Artificial Intelligence

Adversarial Search and Games

Dr. Bilgin Avenoğlu

Adversarial Search and Games

- Competitive environments:
  - Two or more agents have conflicting goals, giving rise to adversarial search problems - adversarial search

Two-player zero-sum games

- The games most commonly studied within AI (such as chess and Go) are
  - deterministic, two-player, turn-taking, perfect information, zero-sum games.
- "Perfect information" is a synonym for "fully observable"
- "Zero-sum" means that what is good for one player is just as bad for the other
- There is no "win-win" outcome.
- For games we often use the term move as a synonym for "action" and position as a synonym for "state".

Two-player zero-sum games

- S_0: 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*.

Game Tree

- Tic-tac-toe, the game tree is relatively small—fewer than 9! = 362, 880 terminal nodes (with only 5,478 distinct states).
- But for chess there are over 10^{40} nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world.

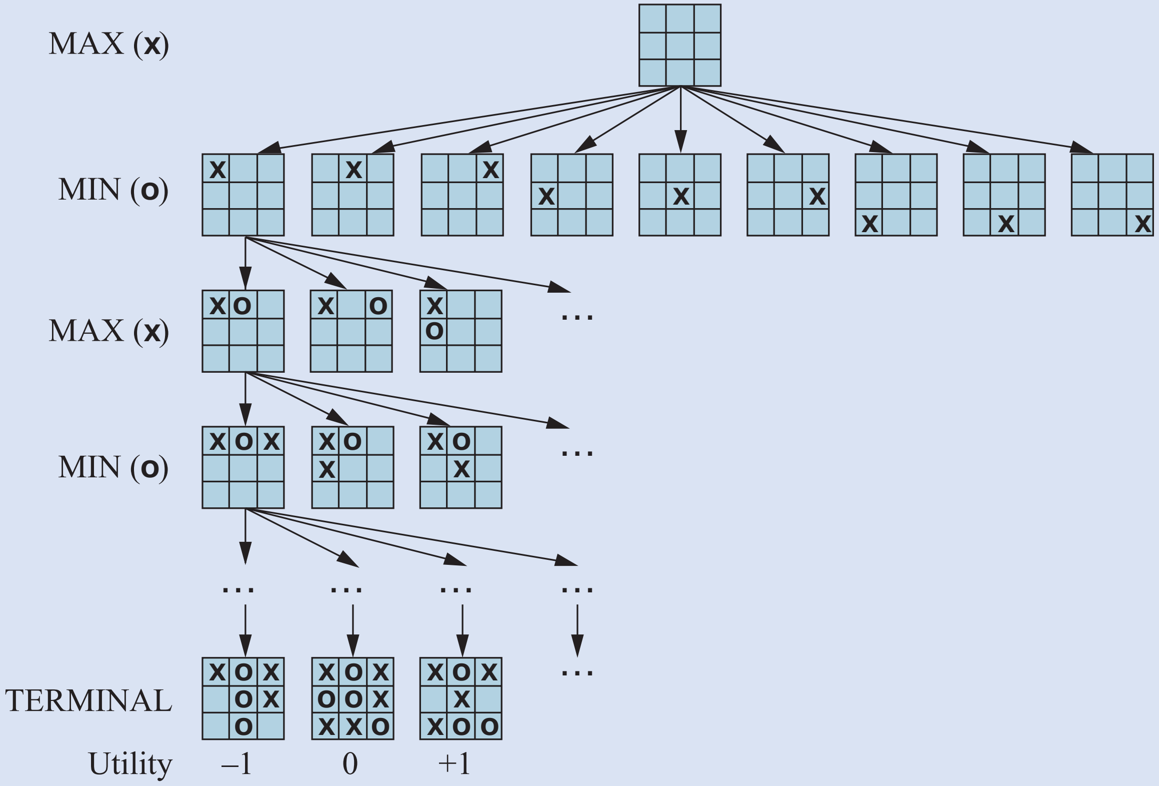


Figure 6.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Optimal Decisions in Games

- The optimal strategy can be determined by working out the minimax value of each state in the tree,
  - *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.
- The minimax value of a terminal state is just its utility.
- In a non-terminal state, MAX prefers to move to a state of maximum value when it is MAX's turn to move,
  - and MIN prefers a state of minimum value (that is, minimum value for MAX and thus maximum value for MIN).

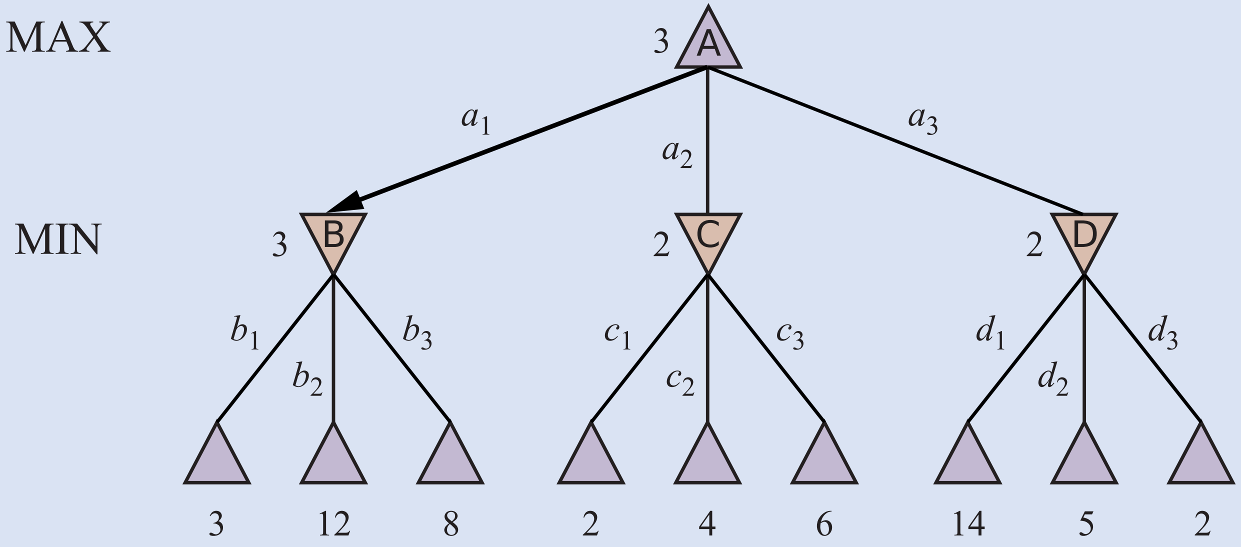
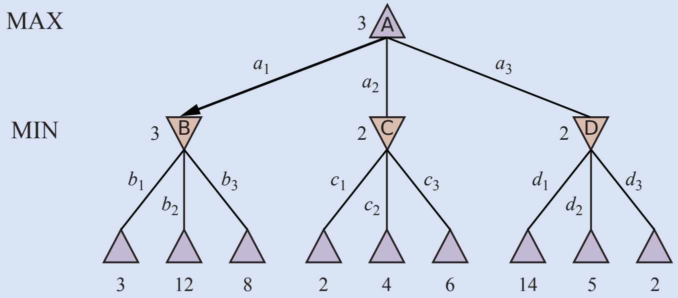


Figure 6.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state 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 Search Algorithm

- The minimax algorithm performs a complete depth-first algorithm.
- If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$.
- The space complexity is $O(bm)$ for an algorithm that generates all actions at once
- The exponential complexity makes MINIMAX impractical for complex games;
  - for example, chess has a branching factor of about 35 and the average game has depth of about 80 ply, and it is not feasible to search $35^{80} \approx 10^{123}$ states.
  - MINIMAX serves as a basis for the mathematical analysis of games.

MINIMAX Algorithm



https://www.youtube.com/watch?v=zDskcx8FStA

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* ← $-\infty$
  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* ← $+\infty$
  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*

Figure 6.3 An algorithm for calculating the optimal move using minimax—the move that leads to a terminal state with maximum 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 and the move to get there.

Optimal decisions in multiplayer games

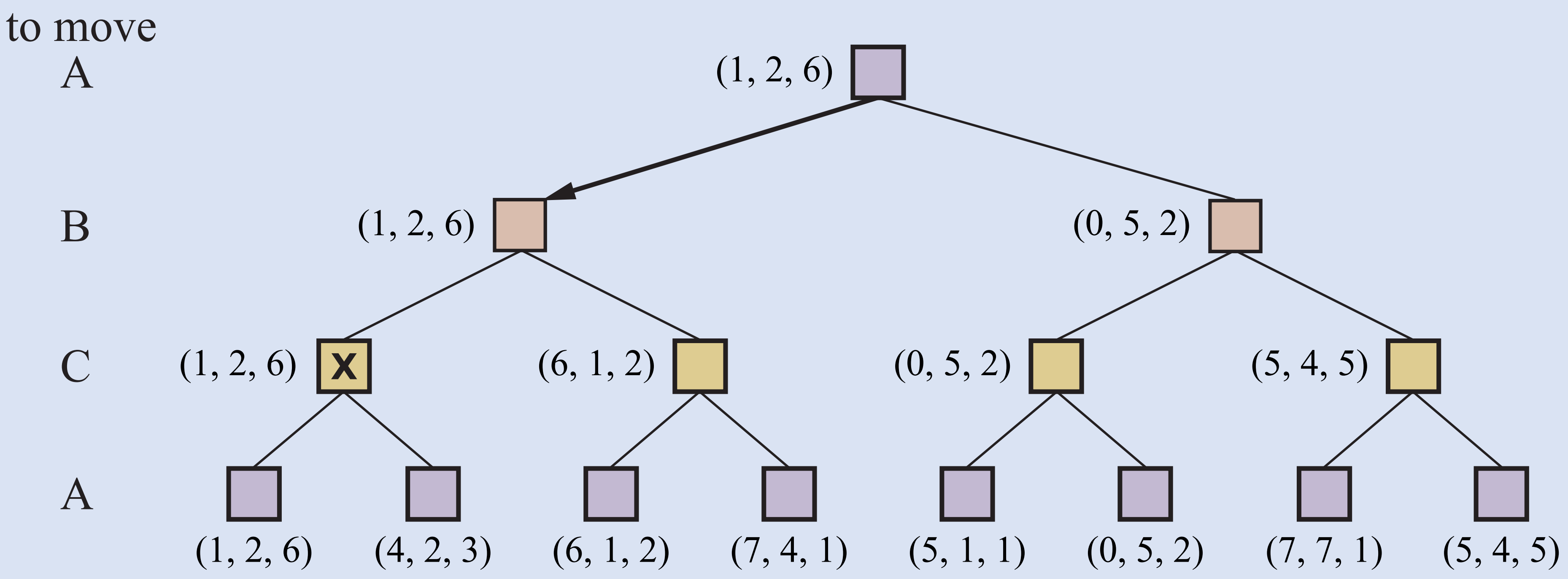


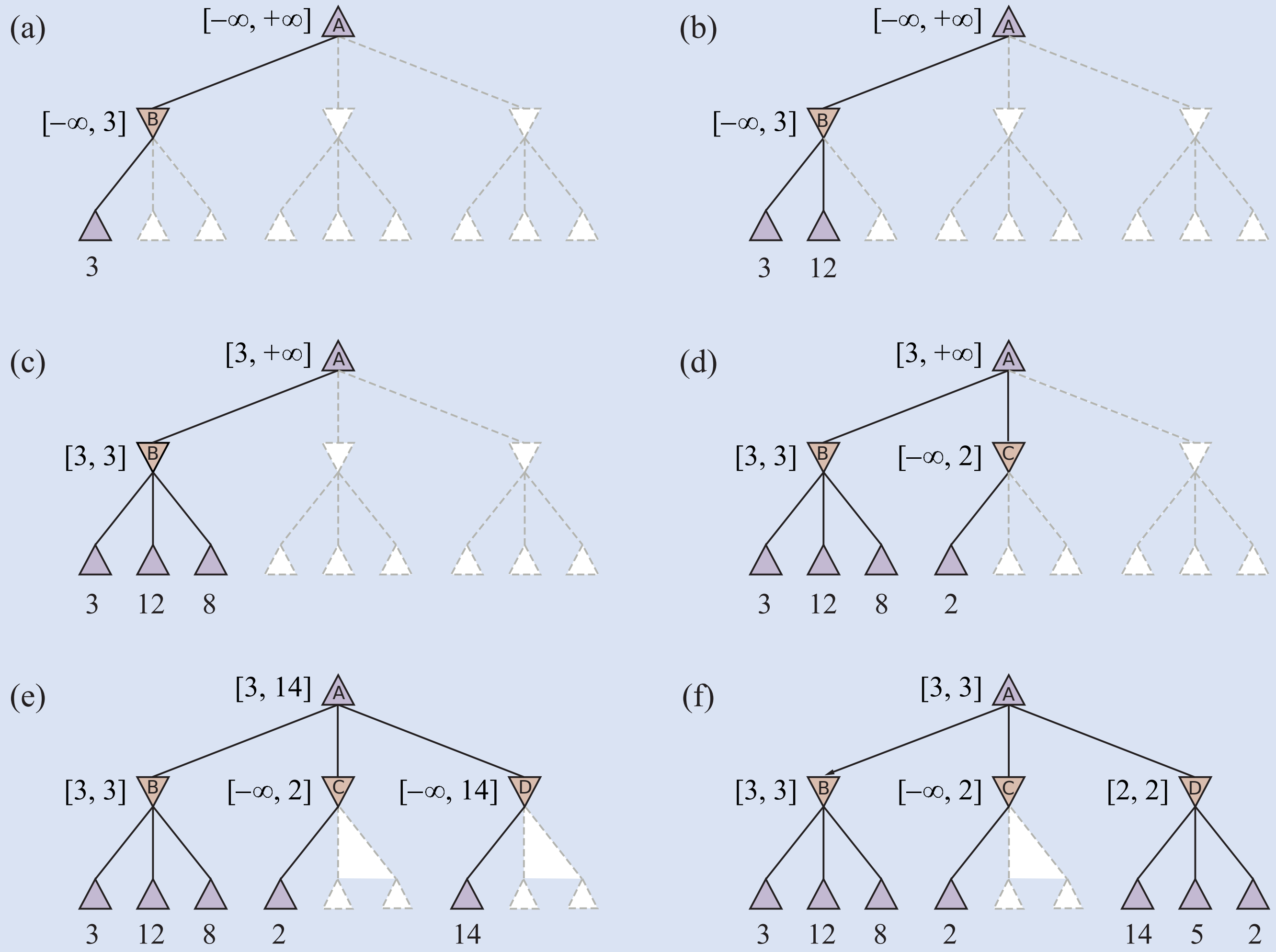
Figure 6.4 The first three ply of a game tree with three players (*A*, *B*, *C*). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Alpha-Beta Pruning

- The number of game states is exponential in the depth of the tree.
- No algorithm can completely eliminate the exponent
- We can sometimes 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.

Alpha-Beta Pruning

- (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. 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.



Alpha-Beta Pruning

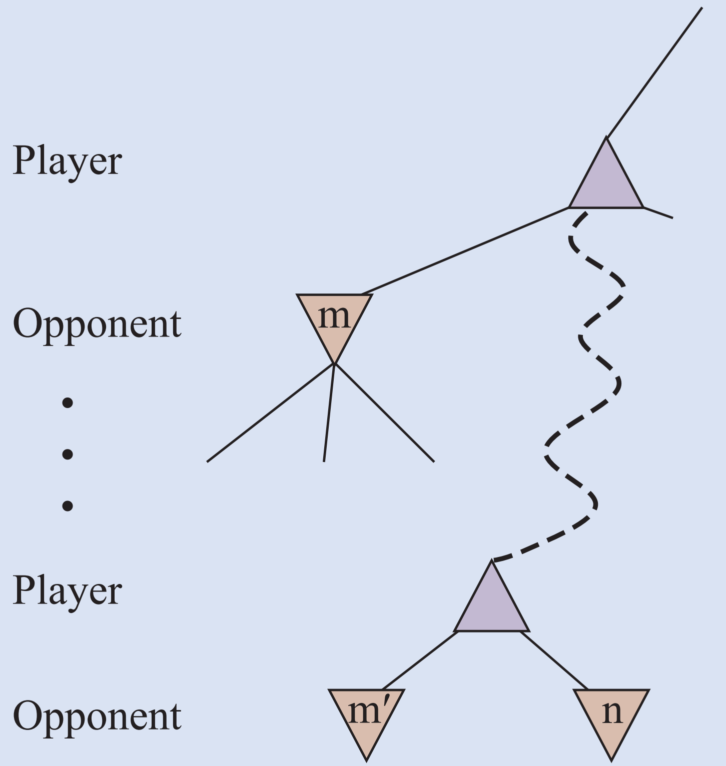


Figure 6.6 The general case for alpha–beta pruning. If m or m' is better than n for Player, we will never get to n in play.

Alpha-Beta alg.

- α = highest-value for MAX.
  - Think: α = "at least."
- β = lowest-value for MIN.
  - Think: β = "at most."

https://pascscha.ch/info2/abTreePractice/

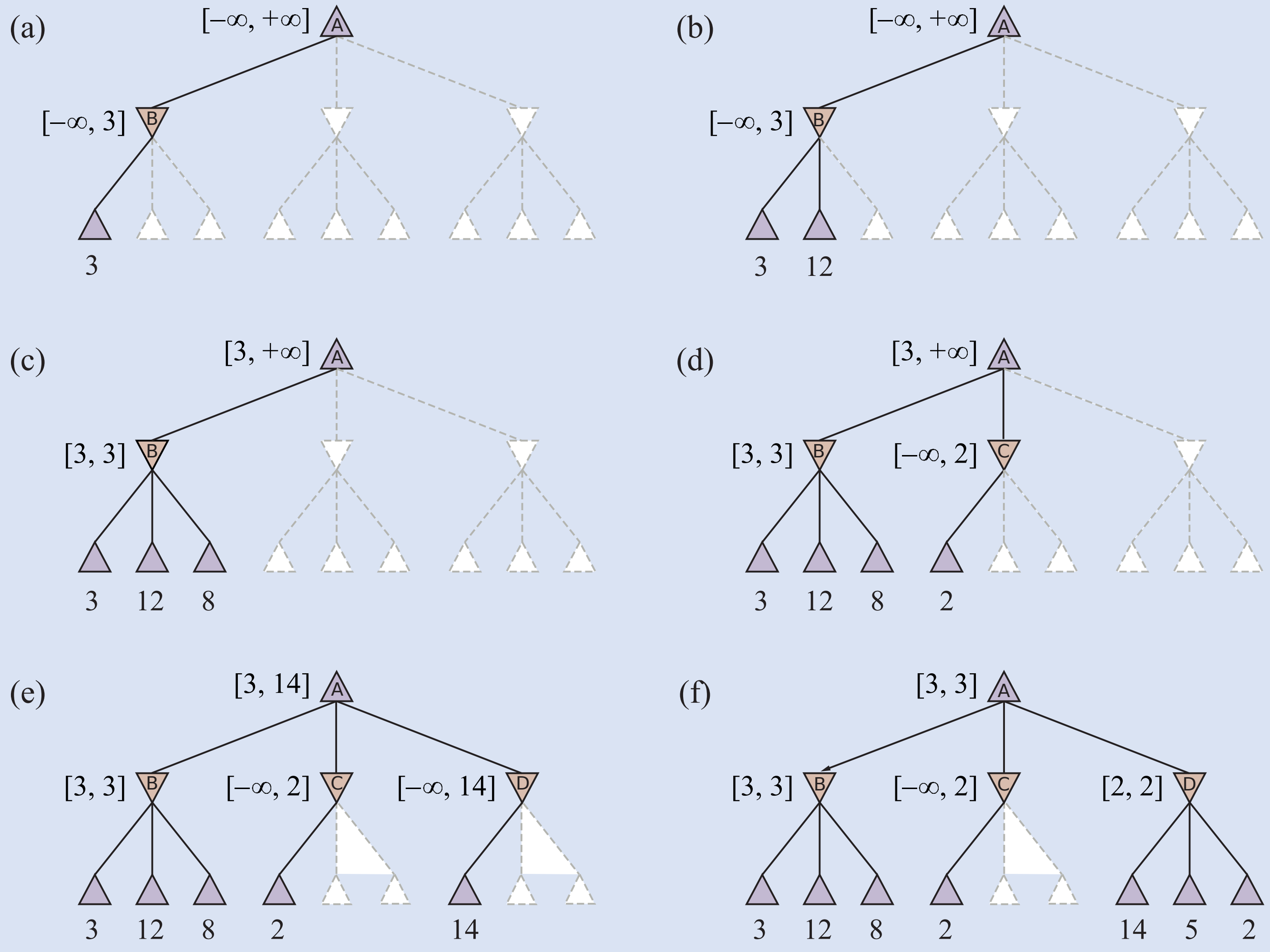
function ALPHA-BETA-SEARCH(*game*, *state*) **returns** an action
  player $\leftarrow$ *game*.TO-MOVE(*state*)
  value, *move* $\leftarrow$ MAX-VALUE(*game*, *state*, $-\infty, +\infty$)
  return *move*

function MAX-VALUE(*game*, *state*, α, β) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  $v \leftarrow -\infty$
  for each *a* **in** *game*.ACTIONS(*state*) **do**
    v2, *a2* $\leftarrow$ MIN-VALUE(*game*, *game*.RESULT(*state*, *a*), α, β)
    if *v2* $>$ *v* **then**
      v, *move* $\leftarrow$ *v2*, *a*
      $\alpha \leftarrow$ MAX(α, *v*)
    if $v \geq \beta$ **then return** *v*, *move*
  return *v*, *move*

function MIN-VALUE(*game*, *state*, α, β) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  $v \leftarrow +\infty$
  for each *a* **in** *game*.ACTIONS(*state*) **do**
    v2, *a2* $\leftarrow$ MAX-VALUE(*game*, *game*.RESULT(*state*, *a*), α, β)
    if *v2* $<$ *v* **then**
      v, *move* $\leftarrow$ *v2*, *a*
      $\beta \leftarrow$ MIN(β, *v*)
    if $v \leq \alpha$ **then return** *v*, *move*
  return *v*, *move*

Move Ordering

- The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined.
- For example, in Figure, (e) and (f), we could not prune any successors of D at all because the worst successors (from the point of view of MIN) were generated first.
- If the third successor of D had been generated first, with value 2, we would have been able to prune the other two successors.



Move Ordering

- If move ordering could be done perfectly,
  - Alpha-beta would need to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$ for minimax.
  - This means that the effective branching factor becomes $\sqrt{b}$ instead of b
    - for chess, about 6 instead of 35.
  - Put another way, alpha-beta with perfect move ordering can solve a tree roughly twice as deep as minimax in the same amount of time.
  - With random move ordering, the total number of nodes examined will be roughly $O(b^{3m/4})$ for moderate b.

Transposition table

- Idea: Cache and reuse information about previous search by using hash table
- Avoid searching the same subtree twice
- Get best move information from earlier, shallower searches
- In game tree search, repeated states can occur because of transpositions
- *$[w_1,b_1,w_2,b_2]$ -> resulting state s*
  - After exploring a large subtree below s, we find its backed-up value, which we store in the transposition table.
  - When we later search the sequence of moves *$[w_2,b_2,w_1,b_1]$*, we end up in s again, and we can look up the value instead of repeating the search.
- In chess, use of transposition tables is very effective, allowing us to double the reachable search depth in the same amount of time.

Claude Shannon’s Strategy

- Even with alpha–beta pruning and clever move ordering, minimax won’t work for games like chess and Go, still too many states to explore in the time.
- *Type A* strategy considers all possible moves to a certain depth in the search tree, and then uses a heuristic evaluation function to estimate the utility of states at that depth.
  - It explores a wide but shallow portion of the tree.
- *Type B* strategy ignores moves that look bad, and follows promising lines “as far as possible.”
  - It explores a deep but narrow portion of the tree.
- Chess programs are often Type A
- Go programs are often Type B (due to the high branching factor)
- Type B programs have shown world-champion-level play across a variety of games, including chess

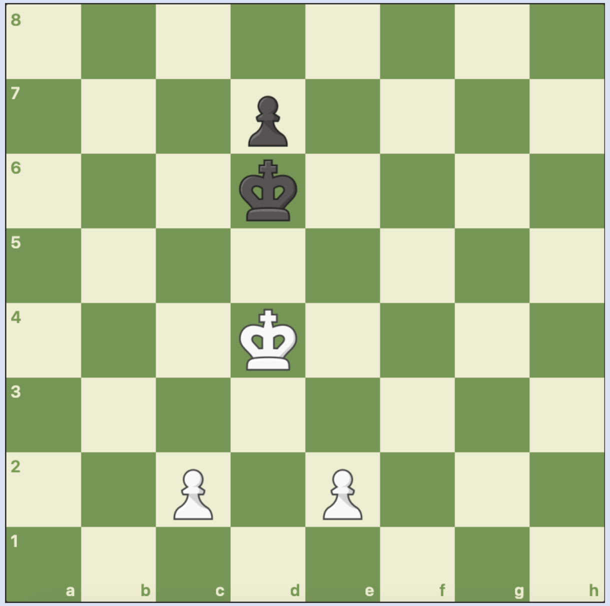
Heuristic Alpha–Beta Tree Search

- Cut off the search early and apply a heuristic evaluation function to states, effectively treating nonterminal nodes as if they were terminal.
- What makes for a good evaluation function?
  - the computation must not take too long!
  - the evaluation function should be strongly correlated with the actual chances of winning
- chances of winning? Chance!

Heuristic Alpha–Beta Tree Search

- Define categories or equivalence classes of states based on experience:
  - 82% of the states in the two-pawns versus one-pawn category lead to a win (utility +1);
  - 2% to a loss (0),
  - 16% to a draw (1/2).
- A reasonable evaluation for states in the category is the expected value:

$(0.82 \times +1) + (0.02 \times 0) + (0.16 \times 1/2) = 0.90$.



- This kind of analysis requires too many categories and hence too much experience to estimate all the probabilities

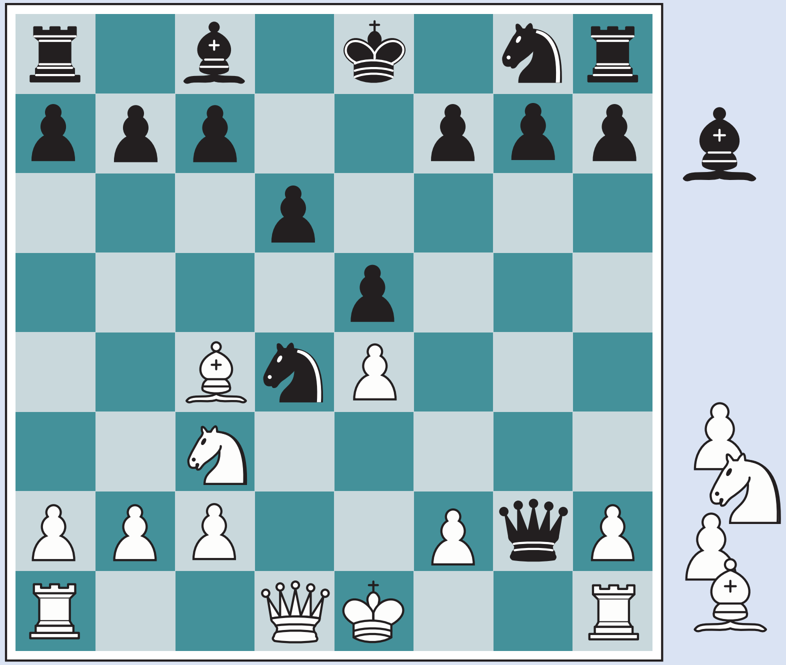
Heuristic Alpha–Beta Tree Search

- Approximate material value
  - each pawn is worth 1,
  - a knight or bishop is worth 3,
  - a rook 5,
  - 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: weighted linear function

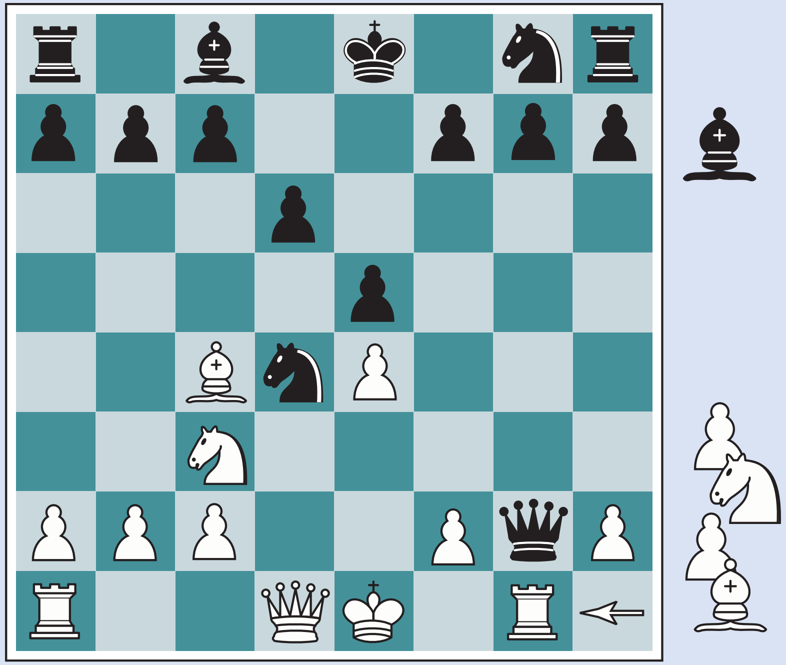
$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

- Each f_i is a feature of the position (such as “number of white bishops”) and each w_i is a weight (saying how important that feature is).

Heuristic Alpha–Beta Tree Search



(a) White to move



(b) White to move

Figure 6.8 Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

Heuristic Alpha–Beta Tree Search

- Adding up the values based on a strong assumption:
  - The contribution of each feature is independent of the values of the other features.
- For this reason, current programs for chess and other games also use nonlinear combinations of features.
  - a pair of bishops might be worth more than twice the value of a single bishop
  - a bishop is worth more in the endgame than earlier - when the move number feature is high or the number of remaining pieces feature is low.
- Where do the features and weights come from?
  - They’re not part of the rules of chess, but they are part of the culture of human chess-playing experience.

Cutting off search

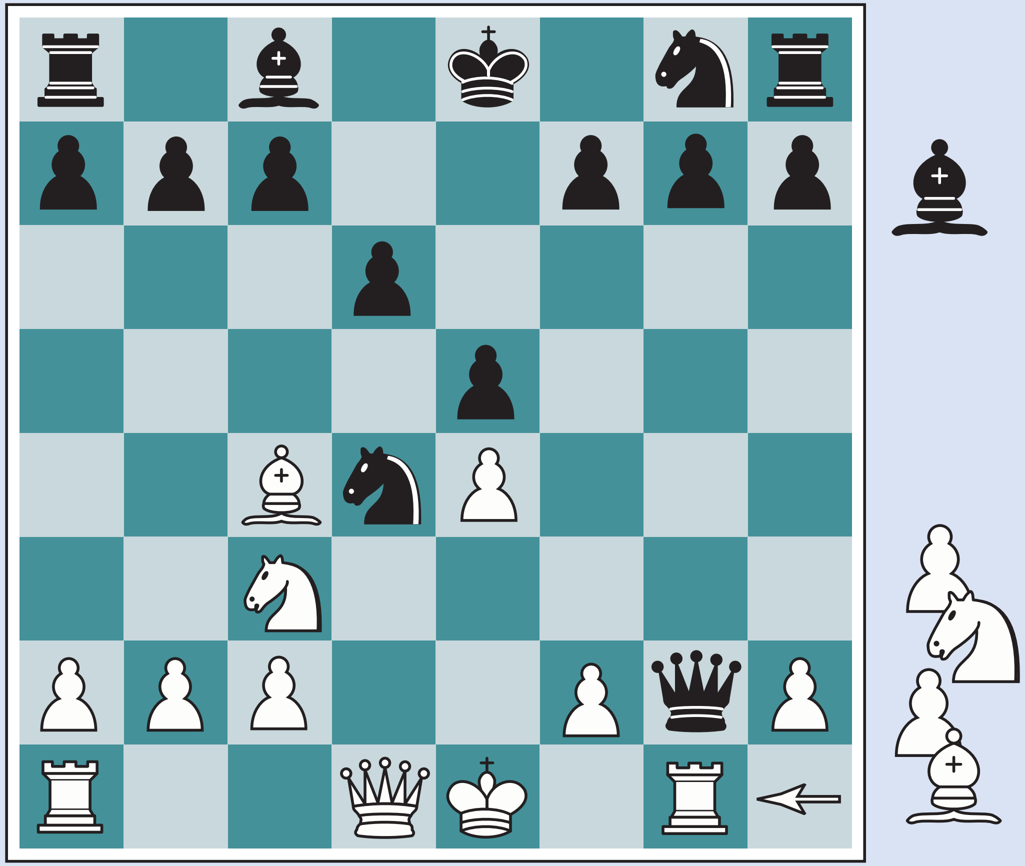
- Replace the two lines in alpha-beta algorithm that mention IS-TERMINAL with the following line:

$$\textbf{if } game.\textsc{Is-Cutoff}(state, depth) \textbf{ then return } game.\textsc{Eval}(state, player), null$$

- Set a fixed depth limit so that IS-CUTOFF(state, depth) returns true for all depth greater than some fixed depth *d*
- 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.
  - In iterative deepening, keep entries in the transposition table, subsequent rounds will be faster, and we can use the evaluations to improve move ordering.

Depth limit problem

- The program searches to the depth limit, reaching the position in Figure, 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 with no compensation.
- Hence, the position is actually favorable for White, but this can be seen only by looking ahead.



(b) White to move

The solution for depth limit problem

- The evaluation function should be applied only to positions that are quiescent
  - that is, positions in which there is no pending move (such as a capturing the queen) that would wildly swing the evaluation.
  - For nonquiescent positions the *IS-CUTOFF* returns false, and the search continues until quiescent positions are reached.
  - This extra quiescence search is sometimes restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.

The Horizon Effect

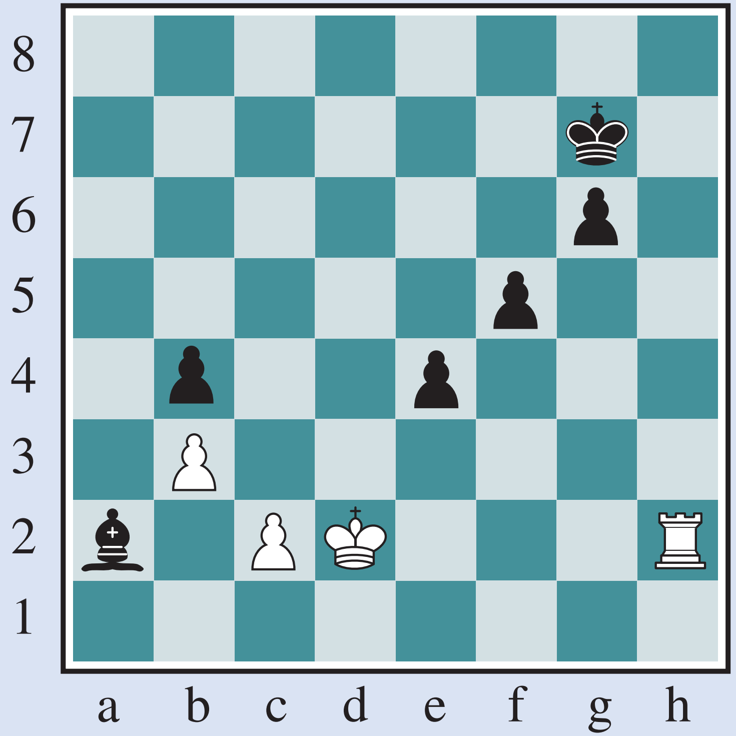


Figure 6.9 The horizon effect. With Black to move, the black bishop is surely doomed. But Black can forestall that event by checking the white king with its pawns, encouraging the king to capture the pawns. This pushes the inevitable loss of the bishop over the horizon, and thus the pawn sacrifices are seen by the search algorithm as good moves rather than bad ones.

Forward pruning

- Alpha-beta pruning prunes branches of the tree that can have no effect on the final evaluation
- Forward pruning prunes moves that appear to be poor moves, but might possibly be good ones.
- Thus, the strategy saves computation time at the risk of making an error.
- In Shannon’s terms, this is a Type B strategy.
- Clearly, most human chess players do this, considering only a few moves from each position (at least consciously).

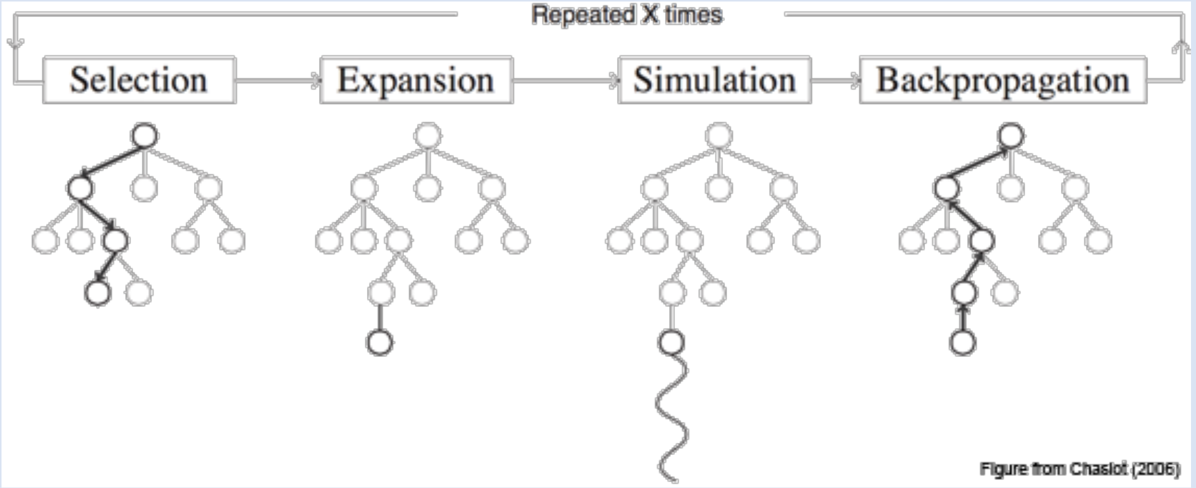
PROBCUT - Probabilistic Cut

- 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 prunes any node that is *provably* outside the current (α,β) window.
- PROBCUT also prunes nodes that are *probably* outside the window.

Monte Carlo Tree Search

- MCTS does not use a heuristic evaluation function.
- The value of a state is estimated as the average utility over a number of simulations of complete games starting from the state.
- A simulation / playout 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 (not fallible heuristics) 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."

MCTS



1. Selection
    - Starting at root node R, recursively sele optimal child nodes until a leaf node L is reached.
2. Expansion
    - If L is a not a terminal node then create one or more child nodes and select one C.
3. Simulation
    - Run a simulated playout from C until a result is achieved.
4. Backpropagation
    - Update the current move sequence with the simulation result.

- Each node must contain two information:
    - an estimated value based on simulation results
    - the number of times it has been visited.

MCTS

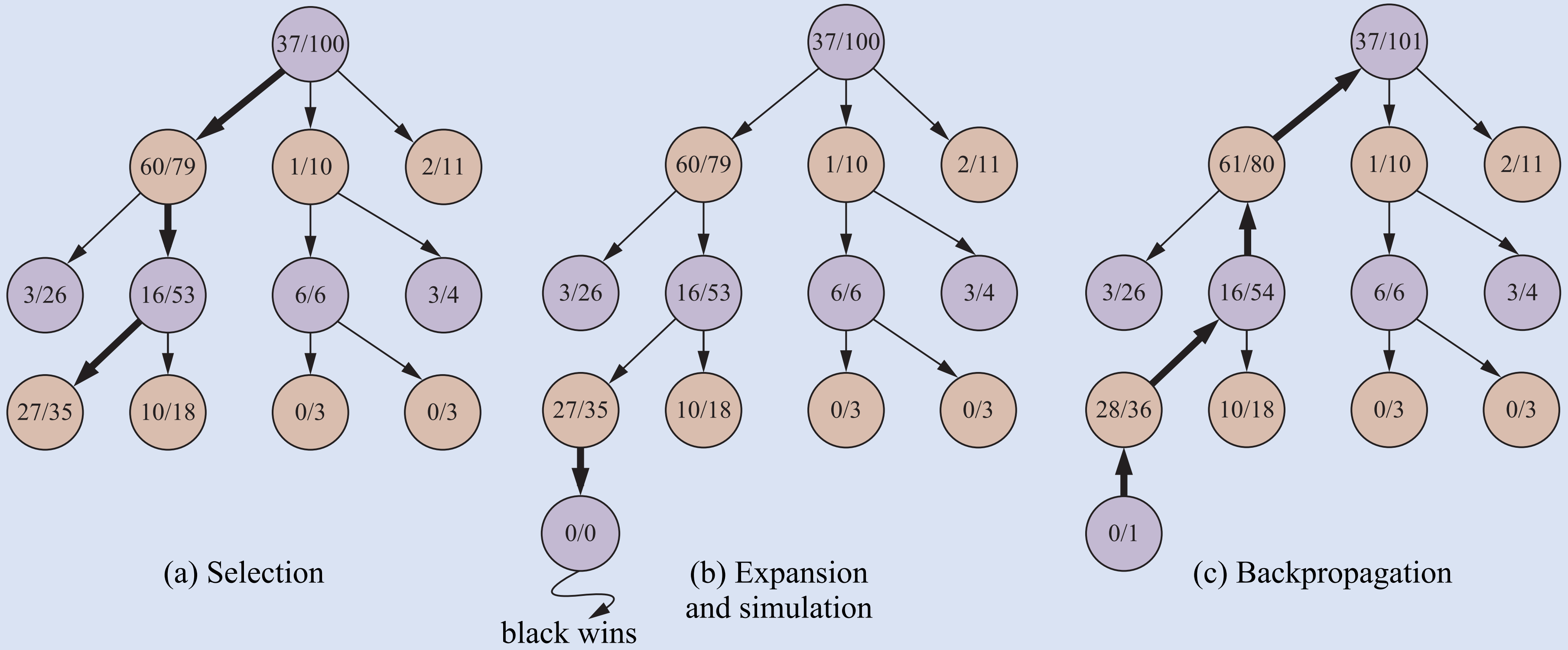


Figure 6.10 One iteration of the process of choosing a move with Monte Carlo tree search (MCTS) using the upper confidence bounds applied to trees (UCT) selection metric, shown after 100 iterations have already been done. In (a) we select moves, all the way down the tree, ending at the leaf node marked 27/35 (for 27 wins for black out of 35 playouts). In (b) we expand the selected node and do a simulation (playout), which ends in a win for black. In (c), the results of the simulation are back-propagated up the tree.

MCTS

- Consider a game with a branching factor of 32, where the average game lasts 100 ply.
- If we have enough computing power to consider a billion game states before we have to make a move, then minimax can search 6 ply deep, alpha–beta with perfect move ordering can search 12 ply, and Monte Carlo search can do 10 million playouts.

Stochastic Games

- Backgammon is a typical stochastic game that combines luck and skill.
- In Figure, Black has rolled a 6–5 and has four possible moves (each of which moves one piece forward (clockwise) 5 positions, and one piece forward 6 positions).

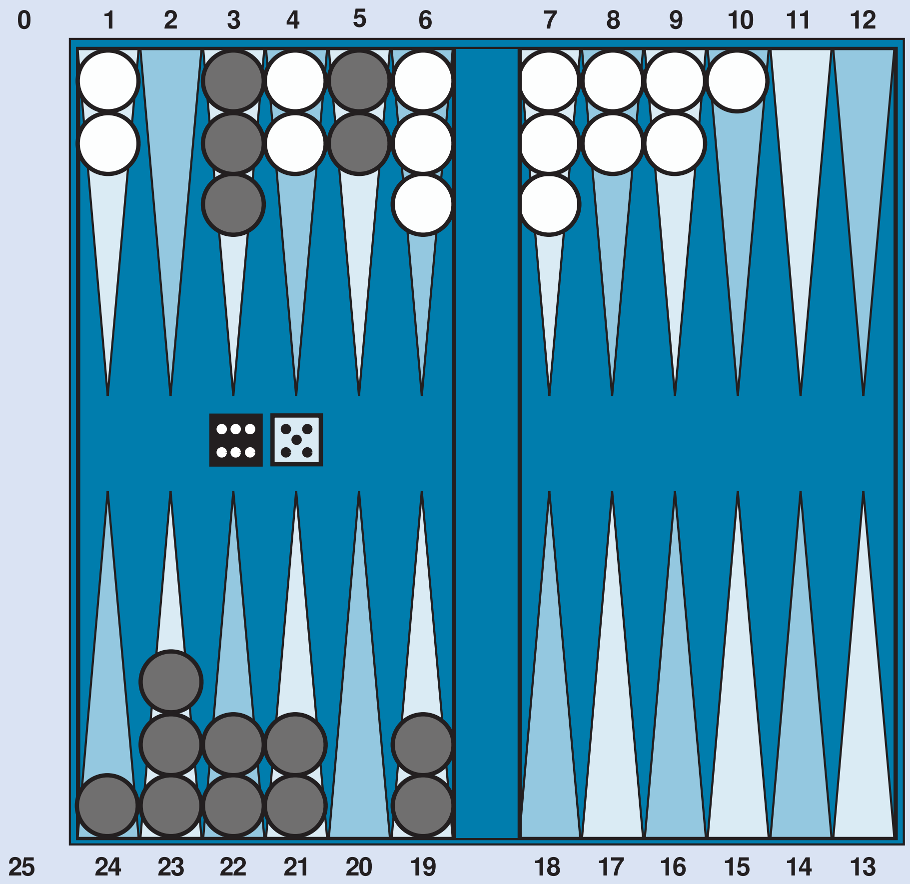


Figure 6.12 A typical backgammon position. The goal of the game is to move all one's pieces off the board. Black moves clockwise toward 25, and White moves counterclockwise toward 0. A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over. In the position shown, Black has rolled 6–5 and must choose among four legal moves: (5–11,5–10), (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.

Backgammon

- *Black* knows what moves can be made, but does not know what *White* is going to roll and thus does not know what White's legal moves will be.
- A game tree in backgammon must include chance nodes in addition to *MAX* and *MIN* nodes.
- 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.
- 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.
- The six doubles (1–1 through 6–6) each have a probability of 1/36, so we say P(1–1) = 1/36. The other 15 distinct rolls each have a 1/18 probability.

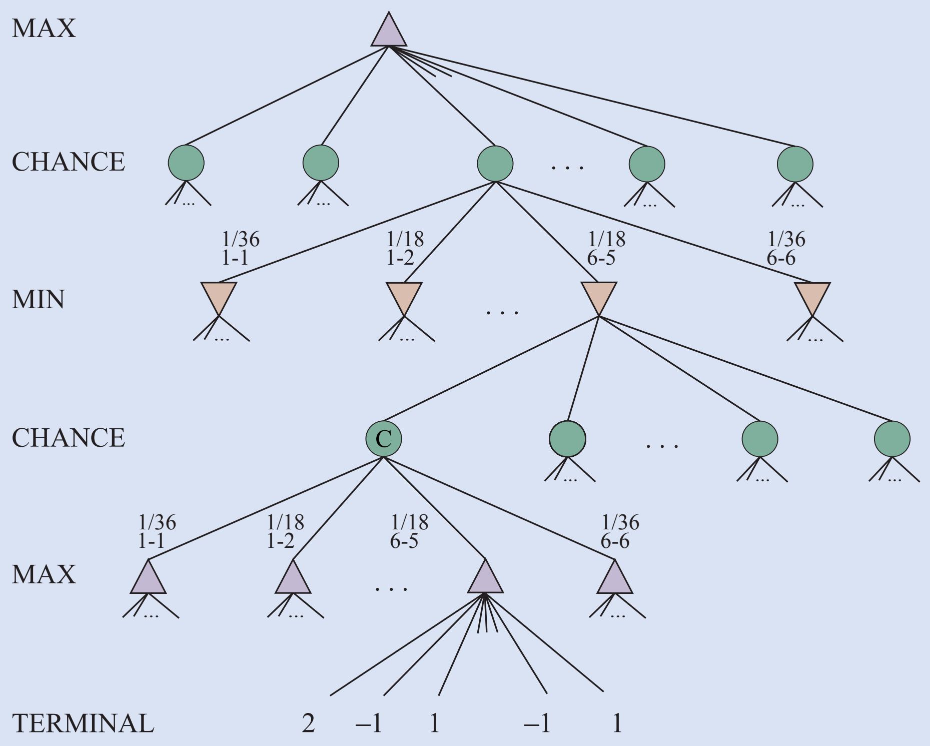


Figure 6.13 Schematic game tree for a backgammon position.

The End!