KAIST Include 동아리 스터디

AlphaGo와 AlphaGo Zero를 만들며 익히는 딥러닝 및 강화학습

Chris Ohk
utilForever@gmail.com

들어가며

- 트리 탐색 알고리즘 : 최적의 결과 하나를 찾기 위해 수많은 가능 경로를 돌아다니는 방법
  - 미니맥스(Minimax) 탐색 알고리즘 : 두 플레이어가 서로 자기 차례를 진행하는 동안 관점을 바꿔서 보는 방식. 완벽한 진행 순서를 찾을 수 있지만 복잡한 게임에 적용하면 매우 느려진다는 단점이 있다.
  - 가지치기(Pruning) : 트리의 일부분을 잘라내서 탐색 속도를 높이는 방법. 효과적으로 적용하려면 해당 문제의 실질적인 지식을 코드에 넣어야 한다.
  - 몬테카를로 트리 탐색(Monte-Carlo Tree Search, MCTS) : 해당 분야 관련 지식이 없이도 좋은 결과를 찾아주는 난수 탐색 알고리즘

게임 분류

- 트리 탐색 알고리즘 = 주로 턴제 게임과 관련이 있다.
  - 예를 들어, 보드 게임, 카드 게임 등이 있다.
  - 반면 농구, 월드 오브 워크래프트 등의 게임에는 도움이 되지 않는다.
- 보드 게임과 카드 게임은 두 가지 특성에 따라 분류할 수 있다.
  - 결정론적과 비결정론적
    - 결정론적 : 게임 진행이 플레이어의 판단에 의해서만 결정된다.
    - 비결정론적 : 주사위 던지기나 카드 섞기 등 임의의 요소가 들어간다.
  - 완전한 정보와 숨겨진 정보
    - 완전한 정보 : 양쪽 플레이어가 언제나 게임의 전체 상황을 볼 수 있다.
    - 숨겨진 정보 : 각 플레이어가 게임 전체 상황의 일부만 볼 수 있다.

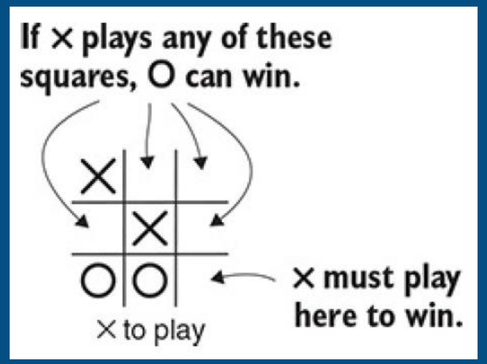
	결정론적	비결정론적
완전한 정보	바둑, 체스	백개먼
숨겨진 정보	배틀쉽, 스트라테고	포커, 스크래블

게임 분류

- 여기서는 결정론적이고 완전한 정보가 주어지는 게임을 살펴본다.
  - 매 차례에서 이론적으로 최고의 수가 존재한다.
  - 수를 선택하기 전에 상대방이 그 다음에 선택할 수를 알 수 있고,
    그 후에 어떻게 할 지도 예상할 수 있으며, 이런 식으로 종국까지 생각할 수 있다.
  - 이론적으로는 첫 수 이후에 게임이 종료될 때까지의 계획을 세울 수 있다.
  - 미니맥스 알고리즘은 완벽한 게임을 만들기 위해 이 과정을 정확히 따른다.
- 하지만 실제로 체스나 바둑은 가능한 수가 엄청나게 많다.
  여기서는 틱택토를 통해 미니맥스 알고리즘을 살펴보고자 한다.

미니맥스 탐색을 사용한 상대 수 예측

- **미니맥스(Minimax)** = **최소화(Minimizing)** + **최대화(Maximizing)**
  - 상대방이 여러분의 점수를 최소화하려고 하는 상황에서 점수를 최대하는 것
- 다음 그림을 살펴 보자. (X를 두는 플레이어 차례)



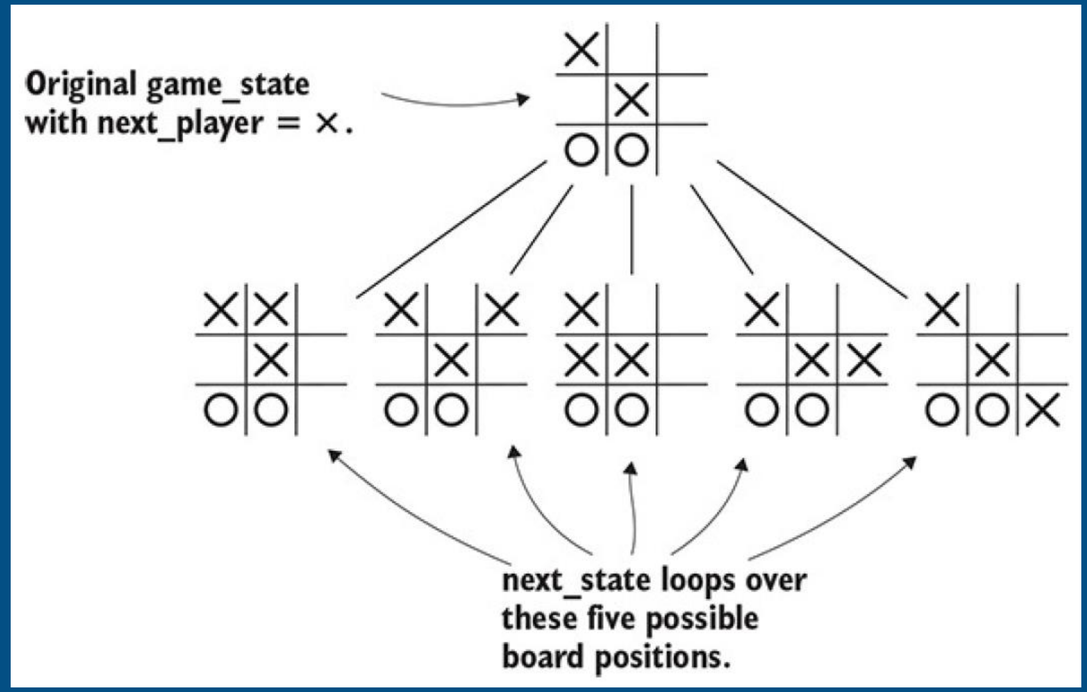
미니맥스 탐색을 사용한 상대 수 예측

- 바로 이길 수를 찾는 함수

```
def find_winning_move(game_state, next_player):
  for candidate_move in game_state.legal_moves(next_player):
    next_state = game_state.apply_move(candidate_move)
    if next_state.is_over() and next_state.winner == next_player:
      return candidate_move
  return None
```

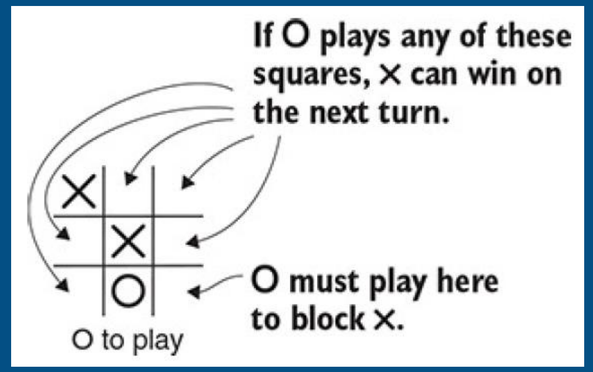
미니맥스 탐색을 사용한 상대 수 예측

- 게임 트리(Game Tree)



미니맥스 탐색을 사용한 상대 수 예측

- 조금 뒤로 돌아가서 O를 두는 플레이어 입장에서 생각해 보자.



- X 플레이어 입장의 '바로 이길 수를 찾자'와 O 플레이어 입장의 '상대가 이길 수 있는 수를 두지 말자'는 필연적으로 부딪치게 된다.

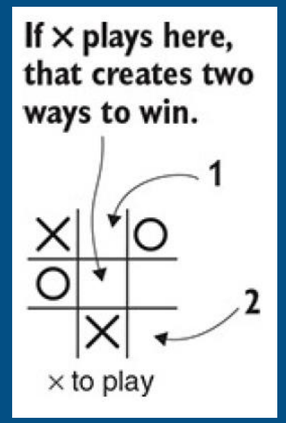
미니맥스 탐색을 사용한 상대 수 예측

- 상대가 필승점에 두는 것을 막는 함수

```
def eliminate_losing_moves(game_state, next_player):
  opponent = next_player.other()
  possible_moves = []
  for candidate_move in game_state.legal_moves(next_player):
    next_state = game_state.apply_move(candidate_move)
    opponent_winning_move = find_winning_move(next_state, opponent)
    if opponent_winning_move is None:
      possible_moves.append(candidate_move)
  return possible_moves
```

미니맥스 탐색을 사용한 상대 수 예측

- 이제 상대가 필승점에 두는 걸 반드시 막아야 한다는 걸 깨달았을 거다.
  따라서 상대방 역시 여러분과 동일한 방식을 쓸 거라고 가정해야 한다.
- 이제 이기려면 어떻게 해야 할까?



미니맥스 탐색을 사용한 상대 수 예측

- 이길 수 있는 2회 연속 수를 찾는 함수

```
def find_two_step_win(game_state, next_player):
  opponent = next_player.other()
  for candidate_move in game_state.legal_moves(next_player):
    next_state = game_state.apply_move(candidate_move)
    good_responses = eliminate_losing_moves(next_state, opponent)
    if not good_responses:
      return candidate_move
  return None
```

미니맥스 탐색을 사용한 상대 수 예측

- 미니맥스 탐색을 사용한 일반적인 전략
  1. 다음 수에서 이길 수 있는지 확인하자. 그렇다면 그 수를 둔다.
  2. 아니라면 다음 차례에 상대방이 이길지 살펴보자. 그렇다면 그 수를 막는다.
  3. 아니라면 두 수 뒤 이길 수 있는지 확인한다. 그렇다면 그렇게 둔다.
  4. 아니라면 상대방이 자기 차례로부터 두 수 후에 이길 수 있는지 확인한다.

미니맥스 예제 – 틱택토 풀기

- 게임 결과를 나타내는 enum 변수

```
class GameResult(enum.Enum):
  loss = 1
  draw = 2
  win = 3
```

미니맥스 예제 – 틱택토 풀기

- 미니맥스 탐색으로 구현한 게임 실행 에이전트

```
class MinimaxAgent(Agent):
  def select_move(self, game_state):
    winning_moves = []
    draw_moves = []
    losing_moves = []
    for possible_move in game_state.legal_moves():
      next_state = game_state.apply_move(possible_move)
      opponent_best_outcome = best_result(next_state)
      our_best_outcome = reverse_game_result(opponent_best_outcome)
      if our_best_outcome == GameResult.win:
        winning_moves.append(possible_move)
      elif our_best_outcome == GameResult.draw:
        draw_moves.append(possible_move)
      else:
        losing_moves.append(possible_move)
    if winning_moves:
      return random.choice(winning_moves)
    if draw_moves:
      return random.choice(draw_moves)
    return random.choice(losing_moves)
```

미니맥스 예제 – 틱택토 풀기

- 미니맥스 탐색 알고리즘 첫 단계

```python
def best_result(game_state):
  if game_state.is_over():
    if game_state.winner() == game_state.next_player:
      return GameResult.win
    elif game_state.winner() is None:
      return GameResult.draw
    else:
      return GameResult.loss
```

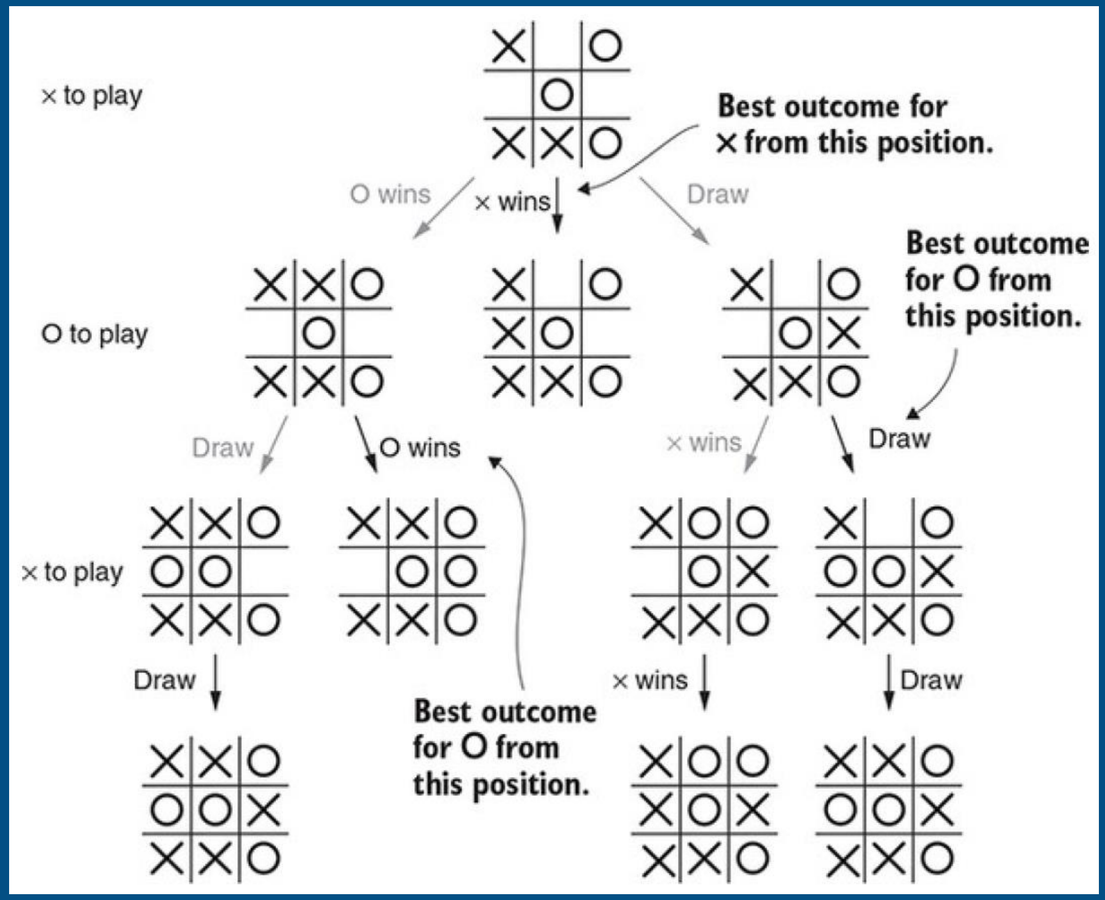
미니맥스 예제 – 틱택토 풀기

- 미니맥스 탐색 구현

```
best_result_so_far = GameResult.loss
for candidate_move in game_state.legal_moves():
  next_state = game_state.apply_move(candidate_move)
  opponent_best_result = best_result(next_state)
  our_result = reverse_game_result(opponent_best_result)
  if our_result.value > best_result_so_far.value:
    best_result_so_far = our_result
return best_result_so_far
```

미니맥스 예제 – 틱택토 풀기

- 틱택토 게임 트리



가지치기를 통한 탐색 공간 축소

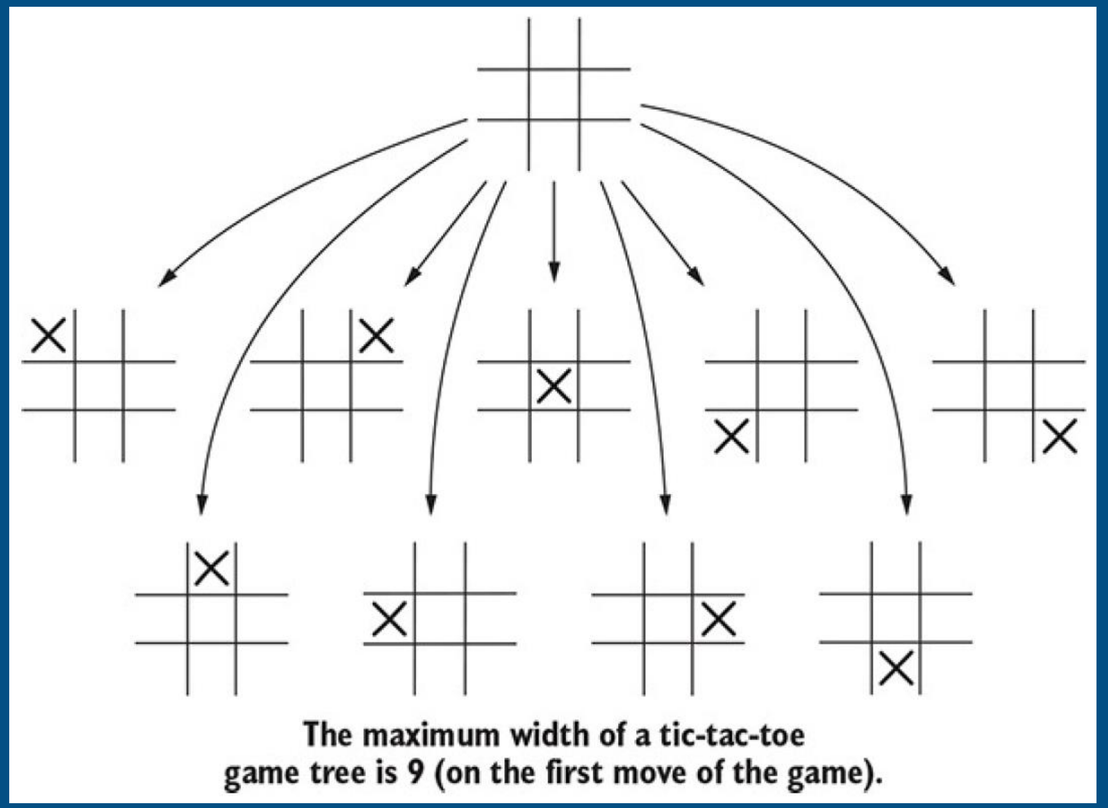
- 가능한 틱택토 게임 수의 조합 : $\approx 3 \times 10^5$ → 식은 죽 먹기
  가능한 체커 게임 수의 조합 : $\approx 5 \times 10^{20}$ → 수 년
  가능한 바둑 게임 수의 조합 : …
- 복잡한 게임을 할 때 트리 탐색을 사용하려면 트리의 일부를 제거하는 방법을 사용해야 한다. 트리의 어느 부분을 건너뛸 건지 결정하는 방식을 **가지치기(Pruning)**라고 한다.

가지치기를 통한 탐색 공간 축소

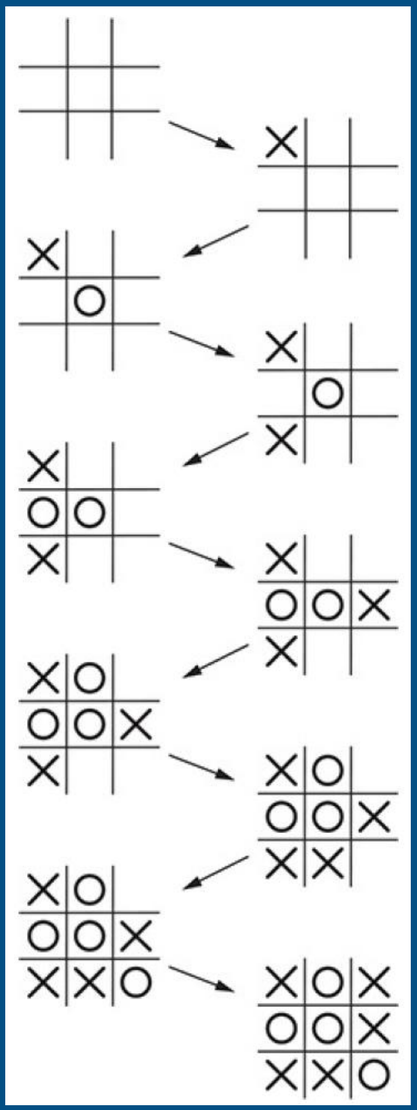
- 게임 트리는 폭과 깊이를 갖는 2차원 구조다.
  - **폭(Width)**은 주어진 판에서 가능한 수의 개수다.
  - **깊이(Depth)**는 현재 판에서 최종 게임 상태까지의 차례 개수다.
- 트리 크기는 특정 게임에 대해 일반적인 폭과 깊이를 고려해서 추정한다.
  - 게임 트리의 기보 수는 대충 W^d라는 식으로 구한다. (W는 평균 폭, d는 평균 깊이다.)
    - 체스의 경우 각 수에 대해 약 30개의 옵션이 있고, 게임은 보통 80수 정도를 거쳐야 끝난다.
      이 경우 게임 트리의 크기는 대략 $30^{80} \approx 10^{118}$ 위치와 같다.
    - 바둑의 경우 보통 매 차례 250개의 가능한 수가 있고, 게임은 150차례에 걸쳐 진행된다.
      이 경우 게임 트리의 크기는 $250^{150} \approx 10^{359}$ 위치 정도다.

가지치기를 통한 탐색 공간 축소

- 틱택토 게임 트리의 폭/깊이

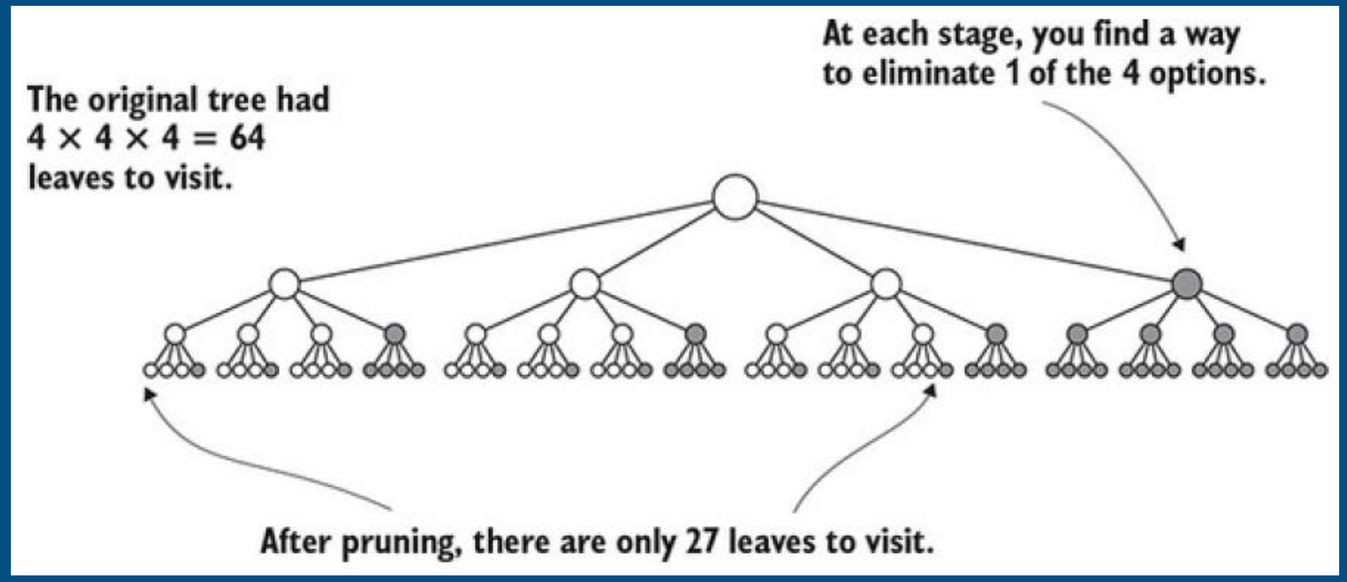


The maximum width of a tic-tac-toe game tree is 9 (on the first move of the game).



가지치기를 통한 탐색 공간 축소

- 가지치기 기법
  - 위치 평가 함수 : 탐색 깊이를 줄이기 위한 기법
  - 알파-베타 가지치기 : 탐색 폭을 줄이기 위한 기법
- 가지치기의 효과



위치 평가를 통한 탐색 깊이 축소

- 사람은 보통 게임 중반부쯤에서 이길 것 같은 쪽을 판단할 수 있다.
- 만약 컴퓨터 프로그램에서도 이를 판단할 수 있다면
  탐색해야 할 깊이를 줄일 수 있을 것이다. → **위치 평가 함수**
- 위치 평가 함수는 게임에 대한 지식을 사용해서 만들게 된다.
  - 체커 : 일반 말을 1점으로 세고, 킹의 경우 2점을 더한다.
    본인의 점수를 센 후 상대방의 점수를 뺀다.
  - 체스 : 각 폰은 1점, 비숍이나 나이트는 3점, 룩은 5점, 퀸은 9점으로 센다.
    본인의 점수를 센 후 상대방의 점수를 뺀다.

위치 평가를 통한 탐색 깊이 축소

- 바둑은 잡은 돌 수를 더하고 잡힌 돌 수를 빼는 방식을 사용할 수 있다.
  (물론 효과적인 평가 함수가 아니라는 건 금방 알 수 있다.)
  - 바둑에서는 돌을 잡을 것처럼 위협하는 게 실제로 잡는 것보다 훨씬 중요하다.
  - 경기 상태의 숨은 의미까지 정확히 읽어내는 평가 함수를 만드는 건 상당히 어렵다.
- 평가 함수를 선택한 후에는 **깊이 가지치기 (Depth Pruning)**를 구현할 수 있다.

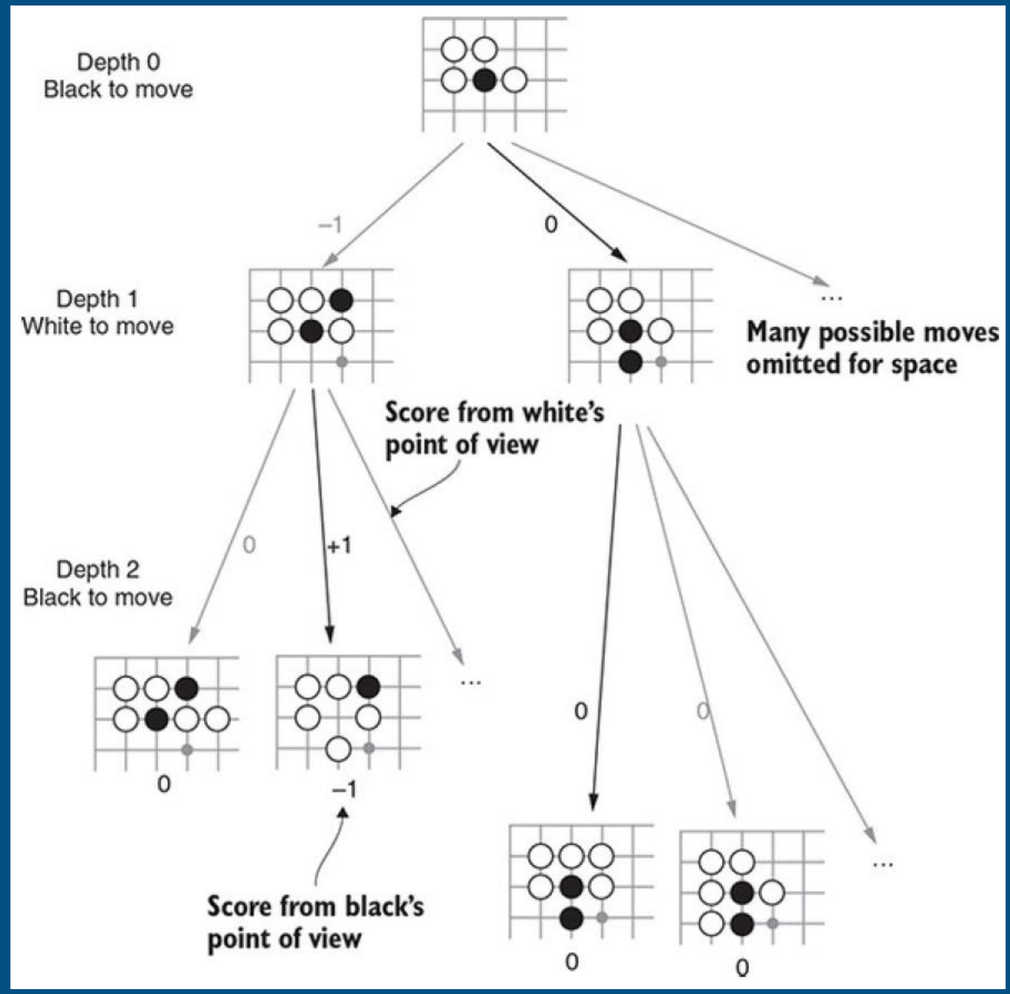
위치 평가를 통한 탐색 깊이 축소

- 고도로 단순화된 직관적 바둑판 평가

```
def capture_diff(game_state):
  black_stones = 0
  white_stones = 0
  for r in range(1, game_state.board.num_rows + 1):
    for c in range(1, game_state.board.num_cols + 1):
      p = gotypes.Point(r, c)
      color = game_state.board.get(p)
      if color == gotypes.Player.black:
        black_stones += 1
      elif color == gotypes.Player.white:
        white_stones += 1
  diff = black_stones - white_stones
  if game_state.next_player == gotypes.Player.black:
    return diff
  return -1 * diff
```

위치 평가를 통한 탐색 깊이 축소

- 깊이 가지치기가 된 게임 트리의 일부



위치 평가를 통한 탐색 깊이 축소

- 깊이 가지치기용 미니맥스 탐색

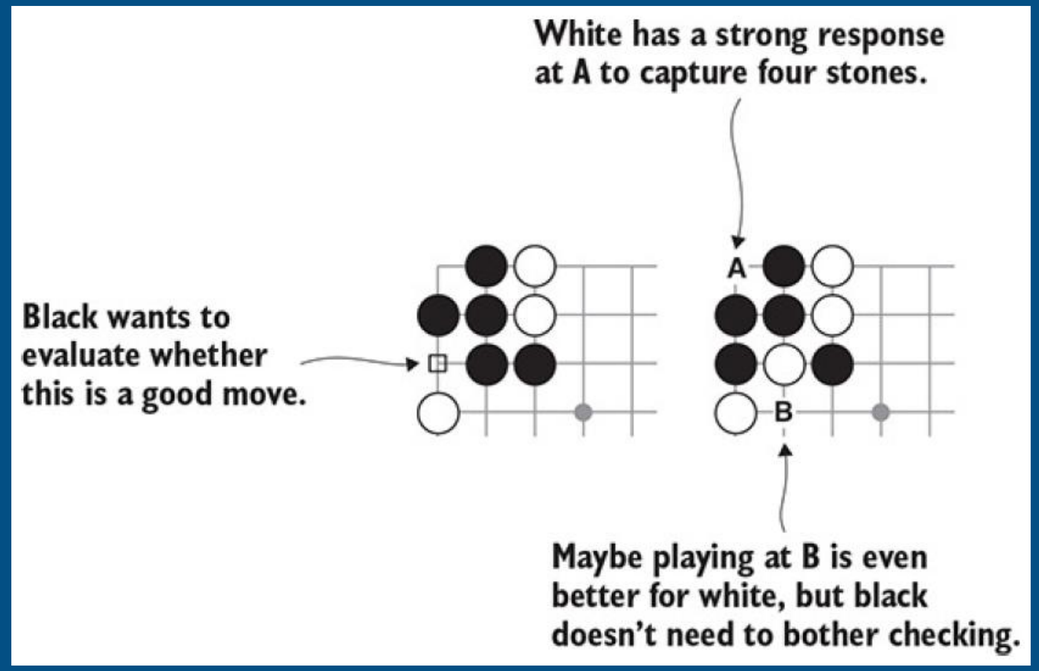
```
def best_result(game_state, max_depth, eval_fn):
  if game_state.is_over():
    if game_state.winner() == game_state.next_player:
      return MAX_SCORE
    else:
      return MIN_SCORE

  if max_depth == 0:
    return eval_fn(game_state)

best_so_far = MIN_SCORE
for candidate_move in game_state.legal_moves():
  next_state = game_state.apply_move(candidate_move)
  opponent_best_result = best_result(next_state, max_depth - 1, eval_fn)
  our_result = -1 * opponent_best_result
  if our_result > best_so_far:
    best_so_far = our_result

return best_so_far
```

위치 평가를 통한 탐색 깊이 축소

- 이전에 봤던 미니맥스 탐색 코드와 다른 점
  - 승/패/무승부를 반환하는 대신 기보 평가 함수에서 나온 값을 반환한다. 여기서는 다음 차례 플레이어 관점에서의 점수를 반환하기로 한다. 점수가 크면 다음 차례 선수가 이길 거 같다는 의미다. 상대방의 관점에서 기보를 평가하고자 한다면 본인 관점에서의 점수에 -1을 곱해서 뒤집으면 된다.
  - max_depth 파라미터는 앞으로 탐색할 수의 개수를 조정한다. 매 차례마다 이 값에서 1을 뺀다. max_depth가 0이 되면 탐색을 멈추고 기보 평가 함수를 호출한다.

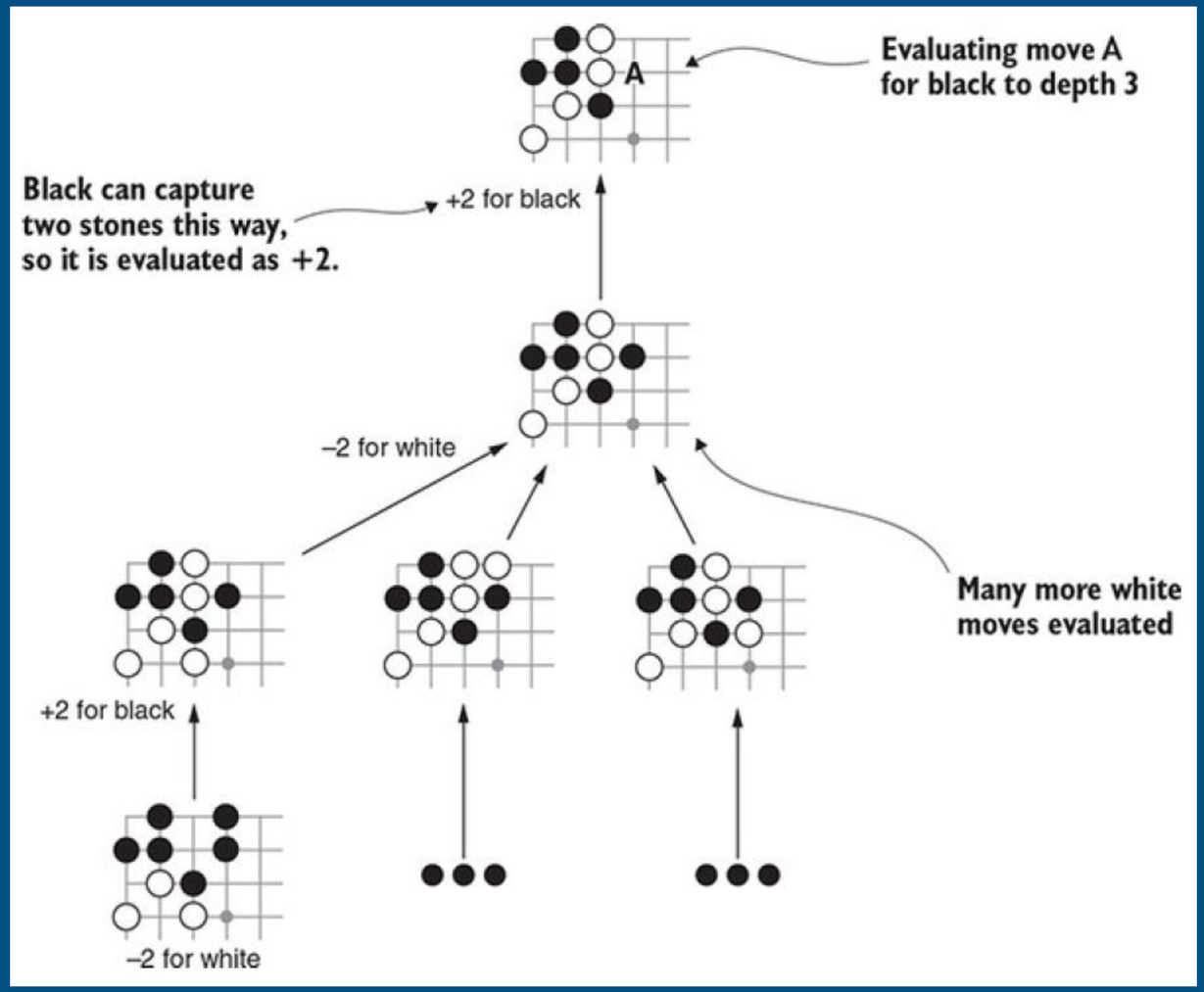
알파-베타 가지치기를 통한 폭 축소

- 알파-베타 가지치기에 내재된 개념



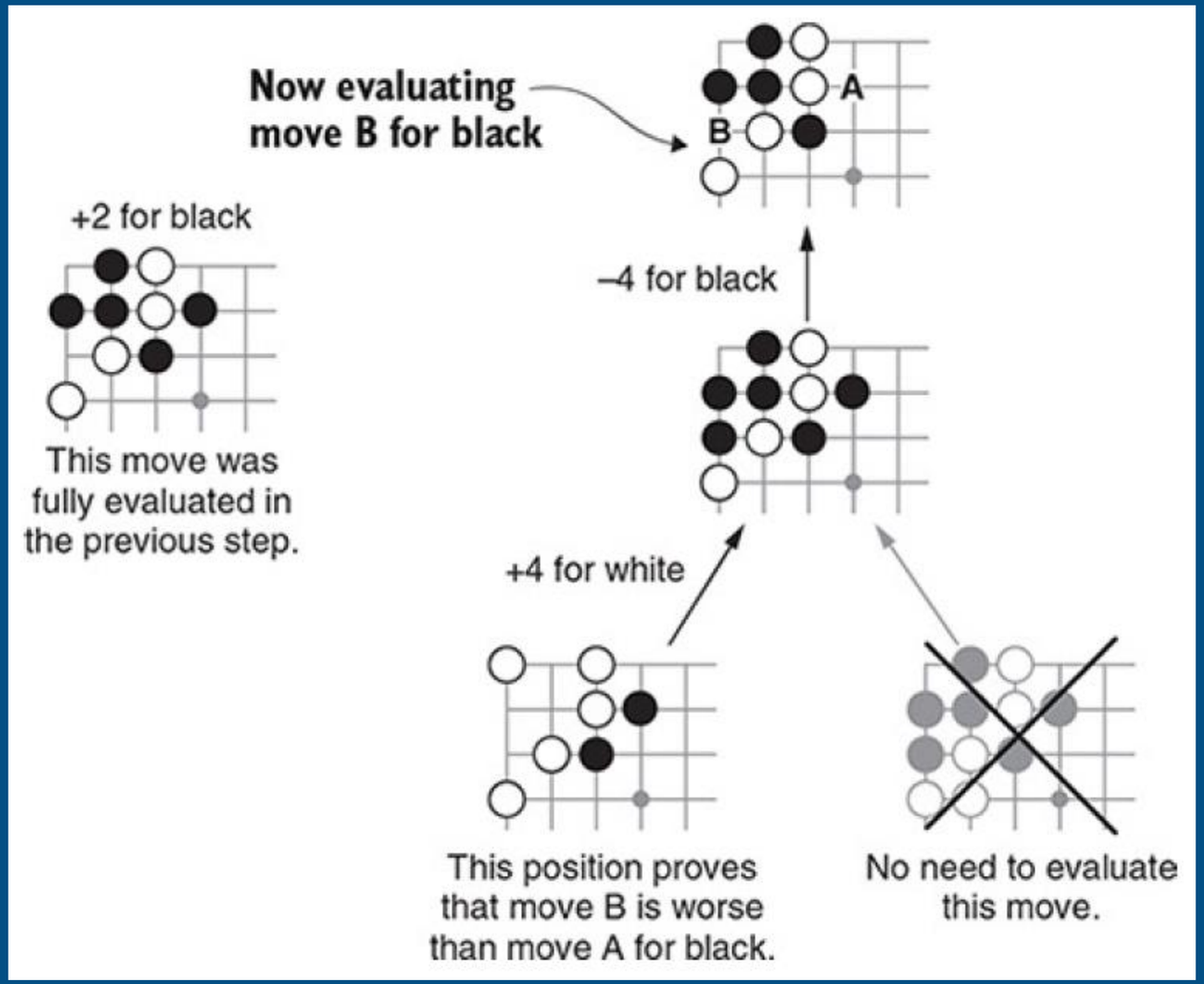
알파-베타 가지치기를 통한 폭 축소

- 알파-베타 가지치기 1단계



알파-베타 가지치기를 통한 폭 축소

- 알파-베타 가지치기 2단계



알파-베타 가지치기를 통한 폭 축소

- 가지 평가의 중지 여부 확인

```
def alpha_beta_result(game_state, max_depth, best_black, best_white, eval_fn):
...
    if game_state.next_player == Player.white:
      if best_so_far > best_white:
        best_white = best_so_far
      outcome_for_black = -1 * best_so_far
      if outcome_for_black < best_black:
        return best_so_far
```

알파-베타 가지치기를 통한 폭 축소

- 알바-베타 가지치기 전체 구현

```python
def alpha_beta_result(game_state, max_depth, best_black, best_white, eval_fn):
  if game_state.is_over():
    if game_state.winner() == game_state.next_player:
      return MAX_SCORE
    else:
      return MIN_SCORE

  if max_depth == 0:
    return eval_fn(game_state)

  best_so_far = MIN_SCORE
  for candidate_move in game_state.legal_moves():
    next_state = game_state.apply_move(candidate_move)
    opponent_best_result = alpha_beta_result(
      next_state, max_depth - 1, best_black, best_white, eval_fn)
    our_result = -1 * opponent_best_result
```

알파-베타 가지치기를 통한 폭 축소

- 알바-베타 가지치기 전체 구현

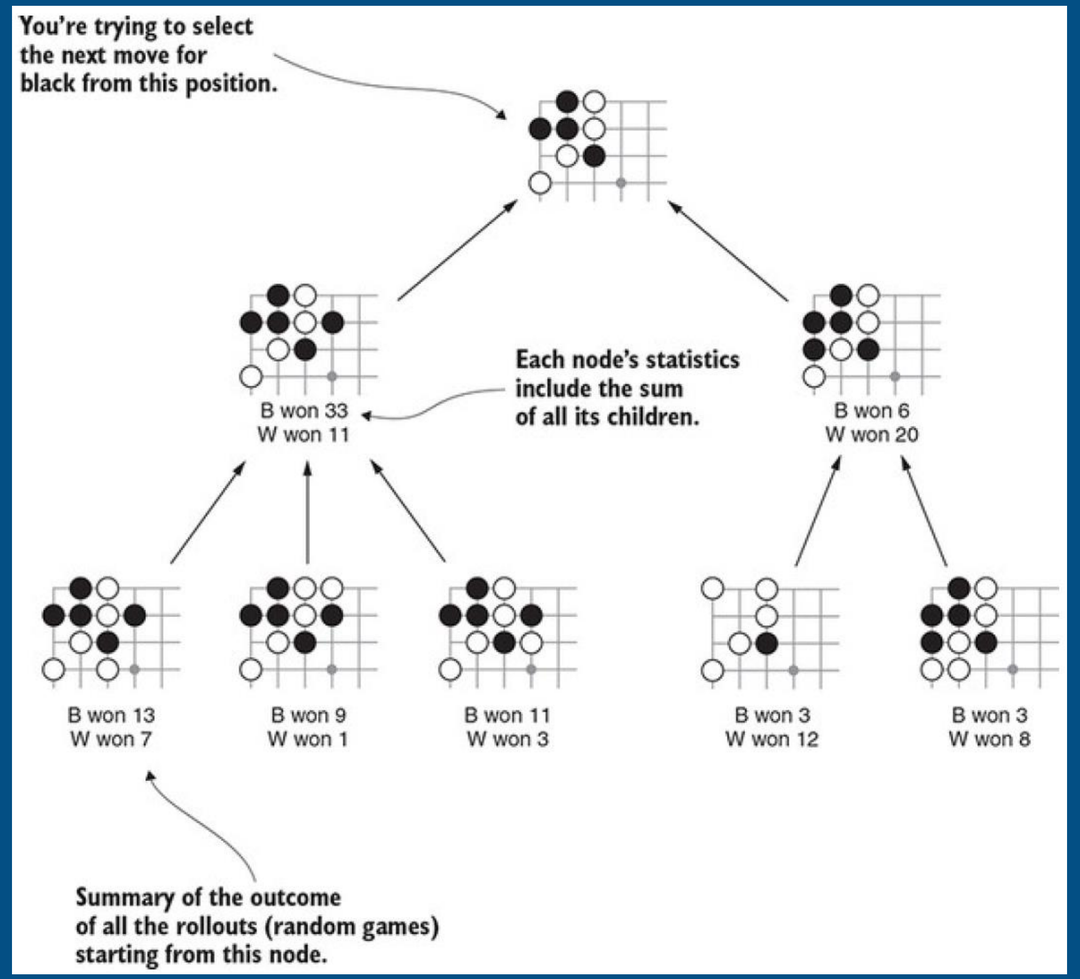
```
    if our_result > best_so_far:
      best_so_far = our_result
    if game_state.next_player == Player.white:
      if best_so_far > best_white:
        best_white = best_so_far
      outcome_for_black = -1 * best_so_far
      if outcome_for_black < best_black:
        return best_so_far
    elif game_state.next_player == Player.black:
      if best_so_far > best_black:
        best_black = best_so_far
        outcome_for_white = -1 * best_so_far
      if outcome_for_white < best_white:
        return best_so_far
```

몬테카를로 트리 탐색을 이용한 평가

- 알파-베타 가지치기에서 고려해야 할 위치의 수를 줄이기 위해 위치 평가 함수를 사용했다. 하지만 바둑에서 위치 평가는 매우 어렵다.
- **몬테카를로 트리 탐색(Monte-Carlo Tree Search; MCTS)**은 경기에 대한 어떤 전략 관련 지식 없이도 경기 상태를 평가할 수 있게 해준다.
  - 게임별로 특화된 직관 대신 임의의 게임을 시뮬레이션해 위치가 얼마가 좋은지 평가한다.
  - 임의의 게임 방식 중 하나를 **롤아웃(Rollout)** 또는 **플레이아웃(Playout)**이라고 한다.

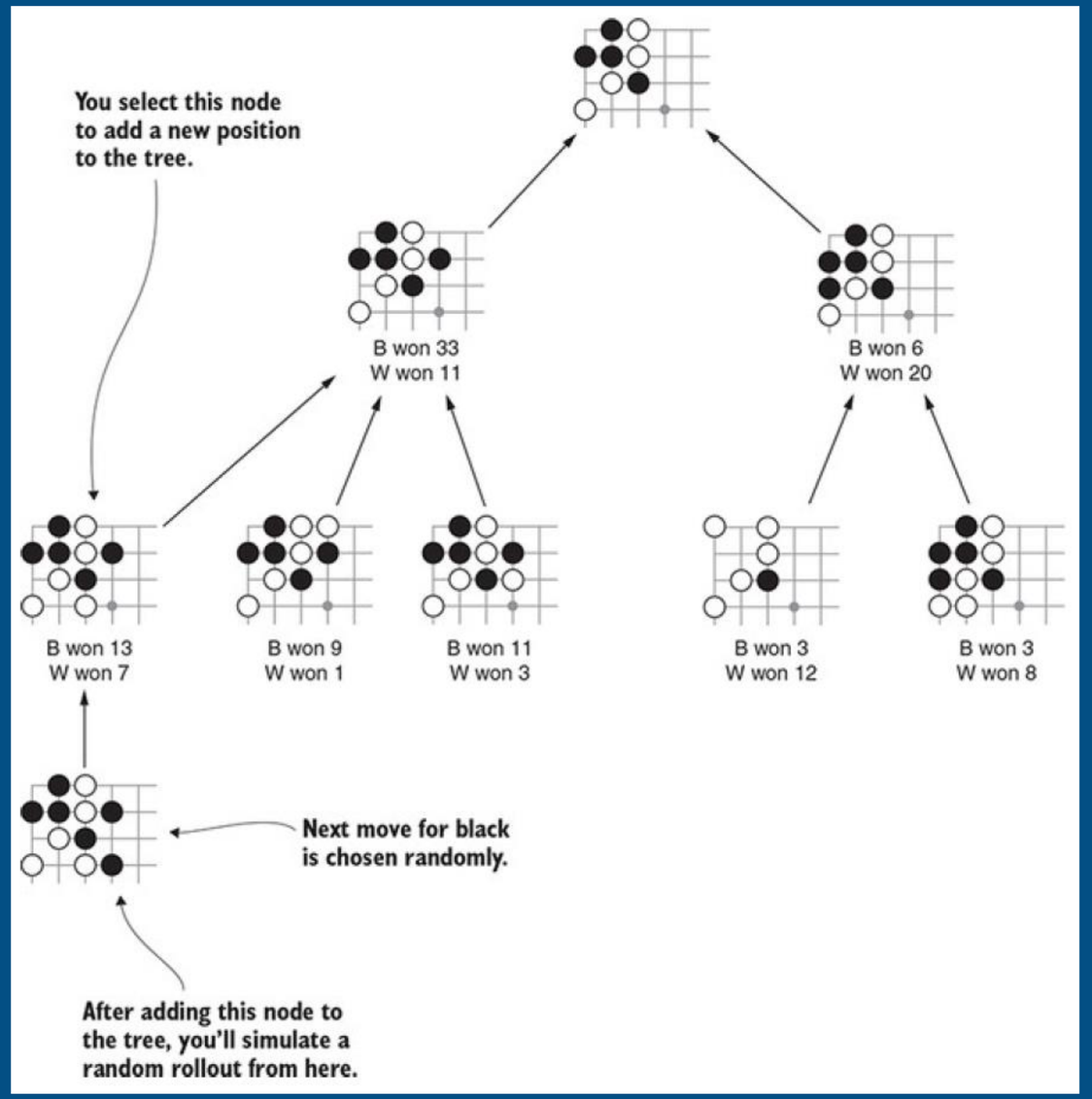
몬테카를로 트리 탐색을 이용한 평가

- MCTS 알고리즘의 세 단계
  1. MCTS 트리에 새로운 바둑판 위치를 추가한다.
  2. 그 위치에서 임의의 경기를 시뮬레이션한다.
  3. 임의의 경기 결과로 트리 통계를 갱신한다.



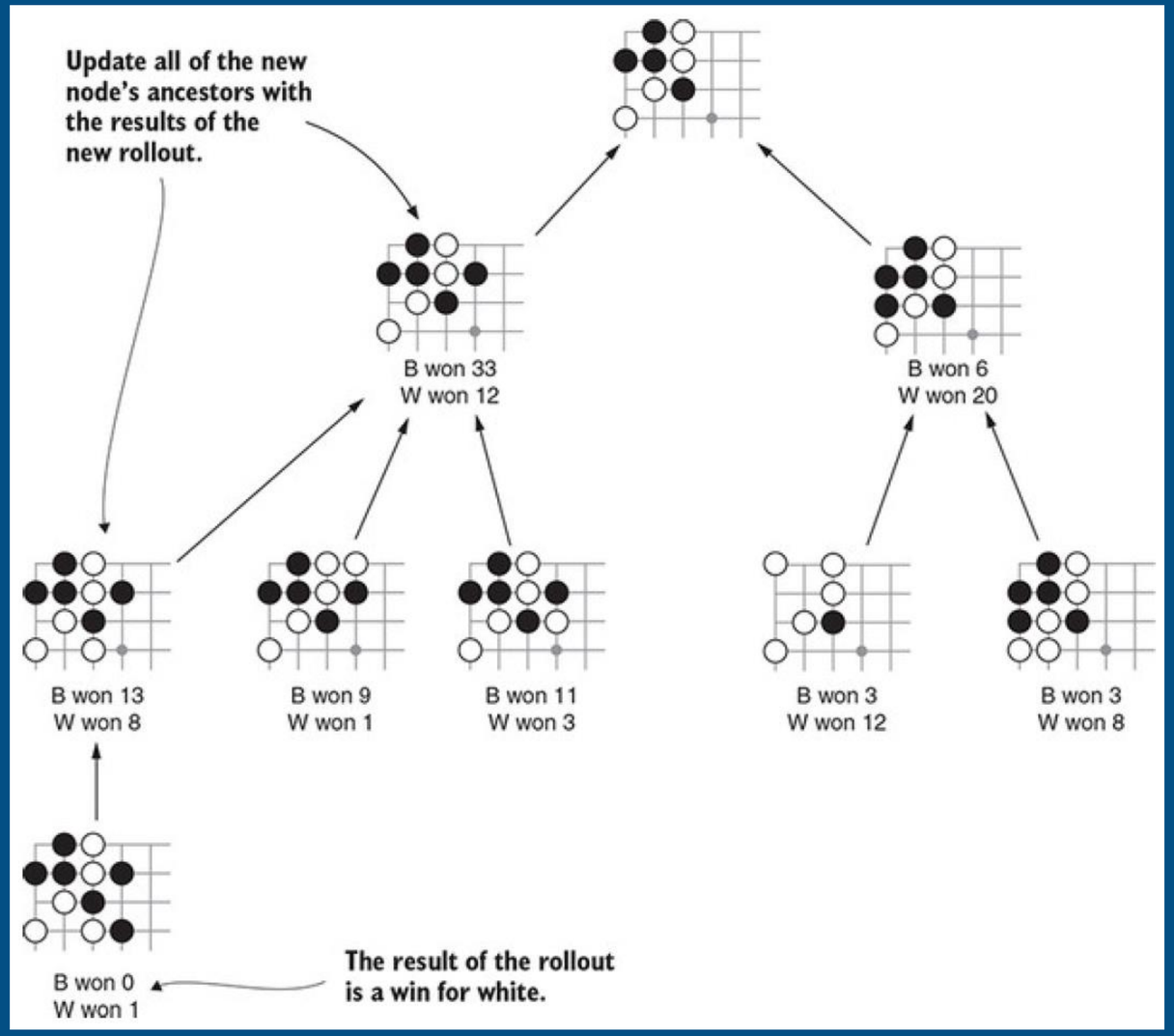
몬테카를로 트리 탐색을 이용한 평가

- MCTS 트리에 새 노드 추가
  - 임의로 다음 수를 고른다.
  - 새 바둑판 위치를 계산한다.
  - 노드를 트리에 추가한다.



몬테카를로 트리 탐색을 이용한 평가

- 새 롤아웃 후 MCTS 갱신
  - 트리의 새 노드는 임의 경기의 시작점.
  - 이제 나머지 경기를 시뮬레이션하고, 경기 종료 때까지 매 순번마다 가능한 수 중 하나를 선택한다.
  - 그리고 점수를 합산하고 승자를 찾는다.
  - 롤아웃 결과를 새 노드에 추가한다.
  - 그리고 모든 노드의 조상까지 올라가면서 새 롤아웃 결과를 더한다.



Python으로 MCTS 구현하기

- MCTS 트리 구현을 위한 자료 구조

```python
class MCTSNode(object):
  def __init__(self, game_state, parent=None, move=None):
    self.game_state = game_state
    self.parent = parent
    self.move = move
    self.win_counts = {
      Player.black: 0,
      Player.white: 0,
    }
    self.num_rollouts = 0
    self.children = []
    self.unvisited_moves = game_state.legal_moves()
```

Python으로 MCTS 구현하기

- MCTS 트리 구현을 위한 자료 구조
  - game_state : 트리의 이 노드에서 경기의 현재 상태 (기보와 다음 선수)
  - parent : 현재 위치에서의 부모 MCTSNode.
    트리의 시작을 나타내고자 할 때는 parent를 None으로 설정하면 된다.
  - move : 이 노드에서의 마지막 수
  - children : 트리의 모든 자식 노드 리스트
  - win_counts, num_rollouts : 이 노드에서 시작한 롤아웃 결과 통계
  - unvisited_moves : 아직 트리에 추가하지 않은 현재 위치에서 가능한 수 리스트.
    트리에 새 노드를 추가할 때마다 unvisited_moves에서 하나 가져와서 이에 대해
    새로운 MCTSNode를 생성하고 이를 children 리스트에 추가한다.

Python으로 MCTS 구현하기

- MCTS 트리의 노드를 갱신하는 메소드

```
def add_random_child(self):
  index = random.randint(0, len(self.unvisited_moves) - 1)
  new_move = self.unvisited_moves.pop(index)
  new_game_state = self.game_state.apply_move(new_move)
  new_node = MCTSNode(new_game_state, self, new_move)
  self.children.append(new_node)
  return new_node

def record_win(self, winner):
  self.win_counts[winner] += 1
  self.num_rollouts += 1
```

Python으로 MCTS 구현하기

- 주요 MCTS 트리 속성에 접근하는 헬퍼 메소드

```
def can_add_child(self):
  return len(self.unvisited_moves) > 0

def is_terminal(self):
  return self.game_state.is_over()

def winning_frac(self, player):
  return float(self.win_counts[player]) / float(self.num_rollouts)
```

Python으로 MCTS 구현하기

- MCTS 알고리즘

```python
class MCTSAgent(agent.Agent):
  def select_move(self, game_state):
    root = MCTSNode(game_state)

    for i in range(self.num_rounds):
      node = root
      while (not node.can_add_child()) and (not node.is_terminal()):
        node = self.select_child(node)

      if node.can_add_child():
        node = node.add_random_child()

      winner = self.simulate_random_game(node.game_state)

      while node is not None:
        node.record_win(winner)
        node = node.parent
```

Python으로 MCTS 구현하기

- MCTS 롤아웃을 끝낸 후 수 선택

```
class MCTSAgent(agent.Agent):
...
  def select_move(self, game_state):
...
    best_move = None
    best_pct = -1.0
    for child in root.children:
      child_pct = child.winning_pct(game_state.next_player)
      if child_pct > best_pct:
        best_pct = child_pct
        best_move = child.move
    return best_move
```

탐색할 가지 선택법

- 게임 AI가 매 차례에 사용할 수 있는 시간에는 한계가 있다.
  즉, 롤아웃을 정해진 수만큼만 실행할 수 있다.
  - 롤아웃은 하나의 가능한 수를 더 잘 평가하게 해준다.
  - 롤아웃을 제한된 자원이라고 생각해보자. 만약 수 A에 대해 추가 롤아웃을 실행했다면 수 B에 대해서는 롤아웃을 하나 적게 해야 한다.
    → 즉, 제한된 예산을 할당하기 위한 전략이 필요하다.
- 이 때 일반적인 전략으로 **트리 신뢰도 상한선 (Upper Confidence Bound for Trees; UCT)** 방식을 사용한다.
  UCT 방식은 두 가지 상반된 목표 사이에 균형을 맞춘다.

탐색할 가지 선택법

- 첫번째 목표 : 주어진 시간 내에 최상의 수를 찾는다.

  → 활용(Exploitation)
  - 발견한 것에 대해 최대한 이득을 얻고자 한다.
  - 가장 승률이 높은 것으로 추정되는 수에 롤아웃을 더 많이 하고 싶을 것이다.
  - 지금은 수들 중 일부는 우연히 승률이 높을 것이다.
  - 하지만 이 가지들에 더 많은 롤아웃을 적용하면 추정치는 보다 정확해질 수 있다.
  - 거짓 양성 값은 점차 낮아질 것이다.

탐색할 가지 선택법

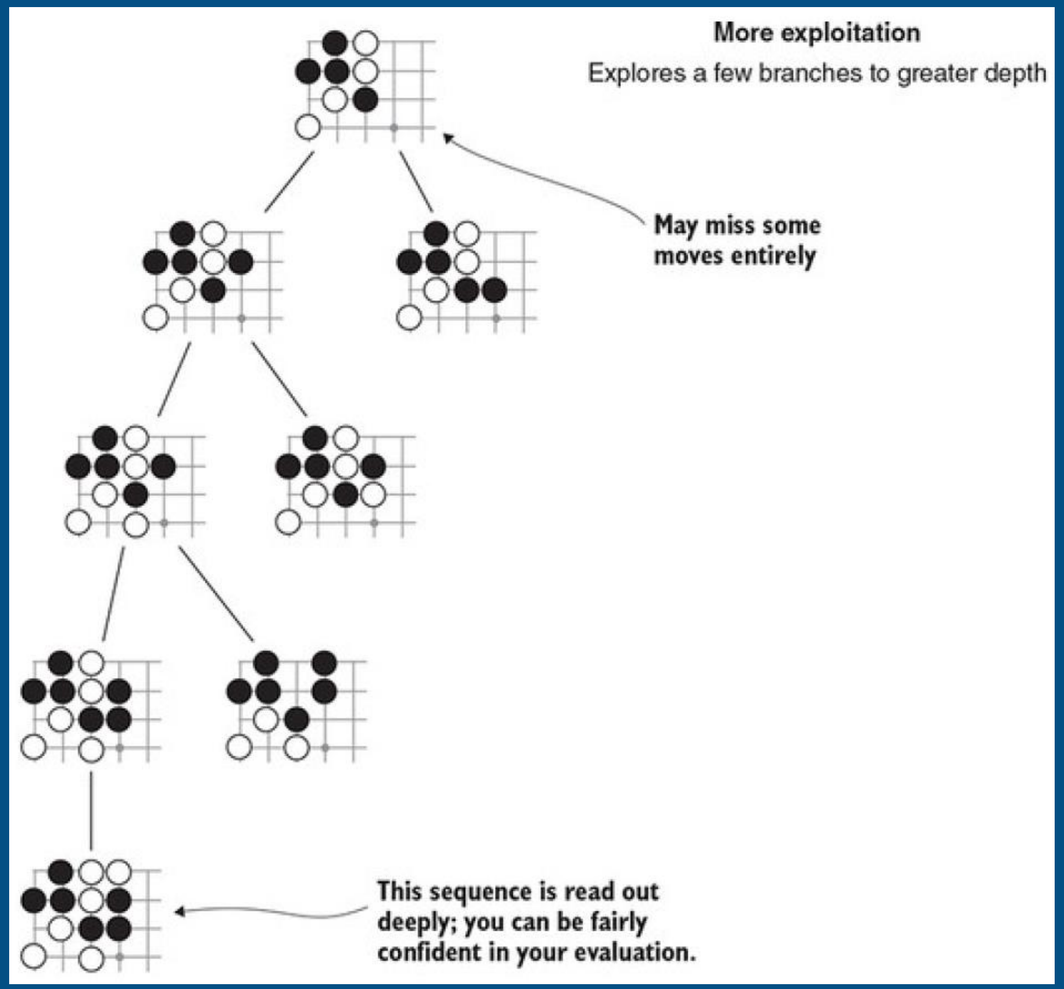
- 두번째 목표 : 최근 방문한 가지에서 보다 정확한 추정치를 얻는다.

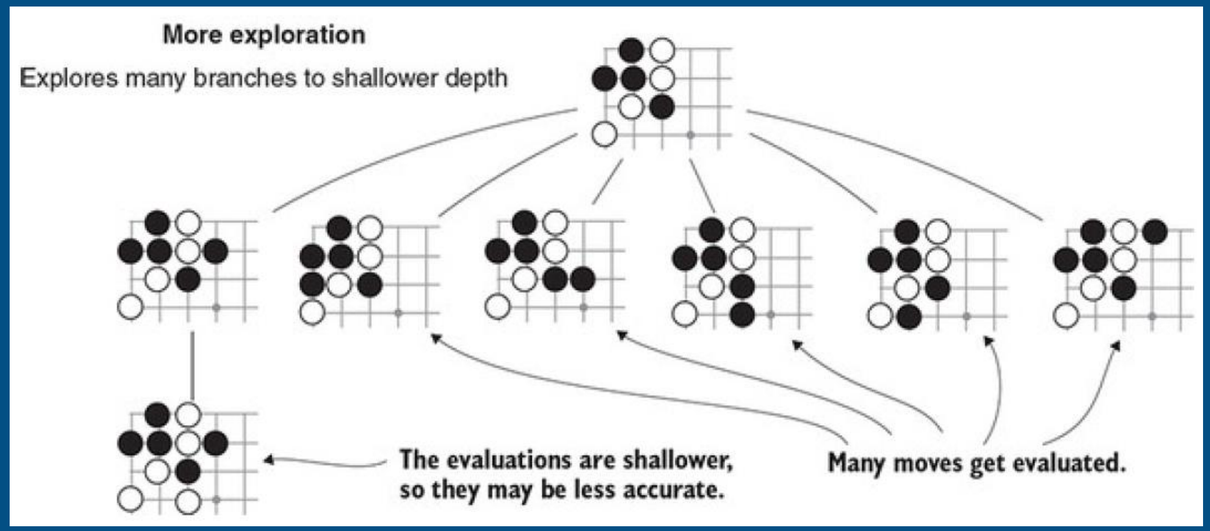
  → 탐험 (Exploration)

  - 노드에 롤아웃을 적은 횟수만 적용하면 추정치는 실제와 많이 달라질 수 있다.
  - 순전히 우연으로 정말 좋은 수에 낮은 추정치가 나올 수도 있다.
  - 롤아웃을 몇 번 더 사용함으로써 실제로 그 수가 얼마나 좋은지 발견할 수 있다.

탐색할 가지 선택법

- 활용-탐험 트레이드오프





탐색할 가지 선택법

- 각 노드에서 활용 목표를 나타내는 승률 w를 계산한다. 탐험을 나타내기 위해서는 전체 롤아웃 횟수인 N과 해당 노드의 롤아웃 수인 n을 구한다.

$$\sqrt{\frac{\log N}{n}}$$

- UCT 방식에서는 이 둘을 결합한다.

$$w + c\sqrt{\frac{\log N}{n}}$$

탐색할 가지 선택법

$$w + c\sqrt{\frac{\log N}{n}}$$

- 여기서는 활용과 탐험 정도를 나타내는 파라미터 c를 사용한다.
- UCT 방식에서는 각 노드별로 점수를 주고,
  가장 UCT 점수가 높은 노드에서 다음 롤아웃을 시작한다.
  - c값이 클수록 가장 적게 탐험한 노드를 방문하는데 시간을 쓸 것이다.
  - c값이 작을수록 가장 가능성이 높은 노드에 대해 더 나은 평가를 수집하는데 시간을 쓸 것이다.
  - 대략 1.5 근처에서 실험을 시작하는 걸 추천한다.
  - c를 온도(Temperature)라고도 한다.
  - 온도가 뜨거워질수록 더 탐색이 잘 퍼질 것이고, 온도가 차가워질수록 탐색이 더 집중적으로 이뤄진다.

탐색할 가지 선택법

- UCT 방식으로 탐색하며 가지 선택하기

```python
def uct_score(parent_rollouts, child_rollouts, win_pct, temperature):
  exploration = math.sqrt(math.log(parent_rollouts) / child_rollouts)
  return win_pct + temperature * exploration

class MCTSAgent:
...
  def select_child(self. node):
    total_rollouts = sum(child.num_rollouts for child in node.children)

    best_score = -1
    best_child = None
    for child in node.children:
      score = uct_score(
        total_rollouts,
        child.num_rollouts,
        child.winning_pct(node.game_state.next_player),
        self.temperature)
      if score > best_score:
        best_score = uct_score
        best_child = child
    return best_child
```

바둑에 MCTS 적용하기

- 빠른 코드가 강력한 봇을 만든다.
  - 적당한 시간 동안 많은 롤아웃을 수행하려면 구현한 코드를 최적화해야 한다.
  - 다른 조건이 모두 동일하다면 롤아웃이 많은 경우 판단 결과가 더 나아진다.
  - 코드를 빠르게 해서 같은 시간 동안 롤아웃을 많이 돌리면 봇은 무조건 더 강해질 수 있다.
- 더 좋은 롤아웃 정책은 더 나은 평가를 만든다.
  - 롤아웃 정책
    - 무거운 롤아웃(Heavy Rollout) : 게임 특화 규칙이 들어간 롤아웃
      (대표적으로 바둑에서 일반적으로 사용하는 기본 전술 형상 리스트를 구축하고 알려진 답을 구현해두는 방식이 있다.)
    - 가벼운 롤아웃(Light Rollout) : 거의 완전한 무작위성이 적용된 롤아웃

바둑에 MCTS 적용하기

- 예의 바른 봇은 떠나야 할 때를 안다.
  - 게임 AI를 만드는 이유가 단순히 최고의 알고리즘을 개발하기 위한 연습은 아니다.
    게임 AI는 인간을 상대로 즐거운 경험을 만들어주는 것이기도 하다.
  - 기본 MCTS 구현 내용 상단에 인간 친화적인 돌던지기 규칙을 손쉽게 추가할 수 있다.
    최상의 선택지가 10% 정도의 낮은 승률을 보인다면 봇이 돌을 던지도록 만들 수도 있다.

감사합니다!

스터디 듣느라 고생 많았습니다.