GEORGIA SOUTHERN UNIVERSITY

Georgia's large-scale, small-feel research university

Artificial Intelligence

CSCI-5430

Fall 2016/ Dr. Kai Wang

Outline

- Games
- Optimal decisions
- Minimax algorithm
- α-β pruning
- Imperfect, real-time decisions

Games

- Multi agent environments: any given agent will need to consider the actions of other agents and how they affect its own welfare.
- The unpredictability of these other agents can introduce many possible contingencies
- There could be competitive or cooperative environments
- Competitive environments, in which the agents' goals are in conflict, require adversarial search – these problems are often called games

Games

- In game theory (economics), any multi-agent environment (either cooperative or competitive) is a game provided that the impact of each agent on the others is significant
- In AI, the most common games are of a specialized kind - deterministic, turn taking, two-player, zero sum games of perfect information
- In our terminology – deterministic, fully observable environments with two agents whose actions alternate and the utility values at the end of the game are always equal and opposite (+1 and –1)

Games – history of chess playing

- 1949 – Shannon paper – originated the ideas
- 1951 – Turing paper – hand simulation
- 1958 – Bernstein program
- 1955~1960 – Simon-Newell program
- 1961 – Soviet program
- 1966~1967 – MacHack 6 – defeated a good player
- 1970s – NW chess 4.5
- 1980s – Cray Bitz
- 1990s – Belle, Hitech, Deep Thought
- 1997 – Deep Blue - defeated Garry Kasparov
- 2016 – AlphaGo - defeated Lee Sedol

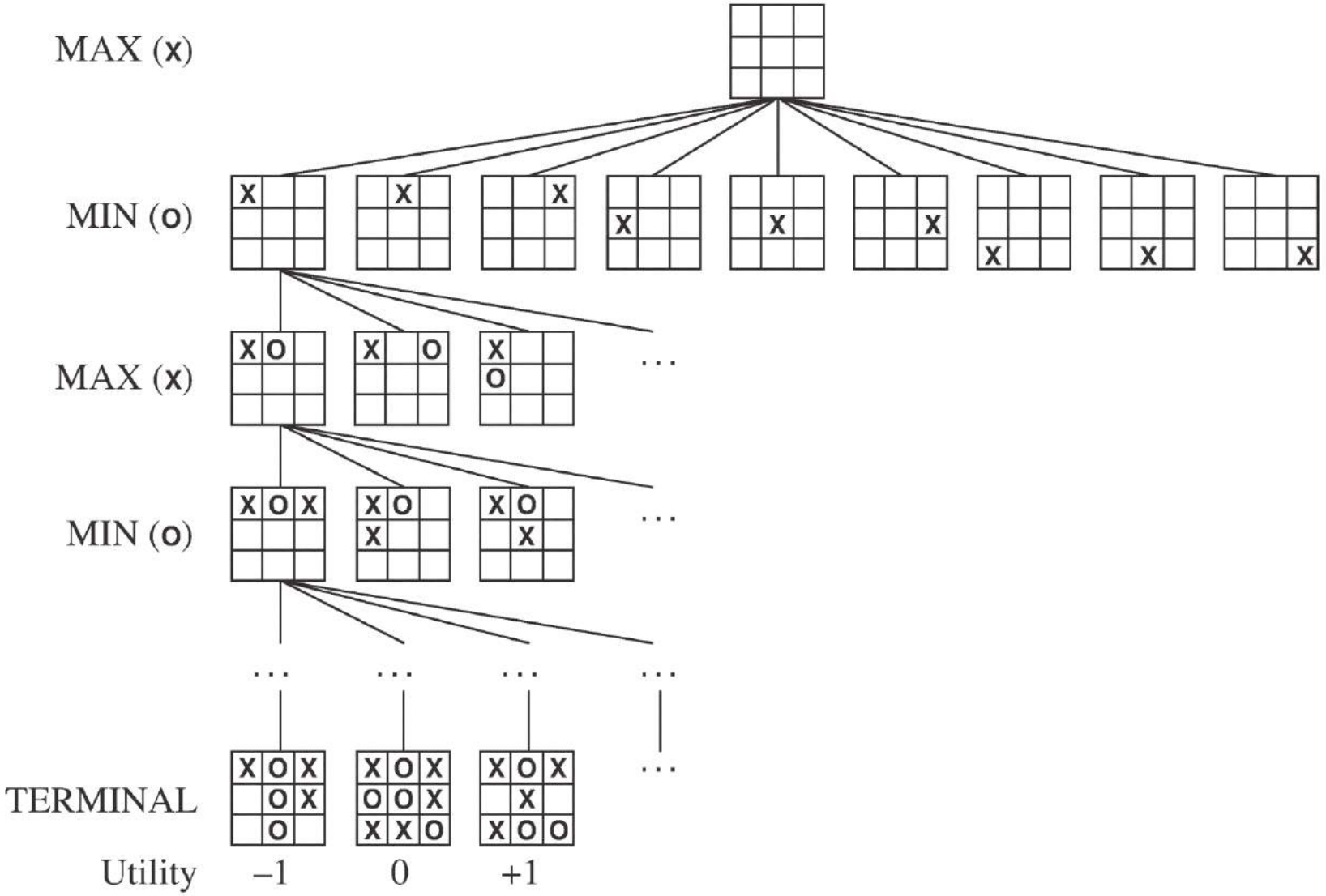
Games formulated as search problems

- Initial state: specifies how the game is initially set up
- 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** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p.

Game Tree Search

- Game tree: defined by the initial state, ACTIONS and RESULT functions. It is a tree where the nodes are game states and the edges are moves. It incorporates all possible games.
- Games with two players MAX and MIN. MAX moves first from the initial state.
- We are not looking for a path, only the next move to make. Our best move depends on what the other player does.
- Entire tree is too large to be stored in memory.

Partial Game Tree for Tic-Tac-Toe

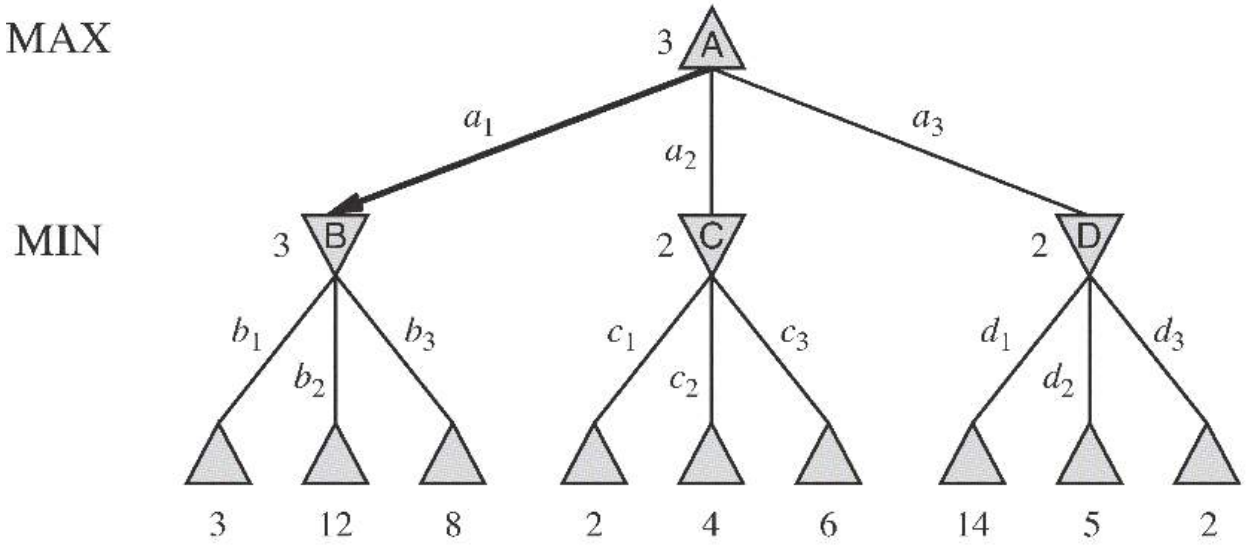


Optimal strategies

- In a normal search problem, the optimal solution would be a sequence of moves leading to a goal
- In a two-player game, MIN has something to say about it and therefore MAX must find a contingent 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
  –…
- An optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent. It maximizes the *worst-case* outcome for MAX.
- If MIN does not play optimally, MAX's optimal strategy will do ever better.

Minimax: brute-force optimal strategy

- Perfect play for deterministic games
- Idea: choose moving to position with highest minimax value = best achievable payoff against best opponent play
- E.g., 1 move (2-ply) game tree (Fig 5.2):



Minimax value

- Given a game tree, the optimal strategy can be determined by examining the minimax value of each node (MINIMAX-VALUE(n))
- The minimax value of a node is the utility of being in the corresponding state, assuming that both players play optimally from there to the end of the game
- Given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value

Minimax formula

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

- For the game in Fig 5.2, MINIMAX-VALUE(root) = max(min(3,12,8), min(2,4,6), min(14,5,2)) = max(3,2,2) = 3

Minimax algorithm (similar to AND-OR search?)

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

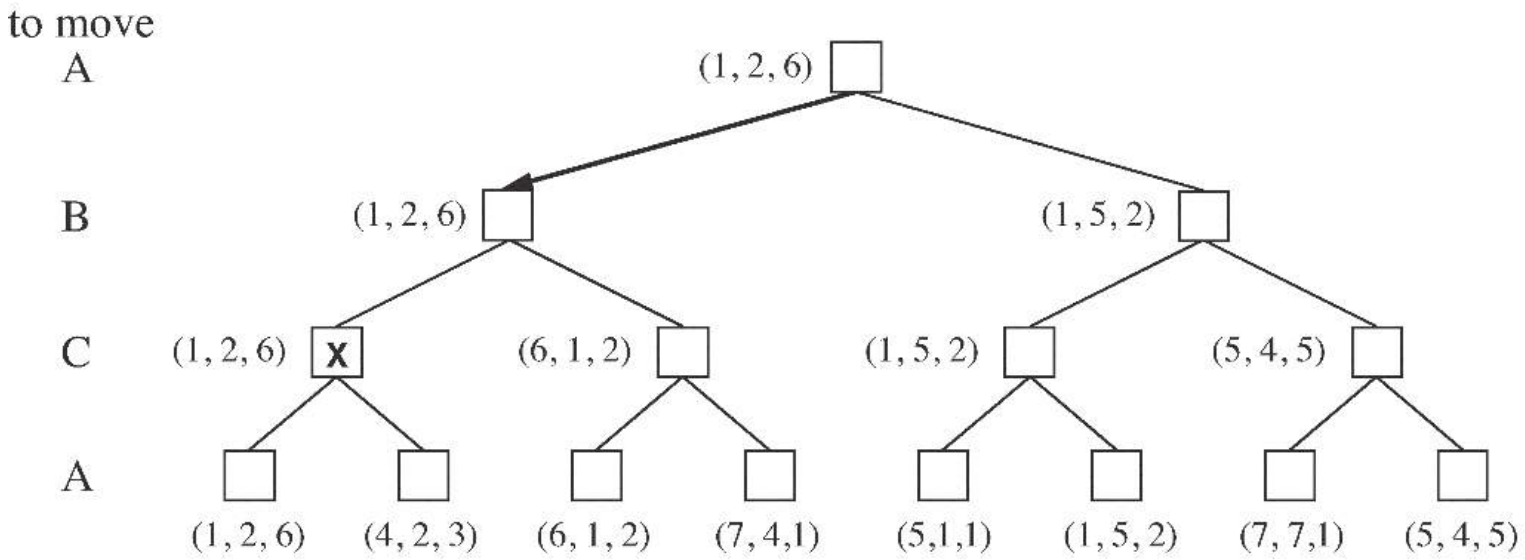
```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v
```

```
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
```

Properties of minimax

- Complete? Yes (if tree is finite)
- Optimal? Yes (against an optimal opponent)
- Time complexity? $O(b^m)$
- Space complexity? O(bm) (depth-first exploration)
- For chess, b ≈ 35, m ≈ 100 for "reasonable" games
  - exact solution completely infeasible

Multiplayer Games

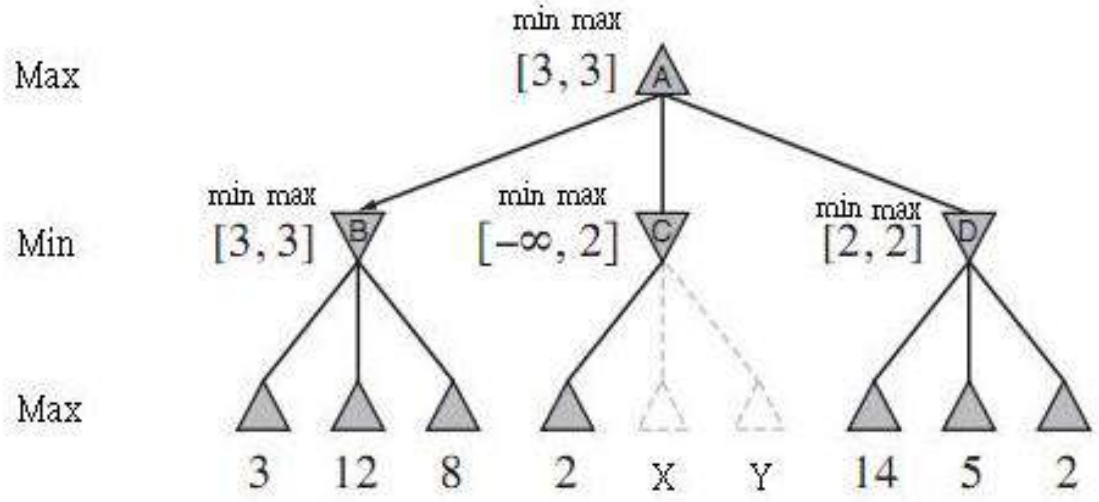


Multiplayer Games

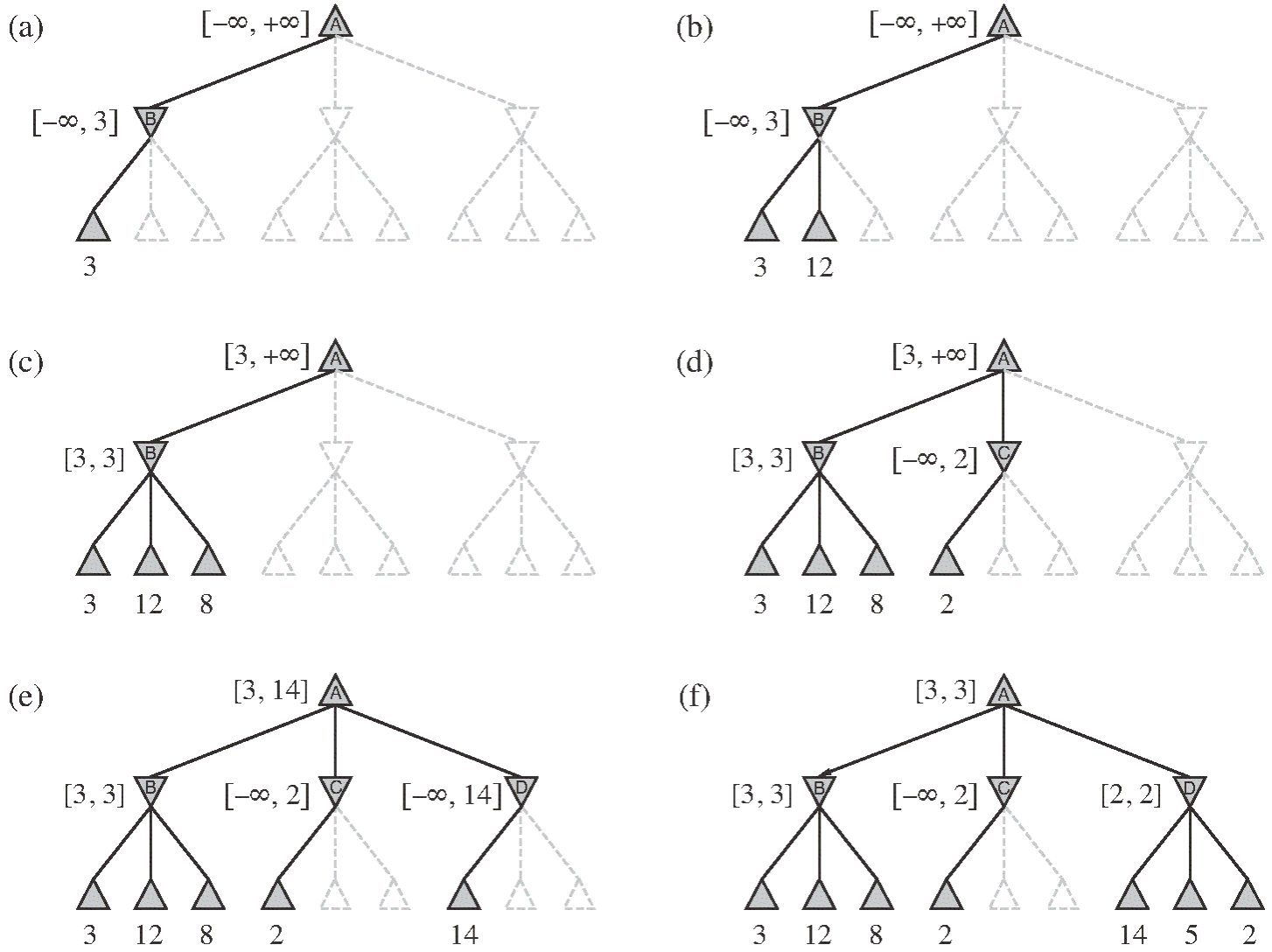
- Use a vector of values as the utilities for each state from each player's perspective.
- The utility value of a node n is always the utility vector of the successor state with the highest value for the player choosing at n.
- Alliance might be a natural consequence of optimal strategies for each player in a multiplayer game. It might emerge or disappear depending on game state.

α-β pruning

- It is possible to compute the correct minimax decision without looking at every node in the game tree
- MINIMAX-VALUE(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) where z≤2
  = 3



α-β pruning


(a)
[−∞, +∞]
A
[−∞, 3]
B
3
(b)
[−∞, +∞]
A
[−∞, 3]
B
3
12
(c)
[3, +∞]
A
[3, 3]
B
3
12
8
(d)
[3, +∞]
A
[3, 3]
B
[−∞, 2]
C
3
12
8
2
(e)
[3, 14]
A
[3, 3]
B
[−∞, 2]
C
[−∞, 14]
D
3
12
8
2
14
(f)
[3, 3]
A
[3, 3]
B
[−∞, 2]
C
[2, 2]
D
3
12
8
2
14
5
2

Why is it called α-β?

- α is the value of the best (i.e., highest value) choice found so far at any choice point along the path for MAX
- If *v* is worse than α, MAX will avoid it
  - prune that branch
- β is the value of the best (i.e., lowest-value) choice found so far at any choice point along the path for MIN
- If *v* is better than β, MAX will not reach it
  - prune that branch

The α-β search algorithm

```
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
```

Properties of α-β

- Pruning does not affect final result
- Good move ordering improves effectiveness of pruning
- With "perfect ordering,“ which means best move first, time complexity= $O(b^{m/2})$ - doubles depth of search with the same amount of computational resources
- With “random ordering," time complexity is roughly $O(b^{3m/4})$.
- It is worthwhile to store the evaluation of explored positions in a hash table, called **transposition table**, so that we don’t have to recompute it on subsequent occurrences. But it is not practical to keep *all* of them in the transposition table. Various strategies have been used to choose which nodes to keep and which to discard.

Another α-β pruning example

- https://www.youtube.com/watch?v=xBXHtz4Gbdo

Imperfect Real-time Decisions

- Games are often played under time limits, e.g. 3 minutes per move - not enough time to search the entire tree
- Standard approach:
  - cutoff test: e.g., depth limit (perhaps add quiescence search)
  - evaluation function: an estimate of the expected utility of the game from a given position

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

Evaluation functions

- An evaluation function should order the *terminal* states in the same way as the true utility function: states that are wins must evaluate better than draws, which in turn must be better than losses.
- An evaluation function should not be too hard to evaluate and for nonterminal states it should be strongly correlated with the actual chances of winning.
- A typical evaluation function is a linear function in which some set of coefficients is used to weight a number of "features" of the board position.
- For chess, typically linear weighted sum of features
  $Eval(s) = w_1f_1(s) + w_2f_2(s) + \ldots + w_nf_n(s)$
- e.g., $w_1 = 9$ with $f_1(s)$ = (number of white queens) – (number of black queens), etc.

Evaluation functions

- Using linear combination of features as an evaluation function involves a strong assumption that the contribution of each feature is *independent* of the values of the other features.
- The assumption may not be true. For example, bishops are more powerful in the endgame, when they have a lot of space to maneuver. Thus, weight for a feature can be adjusted to reflect this to make it more accurate.
- Weights can be learned by machine learning techniques.

Cutting off search

- Replace the two lines in α-β search that mention TERMINAL-TEST with the following line:

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

- We may use a fixed depth limit. But the limit may be unfortunate due to wild swings in the near future.
- In chess, for example, positions in which favorable captures can be made are not quiescent for an evaluation function that just counts material. Nonquiescent positions can be expanded further until quiescent positions are reached. This extra search is called a **quiescence search.**

Cutting off search

- The **horizon effect** is more difficult to eliminate in cut-off search. 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 delaying tactics.
- One strategy to mitigate the horizon effect is the **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.

Forward Pruning

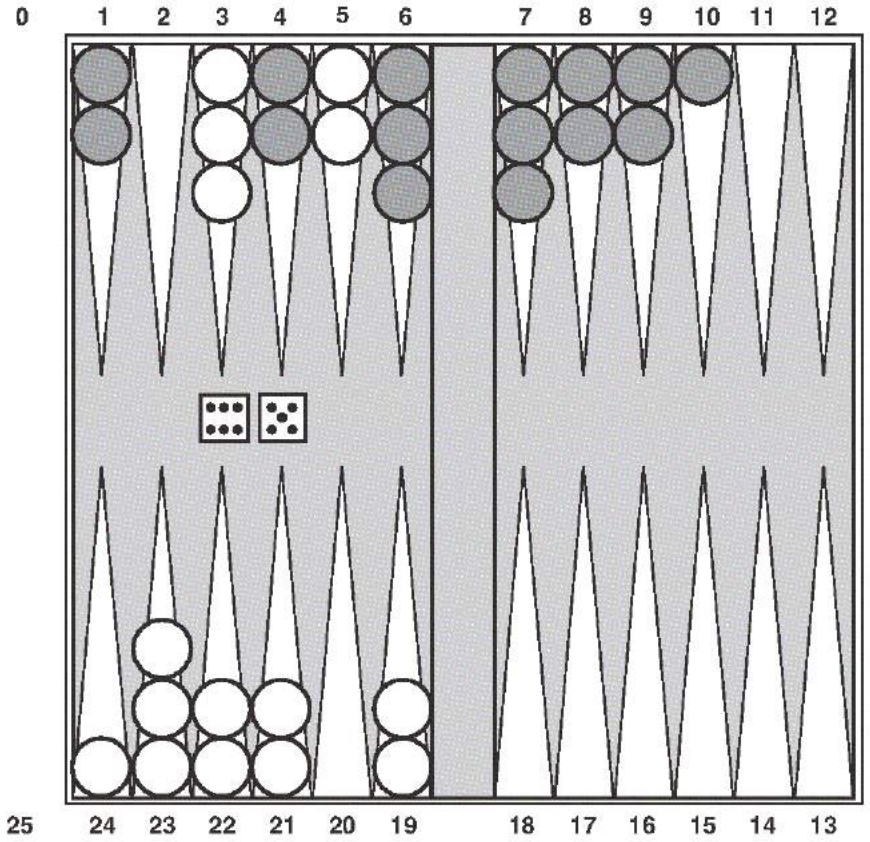
- Idea: prune some moves at a given node immediately without further consideration.
- One approach to forward pruning is **beam search**: on each ply, consider only a “beam” of the n best moves (according to the evaluation function) rather than considering all possible moves.
- The PROBCUT algorithm (Buro, 1995) is 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 prunes nodes that are *probably* outside the window. It computes this probability by doing a shallow search to compute the backed-up value v of a node and then using past experience to estimate how likely it is that a score of v at depth d in the tree would be outside (α, β).

Search vs Lookup

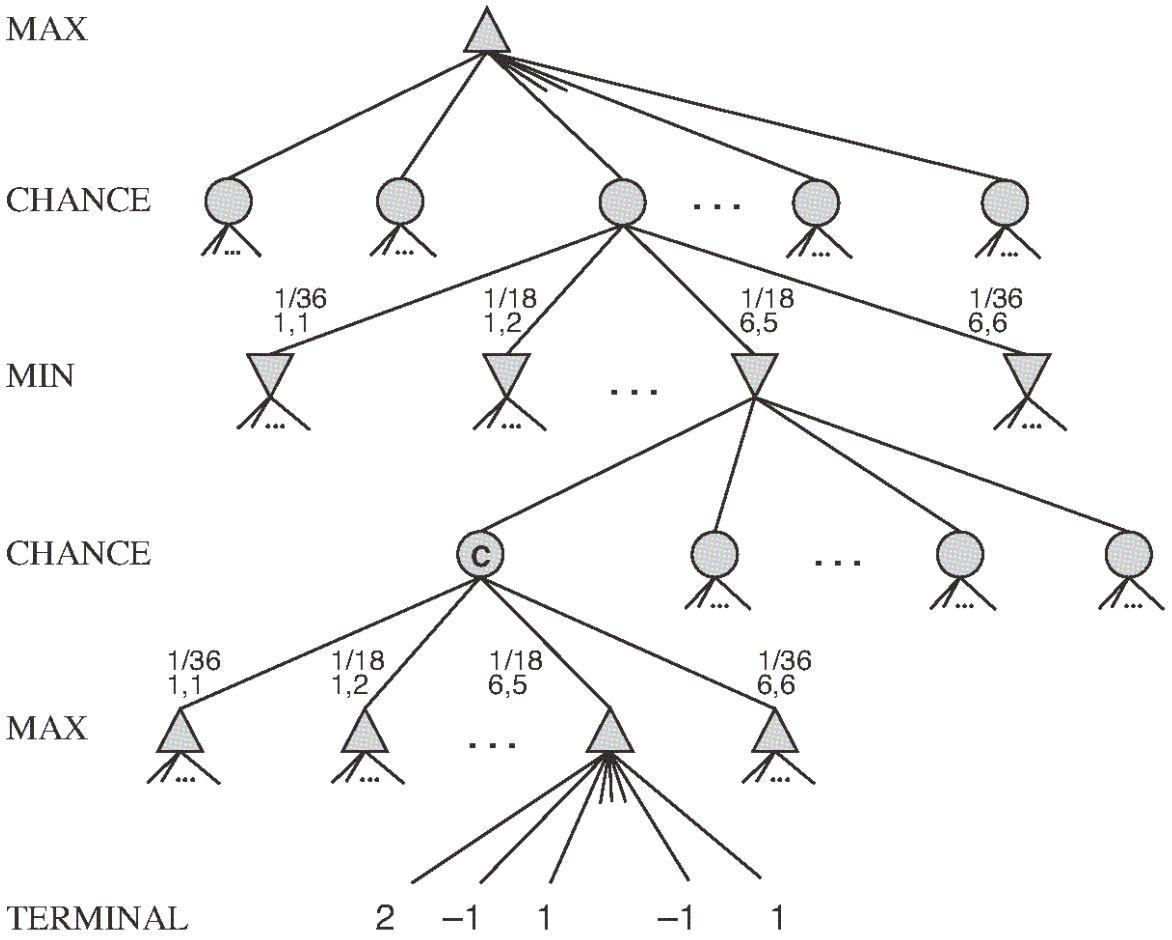
- Opening and endgame strategies in chess can often use table lookup rather search.
- For the openings, the computer is mostly relying on the expertise of humans.
- Endgame strategies can be found by performing a **retrograde** minimax search: reverse the rules of chess to do unmoves rather than moves. Any move by White that, no matter what move Black responds with, ends up in a position marked as a win, must also be a win.

Stochastic Games

- When random event is introduced into a game, it becomes a stochastic game
- Backgammon – a game that involves randomness due to dice rolling before a move



Partial game tree for a backgammon position



How to compute expected minimax value

- A stochastic game tree must include **chance nodes** in addition to MAX and MIN nodes.
- Positions do not have definite minimax values but **expected values (expectiminimax value**): the average over all possible outcomes of the chance nodes.
- Time complexity of EXPECTIMINIMAX: $O((bn)^m)$, where n is the number of possible outcomes of a random event.

How to compute expected minimax value

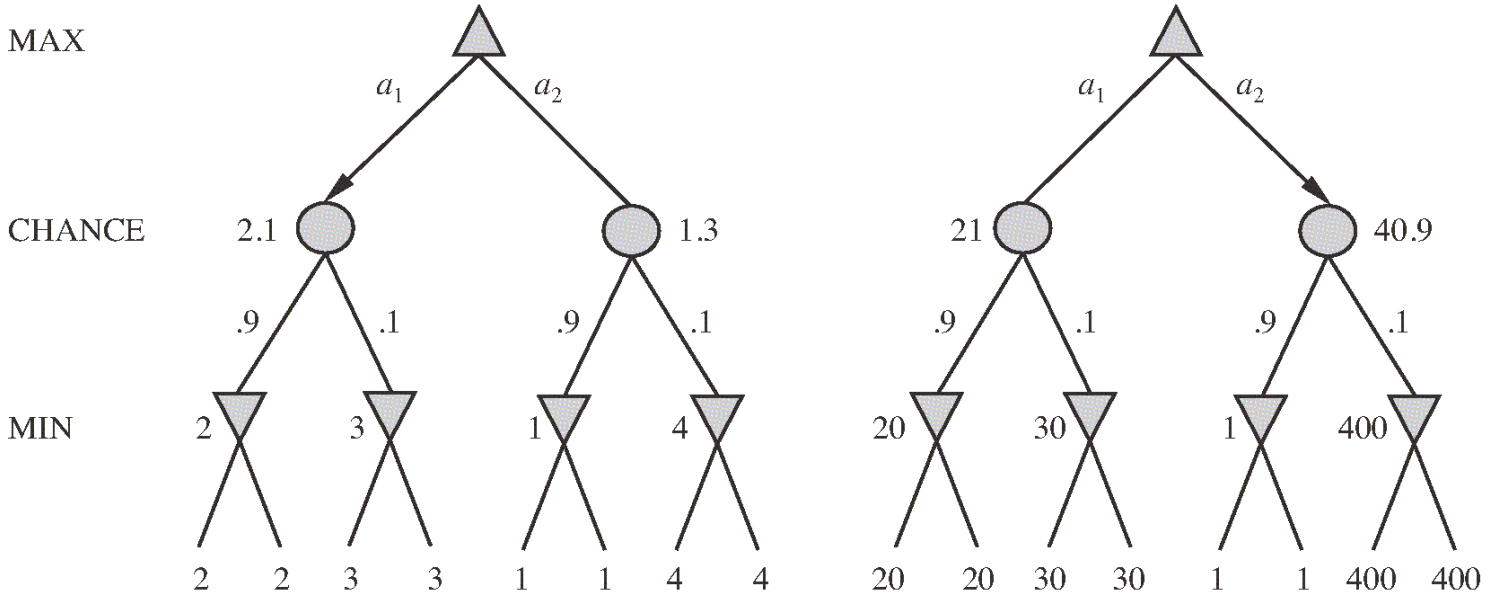
- Terminal nodes and MAX and MIN nodes (for which the dice roll is known) work exactly the same way as before. For chance nodes we compute the expected value, which is the sum of the value over all outcomes, weighted by the probability of each chance action:

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \\ \sum_r P(r)\text{EXPECTIMINIMAX}(\text{RESULT}(s,r)) & \text{if PLAYER}(s) = \text{CHANCE} \end{cases}$$

where r represents a possible dice roll (or other chance event) and $\text{RESULT}(s, r)$ is the same state as s, with the additional fact that the result of the dice roll is r.

Evaluation functions for games of chance

- We could apply an evaluation function to a cut-off position. But evaluation functions in games of chance must be carefully designed because a change in the scale of evaluation function could result in a change of optimal move.



Alpha-beta pruning for stochastic game tree search?

- In order to get good alpha-beta pruning, we need good upper and lower bounds for evaluation function. This could happen when all the utilities are bounded.
- We could use **Monte Carlo simulation** to evaluate a position. Start with an alpha–beta (or other) search algorithm. From a start position, have the algorithm play thousands of games against itself, using random dice rolls. In the case of backgammon, the resulting win percentage has been shown to be a good approximation of the value of the position, even if the algorithm has an imperfect heuristic and is searching only a few plies (Tesauro, 1995). For games with dice, this type of simulation is called a **rollout.**

Summary

- Game, minimax search, alpha-beta pruning