Machine Learning in Practice
#7-1: Combinatorial Games

Sang-Hyun Yoon

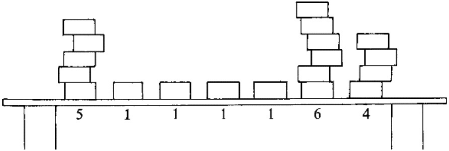
Summer 2019

Outline

Combinatorial Games: Informal Characterization

2-player sequential games with perfect information

- e.g.: Nim, Go, chess, checkers (but not poker, tetris, ...)



- There are 2 players who alternate moves.
  - extension: N-player games (e.g. Blokus with $N = 4$)
- There are no chance devices (e.g. dice).
- Every information is known to both.
  - information: possible moves, history of game states, ...
- Both players play **optimally** (well-defined?).
  - Optimal strategies of 2 players are **interdependent**!

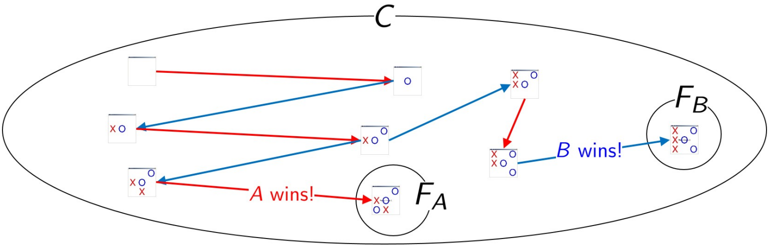
Combinatorial Games: Formulation

A combinatorial game is a tuple (C, F_A, F_B, m_A, m_B) where

- C : the set of all possible positions (configurations)
- $F_i \subset C$: the set of ending positions of player $i \in \{A, B\}$
- $m_i : C \setminus F_{i-1} \to 2^C$: possible moves of player i

Starting from a starting position $c_0 \in C$, the first player that reaches its ending position wins the game:

$$c^0 \Rightarrow c_A^1 \in m_A(c^0) \Rightarrow c_B^1 \in m_B(c_A^1) \Rightarrow c_A^2 \in m_A(c_B^1) \Rightarrow \ldots \Rightarrow c_i^k \in F_i$$



Example: Go

(C, F_A, F_B, m_A, m_B) (player A/B: Black/White)

- $C = \{백, 흑, 무\}^{[19]^2}$ $(= \{c : [19]^2 \to \{백, 흑, 무\}\})$
  - Under the repetition rule, $C = \{백, 흑, 무\}^{[19]^2} \times 2^{[19]^2}$
- $F_i = \{c \in C \mid m_A(c) = m_B(c) = \emptyset$ and player i's score is higher at $c\}$
- $m_i(c) = \{\tau(c_{pq}) \,\big|\, c(p,q) = 무\}$ where

$$c_{pq}(x,y) = \begin{cases} i & \text{if } (x,y) = (p,q) \\ c(x,y) & \text{otherwise} \end{cases}$$

  - where $\tau : C \to C$; $(\tau(c))(x,y) =$

$$\begin{cases} 무 & \text{if } c(x,y) \text{ is a dead stone} \\ c(x,y) & \text{otherwise} \end{cases}$$

Winning/Losing Positions

- $W_A = \{c \in C \mid \alpha_A(c)\}$ winning positions of player A
- $L_A = \{c \in C \mid \beta_A(c)\}$ losing positions of player A

where

- $$\begin{aligned}\alpha_A(c) = \text{“}&\exists c_A^1 \in m_A(c), \forall c_B^1 \in m_B(c_A^1),\\ &\exists c_A^2 \in m_A(c_B^1), \forall c_B^2 \in m_B(c_A^2), \cdots,\\ &\exists c_A^k \in m_A(c_B^{k-1}), c_A^k \in F_A\text{”}\end{aligned}$$
  - i.e. starting/resuming from c, A can force a win
- $$\begin{aligned}\beta_A(c) = \text{“}&\forall c_A^1 \in m_A(c), \exists c_B^1 \in m_B(c_A^1),\\ &\forall c_A^2 \in m_A(c_B^1), \exists c_B^2 \in m_B(c_A^2), \cdots,\\ &\forall c_A^k \in m_A(c_B^{k-1}), \exists c_B^k \in m_B(c_A^k), c_B^k \in F_B\text{”}\end{aligned}$$
  - i.e. A cannot force a win if B does its best

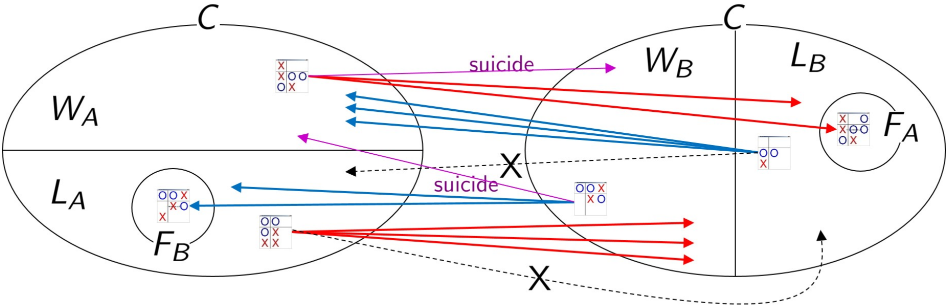
- $W_A \cap L_A = \emptyset$ can be easily proved by De Morgan's laws.
  - winning/losing positions are well-defined
- W_B and L_B can be defined similarly.

Winning/Losing Positions ⇒ Winning Strategy

Proposition (easily derivable from the definition of W_i/L_i)

- $c \in W_A \iff \exists c' \in m_A(c),\ c' \in L_B$
- $c' \in L_B \iff \forall c \in m_B(c'),\ c \in W_A$

- From every winning position, a player can($\exists$) move to the other player's losing position
- From every losing position, a player should($\forall$) move to the other player's winning position

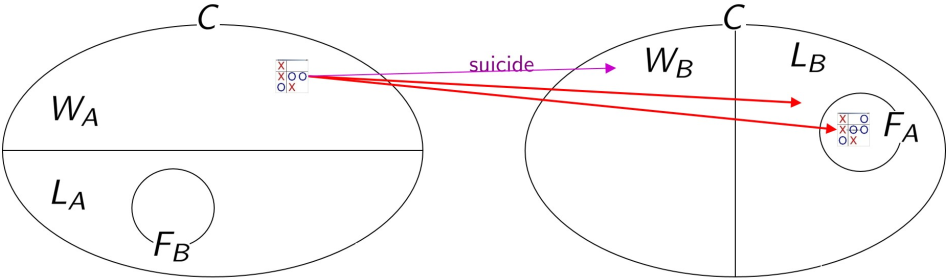


Winning/Losing Positions $\Rightarrow$ Winning Strategy

Proposition (easily derivable from the definition of W_i/L_i)

- $c \in W_A \iff \exists c' \in m_A(c),\ c' \in L_B$

- From every winning position, a player can($\exists$) move to the other player's losing position

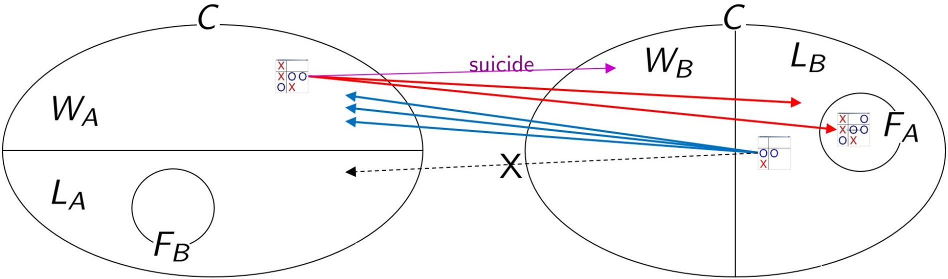


Winning/Losing Positions $\Rightarrow$ Winning Strategy

Proposition (easily derivable from the definition of W_i/L_i)

- $c \in W_A \iff \exists c' \in m_A(c),\ c' \in L_B$
- $c' \in L_B \iff \forall c \in m_B(c'),\ c \in W_A$

- From every winning position, a player can($\exists$) move to the other player's losing position
- From every losing position, a player should($\forall$) move to the other player's winning position

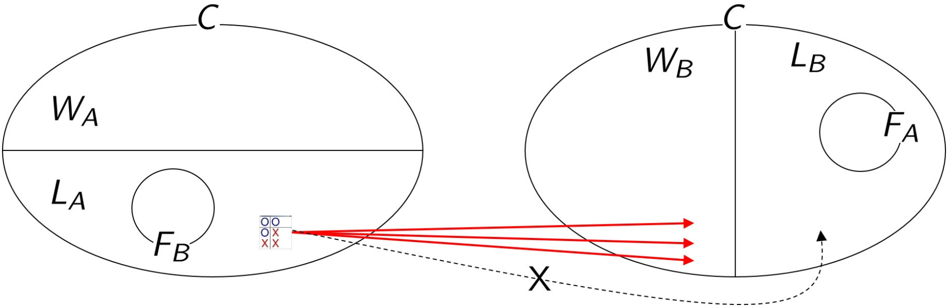


Winning/Losing Positions ⇒ Winning Strategy

Proposition (easily derivable from the definition of W_i/L_i)

- $c' \in L_B \iff \forall c \in m_B(c'),\ c \in W_A$

- From every losing position, a player should($\forall$) move to the other player's winning position

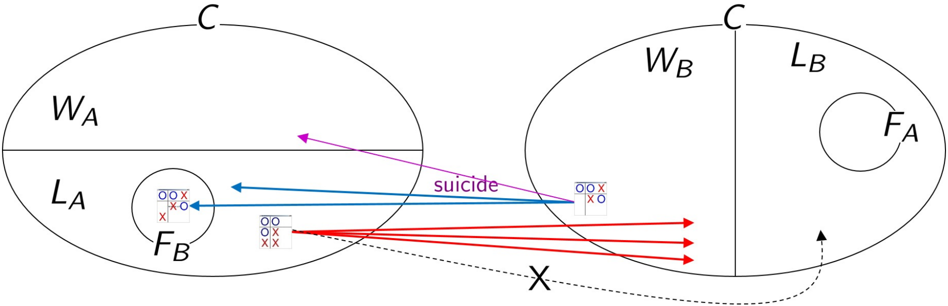


Winning/Losing Positions $\Rightarrow$ Winning Strategy

Proposition (easily derivable from the definition of W_i/L_i)

- $c \in W_A \iff \exists c' \in m_A(c),\ c' \in L_B$
- $c' \in L_B \iff \forall c \in m_B(c'),\ c \in W_A$

- From every winning position, a player can($\exists$) move to the other player's losing position
- From every losing position, a player should($\forall$) move to the other player's winning position



How to Compute Winning/Losing Positions & Winning Strategy

Recursive Definition of Winning/Losing Positions

- $W_A = \{c \in C \mid \exists c' \in m_A(c),\ c' \in L_B\}$
- $L_B = \{c' \in C \mid \forall c \in m_B(c'),\ c \in W_A\}$
- $W_B = \{c' \in C \mid \exists c \in m_B(c'),\ c \in L_A\}$
- $L_A = \{c \in C \mid \forall c' \in m_A(c),\ c' \in W_B\}$
- $F_A \subseteq L_B$
- $F_B \subseteq L_A$

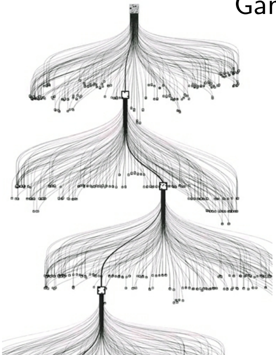
```
def in_W_A(c):
    if c ∈ F_B:
        return False
    for each c' ∈ m_A(c):
        if in_W_B(c') = False:
            return True
    return False
```

```
def in_W_B(c'):
    if c' ∈ F_A:
        return False
    for each c ∈ m_B(c'):
        if in_W_A(c) = False:
            return True
    return False
```

Exact Exhaustive Algorithm

```
def in_W_A(c):
    if c ∈ F_B:
        return False
    for each c' ∈ m_A(c):
        if in_W_B(c') = False:
            return True
    return False
```

```
def in_W_B(c'):
    if c' ∈ F_A:
        return False
    for each c ∈ m_B(c'):
        if in_W_A(c) = False:
            return True
    return False
```



Game-tree of Go: about $5 \cdot 10^{359}$ nodes!

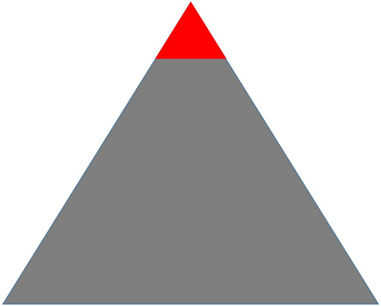
- The above algorithm is exact, but impractical

Inexact Fast Heuristics: Search Space Pruning (to be coverd later)

```
def max_A(c, depth):
    if depth ≥ d_th:
        return value(c)
    for each c' ∈ m_A(c):
        v := min_B(c', depth+1)
        max_v := max{max_v, v}
    return max_v
```

```
def min_B(c', depth):
    if depth ≥ d_th:
        return value(c)
    for each c ∈ m_B(c'):
        v := max_A(c, depth+1)
        min_v := min{min_v, v}
    return min_v
```

Minimax/α-β Search

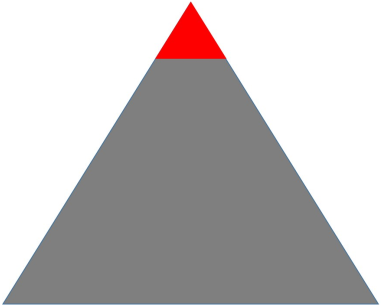


Inexact Fast Heuristics: Search Space Pruning (to be coverd later)

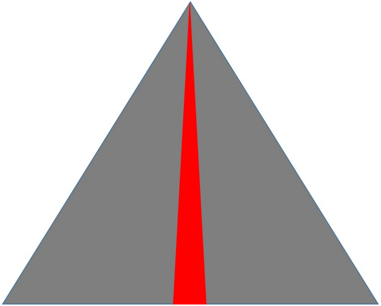
```
def max_A(c, depth):
    if depth ≥ d_th:
        return value(c)
    for each c' ∈ m_A(c):
        v := min_B(c', depth+1)
        max_v := max{max_v, v}
    return max_v
```

```
def min_B(c', depth):
    if depth ≥ d_th:
        return value(c)
    for each c ∈ m_B(c'):
        v := max_A(c, depth+1)
        min_v := min{min_v, v}
    return min_v
```

Minimax/α-β Search



MCTS (Monte-Carlo Tree Search)



Any Hope for Solving Game Exactly in Reasonable Time?

- Most impartial games can be solved in polynomial-time
  - They are poly-time reducible to Nim by Sprague-Grundy thm
  - Nim is solvable in poly-time
- Most partizan games (e.g. Go, chess) are provably hard
  - PSPACE-hard, in general
- Some (artifact) partizan games can be solved in poly-time
  - Those that appear in math/programming competitions
  - For them, the boundaries bet'n winning/losing positions that are computationally easy are derivable by math analysis
- Small-sized instances of hard games can be solved in reasonable time
  - e.g. 3×3 Go

Outline

How to (exhaustively) compute the winner and a winning strategy?

- $F_B \subseteq L_A$, $F_A \subseteq L_B$
- $W_A = \{c \in C \mid \exists c' \in m_A(c),\ c' \in L_B\}$
- $L_A = \{c \in C \mid \forall c' \in m_A(c),\ c' \in W_B\}$

Exhaustive bottom-up algorithm (exponential space)

$L_A := L_B := F$; $W_A := W_B := \{\}$

do {

$W_A := W_A \cup \{c \mid m_A(c) \cap L_B \neq \emptyset\}$

$W_B := W_B \cup \{c \mid m_B(c) \cap L_A \neq \emptyset\}$

$L_A := L_A \cup \{c \mid m_A(c) \subseteq W_B\}$

$L_B := L_B \cup \{c \mid m_B(c) \subseteq W_A\}$

} while (there is a change in one of W_A, L_A, W_B, L_B)

if (starting position is in L_A)

A is the winner

else

B is the winner

Exhaustive top-down recursive algorithm (poly-space)

- $F_B \subseteq L_A\,,\quad F_A \subseteq L_B$
- $W_A \;=\; \{c \in C \,|\, \exists c' \in m_A(c),\; c' \in L_B\}$
- $L_A \;=\; \{c \in C \,|\, \forall c' \in m_A(c),\; c' \in W_B\}$

```
def AcanWin(c): # True iff c ∈ W_A
    if c ∈ F_B: return False

    for each c' ∈ m_A(c):
        if not BcanWin(c'):
            return True      # c ⇝ c' guarantee A's winning
    return False  # whichever position A selects, A loses

def BcanWin(c): # True iff c ∈ W_B
    if c ∈ F_A: return False

    for each c' ∈ m_B(c):
        if not AcanWin(c'):
            return True      # c ⇝ c' guarantee B's winning
    return False  # whichever position B selects, B loses

# A is the winner iff AcanWin(starting_position) is True
```

Exhaustive top-down recursive algorithm for impartial games

(C, F_A, F_B, m_A, m_B) is called an impartial game (대칭 게임) if

- $F_A = F_B$ and $m_A = m_B$

Otherwise, it is called a partizan game (비대칭 게임)

```
def canWin(c): # True iff c ∈ W_A (= W_B)
    if c ∈ F_B (= F_A): return False
    for each c' ∈ m_A(c) (= m_B(c)):
        if not canWin(c'):
            return True          # c ⇝ c' guarantee winning
    return False    # whichever position selects, it loses

# A is the winner iff canWin(starting_position) is True
```

Rules of Go

- http://gall.dcinside.com/board/view/?id=baduk&no=30064
- 남은 수업 기간 동안 규칙을 단순화 시킨 5×5 바둑 문제를 다루므로 위 규칙을 어느 정도 이해는 하고 있어야 함
  - 규칙 자체는 구현 완료된 형태로 제공됨
- 다음 수업: Monte-Carlo tree search & minimax/α-β search
  - Exhaustive search에서 width/height를 가지치기(pruning)한 것으로 오늘 내용을 완벽히 숙달해야 함
- 다음^2 수업: Monte-Carlo policy iteration
- 최종과제: MCTS/α-β + 강화학습
  - 풀리그전 결과로 점수 결정