

탐색과 최적화 - 2

이건명

충북대학교 소프트웨어학과

인공지능 : 튜링 테스트에서 딥러닝까지

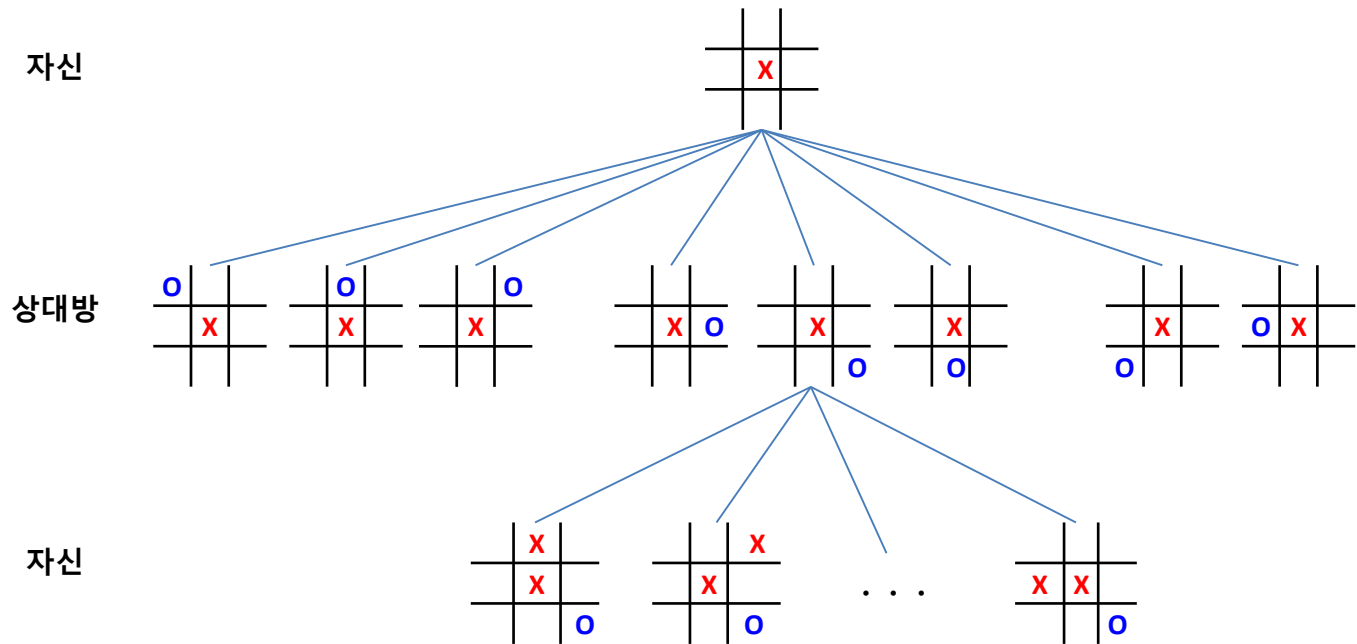
학습 내용

- 게임 트리의 용도에 대해서 알아본다.
- 게임 트리에 대한 mini-max 알고리즘과 α - β 가지치기 에 대해서 살펴본다.
- 몬테 카를로 트리 탐색(MCTS)의 동작 방식에 대해 알아본다.
- 알파고의 탐색 방법에 대해 알아본다.
- 제약조건 최적화 문제에 적용될 수 있는 백트래킹 탐색, 제약조건 전파에 대해서 알아본다.

4. 게임에서의 탐색

❖ 게임 트리(game tree)

- **상대가 있는 게임**에서 자신과 상대방의 가능한 게임 상태를 나타낸 트리
 - 틱-택-톡(tic-tac-toe), 바둑, 장기, 체스 등
- 게임의 결과는 마지막에 결정
- 많은 수(**lookahead**)를 볼 수록 유리



게임에서의 탐색

❖ mini-max 알고리즘(mini-max algorithm)

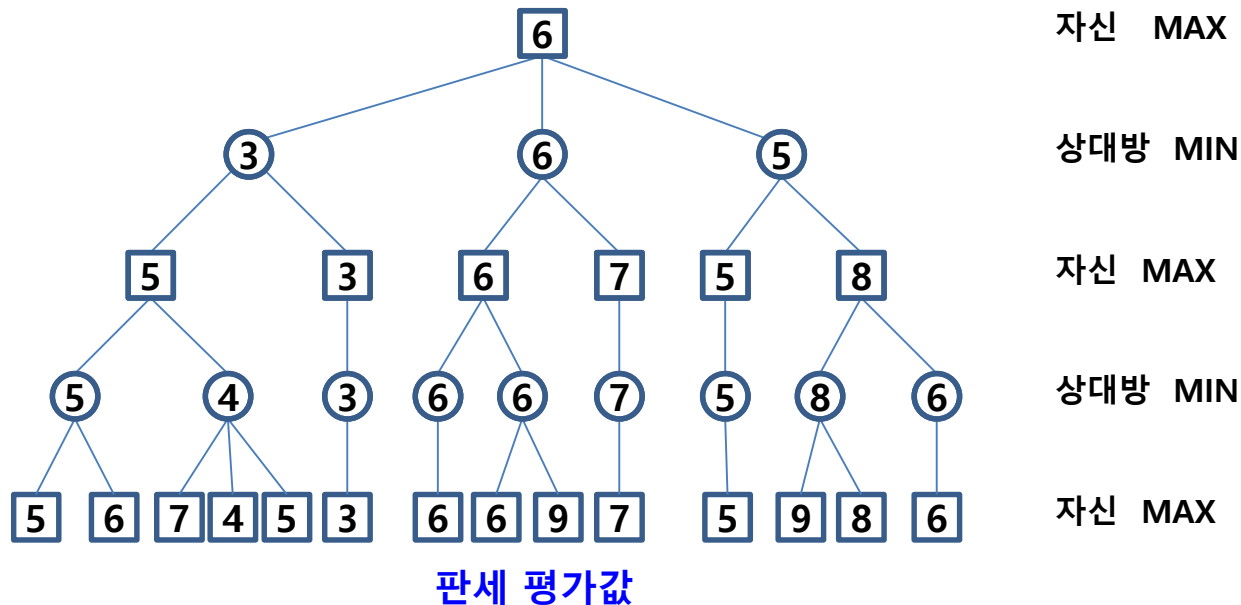
■ MAX 노드

- 자신에 해당하는 노드로 자기에게 유리한 최대값 선택

■ MIN 노드

- 상대방에 해당하는 노드로 최소값 선택

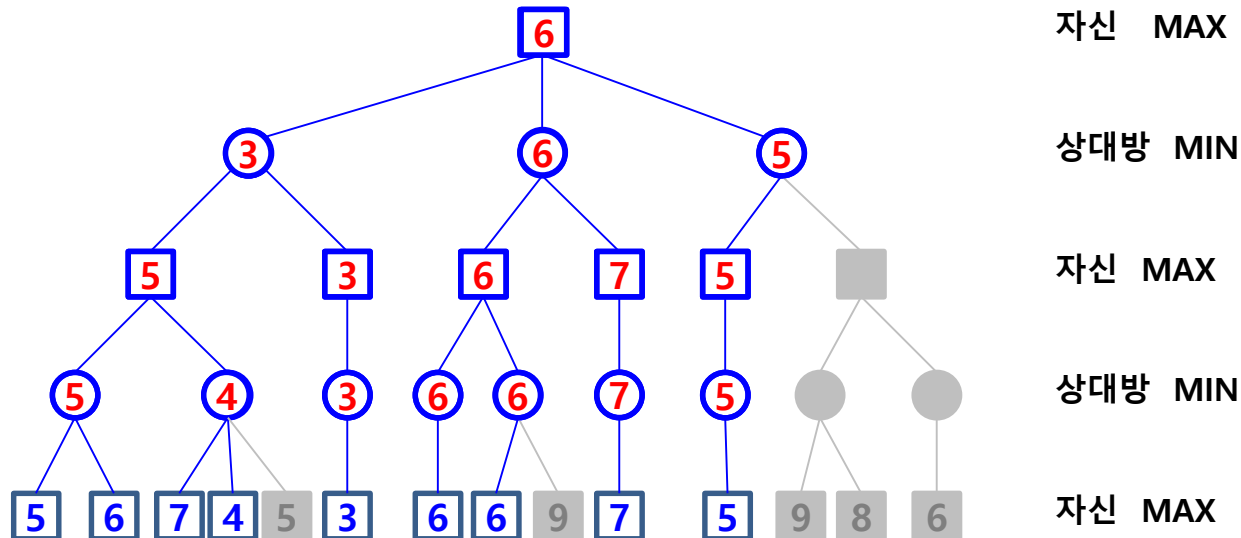
- 단말 노드부터 위로 올라가면서 최소(minimum)-최대(maximum) 연산을 반복하여 자신이 선택할 수 있는 방법 중 가장 좋은 것은 값을 결정



게임에서의 탐색

❖ α - β 가지치기 (pruning)

- 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 부모노드의 현재 값보다 작거나 같으면, 나머지 자식 노드 탐색 중지
 - β -자르기** : MAX 노드의 현재값이 부모노드의 현재 값보다 같거나 크면, 나머지 자식 노드 탐색 중지

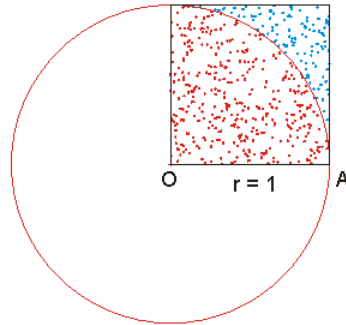


간단한 형태의 α - β 가지치기 예

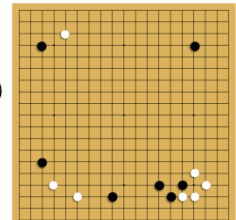
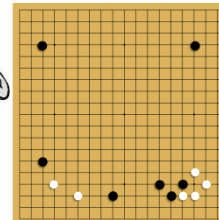
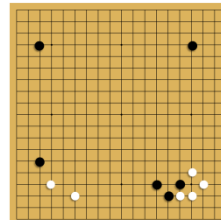
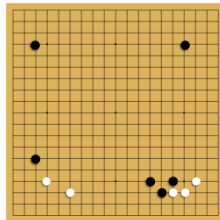
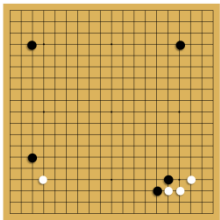
게임에서의 탐색

❖ 몬테카를로 시뮬레이션 (Monte Carlo Simulation)

- 특정 확률 분포로부터 무작위 표본(random sample)을 생성하고,
- 이 표본에 따라 행동을 하는 과정을 반복하여 결과를 확인하고,
- 이러한 결과확인 과정을 반복하여 최종 결정을 하는 것

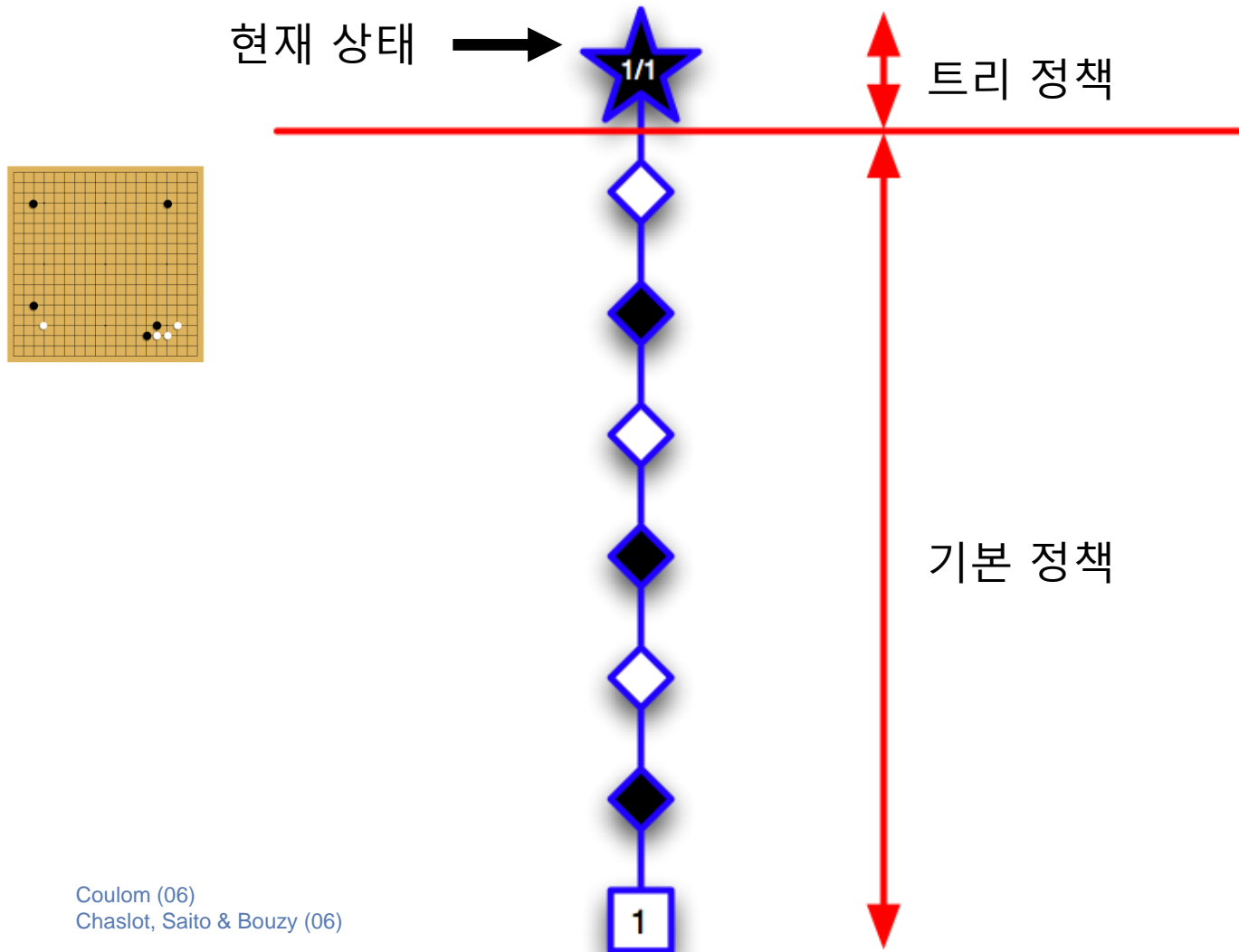


$$\frac{\text{원 안의 샘플 개수}}{\text{전체 샘플의 개수}} \rightarrow \frac{\pi}{4}$$

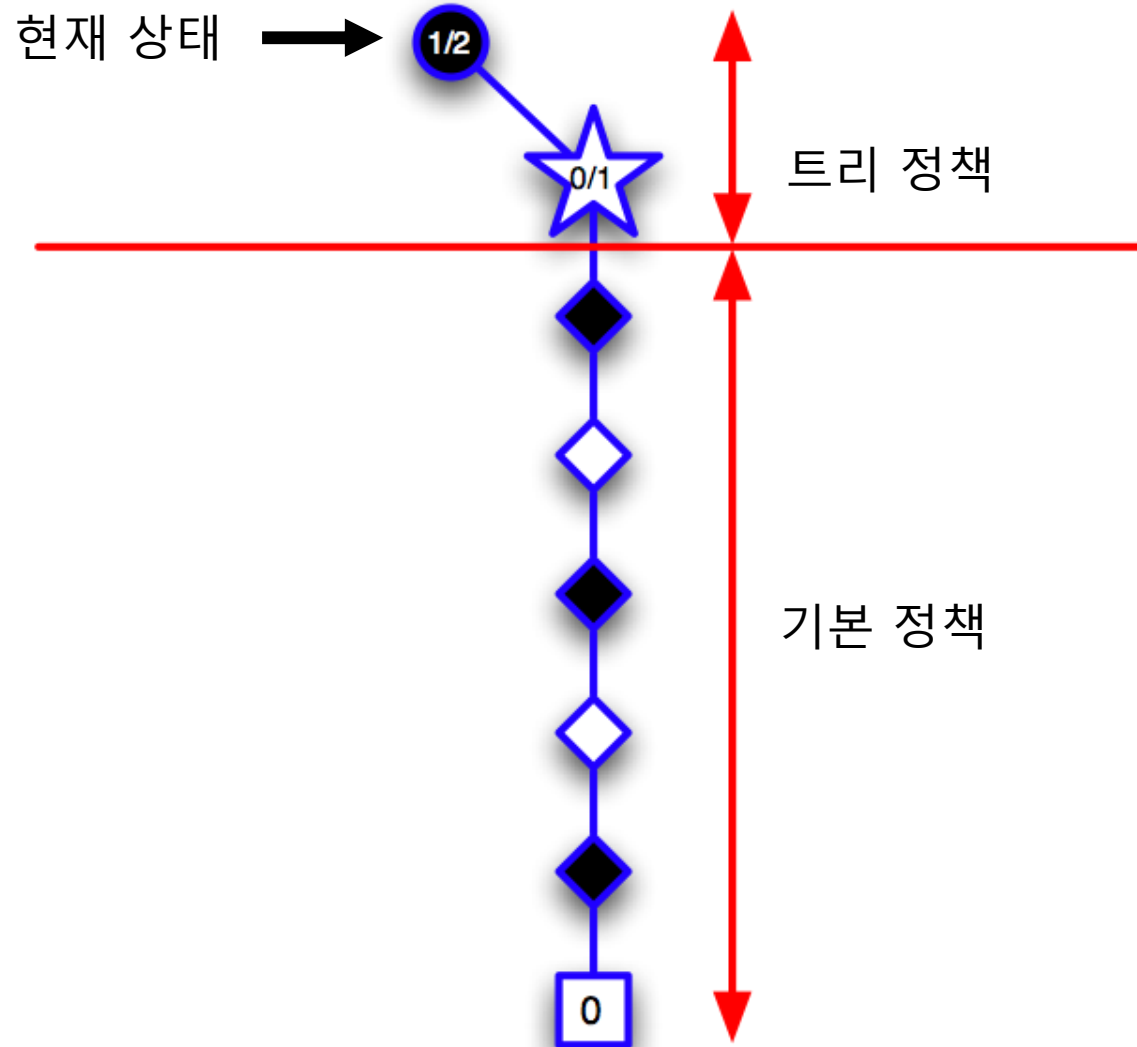


몬테카를로 트리 탐색

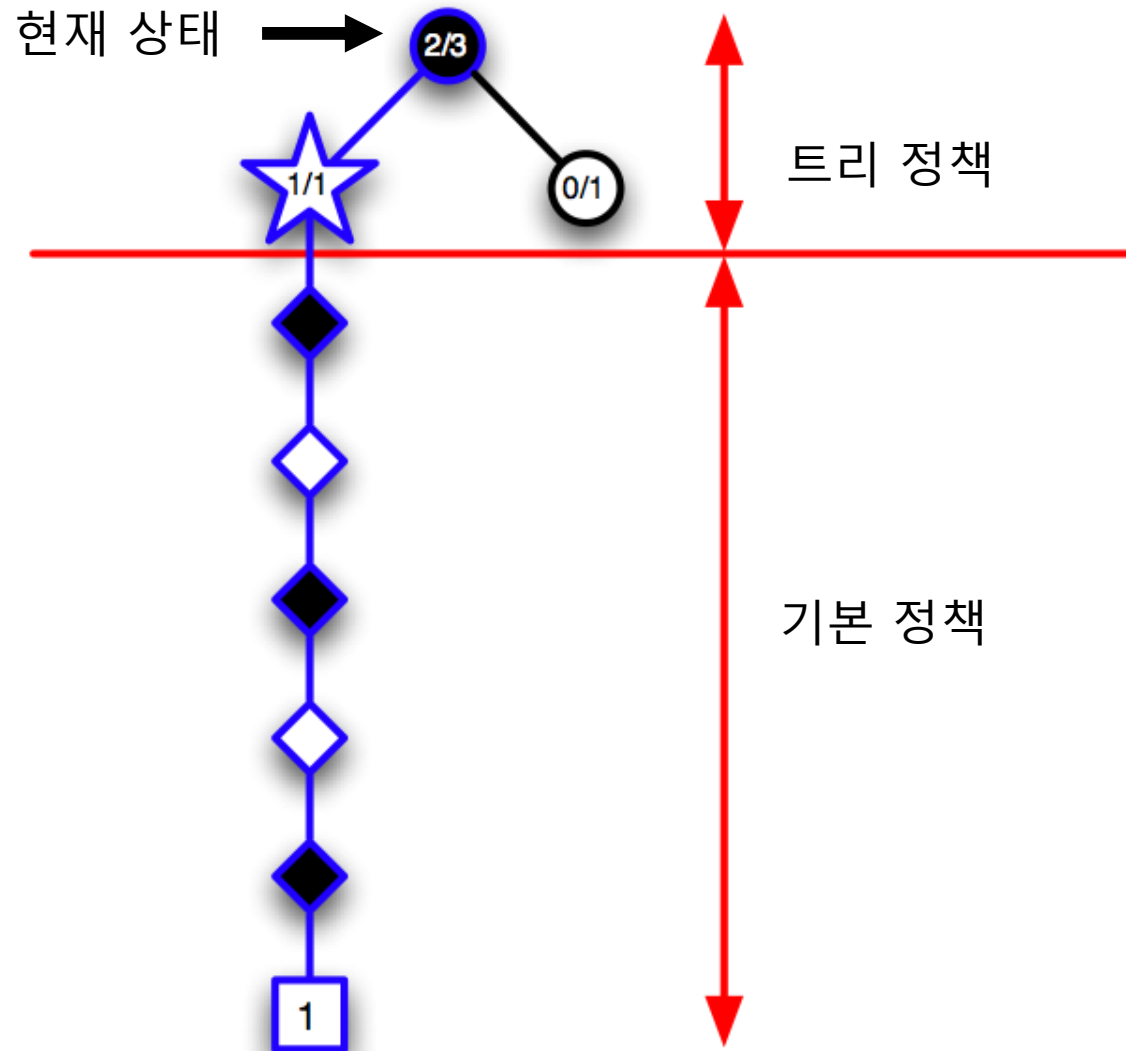
❖ 몬테카를로 트리 탐색(Monte Carlo Tree Search, MCTS)



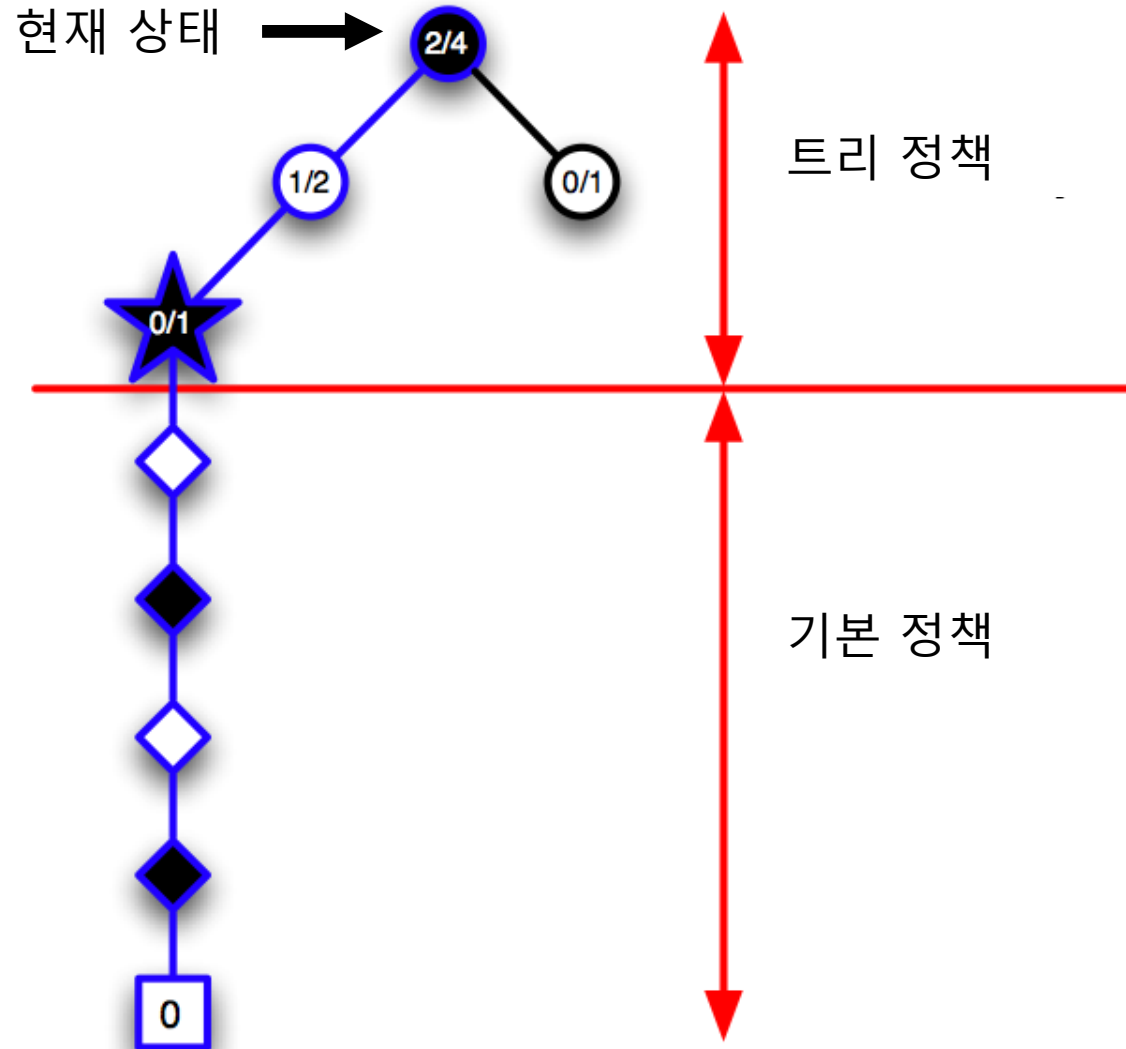
몬테카를로 트리 탐색



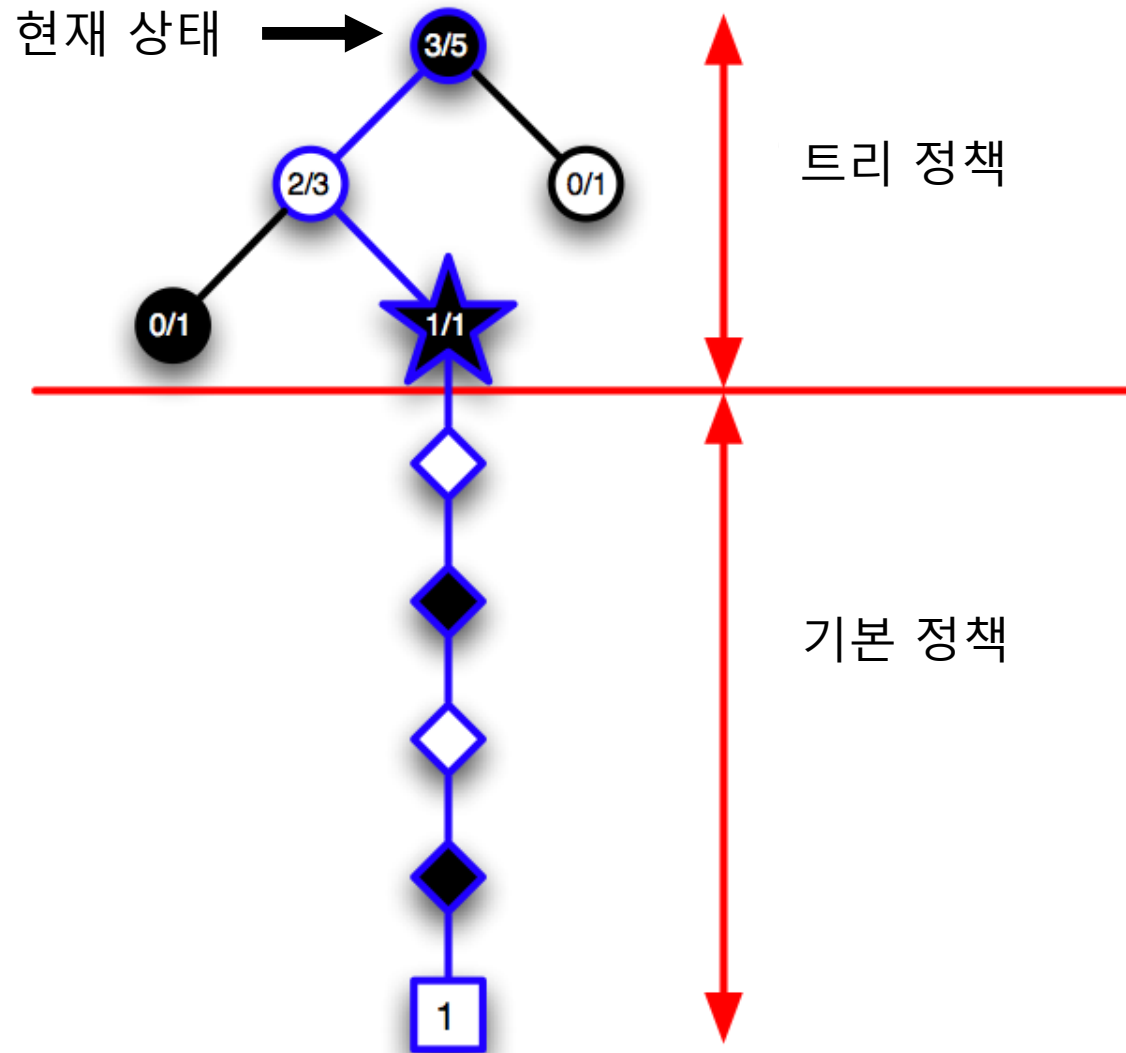
몬테카를로 트리 탐색



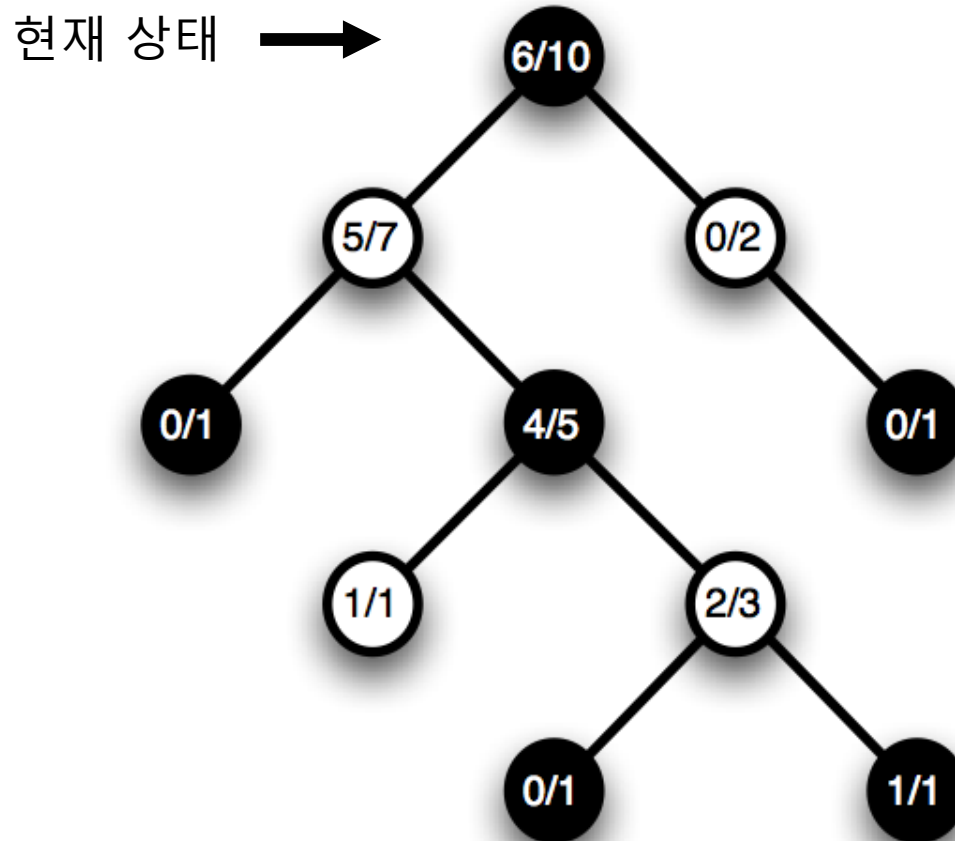
몬테카를로 트리 탐색



몬테카를로 트리 탐색

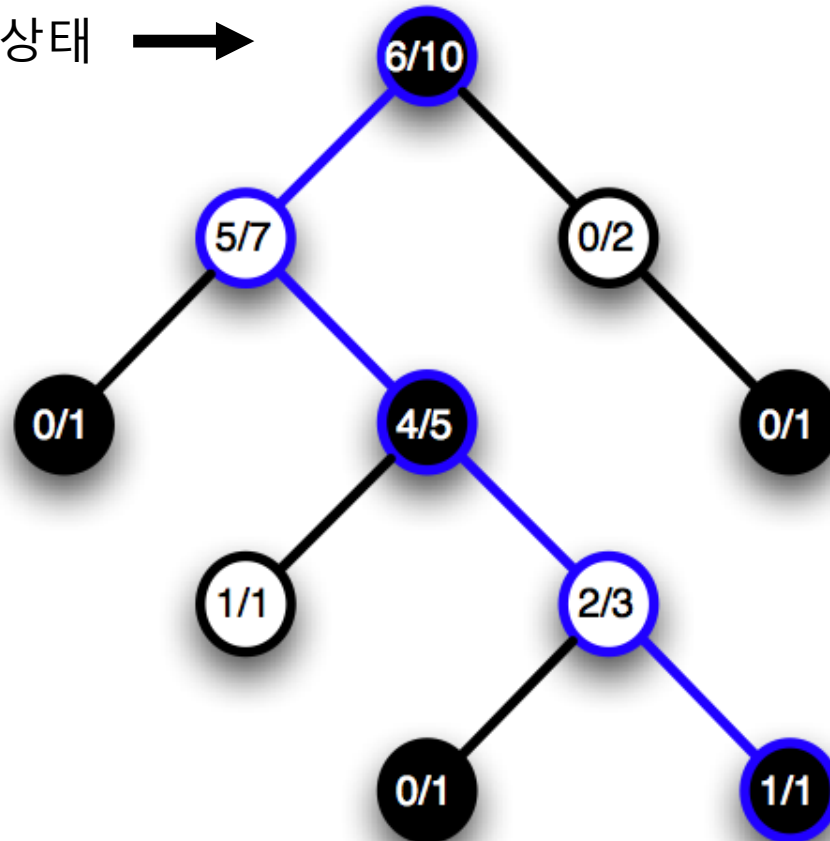


몬테카를로 트리 탐색

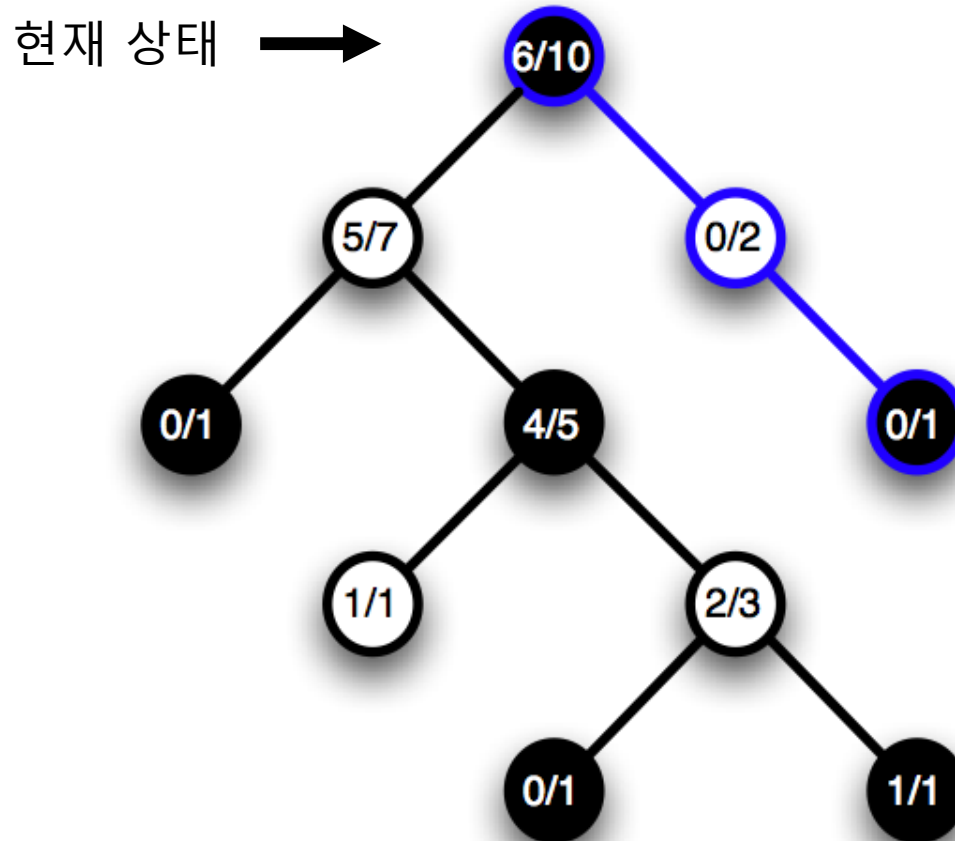


몬테카를로 트리 탐색

현재 상태 →



몬테카를로 트리 탐색



게임에서의 탐색

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

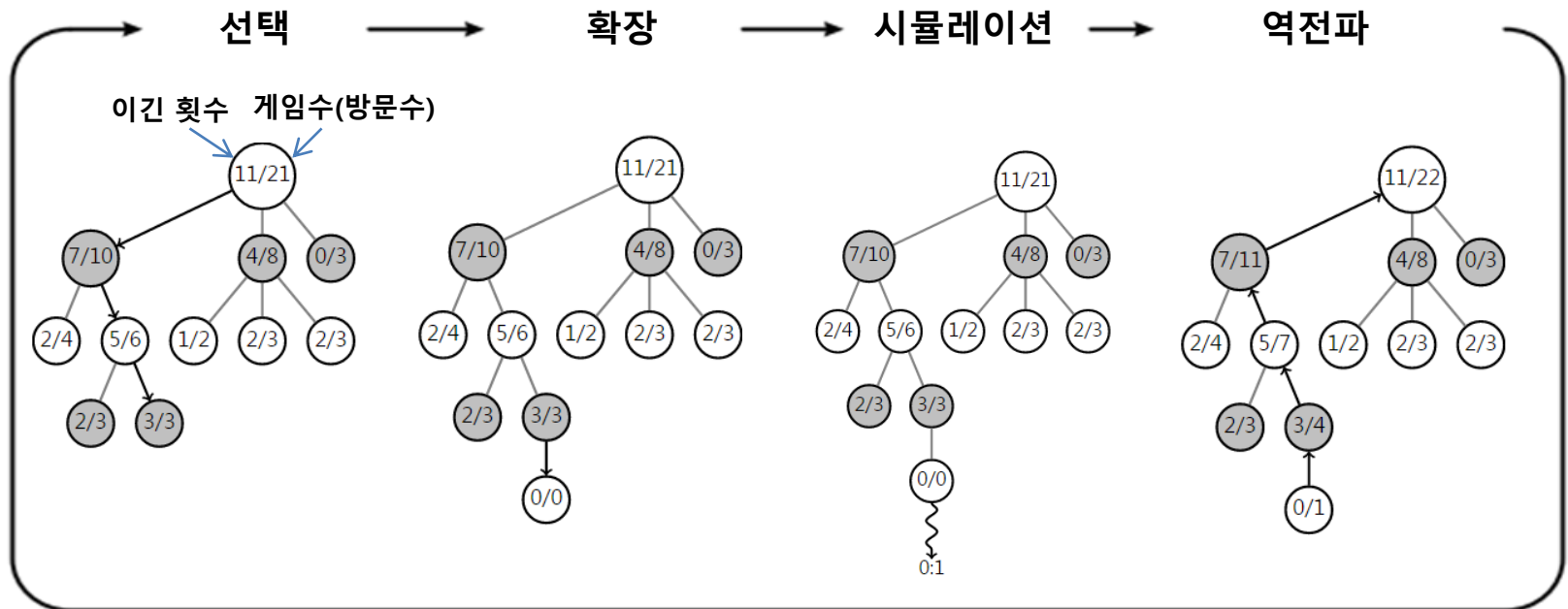
- 탐색 공간(search space)을 무작위 표본추출(random sampling)을 하면서, 탐색트리를 확장하여 가장 좋아 보이는 것을 선택하는 휴리스틱 탐색 방법
- 4개 단계를 반복하여 시간이 허용하는 동안 트리 확장 및 시뮬레이션

선택(selection)

→ 확장(expansion)

→ 시뮬레이션(simulation) : 몬테카를로 시뮬레이션

→ 역전파(back propagation)



게임에서의 탐색

❖ 몬테카를로 트리 탐색 - cont.

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

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

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

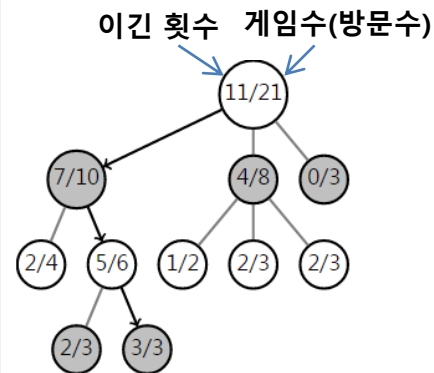
활용(exploitation) 탐험(exploration)

선택

확장

시뮬레이션

역전파



게임에서의 탐색



❖ 몬테카를로 트리 탐색 - cont.

■ 확장(expansion)

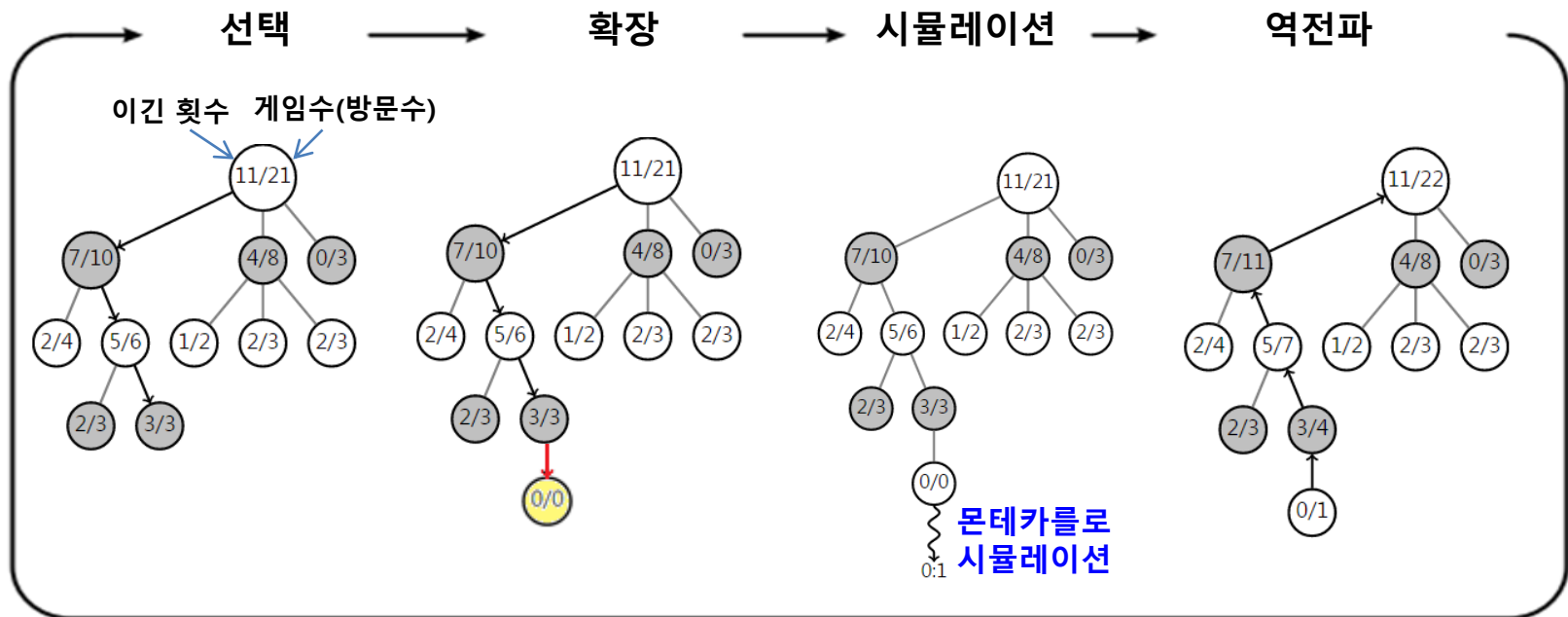
- 단말노드에서 트리 정책에 따라 노드 추가
 - 예. 일정 횟수 이상 시도된 수(move)가 있으면 해당 수에 대한 노드 추가

■ 시뮬레이션(simulation)

- 기본 정책(default/rollout policy)에 의한 몬테카를로 시뮬레이션 적용
- 무작위 선택(random moves) 또는 약간 똑똑한 방법으로 게임 끝날 때까지 진행

■ 역전파(backpropagation)

- 단말 노드에서 루트 노드까지 올라가면서 승점 반영



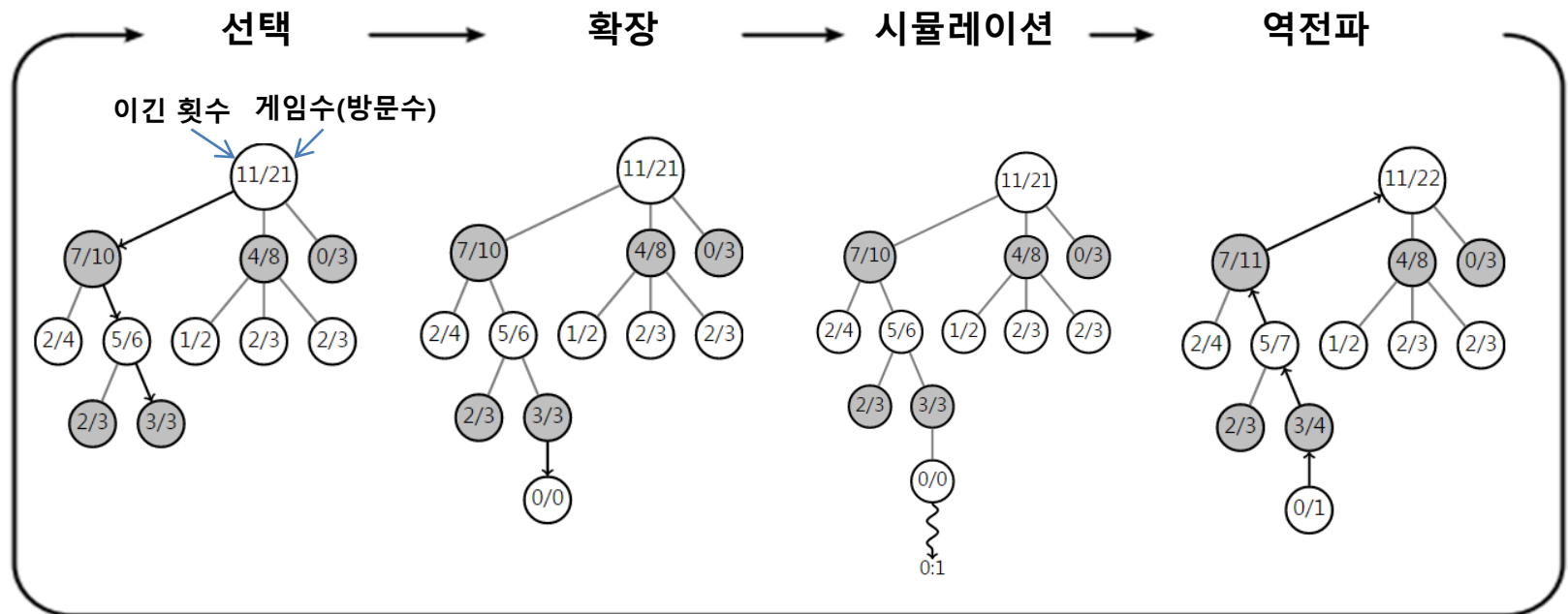
몬테카를로 트리 탐색

❖ 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 bound**값의 **최소값**이 가장 큰, 루트의 자식 노드 선택

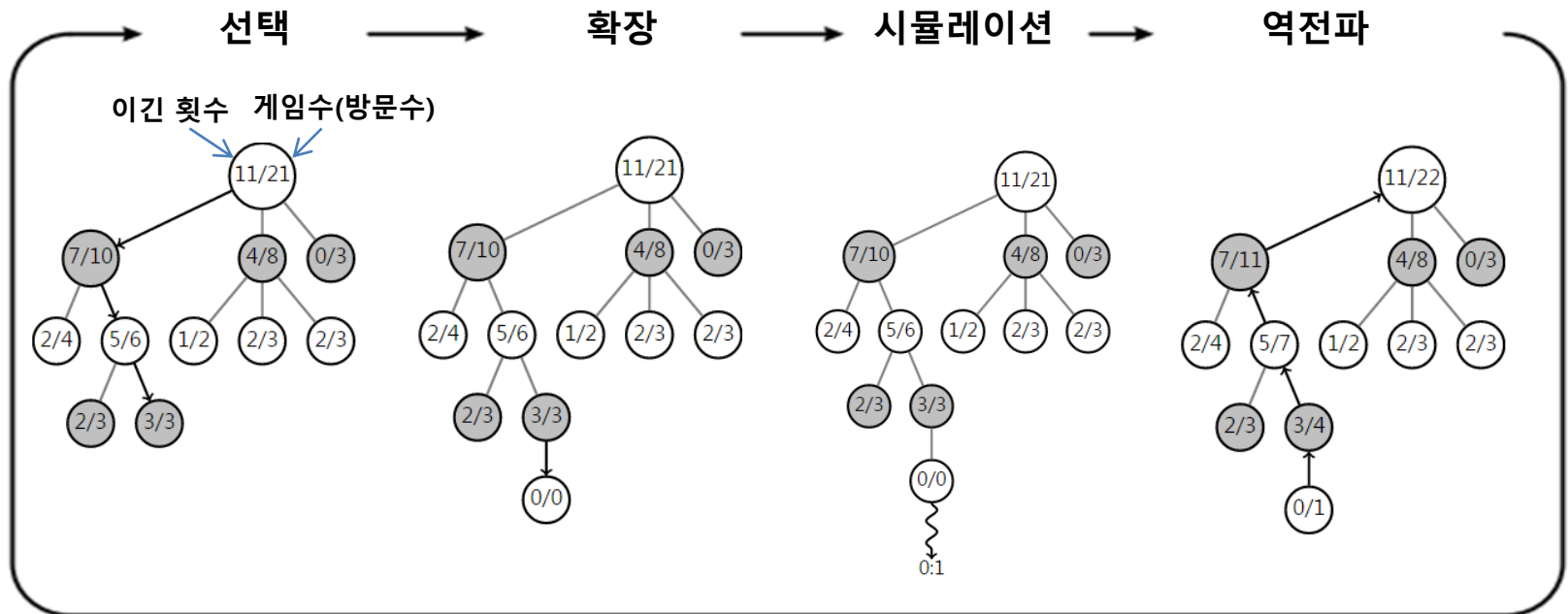
$$\frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}}$$



몬테카를로 트리 탐색

❖ 몬테카를로 트리 검색 - cont.

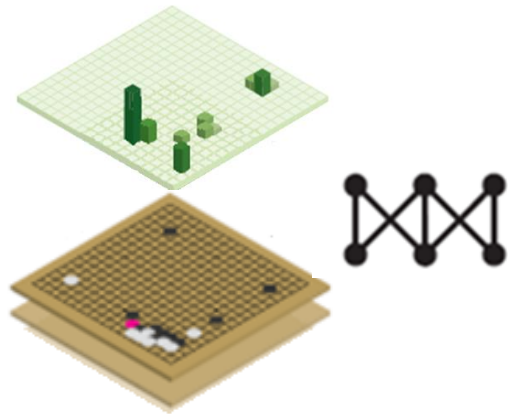
- 판의 형세판단을 위해 휴리스틱을 사용하는 대신, 가능한 많은 수의 몬테카를로 시뮬레이션 수행
- 일정 조건을 만족하는 부분은 트리로 구성하고, 나머지 부분은 몬테카를로 시뮬레이션
 - 가능성이 높은 수(move)들에 대해서 노드를 생성하여 트리의 탐색 폭을 줄이고, 트리 깊이를 늘리지 않기 위해 몬테카를로 시뮬레이션을 적용
 - 탐색 공간 축소



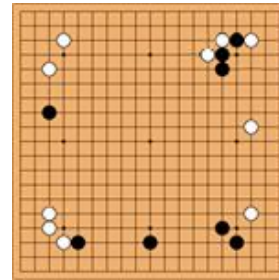
알파고의 탐색

❖ 알파고의 몬테카를로 트리 검색

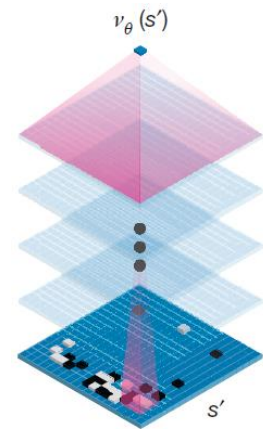
- 바둑판 형세 판단을 위한 한가지 방법으로 몬테카를로 트리 검색 사용
- 무작위로 바둑을 두는 것이 아니라, 프로 바둑기사들을 기보를 학습한 **확장 정책망**(rollout policy network)이라는 간단한 계산모델을 사용



정책망 : 가능한 착수(着手)들에
대한 선호 확률분포



⇒ 0.6



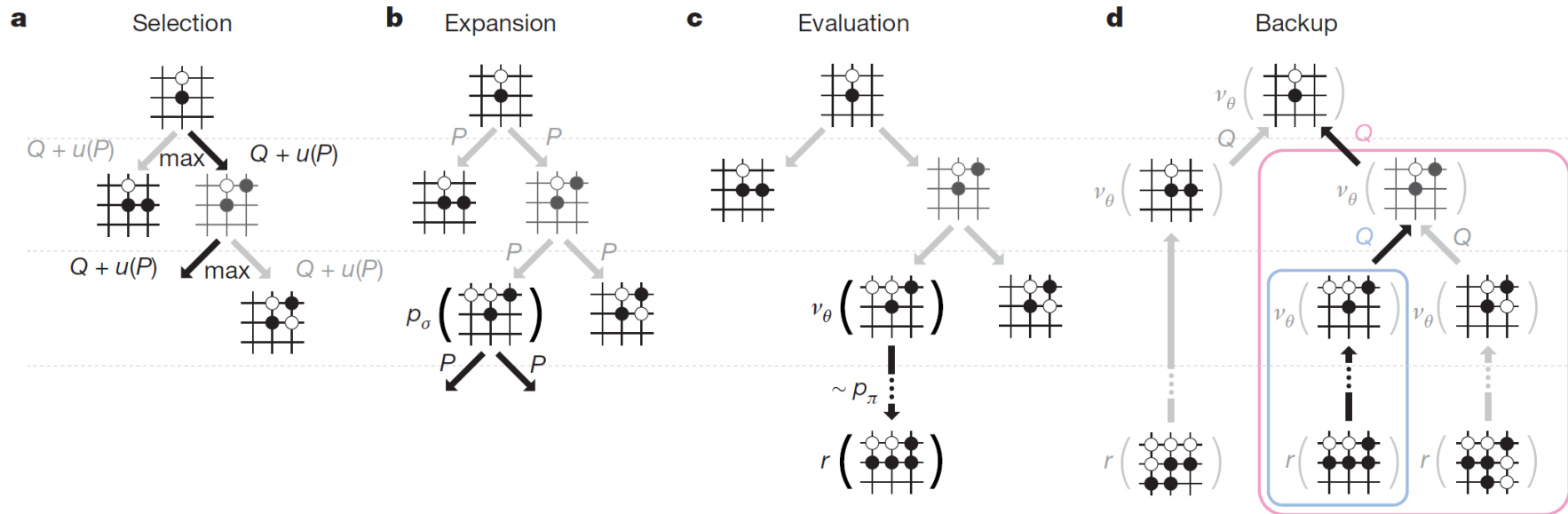
가치망 : 바둑판의 형세 값을
계산하는 계산모델

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

알파고의 탐색

❖ 알파고의 몬테카를로 트리 검색

- 많은 수의 몬테카를로 시뮬레이션과 딥러닝 모델의 신속한 계산을 위해 다수의 CPU와 GPU를 이용한 분산처리 – 초기 모델



알파고의 탐색

❖ 알파고의 버전



버전	하드웨어	Elo 점수	경기 기록
AlphaGo Fan	176 GPUs, 분산처리	3,144	5:0 (Fan Hui)
AlphaGo Lee	48 TPUs, 분산처리	3,739	4:1 (Lee Sedol)
AlphaGo Master	4 TPUs v2, 단일 기계	4,858	60:0 (프로 기사) Future of Go Summit
AlphaGo Zero	4 TPUs v2, 단일 기계	5,185	100:0 (AlphaGo Lee) 89:11(AlphaGo Master)

Games	67245
Players	1873
Most Recent Game	2017-10-20

Rank	Name	♂♀	Flag	Elo
1	Ke Jie	♂		3670
2	Park Junghwan	♂		3628
3	Mi Yuting	♂		3574
4	Shin Jinseo	♂		3564
5	Iyama Yuta	♂		3556
6	Shi Yue	♂		3535
7	Gu Zihao	♂		3530
8	Kim Jiseok	♂		3517
9	Lee Sedol	♂		3517
10	Tuo Jiaxi	♂		3511

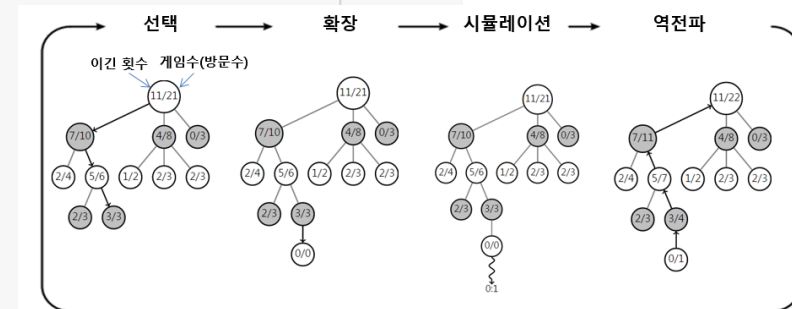
프로그래밍 실습 : 틱-택-토 MCTS

```

1 from abc import ABC, abstractmethod # abstract base class
2 from collections import defaultdict
3 import math
4
5 class MCTS:
6     "Monte Carlo tree searcher. 먼저 rollout한 다음, 위치(move) 선택 "
7     def __init__(self, c=1):
8         self.Q = defaultdict(int) # 노드별 이긴 횟수(reward) 값을 0으로 초기화
9         self.N = defaultdict(int) # 노드별 방문횟수(visit count)를 0으로 초기화
10        self.children = dict() # 노드의 자식노드
11        self.c = c # UCT 계산에 사용되는 계수
12
13    def choose(self, node):
14        "node의 최선인 자식 노드 선택"
15        if node.is_terminal(): # node가 단말인 경우 오류
16            raise RuntimeError(f"choose called on terminal node {node}")
17        if node not in self.children: # node가 children에 포함되지 않으면 무작위 선택
18            return node.find_random_child()
19
20    def score(n): # 점수 계산
21        if self.N[n] == 0:
22            return float("-inf") # 한번도 방문하지 않은 노드인 경우 - 선택 배제
23        return self.Q[n] / self.N[n] # 평균 점수
24
25    return max(self.children[node], key=score)
26
27    def do_rollout(self, node):
28        "게임 트리에서 한 층만 더 보기"
29        path = self._select(node)
30        leaf = path[-1]
31        self._expand(leaf)
32        reward = self._simulate(leaf)
33        self._backpropagate(path, reward)

```

○	×	○
○	○	×
×	×	

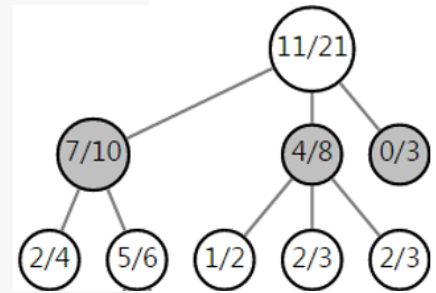


Original developer: Luke Harold Miles, 2019

```

35 def _select(self, node): # 선택 단계
36     "node의 아직 시도해보지 않은 자식 노드 찾기"
37     path = []
38     while True:
39         path.append(node)
40         if node not in self.children or not self.children[node]:
41             # node의 child나 grandchild가 아닌 경우: 아직 시도해보지 않은 것 또는 단말 노드
42             return path
43         unexplored = self.children[node] - self.children.keys() # 차집합
44         if unexplored:
45             n = unexplored.pop()
46             path.append(n)
47             return path
48         node = self._uct_select(node) # 한 단계 내려가기
49
50 def _expand(self, node): # 확장 단계
51     "children에 node의 자식노드 추가"
52     if node in self.children:
53         return # 이미 확장된 노드
54     self.children[node] = node.find_children() # 선택가능 move들을 node의 children에 추가
55
56 def _simulate(self, node): # 시뮬레이션 단계
57     "node의 무작위 시뮬레이션에 대한 결과(reward) 반환"
58     invert_reward = True
59     while True:
60         if node.is_terminal(): # 단말에 도달하면 승패여부 결정
61             reward = node.reward()
62             return 1 - reward if invert_reward else reward
63         node = node.find_random_child() # 선택할 수 있는 것 중에서 무작위로 선택
64         invert_reward = not invert_reward
65

```




```

66 def _backpropagate(self, path, reward): # 역전파 단계
67     "단말 노드의 조상 노드들에게 보상(reward) 전달"
68     for node in reversed(path): # 역순으로 가면서 Monte Carlo 시뮬레이션 결과 반영
69         self.N[node] += 1
70         self.Q[node] += reward
71         reward = 1 - reward # 자신에게는 1 상대에게는 0, 또는 그 반대
72
73 def _uct_select(self, node): # UCB 정책 적용을 통한 노드 확장 대상 노드 선택
74     "탐험(exploration)과 이용(exploitation)의 균형을 맞춰 node의 자식 노드 선택"
75     # node의 모든 자신 노드가 이미 확장되었는지 확인
76     assert all(n in self.children for n in self.children[node])
77     log_N_vertex = math.log(self.N[node])
78
79     def uct(n):
80         "UCB(Upper confidence bound) 점수 계산 "
81         return self.Q[n] / self.N[n] + self.c * math.sqrt(2*log_N_vertex / self.N[n])
82
83     return max(self.children[node], key=uct)
84

```

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

```
84
85 class Node(ABC):
86     " 게임 트리의 노드로서 보드판의 상태 표현 "
87     @abstractmethod
88     def find_children(self): # 해당 보드판 상태의 가능한 모든 가능한 후속 상태
89         return set()
90
91     @abstractmethod
92     def find_random_child(self): # 현 보드에 대한 자식 노드 무작위 선택
93         return None
94
95     @abstractmethod
96     def is_terminal(self): # 자식 노드인지 판단
97         return True
98
99     @abstractmethod
100     def reward(self): # 점수 계산
101         return 0
102
103     @abstractmethod
104     def __hash__(self): # 노드에 해시적용 가능하도록(hashable) 함
105         return 123456789
106
107     @abstractmethod
108     def __eq__(node1, node2): # 노드는 서로 비교 가능해야 함
109         return True
```

```

1 from collections import namedtuple
2 from random import choice
3 #from monte_carlo_tree_search import MCTS, Node
4
5 TTTB = namedtuple("TicTacToeBoard", "tup turn winner terminal")
6
7 class TicTacToeBoard(TTTB, Node): # TTTB의 속성들도 상속
8     def find_children(board): # 전체 가능한 move들 집합으로 반환
9         if board.terminal: # 게임이 끝나면 아무것도 하지 않음
10             return set()
11         return { # 그렇지 않으면, 비어있는 곳에 각각 시도
12             board.make_move(i) for i, value in enumerate(board.tup) if value is None
13         }
14
15     def find_random_child(board): # 무작위로 move 선택
16         if board.terminal:
17             return None # 게임이 끝나면 아무것도 하지 않음
18         empty_spots = [i for i, value in enumerate(board.tup) if value is None]
19         return board.make_move(choice(empty_spots))
20
21     def reward(board): # 점수 계산
22         if not board.terminal:
23             raise RuntimeError(f"reward called on nonterminal board {board}")
24         if board.winner is board.turn:
25             # 자기 차례이면서 자기가 이긴 상황은 불가능
26             raise RuntimeError(f"reward called on unreachable board {board}")
27         if board.turn is (not board.winner):
28             return 0 # 상대가 이긴 상황
29         if board.winner is None:
30             return 0.5 # 비긴 상황
31         # 일어날 수 없는 상황
32         raise RuntimeError(f"board has unknown winner type {board.winner}")
33

```

```

34 def is_terminal(board): # 게임 종료 여부
35     return board.terminal
36
37 def make_move(board, index): # index 위치에 board.turn 표시하기 하기
38     tup = board.tup[:index] + (board.turn,) + board.tup[index + 1 :]
39     turn = not board.turn # 순서 바꾸기
40     winner = find_winner(tup) # 승자 또는 미종료 판단
41     is_terminal = (winner is not None) or not any(v is None for v in tup)
42     return TicTacToeBoard(tup, turn, winner, is_terminal) # 보드 상태 반환
43
44 def display_board(board): # 보드 상태 출력
45     to_char = lambda v: ("X" if v is True else ("O" if v is False else " "))
46     rows = [
47         [to_char(board.tup[3 * row + col]) for col in range(3)] for row in range(3)
48     ]
49     return ("Wn 1 2 3Wn"
50         + "Wn".join(str(i + 1) + " " + " ".join(row) for i, row in enumerate(rows)) + "Wn")
51

```



```
52 def play_game(): # 게임하기
53     tree = MCTS()
54     board = new_Board()
55     print(board.display_board())
56     while True:
57         row_col = input("위치 row,col: ")
58         row, col = map(int, row_col.split(", "))
59         index = 3 * (row - 1) + (col - 1)
60         if board.tup[index] is not None: # 비어있는 위치가 아닌 경우
61             raise RuntimeError("Invalid move")
62         board = board.make_move(index) # index 위치의 보드 상태 변경
63         print(board.display_board())
64         if board.terminal: # 게임 종료
65             break
66
67         for _ in range(50): # 매번 50번의 rollout을 수행
68             tree.do_rollout(board)
69         board = tree.choose(board) # 최선의 값을 갖는 move 선택하여 보드에 반영
70         print(board.display_board())
71         if board.terminal:
72             print('게임 종료')
73             break
74
```

```
1 2 3
1
2
3
위치 row,col: 1,1
```

```
1 2 3
1 X
2
3
```

```
1 2 3
1 X
2
3 0
위치 row,col: 1,2
```

```
1 2 3
1 X X
2
3 0
```

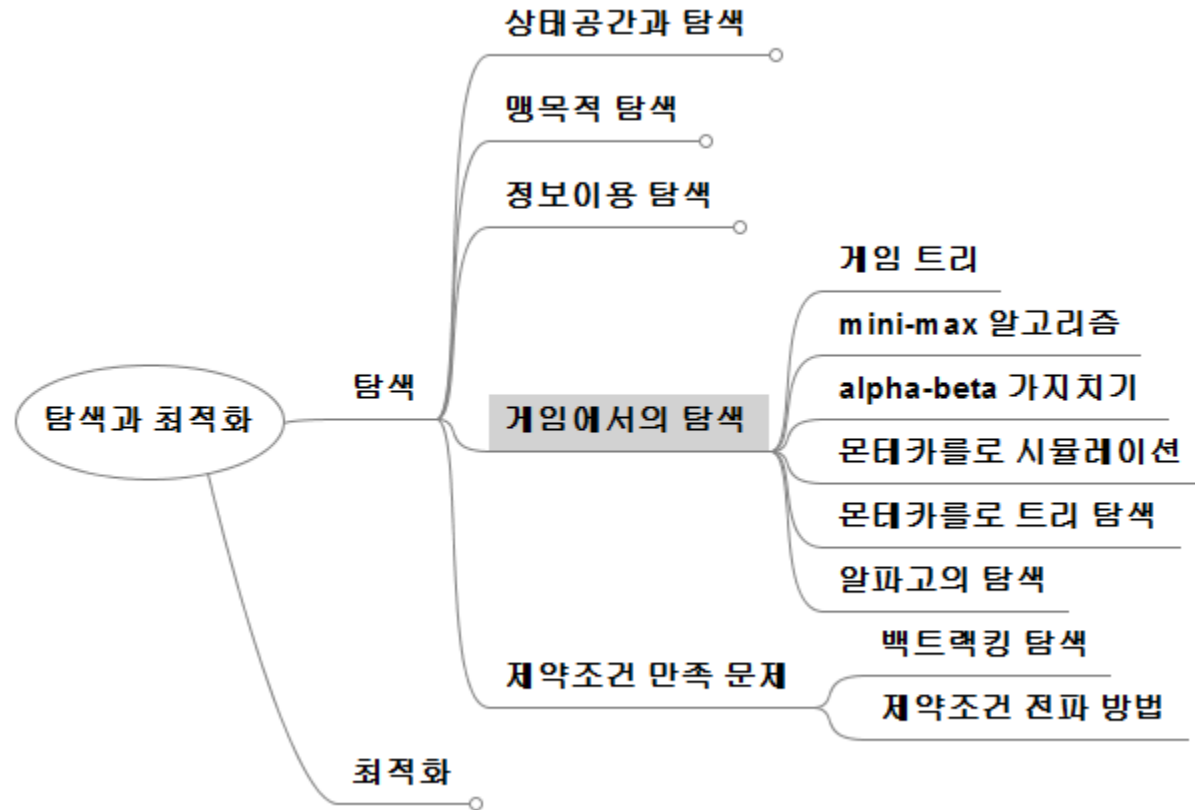
```
1 2 3
1 X X 0
2
3 0
위치 row,col: 3,1
```

```
1 2 3
1 X X 0
2
3 X 0
```

```
1 2 3
1 X X 0
2 0
3 X 0
게임 종료
```

```
75 def winning_combos(): # 이기는 배치 조합
76     for start in range(0, 9, 3): # 행에 3개 연속
77         yield (start, start + 1, start + 2)
78     for start in range(3): # 열에 3개 연속
79         yield (start, start + 3, start + 6)
80     yield (0, 4, 8) # 오른쪽 아래로 가는 대각선 3개
81     yield (2, 4, 6) # 왼쪽 아래로 가는 대각선 3개
82
83 def find_winner(tup): # X가 이기면 True, O가 이기면 False, 미종료 상태이면 None 반환
84     for i1, i2, i3 in winning_combos():
85         v1, v2, v3 = tup[i1], tup[i2], tup[i3]
86         if False is v1 is v2 is v3:
87             return False
88         if True is v1 is v2 is v3:
89             return True
90     return None
91
92 def new_Board(): # 비어있는 보드판 생성
93     return TicTacToeBoard(tup=(None,) * 9, turn=True, winner=None, terminal=False)
94
95 if __name__ == "__main__":
96     play_game()
```

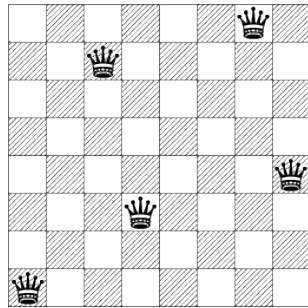
게임에서의 탐색



5. 제약조건 만족 문제

❖ 제약조건 만족 문제(constraint satisfaction problem)

- 주어진 제약조건을 만족하는 **조합 해**(combinatorial solution)를 찾는 문제
- 예. 8-퀸(queen) 문제



▪ 탐색 기반의 해결방법

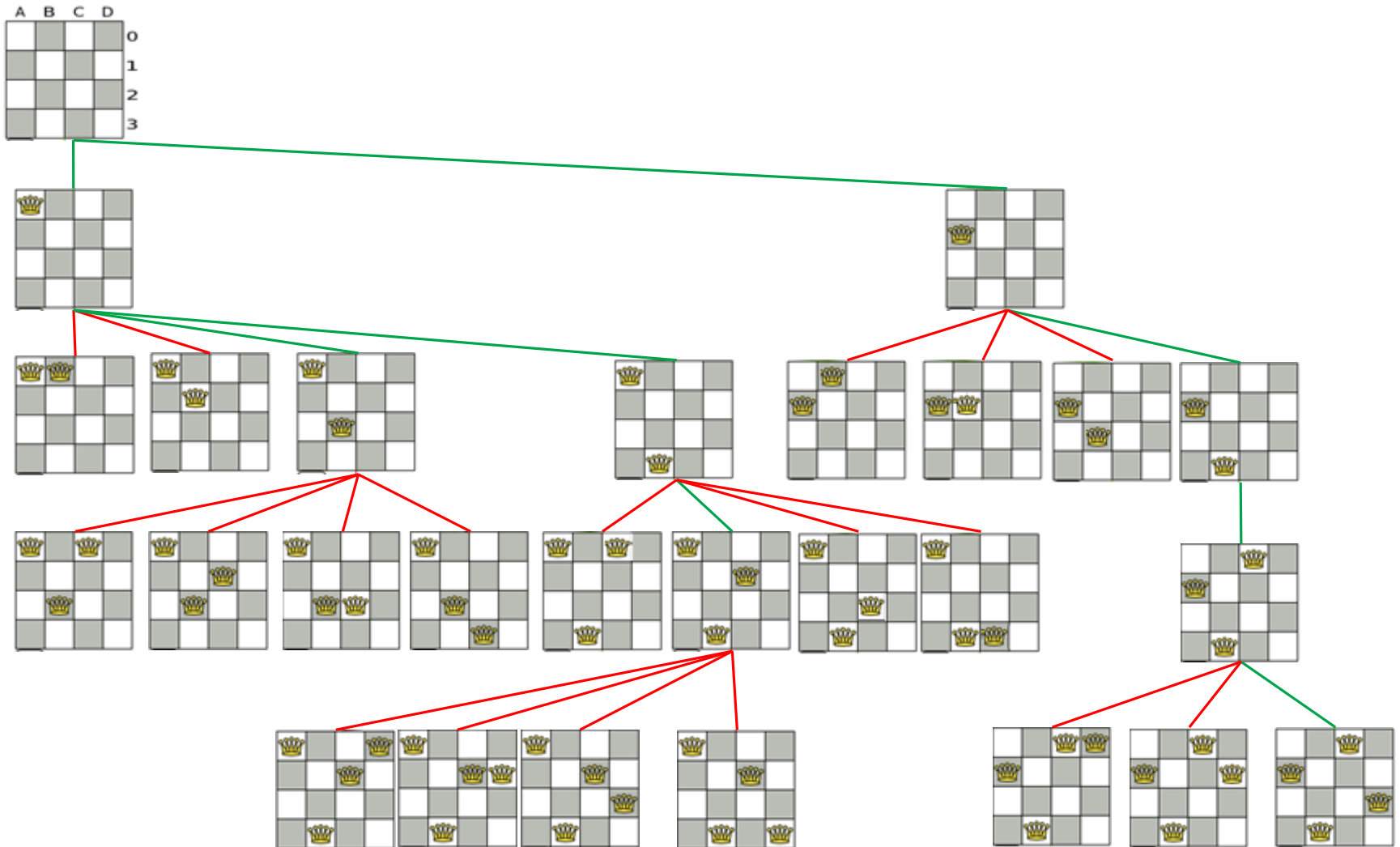
- 백트래킹 탐색
- 제약조건 전파

▪ **백트래킹 탐색**(backtracking search)

- 깊이 우선 탐색을 하는 것처럼 변수에 허용되는 값을 하나씩 대입
- 모든 가능한 값을 대입해서 만족하는 것이 없으면 이전 단계로 돌아가서 이전 단계의 변수에 다른 값을 대입

제약조건 만족 문제



❖ 예. 백트래킹 탐색을 이용한 4-퀸(queen) 문제

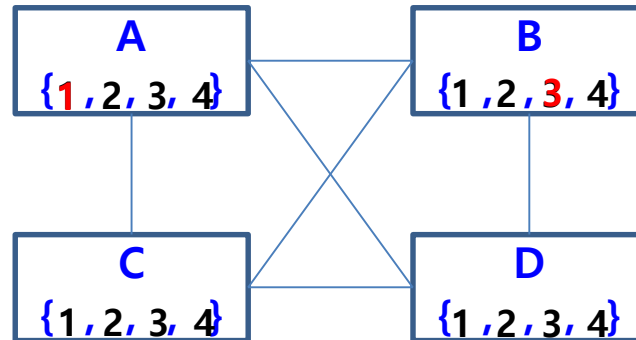


제약조건 만족 문제

❖ 제약조건 전파(constraint propagation)

- 인접 변수 간의 제약 조건에 따라 각 변수에 허용될 수 없는 값들을 제거하는 방식



	A	B	C	D
1		×	×	×
2		×	×	
3			×	×
4			×	×

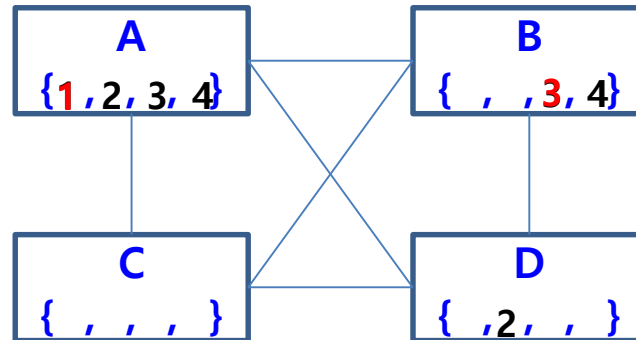






제약조건 만족 문제

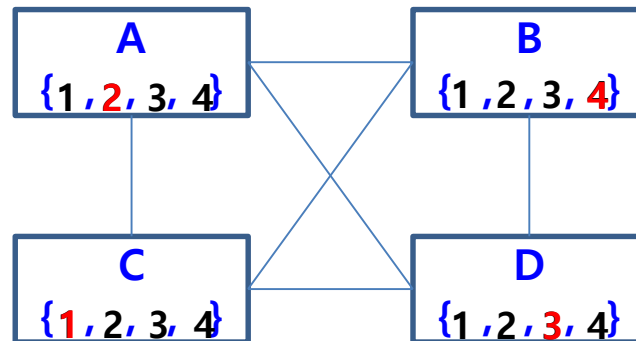
❖ 제약조건 전파(constraint propagation)

- 인접 변수 간의 제약 조건에 따라 각 변수에 허용될 수 없는 값들을 제거하는 방식

	A	B	C	D
1		X	X	X
2		X	X	
3			X	X
4			X	X



	A	B	C	D
1		X		X
2		X	X	X
3		X	X	
4			X	X



Quiz

❖ 게임 트리에 대한 설명으로 적합하지 않은 것을 선택하시오.

- ① 상대가 있는 게임에서 자신과 상대방의 가능한 게임 상태를 나타낸 트리이다.
- ② 루트 노드는 게임트리를 만든 플레이어에 해당하고, 이 플레이어의 다음 행동을 결정하기 위해 게임트리를 사용한다.
- ③ 게임 트리는 게임이 끝나는 시점까지 모든 가능한 상태를 나타내게 된다.
- ④ 게임의 승패 또는 유리한 정도를 일반적으로 수치값으로 나타내며, 이 값은 루트 노드에 해당하는 플레이어에게는 큰 값일수록, 상대 플레이어에게는 작은 값일수록 바람직한 것이다.

❖ 게임 트리에 대한 알고리즘에 대한 설명으로 적합하지 않은 것을 선택하시오.

- ① mini-max 알고리즘은 게임 트리의 루트 노드를 MAX 노드로 하고, MAX 노드의 자식 노드는 MIN 노드, MIN 노드의 자식 노드는 MAX 노드로 간주한다.
- ② mini-max 알고리즘에서 MAX 노드는 자식 노드 중에서 가장 큰 값을 선택하고, MIN 노드는 자식 노드 중에서 가장 작은 값을 선택한다.
- ③ 게임 트리를 만들 때 단말 노드에서 승패가 결정되지 않은 경우에는 판세 평가값을 해당 노드의 값으로 사용한다.
- ④ 루트 노드에 해당하는 플레이어는 자식 노드들 중에서 가장 작은 값을 갖는 노드에 해당하는 행동을 선택하여 실행한다.

❖ $\alpha - \beta$ 가지치기에 대한 설명으로 적합하지 않은 것을 선택하시오.

- ① MIN 노드의 현재값이 부모 노드의 현재 값보다 작거나 같으면, 해당 MIN 노드의 자식 노드는 더 이상 탐색하지 않는다.
- ② 게임 트리에 대한 mini-max 알고리즘을 적용할 때 검토할 필요한 없는 부분을 탐색하지 않도록 하는 기법이다.
- ③ MAX 노드의 현재값이 부모 노드의 현재값보다 같거나 크면, 해당 MAX 노드의 자식 노드를 더 이상 탐색하지 않는다.
- ④ 너비 우선 탐색 방식으로 게임 트리를 만들어가면서 mini-max 알고리즘을 적용할 때 사용되는 탐색 효율화 기법이다.

❖ **몬테카를로 트리 탐색(MCRS)에 대한 설명으로 적합하지 않은 것을 선택하시오.**

- ① 몬테카를로 시뮬레이션을 통해서 게임 상태에 대한 형세 판단값을 계산한다.
- ② 확장 단계에서 시뮬레이션을 통해 단말 노드의 형세 판단값을 계산한다.
- ③ 루트 노드에서부터 트리를 따라 내려갈 때 노드는 승률과 노드 방문횟수를 고려하는데 이때 사용될 수 있는 것이 UCB 정책이다.
- ④ 몬테카를로 시뮬레이션을 해야 하기 때문에 많은 계산 비용 요구된다.

❖ **제약조건 만족 문제 해법에 대한 설명으로 적합하지 않은 것을 선택하시오.**

- ① 백트래킹 탐색에서는 깊이 우선 탐색을 하는 방식으로 변수에 허용되는 값을 대입해보면서 모든 가능한 값을 대입해서 만족하는 것이 없으면 이전 단계로 돌아가서 이전 단계의 변수에 다른 값을 대입해보는 과정을 반복한다.
- ② 제약조건 만족 문제는 주어진 제약조건을 만족하는 조합해를 찾는 것으로 8-퀸 문제 등이 이에 속한다.
- ③ 제약조건 전파 기법에서는 인접 변수 간의 제약 조건에 따라 각 변수에 허용될 수 없는 값을 점진적으로 제거하는 방식으로 해를 찾는다.
- ④ 제약조건 전파 기법에서는 전체적인 제약조건이 동시에 고려되기 때문에 어떠한 제약조건 만족 문제에 대해서도 해를 찾을 수 있다.