Adversarial Search and Games

Fundamentals of Artificial Intelligence

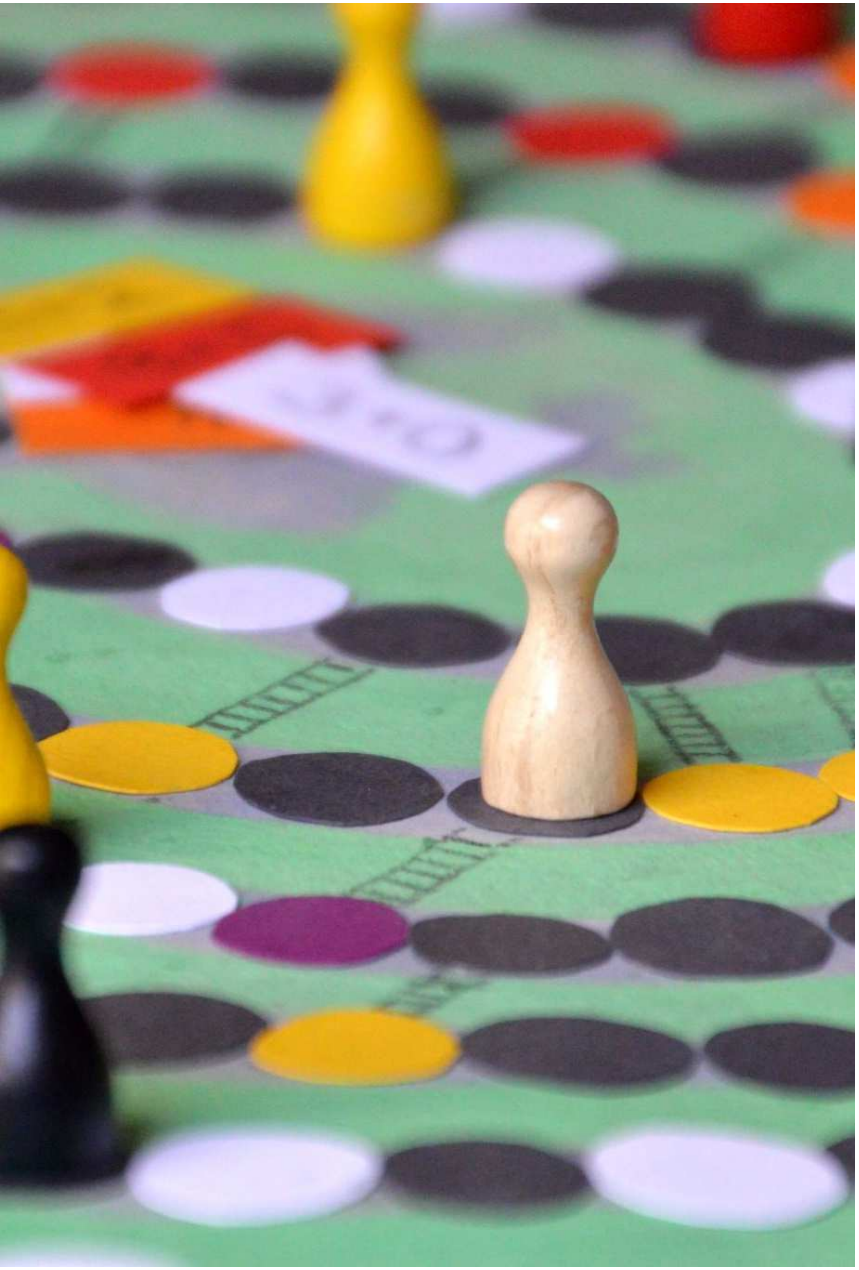
MSc in Applied Artificial Intelligence, 2023-24

Contents

- Topics included
  - Optimal Decisions in Games
  - Alpha–Beta Tree Search
  - Monte Carlo Tree Search

- These slides were based essentially on the following bibliography:
  - Norvig, P, Russell, S. (2021). Artificial Intelligence: A Modern Approach, 4th Edition. Pearson, ISBN-13: 978-1292401133

Optimal Decisions in Games



AI and games

- Working with games is extremely interesting
  - Easy to test new ideas!
  - Easy to compare agents with other agents
  - Easy to compare agents with humans
- Games illustrate several interesting points of AI
  - Perfection is unattainable => you have to approximate!
  - Good idea to think about what to think about!
  - Uncertainty restricts assignment of values to states!
- Games are the test tube for AI

Games as search problems

- **Multi-agent** environments
  - Hostile Agent (adversary) included in the world!
  - Unpredictable Opponent → Solution is a Contingency Plan
  - Time limit → Unlikely to find goal → Approach required

- Chess game takes intelligence to play — simple rules but the game is complex
  - World is fully observable to the agent
  - There are 10^40 legal positions — the average branching factor of 35
    - Match with 50 moves => 35^100 leaves in the search tree!

- In this section will discuss the optimal decisions in games
  - In next sections will see Alpha–Beta Tree Search and Monte Carlo Tree Search (MCTS)
  - We will not address **Stochastic Games** and **Partially Observable Games**

Types of games

- Information Games
  - **Perfect information**: Chess, Checkers, Go, Othello, Backgammon, Monopoly
  - **Imperfect information**: Poker, Scrabble, Bridge, King

- Games of Chance/Deterministic:
  - Deterministic: Chess, Checkers, Go, Othello
  - Games of chance: Backgammon, Monopoly, Poker, Scrabble, Bridge, King

- Plan of "Attack":
  - Algorithm for the perfect game
  - Tree pruning to reduce costs
  - Finite horizon, approximate evaluation

Multi-agent environments

Different **approaches** can be adopted

- When there are a very large number of agents, we can consider them in the aggregate as an **economy**,
  - We try to understand the whole, without predict the action of any individual agent
- We can consider adversarial agents as just a part of the environment that makes it nondeterministic.
  - But we should keep in mind the idea that our adversaries are actively trying to defeat us
- We can explicitly model the adversarial agents with the techniques of **adversarial game-tree search**.
  - **Minimax search** is part of a restricted class of games algorithm that help to find the optimal move
  - It uses **pruning to make the search more efficient**

- For nontrivial games — no time to be sure of finding the optimal move
  - For each state where we choose to stop searching, we ask who is winning applying a **heuristic evaluation** function to estimate (Imperfect information — partially observable) who is winning
  - Alternatively, we can **average the outcomes** of many fast simulations of the game from that state all the way to the end (Monte Carlo)

Two-player zero-sum games

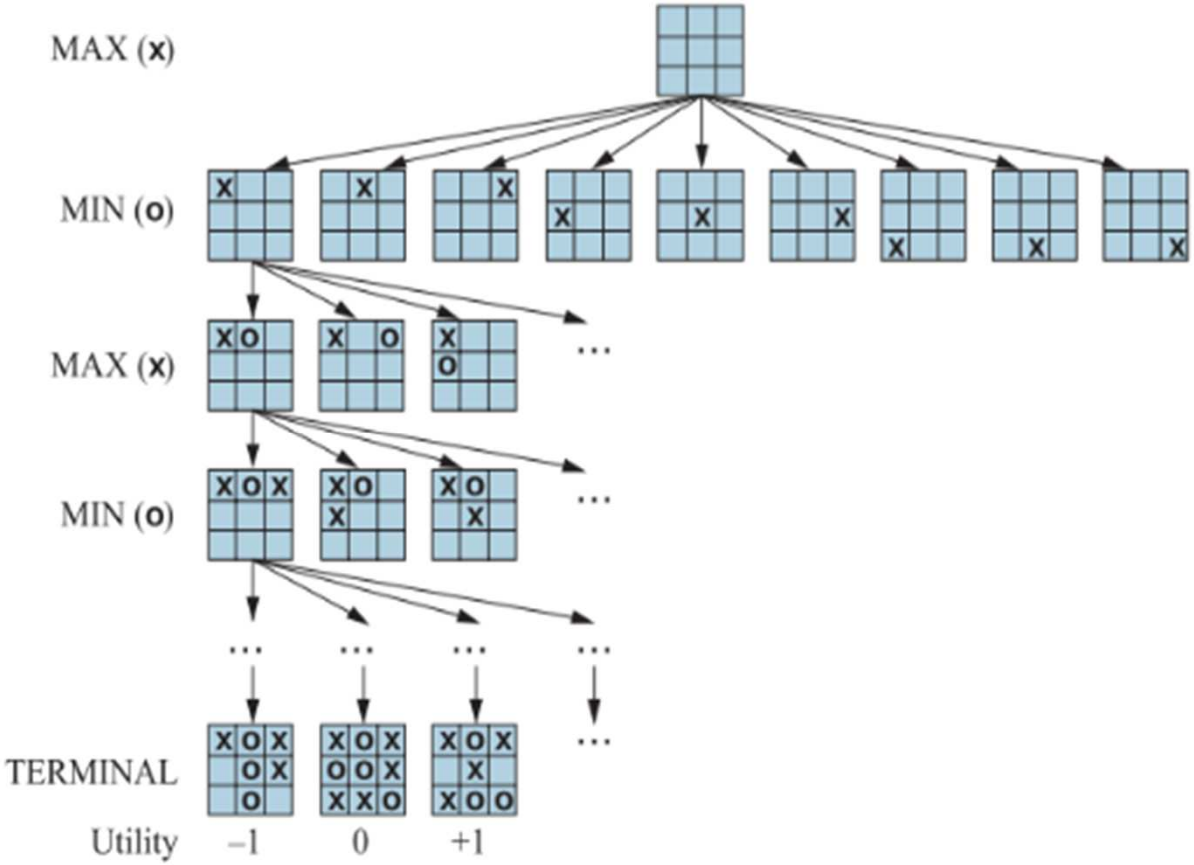
- The games most commonly studied within AI are what game theorists call **deterministic**
  - two-player, turn-taking, perfect information, zero-sum games.
  - Perfect information synonym for "fully observable"
  - zero-sum means that what is good for one player is just as bad for the other
  - For games we often use the term move as instead of “action” and position instead of “state.”
  -
- We will call our **two players MAX and MIN**, for reasons that will soon become obvious.
  - MAX moves first, and then the players take turns moving until the game is over.
  - At the end of the game, points are awarded to the winning player and penalties are given to the loser.

A game can be defined with the following elements:

- **State 0 (s0)**: The initial state, which specifies how the game is set up at the start.
- **TO-MOVE(s)**: The player whose turn it is to move in state s.
- **ACTIONS(s)**: The set of legal moves in state *s*.
- **RESULT(s, a)**: The transition model, which defines the state resulting from taking action a in state *s*
- **IS-TERMINAL(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 which defines the final numeric value to player *p* when the game ends in terminal state *s*.

Tic-tac-toe game

- From the initial state, MAX has nine possible moves.
- Play alternates between MAX's placing an **X** and MIN's placing an **O** until we reach leaf nodes
- **Leaf nodes** correspond to terminal states — one player has three squares 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 good for MAX and bad for MIN (which is how the players get their names).
- There are 9! = 362,880 terminal nodes, but only 5,478 distinct states



Optimal decisions in games

- MAX's strategy must be a **conditional plan**
  - a contingent strategy specifying a response to each of MIN's possible moves.
  - In games with a win/loose outcome, we could use AND–OR search to generate the conditional plan.
  - For games with multiple outcome scores, we need a slightly more general algorithm called minimax search

- Consider the trivial game in next slide's figure
  - The possible moves for MAX at the root node are labeled a1, a2, a3.
  - The possible replies to a1 for MIN are b1, b2, b3, and so on.
  - This particular game ends after one move each by MAX and MIN.
  - The utilities of the terminal states in this game range from 2 to 14.

- The optimal strategy can be determined by working out the minimax value of each state = MINIMAX(s)
  - The minimax value is the **utility (for MAX) of being in that state**, assuming that both players play optimally from there to the end of the game.

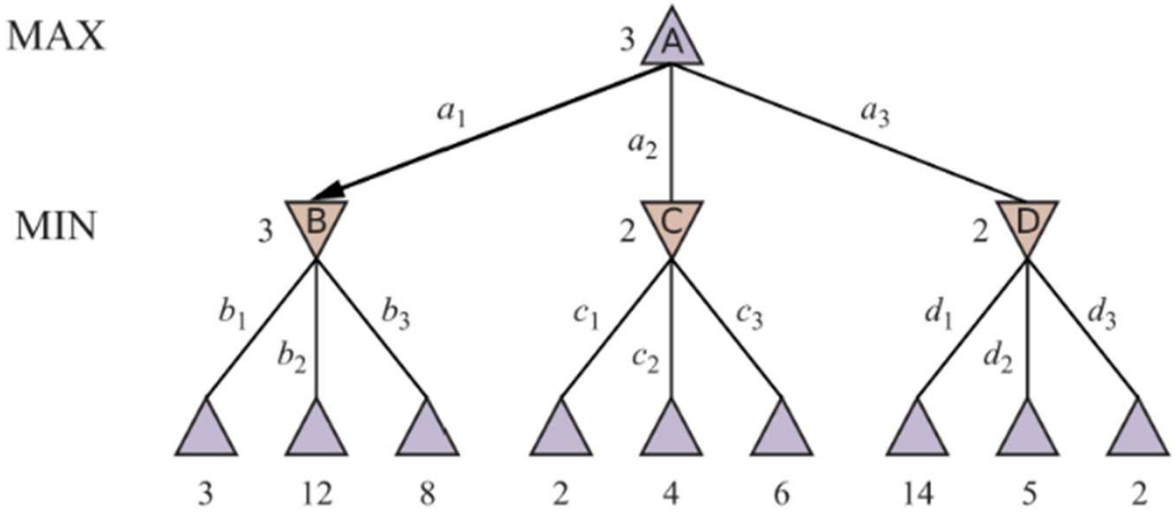
Minimax algorithm

Strategy

- Choose the move that has the highest minimax value: best that can be achieved against the opponent's best responses!

Action a1 is the optimal choice for MAX in the figure that represents a **two-ply game tree**

- The first MIN node (B), has three successor states (3, 12, 8) so its minimax value is 3.
- The other two MIN nodes have minimax 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.



$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

The minimax search algorithm

It is a recursive algorithm

1. Proceeds all the way down to the leaves of the tree and then
2. backs up the minimax values through the tree as the recursion unwinds.

```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

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

Minimax properties

- First, as shown In the next figure,
  - the algorithm recurses down to the three bottom-left nodes
  - next, it uses the UTILITY function on them to discover the values (3, 12, 8)
  - then, it takes the minimum of these values, 3, and returns it as the backed-up value of node B
  - A similar process gives the backed-up values of 2 for C and 2 for D.
- Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node.

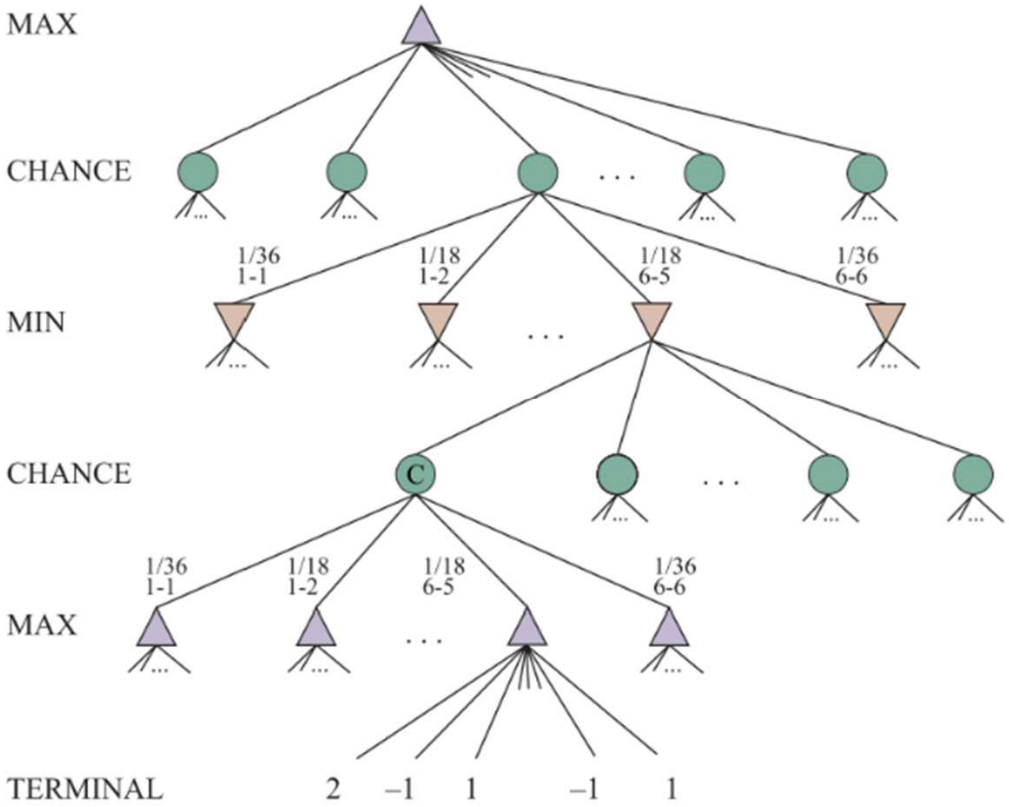
- Complete? Yes, if the tree is finite!
- Optimal? Yes, against an optimal opponent! Otherwise?
- Complexity in Time? O(b^m)
- Complexity in Space? O($b \times m$), first in depth

- Problem: Infeasible for any minimally complex game
  - Example: Chess (b=35, m=100)
    b*m = 35^100 = **2,5x10^154**
  - Assuming 450 million hypotheses are analysed per second, it would take 2x10^138 years to get to the solution!

Games of chance or Stochastic

- In many games, unlike chess, there are external events that affect the game, such as drawing a card or rolling a die!
- Examples: Card games, Backgammon, Scrabble, ...
- Search trees must include chance nodes. Decision is made based on expected value: **ExpectMiniMax**

- Left figure shows the schematic game tree for a backgammon position



Deterministic games

- Checkers
  - Chinook, a computer program developed at the University of Alberta, ended the 40-year reign of human champion Marion Tinsley in 1994.
  - It used a database for match endings defining the perfect way to win for all positions involving 8 or fewer pieces (in total 443 748 401 247 positions).
- Chess
  - Deep Blue, developed by IBM company, defeated the human world champion Gary Kasparov in a 6-game match in 1997
  - It searches 200 million positions per second and uses an extremely sophisticated evaluation function and methods to extend some search lines beyond depth 40!
- Go
  - Go was considered beyond the reach of the most sophisticated computers
  - In 2016, the human champion, Lee Se-dol, was defeated by AlphaGo AI System https://www.theverge.com/2019/11/27/20985260/ai-go-alphago-lee-se-dol-retired-deepmind-defeat

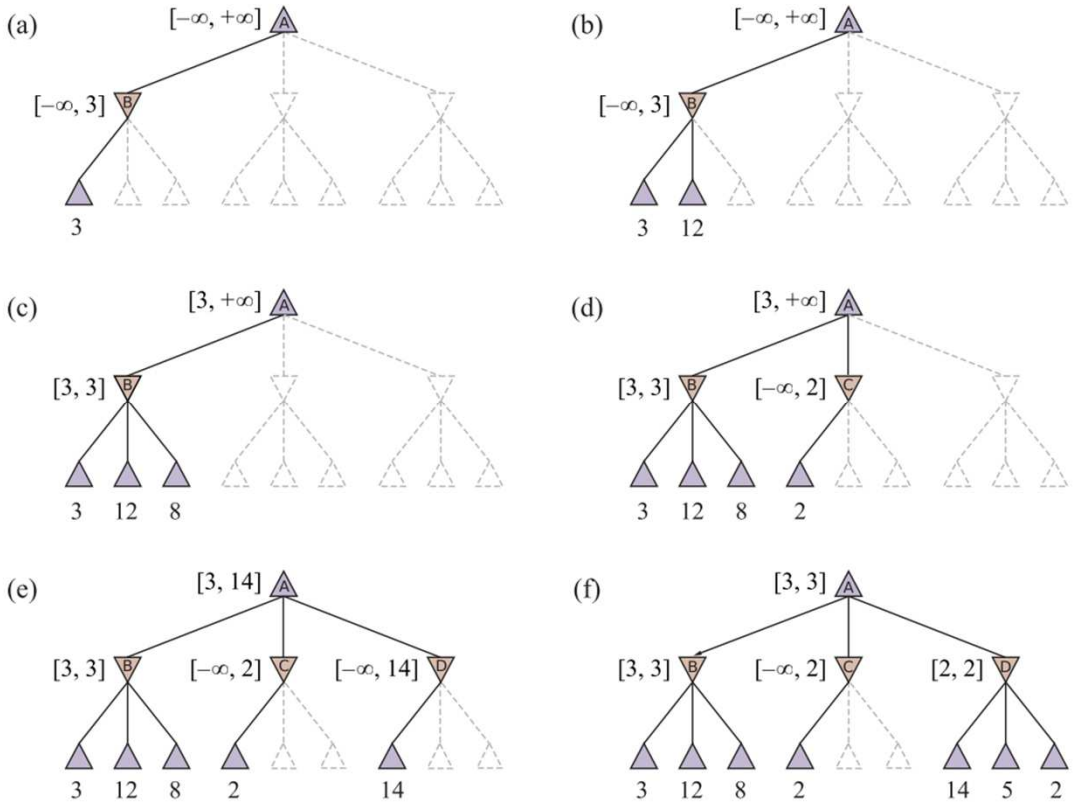
Alpha-beta pruning

- In many deterministic games, the number of game states is exponential in the depth of the tree
  - No algorithm can completely eliminate the exponent
  - Sometimes we can cut it in half, **computing the correct minimax decision without examining every state** by pruning large parts of the tree that make no difference to the outcome.
- Consider again the previous two-ply game tree.
  - Calculate the optimal decision paying careful attention to what we know at each point in the process
  - The steps are explained in next slide's Figure .
- The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.
  - Let the two unevaluated successors of node C have values x and y.
  - Then the value of the root node is given by

$$\begin{aligned} MINIMAX(root) &= max(min(3,12,8),\mathbf{min(2,x,y)},min(14,5,2)) \\ &= max(3,min(2,x,y),2) = max(3,\ z,2),\ \ where\ z = min(2,x,y) \le 2 \\ &= 3 \end{aligned}$$

  - The minimax decision are independent of the values of the leaves x and y, so they can be pruned.

Alpha-beta pruning (2)

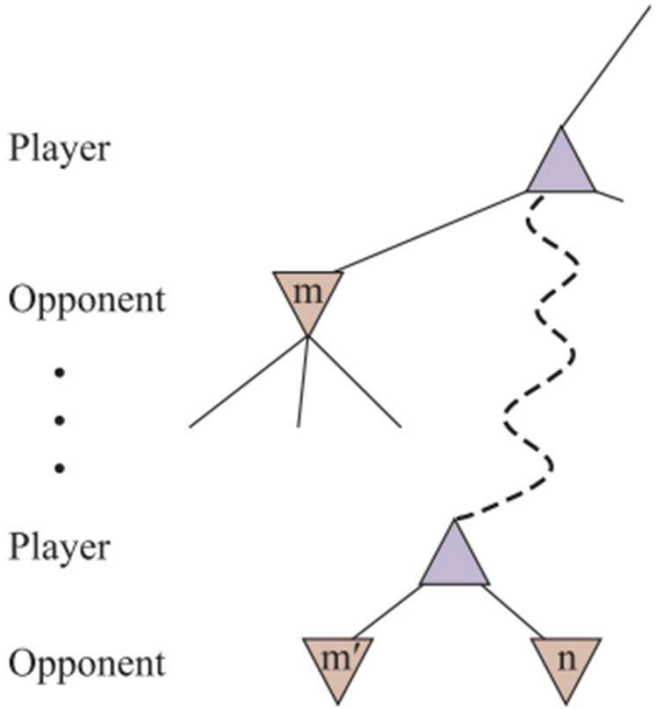


- Alpha–beta pruning can be applied to trees of any depth
- It is often possible to prune entire subtrees rather than just leaves
- Minimax search is **depth-first**,
  - Just considers a single path in the tree.

- Alpha–beta pruning in the function MAX-VALUE(state,α,β)
  - 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**. Think: α = "at least."
  - 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**. Think: β = "at most."

Alpha-beta pruning (3)

- Alpha-Beta cuts do not affect the final result
- Good **sorting improves the efficiency of the cuts**
  - If m or m' is better than n for Player, we will never get to n in play (figure)
  - Should try to first examine the successors that are likely to be best.
  - The best moves are known as killer moves, and to try them first is called the **killer move heuristic**.
- Instead of O(b^m) for minimax, with perfect sorting
  - We get **time complexity = O($b^{m/2}$)**
  - Doubles the search depth (Depth 8 => Good chess player)
- Repeated states can occur because of transpositions
  - Different permutations of the move sequence that end up in the same position
  - The problem can be addressed with a **transposition table** that caches the heuristic value of states.



Alpha–Beta Tree Search

Heuristic Alpha–Beta Tree Search

- Even with alpha–beta pruning and clever move ordering, minimax won't work for games like chess and Go, because there are still too many states to explore in the time available.

- Claude Shannon (1950) proposed two strategies
  - **Type A strategy**, used often in chess, considers all possible moves to a certain depth in the search tree
    - It explores a wide but superficial portion of the tree
  - **Type B strategy** ignores moves that look bad and follows promising lines "as far as possible."
    - t explores a deep but narrow portion of the tree
    - Go playing programs use Type B because of its higher branching factor

- We **can cut off the search early**, treating nonterminal nodes as if they were terminal (**depth bound**)
  - UTILITY function is replaced by a heuristic Evaluation function, which estimates a state's utility
  - Terminal test is replaced by a cutoff test, which decides when to cut off the search and return true for terminal states

Evaluation functions

- A heuristic evaluation function EVAL(s, p) returns an estimate of the expected utility of state *s* to player *p*.
  - For terminal states, it must be that EVAL(s, p) = UTILITY(s, p)
  - For nonterminal states, the evaluation must be somewhere between a loss and a win: UTILITY(loss, p) ≤ EVAL(s, p) ≤ UTILITY(win, p).
- Characteristics of a **good evaluation** function
  - the computation must not take too long!
  - the evaluation function should be strongly correlated with the actual chances of winning
- **Evaluation functions** work by calculating various **features of the state**
  - In chess, the features are the number of white pawns, black pawns, white queens, black queens, etc.
  - The features, taken together, define various categories or equivalence classes of states
  - The states in each category have the same values for all the features.
- Evaluation function should only be applied to stable positions in terms of their value
  - For example, positions with possible captures should be explored further...

Evaluation functions (2)

- The **evaluation function returns a single value** that estimates the proportion of states with each outcome.
  - E.g., suppose our experience suggests that 82% of the states encountered in the two-pawns versus one-pawn category lead to a win (utility +1); 2% to a loss (0), and 16% to a draw (1/2).
  - Then an evaluation for states in the category is the **expected value**: (0.82×+1)+(0.02×0)+ (0.16×1/2) = 0.90.
  - This analysis requires too many categories and hence too much experience to estimate all the probabilities.
  - Instead, most evaluation functions compute separate numerical contributions from each feature and then combine them to find the **total value**.

- In chess is used an approximate **material value** for each piece:
  - each pawn is worth 1, a knight or bishop is worth 3, a rook 5, and the queen 9
  - Other features such as “good pawn structure” and “king safety” might be worth half a pawn
  - These feature values are then simply added up to obtain the evaluation of the position.
  - **Assumption**: the contribution of each feature is independent of the values of the other features!!

Cutting off search

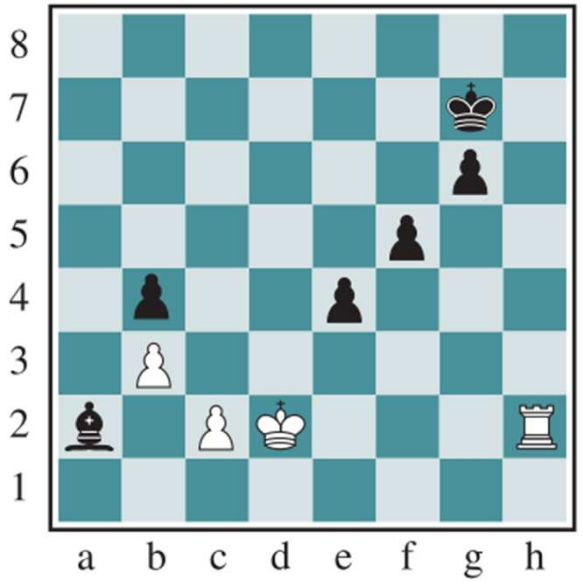
- The ALPHA-BETA-SEARCH can be modified so that it will call the heuristic Evaluation function when it is appropriate to cut off the search
  - The two lines in the **minimax search algorithm** that mention IS-TERMINAL are replaced with the line:

    if *game.IS-CUTOFF(state, depth)* ***then return*** *game.EVAL(state, player), null*

- The current depth is incremented on each recursive call.
  - It is defined a **fixed depth limit**, so that IS-CUTOFF(state, depth) returns true for all depth greater than some fixed depth ***d*** (as well as for all terminal states).
  - The depth ***d*** is chosen so that a move is selected within the allocated time

- A more robust approach is to apply **iterative deepening**
  - When time runs out, the program returns the move selected by the deepest completed search.
  - If in each round of iterative deepening we keep entries (state, evaluation) in the **transposition table**,
  - Subsequent rounds will be faster, and we can use the evaluations to improve move ordering.

Horizon effect



- The evaluation function should be applied only to **quiescent/calm** positions
  - positions in which there is no pending move that would wildly swing the evaluation.
  - For non-quiescent positions, the IS-CUTOFF is false, and the search continues until quiescent positions are reached
- The **horizon effect** is more difficult to eliminate
  - It arises when the program is facing an opponent's move that causes serious damage and is ultimately unavoidable, but can be temporarily avoided by the use of delaying tactics.
  - In the figure, It is clear that there is no way for the black bishop to escape. E.g., the white rook can capture it by moving to h1, then a1, then a2; a capture at depth 6 ply.

- One strategy to mitigate the horizon effect is to allow a few **singular extensions**, moves that are "clearly better" than all other moves in a given position, even when the search would normally be cut off at that point.
  - E.g. Black will also consider the sequence of moves that starts by checking the king with a pawn
  - The strategy does not add many total nodes to the tree, and has proven to be effective in practice.

Forward pruning

- *Alpha–beta pruning* prunes branches that can have no effect on the final evaluation (Type A strategy)
- **Forward pruning** prunes moves that appear to be poor moves but might possibly be good ones
  - the strategy saves computation time at the risk of making an error (Type B strategy)
  - Most human chess players do this, considering only a few moves from each position
- **Beam search** is one approach to forward pruning
  - on each ply, consider only a "beam" of the n best moves rather than considering all possible moves
  - This approach does not guarantee that the best move will not be pruned away.
- **Late move reduction** works under the assumption that move ordering has been done well
  - But rather than pruning completely, we just reduce the depth to which we search to save time.
  - If we get a value above the current α value in reduced search, we can re-run the search with the full depth

- Minimax search in not enough efficient to play chess, so we need combine several techniques
  - It just allows, in acceptable conditions, to achieve Depth 4 to 6, equivalent to a beginner level
  - A good human player achieves depth 8 – Kasparov could achieve depth 12

Search versus lookup

- Many chess programs use **table lookup** rather than search a tree of a billion game states to start a game
  - For the openings, the computer mostly relies on the expertise of humans
  - For the first few moves there are few possibilities, and most positions will be in the table
- We can also use **statistics of previously played games** to see which **opening sequences** most often lead to a win
  - After 10 or 15 moves we got a rarely seen position, and we must switch from table lookup to search.
- **Near the end of the game** there are again fewer possible positions, and thus it is easier to do lookup.
  - But here it is the computer that has the expertise
  - Computer analysis of the chances to end the game goes far beyond human abilities. Why?

Monte Carlo Tree Search

Monte Carlo tree search (MCTS)

- The game of Go illustrates two major weaknesses of heuristic alpha–beta tree search
  - Go has a branching factor that starts at 361, so alpha–beta search would be limited to 4 or 5 ply
  - It is difficult to define a good evaluation function

- Modern Go programs use a strategy called **Monte Carlo tree search** (MCTS)
  - The basic MCTS strategy does not use a heuristic evaluation function to estimate the value of a state
  - It is estimated as the average utility over simulations of complete games starting from the state

- A simulation, also called a **playout** or **rollout**,
  - Chooses moves first for one player
  - Then for the other, repeating until a terminal position is reached.
  - At that point the rules of the game determine who has won or lost, and by what score.
- For games in which the only outcomes are a win or a loss, “average utility” is the same as “win percentage.”

Playout police

- We need to decide two things
  - From what positions do we start the playouts?
  - How many playouts do we allocate to each position?

- The simplest answer, called pure **Monte Carlo search**
  - Do N simulations starting from the current state of the game
  - Track which of the possible moves from the current position has the highest win percentage
- For some stochastic games this converges to optimal play as N increases,
  - When it is not sufficient, we need a selection policy that selectively focuses the computational resources on the important parts of the game tree.
- To get a more accurate estimate of their value, it balances two factors
  - **Exploration** of states that have had few playouts
  - **Exploitation/usage** of states that have done well in past playouts.

MCTS Steps

- MCTS maintains a search tree and growing it on each iteration of the following four steps
  - **Selection** – starting at the root of the search tree, we choose a move (guided by the selection policy), leading to a successor node, and repeat that process, moving down the tree to a leaf
  - **Expansion** – We grow the search tree by generating a new child of the selected node
  - **Simulation** – We perform a playout from the newly generated child node, choosing moves for both players according to the playout policy. These moves are not recorded in the search tree.
  - **Back-propagation** – We now use the result of the simulation to update all the search tree nodes going up to the root.
- These four steps are repeated either for a set number of iterations, or until the allotted time has expired
  - In each iteration is returned the move with the highest number of playouts

- One very effective selection policy is called "**upper confidence bounds** applied to trees" or UCT
  - The policy ranks each possible move based on an upper confidence bound formula called UCB1
  - The node with the most playouts is almost always the node with the highest win percentage
  - The selection process favors win percentage more and more as the number of playouts goes up.

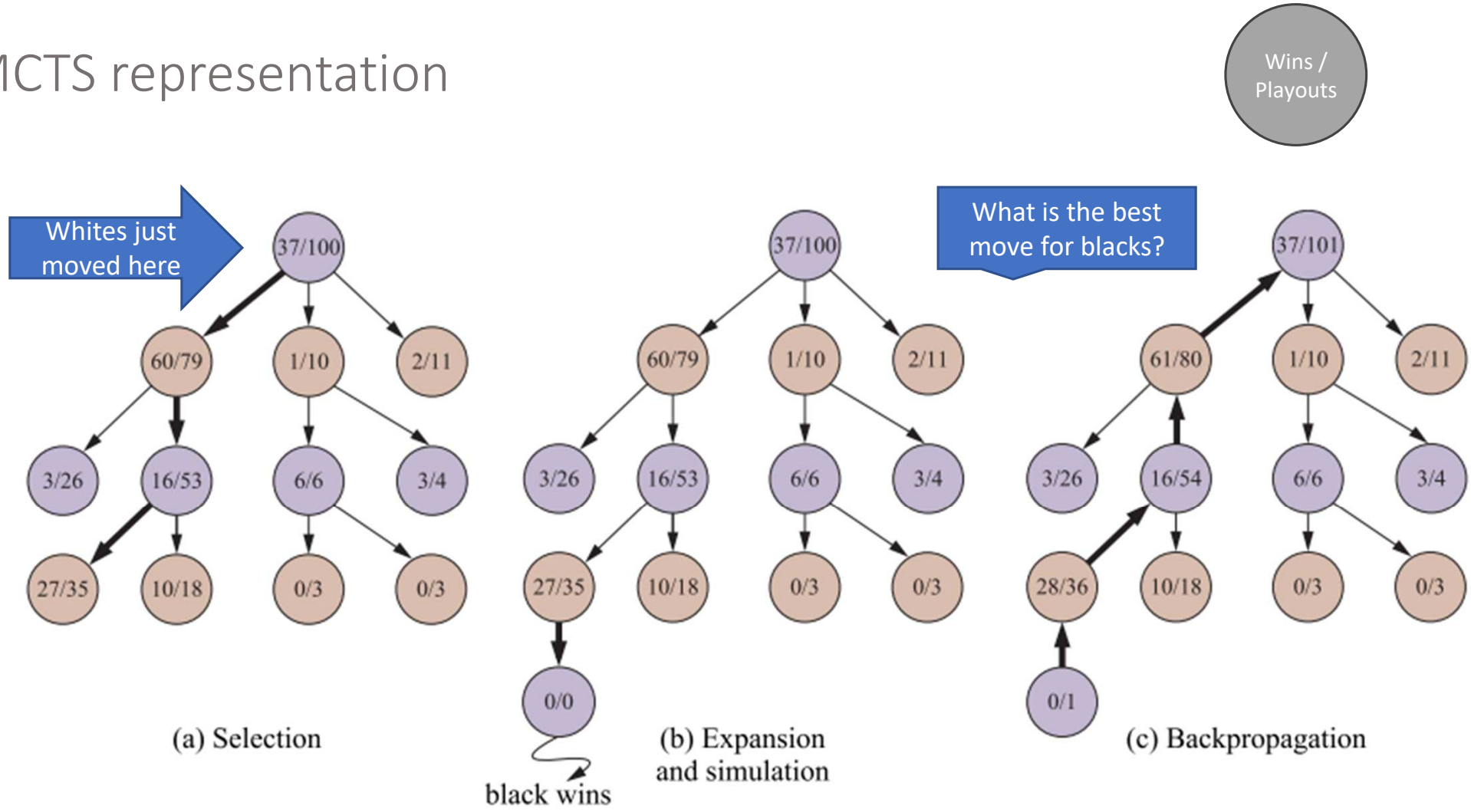
MCTS representation (2)

- In the next Figure, the root represents the state the **white** has just moved (has won 37 out of the 100 playouts done)
  - The thick arrow shows the selection of a move by **black** they has won 60/79 playouts.
  - This is the best win percentage among the three moves, so selecting it is an example of exploitation.
  - But it would also have been reasonable to select the 2/11 node for the sake of exploration
  - Selection continues on to the leaf node marked 27/35.

- Figure (b) shows the new node marked with 0/0. The simulation results in a win for **black**.

- Figure (c) – Since black won the playout, black nodes are incremented in both the number of wins/playouts

MCTS representation



MCTS versus Alpha–Beta Tree Search

- **Monte Carlo** search has an advantage over alpha–beta for games like Go
  - The branching factor is very high (and thus alpha–beta can't search deep enough),
  - It is difficult to define a good evaluation function.
- **Alpha–beta** chooses the path to a node that has the highest achievable evaluation function score
  - If the evaluation function is inaccurate, alpha–beta will be inaccurate.
  - A miscalculation on a single node can lead alpha–beta to erroneously choose/avoid a path to that node

- **Monte Carlo** search relies on the aggregate of many playouts, and it is not as vulnerable to a single error.
  - It is possible to combine MCTS and evaluation functions, doing a playout for a certain number of moves and then truncating the playout and apply an evaluation function.
- It is also possible to **combine aspects of MCTS and alpha–beta**
  - E.g., in games that can last many moves, we may want to use early playout termination, in which we stop a playout that is taking too many moves, and either evaluate it with a heuristic evaluation function or just declare it a draw.

Thank you!