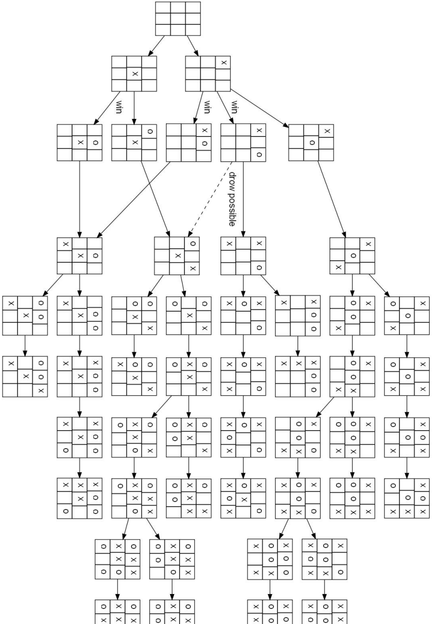
Machine Learning in Practice
#7-2: Search Heuristics for Combinatorial Games

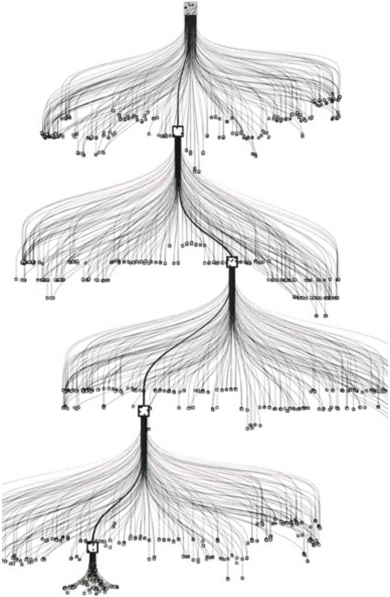
Sang-Hyun Yoon

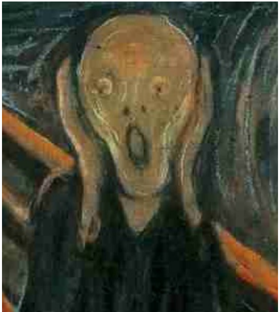
Summer 2019

Game Tree of Tic-Tac-Toe: $9! \approx 3.6 \cdot 10^5$



Game Tree of Go: $250^{150} \approx 5 \cdot 10^{359}$





Outline

1. Minimax/α-β Search

2. Monte Carlo Tree Search

Recall Full Search for Combinatorial Games

```
AcanWin(state):
    if the game ends in this state with B winning:
        return False                        # depth of game tree

    for each of the A's possible move:      # width of game tree
        state' = new state reached by the move
        if not BcanWin(state'):
          return True  # state => state' guarantees a win for A
    return False  # whichever move A selects, B wins

BcanWin(state):
    if the game ends in this state with A winning:
        return False                        # depth of game tree

    for each of the B's possible move:      # width of game tree
        state' = new state reached by the move
        if not AcanWin(state'):
          return True  # state => state' guarantees a win for B
    return False  # whichever move B selects, A wins
```

Minimax Search: Cuts Search Depth

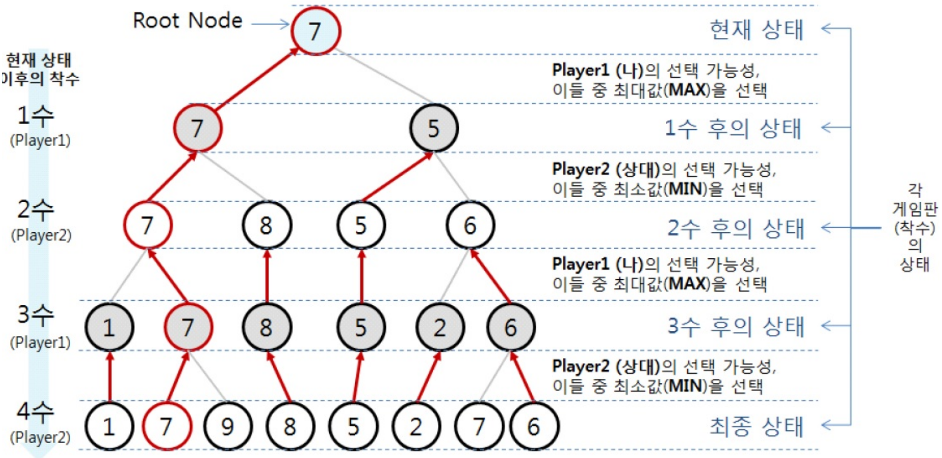
```
maxValue(state, depth):  # maximizing player (me)
    if depth >= threshold:
        return value(state)                # depth of game tree

    max_value = -∞
    for each of the A's possible move:      # width of game tree
        state' = new state reached by the move
        max_value = max(max_value, minValue(state', depth+1))
    return max_value

minValue(state, depth):  # minimizing player (opponent)
    if depth >= threshold:
        return value(state)                # depth of game tree

    min_value = ∞
    for each of the B's possible move:      # width of game tree
        state' = new state reached by the move
        min_value = min(min_value, maxValue(state', depth+1))
    return min_value
```

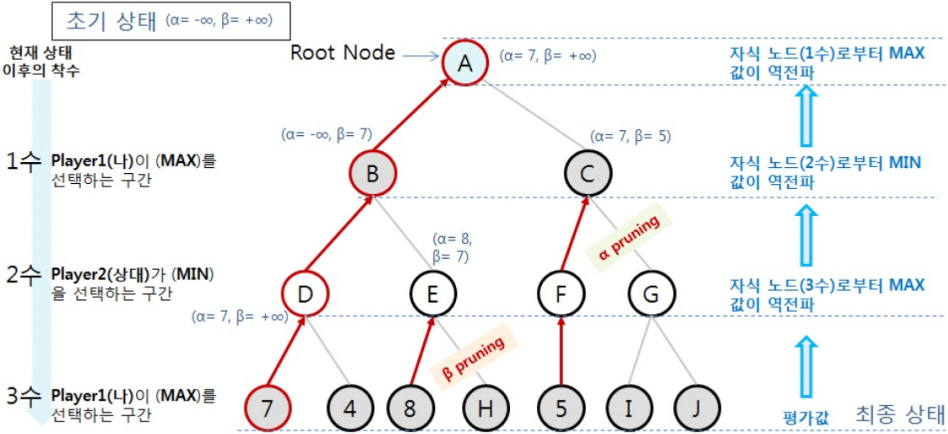
Minimax Search



- Root Node는 현재의 상태
- 회색노드는 MAX 값 선택 (Player1)
- 흰색노드는 MIN 값 선택 (player2)
- 맨 아래 노드의 숫자는 Player1(나)을 기준으로 한 게임의 평가 값을 의미
- 붉은색 화살표는 맨아래 노드의 평가 값이 상위노드로 반영되는 상태
- 붉은색 원은 최종적으로 선택된 서브트리

α-β Pruning: Faster Equivalence of Minimax Search

- Much of the minimax search can be eliminated
- Standard method for Chess (Deep Blue: α-β with depth 12)



- Root Node는 현재의 상태
- 회색노드는 MAX 값 선택 (Player1)
- 흰색노드는 MIN 값 선택 (player2)
- 맨 아래 노드의 숫자는 Player1(나)을 기준으로 한 게임의 평가 값을 의미
- 붉은색 화살표는 맨아래 노드의 평가 값이 상위노드로 반영되는 상태
- 붉은색 원은 최종적으로 선택된 서브트리

α-β Pruning: Faster Equivalence of Minimax Search

```
maxValue(state, alpha, beta, depth):  # maximizing player (me)
    if depth >= threshold:
        return value(state)

    for each of the A's possible move:
        state' = new state reached by the move
        alpha = max(alpha, minValue(state', alpha, beta, depth+1
        if beta <= alpha: break  # beta cut-off
    return max_value

minValue(state, alpha, beta, depth):  # minimizing player (oppon
    if depth >= threshold:
        return value(state)

    for each of the B's possible move:
        state' = new state reached by the move
        beta = min(beta, maxValue(state', alpha, beta, depth+1))
        if beta <= alpha: break  # alpha cut-off
    return min_value
```

α-β Pruning: Faster Equivalence of Minimax Search

Theorem

α-β pruning is equivalent to minimax search.

Proof: Try by yourself! (by induction on tree depth)

Value of Each State

- 체스의 경우 남은 말들의 가치를 합하여 value를 부여하면 그럭저럭 괜찮은 편 (폰:1, 나이트/비숍:3, 룩:5, 퀸:9)
- 장기의 경우도 마찬가지

4	4	4	10	4	10	4	4	4
4	10	17	11	8	11	17	10	4
6	12	13	17	13	17	13	12	6
7	22	13	21	14	21	13	22	7
4	14	13	17	18	17	13	14	4
4	12	15	16	17	16	15	12	4
6	8	12	9	12	9	12	8	6
7	6	8	9	6	9	8	6	7
-4	4	6	7	-6	7	6	4	-4
0	-5	4	7	4	7	4	-5	0

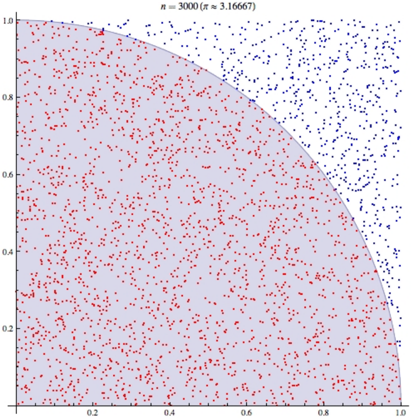
- 바둑의 경우?
  - 각 바둑판 상태의 value도 매기기 힘듬 (끝까지 가봐야함)
  - tree width가 체스/장기보다 훨씬 커서 α-β로는 불가

Outline

1. Minimax/α-β Search
2. **Monte Carlo Tree Search**

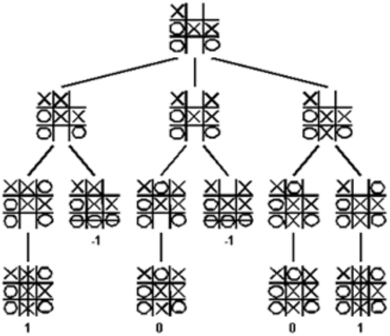
Monte Carlo Methods

https://en.wikipedia.org/wiki/Monte_Carlo_method



- 랜덤 샘플에 대해서만 실험을 수행해도 모든 샘플에 대해서 실험한 것과 그리 큰 차이가 나지 않을 수 있음
- 랜덤 샘플 수를 증가시키면 정확한 실험치에 수렴함

Monte Carlo Tree Search: Idea



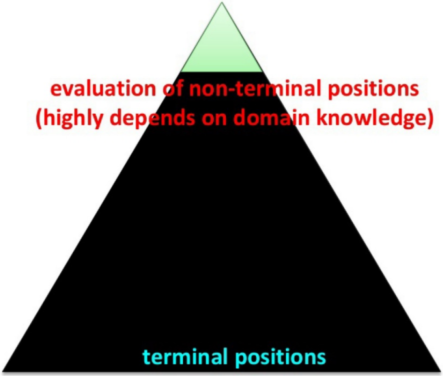
루트 노드 s에서 X가 선택할 다음 최적 수는?

1. 랜덤하게 끝까지 게임을 진행하는 실험을 많은 횟수 반복
2. s의 각 child s'의 value를 s'를 따라간 실험 경로에서의 leaf node의 reward의 평균으로 설정
3. Value가 가장 큰 child를 선택

이 과정을 좀 더 섬세하게 다듬으면 최적해에 가깝지 않을까?

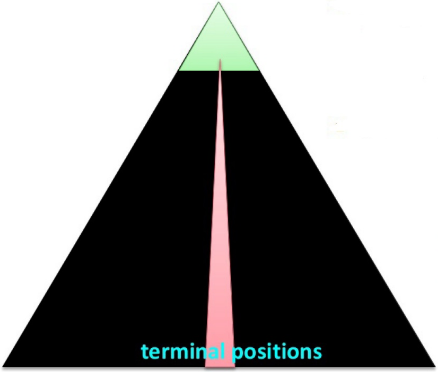
Search Space: Minimax Search vs. Monte Carlo Search

Minimax



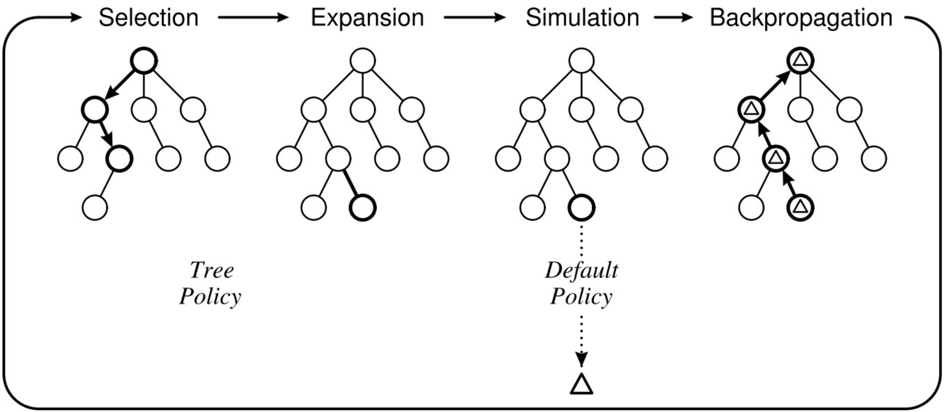
- 깊이를 제한
- 그 깊이까지는 모두 검색
- value가 적절하면 효과적

Monte Carlo



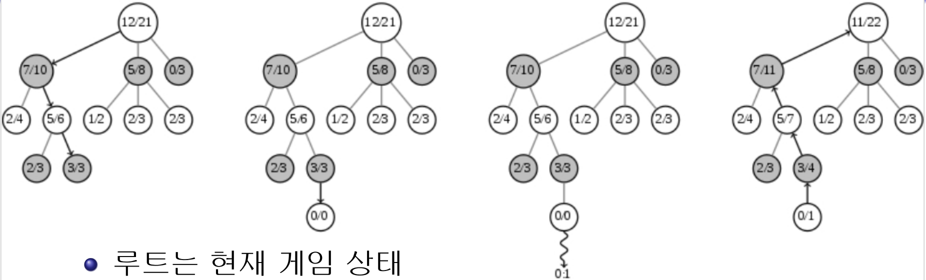
- 너비가 제한됨
- 그 너비까지는 끝까지 파고듬
- 랜덤 샘플링 잘하면 효과적

Monte Carlo Tree Search (MCTS): Generic Template

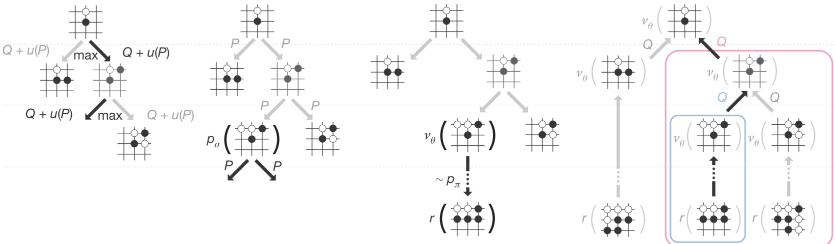


function MCTSSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD(v_0))

Monte Carlo Tree Search (MCTS): Example



- 루트는 현재 게임 상태
- 루트의 자식 노드 중 어디로 갈지를 결정하기 위해 랜덤 move를 선택해 많이 이기거나 많이 실험된 쪽으로 선택
  - 이길 확률이 높은 쪽으로 실험 횟수가 쏠리도록 설계됨



MCTS with UCT (Upper Confidence Bounds)

function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)

$$\textbf{return} \quad \underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

MCTS with UCT (Upper Confidence Bounds)

$$\underset{v' \in child(v)}{\operatorname{argmax}} \left(\underbrace{\frac{Q(v')}{N(v')}}_{\text{exploitation}} + c \underbrace{\sqrt{\frac{2 \ln N(v)}{N(v')}}}_{\text{exploration}} \right)$$

Let

- $N(v')$: 노드 v'를 지나가는 실험 횟수 ($N(v) = \sum_{v' \in child(v)} N(v')$)
- $Q(v')$: 노드 v'를 지나갔을 때 이긴 실험 횟수

Then

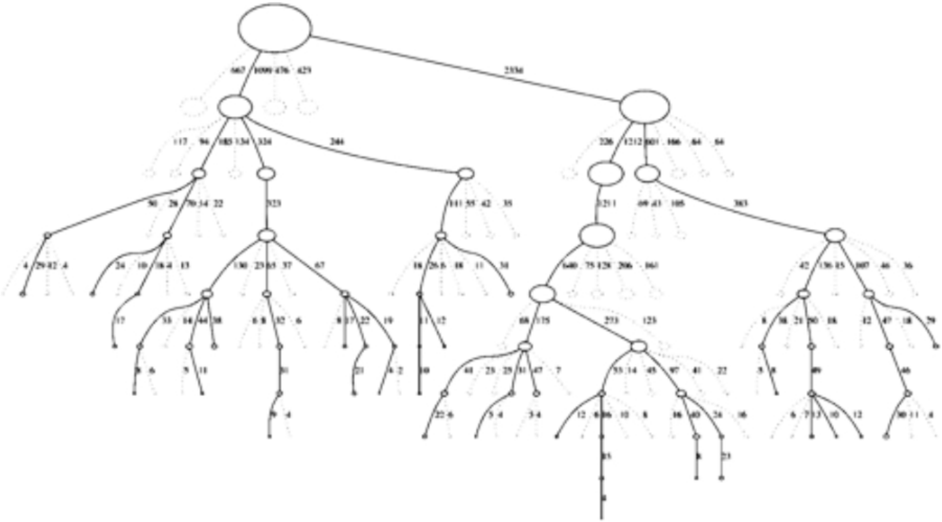
- $\frac{Q(v')}{N(v')}$: 노드 v'를 택했을 때 이길 확률
- $\sqrt{\frac{2 \ln N(v)}{N(v')}}$: 노드 v'쪽으로 실험이 적게 이루어질수록 커짐

이길 확률이 높은 쪽으로 더 자주 실험하되 (exploitation),
실험이 충분히 되지 않은 쪽으로도 실험을 유도 (exploration)

횟수를 무한히 늘리면 optimal strategy에 수렴함이 증명됨

Asymmetic Tree Growth

더 중요한 쪽으로 집중적으로 탐색



Example Code for “Simple-Go”

- constant.py: 광범위하게 사용되는 간단한 상수/함수
- graphics/graphics_lib.py: 그래픽 관련
- board.py: 바둑 규칙 ($C, F_{흑}, F_{백}, m_{흑}, m_{백}$)
- tree_search_MCTS.py: MCTS
  - MCTS survey 논문 인쇄본의 자세한 설명도 참고
  - 현재 코드는 100번씩 실험. 한수에 1초 이내. 성능 매우 나쁨
- tree_search_minimax.py: minimax/α-β 탐색
- play_auto.py: 컴퓨터(MCTS/minimax/α-β)간 경기
  - MCTS/minimax/α-β를 호출하는 부분 주목
- play_human.py: 컴퓨터와 사람간 경기

play_human.py나 play_auto.py를 수행하면 됨

Simplifed Rules of Our 5×5-Go

board.py 참고

- 5×5 격자 (# states $= 3^{5^2} \approx 8.47 \cdot 10^{11}$)
- 영역 크기 계산시 잡은 돌의 갯수는 무시
- Passing은 valid move가 없을 경우에만 허용됨
- 흑/백 모두 valid move가 없을 때까지 계속 play
  - 사석은 모두 capture되어야 함
  - 공배(neutral space)는 모두 채워져야 함
  - 크기가 1보다 큰 영역은 채워져서 1인 영역들만 남아야 함
- 자신의 크기 1인 영역으로 들어갈 수는 없는데, 예외적으로 그렇게 함으로써 자신의 돌을 보호할 수 있을 때만 허용
  - 즉, 상대가 그 영역으로 들어가면 자신의 돌을 잡을 때
- 백의 덤은 1.5집

play_human.py로 직접 두거나 play_auto.py로 컴퓨터간 경기를 관찰하면 규칙 파악에 도움이 됨

Homework: α-β Pruning (`tree_search_minimax.py`)

- `Minimax` class의 `.__maxValue`/`.__minValue`를 참고하여 `AlphaBeta` class의 `.__maxValue`/`.__minValue`를 구현
  - 슬라이드의 minimax/α-β pseudo-code를 비교하면서