과제 1016

2315028 김성현

이론

▼ 적대적 및 게임탐색

▼ 경쟁적 환경 (ft.게임)

에이전트가 복수인 경우, 에이전트가 경쟁적인 경우에 탐색수행

완전관측가능한 - 정보가 모두 공개된 경우 ex_체스, 바둑 <> ex_ 포커

2인용의 교대식 - 번갈아 가면서 진행되는 경우

제로섬 게임 - 이기고 지는 사람이 명확한 경우

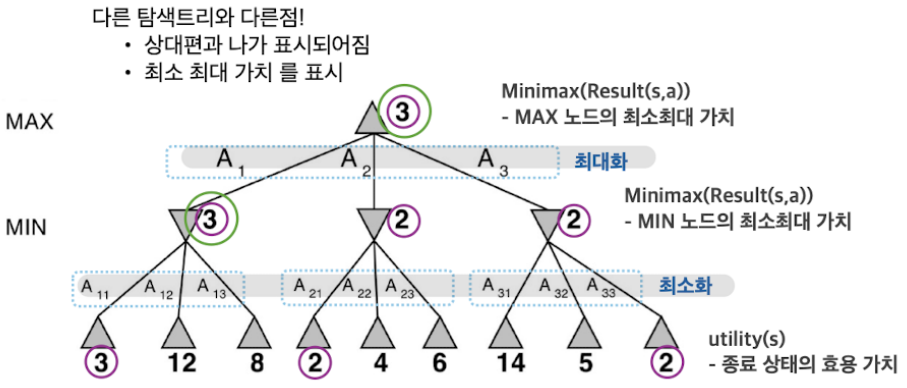
▼ 적대적 탐색 (Adversarial search)

게임 문제푸는 방법 1) 여러 에이전트를 하나의 경제로 2) 대립하는 에이전트를 환경의 일부로 간주하는방법들이 존재

- 적대적 게임트리 탐색기법

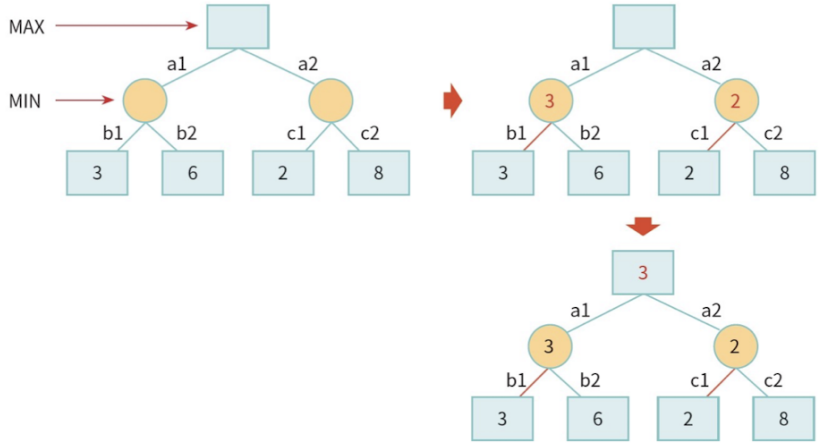
▼ **최소 최대 탐색** : 게임 내 상태트리인 게임트리를 만들어 특정 플레이어가 최대한 이익을 얻고, 상대방의 이익은 최소화하는 상황을 추적

트리에서 각 상태의 최소 최대 가치를 구해 표시함. (최소 최대 가치 : 특정플레이어를 기준으로 최대 값을 상대 플레이어를 기준으로 최솟 값을 의미함)



min : 최소화 / max : 최대화 ← 서로 이기려고 하는 것

▼ 과정

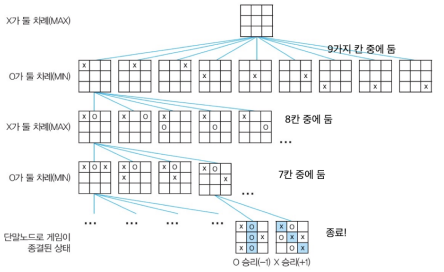


단말노드에서 최소 최대로 가치를 계산해 최선의 전략은 '3'

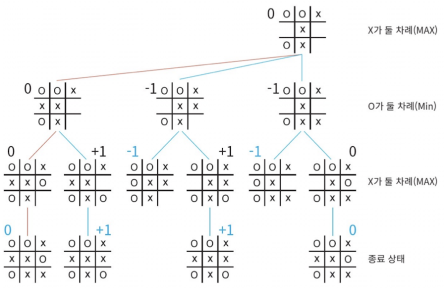
▼ 예시_ 틱택토 게임

삼목게임과 같은것으로 한줄을 먼저 채우면 승리. MAX를 먼저 두며 번갈아 수를 둠

▼ 상태트리



상태표기 : 승리시 +1 / 무승부 0 / MIN이 승리 (패배시) -1



▼ 알고리즘

```
Function MINIMAX(game, state)
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

Function MAX_VALUE (game, state)   게임이 끝났니? 그때의 효용값을 출력
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← -∞, null    V에 -무한대를 넣음(가장 작은 값으로)
  for each a in game.ACTIONS(state) do
      v2, a2 ← MIN_VALUE(game, game.RESULT(state, a))   Max, min을 번갈아가며 확인하면서 들어가져야함
      if v2 > v then
          v, move ← v2, a  최솟값을 찾아 넣음
  return v, move

Function MIN_VALUE (game, state)  게임이 끝났니? 그때의 효용값을 출력
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← +∞, null   V에 +무한대를 넣음(가장 작은 값으로)
  for each a in game.ACTIONS(state) do
      v2, a2 ← MAX_VALUE(game, game.RESULT(state, a))   Max, min을 번갈아가며 확인하면서 들어가져야함
      if v2 < v then          그전값이 최솟값이었는지 확인
          v, move ← v2, a     최댓값을 찾아 넣음
  return v, move
```

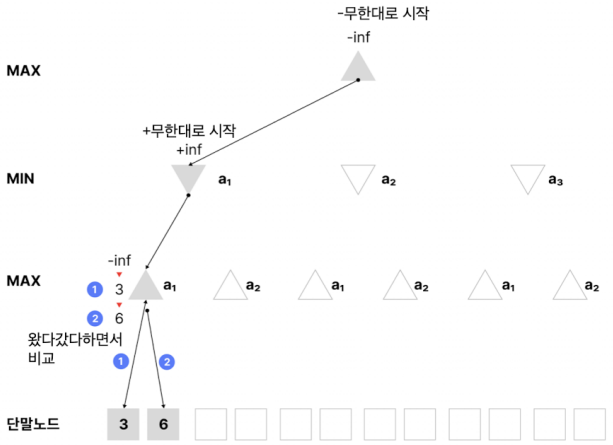
재귀적 알고리즘

초기값을 (-/+)무한대로

최대 : - 무한대 , 어떤 값이 들어와도 그것보다 크도록

최소 : +무한대, 어떤 값이들어와도 그것보다 작도록

깊이우선탐색과 유사함



- 완결성 : 유한한 탐색트리 안에 해답이 존재하면 반드시 찾음
- 시간복잡도 : O(b^m), 깊이우선탐색과 유사
- 공간복잡도 : O(bm)
- 최적성 : 게임 상태공간을 완전히 탐색하고 최소 최대값을 살펴 탐색하기에 최적성이 보장됨

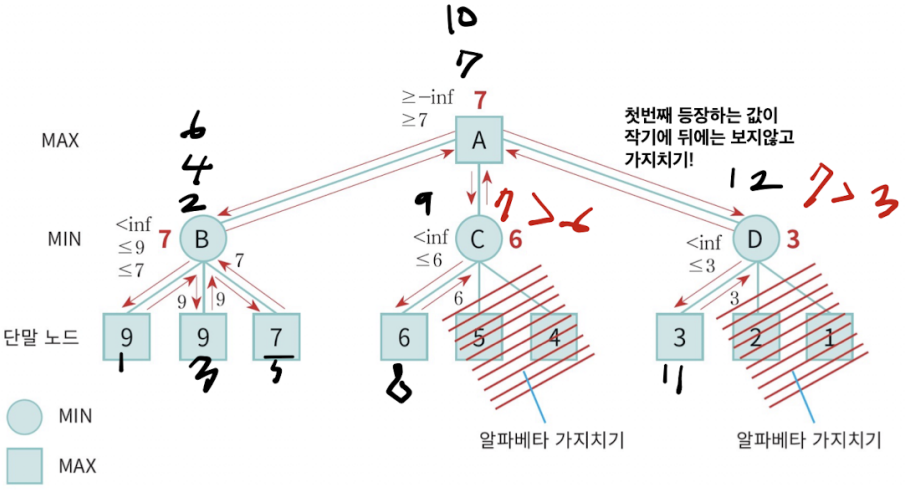
▼ **알파베타 가지치기**(alpha-beta pruning) : 불필요한 가지들을 쳐내는 방법

- a(알파) : MAX가 찾아낸 가장 큰 값 / b(베타) : MIN이 찾아낸 가장 작은 값

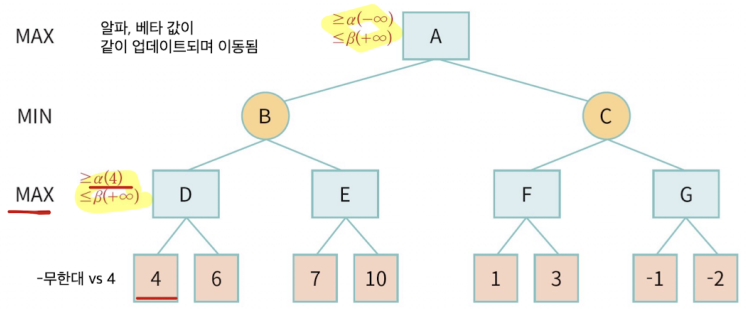
  a(알파) 가지치기 : MIN노드에서 자식노드의 값이 알파값보다 작거나 같으면 가지치기

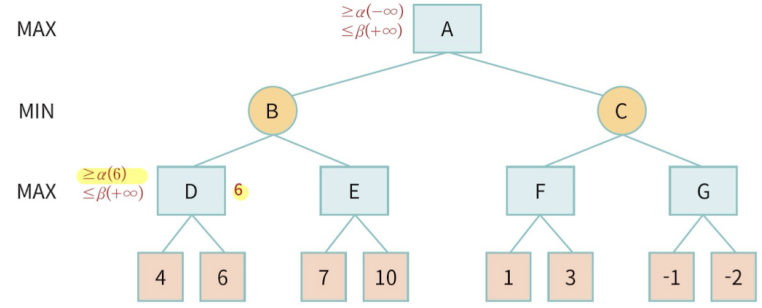
  b(베타) 가지치기 : MAX노드에서 자식노드의 값이 베타값보다 크거나 같으면 가지치기

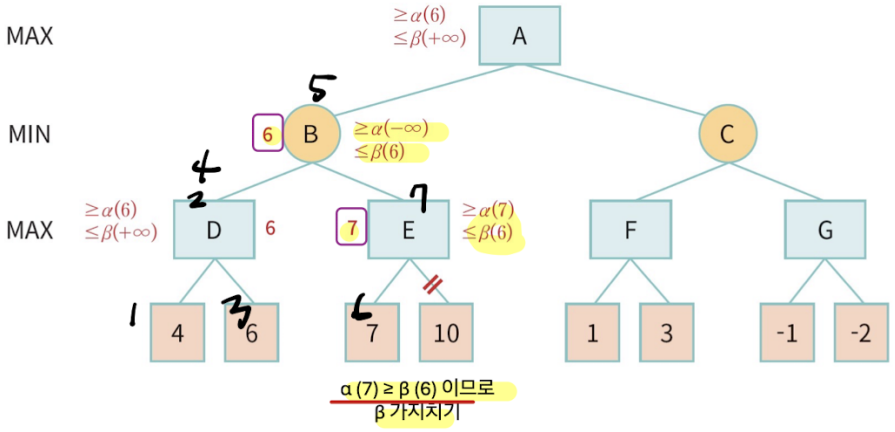
MAX는 a(알파)만 업데이트 / MIN은 b(베타)만 업데이트



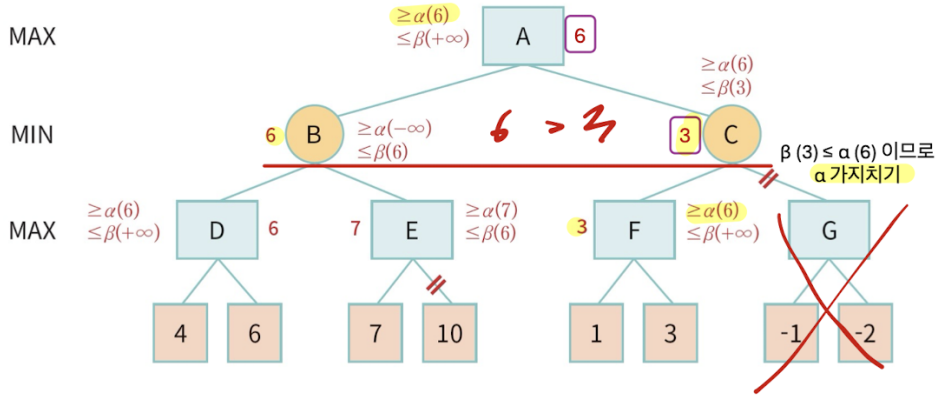
▼ 과정_ **가지치기를 주시하며







베타 가지치기



알파 가지치기

부모와 자식 노드간의 계속 비교해가며 가지치기를 결정!

▼ 알고리즘

최소최대탐색과 유사함

다른점 : 파라미터에 a,b(알파, 베타)를 추가함

MAX는 알파만 업데이트 / MIN은 베타만 업데이트

```
Function ALPHABETA(game, state)
  player ← game.TO-MOVE(state)
  value, move ← MAX_VALUE(game, state, -∞, +∞)
  return move

Function MAX_VALUE (game, state, α, β)
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← -∞, null
  for each a in game.ACTIONS(state) do
      v2, a2 ← MIN_VALUE(game, game.RESULT(state, a), α, β)
      if v2 > v then
          v, move ← v2, a
          α ← MAX(α, v)
      if v >= β then break
  return v, move

Function MIN_VALUE (game, state , α, β)
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← +∞, null
  for each a in game.ACTIONS(state) do
      v2, a2 ← MAX_VALUE(game, game.RESULT(state, a), α, β)
      if v2 < v then
          v, move ← v2, a
          β ← MIN(β, v)
      if v <= α then break
  return v, move
```

⇒ 깊은 트리의 경우 단말노드까지 도달해야하기에 너무 오래걸리므로.. 그전에 현재 상태를 평가하면서 진행하면? 휴리스틱 평가함수(: 비단말노드이지만 휴리스틱값을 이용해 업데이트하며 다음값으로 이동)

실습

▼ 적대적 탐색

▼ 틱택톡 게임

: X와 O가 번갈아가며 두는 삼목(오목의 3x3짜리 버전) 게임 문제

- 현재 플레이어가 누구인지 명시하고, 플레이어를 교체할 수 있어야 함
- 현재 상태에서 어떤 수를 둘 수 있는지 확인하고, 수를 둘 수 있어야 함
- 수를 둔 후에 새로운 보드 상태를 반환해야 함
- 게임의 종료 상태를 판단 및 특정 플레이어의 승리 여부 확인할 수 있어야 함
- 종료 상태 도달 시 효용 값 반환할 수 있어야 함

```
class TicTacToe:
    def __init__(self):
        self.board = [' '] * 9  # 9개의 빈 칸으로 구성된 보드 생성
        self.player = 'X'  # 첫 번째 플레이어를 'X'로 설정
        self.round = 0 # 게임 턴

    def empty_cells(self): # 가능한 수 추적 (현재 보드에서 빈 칸의 인덱스 반환), 빈칸을 찾음
        return [i for i, cell in enumerate(self.board) if cell == ' ']

    def valid_move(self, x): # 주어진 위치 x 에 수를 둘 수 있는지 판단
        return x in self.empty_cells()

    def move(self, x): # 플레이어가 (가능한) 위치 x에 수를 놓음
        if self.valid_move(x):
            self.board[x] = self.player
            return True
        return False

    def check_win(self, player): # 특정 플레이어가 승리했는지 확인 (총8케이스), 승패를 확인
        win_conditions = [  #승리조건
            [0, 1, 2], [3, 4, 5], [6, 7, 8],  # 가로 승리 조건
            [0, 3, 6], [1, 4, 7], [2, 5, 8],  # 세로 승리 조건
            [0, 4, 8], [2, 4, 6]              # 대각선 승리 조건
        ]
        for condition in win_conditions: #all함수, 안에 인자가 true/false를 반환
            if all(self.board[i] == player for i in condition): # 인자로 받은 요소가 모두
                return True
        return False

    def is_terminal(self): # 게임이 종료되었는지 확인 (승리, 패배, 무승부-승자없고 빈칸없을때)
        return self.check_win('X') or self.check_win('O') or not self.empty_cells()
```

```
def utility(self): # 게임의 결과에 따른 점수 반환
    if self.check_win('X'):
        return 1 # 승리
    elif self.check_win('O'):
        return -1 # 패배
    else:
        return 0  # 무승부

def switch_player(self): # 현재 수를 둘 플레이어 변경
    self.player = 'O' if self.player == 'X' else 'X'

def result(self, move): # 보드에서 특정 수를 두었을 때 결과 보드를 반환 (새로운 상태)
    new_game = TicTacToe()
    new_game.board = self.board[:]
    new_game.player = self.player
    new_game.round = self.round
    new_game.move(move)
    new_game.switch_player()
    return new_game

def __str__(self): #보기좋게 만드는 출력함수
    board_str = f'== Round {self.round} ==\n'
    for i, cell in enumerate(self.board):
        if i % 3 == 0 and i != 0:
            board_str += '\n'
        board_str += f'|{cell}|'
    return board_str
```

`empty_cells` : 가능한 수를 추적 (현재 보드에서의 빈칸 인덱스)

`valid_move` : 주어진 위치 x에 수를 둘 수 있는지 판단

move : 플레이어가 가능한 위치 x에 수를 놓음

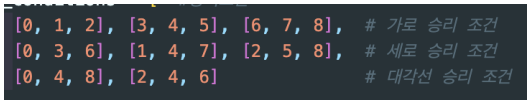
`check_win` : 특정 플레이어가 승리했는지 확인 (총 8케이스).

`is_terminal` : 게임이 종료되었는지 확인(승리/패배/무승부)

`utility` : 게임 결과에 따른 점수 변환

`switch_player` : 현재 수를 둘 플레이어를 변경

`result` : 보드에서 특정 수를 두었을때 결과 보드를 반환



```
[0, 1, 2], [3, 4, 5], [6, 7, 8],  # 가로 승리 조건
[0, 3, 6], [1, 4, 7], [2, 5, 8],  # 세로 승리 조건
[0, 4, 8], [2, 4, 6]              # 대각선 승리 조건
```

▼ 최소 최대 탐색

: MAX_VALUE(가장 높은 값 찾음) 및 MIN_VALUE(가장 낮은 값 찾음) 함수 에서 MAX 및 MIN 플레이어가 최선의 행동을 택함. 두 함수가 재귀적으로 호출하면서 게임 트리의 가능한 모든 상태를 탐색함

- 의사코드

```
Function MINIMAX(game, state)
  player ← game.TO_MOVE(state)
  value, move ← MAX_VALUE(game, state)
  Return move

Function MAX_VALUE (game, state)
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← −∞, null
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN_VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

Function MIN_VALUE (game, state)
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← +∞, null
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX_VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

- 코드

```python
def minimax(game):
    if game.is_terminal():
        print(game)
        if game.check_win('X'):
            return 'X 승리!'
        elif game.check_win('O'):
            return 'O 승리!'
        else:
            return '비겼습니다!'
    else:
        if game.player == 'X': # 플레이어 X로 시작하면 최대값 탐색으로 시작
            _, move = max_value(game)
        else: # 플레이어 O로 시작하면 최소값 탐색으로 시작
            _, move = min_value(game)
        game.move(move)
        print(game)
        game.switch_player()
        game.round += 1
        return minimax(game) # 자기 자신을 재귀적으로 호출
```

```python
def max_value(game):
    # 실습
    if game.is_terminal():
        return game.utility(), None
    value, move = -float('inf'), None
    for action in game.empty_cells():
        next_game = game.result(action)
        v, _ = max_value(next_game)
        if v > value:
            value, move = v, action
    return value, move

def min_value(game):
    # 실습
    if game.is_terminal():
        return game.utility(), None
    value, move = float('inf'), None
    for action in game.empty_cells():
        next_game = game.result(action)
        v, _ = min_value(next_game)
        if v < value:
            value, move = v, action
    return value, move
```

minimax : 게임 진행

max_value : 가장 높은 값을 찾음

min_value : 가장 낮은 값을 찾음

```
== Round 4 ==
|X||O||X|
|O||X|| |
| || || |
== Round 5 ==
|X||O||X|
|O||X|| |
|O|| || |
== Round 6 ==
...
|X||O||X|
|O||X||X|
|O||X||O|
비겼습니다!
```

▼ 알파베타 가지치기

: 최소 최대 탐색 알고리즘과 유사하지만 알파, 베타가 존재하고 가지치기가 추가됨

- 의사코드

```
Function ALPHABETA(game, state)
  player ← game.TO-MOVE(state)
  value, move ← MAX_VALUE(game, state, −∞, +∞)
  return move

Function MAX_VALUE (game, state, α, β)
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← −∞, null
  for each a in game.ACTIONS(state) do
      v2, a2 ← MIN_VALUE(game, game.RESULT(state, a), α, β)
      if v2 > v then
          v, move ← v2, a
          α ← MAX(α, v)
      if v >= β then break
  return v, move

Function MIN_VALUE (game, state , α, β)
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← +∞, null
  for each a in game.ACTIONS(state) do
      v2, a2 ← MAX_VALUE(game, game.RESULT(state, a), α, β)
      if v2 < v then
          v, move ← v2, a
          β ← MIN(β, v)
      if v <= α then break
  return v, move
```

- 코드

```python
def alphabeta(game):
    if game.is_terminal():
        print(game)
        if game.check_win('X'):
            return 'X 승리!'
        elif game.check_win('O'):
            return 'O 승리!'
        else:
            return '비겼습니다!'
    else:
        if game.player == 'X':
            _, move = max_value_ab(game, -float('inf'), float('inf'))
        else:
            _, move = min_value_ab(game, -float('inf'), float('inf'))
        game.move(move)
        print(game)
        game.switch_player()
        game.round += 1
        return alphabeta(game)
```

```
== Round 4 ==
|X||X||O|
| ||O|| |
|X|| || |
== Round 5 ==
|X||X||O|
|O||O|| |
|X|| || |
== Round 6 ==
...
|X||X||O|
|O||O||X|
|X||O||X|
비겼습니다!
```

최소 최대 탐색 알고리즘과 결과가 같음!

⇒ why? 효율적인 값을 탐색하는 알고리즘들이기에..

```python
def max_value_ab(game, alpha, beta):
    # 실습
    if game.is_terminal():
        return game.utility(), None
    value, move = -float('inf'), None
    for action in game.empty_cells():
        next_game = game.result(action)
        next_value, _ = min_value_ab(next_game, alpha, beta)
        if next_value > value:
            value, move = next_value, action
        alpha = max(alpha, value)
        if value >= beta:
            break
    return value, move

def min_value_ab(game, alpha, beta):
    # 실습
    if game.is_terminal():
        return game.utility(), None
    value, move = float('inf'), None
    for action in game.empty_cells():
        next_game = game.result(action)
        next_value, _ = max_value_ab(next_game, alpha, beta)
        if next_value < value:
            value, move = next_value, action
        beta = min(beta, value)
        if value <= alpha:
            break
    return value, move
    return value, move
```

max_value_ab에서 알파 가지치기

min_value_ab에서 베타 가지치기