

인공지능



2판

파이썬으로 배우는 머신러닝과 딥러닝

제3장 게임트리

학습 목표

- 미니맥스 알고리즘을 살펴본다.
- 알파베타 가지치기 알고리즘을 이해한다.

이번장에서 다루는 게임의 조건

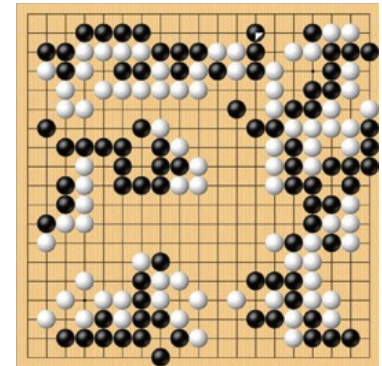
- 이번 장에서는 게임을 위한 프로그램을 작성하는 문제를 생각해보자. 설명을 단순화하기 위해 우리는 다음과 같은 속성을 가진 게임만 고려할 것이다. 바둑이나 체스가 여기에 속한다.
- 두 명의 경기자 - 경기자들이 연합하는 경우는 다루지 않는다.
- 제로섬 게임 - 한 경기자의 승리는 다른 경기자의 패배이다. 협동적인 승리는 없다.
- 차례대로 수를 두는 게임만을 대상으로 한다. (순차적인 게임)

인공지능과 게임

- 게임은 예전부터 인공지능의 매력적인 연구 주제였다.
- Tic-Tac-Toe나 체스, 바둑과 같은 게임은 추상적으로 정의할 수 있고 지적 능력과 연관이 있는 것으로 생각되었다.
- 이들 게임은 비교적 적은 수의 연산자들을 가진다. 연산의 결과는 엄밀한 규칙으로 정의된다.

바둑에서 나타나는 모든 경우의 수

- 바둑판에는 돌을 놓을 수 있는 곳이 $19 \times 19 = 361$ 이다. 한 곳에는 흰돌(white) 또는 검은돌(black) 또는 비워놓을 수 있다(empty). 따라서 각 361개의 점마다 최대 3가지의 선택이 있다. 따라서 발생할 수 있는 상태의 상한은 다음과 같다.
- 관련 논문 $3 \times 3 \times 3 \times \dots = 3^{361}$ 170 정도의 숫자라고 한다. 이것은 엄청난 숫자로서 우주에 존재하는 원자의 개수로 믿어지는 숫자인 1080보다도 훨씬 많다. 따라서 완벽한 탐색은 불가능하다.



게임의 정의

- 2인용 게임
- 두 경기자를 MAX와 MIN으로 부르자.
- 항상 MAX가 먼저 수를 둔다고 가정한다.

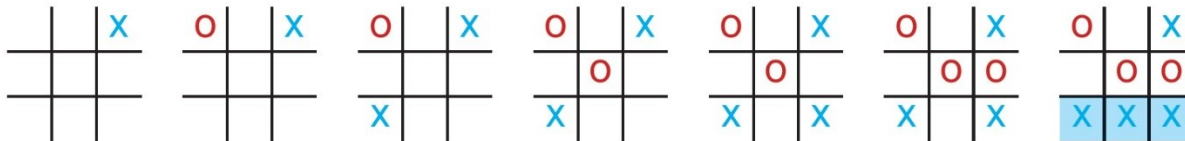
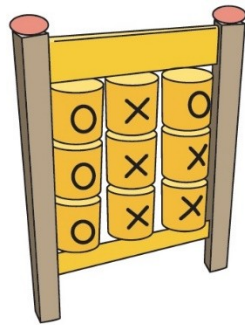


그림 3-1 Tic-Tac-Toe 게임

Tic-Tac-Toe의 게임 트리

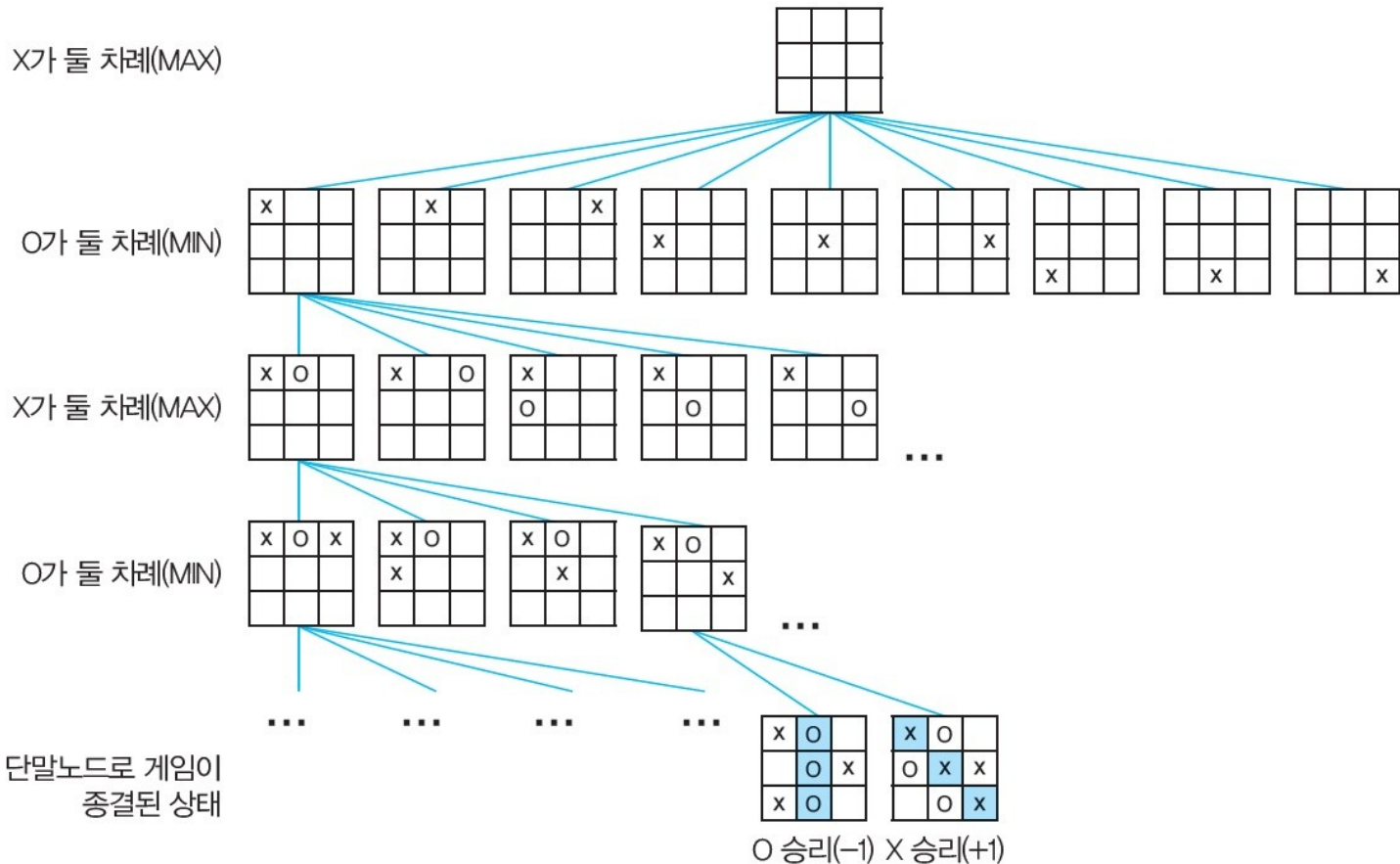


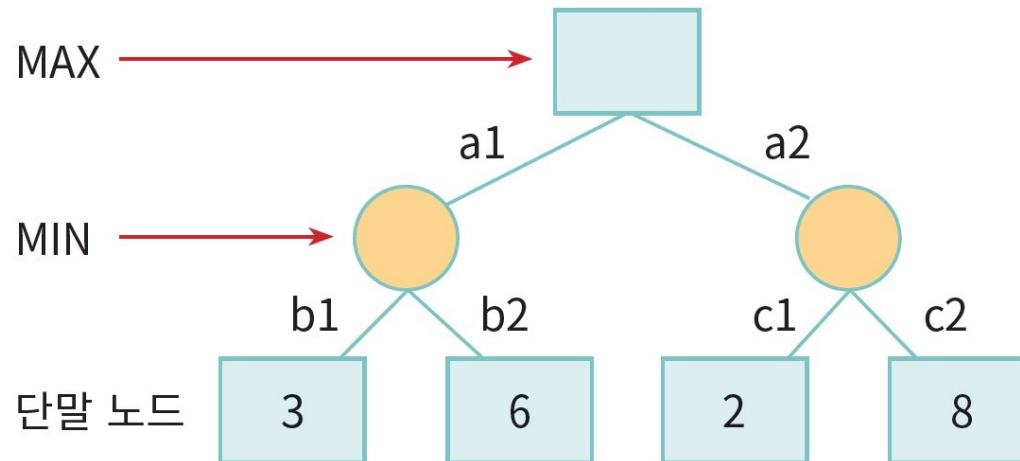
그림 3-2 틱택토 게임의 게임 트리(일부)

Tic-Tac-Toe 게임 트리의 크기

- Tic-Tac-Toe의 게임 트리는 크기가 얼마나 될까?
- Tic-Tac-Toe 게임 보드는 3×3 크기를 가지고 있고 한 곳에 수를 놓으면 다른 사람이 놓을 수 있는 곳은 하나가 줄어들게 된다. $9 \times 8 \times 7 \times \dots \times 1 = 9! = 362,880$
- 하지만 대칭이나 반사를 제외하면 서로 다른 상태는 5478개뿐이라고 한다.

미니맥스 알고리즘

- 안전하게 하려면 상대방이 최선의 수를 둔다고 생각하면 된다.



미니맥스(minimax) 알고리즘

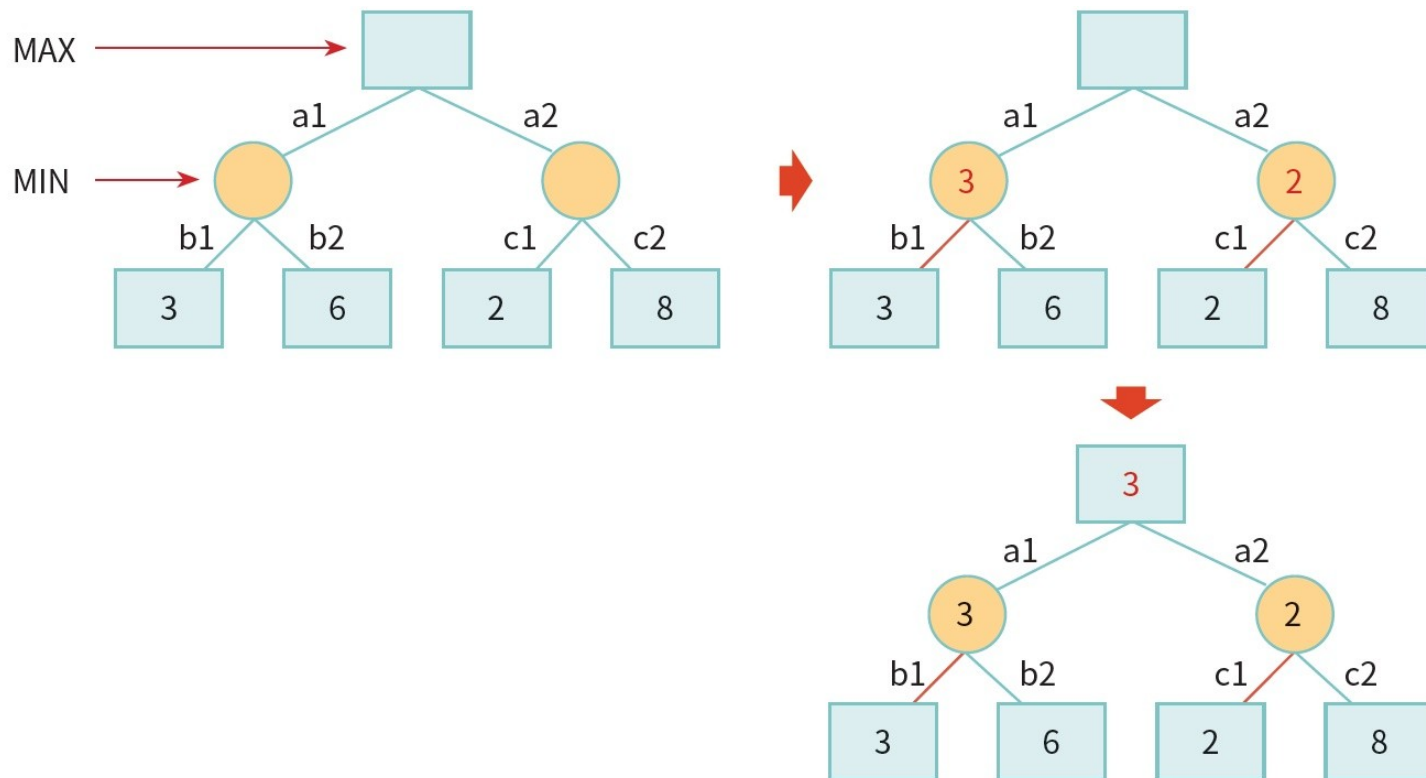


그림 3-4 미니맥스 알고리즘

틱택토 게임에서의 미니맥스

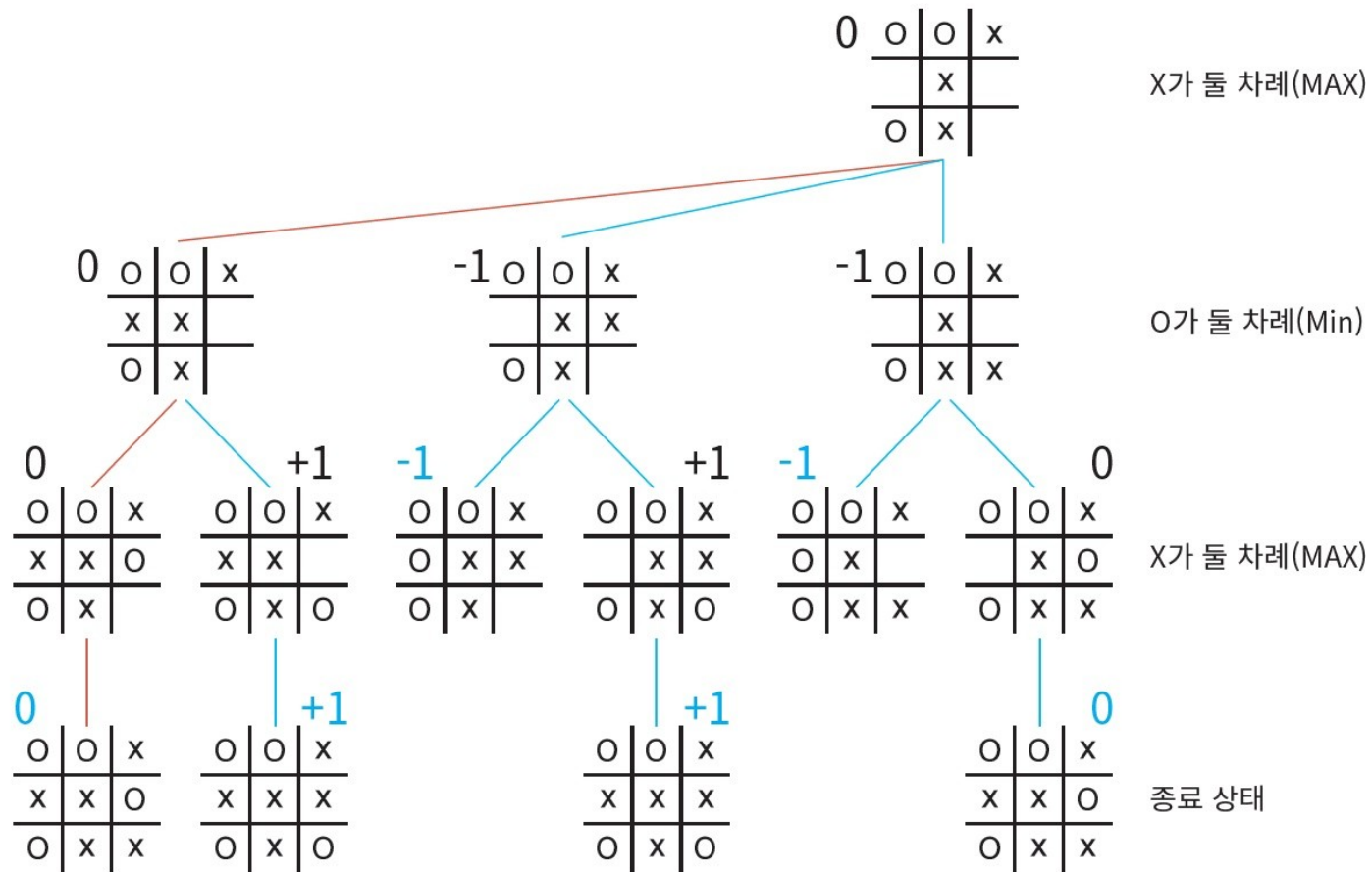
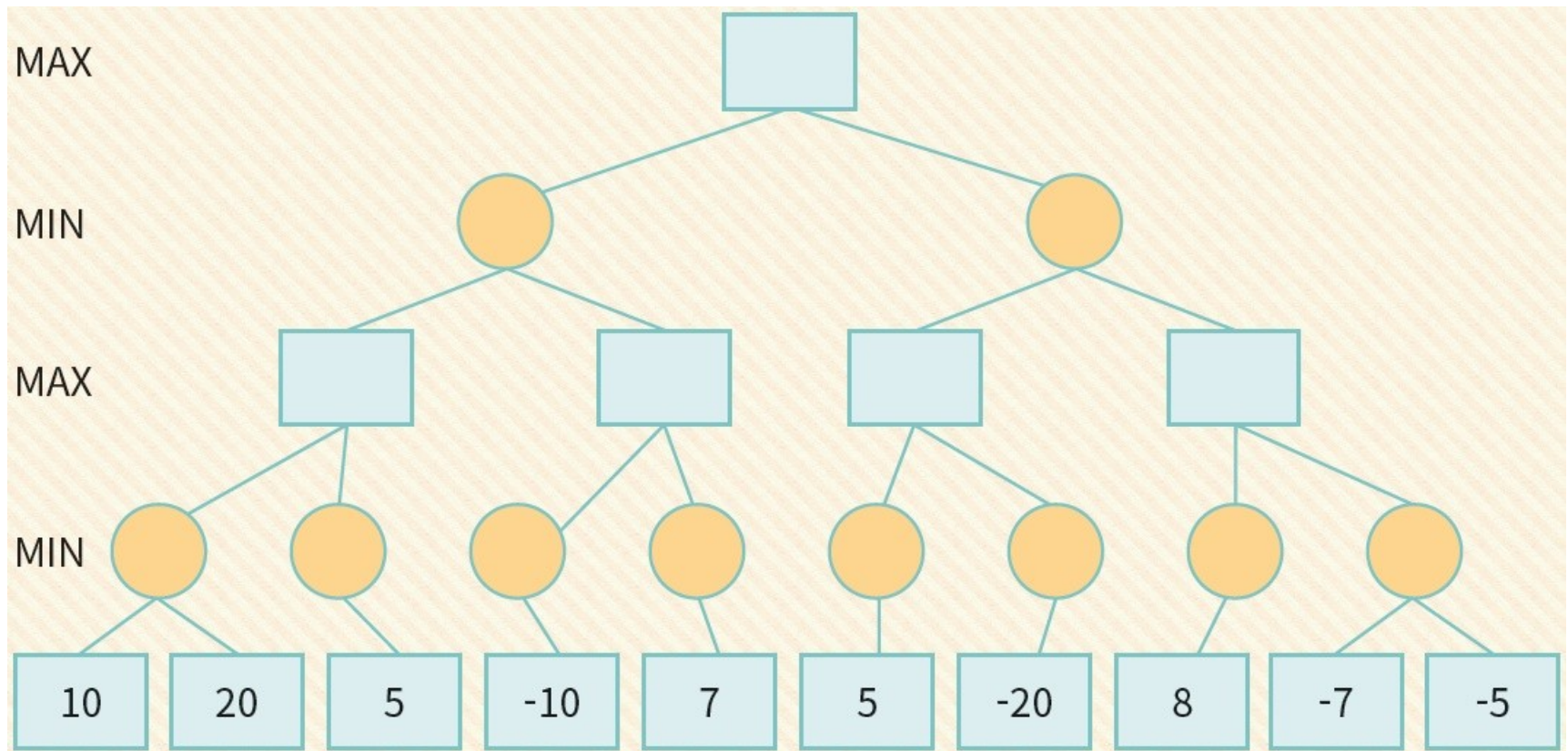


그림 3-5 틱택토 게임에서 미니맥스 알고리즘

Lab: 미니맥스 알고리즘 실습



미니맥스 알고리즘

```
function minimax(node, depth, maxPlayer)
  if depth == 0 or node가 단말 노드 then
    return node의 휴리스틱 값
  if maxPlayer then
    value  $\leftarrow -\infty$ 
    for each child of node do
      value  $\leftarrow \max(\text{value}, \text{minimax}(\text{child}, \text{depth} - 1, \text{FALSE}))$ 
    return value
  else // 최소화 노드
    value  $\leftarrow +\infty$ 
    for each child of node do
      value  $\leftarrow \min(\text{value}, \text{minimax}(\text{child}, \text{depth} - 1, \text{TRUE}))$ 
    return value
```

미니맥스 알고리즘의 분석

- 완결성: 미니맥스 알고리즘은 완결될 수 있다. 유한한 탐색 트리 안에 해답이 존재하면 반드시 찾는다.
- 최적성: 미니맥스 알고리즘은 최적의 알고리즘이다.
- 시간 복잡도: 만약 트리의 최대 깊이가 m 이고 각 노드에서의 가능한 수가 b 개라면, 미니맥스 알고리즘의 시간 복잡도는 $O(b^m)$ 이다.
- 공간 복잡도: 공간 복잡도도 $O(b^m)$ 이다.

Tic-Tac-Toe 구현

보드는 1차원 리스트로 구현한다.

```
game_board = [' ', ' ', ' ',  
              ' ', ' ', ' ',  
              ' ', ' ', ' ']
```

비어 있는 칸을 찾아서 리스트로 반환한다.

```
def empty_cells(board):  
    cells = []  
    for x, cell in enumerate(board):  
        if cell == ' ':  
            cells.append(x)  
    return cells
```

비어 있는 칸에는 놓을 수 있다.

```
def valid_move(x):  
    return x in empty_cells(game_board)
```

Tic-Tac-Toe 구현

```
# 위치 x에 놓는다.
def move(x, player):
    if valid_move(x):
        game_board[x] = player
        return True
    return False

# 현재 게임 보드를 그린다.
def draw(board):
    for i, cell in enumerate(board):
        if i%3 == 0:
            print("\n-----")
            print('|', cell, '|', end="")
            print("\n-----")

# 보드의 상태를 평가한다.
def evaluate(board):
    if check_win(board, 'X'):
        score = 1
    elif check_win(board, 'O'):
        score = -1
    else:
        score = 0
    return score
```


Tic-Tac-Toe 구현

```
# 1차원 리스트에서 동일한 문자가 수직선이나 수평선, 대각선으로 나타나면  
# 승리한 것으로 한다.
```

```
def check_win(board, player):  
    win_conf = [  
        [board[0], board[1], board[2]],  
        [board[3], board[4], board[5]],  
        [board[6], board[7], board[8]],  
        [board[0], board[3], board[6]],  
        [board[1], board[4], board[7]],  
        [board[2], board[5], board[8]],  
        [board[0], board[4], board[8]],  
        [board[2], board[4], board[6]],  
    ]  
    return [player, player, player] in win_conf
```

```
# 1차원 리스트에서 동일한 문자가 수직선이나 수평선, 대각선으로 나타나면  
# 승리한 것으로 한다.
```

```
def game_over(board):  
    return check_win(board, 'X') or check_win(board, 'O')
```

```

def minimax(board, depth, maxPlayer):
    pos = -1
    # 단말 노드이면 보드를 평가하여 위치와 평가값을 반환한다.
    if depth == 0 or len(empty_cells(board)) == 0 or game_over(board):
        return -1, evaluate(board)

    if maxPlayer:
        value = -10000 # 음의 무한대
        # 자식 노드를 하나씩 평가하여서 최선의 수를 찾는다.
        for p in empty_cells(board):
            board[p] = 'X' # 보드의 p 위치에 'X'을 놓는다.

            # 경기자를 교체하여서 minimax()를 순환호출한다.
            x, score = minimax(board, depth-1, False)
            board[p] = '.' # 보드는 원 상태로 돌린다.
            if score > value:
                value = score # 최대값을 취한다.
                pos = p # 최대값의 위치를 기억한다.
        else:
            value = +10000 # 양의 무한대
            # 자식 노드를 하나씩 평가하여서 최선의 수를 찾는다.
            for p in empty_cells(board):
                board[p] = 'O' # 보드의 p 위치에 'O'을 놓는다.

                # 경기자를 교체하여서 minimax()를 순환호출한다.
                x, score = minimax(board, depth-1, True)
                board[p] = '.' # 보드는 원 상태로 돌린다.
                if score < value:
                    value = score # 최소값을 취한다.
                    pos = p # 최소값의 위치를 기억한다.
    return pos, value # 위치와 값을 반환한다.

```

Tic-Tac-Toe 구현

```
player='X'
# 메인 프로그램
while True:
    draw(game_board)
    if len(empty_cells(game_board)) == 0 or game_over(game_board):
        break
    i, v = minimax(game_board, 9, player=='X')
    move(i, player)
    if player=='X':
        player='O'
    else:
        player='X'

if check_win(game_board, 'X'):
    print('X 승리!')
elif check_win(game_board, 'O'):
    print('O 승리!')
else:
    print('비겼습니다!')
```

실행결과

| | | |

| | | |

| | | |

|X | | |

| | | |

| | | |

|X | | |

| |O | |

| | | |

...

|X |X |O |

|O |O |X |

|X |O |X |

비겼습니다!

알파베타 가지치기

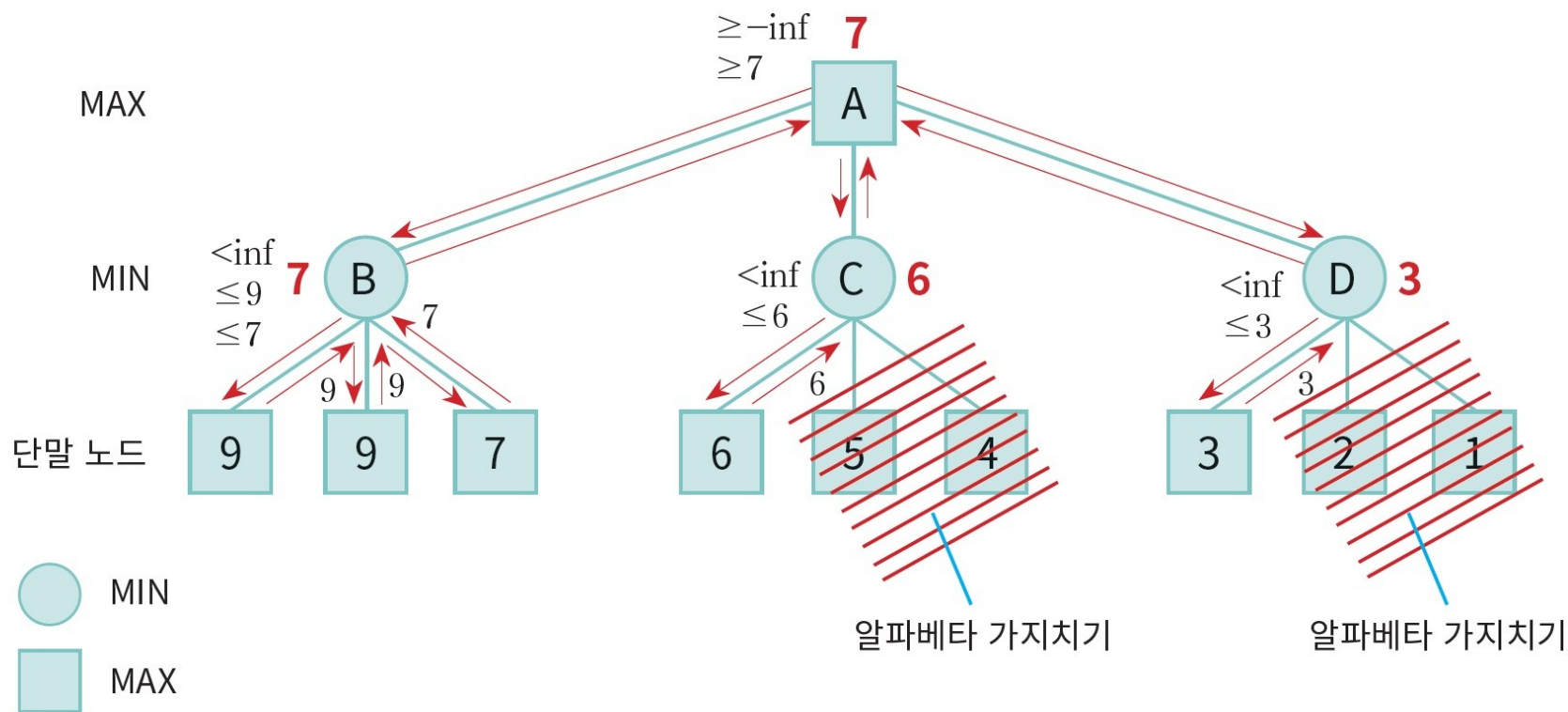


그림 3-6 알파베타 가지치기

알파베타 가지치기

- 미니맥스 알고리즘에서 형성되는 탐색 트리 중에서 상당 부분은 결과에 영향을 주지 않으면서 가지들을 쳐낼 수 있다.
- 이것을 알파베타 가지치기라고 한다.
- 탐색을 할 때 알파값과 베타값이 자식 노드로 전달된다. 자식 노드에서는 알파값과 베타값을 비교하여서 쓸데없는 탐색을 중지할 수 있다.
- MAX는 알파값만을 업데이트한다. MIN은 베타값만을 업데이트한다.

알파베타 알고리즘

```
function alphabeta(node, depth,  $\alpha$ ,  $\beta$ , maxPlayer)
```

```
  if depth == 0 or node가 단말 노드 then
```

```
    return node의 휴리스틱 값
```

```
  if maxPlayer then // 최대화 경기자
```

```
    value  $\leftarrow -\infty$ 
```

```
    for each child of node do
```

```
      value  $\leftarrow \max(\text{value}, \text{alphabeta}(\text{child}, \text{depth}-1, \alpha, \beta, \text{FALSE}))$ 
```

```
       $\alpha \leftarrow \max(\alpha, \text{value})$ 
```

```
      if  $\alpha \geq \beta$  then
```

```
        break //이것이  $\beta$  컷이다.
```

```
    return value
```

```
  else // 최소화 경기자
```

```
    value  $\leftarrow +\infty$ 
```

```
    for each child of node do
```

```
      value  $\leftarrow \min(\text{value}, \text{alphabeta}(\text{child}, \text{depth}-1, \alpha, \beta, \text{TRUE}))$ 
```

```
       $\beta \leftarrow \min(\beta, \text{value})$ 
```

```
      if  $\alpha \geq \beta$  then
```

```
        break //이것이  $\alpha$  컷이다.
```

```
    return value
```

현재 노드의 최대값이 부모 노드의 값(β)보다 커지게
되면 더 이상 탐색할 필요가 없음

현재 노드의 최소값이 부모 노드의 값(α)보다 작으면
되면 더 이상 탐색할 필요가 없음

알파베타 알고리즘

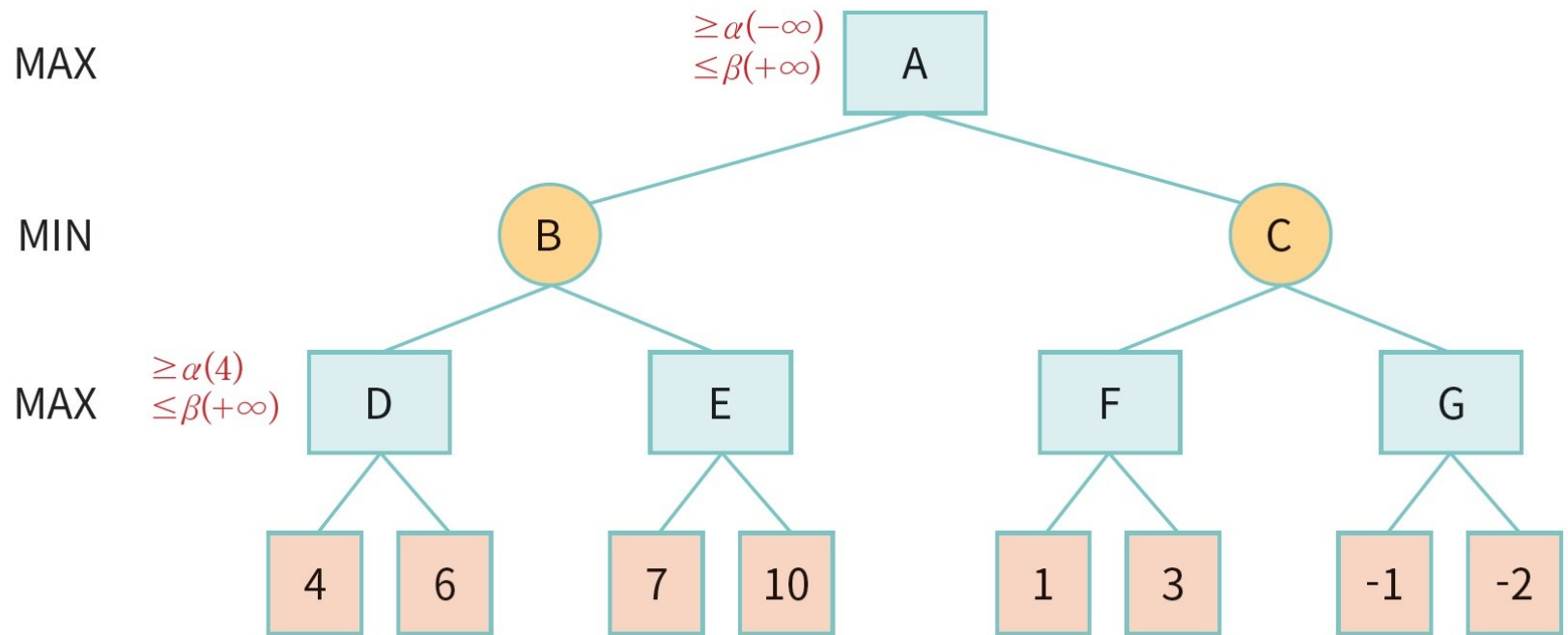


그림 3-7 알파베타 가지치기 알고리즘 I

알파베타 알고리즘

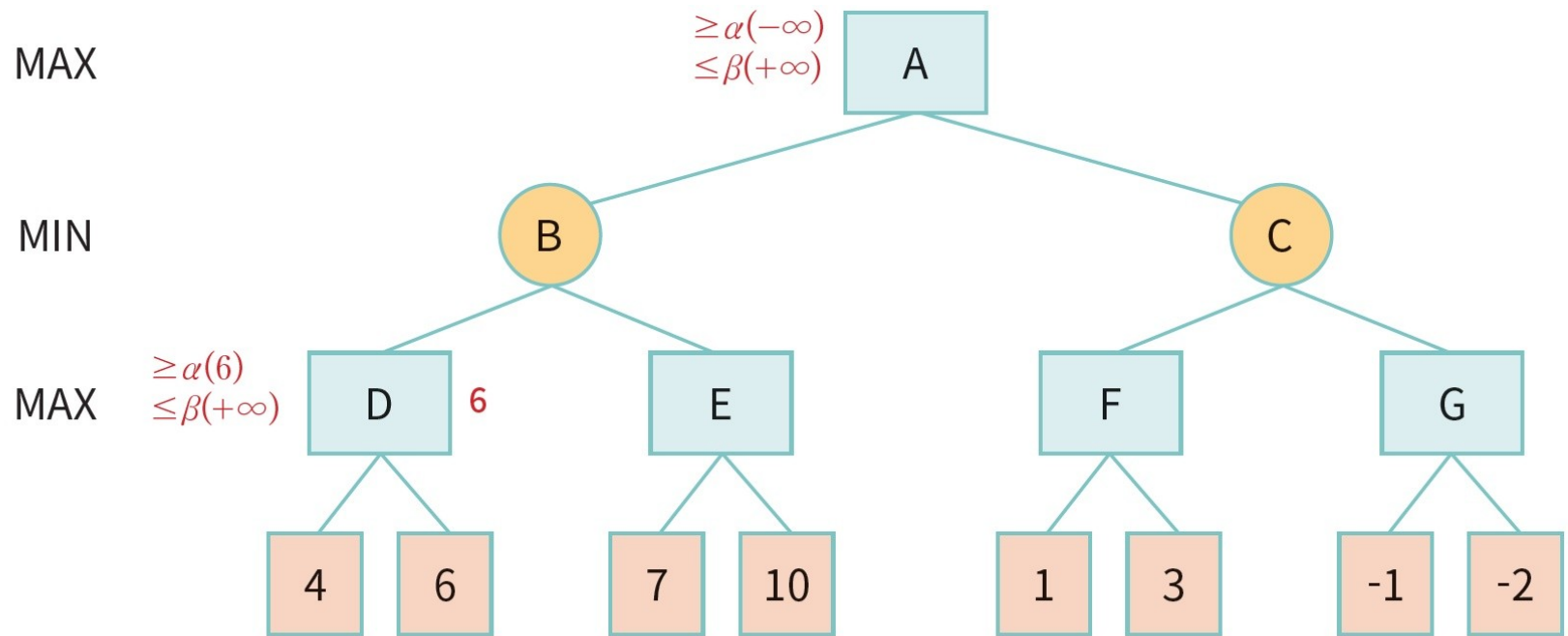


그림 3-8 알파베타 가지치기 알고리즘 II

알파베타 알고리즘

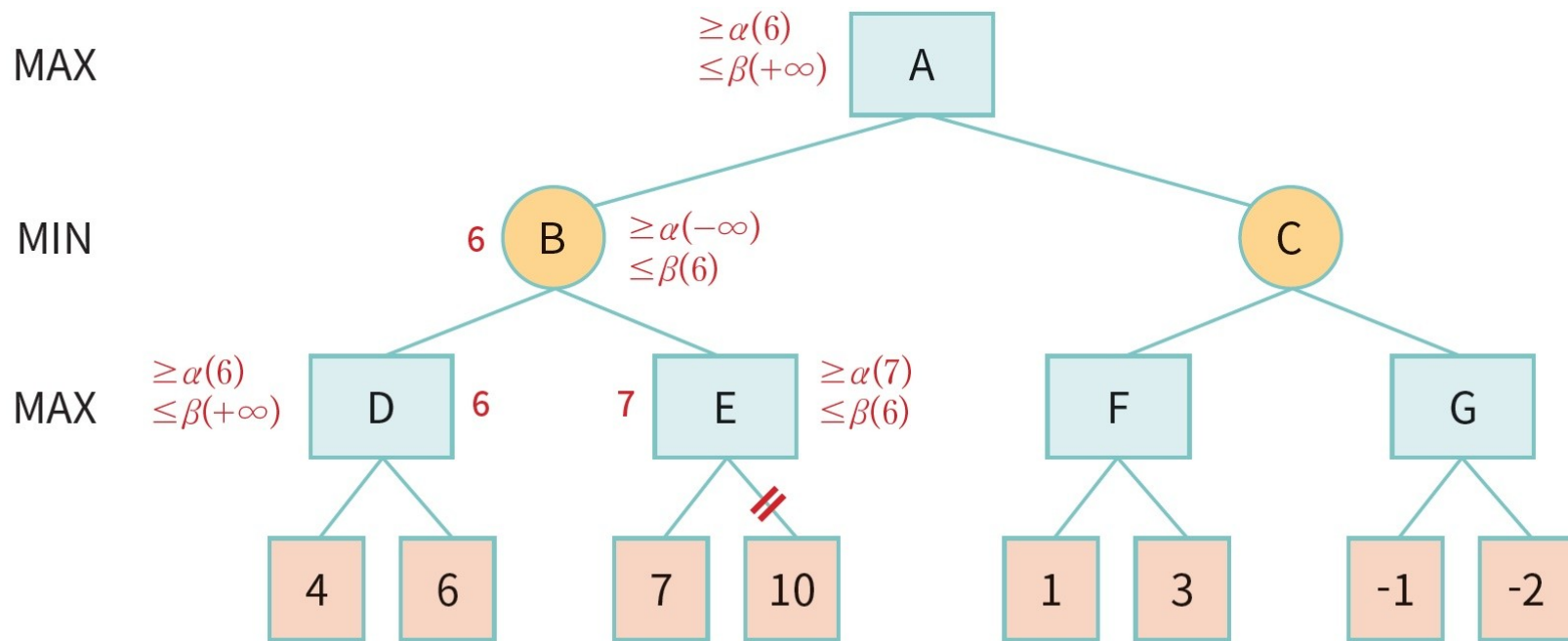


그림 3-9 알파베타 가지치기 알고리즘 III

알파베타 알고리즘

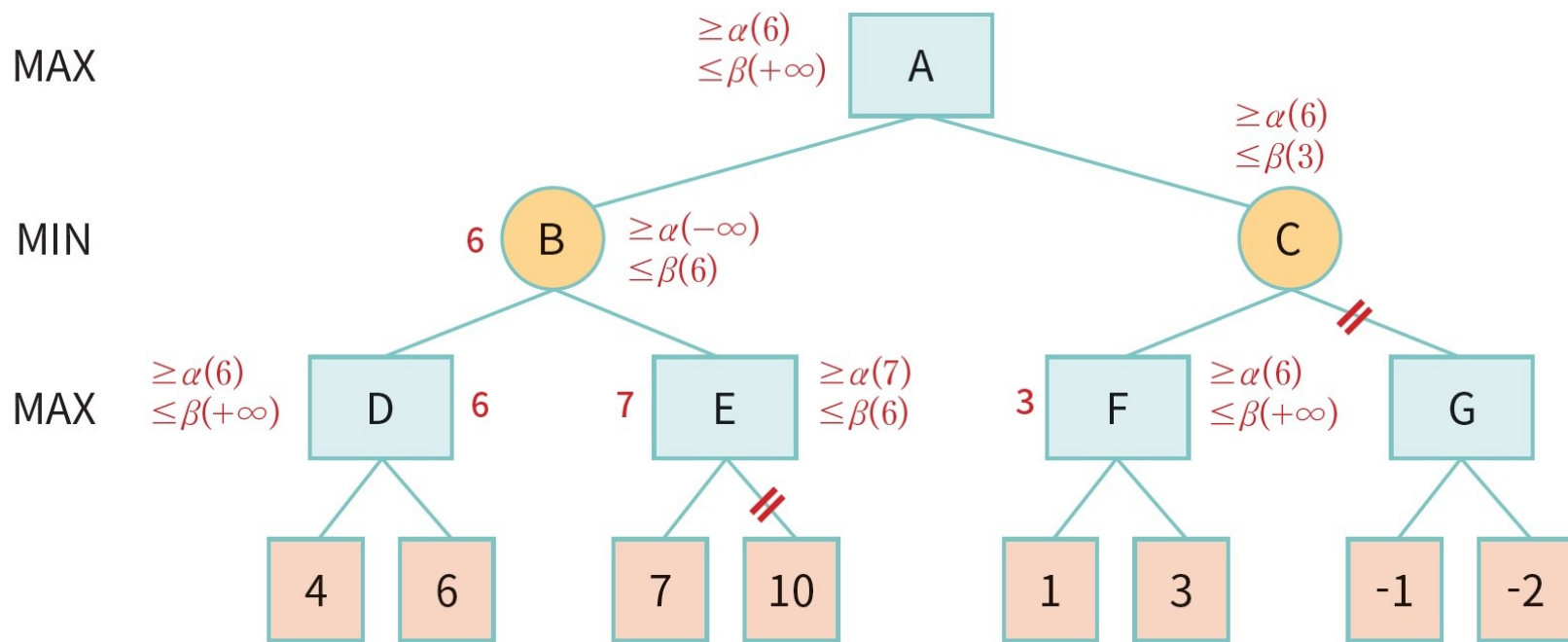


그림 3-10 알파베타 가지치기 알고리즘 IV

불완전한 결정

- 미니맥스 알고리즘은 탐색 공간 전체를 탐색하는 것을 가정한다. 하지만 실제로는 탐색 공간의 크기가 무척 커서 우리는 그렇게 할 수 없다. 실제로는 적당한 시간 안에 다음 수를 결정하여야 한다. 어떻게 하면 될까?
- 이때는 탐색을 끝내야 하는 시간에 도달하면 탐색을 중단하고 탐색 중인 상태에 대하여 휴리스틱 평가 함수(evaluation function)를 적용해야 한다. 즉 비단말 노드이지만 단말 노드에 도달한 것처럼 생각하는 것이다.

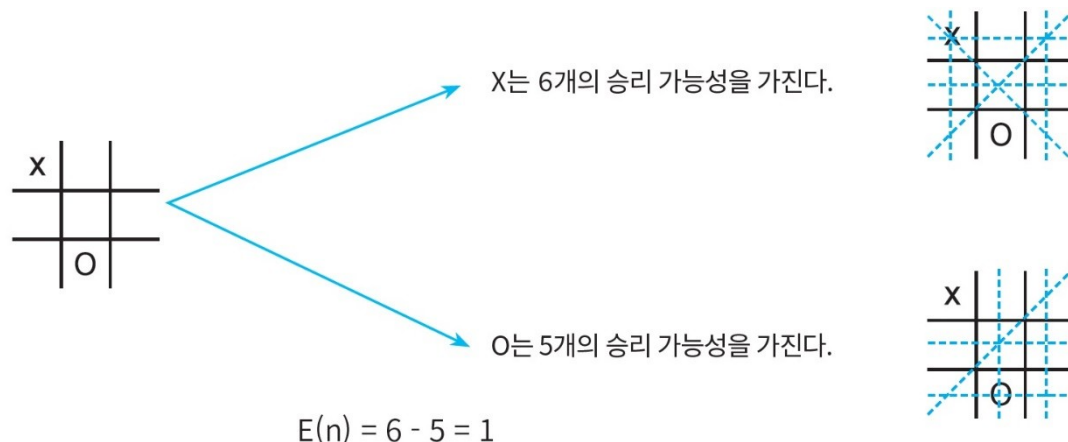


그림 3-11 평가 함수

Summary

- 게임에서는 상대방이 탐색에 영향을 끼친다. 이 경우에는 미니맥스 알고리즘을 사용하여 탐색을 진행할 수 있다. 미니맥스 알고리즘은 상대방이 최선의 수를 둔다고 가정하는 알고리즘이다.
- 두 명의 경기자 MAX와 MIN이 있으며, MAX는 평가 함수값이 최대인 자식 노드를 선택하고 MIN은 평가 함수값이 최소인 자식 노드를 선택한다.
- 탐색 트리의 어떤 부분은 제외하여도 결과에 영향을 주지 않는다. 이것을 알파베타 가지치기(alpha-beta pruning)라고 한다.

Q & A

