COE206 – Principles of Artificial Intelligence

Mustafa MISIR

Istinye University, Department of Computer Engineering

mustafa.misir@istinye.edu.tr

http://mustafamisir.github.io
http://memoryrlab.github.io

ARTIFICIAL INTELLIGENCE IN MEDICINE RESEARCH CENTER

ISTINYE UNIVERSITY

MEMORY

L5: Adversarial Search Game Playing[1]

[1] MIT 6.034 Artificial Intelligence (Fall 2010) - Search: Games, Minimax, and Alpha-Beta - https://www.youtube.com/watch?v=STjW3eHOCik

Outline

- Formal Definition
- Optimal Decision in Games (MiniMax)
- Alpha-Beta Pruning
- Stochastic Games
- Partially Observable Games

Games

A **game** can be formally defined as a kind of search problem with the following elements:

- S_0: The **initial state**, which specifies how the game is set up at the start.
- PLAYER(s): Defines which player has the move in a state.
- ACTIONS(s): Returns the set of legal moves in a state.
- RESULT(s, a): The **transition model**, which defines the result of a move.
- TERMINAL-TEST(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.
- UTILITY(s, p): A **utility function** (objective / payoff function), defines the final numeric value for a game that ends in **terminal state** s for a player p

Games – Utility

In chess, the outcome is a win, loss, or draw, with $+1$, 0, or $\frac{1}{2}$.

Some **games** have a wider variety of possible outcomes;

- the payoffs in backgammon range from 0 to $+192$

A **zero-sum game** is defined as one where 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 $0 + 1$, $1 + 0$ or $\frac{1}{2} + \frac{1}{2}$.

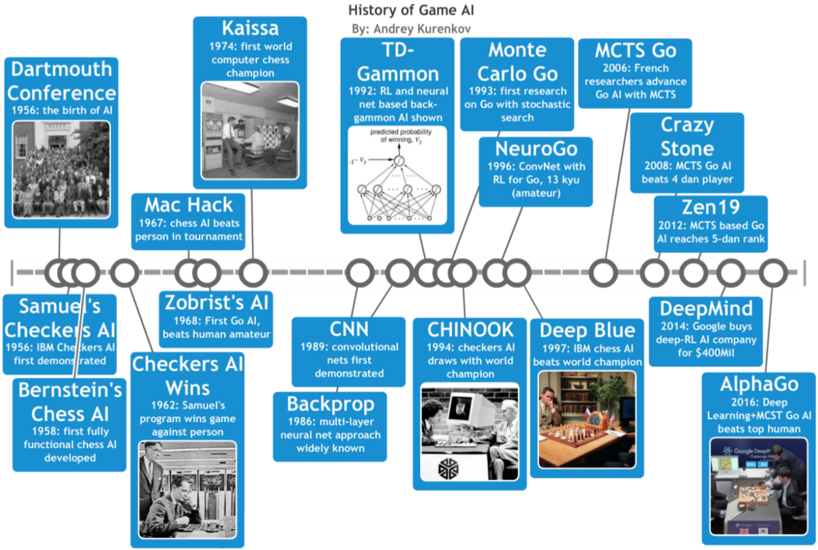
Games – Types

- **Two-player game**: Player A and B. Player A starts.
- **Deterministic**: None of the moves/states are subject to chance (no random draws).
- **Perfect information**: Both players see all the states and decisions. Each decision is made sequentially.
- **Zero-sum**: Player's A gain is exactly equal to player B's loss. One of the player's must win or there is a draw (both gains are equal).

Games – Types, e.g.

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

Games AI – History[2]



[2] image source: https://www.andreykurenkov.com/writing/ai/a-brief-history-of-game-ai/

Adversarial[3] Search

Multiagent environments, in which each agent needs to consider the **actions** of other agents and how they affect its own welfare.

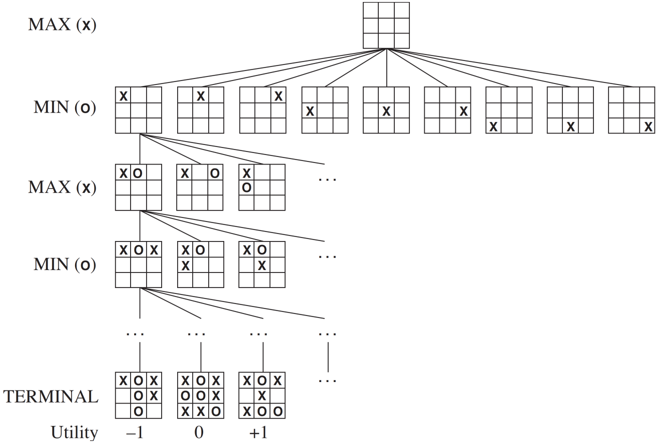
The unpredictability of these other agents can introduce contingencies into the agent's problem-solving process.

In competitive **environments**, in which the agents' **goals** are in conflict, giving rise to **adversarial search** problems — often known as **games**.

[3] involving or characterized by conflict or opposition

Games – Game Tree, e.g. Tic Tac Toe[4]

The **initial state**, **ACTIONS** and **RESULT** functions define the **game tree** for the game—a tree where the nodes are game states and the edges are moves.



[4] A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Games – Game Tree, e.g. Tic Tac Toe

For tic-tac-toe the game tree is relatively small—fewer than $9! = 362,880$ **terminal nodes**.

- From the **initial state**, **MAX** has 9 possible moves.
- Play alternates between **MAX**'s placing an X and **MIN**'s placing an O until we reach leaf nodes corresponding to **terminal states** such that one player has three in a row or all the squares are filled.
- The number on each leaf node indicates the **utility value** of the **terminal state** from the point of view of **MAX**; high values are assumed to be good for **MAX** and bad for **MIN**.

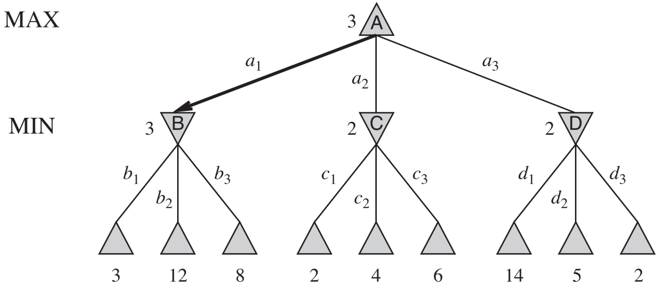
Optimal Decisions in Games

In a normal search problem, the optimal solution would be a sequence of actions leading to a **goal state** – a **terminal state** that is a win.

In **adversarial search**,

- **MIN** has something to say about it.
- **MAX** must find a strategy, which specifies **MAX**'s move in the **initial state**, then **MAX**'s moves in the states resulting from every possible response by **MIN**, then **MAX**'s moves in the states resulting from every possible response by **MIN** to those moves, and so on.

Optimal Decisions in Games – MiniMax[5]



- The possible moves for **MAX** at the root node are labeled a_1, a_2, and a_3
- The possible replies to a_1 for **MIN** are b_1, b_2, b_3, and so on.
- The utilities of the terminal states range from 2 to 14.

[5] Even a simple game like tic-tac-toe is too complex to draw the entire game tree on one page, so we will switch to the trivial game – 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.

Optimal Decisions in Games – MiniMax

Given a **game tree**, the optimal strategy can be determined from the minimax value of each node, which we write as MINIMAX(n).

- The **minimax value** of a node is the **utility** (for **MAX**) of being in the corresponding state, assuming that both players play optimally from there to the end of the game.
- The **minimax value** of a **terminal state** is just its **utility**.
- **MAX** prefers to move to a state of maximum value, whereas **MIN** prefers a state of minimum value.

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

Optimal Decisions in Games – MiniMax[6]

function MINIMAX-DECISION(*state*) **returns** *an action*
  return $\arg\max_{a \,\in\, \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

---

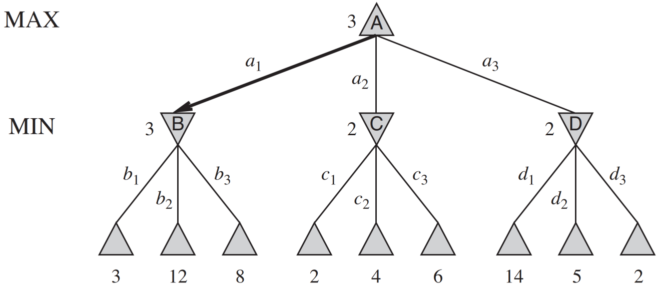
function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v

[6] 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.

Optimal Decisions in Games – MiniMax

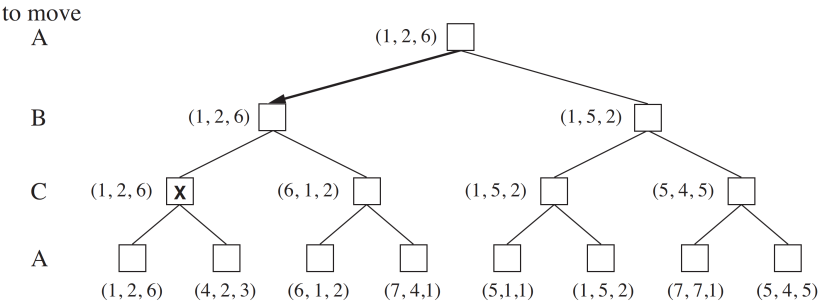


- The **terminal nodes** on the bottom level get their utility values from the game's UTILITY function.
- The first **MIN** node, labeled B, has 3 successor states with values 3, 12, and 8, so its minimax value is 3.
- The other two **MIN** nodes have minimax value 2.
- The root node is a **MAX** node; its successor states have minimax values 3, 2, and 2; so it has a minimax value of 3.

Optimal Decisions in Games – MiniMax, e.g.[8]

With 3 players of A, B and C, we need to replace the single value for each node with a vector of values.

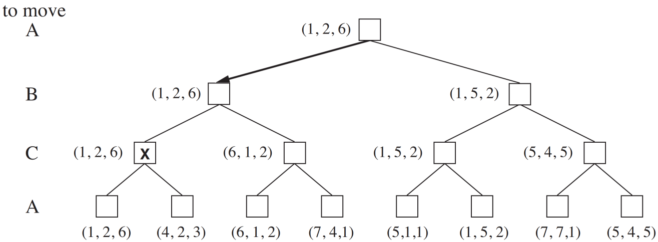
- A vector $\langle v_A, v_B, v_C \rangle$ is associated with each node.
- This vector gives the utility of the state from each player's viewpoint[7].



[7] In two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.

[8] a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Optimal Decisions in Games – MiniMax, e.g.



Consider the node marked X in the game, where player C chooses what to do

- The two choices lead to **terminal states** with **utility** vectors $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$ and $\langle v_A = 4, v_B = 2, v_C = 3 \rangle$
- Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to a **terminal state** with **utilities** $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$

Optimal Decisions in Games – Alliances

Anyone who plays multiplayer games quickly becomes aware that much more is going on than in two-player games.

- Multiplayer games usually involve **alliances**, whether formal or informal, among the players.
- **Alliances** are made and broken as the game proceeds.

e.g. suppose A and B are in weak positions and C is in a stronger position.

- Then it is often optimal for both A and B to attack C rather than each other, lest C destroy each of them individually.
- Collaboration emerges from purely selfish behavior.

As soon as C weakens, the **alliance** loses its value, and either A or B could violate the agreement.

Optimal Decisions in Games – Alliances

If the game is not **zero-sum**, then **collaboration** can also occur with just two players.

- e.g. there is a terminal state with utilities $\langle v_A = 1000, v_B = 1000 \rangle$ and that 1000 is the highest possible utility for each player.
- the optimal strategy is for both players to do everything possible to reach this state—that is, the players will automatically cooperate to achieve a mutually desirable goal.

MiniMax – Properties

Performs a complete **depth-first** exploration of the **game tree**

- **Completeness**[9]: Yes
- **Time Complexity**[10]: $O(b^m)$
- **Space Complexity**[11]: $O(bm)$ (**depth-first** exploration)
- **Optimality**[12]: Yes – against an optimal opponent

Yet, e.g. for chess, $b \approx 35$, $m \approx 100$ for "reasonable" games $\rightsquigarrow$ exact solution completely infeasible

- do we need to explore every path?

9 Is the algorithm guaranteed to find a solution when there is one?

10 How long does it take to find a solution?

11 How much memory is needed to perform the search?

12 Does the strategy find the optimal solution?

Alpha–Beta Pruning[14] – Improving MiniMax

The problem with **minimax search** is that the number of game states it has to examine is exponential in the depth of the tree.

- can't eliminate the exponent, but it can cut it in half.
- The trick is that it is possible to compute the correct **minimax** decision without looking at every node in the game tree

Perform **pruning**[13] – eliminate (large) parts of the tree from consideration:

- **prunes** away branches that cannot possibly influence the final decision

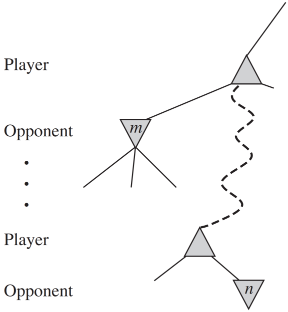
[13] selective removal of certain parts of a plant, such as branches, buds, or roots

[14] https://en.wikipedia.org/wiki/Alpha-beta_pruning

Alpha–Beta Pruning[15]

Consider a node n somewhere in a tree, such that **Player** has a choice of moving to that node.

- If **Player** has a better choice m either at the parent node of n or at any choice point further up, then n will never be reached in actual play.
- So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.



[15] can be applied to trees of any depth, and it is often possible to **prune** entire subtrees rather than just leaves

Alpha–Beta Pruning

Alpha–beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the **path**:

- α: the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for **MAX**
- β: the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for **MIN**

Alpha–Beta Pruning[16]

function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

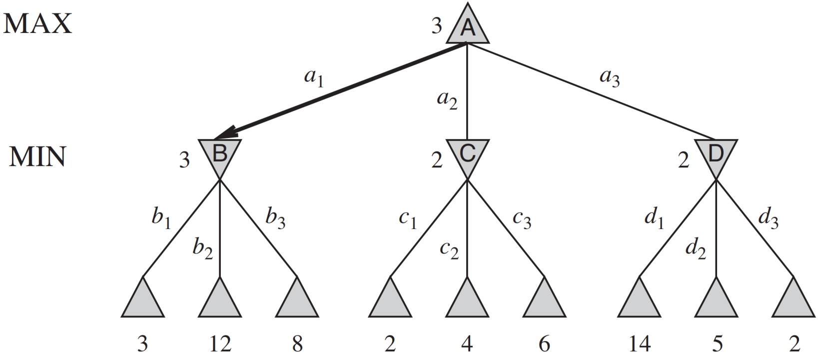
function MIN-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

---

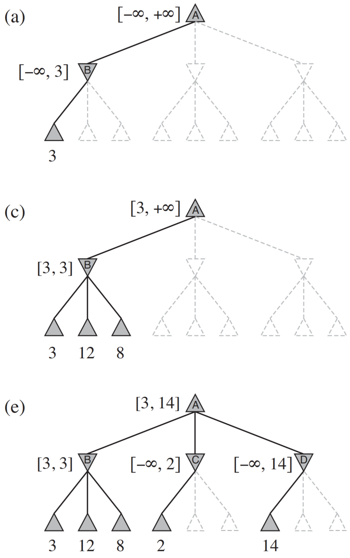
[16] these routines are the same as the MINIMAX functions, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Alpha–Beta Pruning

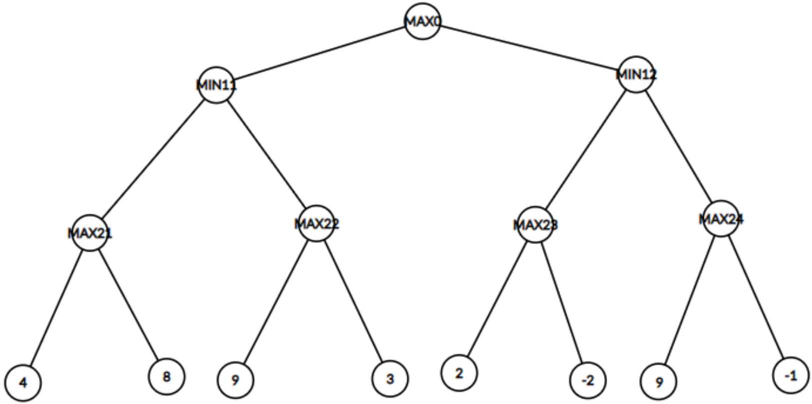
Identify the **minimax** decision without ever evaluating two of the leaf nodes.



Alpha–Beta Pruning



Alpha–Beta Pruning, e.g.[17]



[17] Alpha–Beta Pruning example by John Levine (U. Strathclyde): https://www.youtube.com/watch?v=zp3VMe0Jpf8

Alpha–Beta Pruning – Properties

Pruning preserves **completeness** and **optimality** of original **minimax algorithm**

Degrades **Time Complexity**[18] from $O(b^m)$ to $O(b^{m/2})$

- doubles the depth of search

[18] How long does it take to find a solution?

Imperfect Real-Time Decisions

The **minimax algorithm** generates the entire game search space, whereas the **alpha–beta algorithm** allows us to prune large parts of it.

- However, **alpha–beta** 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.

Imperfect Real-Time Decisions – Evaluation Functions

Suggestion: 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:

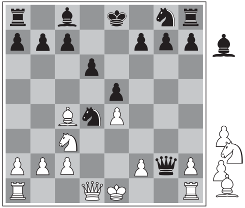
- replace the **utility function** by a **heuristic evaluation function EVAL**, which estimates the position's **utility**
- replace the **terminal test** by a cutoff test that decides when to apply **EVAL**

That gives us the following for **heuristic minimax** for state s and maximum depth d:

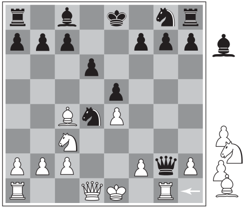
$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN}. \end{cases}$$

Imperfect Real-Time Decisions – Evaluation Functions

An **evaluation function** returns an estimate of the **expected utility** of the game from a given position, just as the **heuristic functions** discussed before, i.e. estimate of the distance to the **goal**.



(a) White to move

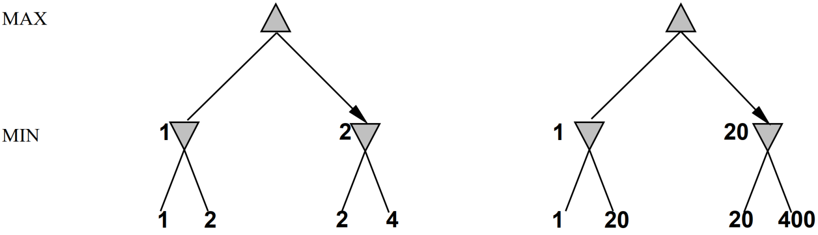


(b) White to move

A linear weighted sum of features **evaluation function** for chess, e.g. $f_1(s) =$ (number of white queens) – (number of black queens)

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

Imperfect Real-Time Decisions – Evaluation Functions



Behaviour is preserved under any **monotonic transformation** of $Eval$

- Only the order matters: payoff in **deterministic** games acts as an **ordinal utility function**

Imperfect Real-Time Decisions – Cutting Off Search

As a simple example, **Terminal-Test** can be replaced by the following statement, in order to stop the search reaching at the tree depth of d

$$\textbf{if } \textsc{Cutoff-Test}(state, depth) \textbf{ then return } \textsc{Eval}(state)$$

Imperfect Real-Time Decisions – Forward Pruning

It is also possible to do **forward pruning** – some moves at a given node are pruned immediately without further consideration.

- One approach to **forward pruning** is **beam search**[19] – consider only a **beam** of the n best moves (according to the **evaluation function**) rather than considering all possible moves.

Of course, such approaches can be rather dangerous because there is no guarantee that the best move will not be pruned away.

[19] https://en.wikipedia.org/wiki/Beam_search

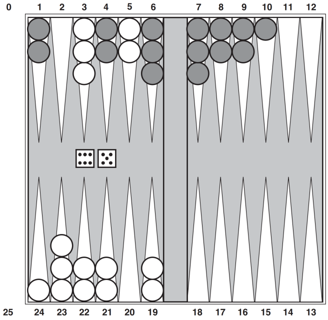
Stochastic Games

Many unpredictable external events can put us into unforeseen situations.

- Many games mirror this unpredictability by including a random element, such as the throwing of dice.

Backgammon is a typical game that combines luck and skill.

- Dice are rolled at the beginning of a player's turn to determine the legal moves.
- e.g. white has rolled a 6–5 and has 4 possible moves.



Stochastic Games

Although White knows what his or her own legal moves are, White does not know what Black is going to roll and thus does not know what Black's legal moves will be.

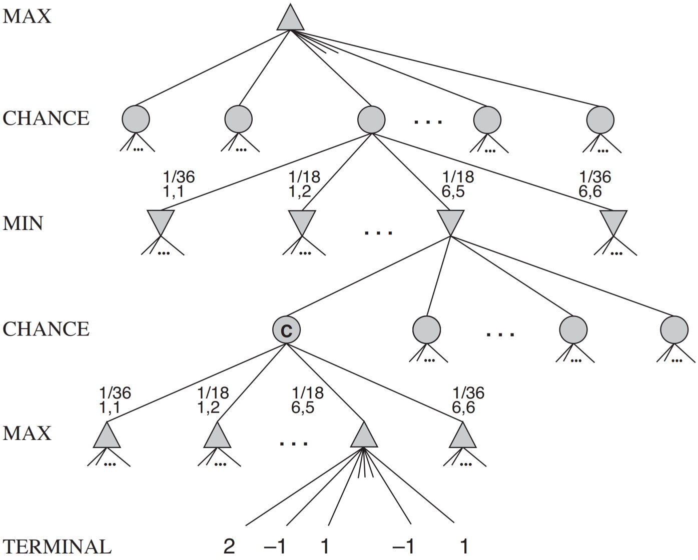
- means that White cannot construct a standard game tree of the sort like in tic-tac-toe.

A **game tree** in backgammon must include **chance nodes** in addition to **MAX** and **MIN** nodes.

- the branches leading from each **chance node** denote the possible dice rolls;
- each branch is labeled with the roll and its probability.

There are 36 ways to roll two dice, each equally likely; but because a 6–5 is the same as a 5–6, there are only 21 distinct rolls.

Stochastic Games[20]



20 **Chance nodes** are shown as circles. The branches leading from each **chance node** denote the possible dice rolls; each branch is labeled with the roll and its probability.

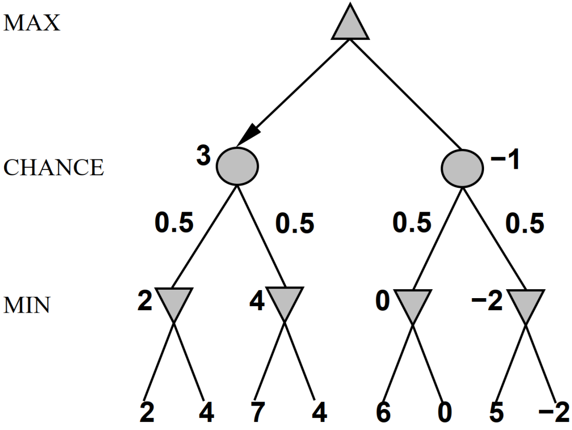
Stochastic Games

Still want to pick the move that leads to the best position without having definite **minimax values**.

- Instead, we can only calculate the **expected value** of a position: the average over all possible outcomes of the **chance nodes**.
- Leads us to generalize the **minimax** value for deterministic games to an **expecti-minimax** value for games with chance nodes.

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

Stochastic Games – Coin Flipping, e.g.

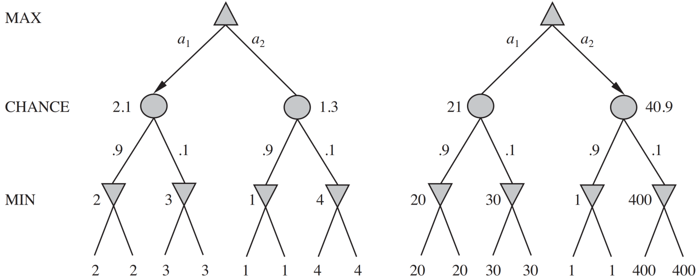


Stochastic Games – Evaluation Functions

The presence of **chance nodes** means that one has to be more careful about what the evaluation values mean.

- with an **evaluation function** that assigns the values [1, 2, 3, 4] to the **leaves** move a_1 is best;
- with values [1, 20, 30, 400], move a_2 is best

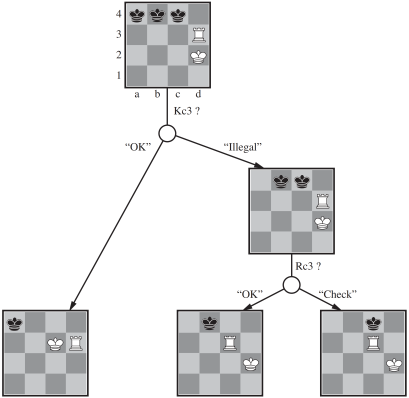
the program behaves totally differently if we make a change in the scale of some evaluation values!



Partially Observable Games

In **deterministic partially observable games**, uncertainty about the state of the board arises entirely from lack of access to the choices made by the opponent

- e.g. the game of Kriegspiel, a **partially observable** variant of chess in which pieces can move but are completely invisible to the opponent.



Partially Observable Games – Card Games

Card games provide many examples of **stochastic partial observability**, where the missing information is generated randomly.

- e.g. in many games, cards are dealt randomly at the beginning of the game, with each player receiving a hand that is not visible to the other players – e.g. bridge, whist, hearts, and some forms of poker.