Adversarial Search

- Adversarial search problems ≡ games
- They occur in multi-agent competitive environments
- There is an **opponent** we can' t control planning again us!
- Game vs. search: optimal solution is not a sequence of actions but a **strategy** (policy) If opponent does *a*, agent does *b*, else if opponent does *c*, agent does *d*, etc.
- Tedious and fragile if hard-coded (i.e., implemented with rules)
- Good news: Games are modeled as **search problems** and use **heuristic evaluation** functions.

Types of games

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

We are mostly interested in deterministic games, fully observable environments, zero-sum, where two agents act alternately.

Zero-sum Games

- Adversarial: Pure competition.
- Agents have different values on the outcomes.
- One agent maximizes one single value, while the other minimizes it.
- Each move by one of the players is called a "ply."

One function: one agent maximizes it and one minimizes it!

Embedded thinking

Embedded thinking or backward reasoning!

计算空间非常大

- One agent is trying to figure out what to do.
- How to decide? He thinks about the consequences of the possible actions.
- He needs to think about his opponent as well...
- The opponent is also thinking about what to do etc.
- Each will imagine what would be the response from the opponent to their actions.
- This entails an embedded thinking.

Formalization

- The **initial state**
- Player(s): defines which player has the move in state s. Usually taking turns.
- Actions(s): returns the set of legal moves in s
- **Transition** function: $S \times A \rightarrow S$ defines the result of a move
- Terminal test: True when the game is over, False otherwise. States where game ends are called **terminal states**
- $Utility(s, p)$: **utility function** or objective function for a game that ends in terminal state s for player p. In Chess, the outcome is a win, loss, or draw with values +1, 0, 1/2. For tic-tac-toe we can use a utility of +1, -1, 0.

Minimax

- Two players: Max and Min
- Players alternate turns
- Max moves first
- Max maximizes results
- Min minimizes the results
- Compute each node' s minimax value that is the best achievable utility against an optimal adversary
- Minimax value ≡ best achievable payoff against best play

- Find the optimal strategy for Max:
  – Depth-first search of the game tree
  – An optimal leaf node could appear at any depth of the tree
  – Minimax principle: compute the utility of being in a state assuming both players play optimally from there until the end of the game
  – Propagate minimax values up the tree once terminal nodes are discovered

- If state is terminal node: Value is utility(state)
- If state is MAX node: Value is highest value of all successor node values (children)
- If state is MIN node: Value is lowest value of all successor node values (children)

For a state s minimax(s) =

$$\begin{cases} Utility(s) & \text{if Terminal-test(s)} \\ max_{a \in Actions(s)} \text{ minimax(Result(s,a))} & \text{if Player(s) = Max} \\ min_{a \in Actions(s)} \text{ minimax(Result(s,a))} & \text{if Player(s) = Min} \end{cases}$$

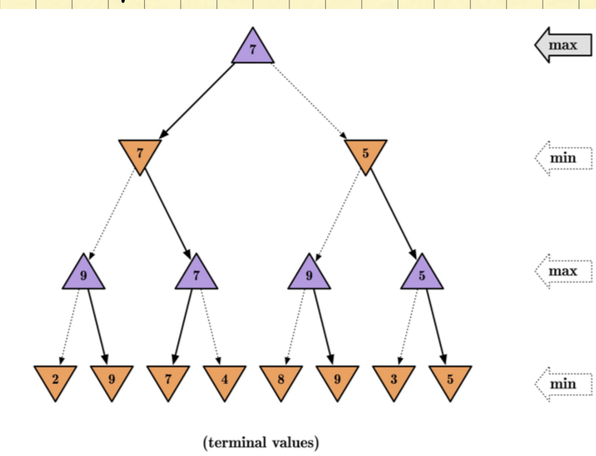
The minimax algorithm

```
/* Find the child state with the lowest utility value */

function MINIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨minChild, minUtility⟩ = ⟨NULL, ∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MAXIMIZE(child)

        if utility < minUtility:
            ⟨minChild, minUtility⟩ = ⟨child, utility⟩

    return ⟨minChild, minUtility⟩
```

```
/* Find the child state with the highest utility value */

function MAXIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨maxChild, maxUtility⟩ = ⟨NULL, −∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MINIMIZE(child)

        if utility > maxUtility:
            ⟨maxChild, maxUtility⟩ = ⟨child, utility⟩

    return ⟨maxChild, maxUtility⟩
```

```
/* Find the child state with the highest utility value */

function DECISION(state)
    returns STATE :

    ⟨child, _⟩ = MAXIMIZE(state)

    return child
```

Example



Properties of minimax

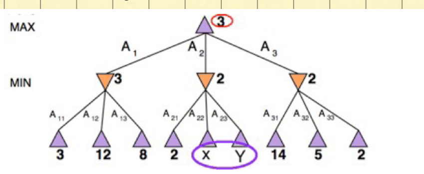
- Optimal (opponent plays optimally) and complete (finite tree)
- DFS time: $O(b^m)$
- DFS space: $O(bm)$
  - **Tic-Tac-Toe**
    - ≈ 5 legal moves on average, total of 9 moves (9 plies).
    - 5^9 = 1,953,125 •9! = 362,880 terminal nodes
  - **Chess**
    - $b \approx 35$ (average branching factor)
    - $d \approx 100$ (depth of game tree for a typical game)
    - $b^d \approx 35^{100} \approx 10^{154}$ nodes
  - **Go** branching factor starts at 361 (19X19 board)

Case of limited resources

- Problem: In real games, **we are limited in time, so we can' t search the leaves.**
- To be practical and run in a reasonable amount of time, minimax can only search to some depth.
- More plies make a big difference.
- **Solution:**
  1. Replace terminal utilities with an evaluation function for non-terminal positions.
  2. Use Iterative Deepening Search (IDS).
  3. Use pruning: eliminate large parts of the tree.

在terminal状态之前也有evaluation函数.
限定搜索深度
剪枝

α-β pruning



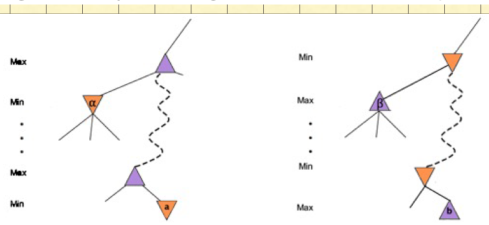
$$Minimax(root) = max(min(3,12,8), min(2,X,Y), min(14,5,2))$$
$$= max(3, min(2,X,Y), 2)$$
$$= max(3, Z, 2) \quad \text{where } Z = min(2,X,Y) \leq 2$$
$$= 3$$

Minimax decisions are independent of the values of X and Y.

- **Strategy:** Just like minimax, it performs a DFS.
- **Parameters**: Keep track of two bounds
  - α: largest value for Max across seen children (current lower bound on MAX' s outcome).
  - β: lowest value for MIN across seen children (current upper bound on MIN' s outcome).
- **Initialization:** $\alpha = -\infty$, $\beta = \infty$
- **Propagation**: Send α, β values down during the search to be used for pruning.
  - Update α, β values by propagating upwards values of terminal nodes.
  - Update α only at Max nodes and update β only at Min nodes.
- **Pruning:** Prune any remaining branches whenever $\alpha \geq \beta$.

α：该节点之上最大值
β：该节点之下最小值

当α≥β时停止，从而剪枝



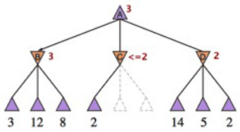
- If α is better than a for Max, then Max will avoid it, that is prune that branch.
- If β is better than b for Min, then Min will avoid it, that is prune that branch.

```
/* Find the child state with the lowest utility value */

function MINIMIZE(state, α, β)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨minChild, minUtility⟩ = ⟨NULL, ∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MAXIMIZE(child, α, β)

        if utility < minUtility:
            ⟨minChild, minUtility⟩ = ⟨child, utility⟩

        if minUtility ≤ α:  →即α≥β
            break

        if minUtility < β:
            β = minUtility

    return ⟨minChild, minUtility⟩
```

```
/* Find the child state with the highest utility value */

function MAXIMIZE(state, α, β)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨maxChild, maxUtility⟩ = ⟨NULL, −∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MINIMIZE(child, α, β)

        if utility > maxUtility:
            ⟨maxChild, maxUtility⟩ = ⟨child, utility⟩

        if maxUtility ≥ β:  →即α≥β
            break

        if maxUtility > α:
            α = maxUtility

    return ⟨maxChild, maxUtility⟩
```

```
/* Find the child state with the highest utility value */

function DECISION(state)
    returns STATE :
                                     α   β
    ⟨child, _⟩ = MAXIMIZE(state, −∞, ∞)

    return child
```

Move ordering



- It does matter as it affects the effectiveness of $\alpha-\beta$ pruning.
- Example: We could not prune any successor of D because the worst successors for Min were generated first. If the third one (leaf 2) was generated first we would have pruned the two others (14 and 5).
- Idea of ordering: examine first successors that are likely best.

搜索先后会影响剪枝效果

- **Worst ordering:** no pruning happens (best moves are on the right of the game tree). Complexity $O(b^m)$.
- **Ideal ordering:** lots of pruning happens (best moves are on the left of the game tree). Complexity $O(b^{m/2})$ (in practice).
- **How to find a good ordering?**
  - Remember the best moves from shallowest nodes.
  - Use domain knowledge.
  - Bookkeep the states, they may repeat!

最坏情况下即为minimax

最好情况下能减非常多，搜索深度加倍(相同时间)

Real-time decisions

- Minimax: generates the entire game search space
- $\alpha-\beta$ algorithm: prune large chunks of the trees
- BUT $\alpha-\beta$ still has to go all the way to the leaves
- Impractical in real-time (moves has to be done in a reasonable amount of time)
- Solution: bound the depth of search (cut off search) and **replace** *utility*(s) **with** *eval*(s), an evaluation function to **estimate** value of current board configurations

用一个启发函数eval来代替终局的utility函数

- *eval*(s) is a heuristic at state s
  - E.g., Othello: # white pieces - # black pieces
  - E.g., Chess: value of all white pieces - value of all black pieces
  - turn non-terminal nodes into terminal leaves!
- An ideal evaluation function would rank terminal states in the same way as the true utility function; but must be fast
- Typical to define features, make the function a linear weighted sum of the features
- Use domain knowledge to craft the best and useful features.

- How does it works?
  - Select useful features $f_1, \ldots, f_n$ e.g., Chess: # pieces on board, value of pieces (1 for pawn, 3 for bishop, etc.)
  - Weighted linear function:

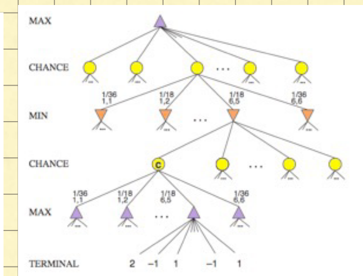
$$eval(s) = \sum_{i=1}^{n} w_i f_i(s)$$

  - **Learn** w_i from the examples
  - Deep blue uses about 6,000 features!

可能有多个影响因素，可加上权重线性组合.

Stochastic games

- Include a random element (e.g., throwing a dice).
- Include chance nodes.
- Backgammon: old board game combining skills and chance.
- The goal is that each player tries to move all of his pieces off the board before his opponent does.



Partial game tree for Backgammon.

Algorithm **Expectiminimax** generalized Minimax to handle chance nodes as follows:

- If state is a Max node then return the highest Expectiminimax-Value of Successors(state)
- If state is a Min node then return the lowest Expectiminimax-Value of Successors(state)
- If state is a chance node then return average of Expectiminimax-Value of Successors(state)

For a state *s*:

Expectiminimax(*s*) =

$$\begin{cases} Utility(s) & \text{if Terminal-test(s)} \\ max_{a \in Actions(s)} \text{ Expectiminimax(Result(s,a))} & \text{if Player(s) = Max} \\ min_{a \in Actions(s)} \text{ Expectiminimax(Result(s,a))} & \text{if Player(s) = Min} \\ \sum_r P(r) \text{ Expectiminimax(Result(s,r))} & \text{if Player(s) = Chance} \end{cases}$$

Where *r* represents all chance events (e.g., dice roll), and Result(*s*, *r*) is the same state as *s* with the result of the chance event is *r*.

Games: conclusion

- Games are modeled in AI as a search problem .
- Minimax algorithm choses the best move given an optimal play from the opponent.
- Minimax goes all the way down the tree which is not practical give game time constraints.
- Alpha-Beta pruning can reduce the game tree search which allow to go deeper in the tree within the time constraints.
- Pruning, bookkeeping, evaluation heuristics, node re-ordering and IDS are effective in practice.