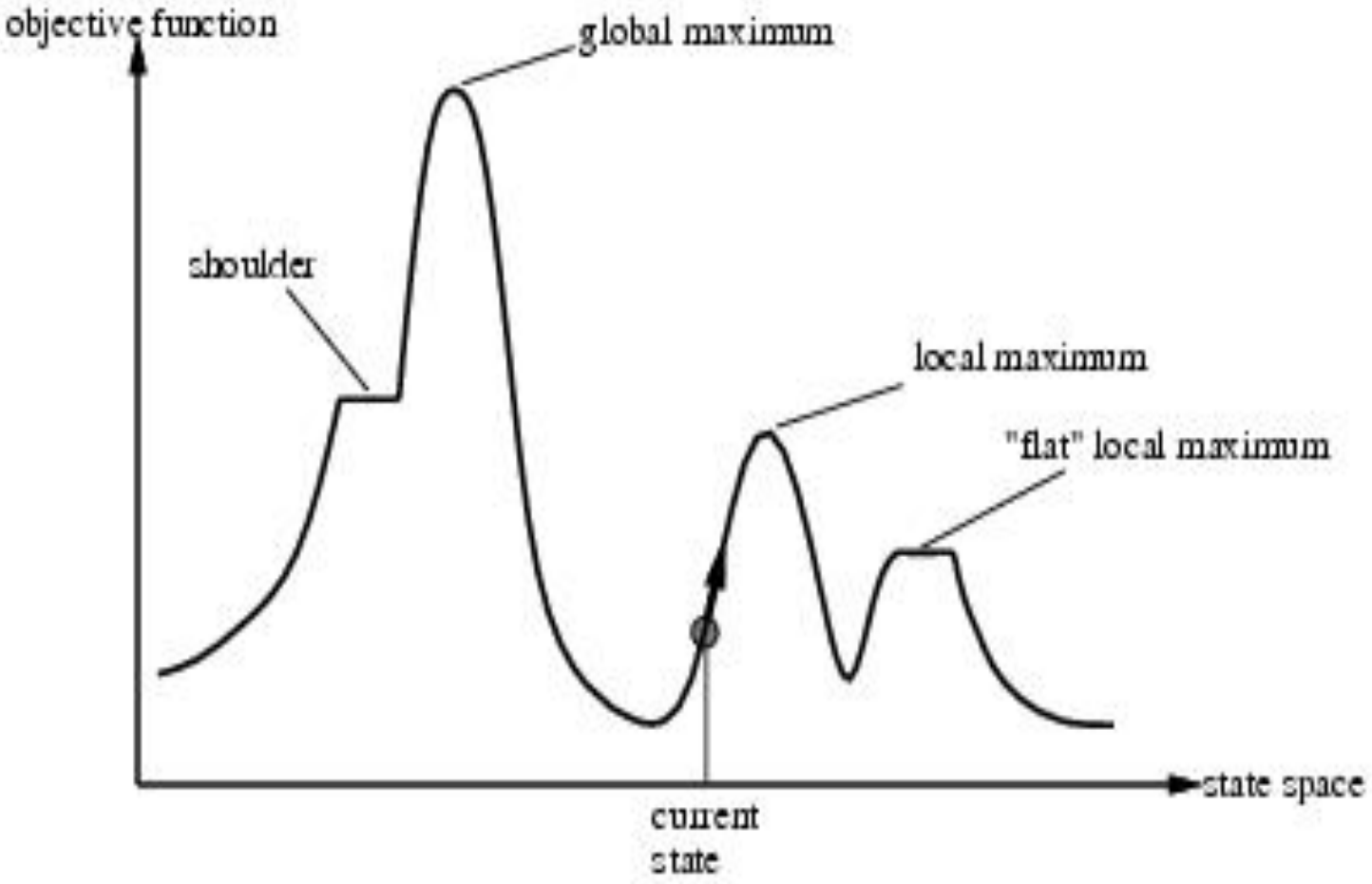
LOCAL SEARCH ALGORITHMS

Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., *n*-queens
- In such cases, we can use local search algorithms

- Local search= use single current state and move to neighboring states.
- Advantages:
  - Use very little memory
  - Find often reasonable solutions in large or infinite state spaces.
- Are also useful for pure optimization problems.
  - Find best state according to some *objective function*.

Local search and optimization



Example: *n*-queens

- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal

HILL-CLIMBING SEARCH

Hill-climbing search

- Simple, general idea:
  - Start wherever
  - Always choose the best neighbor
  - If no neighbors have better scores than current, quit
- Hill climbing does not look ahead of the immediate neighbors of the current state.
- Hill-climbing chooses randomly among the set of best successors, if there is more than one.
- Some problem spaces are great for hill climbing and others are terrible.

Hill-climbing search

function HILL-CLIMBING(*problem*) **return** a state that is a local maximum

input: *problem*, a problem

local variables: *current*, **a node.**

neighbor, **a node.**

current ← MAKE-NODE(INITIAL-STATE[*problem*])

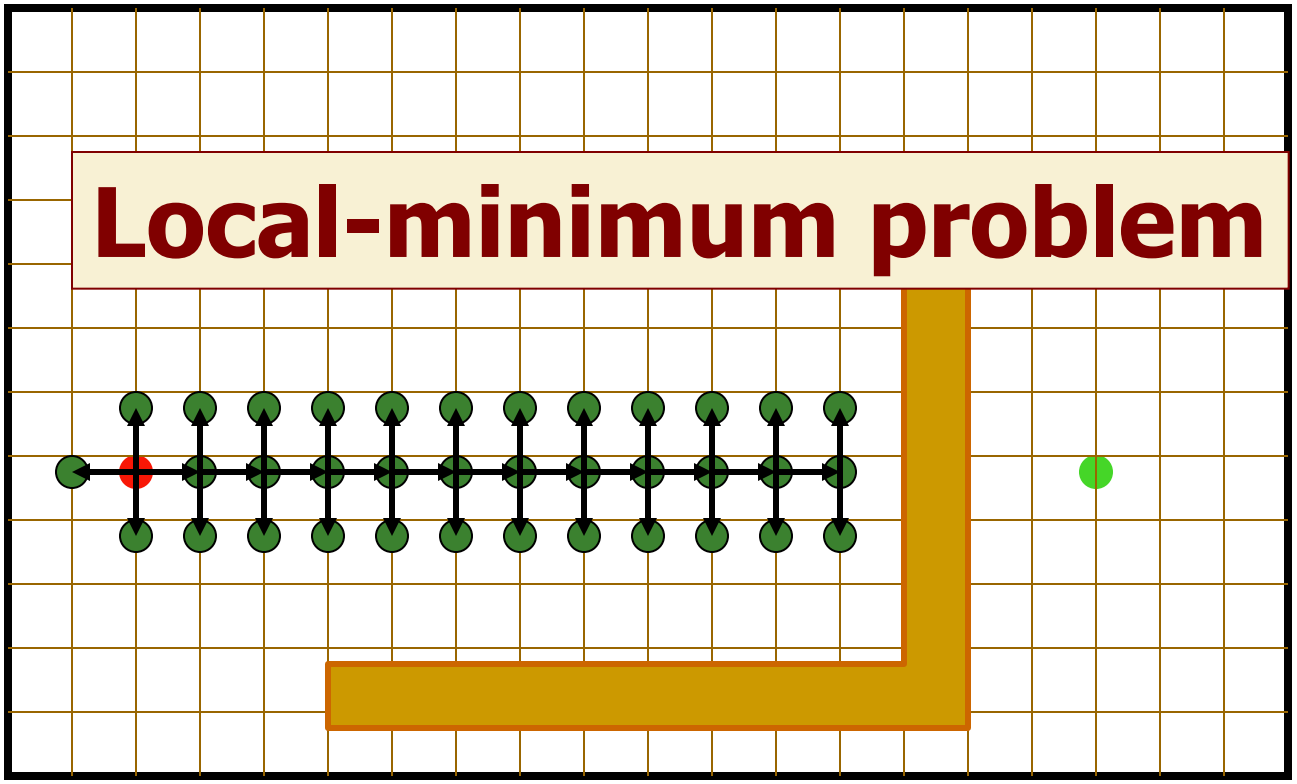
loop do

neighbor ← a highest valued successor of *current*

if VALUE [*neighbor*] < VALUE[*current*] **then return** STATE[*current*]

current ← *neighbor*

Robot Navigation

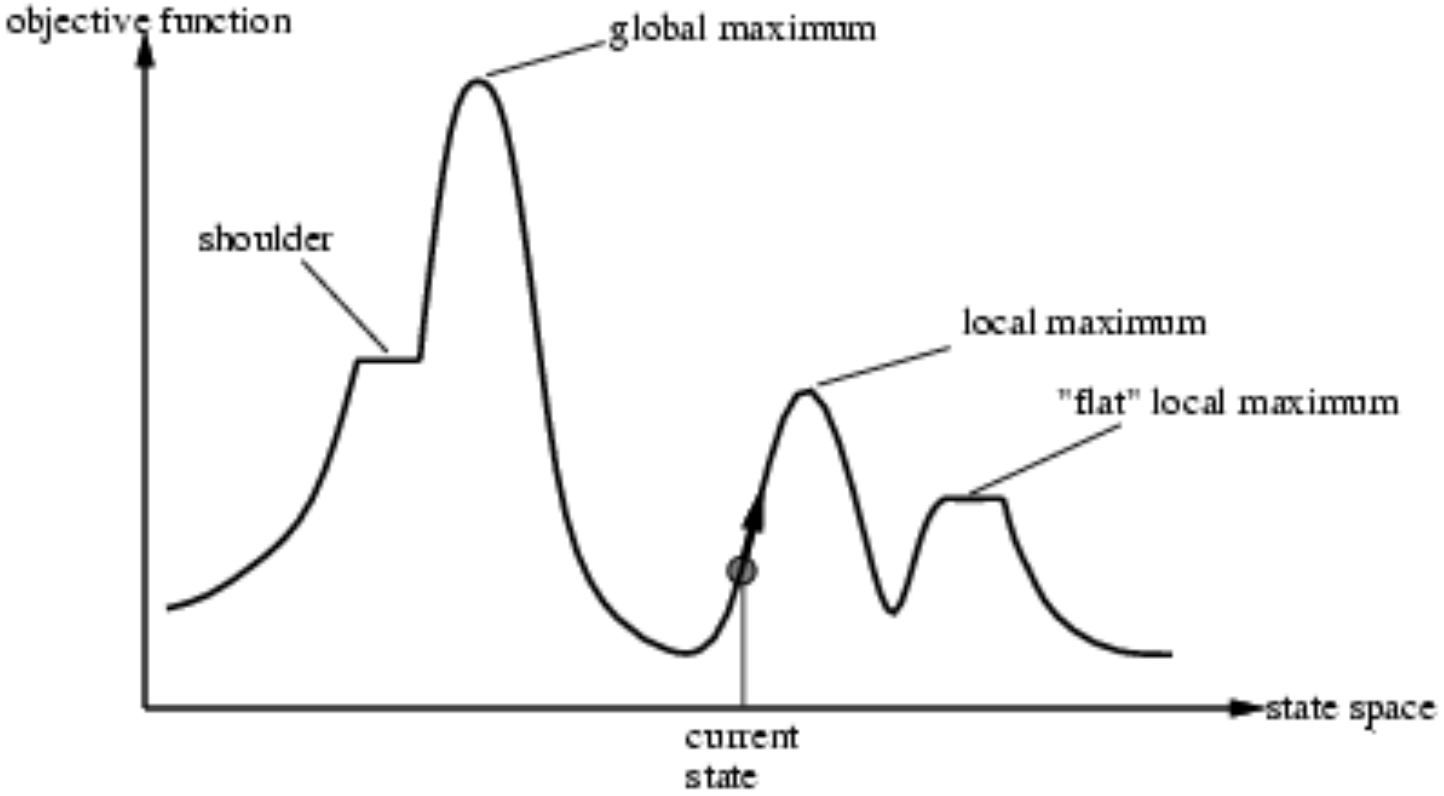


f(N) = h(N) = straight distance to the goal

Drawbacks of hill climbing

- Problems:
  - **Local Maxima**: depending on initial state, can get stuck in local maxima
  - **Plateaus:** the space has a broad flat region that gives the search algorithm no direction (random walk)
  - **Ridges:** flat like a plateau, but with dropoffs to the sides; steps to the North, East, South and West may go down, but a combination of two steps (e.g. N, W) may go up

- Introduce randomness



Hill-climbing variations

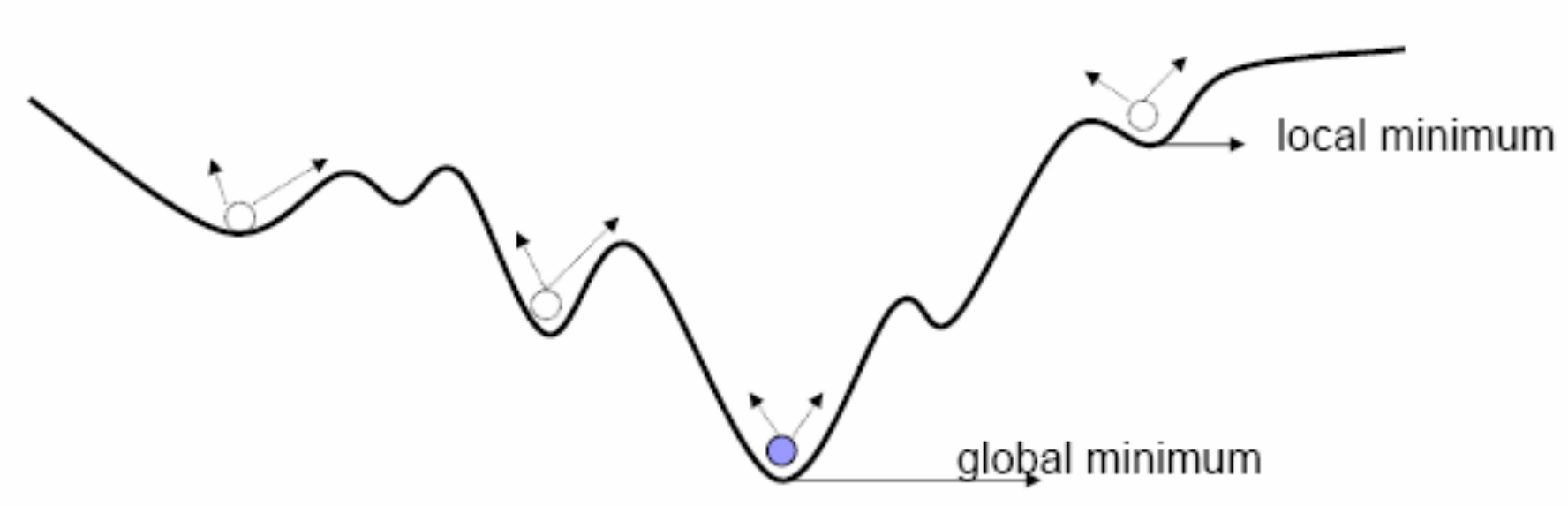
- Stochastic hill-climbing
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.
- First-choice hill-climbing
  - Stochastic hill climbing by generating successors randomly until a better one is found.
- Random-restart hill-climbing
  - Tries to avoid getting stuck in local maxima.
  - If at first you don't succeed, try, try again…

SIMULATED ANNEALING SEARCH

Simulated Annealing

- Simulates slow cooling of annealing process
- Applied for combinatorial optimization problem by S. Kirkpatric (‘83)
- **What is annealing?**
  - Process of slowly cooling down a compound or a substance
  - Slow cooling let the substance flow around → thermodynamic equilibrium
  - Molecules get optimum conformation

Simulated annealing



gradually decrease shaking to make sure the ball escape from local minima and fall into the global minimum

Simulated annealing

- Escape local maxima by allowing “bad” moves.
  - Idea: but gradually decrease their size and frequency.
- Origin; metallurgical annealing
- Implement:
  - Randomly select a move instead of selecting best move
  - Accept a bad move with probability less than 1 ($p<1$)
  - p decreases by time
- If *T* decreases slowly enough, best state is reached.
- Applied for VLSI layout, airline scheduling, etc.

Simulated annealing

function SIMULATED-ANNEALING(*problem, schedule*) **return** a solution state

input: *problem*, a problem

schedule, a mapping from time to temperature

local variables: *current*, a node; *next*, a node.

T, a "temperature" controlling the probability of downward steps

current ← MAKE-NODE(INITIAL-STATE[*problem*])

for t ← 1 to ∞ do

T ← *schedule*[*t*]

if *T* = *0* **then return** *current*

next ← a randomly selected successor of *current*

$\Delta E \leftarrow$ VALUE[*next*] - VALUE[*current*]

if $\Delta E > 0$ **then** *current* ← *next*

else *current* ← *next* only with probability $e^{\Delta E / T}$

Similar to hill climbing, but a **random** move instead of best move

case of improvement, make the move

Otherwise, choose the move with probability that decreases exponentially with the "badness" of the move.

What's the probability when: T → inf?

What's the probability when: T → 0?

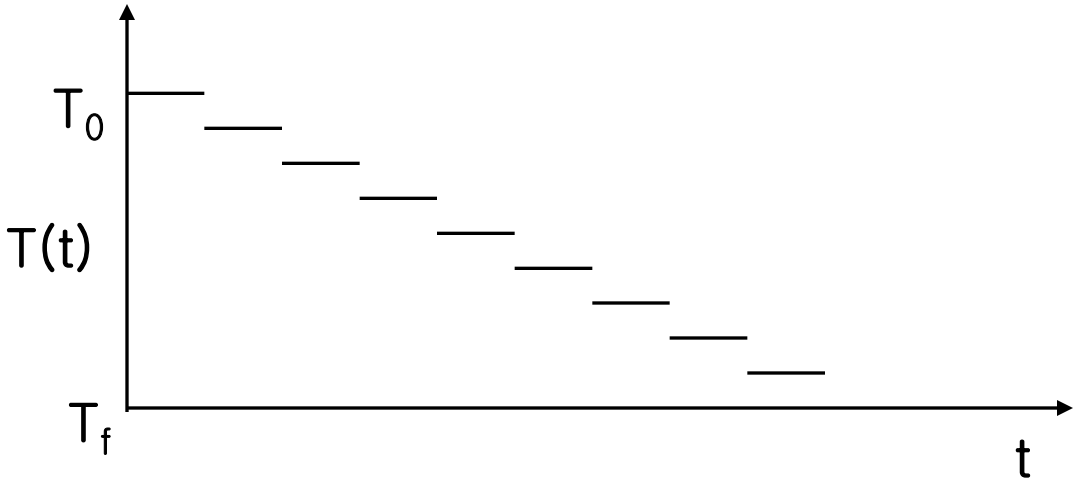
What's the probability when: Δ=0?

What's the probability when: Δ→-∞?

Simulated Annealing parameters

- Temperature T
  - Used to determine the probability
  - High T : large changes
  - Low T : small changes
- Cooling Schedule
  - Determines rate at which the temperature T is lowered
  - Lowers T slowly enough, the algorithm will find a global optimum
- In the beginning, aggressive for searching alternatives, become conservative when time goes by

Simulated Annealing Cooling Schedule



- if T is reduced too fast, poor quality
- Tt = a T(t-1) where a is in between 0.8 and 0.99

Tips for Simulated Annealing

- To avoid of entrainment in local minima
  - Annealing schedule : by trial and error
    - Choice of initial temperature
    - How many iterations are performed at each temperature
    - How much the temperature is decremented at each step as cooling proceeds
- Difficulties
  - Determination of parameters
  - If cooling is too slow →Too much time to get solution
  - If cooling is too rapid → Solution may not be the global optimum

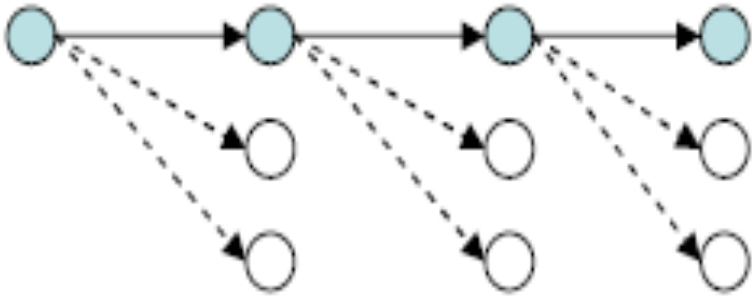
Properties of simulated annealing

- Theoretical guarantee:
  - Stationary distribution: $p(x) \alpha e^{\frac{E(x)}{kT}}$
  - If T decreased slowly enough, will converge to optimal state!
- Is this an interesting guarantee?
- Sounds like magic, but :
  - The more downhill steps you need to escape, the less likely you are to every make them all in a row
  - People think hard about *ridge operators* which let you jump around the space in better ways

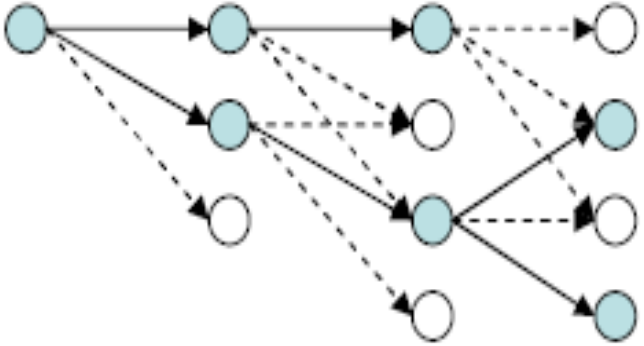
LOCAL BEAM SEARCH

Local beam search

- Like greedy search, but keep K states at all times:
  - Initially: *k* random states
  - Next: determine all successors of *k* states
  - If any of successors is goal → finished
  - Else select *k* best from successors and repeat.



Greedy Search



Beam Search

Local beam search

- Major difference with random-restart search
  - Information is shared among k search threads: If one state generated good successor, but others did not → "come here, the grass is greener!"
- Can suffer from lack of diversity.
  - Stochastic variant: choose k successors at proportionally to state success.
- The best choice in MANY practical settings

GAME AND SEARCH

Games and search

- Why study games?
- Why is search a good idea?

- Majors assumptions about games:
  - Only an agent's actions change the world
  - World is deterministic and accessible

Why study games?

May 1997
Deep Blue - Garry Kasparov
3.5 - 2.5

March 2016, AlphaGo beat Lee Sedol in a five-game match, the first time a computer Go program has beaten a 9-dan professional without handicaps

Machines that teach themselves can produce powerful results, as with DeepMind's series of Go-playing AI systems. | Photo by Google via Getty Images

Why study 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 search a.k.a. *games*

- Why study games?
  - Fun; historically entertaining
  - Interesting subject of study because they are hard
  - Easy to represent and agents restricted to small number of actions

Relation of Games to Search

- Search – no adversary
  - Solution is (heuristic) method for finding goal
  - Heuristics and CSP techniques can find *optimal* solution
  - Evaluation function: estimate of cost from start to goal through given node
  - Examples: path planning, scheduling activities
- Games – adversary
  - Solution is strategy (strategy specifies move for every possible opponent reply).
  - Time limits force an *approximate* solution
  - Evaluation function: evaluate “goodness” of game position
  - Examples: chess, checkers, Othello, backgammon
- Ignoring computational complexity, games are a perfect application for a complete search.
- Of course, ignoring complexity is a bad idea, so games are a good place to study resource bounded searches.

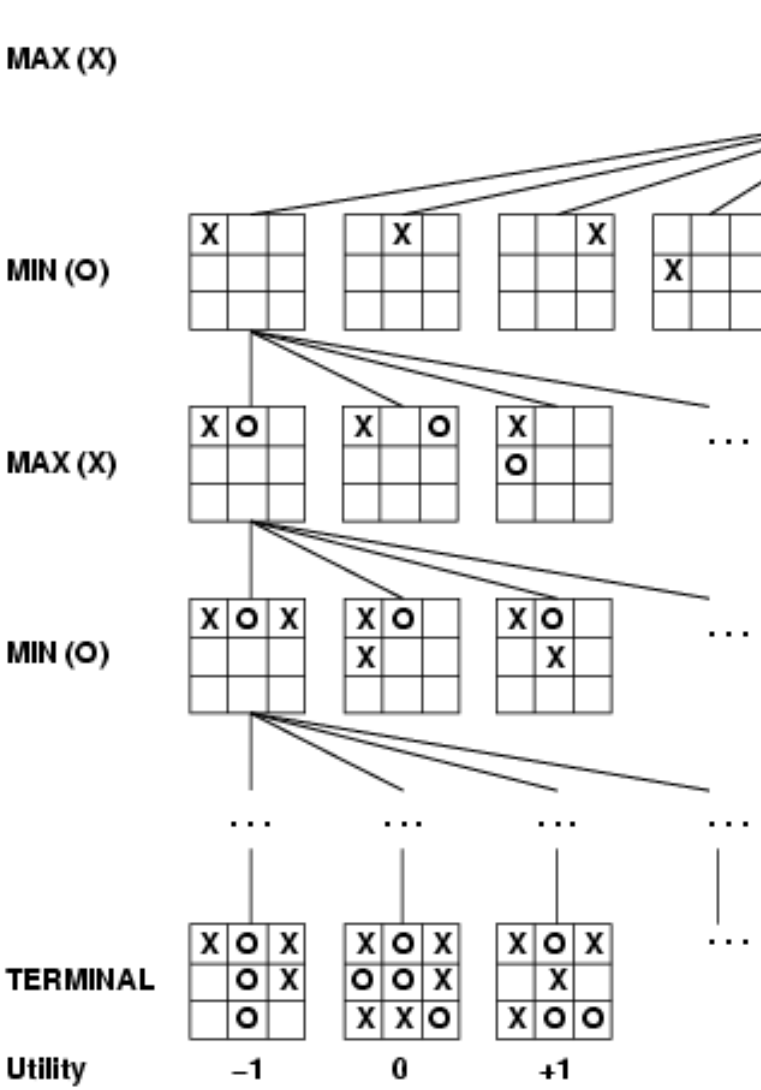
Types of Games

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

Minimax

- 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:** 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.
- Perfect play for deterministic games

Minimax



- From among the moves available to you, take the best one
- The best one is determined by a search using the MiniMax strategy

Optimal strategies

- MAX maximizes a function: find a move corresponding to max value
- MIN minimizes the same function: find a move corresponding to min value

At each step:

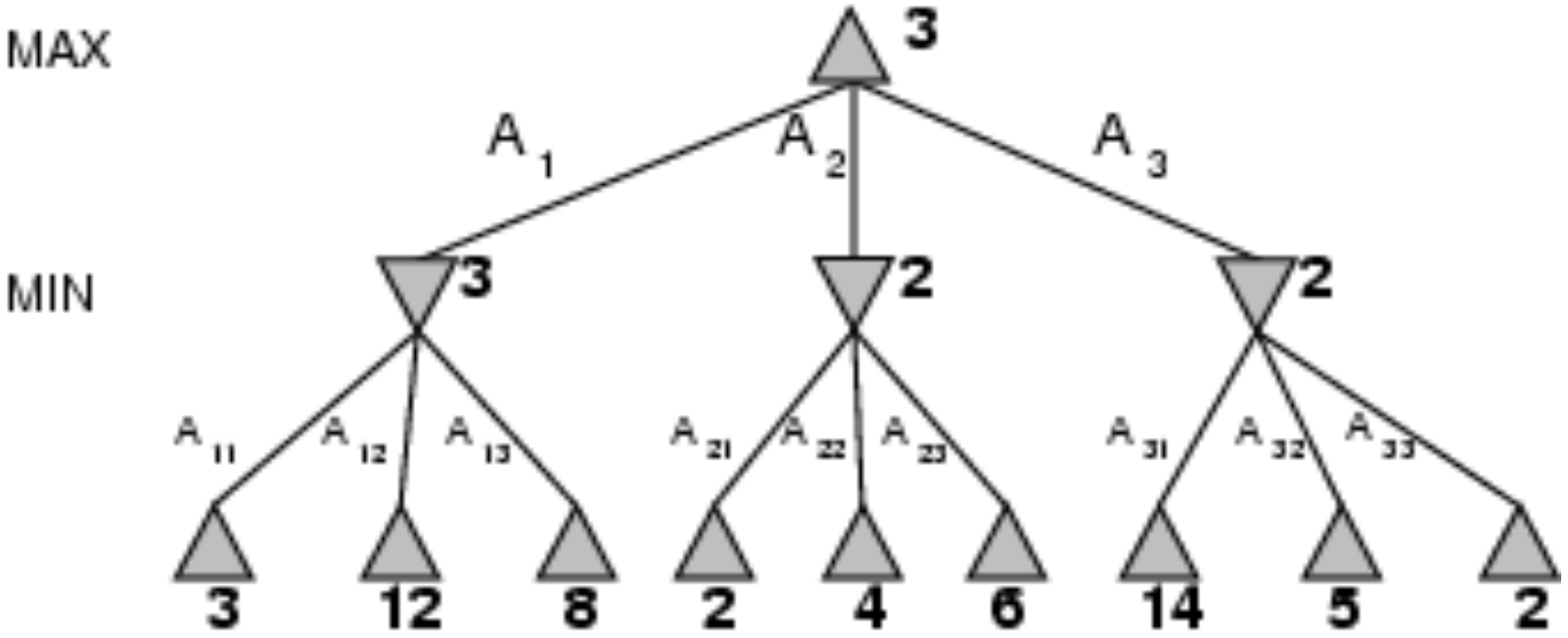
- If a state/node corresponds to a MAX move, the function value will be the maximum value of its childs
- If a state/node corresponds to a MIN move, the function value will be the minimum value of its childs

Given a game tree, the optimal strategy can be determined by using the minimax value of each node:

MINIMAX-VALUE(n)=

UTILITY(n)	If n is a terminal
$\max_{s \in successors(n)}$ MINIMAX-VALUE(s)	If n is a max node
$\min_{s \in successors(n)}$ MINIMAX-VALUE(s)	If n is a min node

Minimax


MAX
MIN
3
A 1
A 2
A 3
3
2
2
A 11
A 12
A 13
A 21
A 22
A 23
A 31
A 32
A 33
3
12
8
2
4
6
14
5
2

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)
- Optimal? Yes (against an optimal opponent)
- 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

Problem of minimax search

- Number of games states is exponential to the number of moves.
  - Solution: Do not examine every node

$\Rightarrow$ Alpha-beta pruning:

- Remove branches that do not influence final decision
- Revisit example …

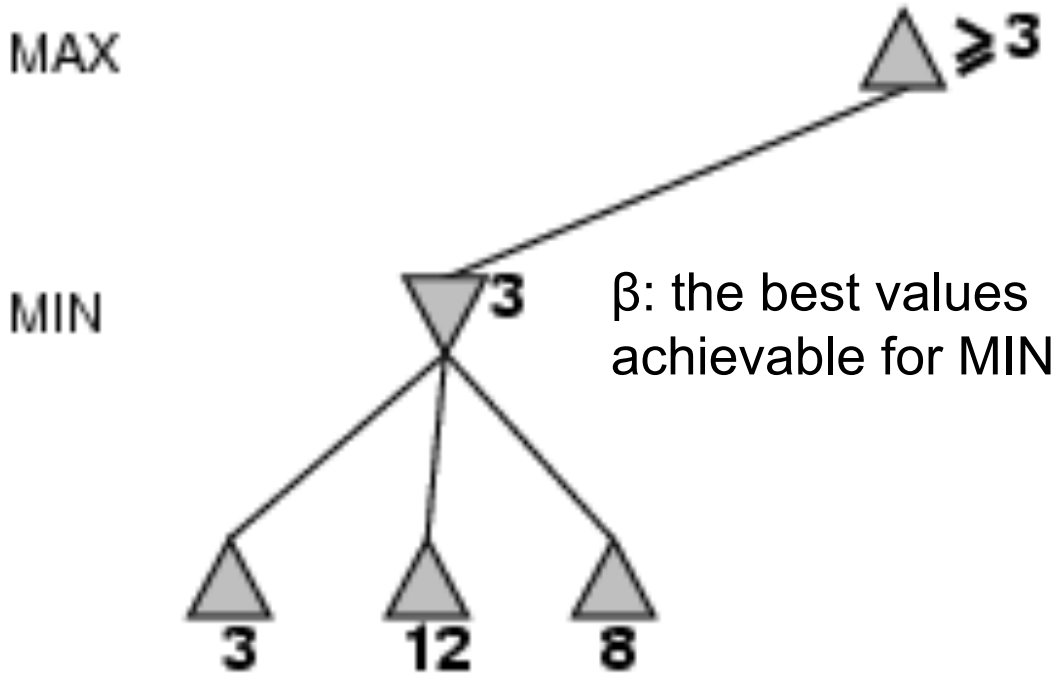
ALPHA-BETA PRUNING

α-β pruning

- **Alpha values**: the best values achievable for MAX, hence the max value so far
- **Beta values**: the best values achievable for MIN, hence the min value so far
- **At MIN level**: compare result V of node to alpha value. If V>alpha, pass value to parent node and BREAK
- **At MAX level**: compare result V of node to beta value. If V<beta, pass value to parent node and BREAK

α-β pruning

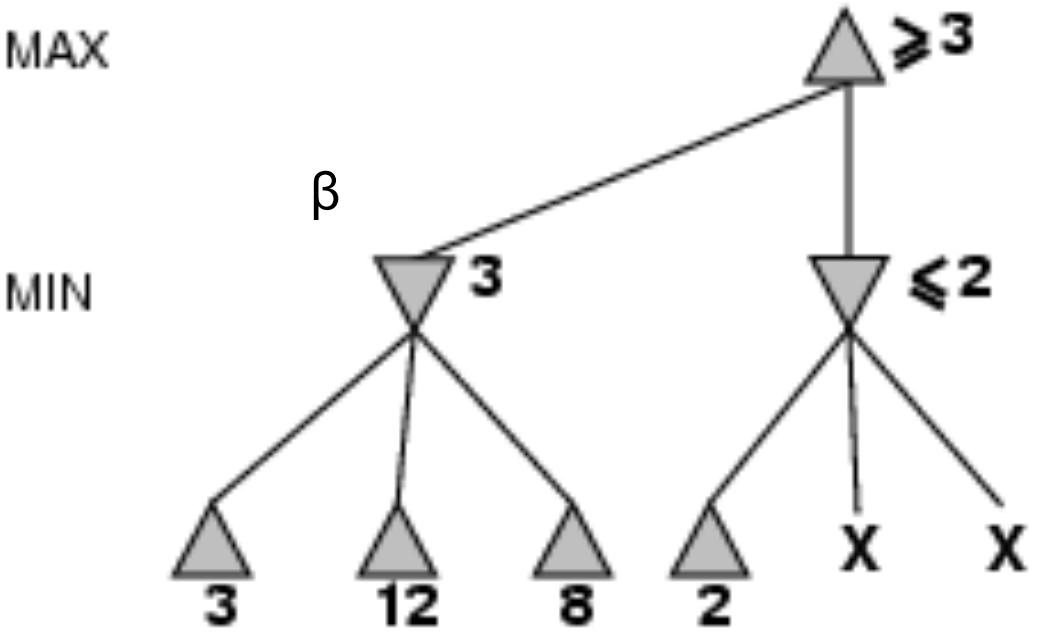
α: the best values achievable for MAX



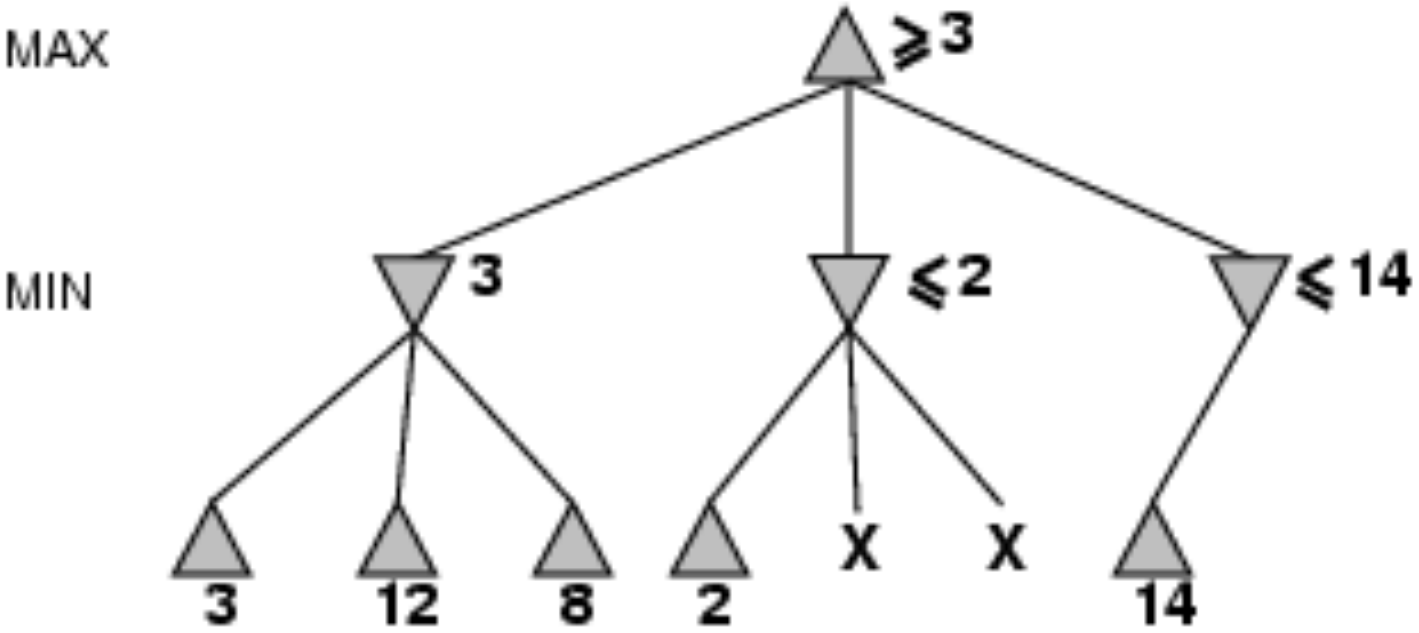
β: the best values achievable for MIN

α-β pruning example

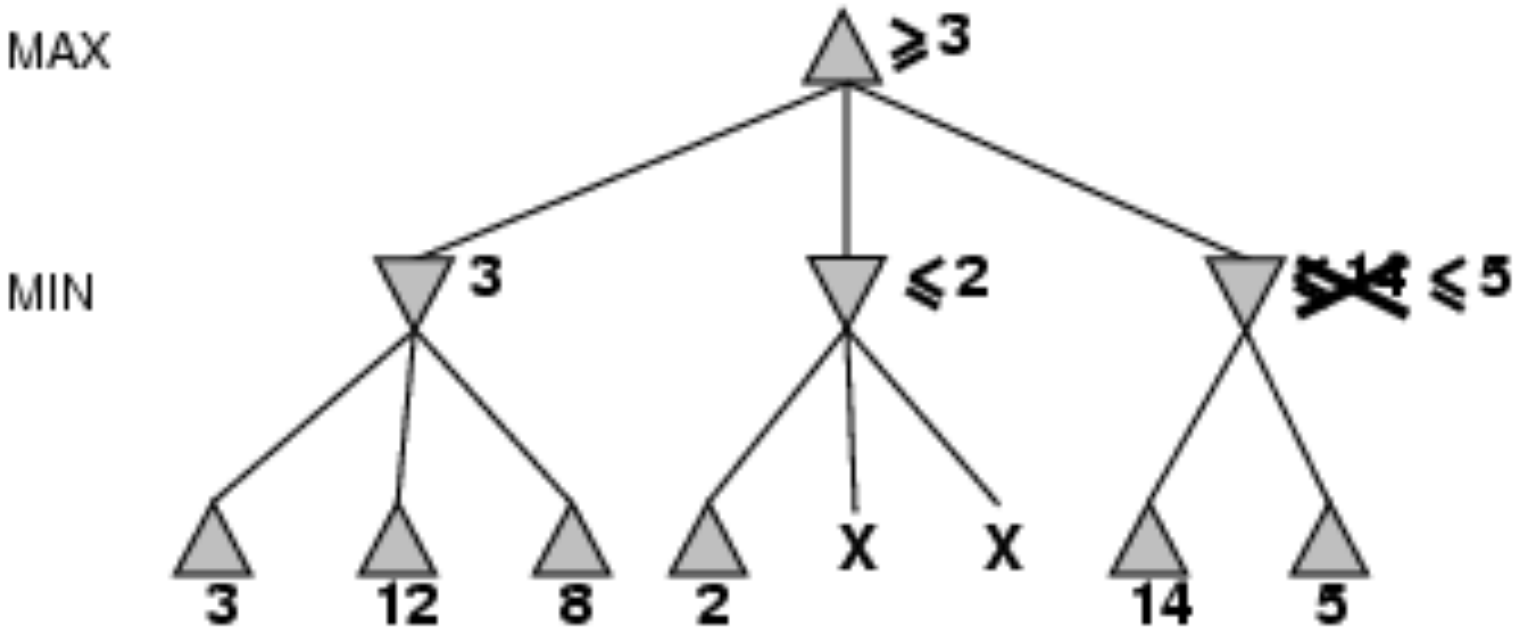
Compare result V of node to β. If V< β, pass value to parent node and BREAK



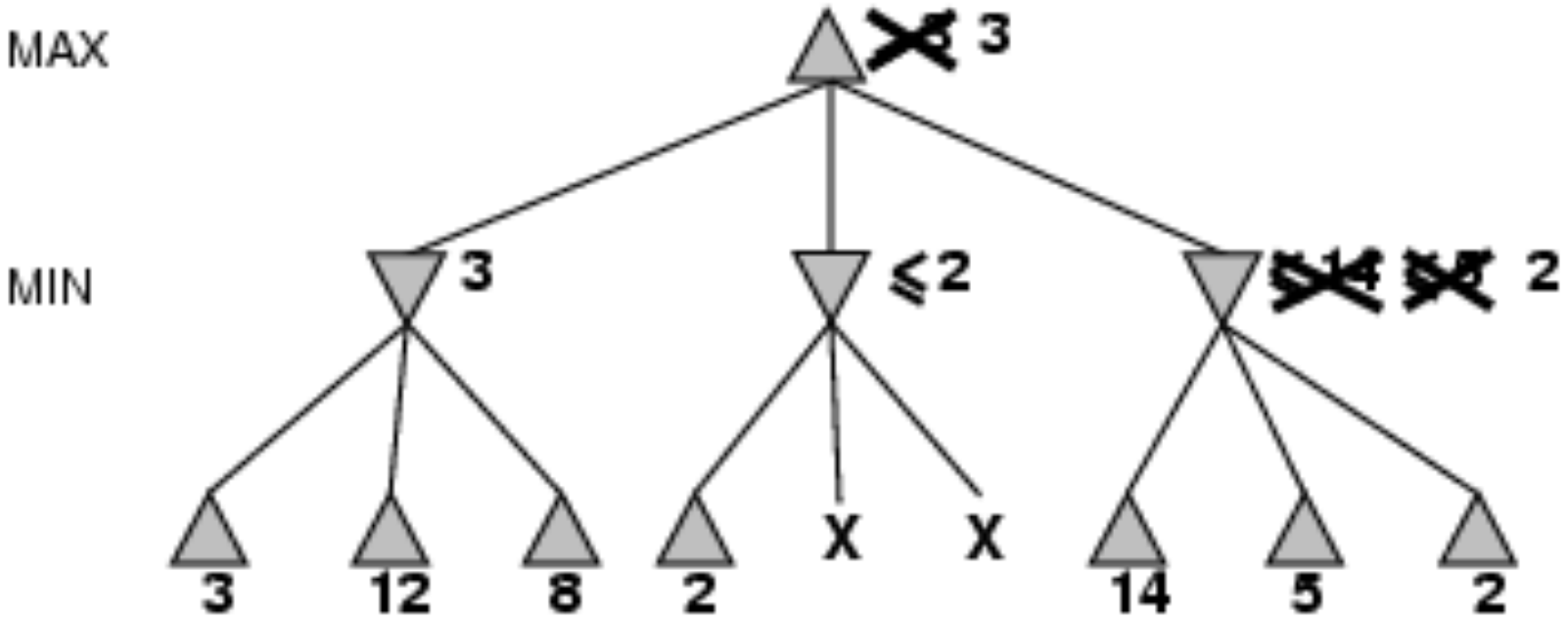
α-β pruning example



α-β pruning example



α-β pruning example



Properties of α-β

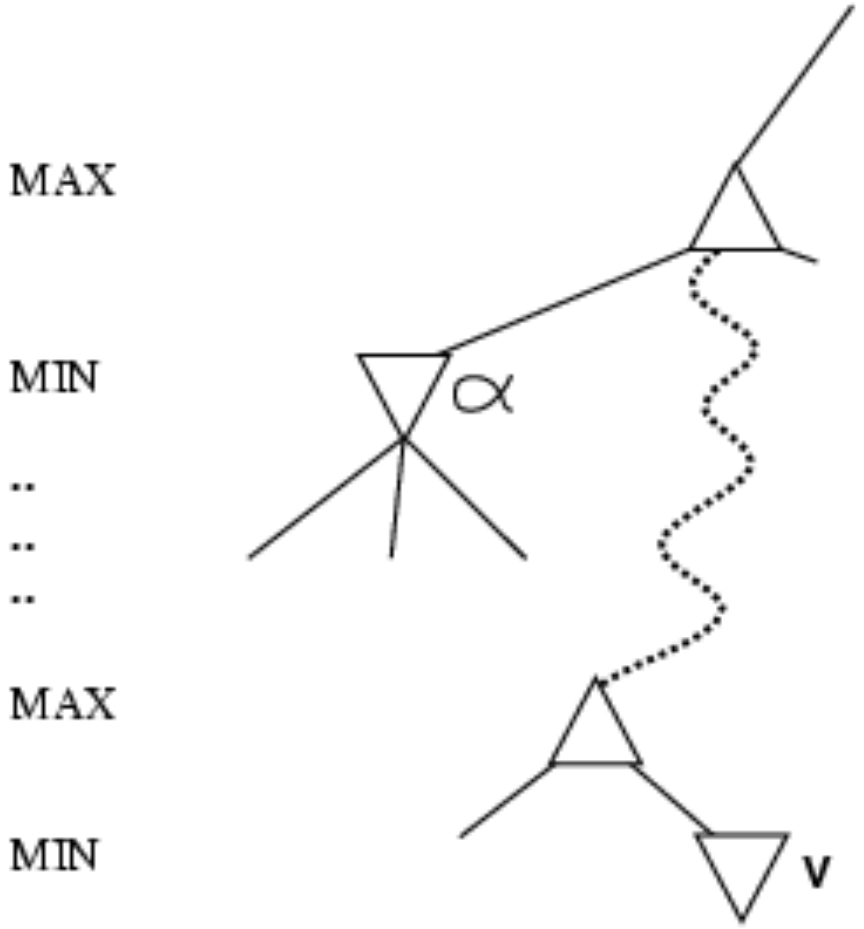
- Pruning does not affect final result
- Entire sub-trees can be pruned.
- Good move ordering improves effectiveness of pruning. With "perfect ordering"
  - time complexity = $O(b^{m/2})$

    → doubles depth of search
  - Branching factor of sqrt(b) !!
  - Alpha-beta pruning can look twice as far as minimax in the same amount of time
- Repeated states are again possible.
  - Store them in memory = transposition table
- A simple example of the value of reasoning about which computations are relevant (a form of metareasoning)

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
```

Imperfect, real-time decisions

- Minimax and alpha-beta pruning require too much leafnode evaluations.
- May be impractical within a reasonable amount of time.
- Suppose we have 100 secs, explore 10^4 nodes/sec
  → 10^6 nodes per move
- Standard approach (SHANNON, 1950):
  - Cut off search earlier (replace TERMINAL-TEST by CUTOFF-TEST)
  - Apply heuristic evaluation function EVAL (replacing utility function of alpha-beta)

Cut-off search

- Change:

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

  into:

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

- Introduces a fixed-depth limit *depth*
  - Is selected so that the amount of time will not exceed what the rules of the game allow.

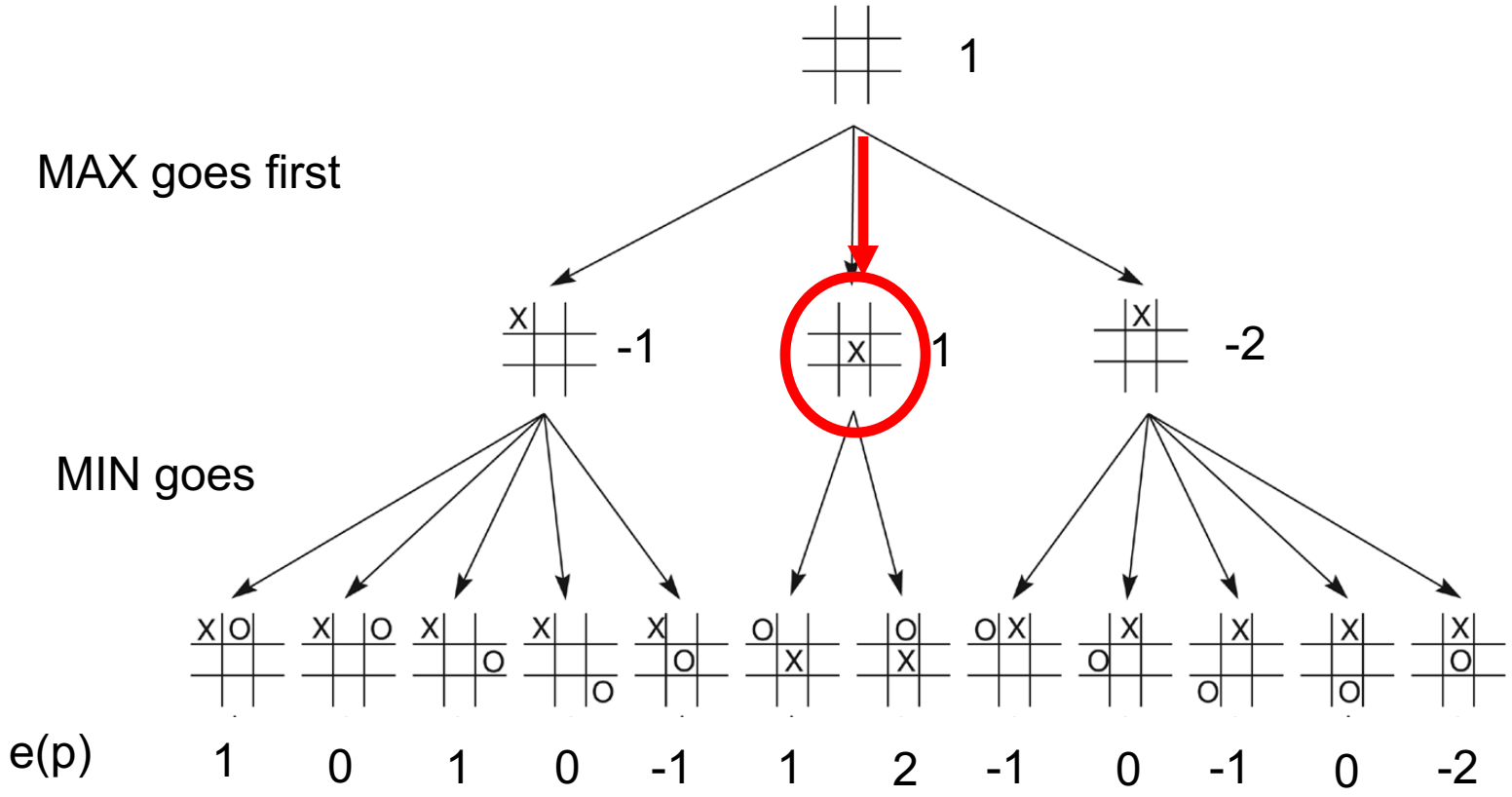
- When cut-off occurs, the evaluation is performed.

Heuristic evaluation (EVAL)

- Idea: produce an estimate of the expected utility of the game from a given position.
- 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.
- Example:
  Expected value e(p) for each state p:
  E(p) = (# open rows, columns, diagonals for MAX)
  - (# open rows, columns, diagonals for MIN)
- MAX moves all lines that don't have o; MIN moves all lines that don't have x

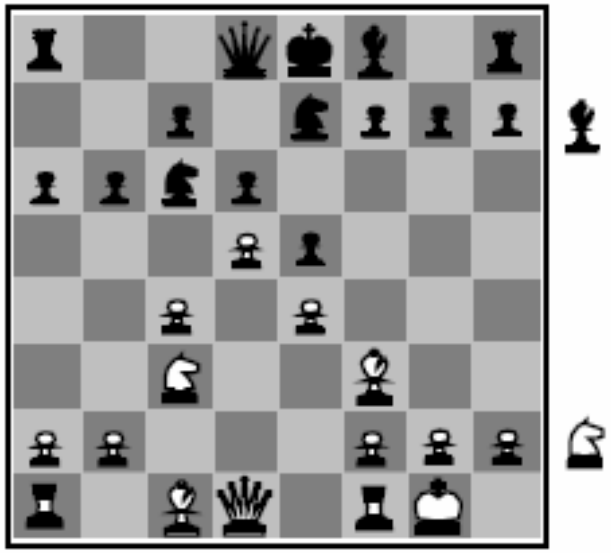
Expected value e(p) for each state p:
E(p) = (# open rows, columns, diagonals for MAX)
- (# open rows, columns, diagonals for MIN)
MAX moves all lines that don't have o; MIN moves all lines that don't have x

on the symmetry of the states



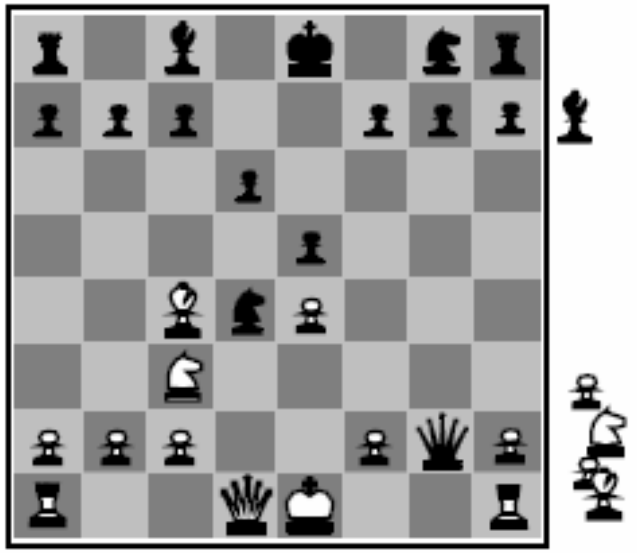
→ A kind of depth-first search

Evaluation function example



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.

Chess complexity

- PC can search 200 millions nodes/3min.
- Branching factor: ~35
  - 35^5 ~ 50 millions
  - if use minimax, could look ahead ***5 plies***, defeated by average player, planning 6-8 plies.
- Does it work in practice?
  - 4-ply ≈ human novice → hopeless chess player
  - 8-ply ≈ typical PC, human master
  - 12-ply ≈ Deep Blue, Kasparov
- To reach grandmaster level, needs a better *extensively tuned evaluation* and a *large database of optimal opening and ending* of the game

Deterministic games in practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a precomputed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.

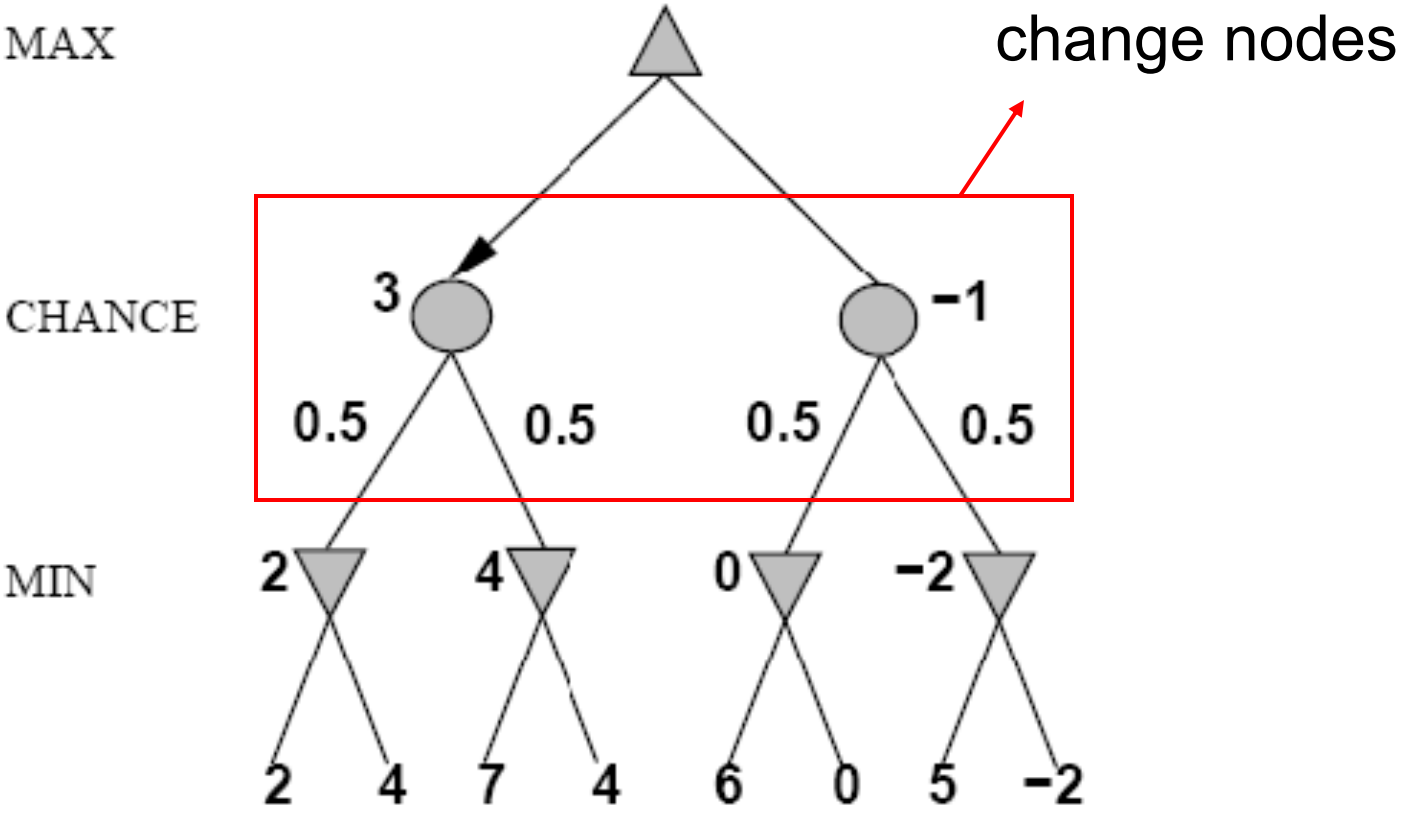
- Chess: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

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

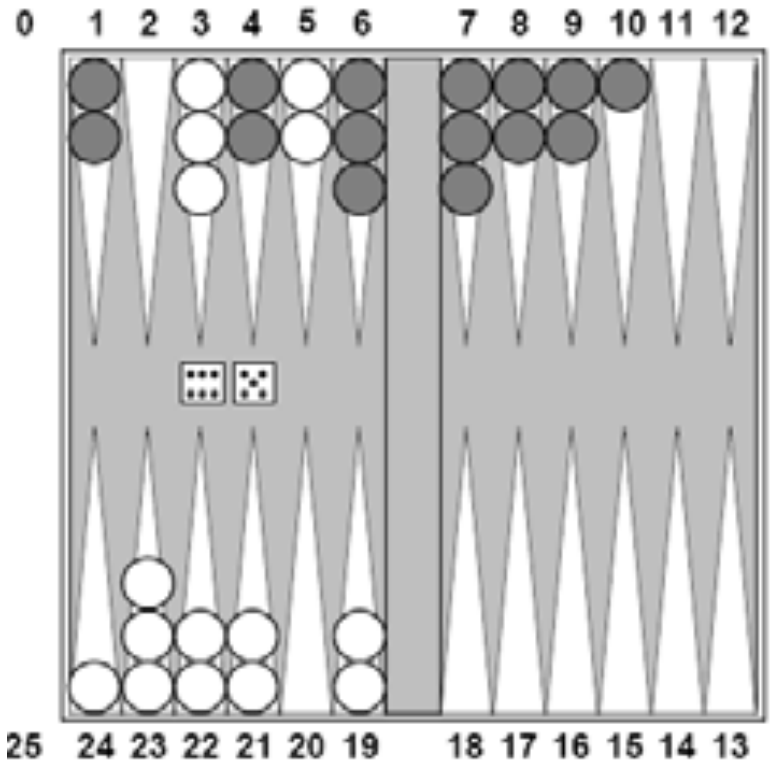
- Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.

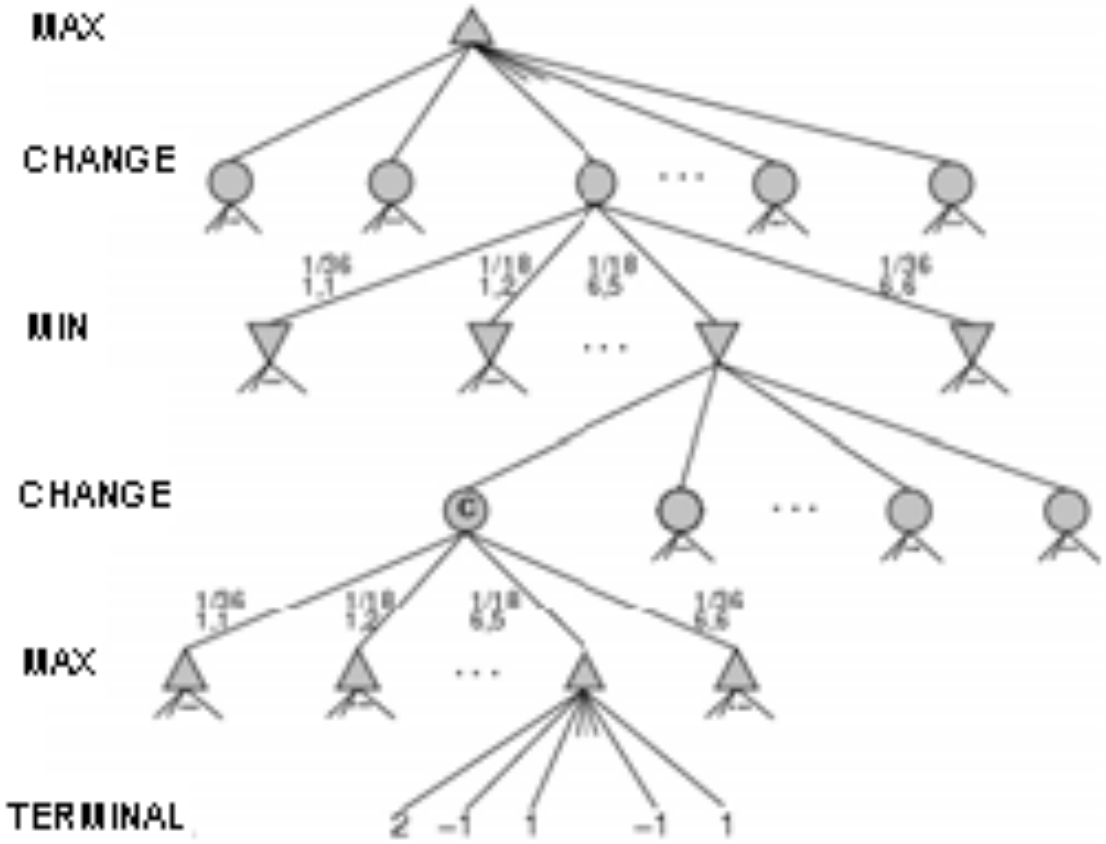
Nondeterministic games

- Chance introduces by dice, card-shuffling, coin-flipping...
- Example with coin-flipping:



Backgammon





Possible moves: (5-10,5-11), (5-11,19-24),(5-10,10-16) and (5-11,11-16)

Expected minimax value

```
...
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)
```

EXPECTED-MINIMAX-VALUE(*n*)=

UTILITY(*n*) If *n* is a terminal

$\max_{s \in successors(n)}$ EXPECTEDMINIMAX(*s*) If *n* is a max node

$\min_{s \in successors(n)}$ EXPECTEDMINIMAX(*s*) If *n* is a max node

$\Sigma_{s \in successors(n)}$ *P(s)* .EXPECTEDMINIMAX(*s*) If *n* is a chance node

P(s) is probability of s occurence

Games of imperfect information

- E.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
  - generating 100 deals consistent with bidding information
  - picking the action that wins most tricks on average

Reading and Suggested Exercises

- Chapter 4.1, 5.1, 5.2
- Suggested exercises: 5.8, 3.30, 4.3, 3.7, 4.11