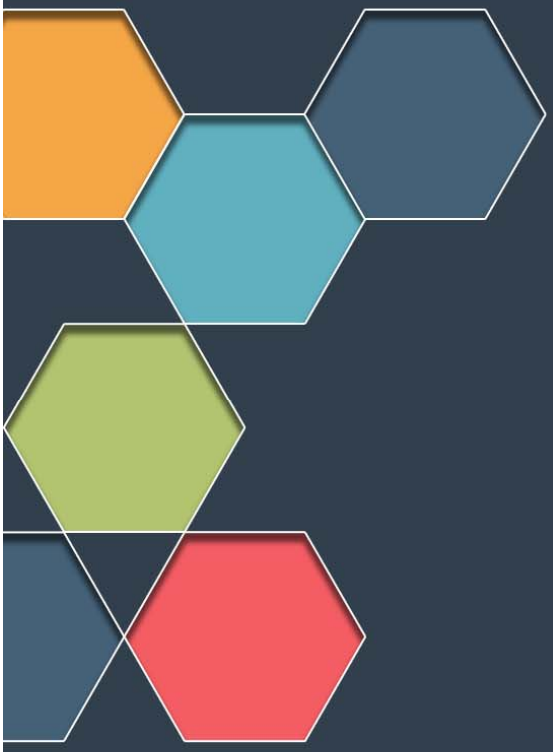


제3장 게임트리

학습 목표

- 미니맥스 알고리즘을 살펴본다.
- 알파베타 가지치기 알고리즘을 이해한다.

이번 장에서 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(zero sum) 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.

게임의 정의

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

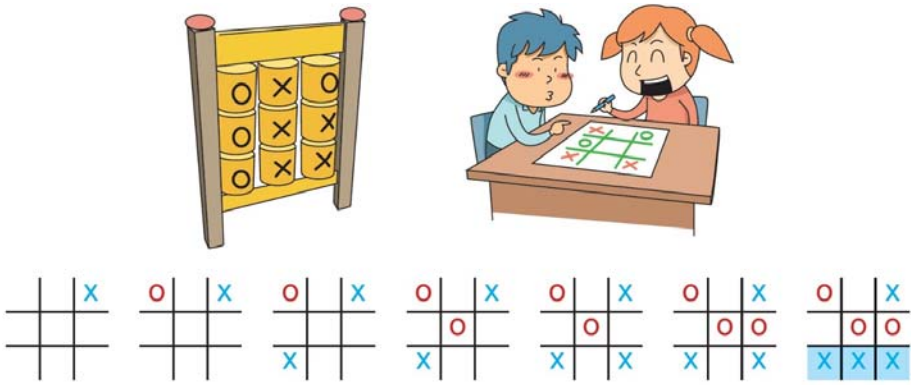


그림 3-1 Tic-Tac-Toe 게임

Tic-Tac-Toe의 게임 트리

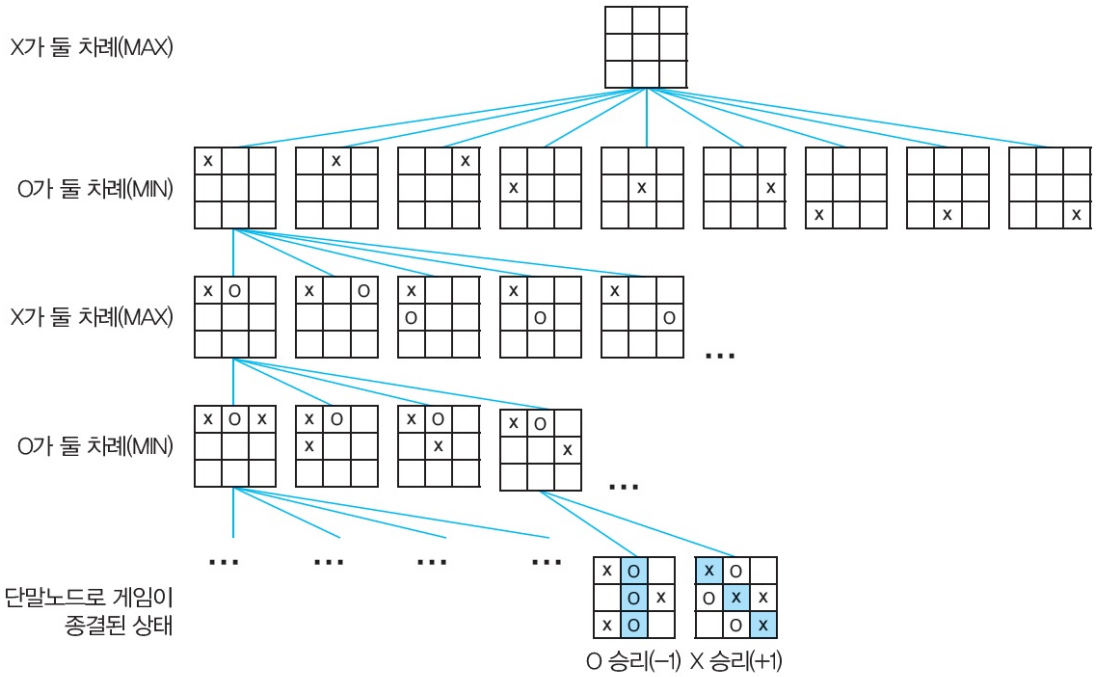


그림 3-2 틱택토 게임의 게임 트리(일부)

Tic-Tac-Toe 게임 트리의 크기

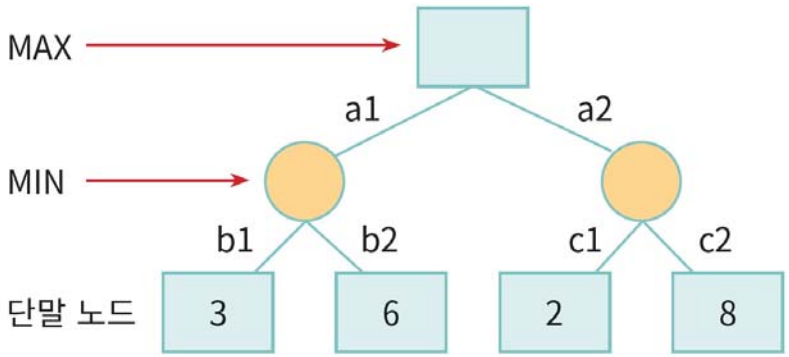
- Tic-Tac-Toe의 게임 트리는 크기가 얼마나 될까?

- Tic-Tac-Toe 게임 보드는 3×3 크기를 가지고 있고 한 곳에 수를 놓으면 다른 사람이 놓을 수 있는 곳은 하나가 줄어들게 된다.

  $9 \times 8 \times 7 \times \ldots \times 1 = 9! = 362{,}880$

02 미니맥스 알고리즘

- 안전하게 하려면 상대방이 최선의 수를 둔다고 생각하면 된다.



미니맥스(minimax) 알고리즘

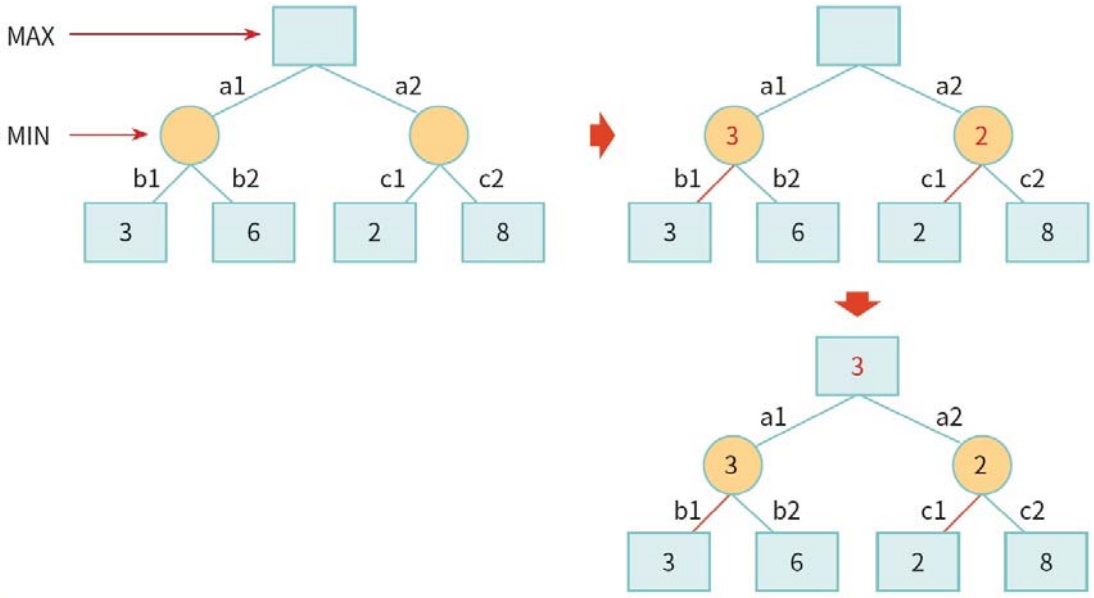


그림 3–4 미니맥스 알고리즘

틱택토 게임에서의 미니맥스

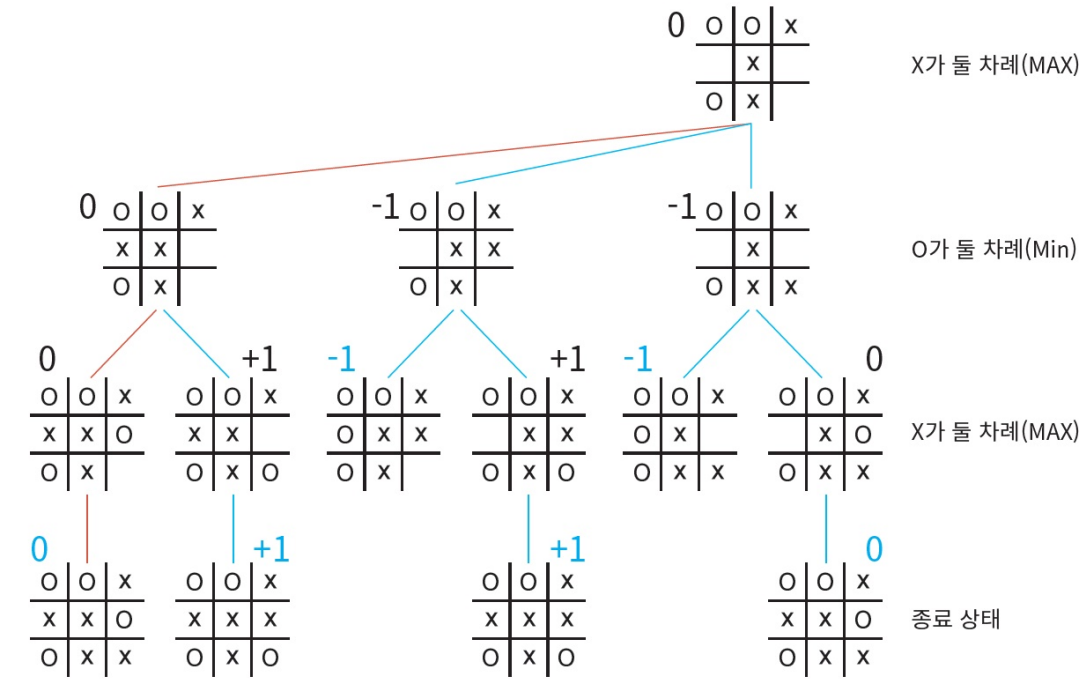
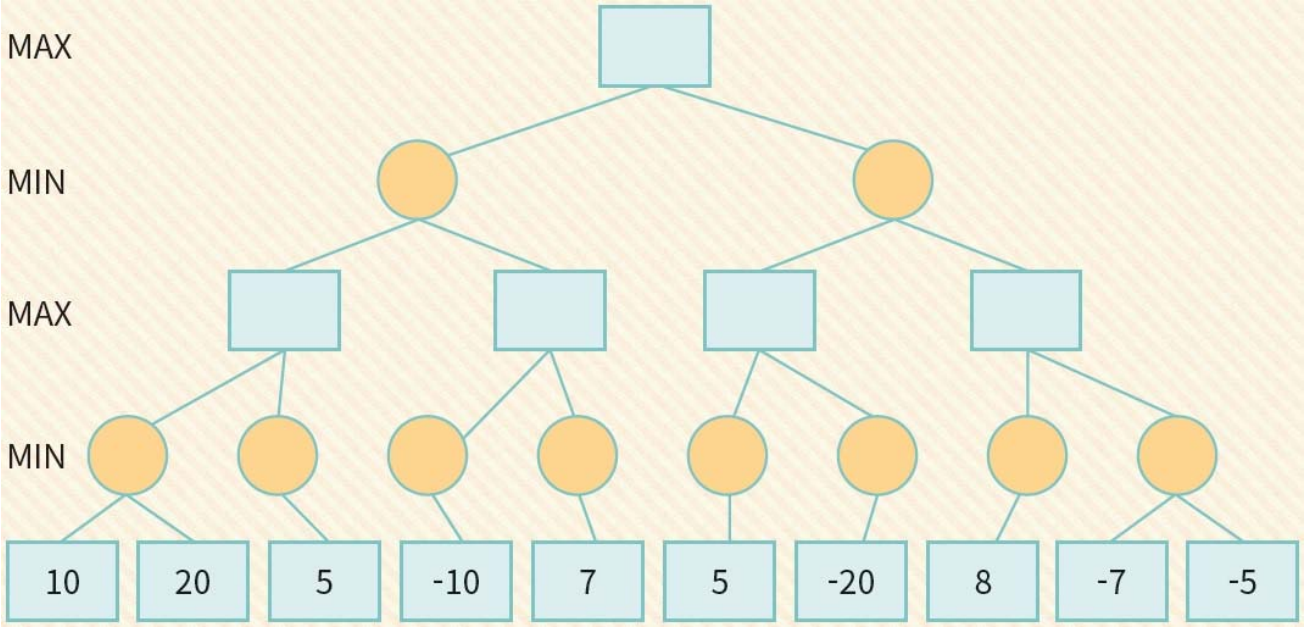


그림 3–5 틱택토 게임에서 미니맥스 알고리즘

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 ← −∞
        for each child of node do
            value ← max(value, minimax(child, depth − 1, FALSE))
        return value
    else   // 최소화 노드
        value ← +∞
        for each child of node do
            value ← min(value, minimax(child, depth − 1, TRUE))
        return value
```

미니맥스 알고리즘의 시간 복잡도

- 미니맥스 알고리즘은 게임 트리에 대하여 완벽한 깊이 우선 탐색을 수행한다. 만약 트리의 최대 깊이가 m이고 각 노드에서의 가능한 수가 b개라면 최대최소 알고리즘의 시간 복잡도는 $O(b^m)$이다.

- 바둑은 경우의 수가 약 316!이다. 이것을 계산해보면 약 10^{761}로 추산된다. 전체 우주는 약 10^{80}개의 원자 만을 포함하는 것으로 추정된다.

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')
```

```
# 미니맥스 알고리즘을 구현한다.
# 이 함수는 순환적으로 호출된다.
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

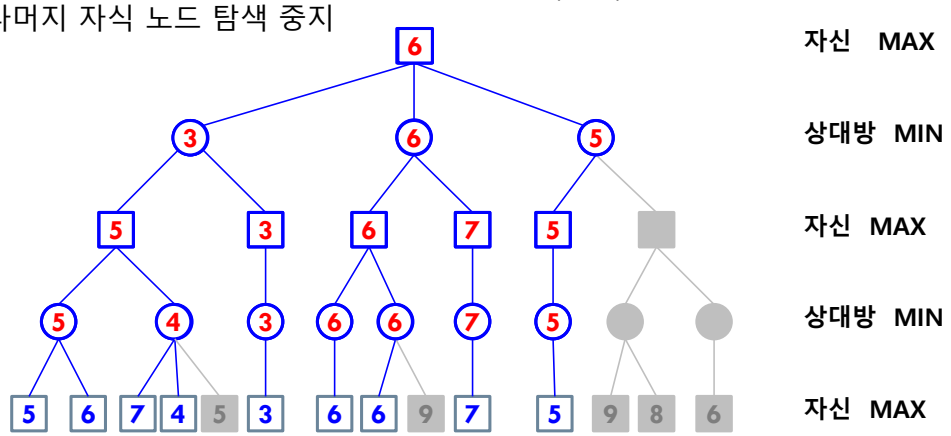

```
-------------
|  ||  ||  |
-------------
|  ||  ||  |
-------------
|  ||  ||  |
-------------

-------------
| X||  ||  |
-------------
|  ||  ||  |
-------------
|  ||  ||  |
-------------

-------------
| X||  ||  |
-------------
|  ||O||  |
-------------
|  ||  ||  |
-------------
...
-------------
| X || X || O |
-------------
| O || O || X |
-------------
| X || O || X |
-------------
비겼습니다!
```

Lab

- minimax 프로그램을 인간과 컴퓨터가 대결하는 것으로 변경하라.

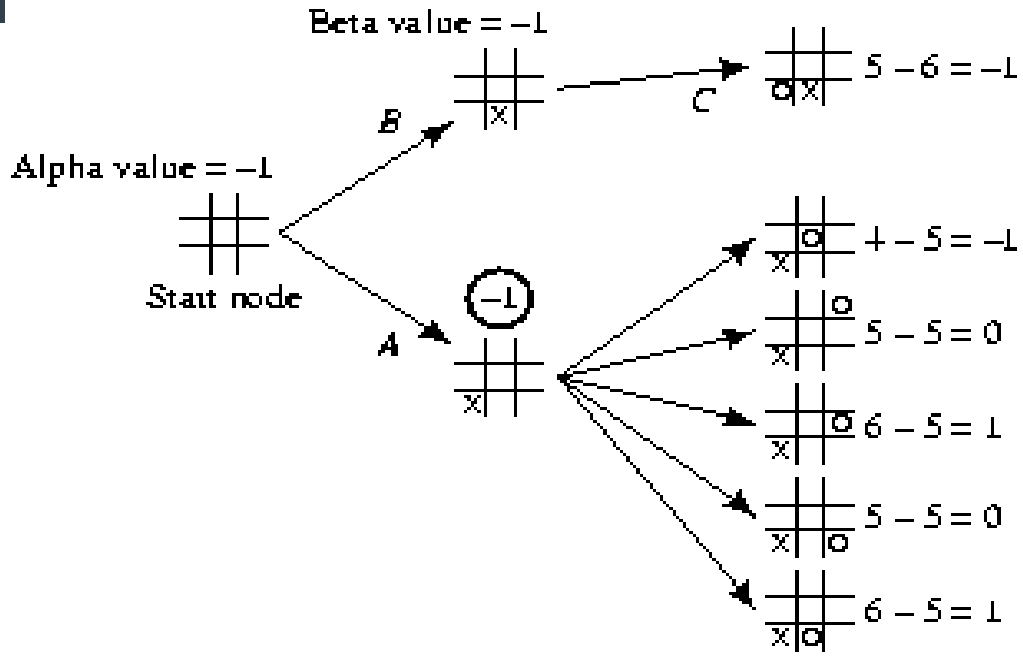
α-β 가지치기 (prunning)

- 검토해 볼 필요가 없는 부분을 탐색하지 않도록 하는 기법
- 깊이 우선 탐색으로 제한 깊이까지 탐색을 하면서, MAX 노드와 MIN 노드의 값 결정
  - **α-자르기(cut-off) :** MIN 노드의 현재값이 부모노드(MAX)의 현재 값보다 작거나 같으면, 나머지 자식 노드 탐색 중지
  - **β-자르기 :** MAX 노드의 현재값이 부모노드(MIN)의 현재 값보다 같거나 크면, 나머지 자식 노드 탐색 중지



간단한 형태의 α-β 가지치기 예

알파 베타 방법 (Alpha-Beta Procedure)

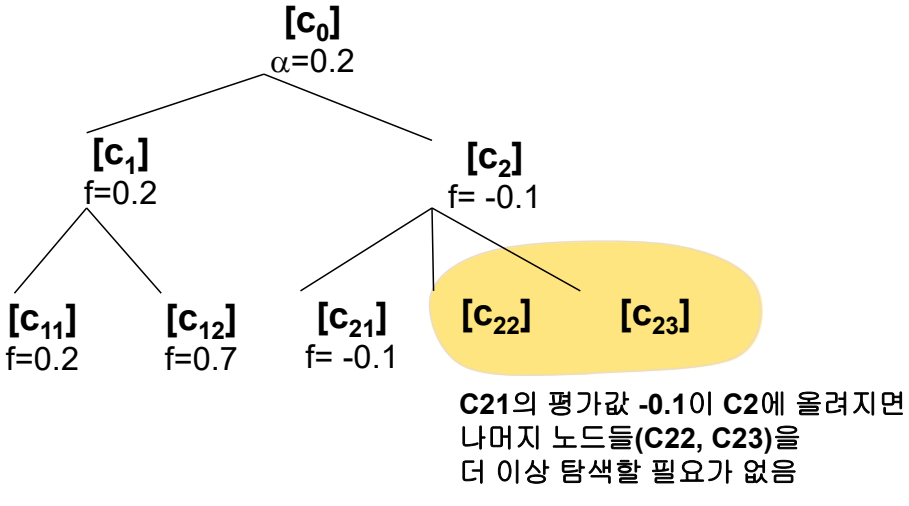


알파 베타 방법

- 알파값과 베타값을 기억하면서 절단을 수행해 가는 모든 과정
- MAX 노드의 알파값 :
  자식 노드의 전달값 중 현재까지 가장 큰 값
- MIN 노드의 베타값 :
  자식 노드의 전달값 중 현재까지 가장 작은 값
- 알파 절단 규칙 :
  조상 MAX 노드의 알파값보다 작거나 같은 베타값을 갖는 MIN 노드 아래에서는 탐색을 중단.
- 베타 절단 규칙 :
  조상 MIN 노드의 베타값보다 크거나 같은 알파값을 갖는 MAX 노드 아래에서는 탐색을 중단.

알파베타 가지치기

- 최대화 노드에서 가능한 최대의 값(알파 α)과 최소화의 노드에서 가능한 최소의 값(베타 β)를 사용한 게임 탐색법
- 기본적으로 DFS로 탐색 진행



C21의 평가값 -0.1이 C2에 올려지면
나머지 노드들(C22, C23)을
더 이상 탐색할 필요가 없음

04 알파베타 가지치기

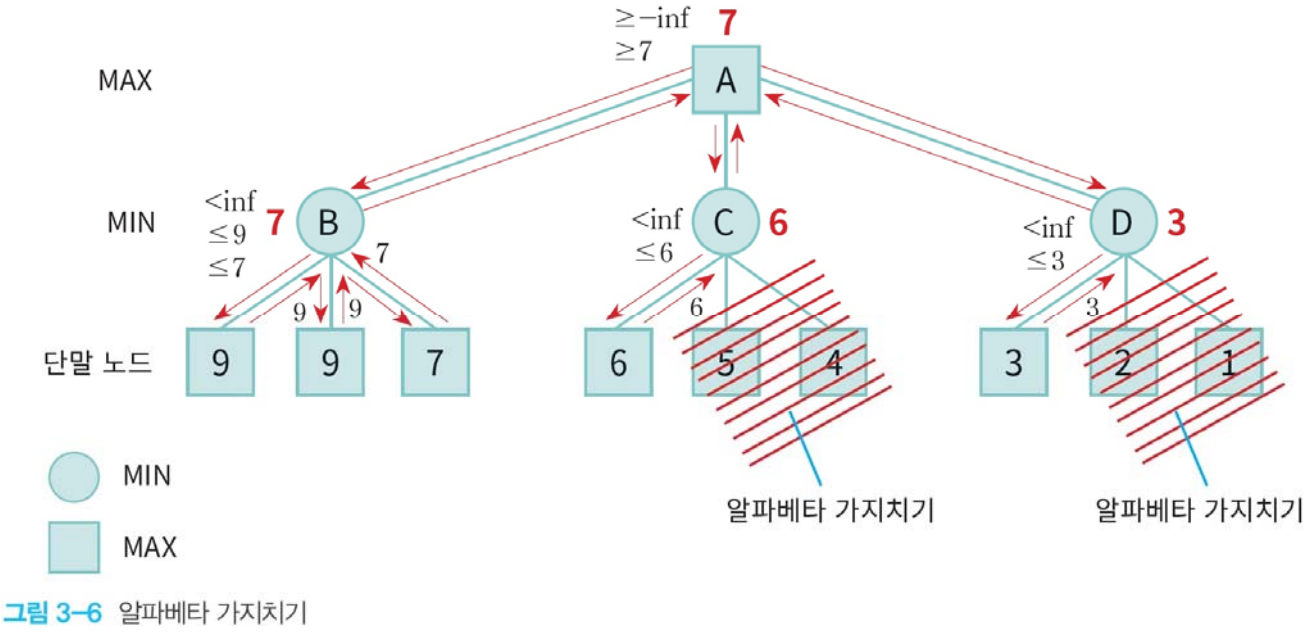


그림 3–6 알파베타 가지치기

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, α, β, maxPlayer)
    if depth == 0 or node가 단말 노드 then
        return node의 휴리스틱 값
    if maxPlayer then              // 최대화 경기자
        value ← −∞
        for each child of node do
            value ← max(value, alphabeta(child, depth−1, α, β, FALSE))
            α ← max(α, value)
            if α ≥ β then
                break //이것이 β 컷이다.
        return value
    else                           // 최소화 경기자
        value ← +∞
        for each child of node do
            value ← min(value, alphabeta(child, depth−1, α, β, TRUE))
            β ← min(β, value)
            if α ≥ β then
                break //이것이 α 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

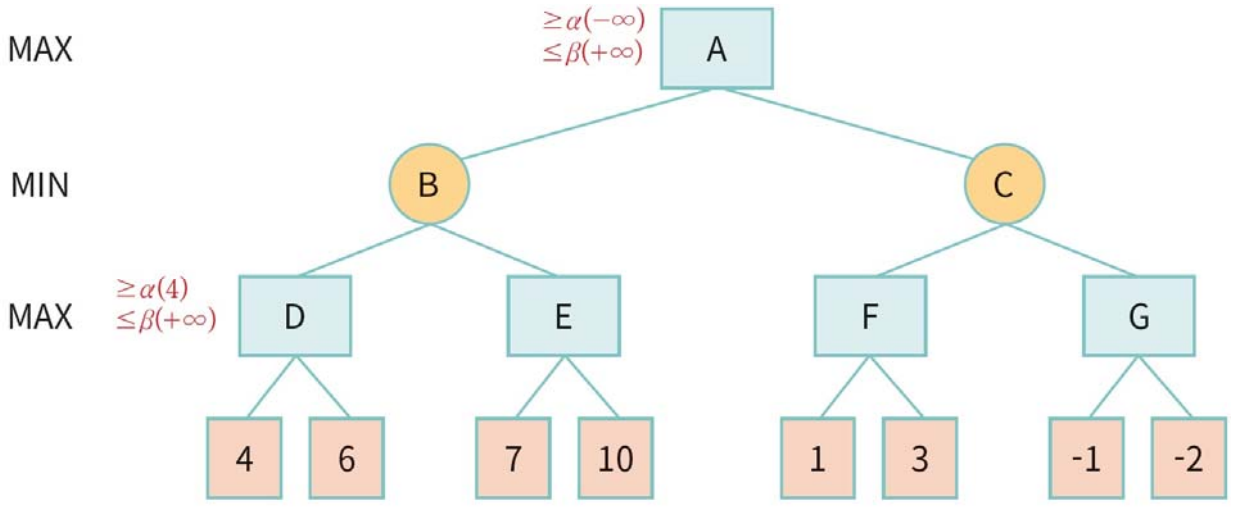


그림 3-7 알파베타 가지치기 알고리즘 I

알파베타 알고리즘

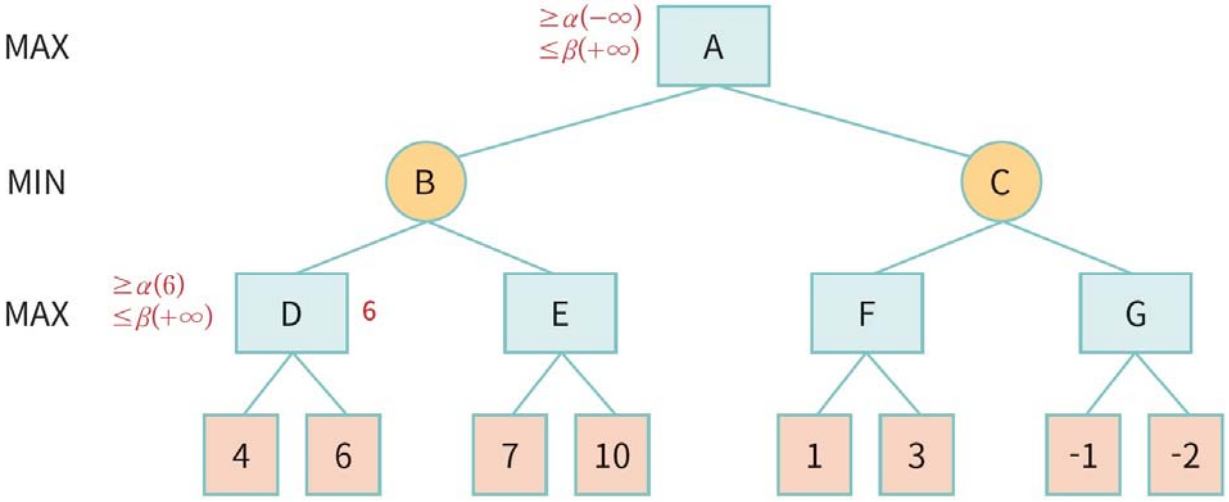


그림 3-8 알파베타 가지치기 알고리즘 II

알파베타 알고리즘

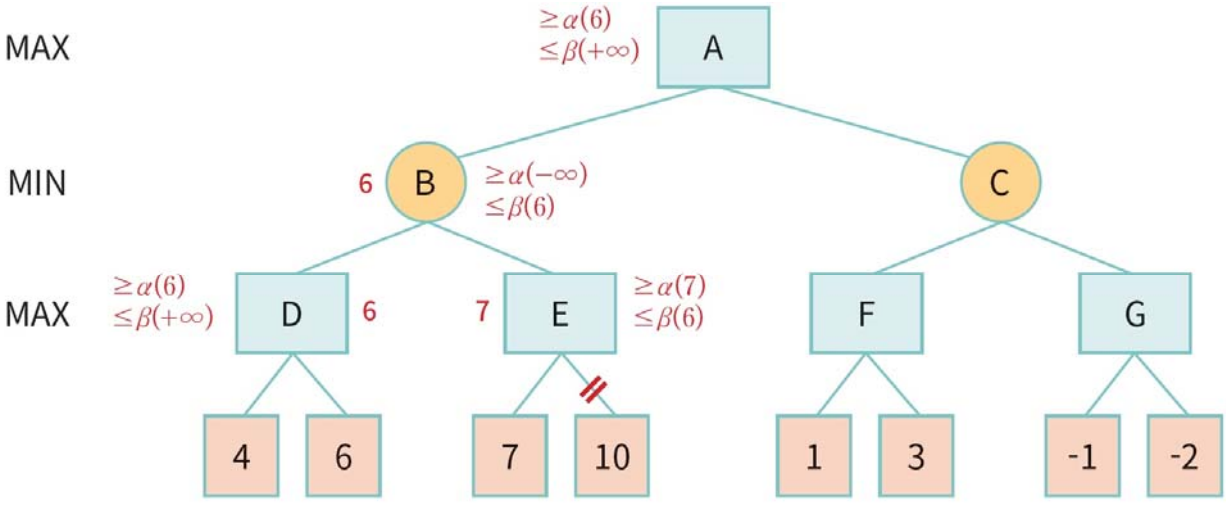


그림 3-9 알파베타 가지치기 알고리즘 III

알파베타 알고리즘

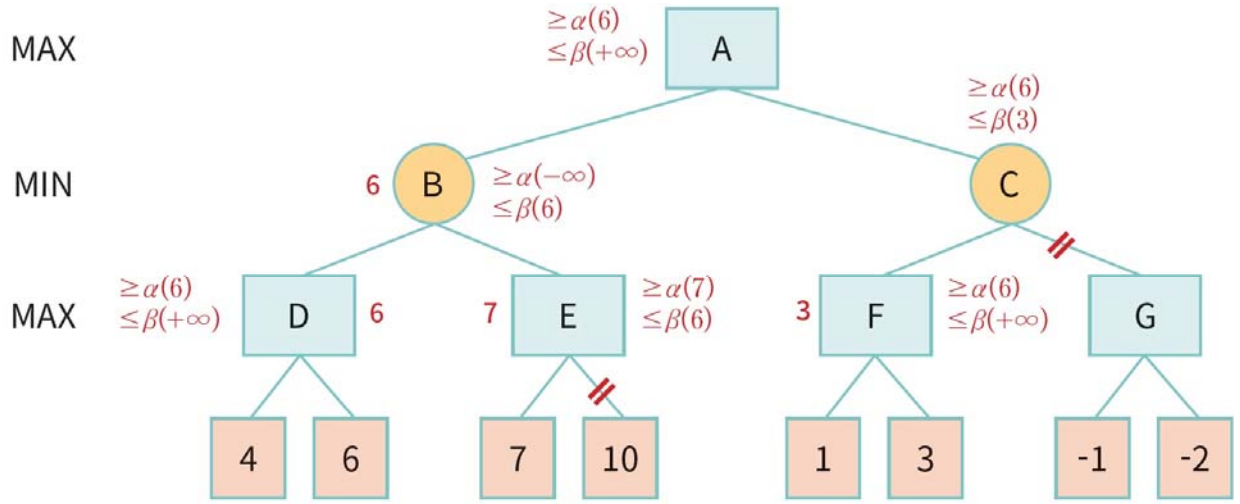
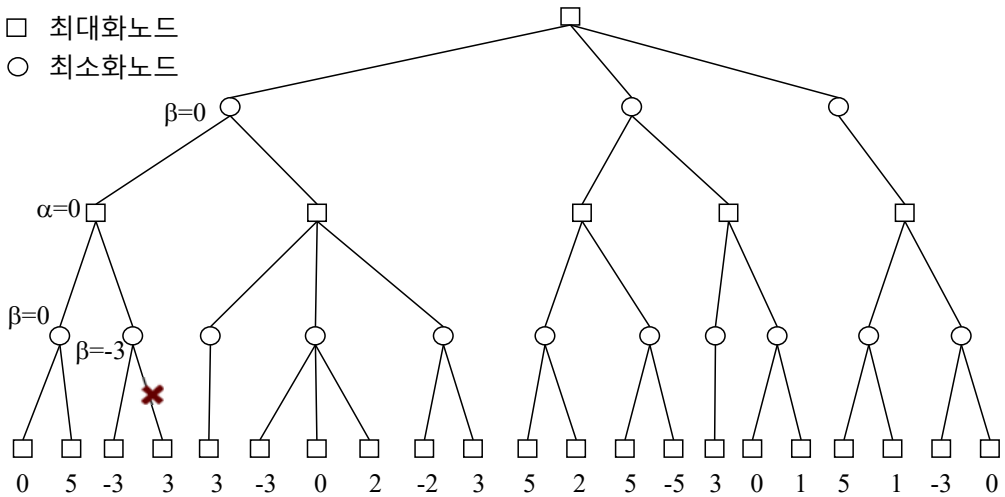


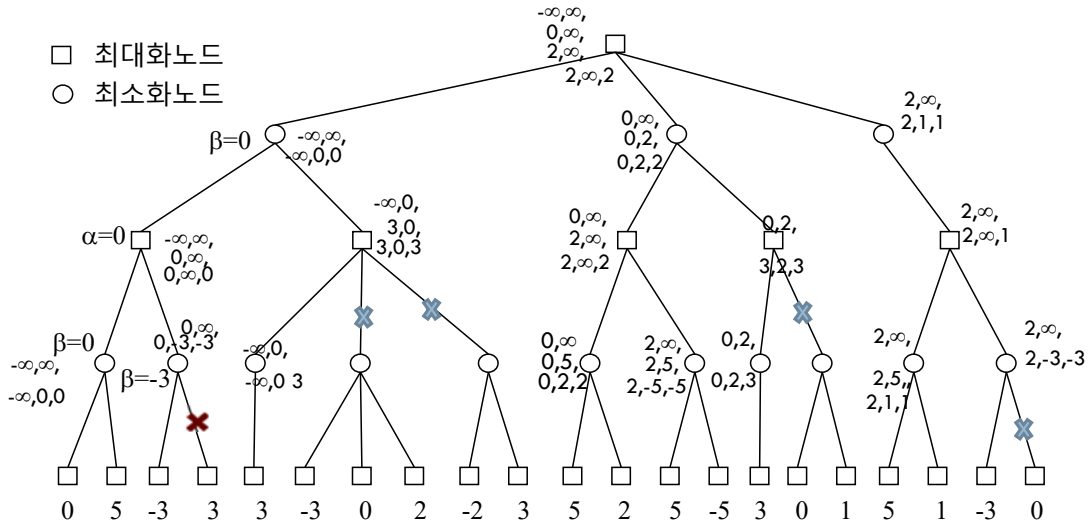
그림 3–10 알파베타 가지치기 알고리즘 IV

가지치기가 일어나는 법칙:

1. 어떤 최소화노드의 베타값이 자신보다 상위(선조노드)에 있는 어떤 최대화 노드의 알파값보다 작거나 같을 때, 이 최소화 노드는 가지치기 된다.
2. 어떤 최대화노드의 알파값이 자신보다 상위(선조노드)에 있는 어떤 최소화 노드의 베타값보다 크거나 같을 때, 이 최대화 노드는 가지치기 된다.
3. 최상위의 최대화노드의 알파값은 최종적으로 올려진 값(backed-up value)로 주어진다.



가지치기가 일어나는 법칙:
1. 어떤 최소화노드의 베타값이 자신보다 상위(선조노드)에 있는 어떤 최대화 노드의 알파값보다 작거나 같을 때, 이 최소화 노드는 가지치기 된다.
2. 어떤 최대화노드의 알파값이 자신보다 상위(선조노드)에 있는 어떤 최소화 노드의 베타값보다 크거나 같을 때, 이 최대화 노드는 가지치기 된다.
3. 최상위의 최대화노드의 알파값은 최종적으로 올려진 값(backed-up value)로 주어진다.



(알파값, 베타값, 리턴값)

05 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.

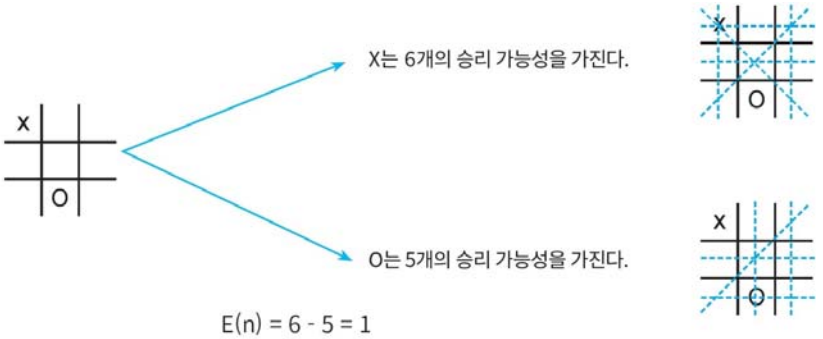


그림 3-11 평가 함수

- 본문의 미니맥스 버전의 틱택토 프로그램을 알파베타 가지치기 버전으로 변경하여 테스트하라.

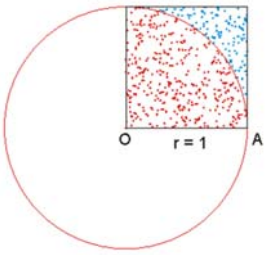
Summary

- 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.

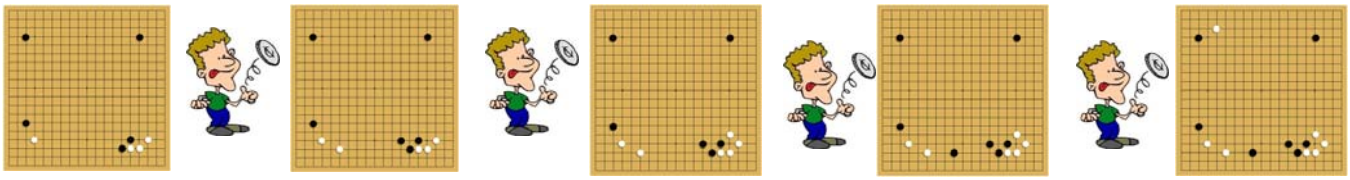
게임에서의 탐색

❖ **몬테카를로 시뮬레이션 (Monte Carlo Simulation)**

- MCTS 4단계 중 평가(Play out) 단계
- 특정 **확률 분포**로부터 **무작위 표본**(random sample)을 생성하고,
- 이 표본에 따라 **행동**을 하는 과정을 반복하여 결과를 확인하고,
- 이러한 결과확인 과정을 반복하여 최종 결정을 하는 것

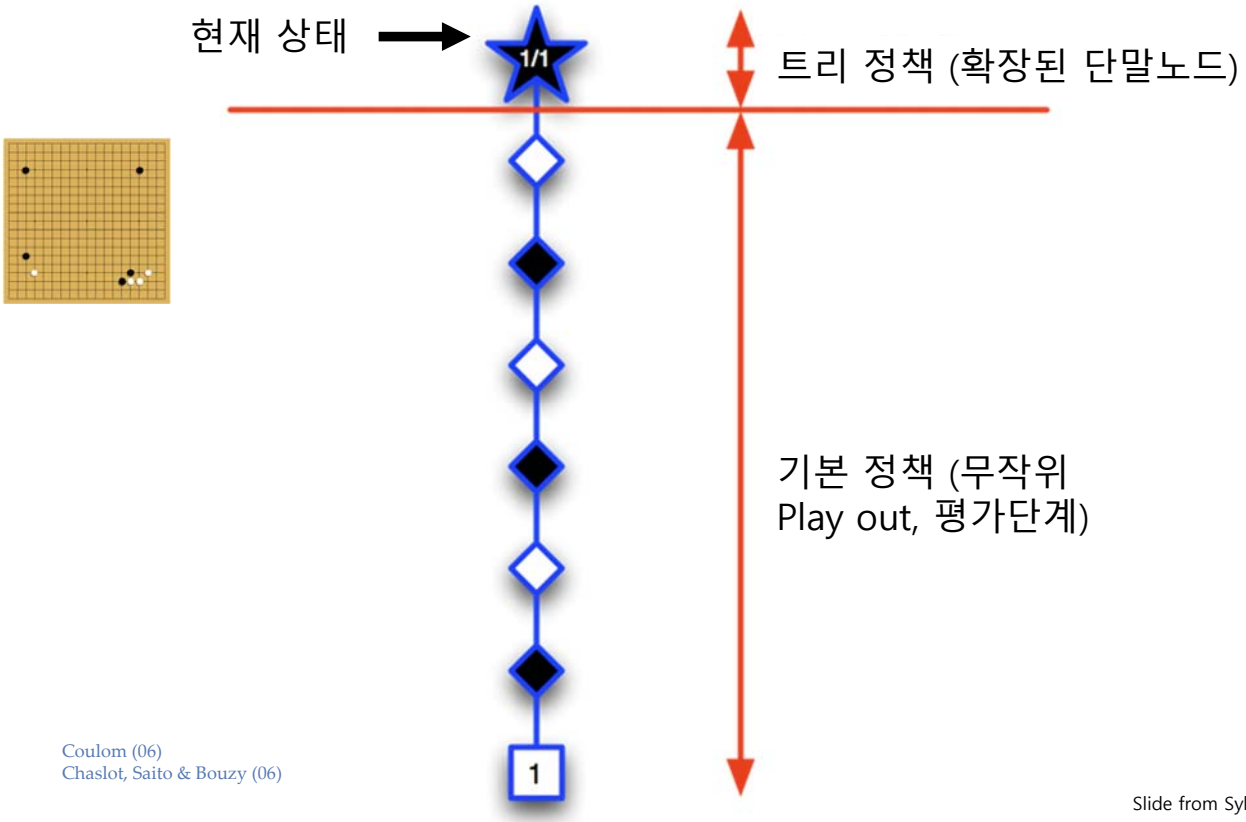


$$\frac{\text{원 안의 샘플 개수}}{\text{전체 샘플의 개수}} \rightarrow \frac{\pi}{4}$$



몬테카를로 트리 탐색

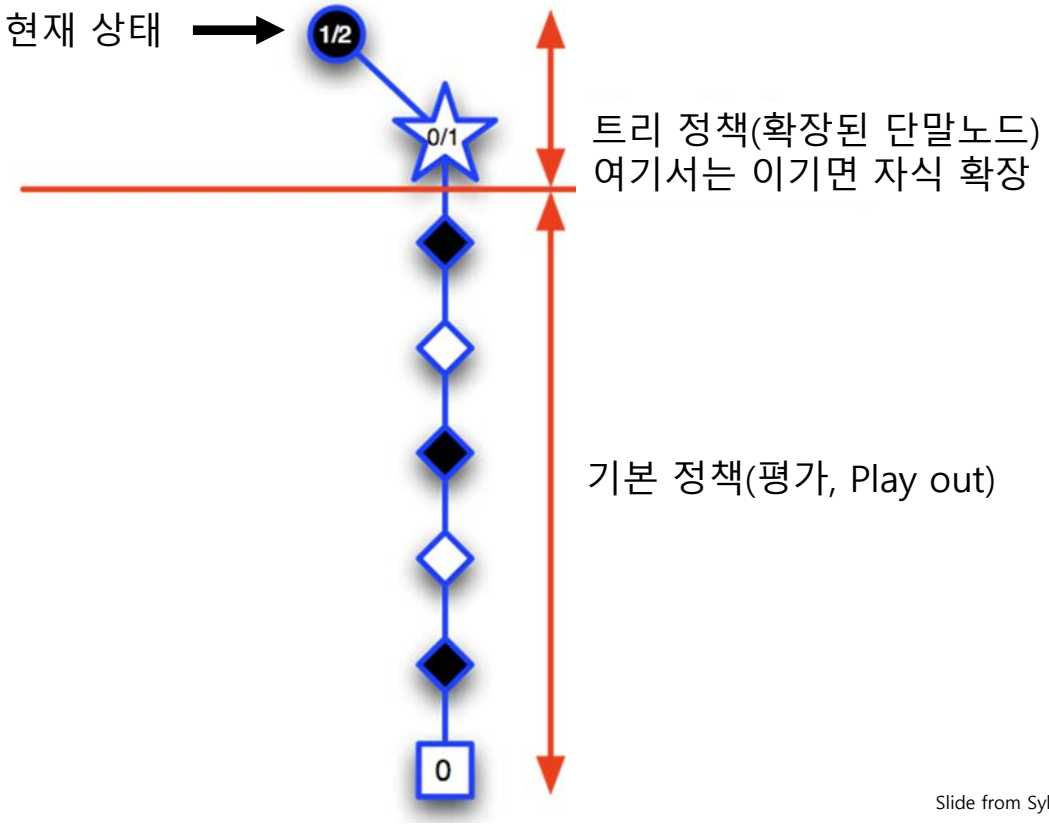
❖ **몬테카를로 트리 탐색(Monte Carlo Tree Search, MCTS)**



Coulom (06)
Chaslot, Saito & Bouzy (06)

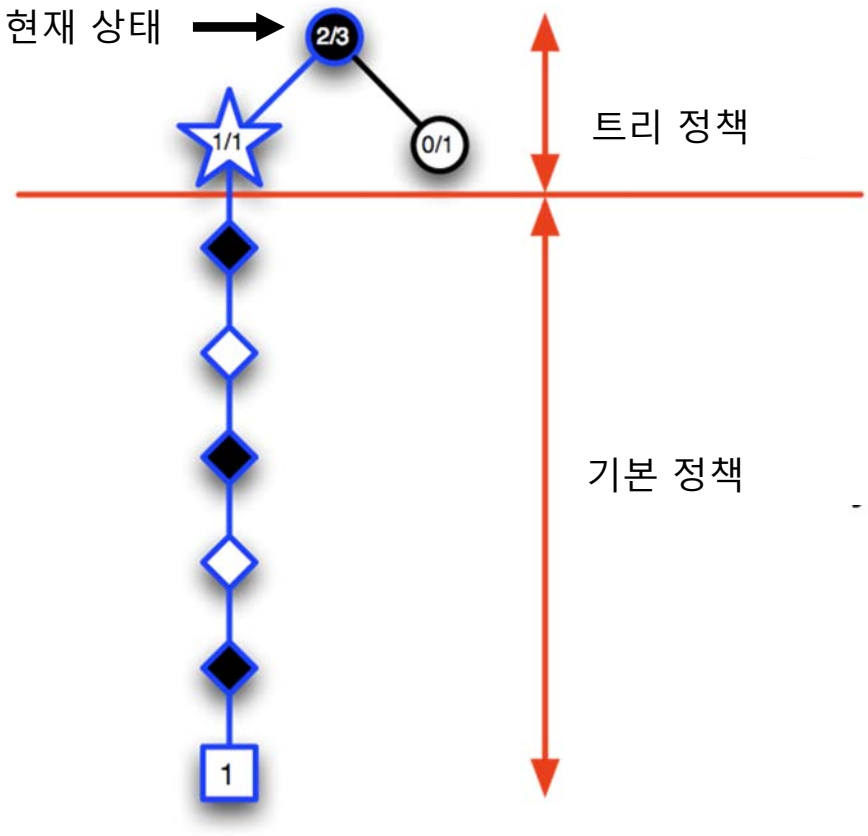
Slide from Sylvain Gelly

몬테카를로 트리 탐색



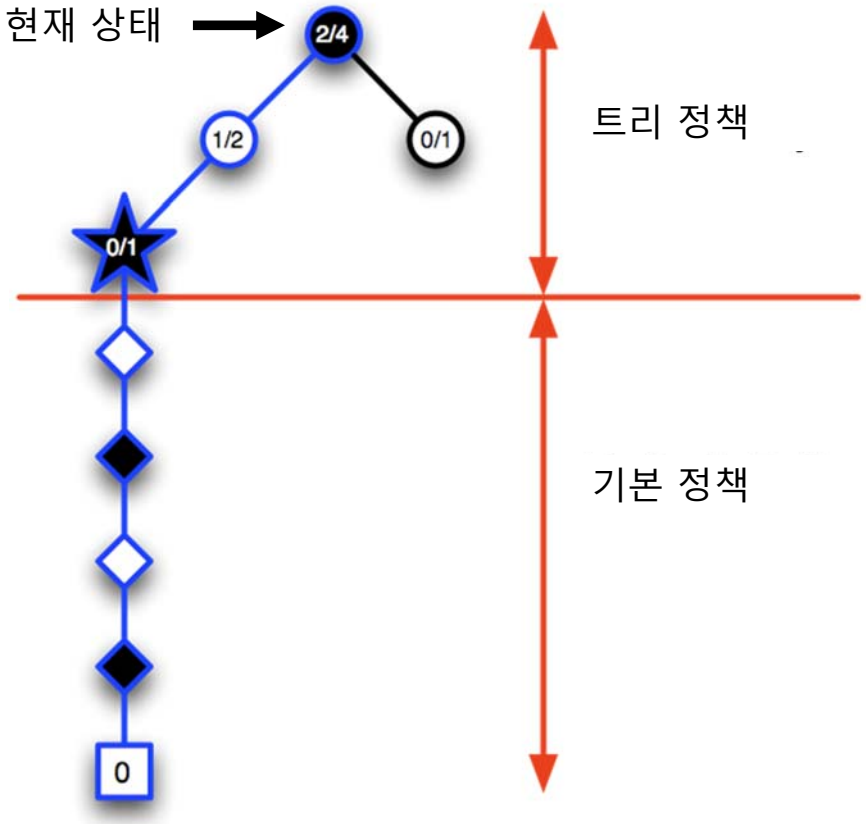
Slide from Sylvain Gelly

몬테카를로 트리 탐색



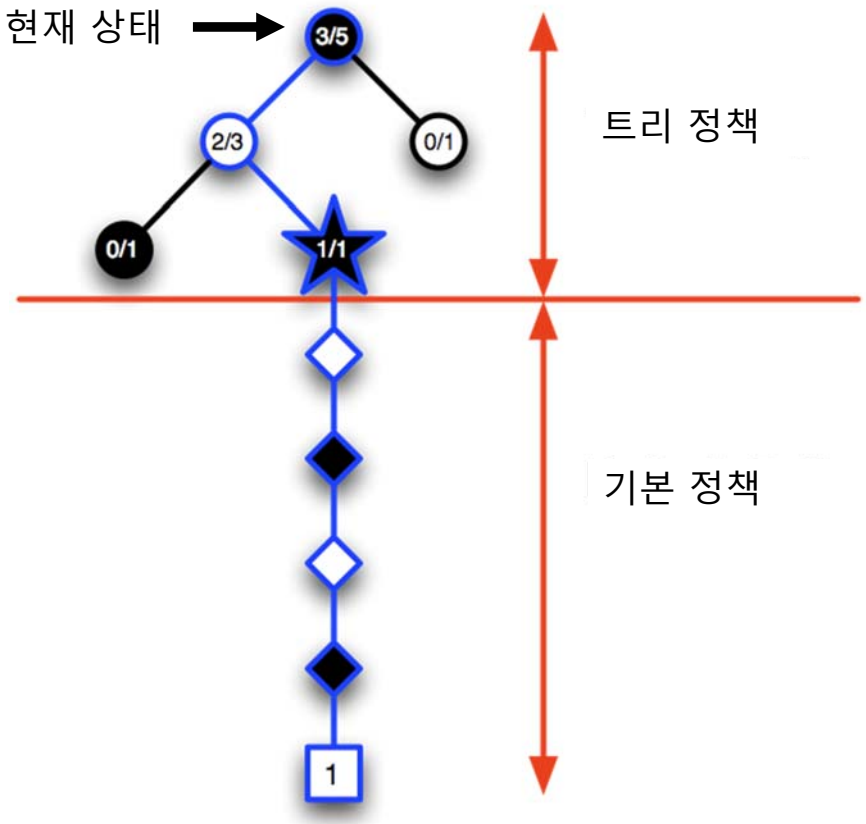
Slide from Sylvain Gelly

몬테카를로 트리 탐색



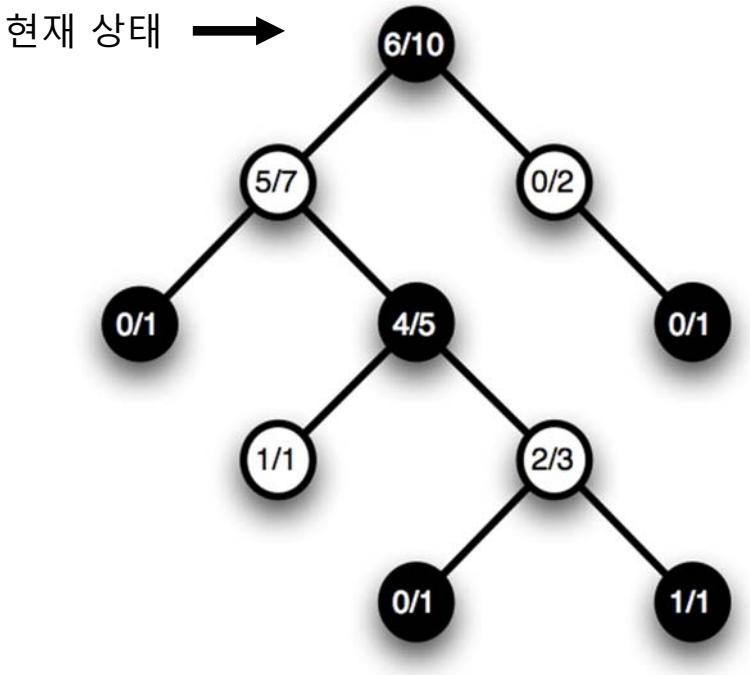
Slide from Sylvain Gelly

몬테카를로 트리 탐색



Slide from Sylvain Gelly

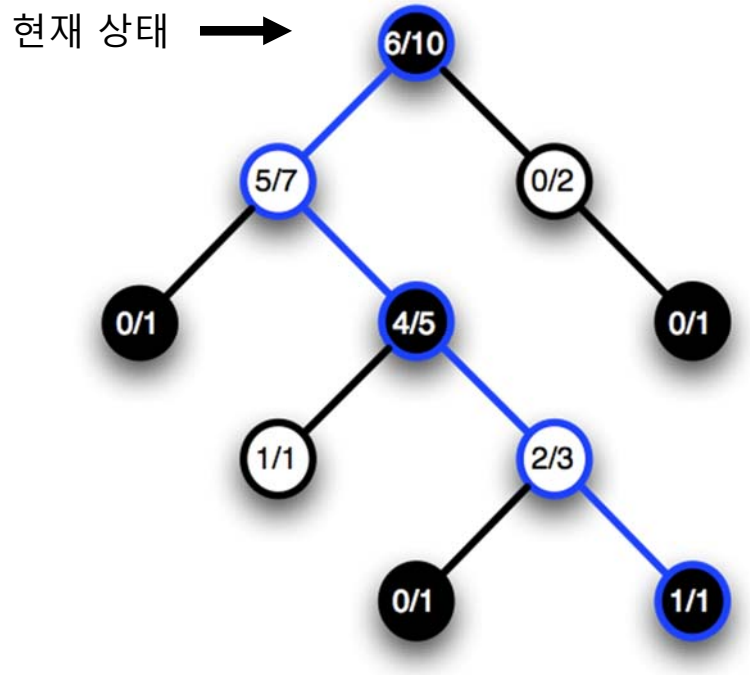
몬테카를로 트리 탐색



Kocsis & Szepesvari (06)

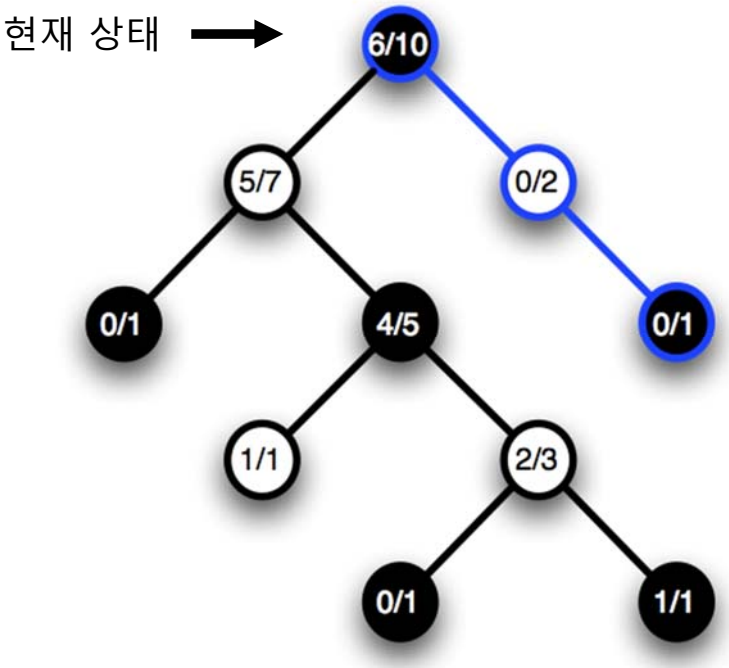
몬테카를로 트리 탐색

Exploitation: 기존 승률이 높은쪽 선택



몬테카를로 트리 탐색

Exploration: 시도가 덜된 노드를 선택



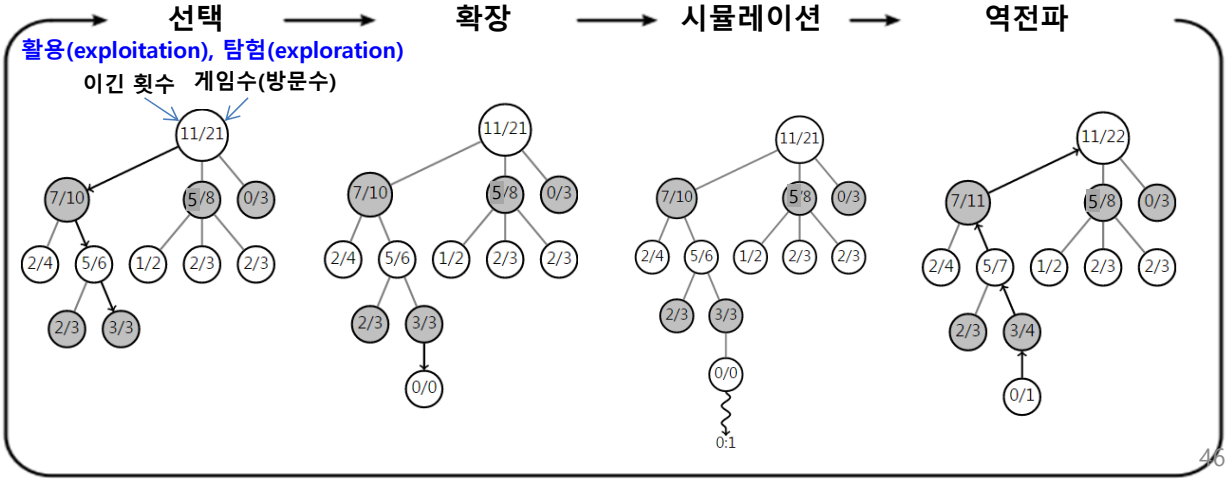
Slide from Sylvain Gelly

게임에서의 탐색

❖ 몬테카를로 트리 탐색(Monte Carlo Tree Search, MCTS)

- 탐색 공간(search space)을 **무작위 표본추출**(random sampling)을 하면서, **탐색트리**를 **확장**하여 가장 좋아 보이는 것을 선택하는 휴리스틱 탐색 방법
- 4개 단계를 반복하여 **시간이 허용하는 동안** 트리 확장 및 시뮬레이션

  선택(selection)
  → **확장(expansion) : 일정조건(예, 시도횟수)을 만족하는 수에 대한 노드를 만듦**
  → **시뮬레이션(simulation) : 몬테카를로 시뮬레이션**
  → **역전파(back propagation) : 단말에서 루트까지 승패 결과를 역방향으로 갱신**



게임에서의 탐색

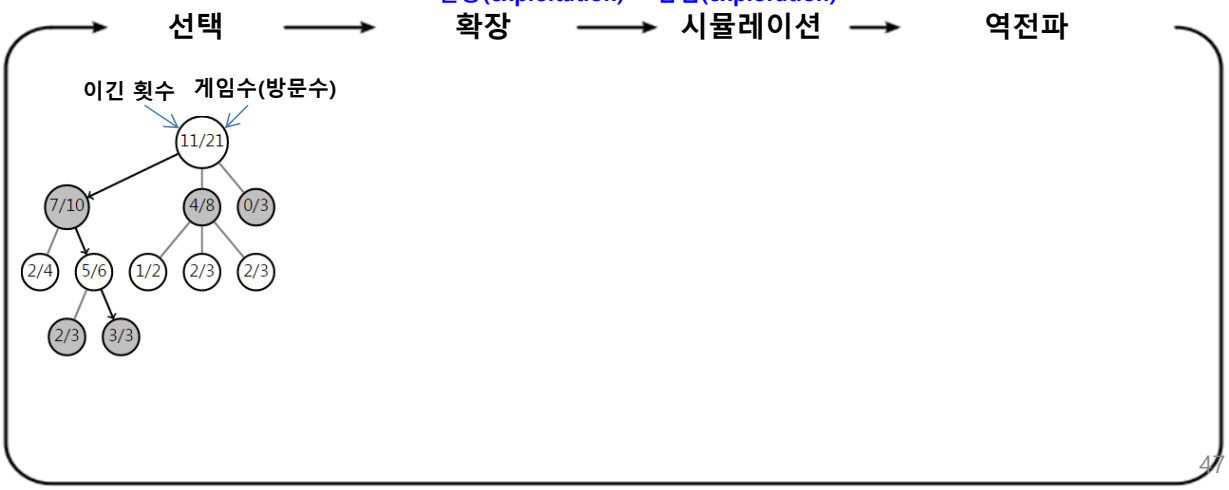
❖ 몬테카를로 트리 탐색 – cont.

- **선택(selection)** : **트리 정책(tree policy)** 적용
  - 루트노드에서 시작
  - 정책에 따라 자식 노드를 선택하여 단말노드까지 내려 감
    - **승률**과 **노드 방문횟수** 고려하여 선택
    - **UCB**(Upper Confidence Bound) **정책 :** UCB가 큰 것 선택

$$\frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$$

활용(exploitation) 탐험(exploration)

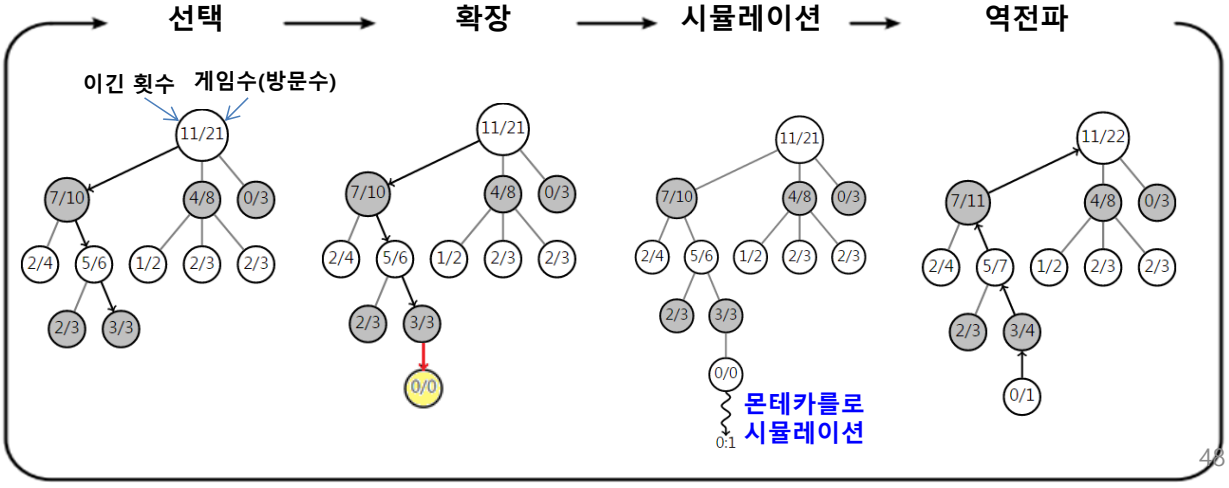
v : 부모노드, v' : 자식노드
$N(v')$: 방문 횟수
$Q(v')$: 점수 (이긴 횟수)



게임에서의 탐색

❖ 몬테카를로 트리 탐색 – cont.

- **확장(expansion: exploitation)**
  - 단말노드에서 **트리 정책**에 따라 노드 추가
    - 예. **일정 횟수이상 시도된 수**(move)가 있으면 해당 수에 대한 **노드 추가**
- **시뮬레이션(simulation: exploration)**
  - **기본 정책(default policy)**에 의한 **몬테카를로 시뮬레이션** 적용
  - **무작위 선택**(random moves) 또는 **약간 똑똑한 방법**으로 **게임 끝날 때까지 진행**
- **역전파(backpropagation)**
  - 단말 노드에서 루트 노드까지 올라가면서 승점 반영

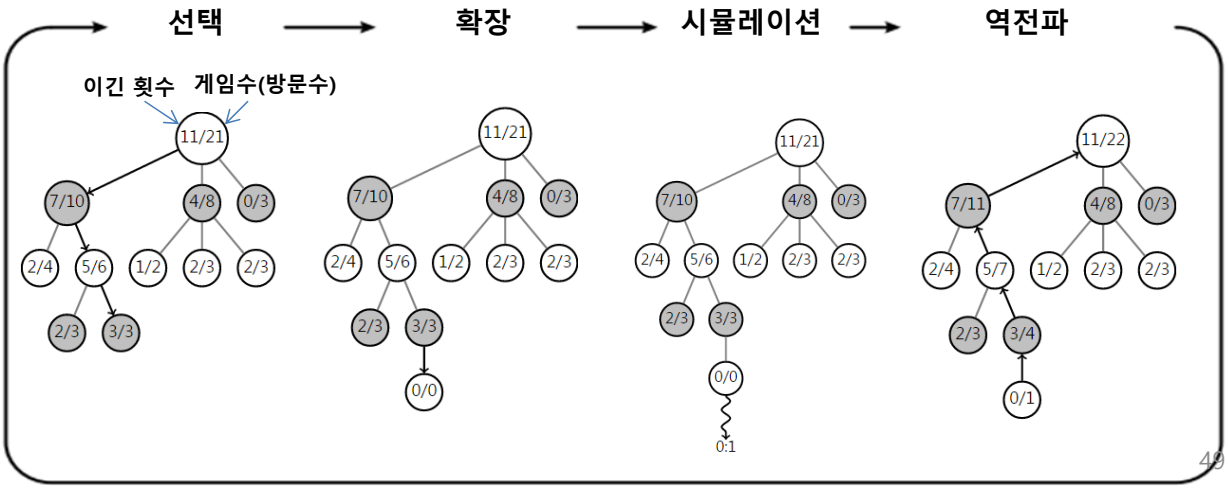


몬테카를로 트리 탐색

❖ 몬테카를로 트리 탐색 – cont.

- **동작 선택 방법**
  - 가장 **승률**이 높은, 루트의 자식 노드 선택
  - 가장 **빈번**하게 방문한, 루트의 자식 노드 선택
  - **승률**과 **빈도**가 가장 큰, 루트의 자식 노드 선택
    없으면, 조건을 만족하는 것이 나올 때까지 탐색 반복
  - 자식 노드의 **confidence bound값의 최소값이 가장 큰**, 루트의 자식 노드 선택

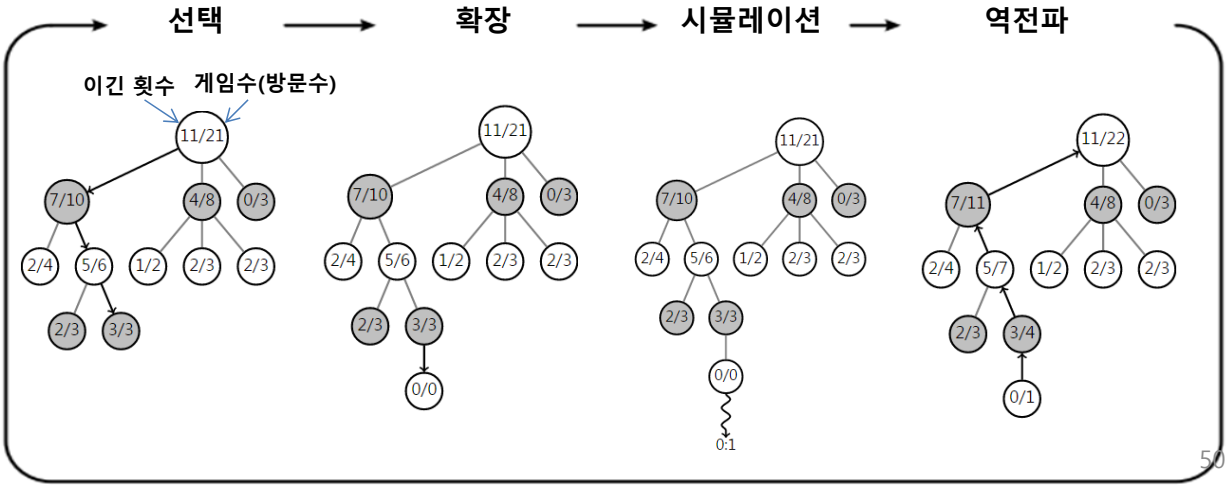
$$\frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$$



몬테카를로 트리 탐색

❖ 몬테카를로 트리 검색 – cont.

- 판의 형세판단을 위해 **휴리스틱**을 사용하는 **대신**, 가능한 많은 수의 **몬테카를로 시뮬레이션** 수행
- 일정 조건을 만족하는 부분은 **트리**로 구성하고, 나머지 부분은 **몬테카를로 시뮬레이션**
  - 가능성이 높은 수(move)들에 대해서 노드를 생성하여 트리의 탐색 폭을 줄이고, 트리 깊이를 늘리지 않기 위해 몬테카를로 시뮬레이션을 적용
  - **탐색 공간 축소**

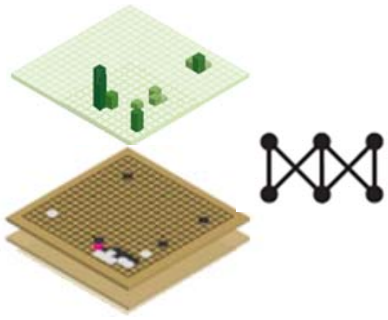


❖ 몬테카를로 트리 탐색에서 방문할 자식 노드를 선택할 때 사용하는 UCB는 어떤 대상에 대해서 우호적인지 왜 그러한지 설명하시오.

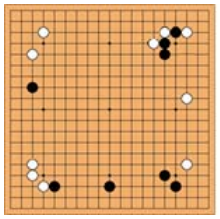
알파고의 탐색

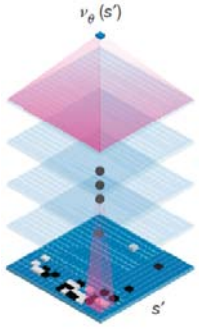
❖ **알파고의 몬테카를로 트리 검색**

- **바둑판 형세 판단**을 위한 한가지 방법으로 **몬테카를로 트리 검색** 사용
- 무작위로 바둑을 두는 것이 아니라, 프로 바둑기사들의 기보를 학습한 **확장 정책망**(rollout policy network)이라는 간단한 계산모델을 사용



정책망 : 가능한 착수(着手)들에 대한 **선호 확률분포**





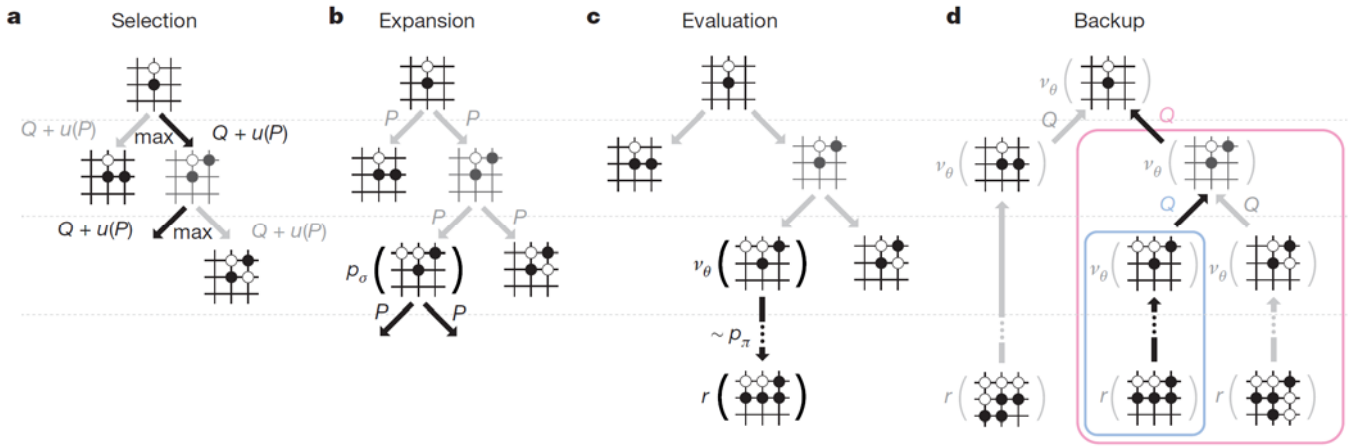
가치망 : 바둑판의 **형세 값**을 계산하는 계산모델

- 확률에 따라 착수를 하여 **몬테카를로 시뮬레이션**을 **반복**하여 해당 바둑판에 대한 **형세판단값 계산**
- 별도로 학습된 **딥러닝 신경망**인 **가치망(value network)**을 사용하여 형세판단값을 계산하여 함께 사용

알파고의 탐색

❖ **알파고의 몬테카를로 트리 검색**

- 많은 수의 **몬테카를로 시뮬레이션**과 **딥러닝 모델**의 신속한 계산을 위해 다수의 **CPU와 GPU**를 이용한 **분산처리**



https://www.youtube.com/watch?v=94IX97APSLs

Image : Nature

❖ 몬텐카를로 트리 탐색이 mini-max 알고리즘과 비교하여 어떤 점에서 우수한지 설명하시오.

Q & A