Artificial Intelligence
A Modern Approach
Third Edition
Stuart Russell & Peter Norvig

Chapter 5
Adversarial Search

1. Games

- **Adversarial search** considers **multi-agent** and **competitive** environments
- **Game theory** considers both **competitive** and **cooperative** environments
- Most common games are **deterministic**, **turn-taking**, **two-player**, **zero-sum games** with **perfect information**
- Generalize to **stochastic games**, **multiple players**, **non zero-sum**, etc.

- Compared to, e.g., "**Prisoner's Dilemma**"
  - Deterministic, NON-turn-taking, NON-zero-sum game of Imperfect information

	B stays silent	B confesses
A stays silent	Each serves 1 yr	A: 5 yrs; B: free
A confesses	A: free; B: 5 yrs	Each serves 3 yrs

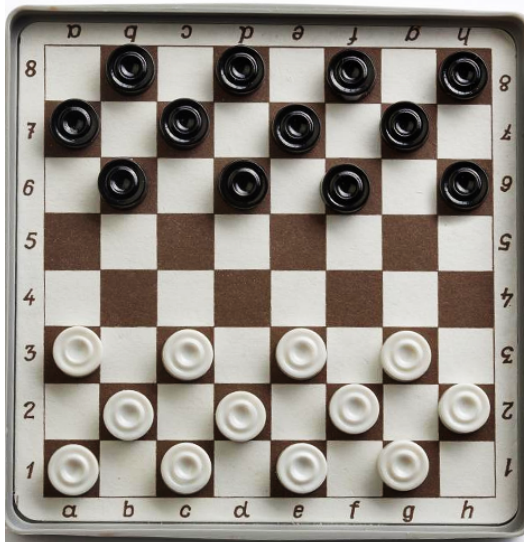
- Let's focus on the following types of game

	Deterministic	**Stochastic**
Perfect Information	Chess, Checkers, Go, Othello	Backgammon, Monopoly
Imperfect Information	Battleships, Bingo	Bridge, Poker

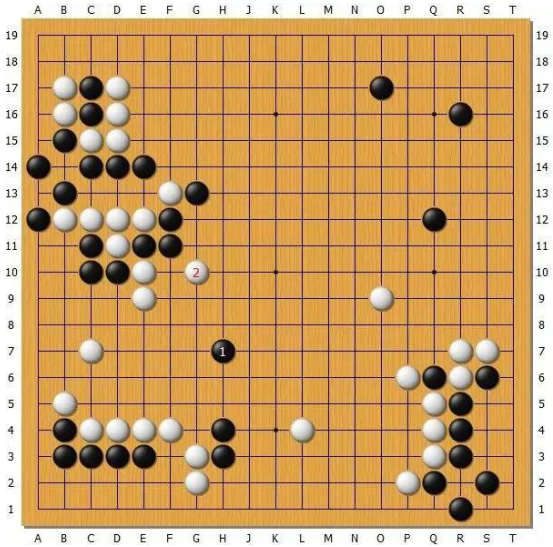
Perfect information, Deterministic

Chess 西洋棋

Checkers 西洋跳棋



GO 圍棋



Othello 黑白棋

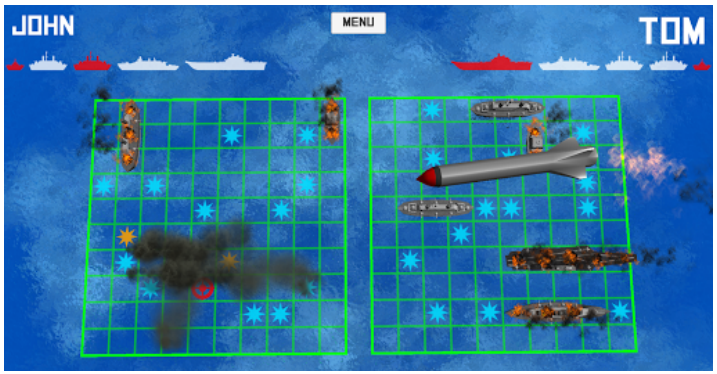
Perfect information, Stochastic

Backgammon 十五子棋，雙陸棋

Monopoly 地產大亨

Imperfect information, Deterministic

Battleship 海戰棋



Bingo (5 x 5)

Imperfect information, Stochastic

Bridge 橋牌

Poker 撲克

- **Types of games**
  - **Perfect-information games**
    - Minimax and Negamax search
    - α - β pruning
    - Pruning more
    - Imperfect decision [time's up decision]
    - Zobrist hashing
  - **Stochastic Games**
    - ExpectiMiniMax
    - Monte Carlo simulation
  - **Partially observable games**
    - Nash equilibrium

Search vs. Games

- **Search** – no adversary
  - **Solution** is a (heuristic) method for finding **goal**
  - **Heuristics** and **CSP** (Constraint Satisfaction Problem) 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

Game as Search

- Two players: MAX and MIN
- MAX moves first and they take turns until the game is over
  - Winner gets reward, loser gets penalty
  - "**Zero sum**" means the sum of the reward and the penalty is a **constant**
- MAX uses **search tree** to determine next move

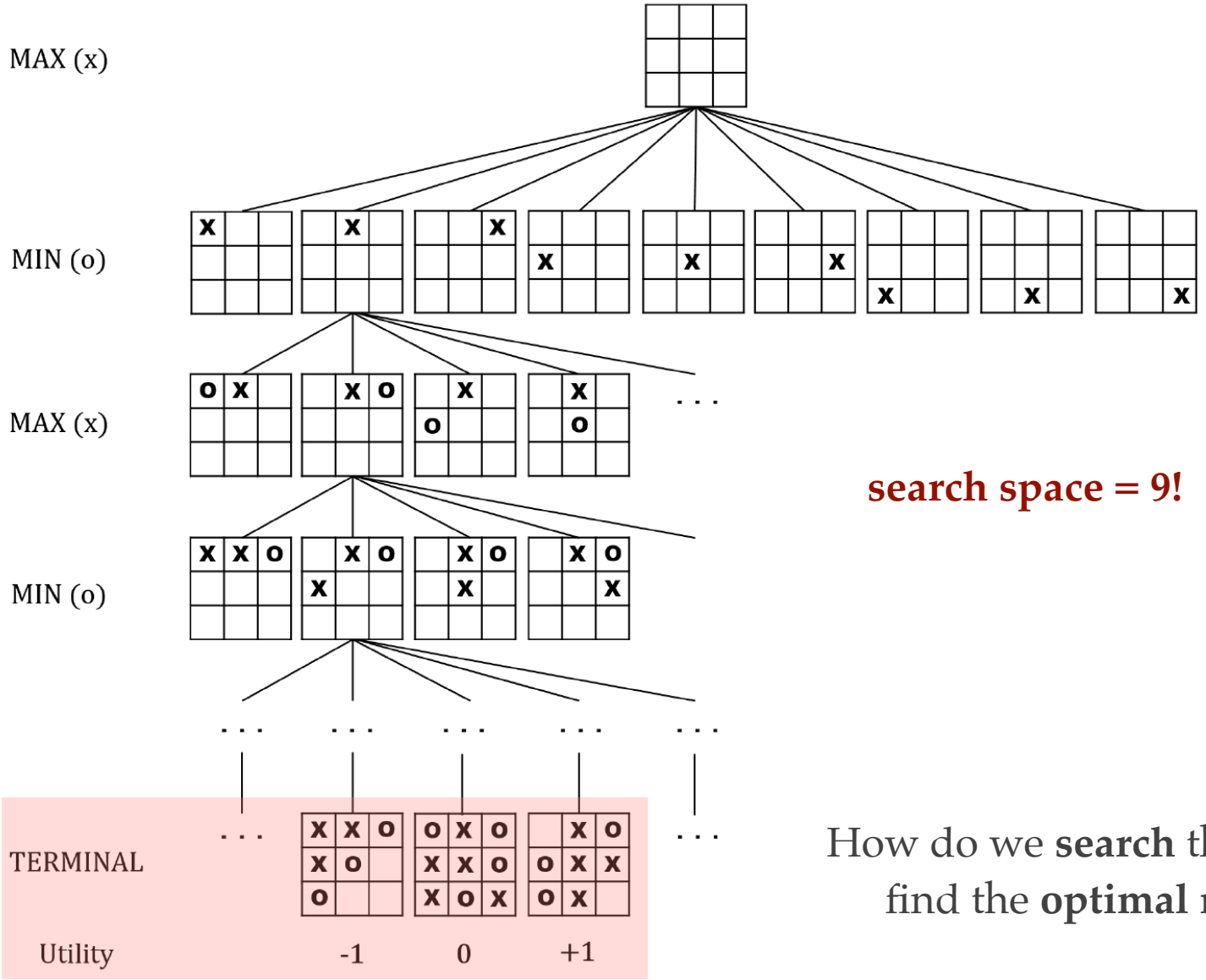
Formal Definition as a Search Problem

- s_0: Initial state
- **Player(s)**: The player in state *s*
- **Action(s)**: Returns the set of **legal moves** in state *s*
- **Result(s, a)**: The **transition model**, which returns the resulting state of a move *a* in state *s*
- **Terminal-Test(s)**:
  - Is the game finished? **True** if finished, **false** otherwise
  - States where the game has terminated are called **terminal states**

- **Utility(s, p)**: A **utility function,** also called **objective function** (in optimization area) or **payoff function** (in Economics or game theory)
  - Applicable only if the **terminal-test** is **TRUE** (finished)
  - Defines the final **numeric value** for a game that ends in terminal state ***s*** for a player ***p***
    - E.g., win (+1), lose (-1), and draw (0) in tic-tac-toe, chess

- **Zero-sum game**: the total payoff to all players is the same for every instance of the game.
  - Chess is zero-sum because every game has payoff of either (-1) + 1, 1 + (-1) or 0 + 0
- Utility for 2-player, **zero-sum** games is **Utility(s)** [instead of Utility(s, p)]
  - Reason: **Utility(s, p_1) = – Utility(s, p_2)**, e.g. $p_1 = 3$, $p_2 = -3$
  - Utility values for each agent are the **opposite** of the other

Game Tree (of Tic-Tac-Toe)



search space = 9!

How do we **search** this tree to find the **optimal move**?

2. Optimal Decisions in Games

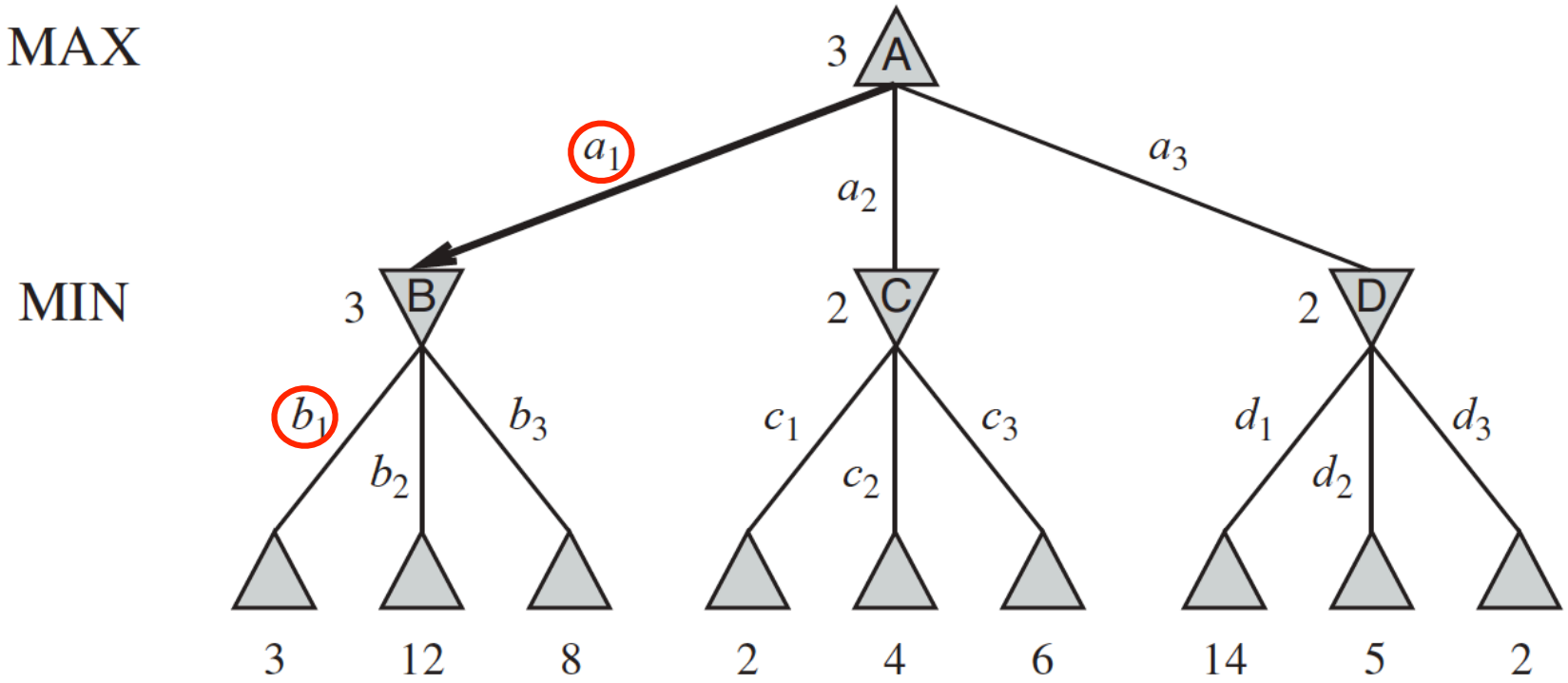
- Min-Max method is designed to find the **optimal strategy** for **Max** and find **best move**

1. Generate the **whole game tree**, down to the leaves
2. Apply **utility** (**payoff**) function to each **leaf**
3. **Back-up values** from leaves through branch nodes
   - A **Max node** computes the Max of its child values
   - A **Min node** computes the Min of its child values
4. At **root**: choose the move leading to the child of **highest value**

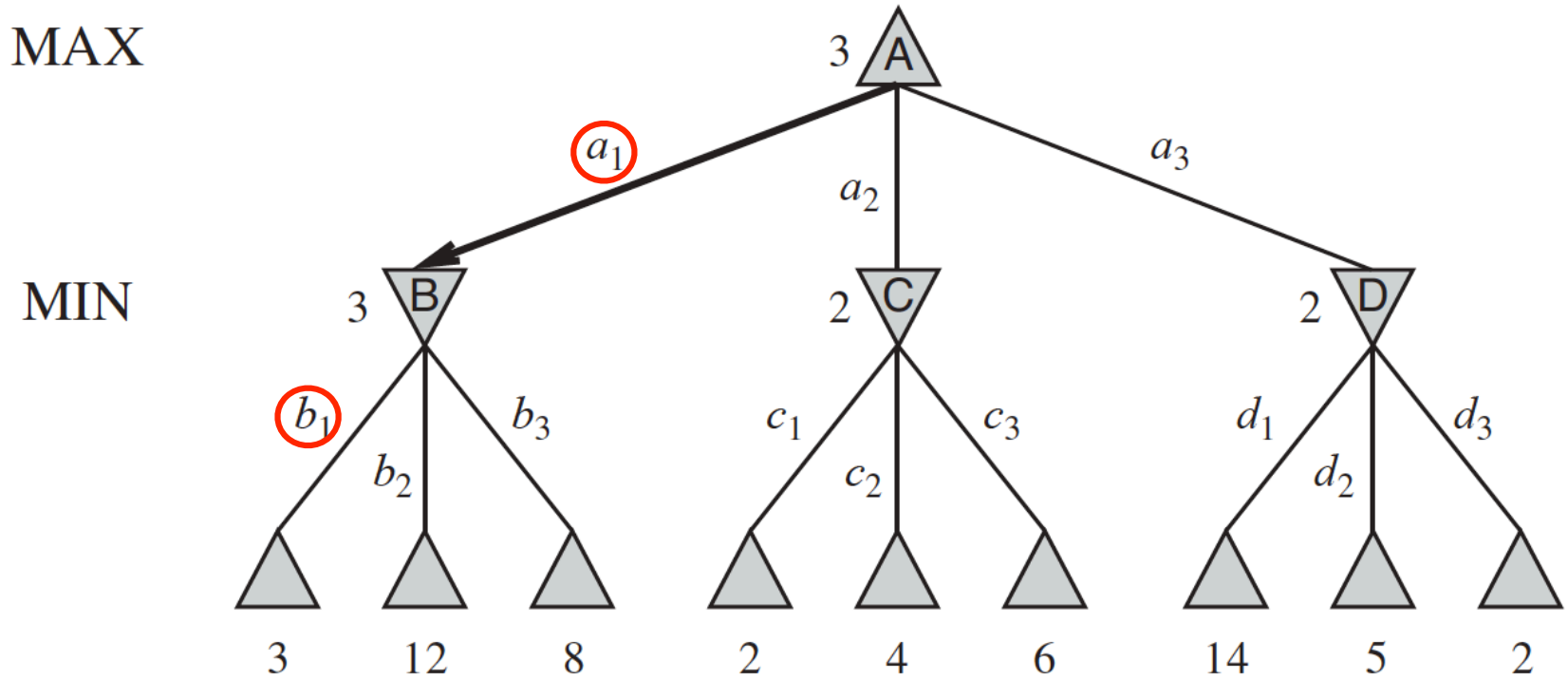
Optimal decision

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

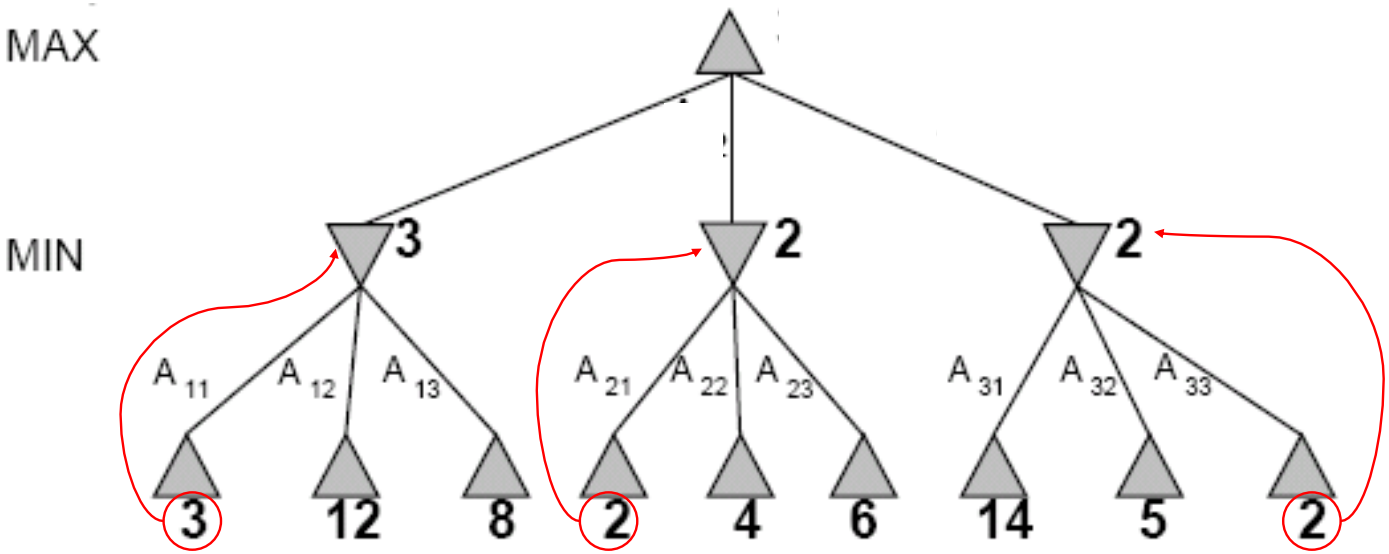
(If TERMINAL then return UTILITY)

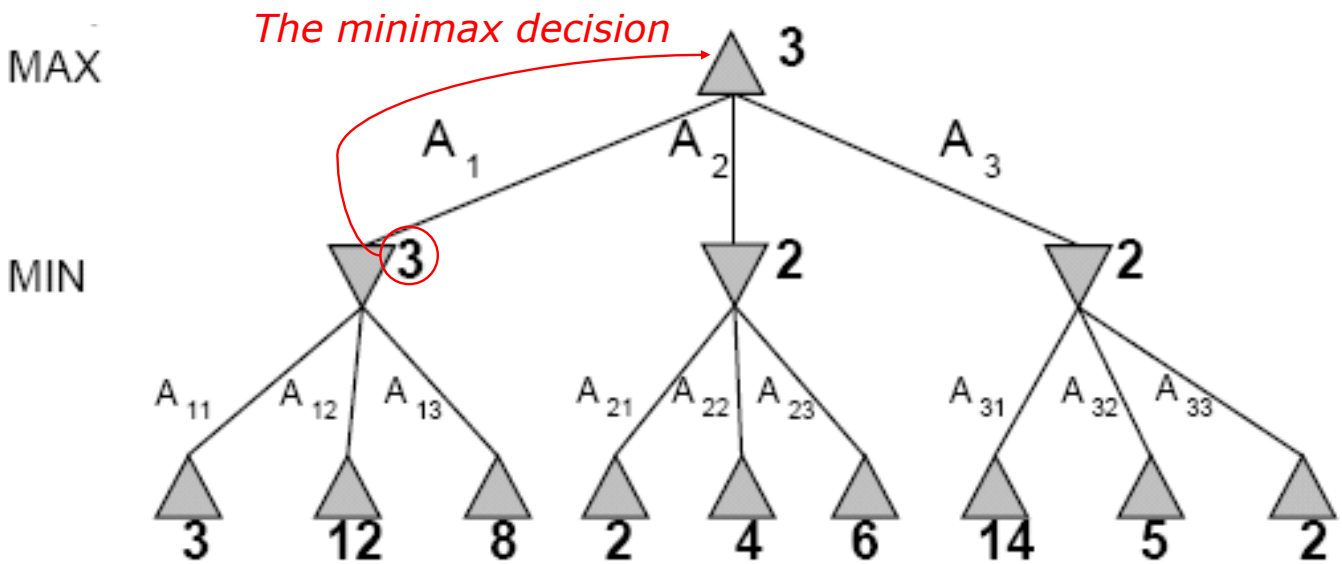


Two players (zero-sum): MAX, MIN



A two-ply game tree. The △ **nodes** are "MAX nodes," in which it is MAX's turn to move, and the ▽ **nodes** are "MIN nodes." **The terminal nodes** show the **utility values** for **MAX**; the **other nodes** are labeled with their **minimax values**. MAX's **best move** at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.





Minimax **maximizes** the utility for the **worst-case outcome** for max

2.1 The Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

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

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

An algorithm for calculating **minimax decisions**. It returns the action corresponding to the **best possible move**, that is, the move that leads to the outcome with the **best utility**, under the **assumption** that the **opponent plays to minimize utility**. The functions **MAX-VALUE** and **MIN-VALUE** go through the whole game tree, all the way to the leaves, to determine the **backed-up value** of a state. The notation $argmax_{a \in S}\, f(a)$ computes the element a of set S that has the **maximum value** of $f(a)$.

Properties of Minimax Search

- **Completeness**: Yes, if tree is **finite**
- **Optimality**
  - **Yes**, against an **optimal opponent**
  - Can it be beaten by an **opponent** playing **sub-optimally**?
    - **No, MAX will do even better!**
- **Time complexity**: **$O(b^m)$**
- **Space complexity**
  - **O(bm)** : **depth-first search**, generate all actions at once
  - **O(m)**: backtracking search, generate actions **one** at a time

Game Tree Size

- **Chess**
  - $b \approx 35$ (approximate average branching factor)
  - $m \approx 100$ (depth of game tree for "typical" game)
  - $b^m \approx 35^{100} \approx 10^{154}$ nodes!!

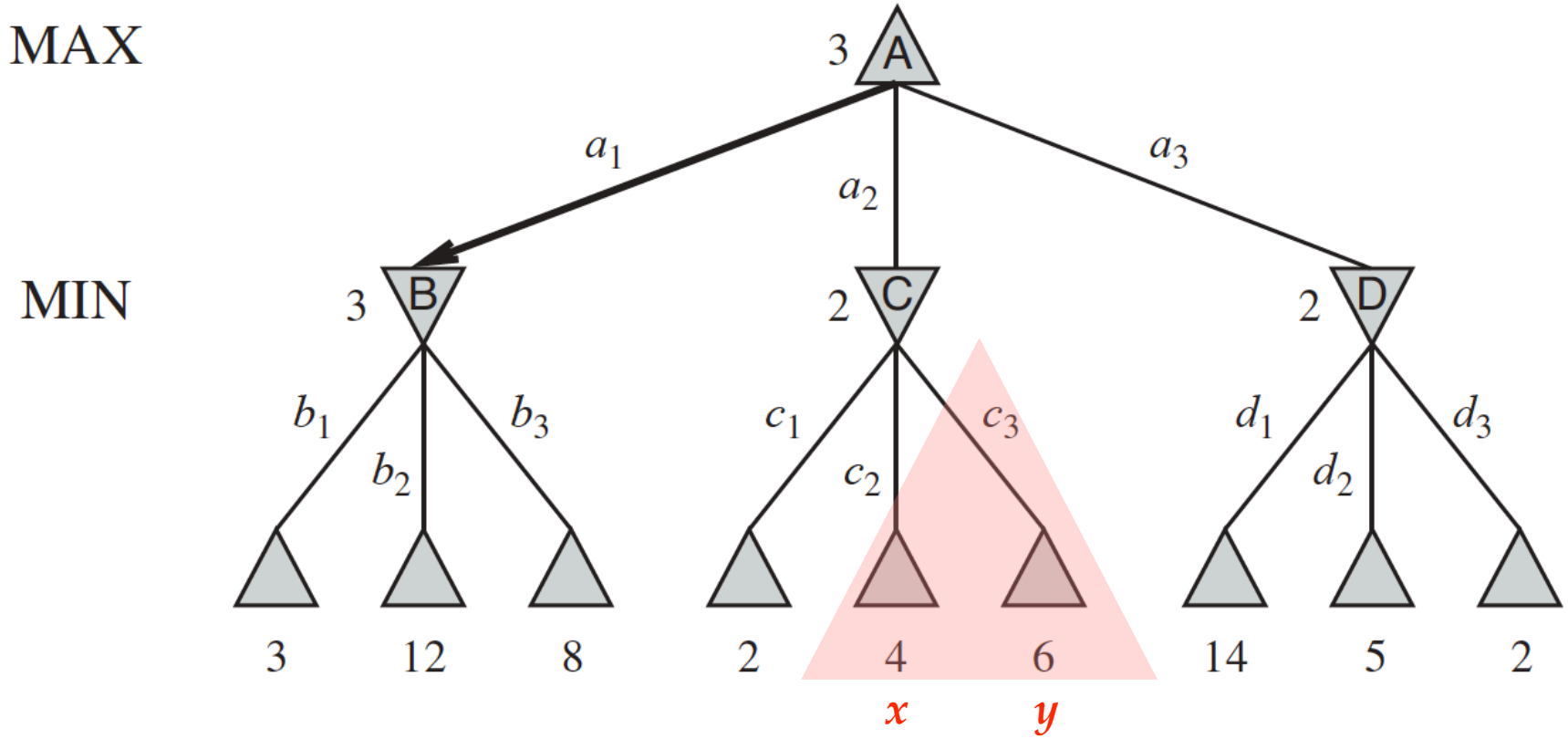
    ➔ optimal decision is practically intractable

- **It is usually impossible to develop the whole search tree**

3. Alpha–Beta Pruning

- If a position is provably **bad**
  - It is **NO USE** expending search time to find out exactly how bad
  - **Bad** means not better than we already know we can achieve elsewhere
- Contrast **normal search**
  - ANY node might be a winner
  - ALL nodes must be considered

- Not every node needs to be evaluated

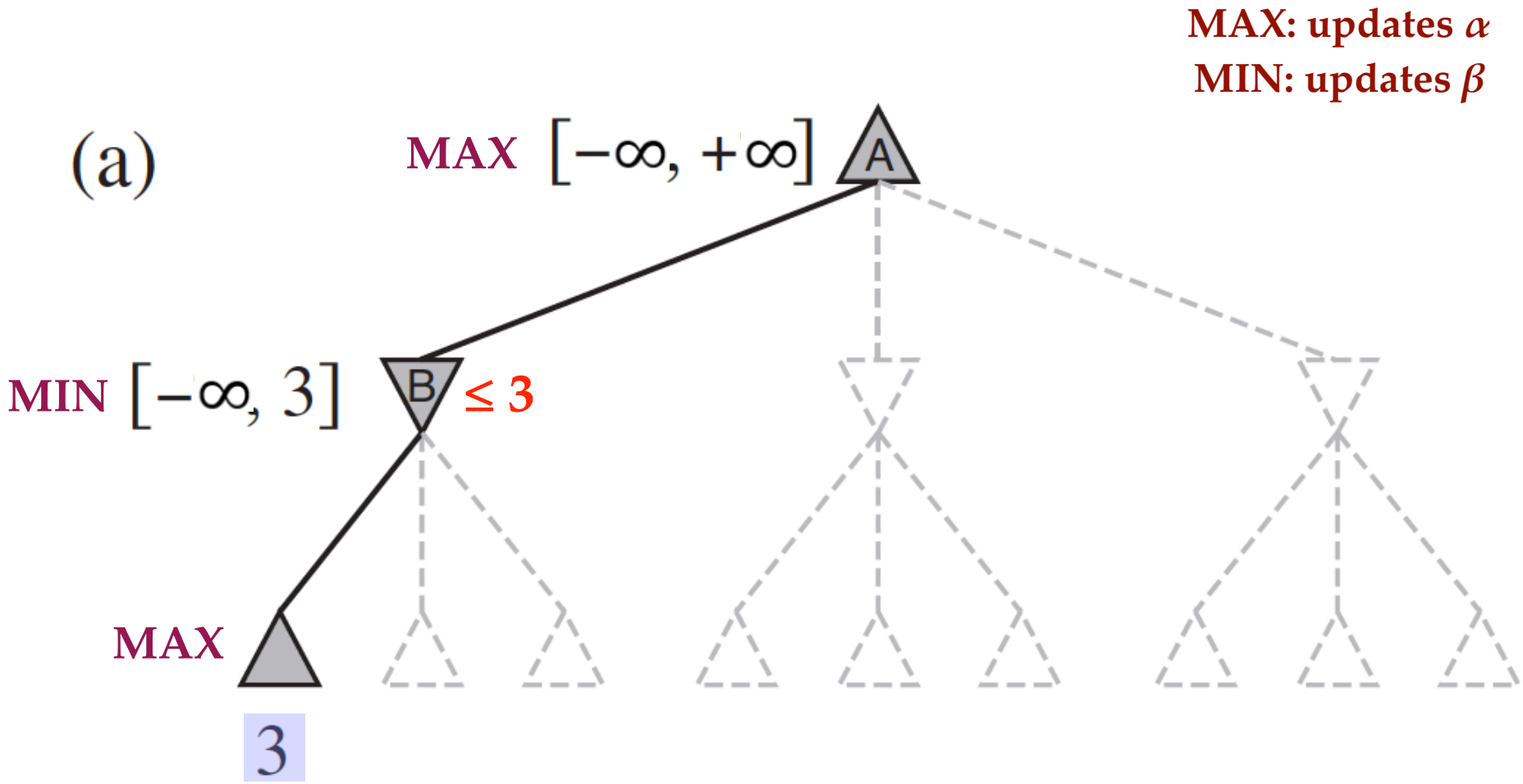


$$
\begin{aligned}
\textsc{Minimax}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3, z, 2) \qquad \text{where } z = \min(2,x,y) \le 2 \\
&= 3.
\end{aligned}
$$

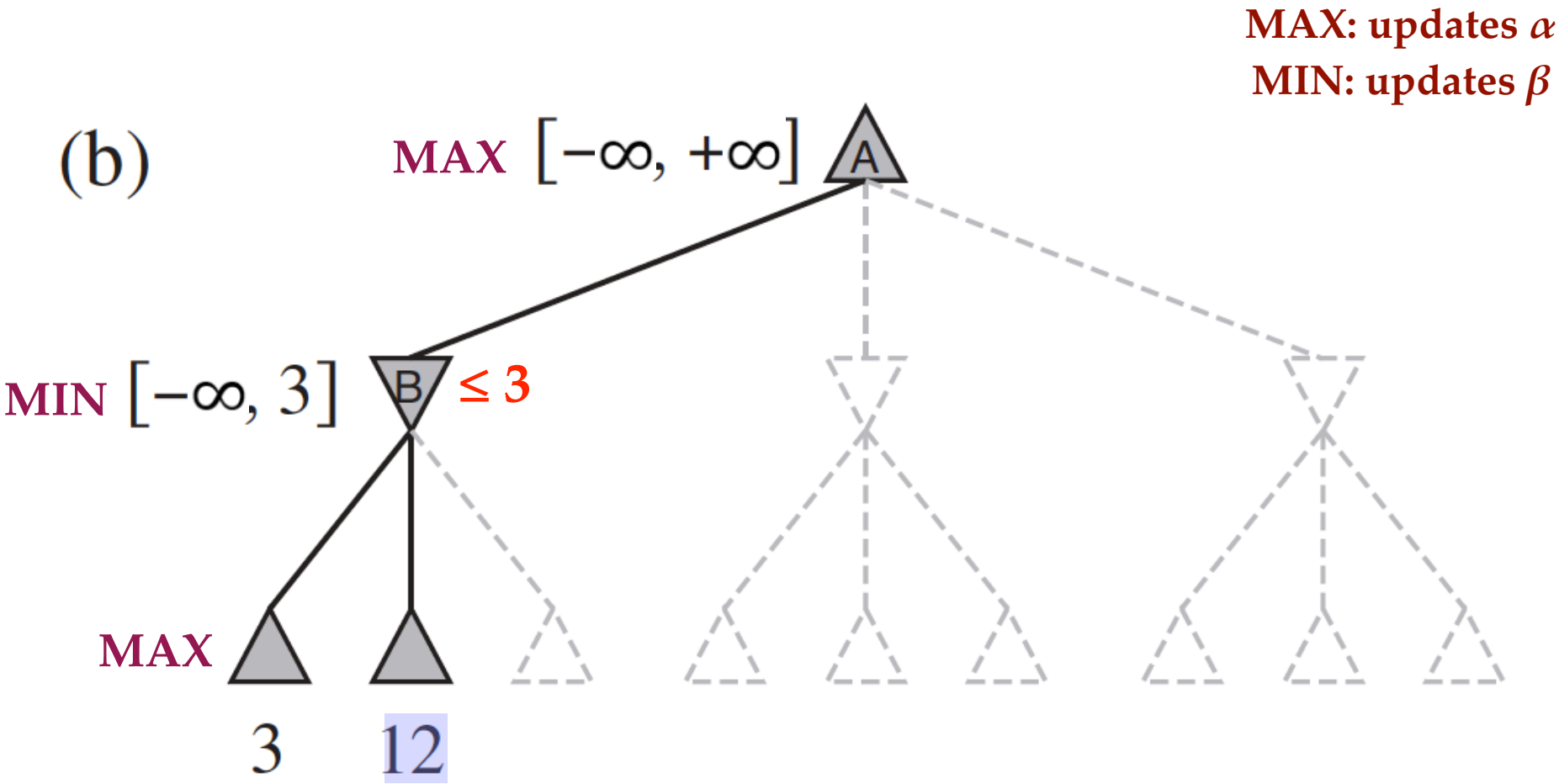
α - β Pruning Algorithm

- **Depth first search**
  - Only considers nodes along a **single path** from root at any time
- α and β
  - Keeping α (**maximum lower bound**) for the max utility for player MAX, **initialized to -∞**
  - Keeping β (**minimum upper bound**) for the min utility for player MIN, **initialized to** ∞

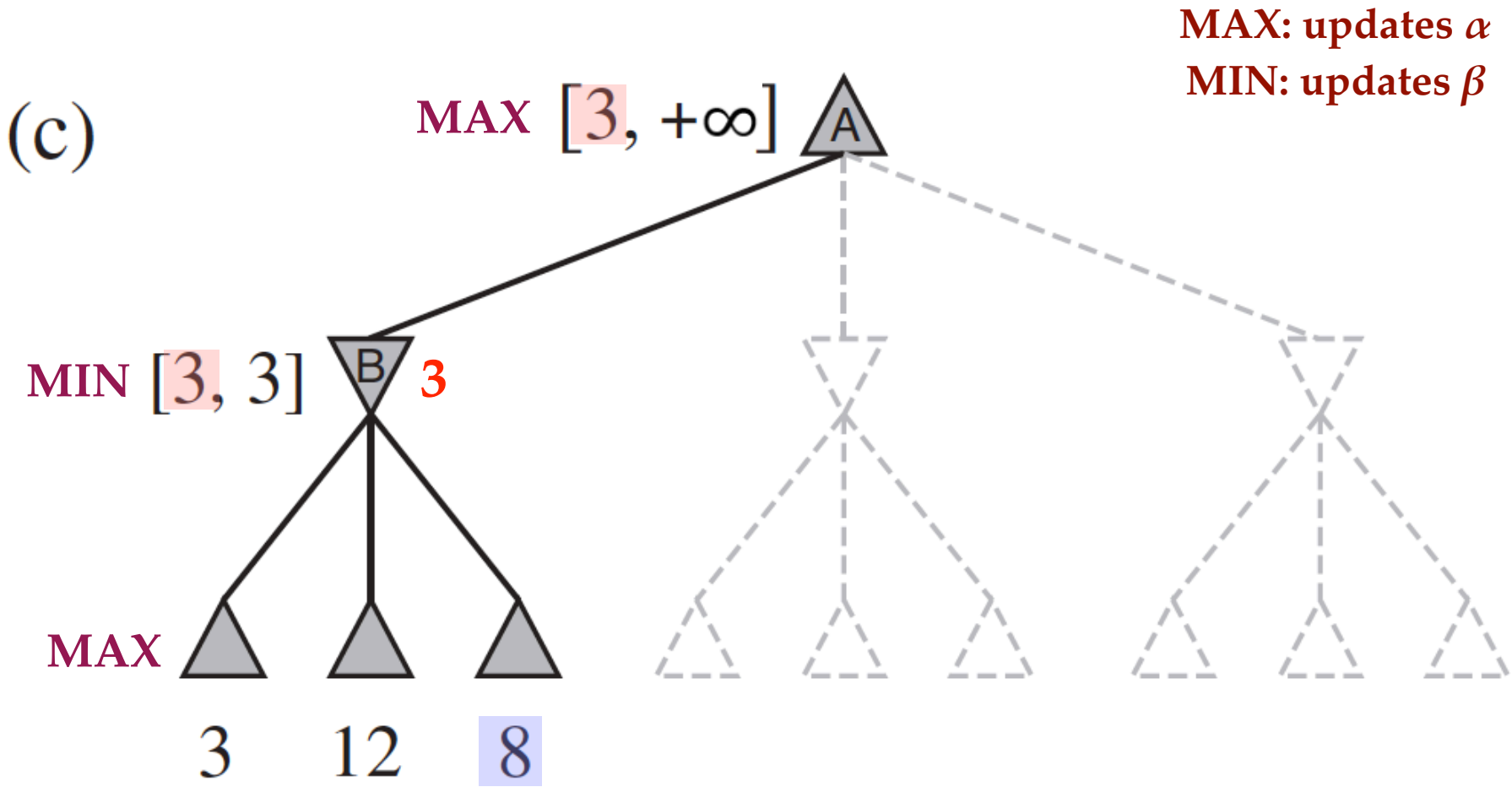
- **Update** values of α and β during search
  - MAX updates α at MAX nodes
  - MIN updates β at MIN nodes
- Only the moves within the $[\alpha, \beta]$ window are expanded; otherwise its branches are pruned, i.e. **prune** remaining branches at a node when $\alpha \geq \beta$
- Pass current values of α and β down to child nodes during search



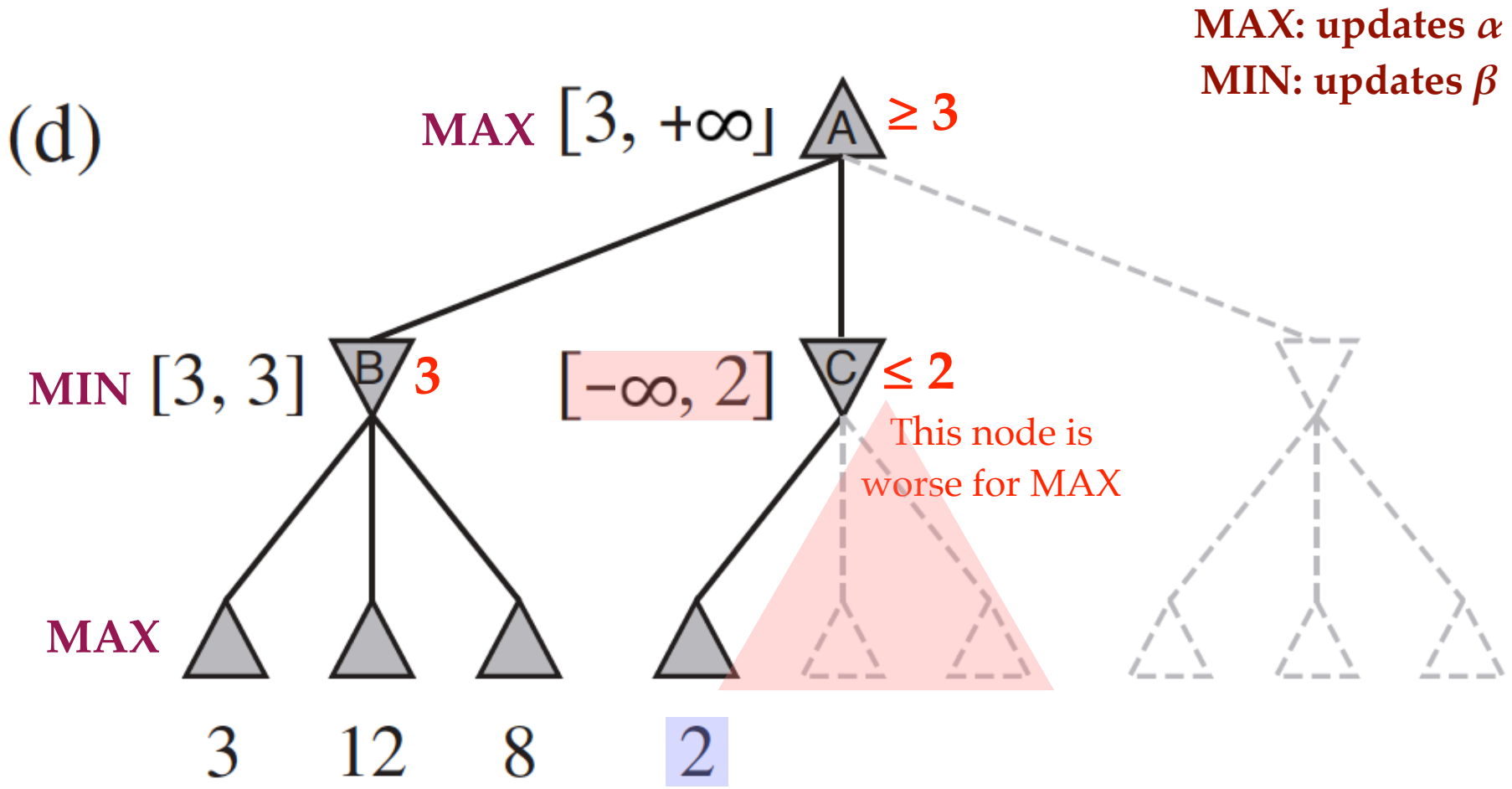
Stages in the calculation of the **optimal decision** for the game tree. At each point, we show the **range** of possible values for each node. **(a)** The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of at most 3.



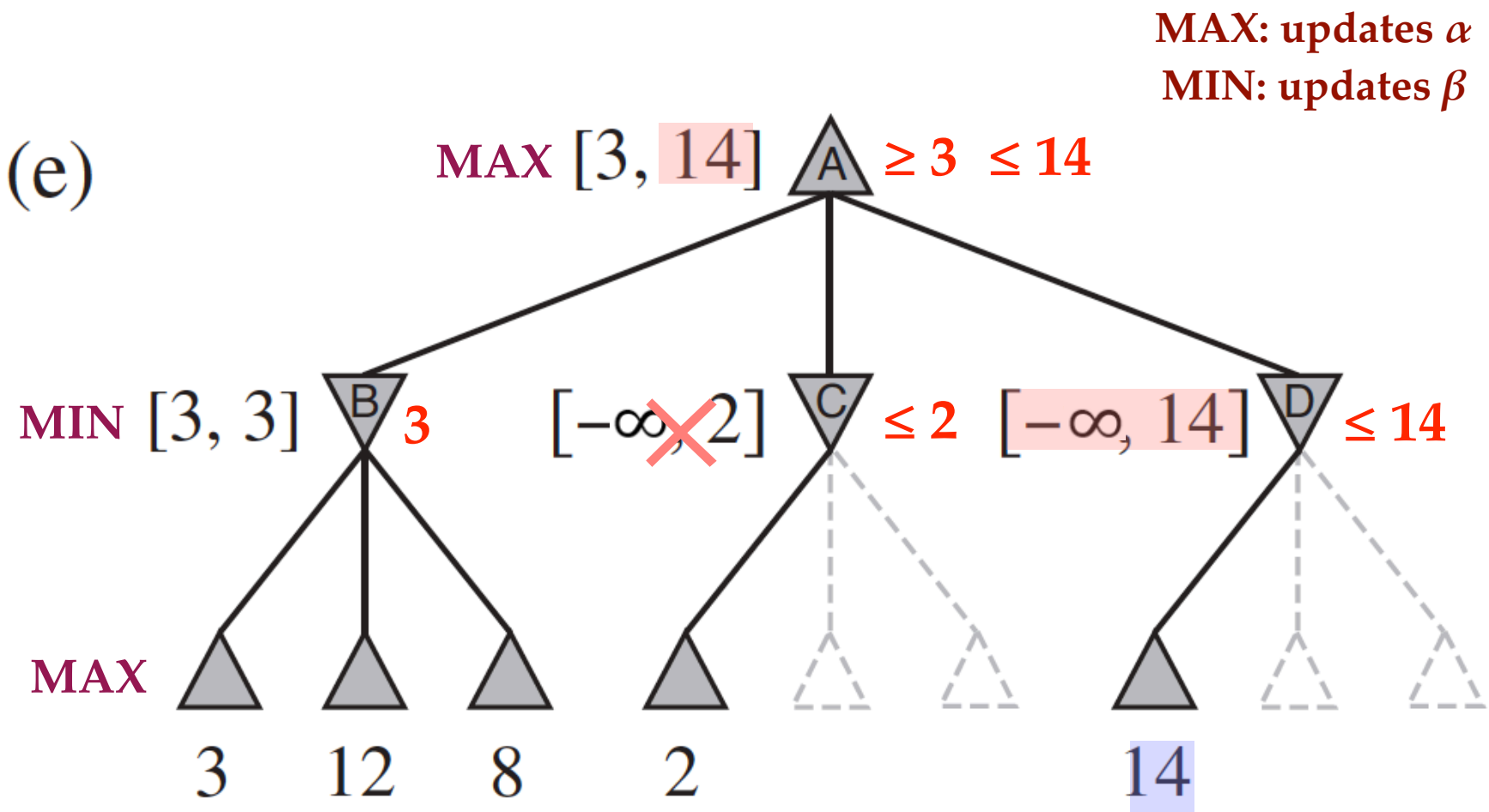
(b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3.



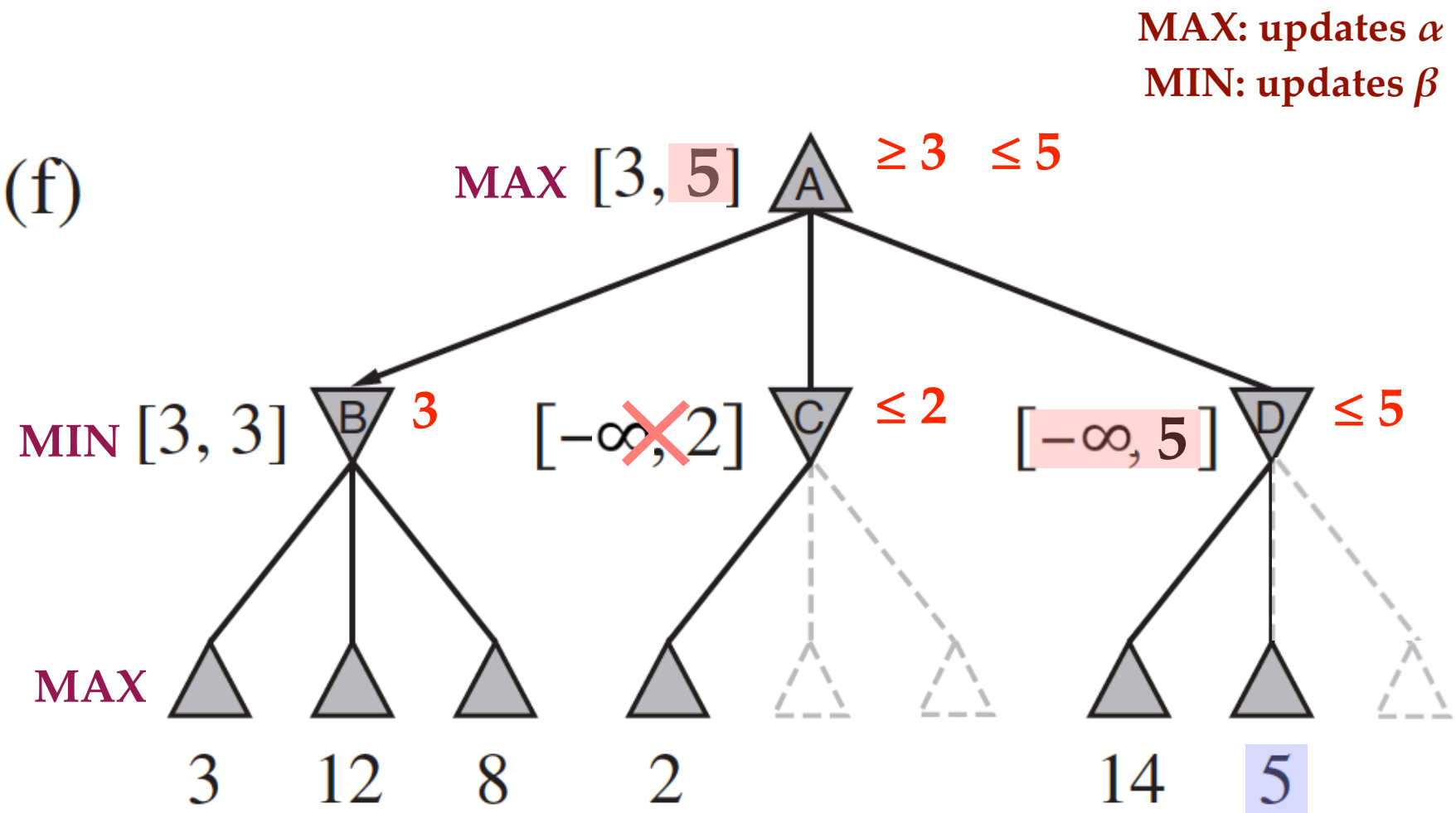
(c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the **root** is at least 3, because MAX has a choice worth 3 at the root.



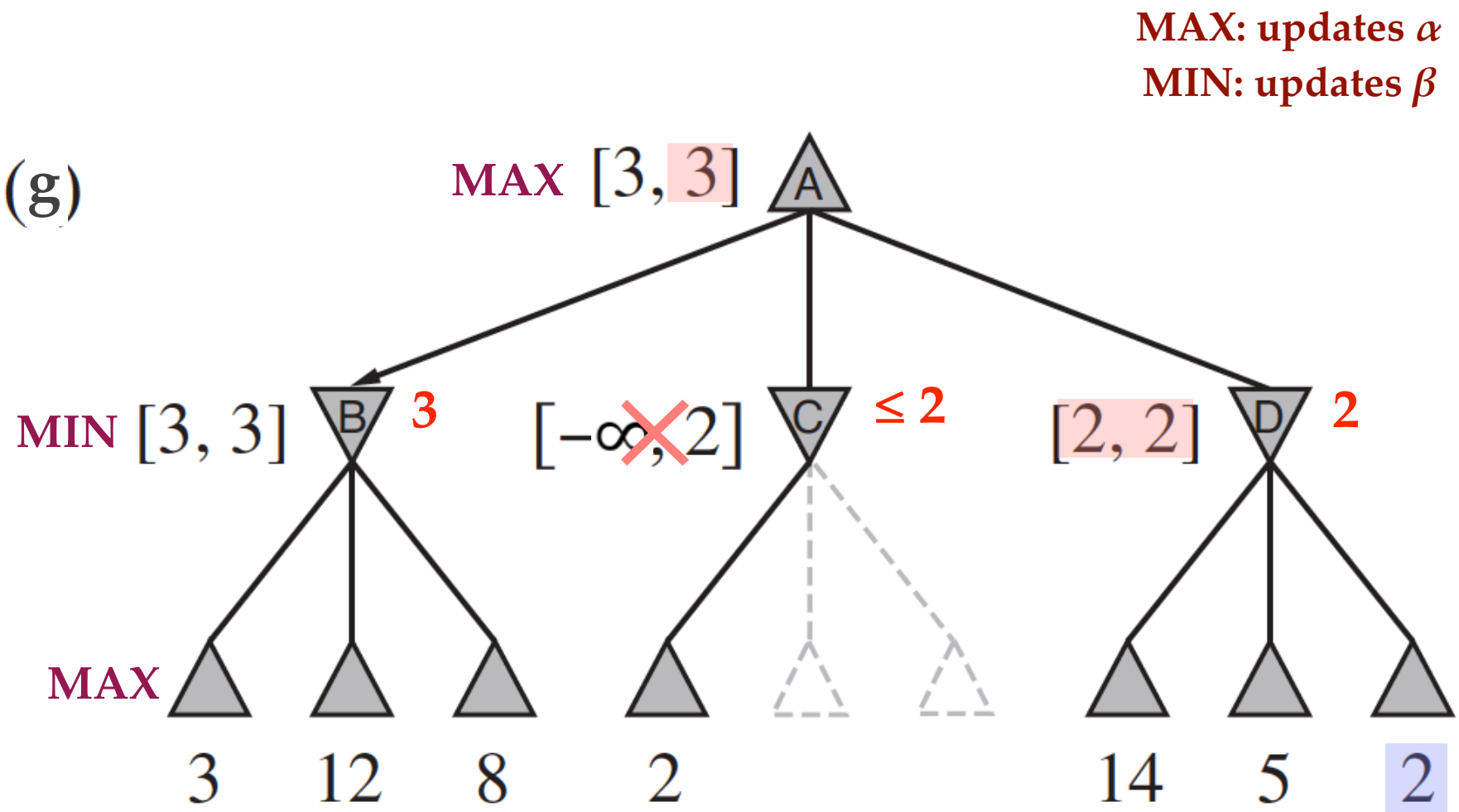
(d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of **alpha–beta pruning**.



(e) The first leaf below D has the value 14, so D is worth at most 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14.



(f) The second successor of D is worth 5, so again we need to keep exploring.



(g) The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

```
function ALPHA-BETA-SEARCH(state) returns an action
   v ← MAX-VALUE(state, −∞, +∞)
   return the action in ACTIONS(state) with value v
```

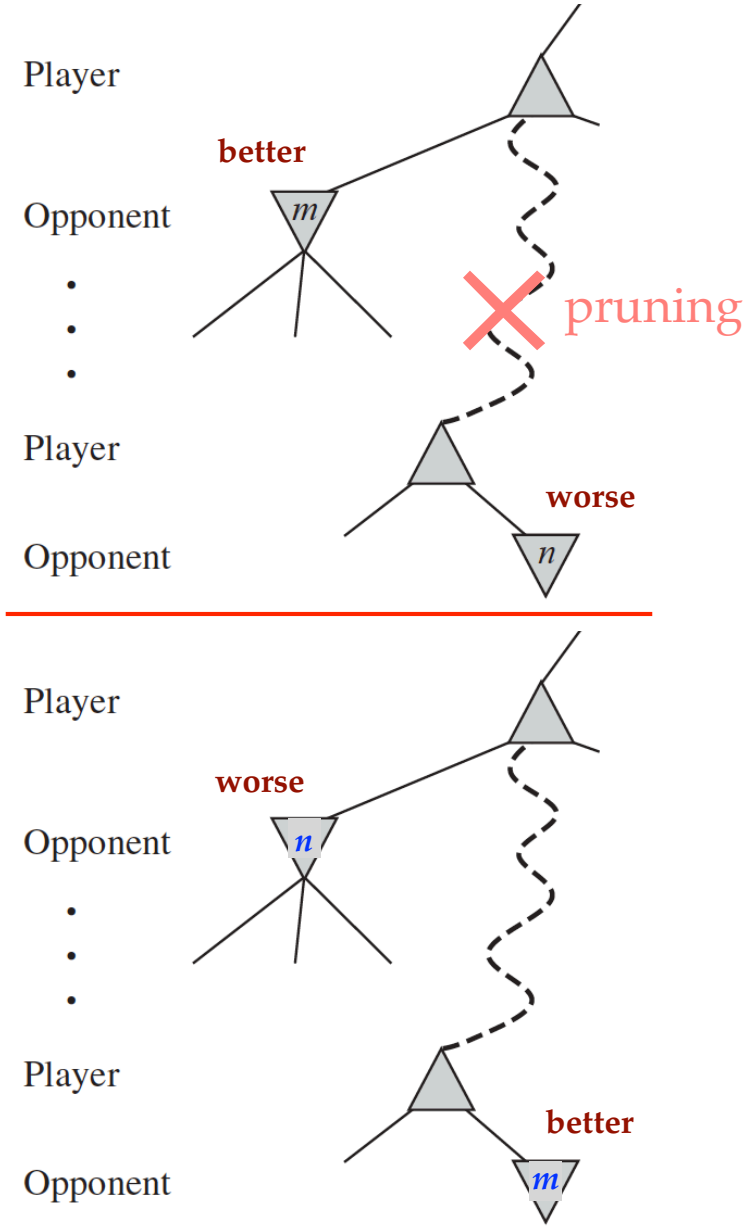
```
function MAX-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state) do
      v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))   [update α]
      if v ≥ β then return v   [cutoff]
      α ← MAX(α, v)   [update α]
   return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for each a in ACTIONS(state) do
      v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))   [update β]
      if v ≤ α then return v   [cutoff]
      β ← MIN(β, v)   [update β]
   return v
```

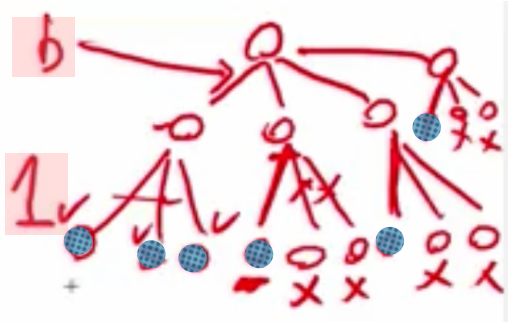
The **alpha–beta search algorithm**. Notice that these routines are the same as the **MINIMAX functions**, except for the two lines in each of MIN-VALUE and MAX-VALUE that **maintain α and β** (and the bookkeeping to pass these parameters along).

Efficiency of α - β Pruning

- The general case for alpha–beta pruning
  - If ***m*** is better than ***n*** for Player, we will never get to ***n*** in play
- Once we have found enough about ***n*** to reach this conclusion, we can prune it
- Pruning efficiency is highly depends on the **order of moves**, e.g. visit ***m*** (better) first or ***n*** (worse) first?



- **Worst case**: no pruning ➔ $O(b^m)$ [same as Minimax]
  - Branches are ordered so that **no pruning** takes place
  - In this case alpha-beta gives no improvement over exhaustive search
- **Best case**: Always check the **best move** first [max cutoff]
  - Each player's best move is the **left-most child** (i.e., evaluated first)
  - Still need to **check every move for the first player** (b)
  - Only need to check **one move for the second player** (1)
  - $O(b \cdot 1 \cdot b \cdot 1 \cdots) = O(b^{m/2})$



- In practice, performance is closer to **best** rather than worst-case
  - E.g., **sort moves** by the remembered move values found last time
  - E.g., run **Iterative Deepening** search, **sort** by value last iteration

- **Average case**: $O(b^{3m/4})$ [random order, complicated analysis]
- Very simple ordering **usually achieves** best case $\mathbf{O(b^{m/2})}$ by adopting **iterative deepening**
  - Reduce the **effective branch factor** to $b^{1/2}$ and we effectively go from b to $b^{1/2}$ $[O(b^{m/2}) = O((b^{1/2})^m)]$
  - Make the search **twice** as deep as before $[m \rightarrow m/2]$
    - e.g., In chess go from $b \sim 35$ to $b \sim 6$ and this permits **much deeper search** in the same amount of time

- Final comments about Alpha-Beta pruning
  - The pruning does **NOT** compromise solution quality (pruning does not affect final results)
  - Entire subtrees can be pruned
  - **Good move ordering** improves **effectiveness** of pruning

4. Imperfect Real-Time Decisions

- Minimax algorithm generates the **entire game search space**
- **Alpha-beta algorithm**
  - Allows us to prune large parts of it
  - Still has to search all the way to **terminal states** for at least a portion of the search space
    - This **depth is usually not practical**, because moves must be made in a **reasonable amount of time** - typically a few minutes at most

- Claude Shannon's paper *Programming a Computer for Playing Chess* (1950)
  - Programs should **cut off** the search **earlier** and apply a **heuristic evaluation function** to states in the search, **effectively turning nonterminal nodes into terminal leaves**
  - Alter minimax or alpha-beta in two ways
    - Replace the **utility function** by a **heuristic evaluation function, EVAL**, which estimates the position's utility
    - Replace the **terminal test** by a **cutoff test** that decides **when to apply EVAL**

UTILITY can return value only at terminal state
EVAL can return value at non-terminal state (real-time)

- **Heuristic minimax** for state ***s*** and maximum depth ***d***

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN}. \end{cases}$$

- **Imperfect real-time decisions**
  - Use **cutoff-test** instead of **terminal-test**
    - Can be as simple as **depth limit**
    - Can adopt **quiescence search** to conquer the **horizon effect**
    - Yet another good reason to adopt **iterative deepening**
    - Return the **current best move** when time's up

- Use **EVAL** instead of **UTILITY**

  [after feature selection]

  - Usually maps **state** s into **feature space** f_i
  - Typically use **linear combination of features** (features: categories or equivalence classes of states)

$$\boxed{\textsc{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)}$$

[good features with appropriate weights]

    - i.e. compute separate numerical contributions from each **feature** and then combine them to find the total value
    - **Strong assumption**: the contribution of each feature is **independent** of the values of the other features
  - Need to fine tune **weights** for strong play

4.1 Evaluation Functions

- **Evaluation function**
  - Estimates **how good** the current board configuration is for a player
  - Typically,

Evaluation function = EVAL(player) - EVAL(opponent)

- Examples
  - **Othello**: **no.** of white pieces - **no.** of black pieces
  - **Chess**: **value** of all white pieces - **value** of all black pieces
- Typical values from **-infinity (loss)** to **+infinity (win)** or **[-1, +1]**
- **"Zero-sum game"** if the board evaluation is **X** for a player, it's **-X** for the opponent

- **Cut off search** at a **certain depth** and compute the value of an **evaluation function** for a state instead of its minimax value
  - The evaluation function may be thought of as
    - The **probability of winning** from a given state, or
    - The **expected value** of that state

- A common evaluation function is a **weighted sum of features**

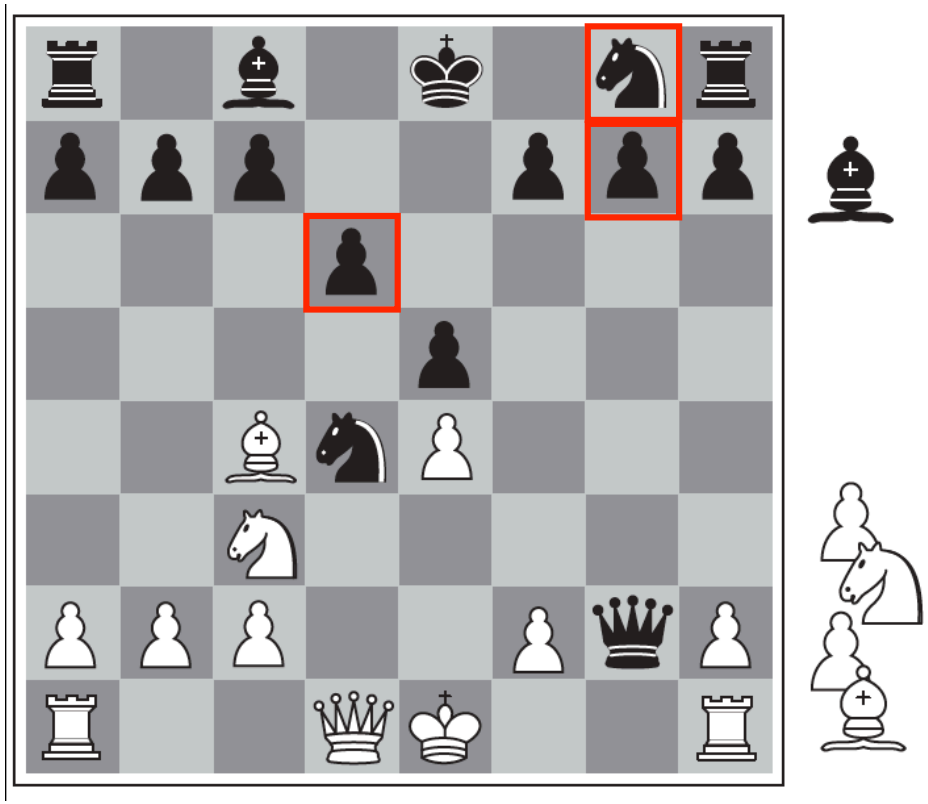
$$\boxed{\textsc{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)}$$

[good features with appropriate weights]

  - **Strong assumption**: the contribution of each feature is **independent** of the values of the other features

- For chess,
  - w_k may be the **material value** of a piece
    - e.g. *pawn* = 1, *knight* = 3, *bishop* = 3, *rook* = 5, *queen* = 9
  - $f_k(s)$ may be the **advantage** in terms of that piece
    - e.g. the numbers of each kind of piece on the board

- Example: A secure advantage equivalent to
  - **A pawn** gives a **substantial likelihood of winning**
  - **Three pawns** should give **almost certain victory**



西洋棋棋子		
國王	King	
后	Queen	
城堡（城堡）	Rook	
主教（主教）	Bishop	
騎士（騎士）	Knight	
兵	Pawn	

Black has an advantage of **a knight** and **two pawns**

- **Evaluation functions** may be learned from **game databases** or by having the program **play many games** against itself
  - Example
    - Suppose our experience suggests that
      - 72% of the states encountered in the **two-*pawns* vs. one-*pawn*** category lead to a win (utility = +1);
      - 20% to a loss (utility = 0)
      - 8% to a draw (utility = 1/2)
    - Then a reasonable **evaluation for states** in the category is the expected value

$$(0.72 \times +1) + (0.20 \times 0) + (0.08 \times 1/2) = 0.76$$

4.2 Cutting Off Search

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
```

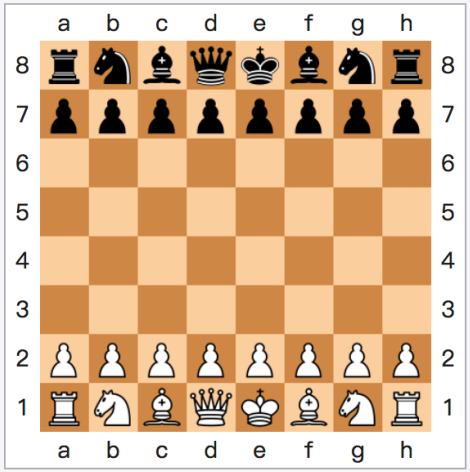
```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

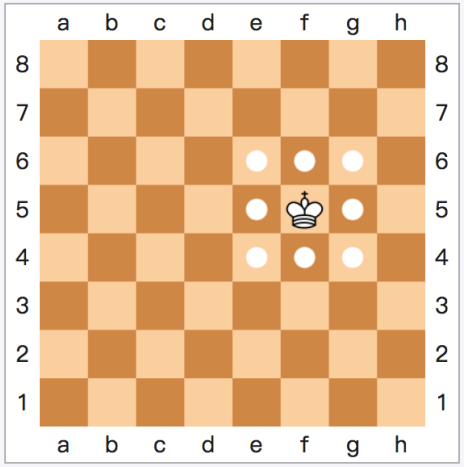
- Modify **ALPHA-BETA-SEARCH** so that it will call the **heuristic EVAL** function when it is appropriate to **cut off the search**
- **Replacement**
  - **TERMINAL-TEST ← CUTOFF-TEST**
  - **UTILITY ← EVAL**

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

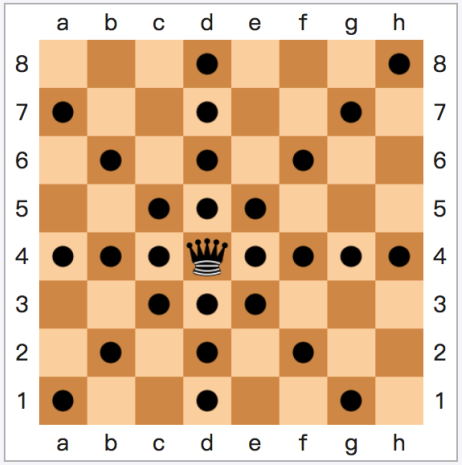
Apply **iterative deepening**, when time runs out, the program returns the **move** selected by the **deepest completed search**



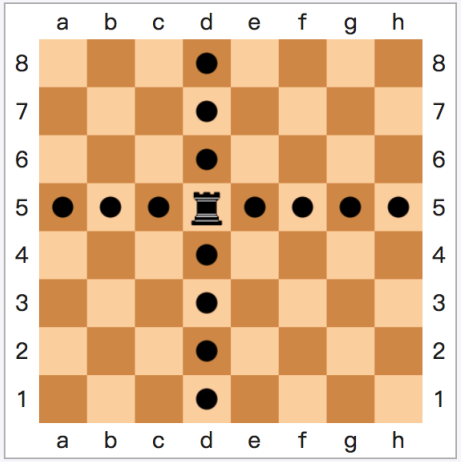
開局時擺法



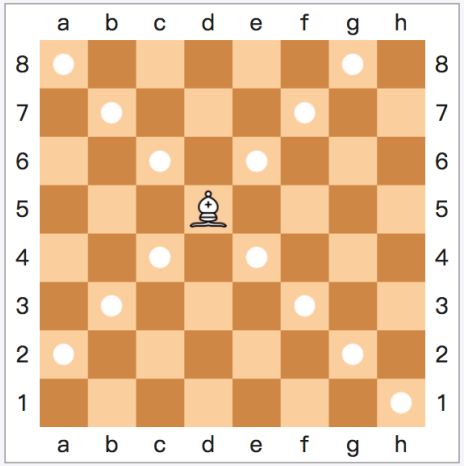
國王的走法



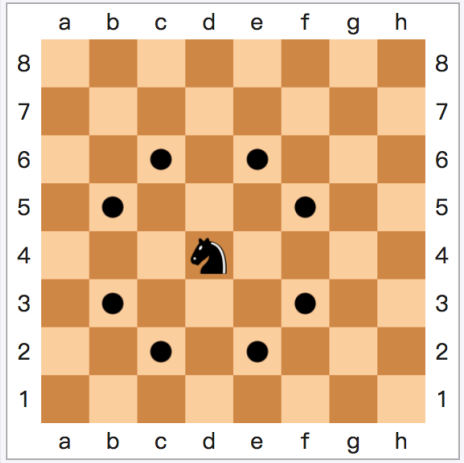
王后的走法



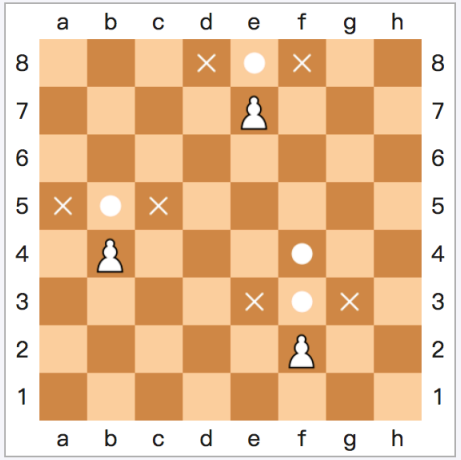
城堡的走法



主教的走法

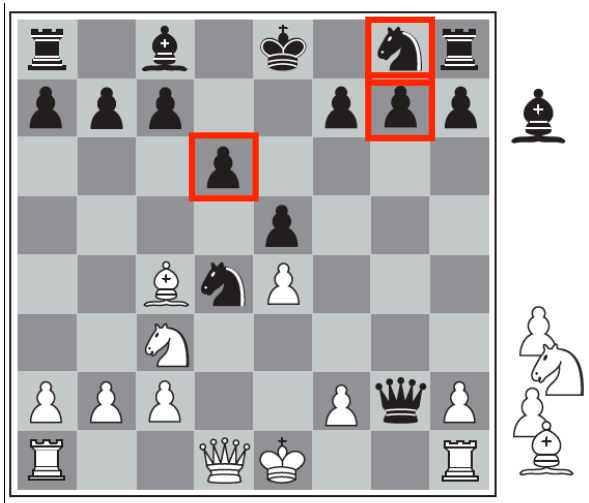


騎士的走法

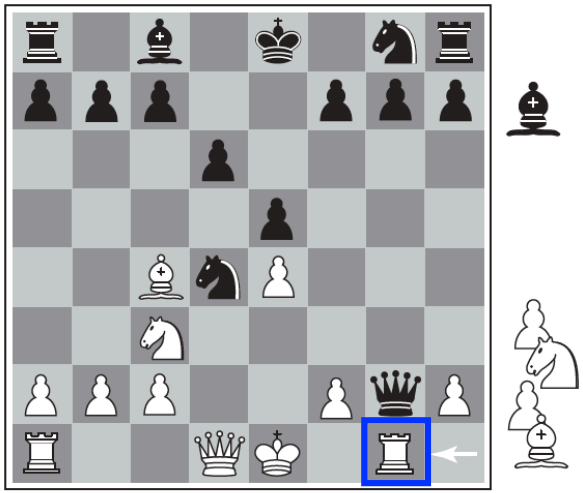


士兵的走法和吃法

- These simple approaches can lead to **errors** due to the **approximate nature** of the evaluation function
- Example
  - Suppose the program searches to the **depth limit**, reaching the position in figure **(a)**, where **Black is ahead** by **a *knight* and two *pawns***
    - It would report this as the **heuristic value** of the state, thereby declaring that the state is a **probable win by Black**
  - But **White's** next move **captures Black's queen** **(b)** with no compensation
    - Hence, the position is really **won for White**, but this can be seen only by **looking ahead one more ply**

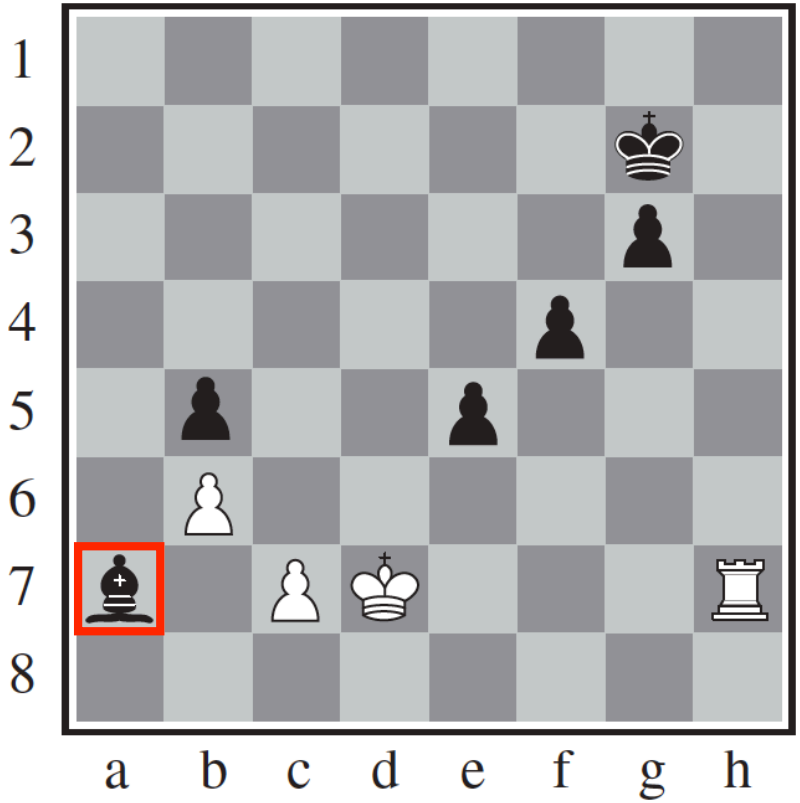


(a) White to move



(b) White to move

- **Horizon effect**
  - **Incorrectly estimate** the value of a state by **overlooking an event** that is just **beyond the depth limit**
  - Example: A program is facing an opponent's move that causes serious damage and is **ultimately unavoidable**, but can be **temporarily avoided** by delaying tactics



- **Horizon effect**
  - With Black to move, the **black *bishop*** is surely **doomed**
  - But Black can forestall that event by **checking the white *king* with its *pawns***, forcing the king to capture the *pawns*. This pushes the **inevitable loss of the *bishop* over the horizon**, and thus the *pawn* sacrifices are seen by the search algorithm as **good moves** rather than bad ones

- **Possible remedies**
  - **Quiescence search**
    - The evaluation function should be **applied only** to positions that are **quiescent**
    - **Nonquiescent positions** can be **expanded further** until quiescent positions are reached
    - **DO NOT cut off** search at positions that are **unstable** – e.g., Are you about to lose an important piece?

- **Singular extension**
  - A move that is "**clearly better**" than all other moves in a given position
  - Once discovered anywhere in the tree in the course of a search, this singular move is **remembered**
  - When the search reaches the **normal depth limit**, the algorithm checks to see if the singular extension is a legal move; if it is, the algorithm allows the move to be considered

- Additional techniques
  - **Transposition table** to **store** previously **expanded states**
    - In many games, **repeated states** occur frequently because of **transpositions** - different permutations of the move sequence that end up in the same position
    - It is worthwhile to **store** the evaluation of the resulting position in a **hash table** the first time it is encountered so that we **don't have to recompute it** on subsequent
    - The **hash table** of previously seen positions is traditionally called a **transposition table**; it is essentially identical to the **explored list** in GRAPH-SEARCH
  - **Forward pruning** to **avoid considering all** possible moves
  - **Lookup tables** for **opening moves** and **endgames**

4.3 Forward Pruning

- Forward pruning does **compromise solution quality** (so is using **EVAL**)
  - Some moves at a given node are **pruned immediately** without further consideration
- **Beam search** is one way to forward pruning
  - On each ply, consider only a "**beam**" of the ***n* best moves** (according to the evaluation function) rather than considering all possible moves, e.g. 20 out of 50 moves
  - Rather **dangerous** because there is **no guarantee that the best move will not be pruned away**

- **ProbCut** (probabilistic cut)
  - A **forward-pruning version** of alpha-beta search that uses **statistics** gained from **prior experience** to lessen the chance that the best move will be pruned
- Alpha-beta search vs. ProbCut
  - **Alpha-beta search** prunes any node that is **provably** outside the current **[**α, β**] window**
  - **ProbCut** prunes nodes by using the scores from previous searches to estimate the **probability** that a node is **outside the [**α, β**] window**

- Combining all the techniques, let us assume we have implemented
  - An **evaluation function** for chess
  - A reasonable **cutoff test** with a **quiescence search**
  - A large **transposition table**
- Let us also assume that, after months of tedious bit-bashing, we can generate and evaluate around **a million nodes per second** on the latest PC (2010), allowing us to search **roughly 200 million nodes** (200 sec) **per move** under standard time controls (**three minutes per move**)
  - The **branching factor** for chess is about **35**, on average, and $\mathbf{35^5}$ is about **50 million**

- If we used **minimax search**, we could look ahead only about **five plies**
  - Though not incompetent, such a program can be fooled easily by an average human chess player, who can occasionally plan six or eight plies ahead
- With **alpha-beta search** we get to about **10 plies**, which results in an **expert level** of play
- To reach **grandmaster** status we would need an **extensively tuned evaluation function** and a **large database of optimal opening and endgame** moves

4.4 Search vs. Lookup

- Many **game-playing** programs use **table lookup** rather than search for the **opening** and **ending** of **games**
  - **Deep search** usually **helps little** at the **beginning**
  - The following technics **help more**
    - **Fast table looking up**
    - **Huge databases**
    - **Statistical analysis**

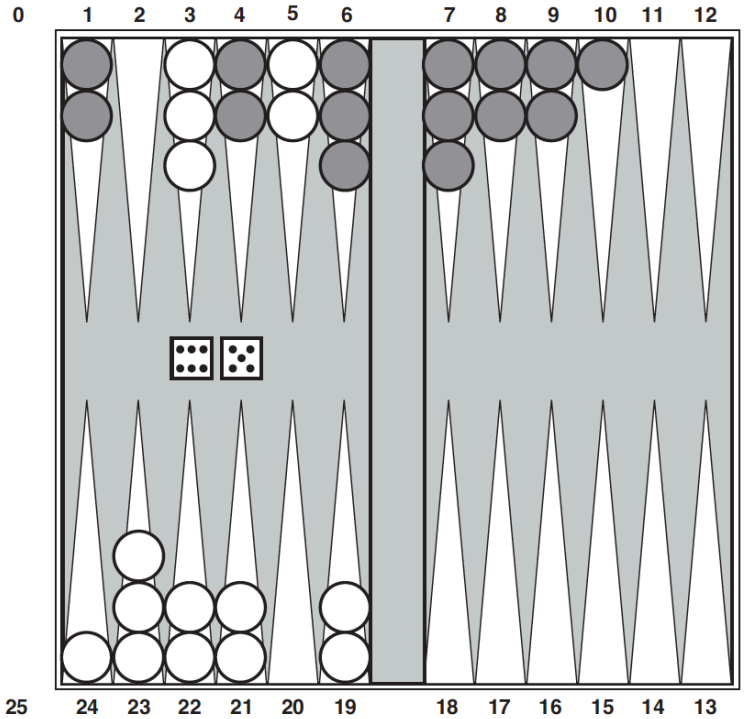
- For the **openings**, there are few choices and the computer is mostly relying on the **expertise of humans**
  - The best advice of human experts on how to play each opening is copied from books and entered into **tables** for the computer's use
  - Computers can also gather **statistics** from a **database** of previously played games to see which **opening sequences** most often lead to a win
- Usually **after ten moves** we end up in a rarely seen position, and the program must **switch from table lookup to search**
- Near the **end of the game** there are again fewer possible positions, and thus more chance to do **lookup**

- But here it is the **computer** that has the expertise: **computer analysis** of **endgames** goes far beyond anything achieved by humans
  - **Human**
    - **King-and-Rook-versus-King (KRK) endgame**
      - Can tell you the **general strategy:** Reduce the opposing king's mobility by squeezing it toward one edge of the board, using your king to prevent the opponent from escaping the squeeze
    - **King, Bishop and Knight vs. King (KBNK) endgame**
      - **Difficult to master** and have no succinct strategy description

- **Computer**
  - Can **completely solve the endgame** by producing a **policy**, which is a **mapping** from **every possible state** to the **best move** in that state
  - Just **look up** the **best move** rather than recompute it anew
  - **How big** will the KBNK lookup table be?
    - There are 462 ways that two *kings* can be placed on the board without being adjacent
    - After the *kings* are placed, there are 62 empty squares for the *bishop*, 61 for the *knight*, and two possible players to move next
    - In sum, there are 462 x 62 x 61 x 2 = 3,494,568 possible positions
    - Continue the search with all 3,494,568 positions are resolved as win, loss, or draw, and you have an infallible lookup table for all KBNK endgames

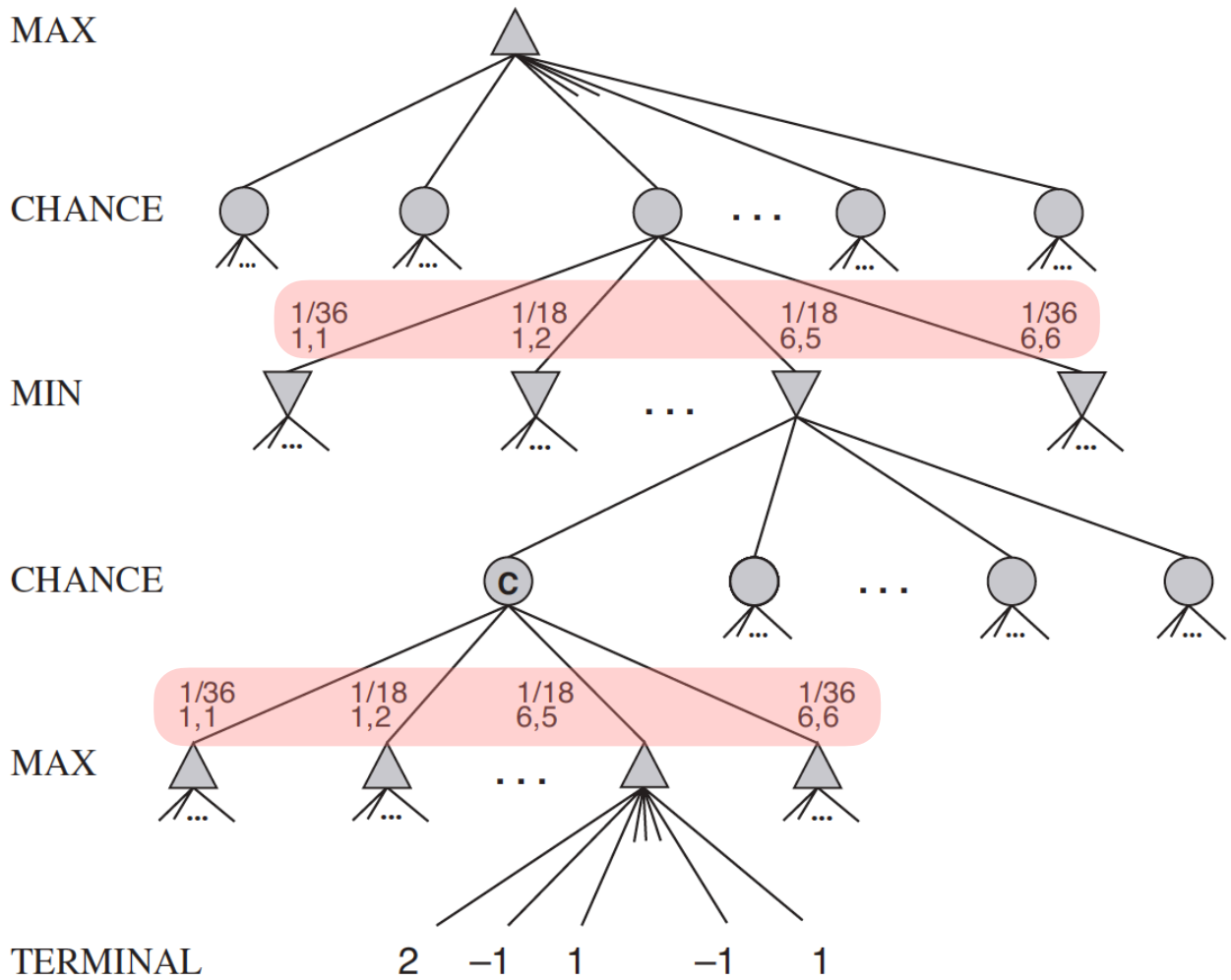
- Bourzutschky (2006) solved all *pawn*-less 6-piece endgames and some 7-piece endgames
  - e.g. 7-piece KQNKRBN endgame requires 517 moves! (**huge complexity**) Table lookup is much faster!

5. Stochastic Games



A typical backgammon position. The **goal** of the game is to **move all one's pieces off the board**. **White** moves **clockwise toward 25**, and **Black** moves **counterclockwise toward 0**. A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over. In the position shown, White has rolled 6–5 and must choose among four legal moves: (5–10,5–11), (5–11,19–24), (5–10,10–16), and (5–11,11–16), where the notation (5–11,11–16) means move one piece from position 5 to 11, and then move a piece from 11 to 16.

- In stochastic games, **chance** induced by dice, card-shuffling
- Introduce **CHANCE nodes** into the minimax tree
- Instead of searching for maximum/minimum values, we now search for **expected maximum/minimum** values



Schematic game tree for a **backgammon position**

- **ExpectiMiniMax** gives **perfect** play assuming an optimal opponent
- Similar to MiniMax, except we must also handle **chance nodes**

$$\textsc{ExpectiminiMax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_a \textsc{ExpectiminiMax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_a \textsc{ExpectiminiMax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \\ \sum_r P(r)\textsc{ExpectiminiMax}(\textsc{Result}(s,r)) & \text{if } \textsc{Player}(s) = \textsc{chance} \end{cases}$$

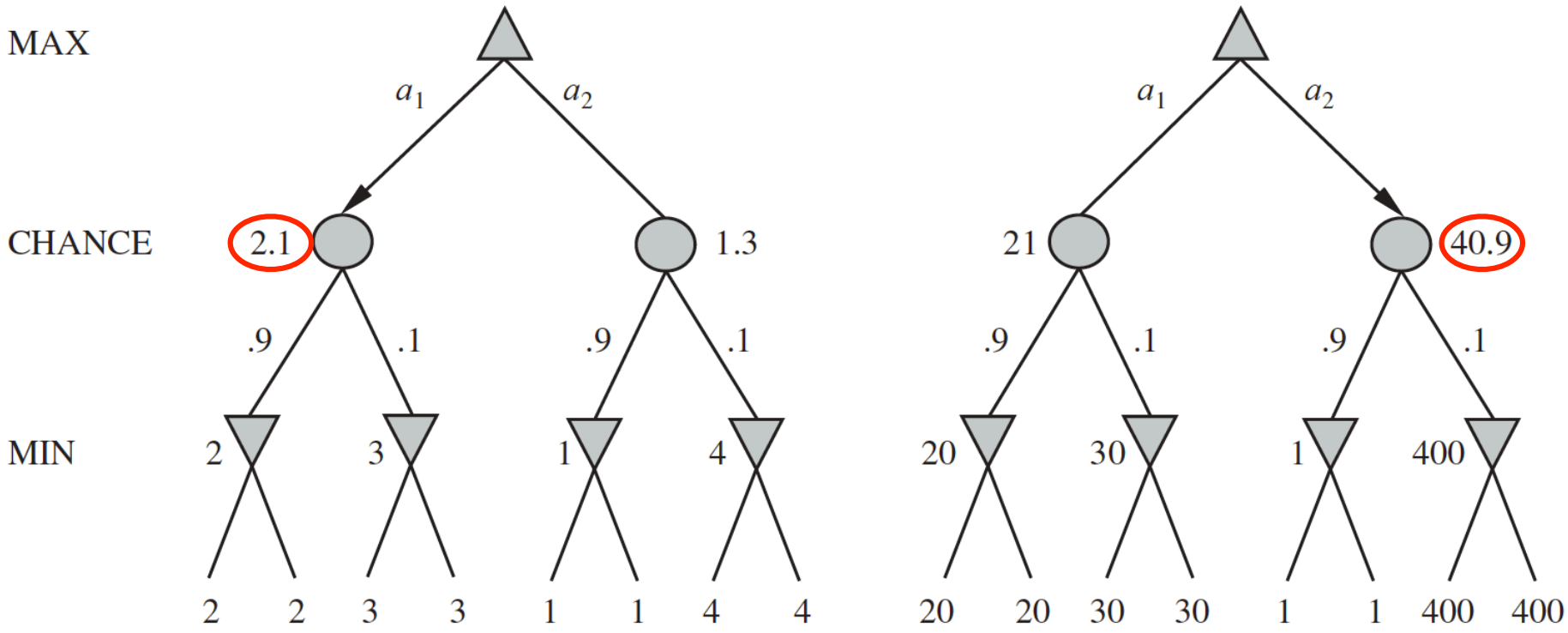
- r: a possible dice roll (or other chance event)
- RESULT (s, r): the same state as s, with the additional fact that the result of the dice roll is r

Games of Chance

- **Expectiminimax**: for chance nodes, average values **weighted** by the **probability** of each outcome
  - Nasty branching factor, defining evaluation functions and pruning algorithms more difficult
- **Monte Carlo simulation**: when you get to a chance node, **simulate** a large number of games with **random dice rolls** and use **win percentage** as evaluation function
  - Can work well for games like Backgammon

5.1 Evaluation Functions for Games of Chance

- Sensitivity to Heuristic
  - We **rarely can** actually use **UTILITY** in ExpectMiniMax
  - Instead, we use **Heuristic** (should be carefully design)
- Heuristic
  - In **MiniMax**, only **relative order** matters
  - In **ExpectiMiniMax**, **actual values** of heuristic matter now
  - e.g. {1, 2, 3, 4} vs. {1, 20, 30, 400}

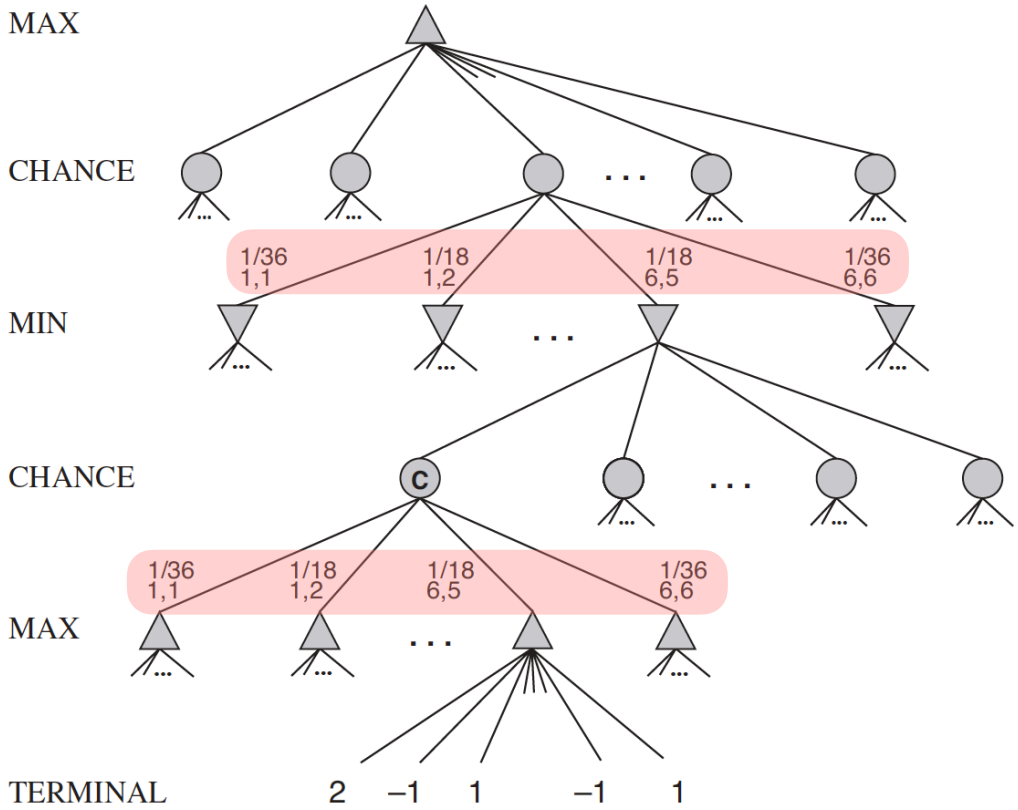


{1, 2, 3, 4} → {1, 20, 30, 400}

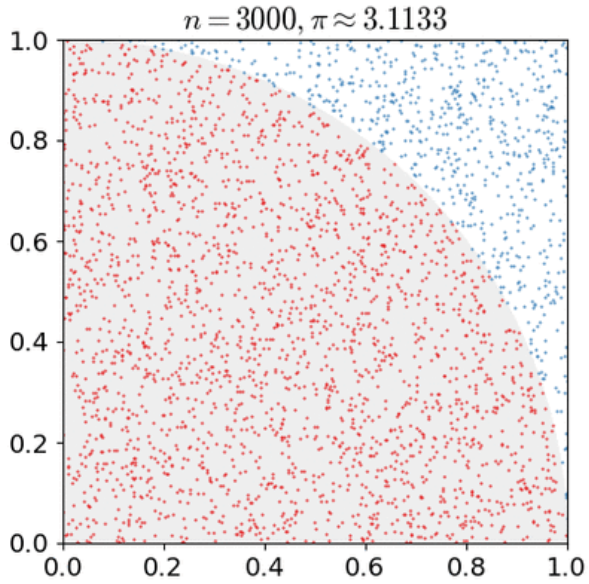
An **order-preserving transformation** on leaf values changes the best move

Performance of ExpectiMiniMax

- α - β pruning now **does not apply** to **max/min nodes**
  - Since chance node **can not** determine which max / min node is the next
- α - β pruning now **still applies** to **chance nodes**
  - Since max / min node **can** determine which chance node is the next



- **Time Complexity**: $O(b^m) \rightarrow O(b^m n^m)$, where n is the number of distinct dice rolls [branch factor: $b \rightarrow bn$]
- Causes ExpectiMiniMax **impractical** in many cases
- **Solution**: Instead of checking every max/min node, adopts **Monte Carlo simulation** at **chance nodes** by using **random dice rolls** to check **only a certain number of paths**
  - e.g. search 10^6 paths to the leaves if time allows instead of searching the whole tree



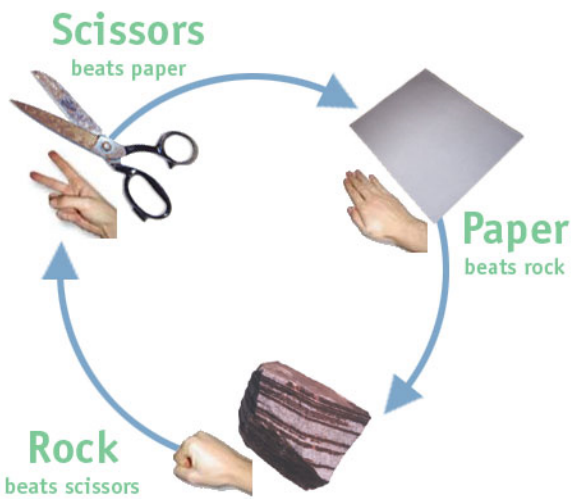
Monte Carlo method applied to approximating the value of π. After placing 30,000 random points, the estimate for π is within **0.07%** of the actual value

6. Partially Observable Games

- Different from stochastic games, **unobservable** parts are usually controlled by **opponents**, not probability
- Examples: Cards held by other player in bridge, folded cards in poker
- Different **strategies** may be applied and **may all considered optimum against different opponents**
- If **equilibrium exists**, it's usually considered as **optimum strategy**

Simultaneous Single-Move Games

- Players must choose their actions at the same time, without knowing what the others will do



Normal form representation:

		Player 1		
		0,0	1,-1	-1,1
Player 2		-1,1	0,0	1,-1
		1,-1	-1,1	0,0

Note: this is a **zero-sum** game

Prisoner's Dilemma

- Two criminals have been arrested and the police visit them separately
- If **one player confesses** and **the other stays silent**, the one who confessed goes **free** and the one who stayed silent gets a **5-year** sentence
- If **both players confess**, they each get a **3-year** sentence
- If **both stay silent**, they each get a **1-year** sentence

	B stays silent	B confesses
A stays silent	Each serves 1 yr	A: 5 yrs; B: free
A confesses	A: free; B: 5 yrs	Each serves 3 yrs

- *A*'s reasoning
  - **Suppose *B* confesses**. Then I get 3 years if I confess and 5 years if I stay silent. So **I should confess!**
  - **Suppose *B* stay silent**. Then I go free if I confess, and get 1 year if I stay silent. So **I should confess!**
- **Dominant strategy**
  - A strategy whose **outcome is better** for the player regardless of the strategy chosen by the other player

	B stays silent	B confesses
A stays silent	Each serves 1 yr	A: 5 yrs; B: free
A confesses	A: free; B: 5 yrs	Each serves 3 yrs

- **Nash equilibrium**
  - A pair of strategies such that **no player can get a bigger payoff by switching strategies**, provided the other player sticks with the same strategy
    - (confess, confess) is a dominant strategy equilibrium

	B stays silent	B confesses
A stays silent	Each serves 1 yr	A: 5 yrs; B: free
A confesses	A: free; B: 5 yrs	Each serves 3 yrs

Summary

- **MiniMax search** for zero-sum two-player games
- **Pruning** techniques enable to search deeper
- Due to time limit, **heuristics** are used to evaluate the "goodness" for a player
- For **stochastic games**, we need to introduce **chance nodes** and search for expected maximum/minimum values
- For stochastic games, α - β pruning is much less efficient, **Monte Carlo simulations** are often adopted to speed up the search
- With **limited observation**, optimality is usually not well defined
  - If **equilibrium** exists, strategies in equilibrium are often considered **optimum**