Adversarial Search and Game Playing

(Where making good decisions requires respecting your opponent)

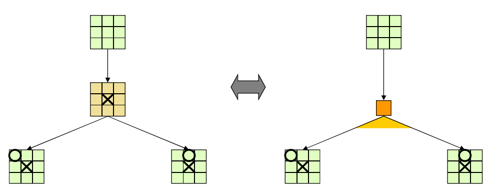
R&N: Chap. 5

- Games like Chess or Go are compact settings that mimic the uncertainty of interacting with the natural world
- For centuries humans have used them to exert their intelligence
- Recently, there has been great success in building game programs that challenge human supremacy

---

Relation to Previous Lecture

- Here, uncertainty is caused by the actions of another agent (MIN), which competes with our agent (MAX)



Relation to Previous Lecture

- Here, uncertainty is caused by the actions of another agent (MIN), which competes with our agent (MAX)
- MIN wants MAX to fail (and vice versa)
- No plan exists that guarantees MAX's success regardless of which actions MIN executes (the same is true for MIN)
- At each turn, the choice of which action to perform must be made within a specified time limit
- The state space is enormous: only a tiny fraction of this space can be explored within the time limit

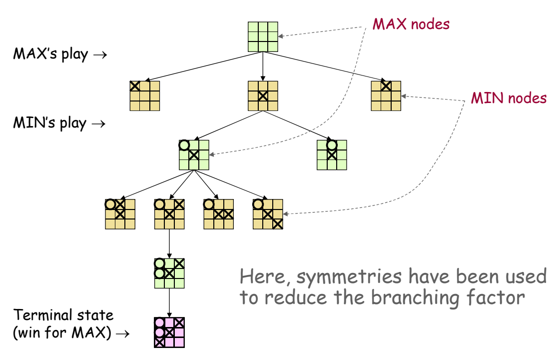
---

Specific Setting

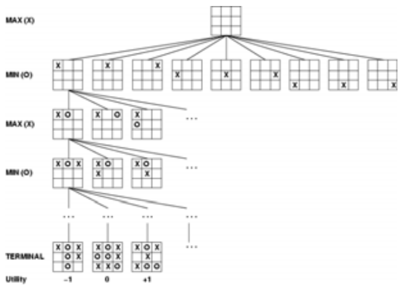
Two-player, turn-taking, deterministic, fully observable, zero-sum, time-constrained game

- State space
- Initial state
- Successor function: it tells which actions can be executed in each state and gives the successor state for each action
- MAX's and MIN's actions alternate, with MAX playing first in the initial state
- Terminal test: it tells if a state is terminal and, if yes, if it's a win or a loss for MAX, or a draw
- All states are fully observable

Game Tree

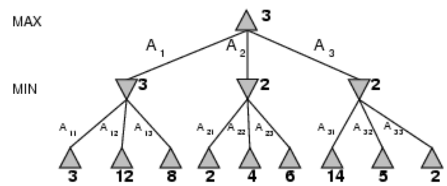


Game tree (2-player, deterministic, turns)



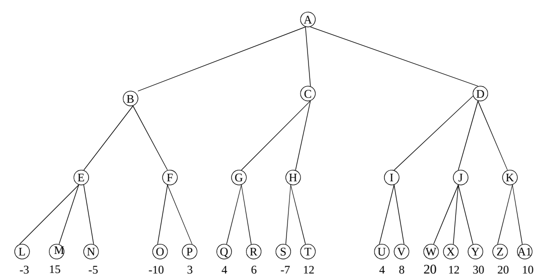
Minimax

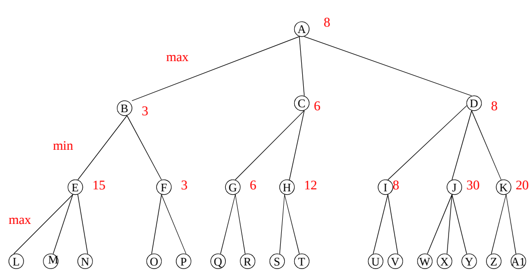
- Perfect play for deterministic game
- Idea: choose move to position with highest minimax value = best achievable payoff against best play
- E.g., 2-ply game:



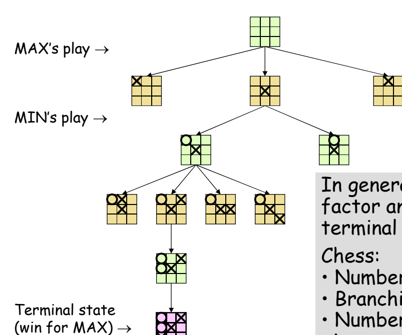
Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
   v ← MAX-VALUE(state)
   return the action in SUCCESSORS(state) with value v

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

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





Game Tree



In general, the branching factor and the depth of terminal states are large

Chess:
- Number of states: ~10^{40}
- Branching factor: ~35
- Number of total moves in a game: ~100

Choosing an Action: Basic Idea

1) Using the current state as the initial state, build the game tree uniformly to the maximal depth h (called horizon) feasible within the time limit
2) Evaluate the states of the leaf nodes
3) Back up the results from the leaves to the root and pick the best action assuming the worst from MIN

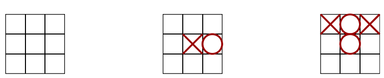
→ Minimax algorithm

Evaluation Function

- Function e: state s → number e(s)
- e(s) is a heuristics that estimates how favorable s is for MAX
- e(s) > 0 means that s is favorable to MAX (the larger the better)
- e(s) < 0 means that s is favorable to MIN
- e(s) = 0 means that s is neutral

Example: Tic-tac-Toe

e(s) = number of rows, columns, and diagonals open for MAX – number of rows, columns, and diagonals open for MIN



8–8 = 0 6–4 = 2 3–3 = 0

Construction of an Evaluation Function

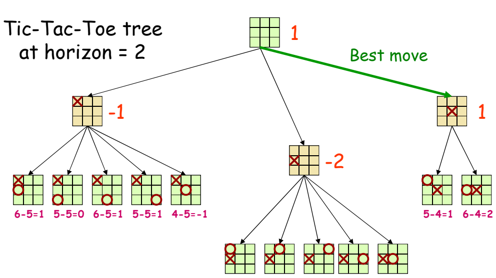
- Usually a weighted sum of "features":

$$e(s) = \sum_{i=1}^{n} w_i f_i(s)$$

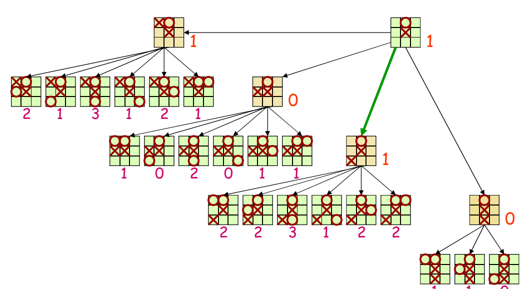
- Features may include
  - Number of pieces of each type
  - Number of possible moves
  - Number of squares controlled

Backing up Values

Tic-Tac-Toe tree at horizon = 2



Continuation



Why using backed-up values?

- At each non-leaf node N, the backed-up value is the value of the best state that MAX can reach at depth h if MIN plays well (by the same criterion as MAX applies to itself)
- If e is to be trusted in the first place, then the backed-up value is a better estimate of how favorable STATE(N) is than e(STATE(N))

Minimax Algorithm

1. Expand the game tree uniformly from the current state (where it is MAX's turn to play) to depth h
2. Compute the evaluation function at every leaf of the tree
3. Back-up the values from the leaves to the root of the tree as follows:
   a. A MAX node gets the maximum of the evaluation of its successors
   b. A MIN node gets the minimum of the evaluation of its successors
4. Select the move toward a MIN node that has the largest backed-up value

Minimax Algorithm

1. Expand the game tree uniformly from the current state (where it is MAX's turn to play) to depth h
2. Compute the evaluation function at every leaf of the tree
3. Back-up the values from the leaves to the root of the tree as
   a. A MAX no
   successors
   b. A MIN no
   successors
4. Select the move toward a MIN node that has the largest backed-up value

Horizon: Needed to return a decision within allowed time

Repeated States

Left as an exercise

[Distinguish between states on the same path and states on different paths]

Game Playing (for MAX)

Repeat until a terminal state is reached

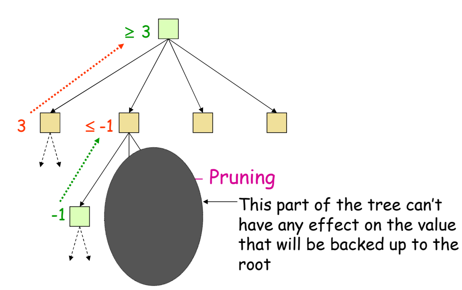
1. Select move using Minimax
2. Execute move
3. Observe MIN's move

Note that at each cycle the large game tree built to horizon h is used to select only one move

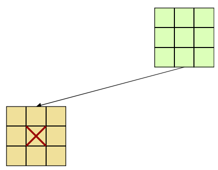
All is repeated again at the next cycle (a sub-tree of depth h-2 can be re-used)

Can we do better?

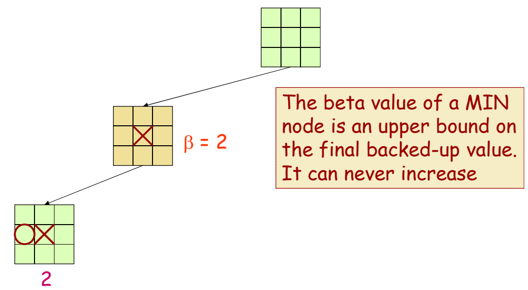
Yes ! Much better !



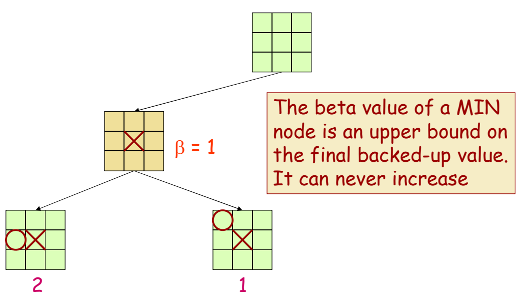
Example



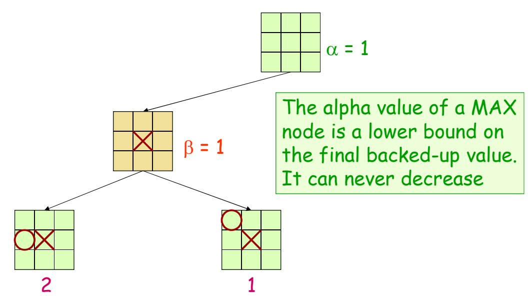
Example



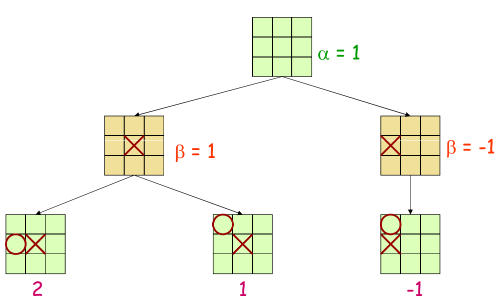
Example



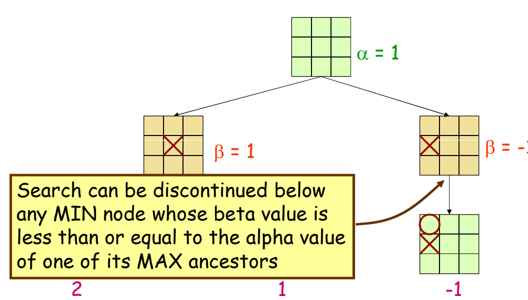
Example



Example



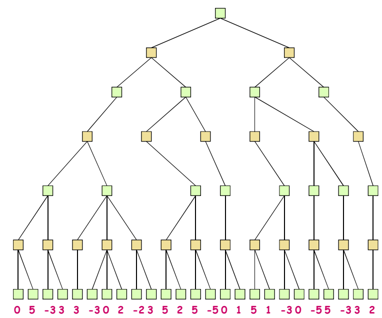
Example



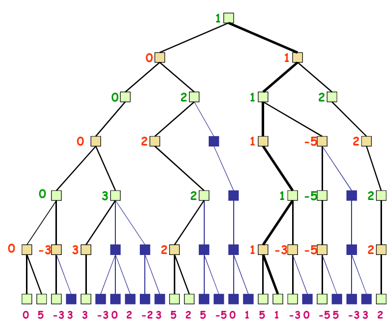
Alpha-Beta Pruning

- Explore the game tree to depth h in depth-first manner
- Back up alpha and beta values whenever possible
- Prune branches that can't lead to changing the final decision

Example



Example



Alpha-Beta Algorithm

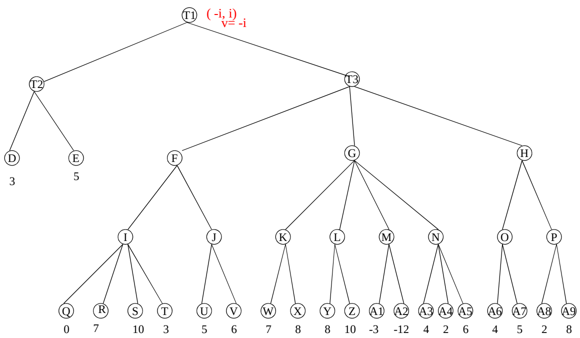
- Update the alpha/beta value of the parent of a node N when the search below N has been completed or discontinued
- Discontinue the search below a MAX node N if its alpha value is ≥ the beta value of a MIN ancestor of N
- Discontinue the search below a MIN node N if its beta value is ≤ the alpha value of a MAX ancestor of N

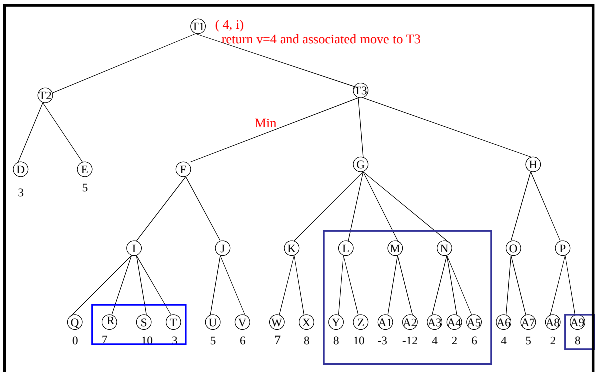
The α-β algorithm

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

function MAX-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           α, the value of the best alternative for MAX along the path to state
           β, the value of the best alternative for MIN along the path to state
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s, α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v
```

The α-β algorithm

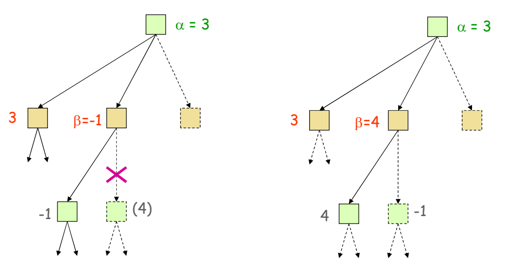
```
function MIN-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           α, the value of the best alternative for MAX along the path to state
           β, the value of the best alternative for MIN along the path to state
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for a, s in SUCCESSORS(state) do
      v ← MIN(v, MAX-VALUE(s, α, β))
      if v ≤ α then return v
      β ← MIN(β, v)
   return v
```





How much do we gain?

Consider these two cases:



How much do we gain?

- Assume a game tree of uniform branching factor b
- Minimax examines $O(b^h)$ nodes, so does alpha-beta in the worst-case
- The gain for alpha-beta is maximum when:
  - The MIN children of a MAX node are ordered in increasing backed up values
  - The MAX children of a MIN node are ordered in decreasing backed up values
- Then alpha-beta examines $O(b^{h/2})$ nodes [Knuth and Moore, 1975]
- But this requires an oracle (if we knew how to order nodes perfectly, we would not need to search the game tree)
- If nodes are ordered at random, then the average number of nodes examined by alpha-beta is ~$O(b^{3h/4})$

Heuristic Ordering of Nodes

- Order the nodes below the root according to the values backed-up at the previous iteration
- Order MIN (resp.MAX) nodes by decreasing (increasing) values of the evaluation function e computed at these nodes

Other Improvements

- Adaptive horizon + iterative deepening
- Extended search: Retain k>1 best paths, instead of just one, and extend the tree at greater depth below their leaf nodes to (help dealing with the "horizon effect")
- Singular extension: If a move is obviously better than the others in a node at horizon h, then expand this node along this move
- Use transposition tables to deal with repeated states
- Null-move search

Deterministic games in practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a precomputed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.
- Chess: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.
- Othello: human champions refuse to compete against computers, who are too good.
- Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.