인공지능 실습

Chapter 5. Adversarial Search

포항공과대학교 컴퓨터공학과

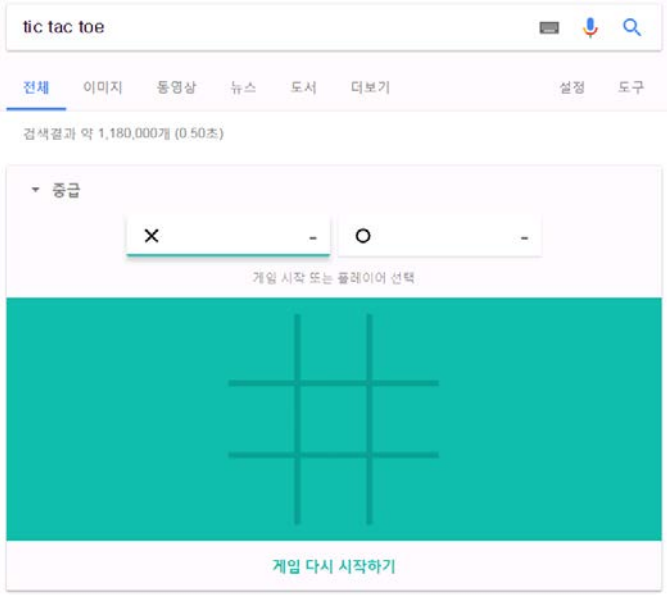
목차

5.1. Adversarial Search 개요

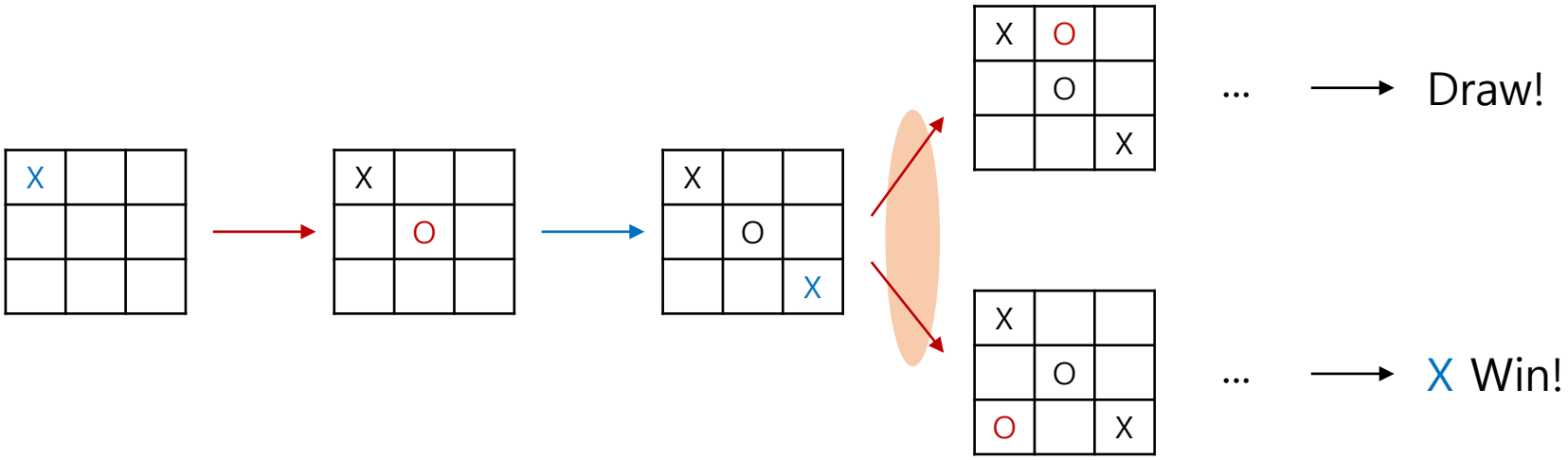
Tic-tac-toe

- 두 명의 player: X(선수) and O(후수)
- 맵: 3 x 3 grid
- 승리조건: 연속된 세 칸에 본인의 돌을 놓으면 승리
- 맵에 빈 자리가 없는데도 승패가 결정이 나지 않으면 비김

구글에서 “tic tac toe” 검색
or
https://playtictactoe.org/



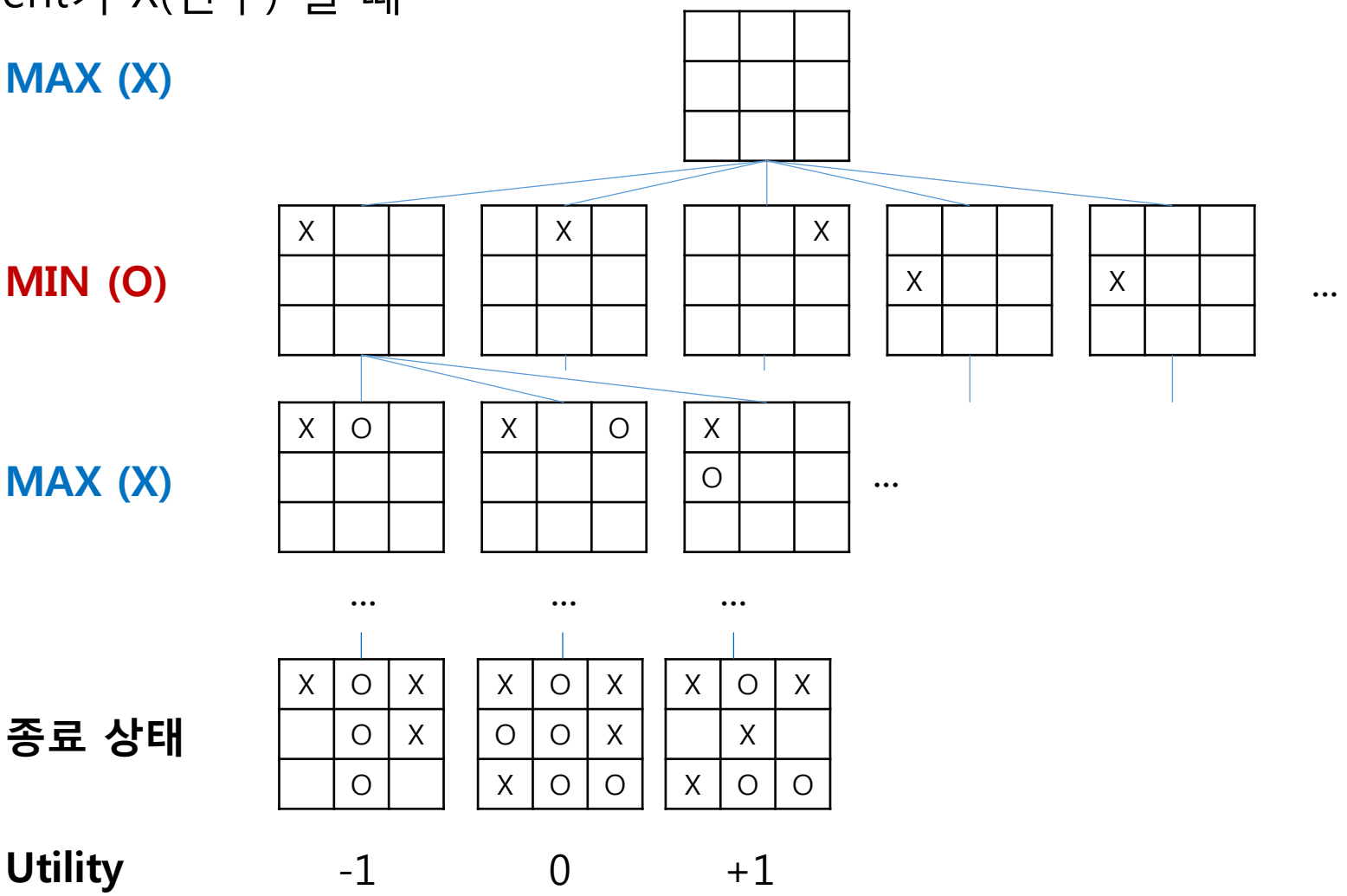
Tic-tac-toe



- 미래의 상대방의 action을 고려해 본인의 action을 결정해야 함

Tic-tac-toe의 Game Tree

- Agent가 X(선수) 일 때



Adversarial Search의 특징

- Deterministic action (v.s. MDP)
- Fully observable environment (v.s. POMDP)
- Zero-sum at the end (v.s. typical search, MDP)
  - 각 플레이어의 **utility**는 서로 동일하거나 반대
  - 중간 action에 대한 utility는 정의되지 않음
    - Chess의 목표는 많은 말을 잡는 것이 아닌 체크메이트(King)를 하는 것이 목적
    - Typical search 및 MDP에 비해 어려운 부분
- Turn-taking of players
  - Typical search에서는 goal state에 도달하기 위한 action sequence를 찾으면 됨
  - Opponent가 agent의 action에 맞춰 대응하므로 action sequence가 아닌 **policy**를 찾아야 함

Policy

- Deterministic policy: $\pi_p(s) = \text{Action}(s)$

  action that player p takes in state s

- Stochastic policy: $\pi_p(s, a) \in [0, 1]$

  probability of player p taking action a in state s

AI Challenges

IBM DeepBlue (Chess) – 1996

Google AlphaGo – 2016

- 사람과의 대결을 통해 AI 기술수준을 보여 준 사례
- 학습의 명확한 기준(승패)이 존재

Adversarial Search의 요소

- s_{start}: 시작 상태
- $\text{Actions}(s)$: s에서 취할 수 있는 모든 action
- $\text{Cost}(s, a)$: s에서 a를 취하는 cost(비용)
- $\text{Succ}(s, a)$: s에서 a를 취했을 때 다음 상태
- $\text{IsEnd}(s)$: s의 종료 조건 만족 여부
- $\text{Utility}(s)$: 종료 상태 s에서 agent의 utility
- $\text{Player}(s)$: s에서 action을 취할 수 있는 player

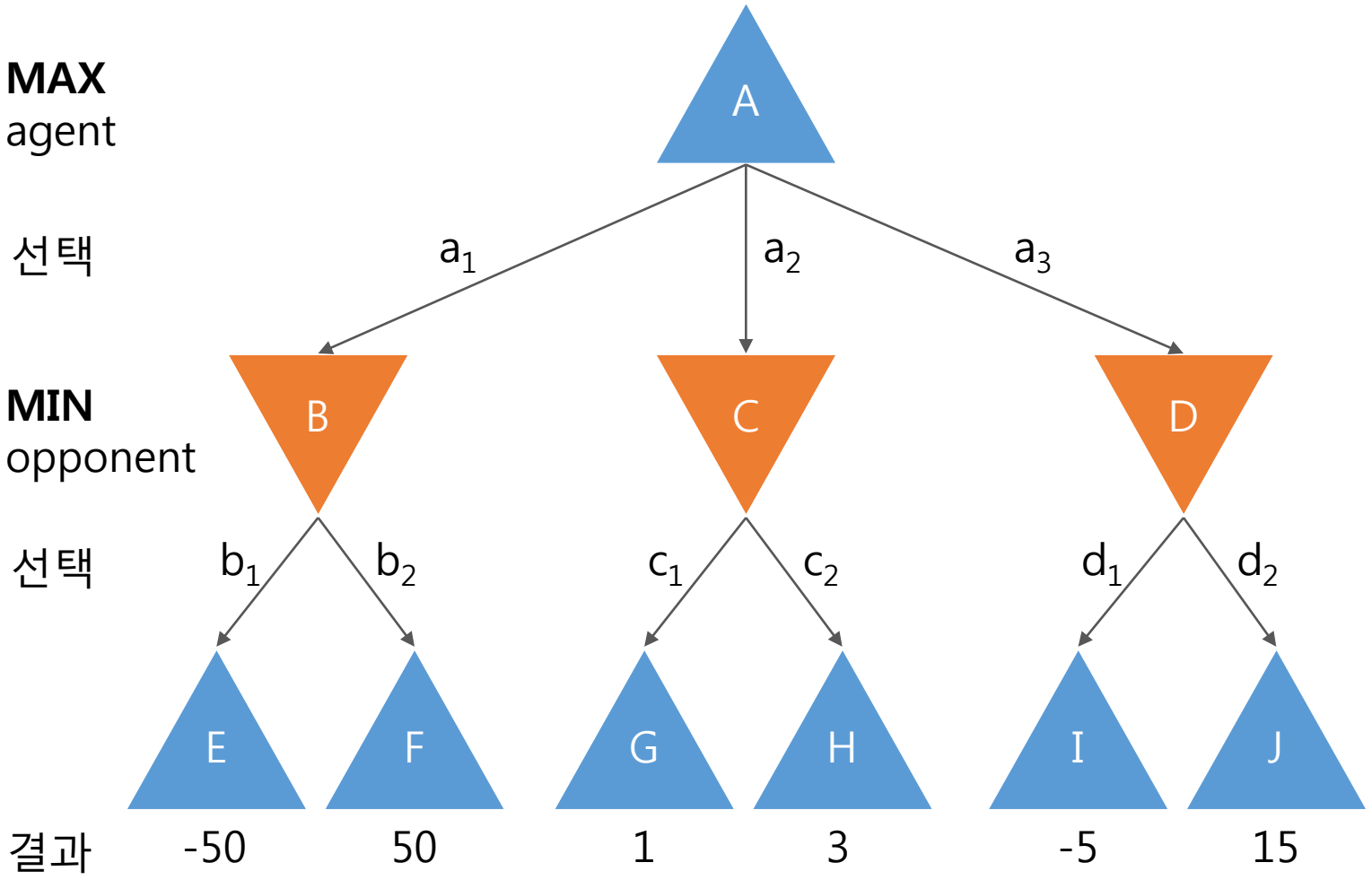
Players = {**agent** (your program); **opp** (opponent)

MAX **MIN**

e.g. 알파고 e.g. 이세돌

Simple Game

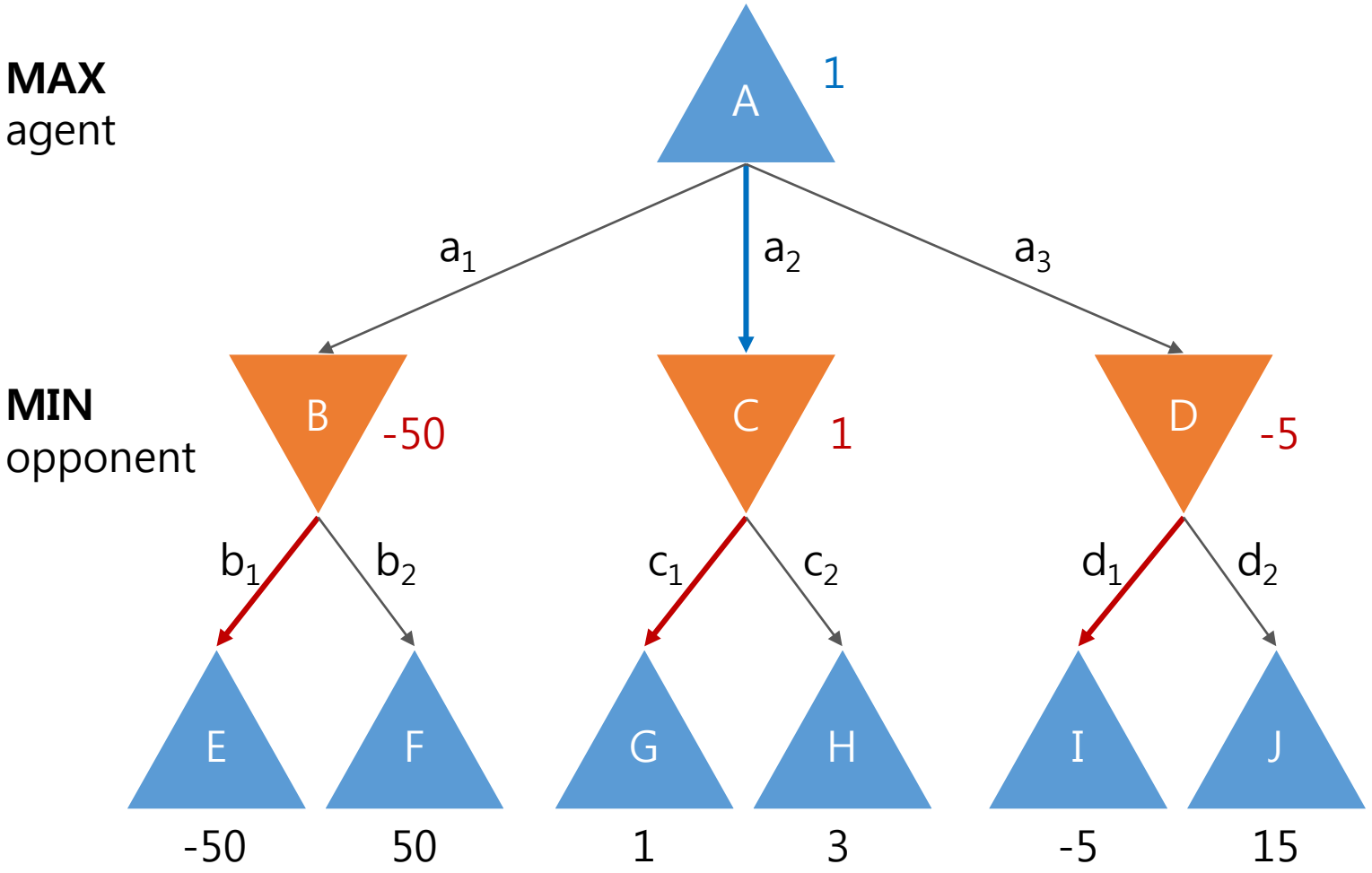
- Agent 및 opponent가 차례로 1회씩 action을 선택하는 게임
  - 상대방의 선택을 알 수 없음



5.2. Minimax & Expectimax Algorithm

Minimax Algorithm 예

- **최악의 경우**를 가정
  - Opponent는 value를 낮추는 선택을 한다



Minimax Algorithm

- Minimax policy

$$\pi_{\max}(s) = \arg\max_{a \in \text{Action}(s)} V_{\max,\min}(\text{Succ}(s,a))$$

$$\pi_{\min}(s) = \arg\min_{a \in \text{Action}(s)} V_{\max,\min}(\text{Succ}(s,a))$$

- Value function for minimax

$$V_{\max,\min}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\max,\min}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\max,\min}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

Minimax Algorithm Pseudocode

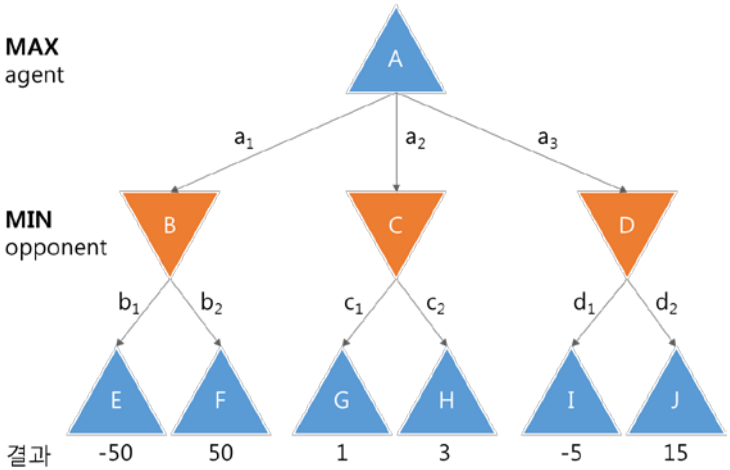

```
def V(state)
    if is_end(state):
        return utility(state)
    if is_maxi_player(state):
        value = -∞
        for action in actions(state):
            value = max(value, V(next_state(state, action)))
    else:
        value = ∞
        for action in actions(state):
            value = min(value, V(next_state(state, action)))
    return value
```

```
def policy(state)
    ...
```

Minimax Algorithm 구현

- Simple Game을 대상으로 구현
  - 상태를 (플레이어, 상태 심볼)로 표현
    - 예. ('MIN', 'C')

- `simple_game/game.py`
  - INT_INF = 100000000000000
  - MAX_PLAYER = 'MAX'
  - MIN_PLAYER = 'MIN'
  - is_end(state)
  - utility(state)
  - get_initial_state()
  - get_next_state(state, action)
  - get_next_player(player)
  - get_possible_actions(state)
  - get_player_from_state(state)



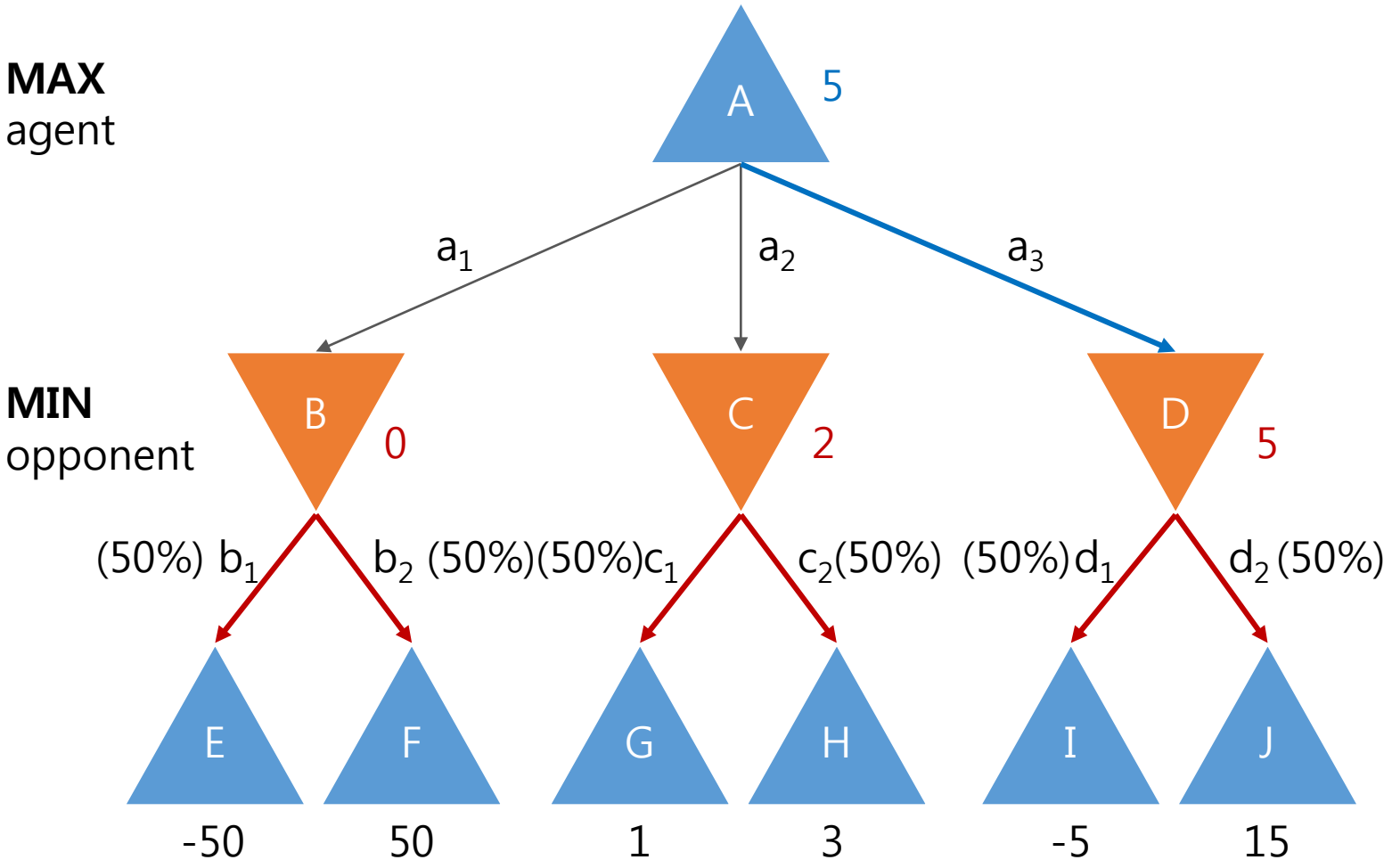
Minimax Algorithm 구현

- `simple_game/play.py`
  - 실행 방법: `play.py minimax`
  - **두 상태에서의 value 및 action 확인**
    - ('MAX', 'A')
    - ('MIN', 'B')

- `simple_game/ai.py`
  - **MinimaxAgent 클래스의 V 및 policy 함수 구현**
  - V(state): 해당 state에서의 value 값 리턴
  - policy(state): policy를 따라 해당 state에서의 action 결정

Expectimax Algorithm 예

- 상대의 **random한 선택**을 가정



Expectimax Algorithm

- Expectimax policy

$$\pi_{\text{max}}(s) = \underset{a \in \text{Action}(s)}{\arg\max} \; V_{\text{max,opp}}(\text{Succ}(s, a))$$

$$\pi_{\text{opp}}(s, a) = \frac{1}{|\text{Action}(s)|} \text{ for } a \in \text{Actions}(s)$$

- Value function for expectimax

$$V_{\text{max,opp}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{max,min}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s, a) V_{\text{max,opp}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{opp} \end{cases}$$

Expectimax Algorithm Pseudocode


```
def V(state)
    if is_end(state):
        return utility(state)
    if is_maxi_player(state):
        value = -∞
        for action in actions(state):
            value = max(value, V(next_state(state, action)))
    else:
        value = 0
        for action in actions(state):
            value += V(next_state(state, action)) / actions(state)
    return value
```

```
# Policy에 따라 결정한 action을 반환
def policy(state)
    ...
```

Expectimax Algorithm 구현

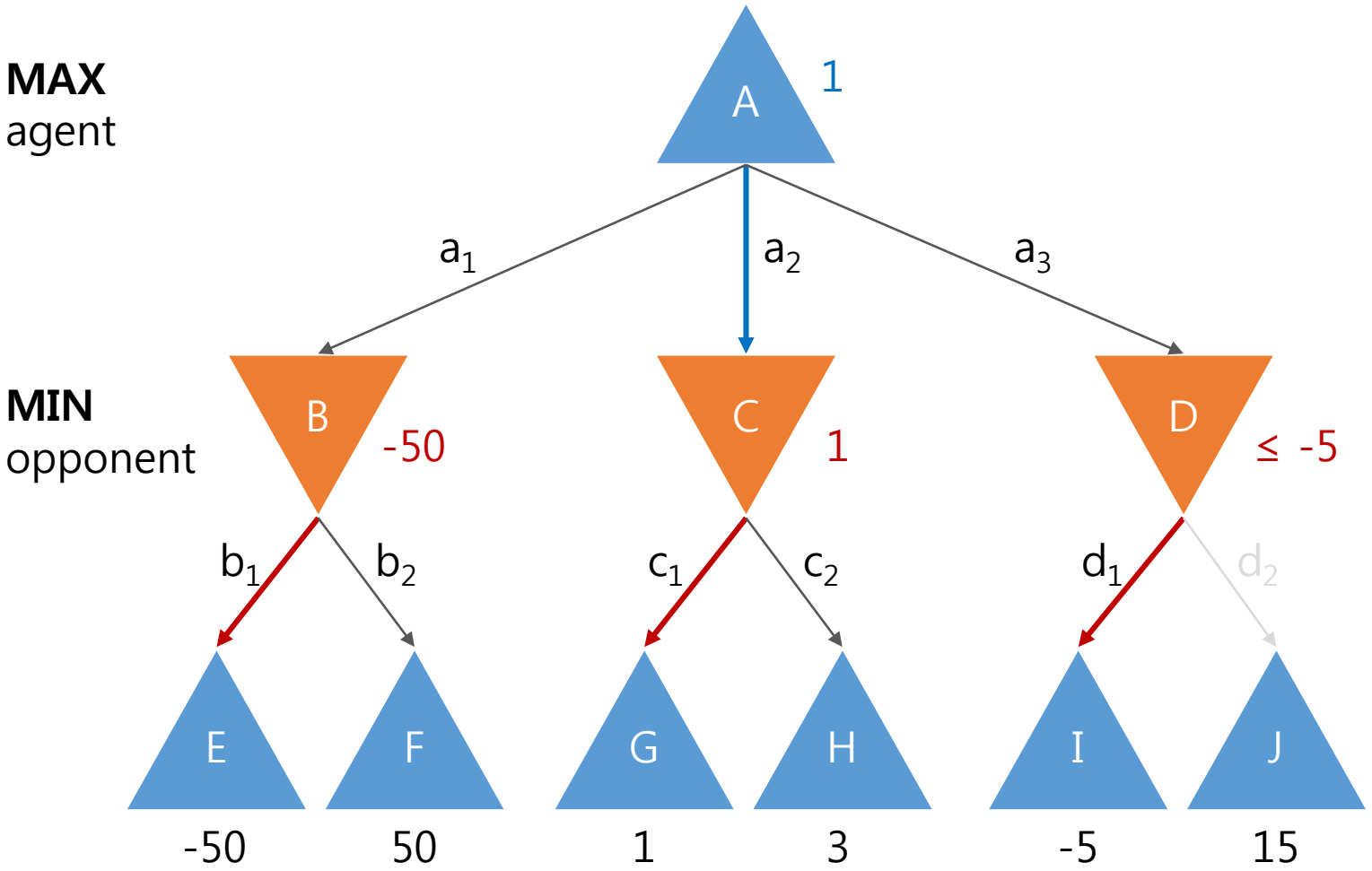
- `simple_game/play.py`
  - 실행 방법: `play.py expectimax`
  - **두 상태에서의 value 및 action 확인**
    - ('MAX', 'A')
    - ('MIN', 'B')

- `simple_game/ai.py`
  - **ExpectimaxAgent 클래스의 V 및 policy 함수 구현**
  - V(state): 해당 state에서의 value 값 리턴
  - policy(state): policy를 따라 해당 state에서의 action 결정

5.3. Pruning

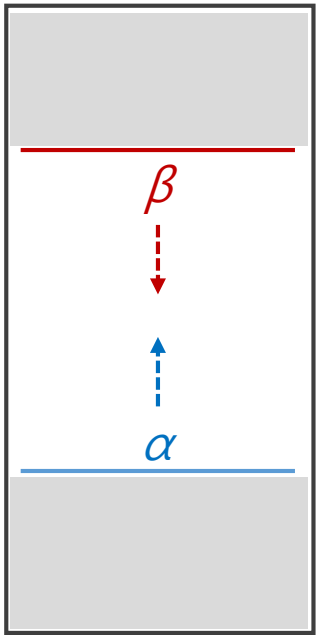
Pruning 예

- d_2는 고려하지 않아도 A에서의 선택은 바뀌지 않음



Alpha-beta Pruning

- Search tree에서 최종 결정에 영향을 안 주는 부분은 무시함으로써 속도 향상
- α : lower bound on value of **max** node s
  - i.e. maximum value found so far
  - Initial α = -∞
  - α = max(α, value)
- β : upper bound on value of **min** mode s
  - i.e. minimum value found so far
  - Initial β = ∞
  - β = min(β, value)
- Pruning: $\beta \leq \alpha$ 가 되면 해당 부분 search 중단



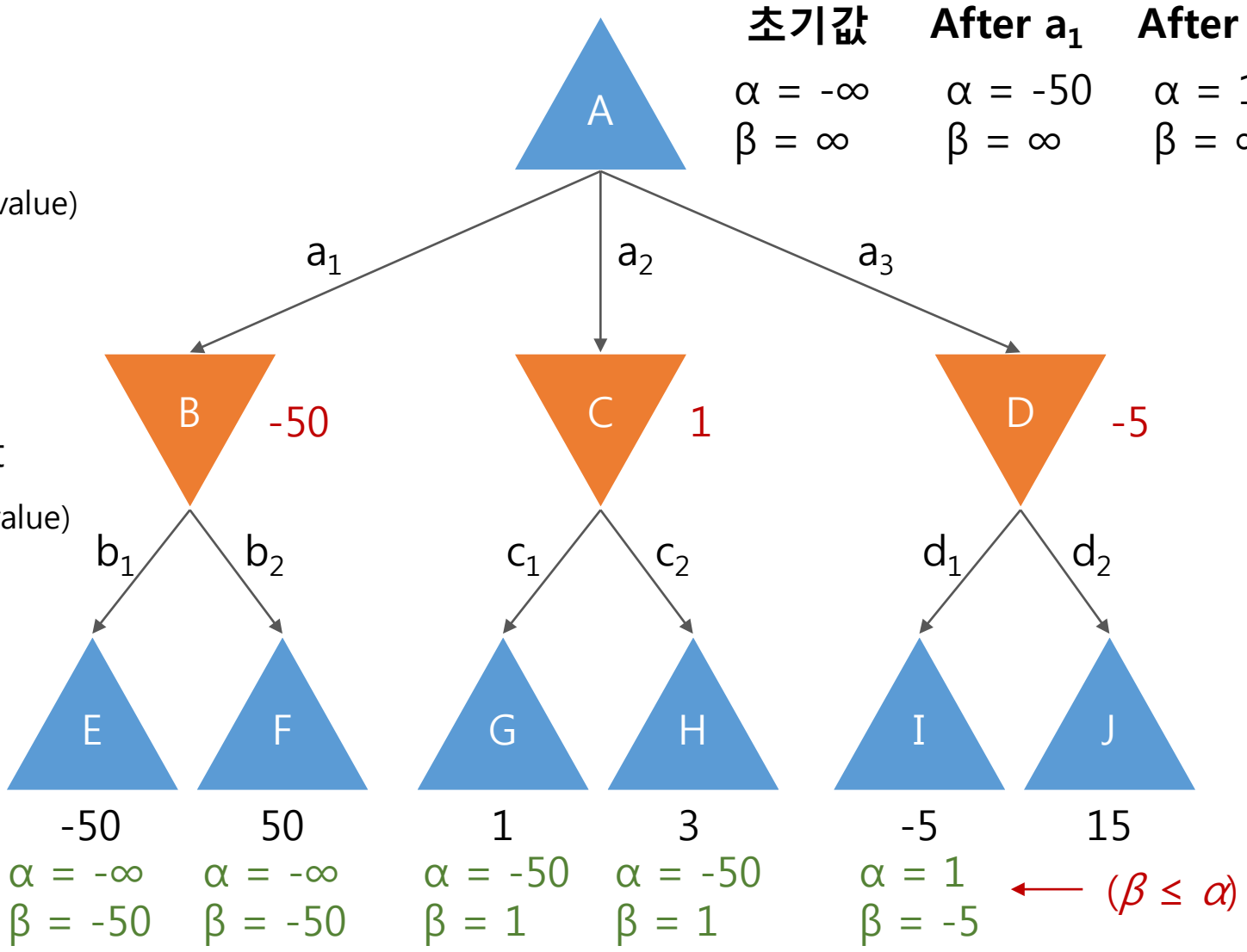
Alpha-beta Pruning 적용 예

MAX
agent

α = max(α, value)

MIN
opponent

β = min(β, value)



Alpha-beta Pruning Pseudocode


```
def V(state, α, β)
    if is_end(state):
        return utility(state)
    if is_maxi_player(state):
        value = -∞
        for action in actions(state):
            value = max(value, V(next_state(state, action), α, β)
            α = max(α, value)
            if β ≤ α: break
    else:
        value = ∞
        for action in actions(state):
            value = min(value, V(next_state(state, action), α, β)
            β = min(β, value)
            if β ≤ α: break
    return value
```

Alpha-beta Pruning 구현

- `simple_game/play.py`
  - 실행 방법: `play.py pruning`
  - **두 상태에서의 value 및 action 확인**
    - ('MAX', 'A')
    - ('MIN', 'B')
  - **V 함수 호출 회수 minimax와 비교**

- `simple_game/ai.py`
  - **PruningMinimaxAgent 클래스의 V 및 policy 함수 구현**
  - V(state): 해당 state에서의 value 값 리턴
  - policy(state): policy를 따라 해당 state에서의 action 결정

Tic-tac-toe 구현

1	2	3
4	5	6
7	8	9

- Dictionary 형태의 상태표현
  - state['PLAYER']: player 저장
  - state[1]: 위치 1에 놓인 돌 ∈ ['X', 'O', '']
  - …
  - state[9]: 위치 9에 놓인 돌 ∈ ['X', 'O', '']

Tic-tac-toe 구현

- `tic_tac_toe/game.py`: Tic-tac-toe 게임 관련 유틸리티
- `tic_tac_toe/ai.py`
  - **simple_game의 AI 코드를 그대로 복사**
- `tic_tac_toe/play.py`: AI와 Tic-tac-toe 게임 플레이
  - `play.py minimax`
  - `play.py pruning`

- **AI의 quality 확인**

- **Minimax와 alpha-beta pruning의 탐색 시간 비교**
  - Minimax는 AI가 선플레이어인 경우 굉장히 느림

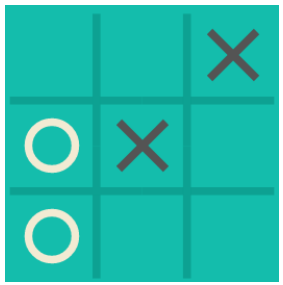
Tic-tac-toe 구현

Tic-tac-toe 최선의 전략
선수: 모서리나 중앙으로 공격
후수:
- 상대방이 모서리에 둔 경우 중앙으로 방어
- 상대방이 중앙에 둔 경우 모서리로 방어

- 플레이를 통해 AI가 위 전략을 구사하는지 확인
- 개발자가 이러한 전략을 모르더라도 최선의 전략을 구사하는 AI를 구현할 수 있다
    - 규칙(전문가) v.s. 계산 및 데이터

5.4. Imperfect Real-time Decisions

Imperfect Real-time Decisions



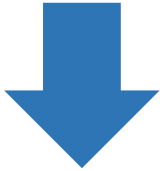
Tic-tac-toe

- Search tree 최대 노드 = 9!

Chess

- 평균 action 수 = 35
- 평균 턴 수 = 50
- Search tree 평균 노드 수 = 35^{100}

복잡한 문제에서는 트리의 leaf를 보고 action을 결정하는 것이 불가능!



Imperfect real-time decisions

Depth-limited Search

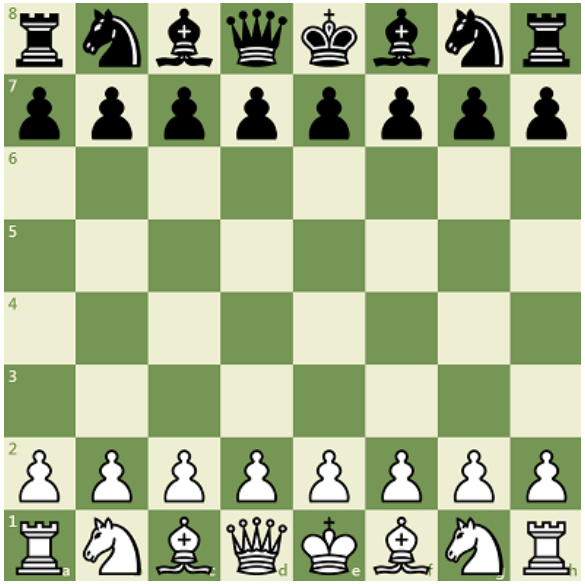
- Value function for depth-limited search

$$V_{\max,\min}(s,d) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \text{Eval}(s) & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\max,\min}(\text{Succ}(s,a), d) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\max,\min}(\text{Succ}(s,a), d-1) & \text{Player}(s) = \text{opp} \end{cases}$$

- Evaluation function Eval(*s*)
  - $V_{max,min}(s)$에 대한 추정(estimation)값
  - Search 문제에서의 FutureCost(*s*)와 유사한 개념

- Use: at state *s*, call $V_{max,min}(s, d_{max})$

Evaluation Function 예. Chess



Chess pieces

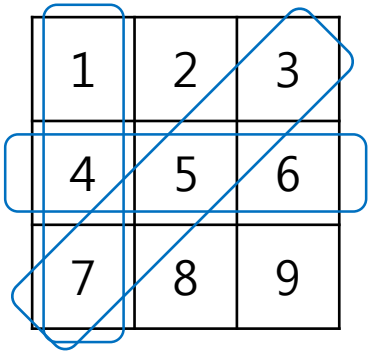
- King (∞)
- Queen (9)
- Rook (5)
- Bishop (3)
- Knight (3)
- Pawn (1)

- Eval(s) = material + mobility + king-safety + center-control
  - material = $10^{100}(K - K') + 9(Q - Q') + 5(R - R') + 3(B - B' + N - N') + 1(P - P')$
  - mobility = 0.1 (num-legal-moves – num-legal-moves')

- 체스에서는 이러한 hand-tuned evaluation function이 잘 작동함

Evaluation Function for Tic-tac-toe

Board



Heuristic array

MAX	MIN			
	0	**1**	**2**	**3**
0	0	-10	-100	-1000
1	10	0	0	0
2	100	0	0	0
3	1000	0	0	0

```
def eval(state):
    value = 0
    for 승리조건 of board:
        maxs = 승리조건에 놓여있는 MAX의 돌 수
        mins = 승리조건에 놓여있는 MIN의 돌 수
        value += heuristic[maxs][mins]
    return value
```

https://kartikkukreja.wordpress.com/2013/03/30/heuristic-function-for-tic-tac-toe/

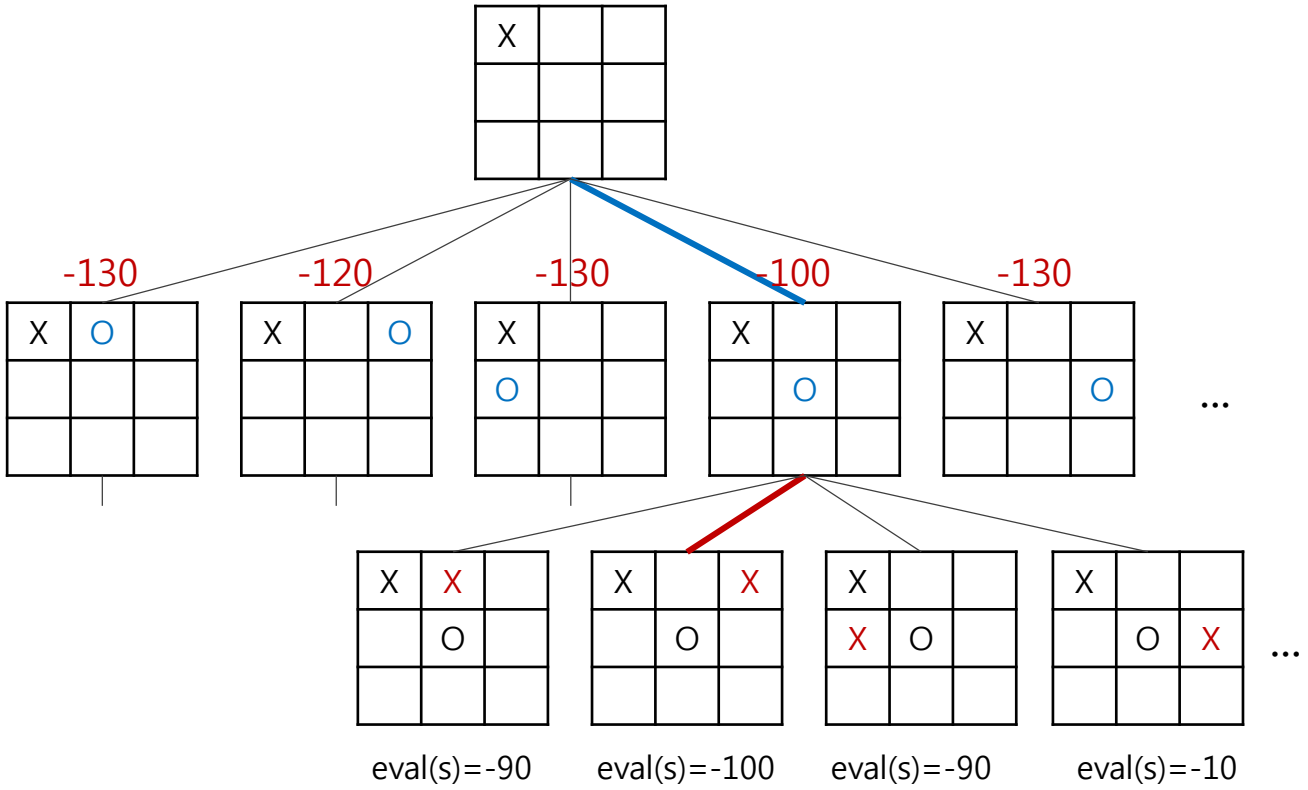
Evaluation Function for Tic-tac-toe

- Agent가 후수(O)

d=1 **MAX (O)** agent

d=1 **MIN (X)** opponent

d=0 **MAX (O)** agent



Agent는 O를 중앙(5)에 배치

Tic-tac-toe 구현 (Alpha Tic-tac-toe!)

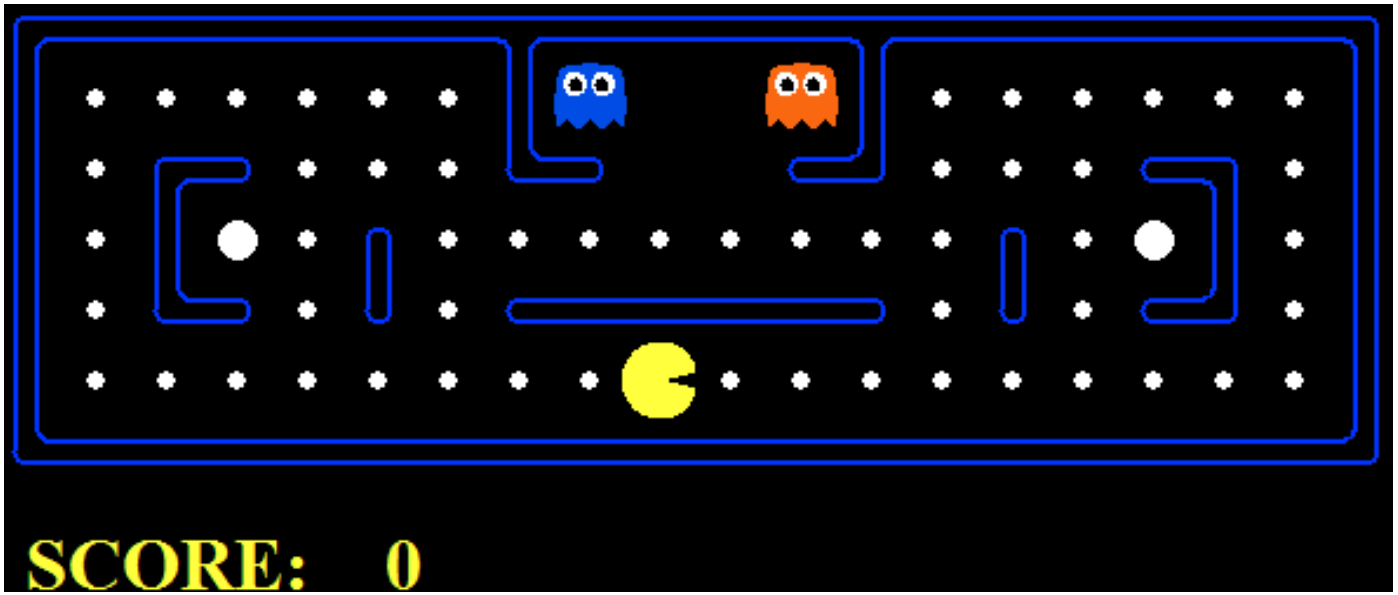
- `tic_tac_toe/game.py`: Tic-tac-toe 게임 관련 유틸리티
- `tic_tac_toe/ai.py`: **DepthLimitedMinimaxAgent 구현**
  - Minimax를 복사해 수정
  - 위 evaluation function을 구현한 eval(s) 사용
- `tic_tac_toe/play.py`: AI와 Tic-tac-toe 게임 플레이
  - `play.py limited 1`

- **선택되는 action의 quality 및 탐색 속도 비교 (v.s. Minimax)**

- **Alpha-beta pruning에도 적용 (with higher depth)**

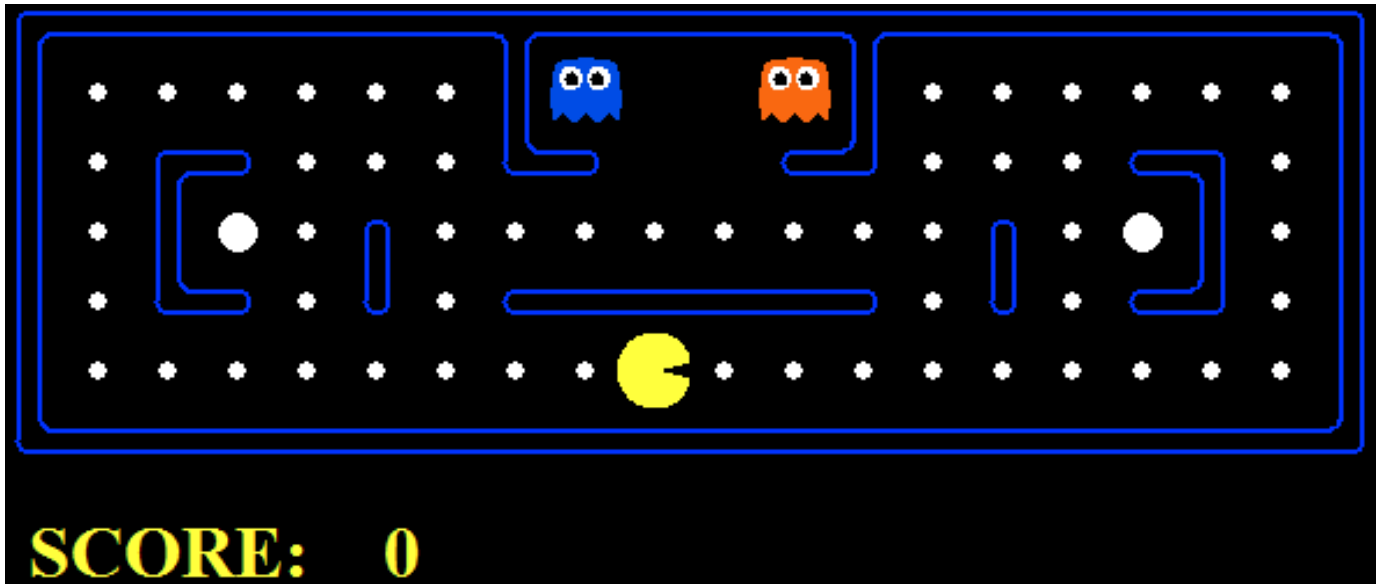
Evaluation Function Design

- Pacman (d = 2) with **simple** evaluation function
  - 두 칸 앞 까지 밖에 못 봄



Evaluation Function Design

- Pacman (d = 2) with **better** evaluation function
  - 추가 feature: Ghost와의 거리, Food와의 거리, 등등...



Parameterized Evaluation Function

- Feature vector: $\phi(s) \in \mathbb{R}^d$

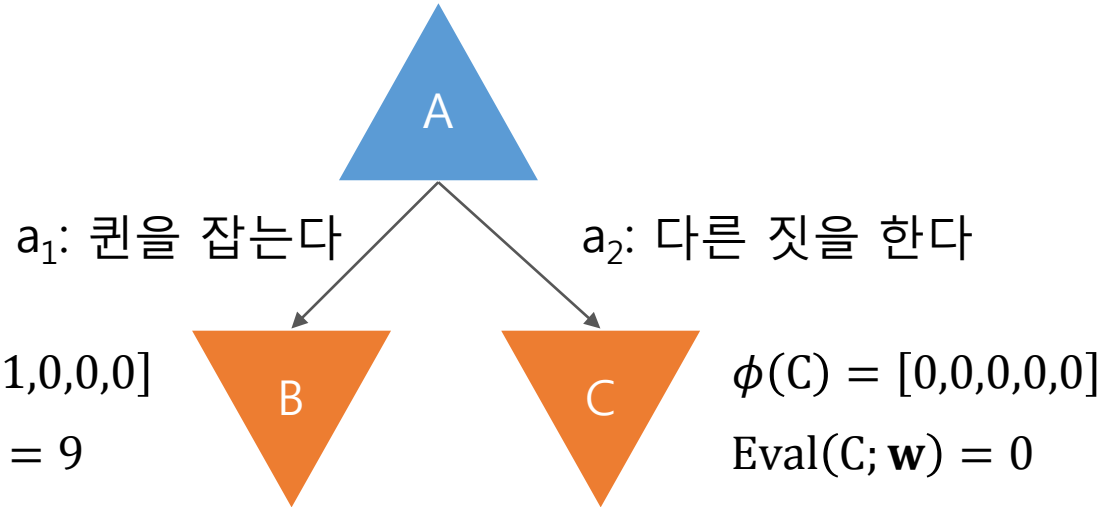
$$\phi_1(s) = K - K' \qquad \phi_2(s) = Q - Q' \qquad \dots$$

- Linear evaluation function

$$\text{Eval}(s; \mathbf{w}) = w \cdot \phi(s)$$

예. $w = [10^{100}, 9,5,3,1]$

MAX
agent



MIN
opponent

$\phi(\text{B}) = [0,1,0,0,0]$

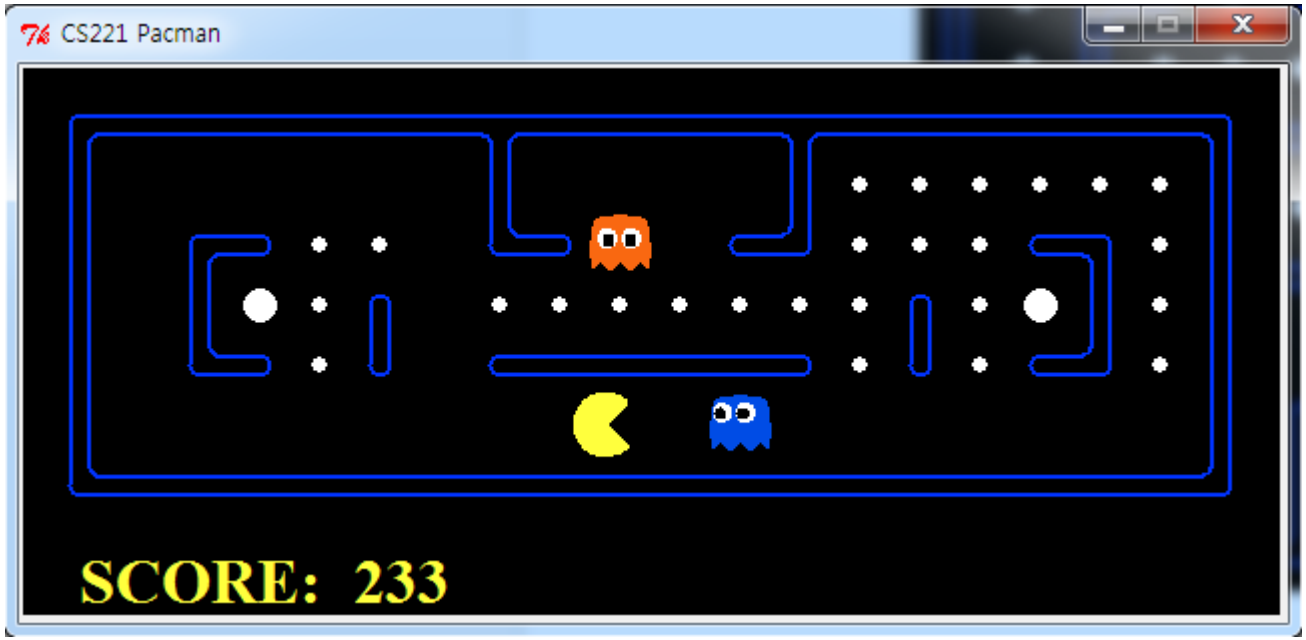
$\text{Eval}(\text{B}; \mathbf{w}) = 9$

$\phi(\text{C}) = [0,0,0,0,0]$

$\text{Eval}(\text{C}; \mathbf{w}) = 0$

5.5. Multi-player Game

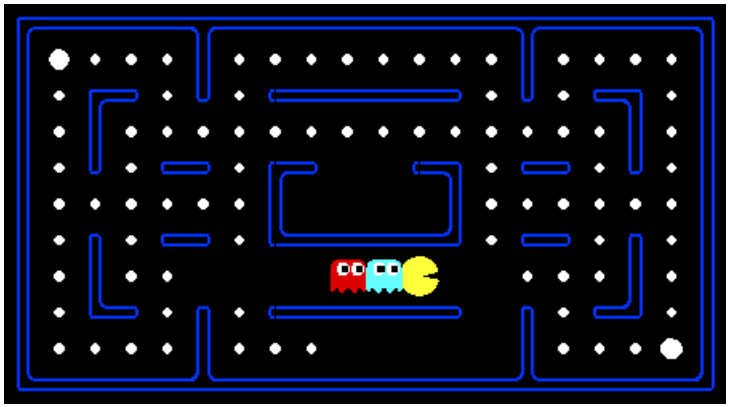
Play Pacman!



- Play Pacman manually!
  - `python pacman.py -l smallClassic`
- 화살표키로 이동

Pacman 게임 설정

- Pacman with multiple ghosts
  - a_0: Agent
  - a_1, ... a_n: 각각의 ghost
    - Multiple MIN layers (one for each ghost)
- 승리조건: 모든 food를 먹음
- 점수
  - 1프레임에 -1점
  - Food를 먹으면 +10점
  - 승리하면 +500점
  - 패배하면 -500점



- 패배조건: Ghost와 만남
- Capsule을 먹으면 일정 시간 동안 Ghost에 면역 (+200점)

Minimax and Expectimax

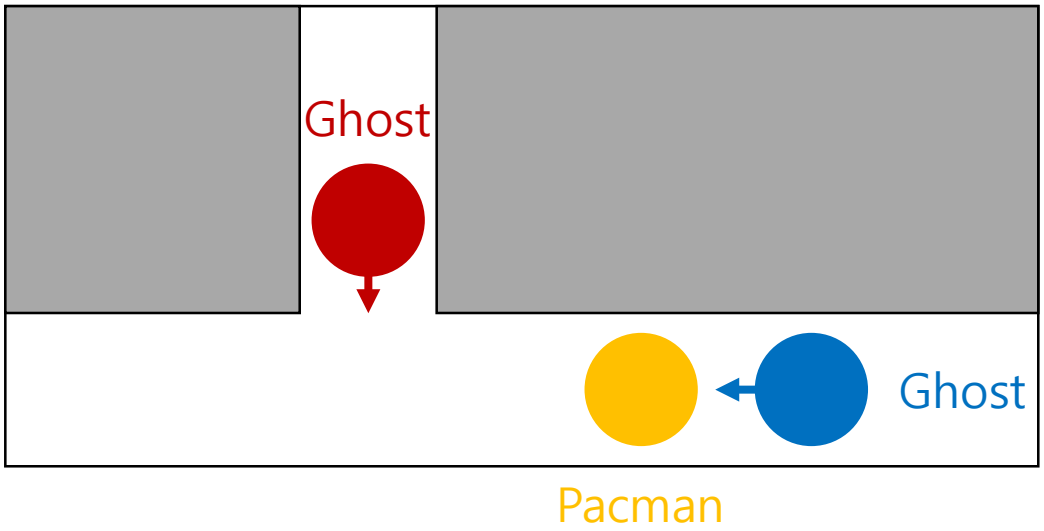
- Value function for multi-player **minimax** game

$$V_{\text{max,min}}(s,d) = \begin{cases} \text{Utility}(s) & \text{if IsEnd}(s) \\ \text{Eval}(s) & \text{if } d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{max,min}}(\text{Succ}(s,a),d) & \text{if Player}(s) = a_0 \\ \min_{a \in \text{Actions}(s)} V_{\text{max,min}}(\text{Succ}(s,a),d) & \text{if Player}(s) = a_1 \ldots, a_{n-1} \\ \min_{a \in \text{Actions}(s)} V_{\text{max,min}}(\text{Succ}(s,a),d-1) & \text{if Player}(s) = a_n \end{cases}$$

- Value function for multi-player **expectimax** game

$$V_{\text{max,opp}}(s,d) = \begin{cases} \text{Utility}(s) & \text{if IsEnd}(s) \\ \text{Eval}(s) & \text{if } d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{max,opp}}(\text{Succ}(s,a),d) & \text{if Player}(s) = a_0 \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s,a) V_{\text{max,opp}}(\text{Succ}(s,a),d) & \text{if Player}(s) = a_1 \ldots, a_{n-1} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s,a) V_{\text{max,opp}}(\text{Succ}(s,a),d-1) & \text{if Player}(s) = a_n \end{cases}$$

Minimax and Expectimax



- 다음 turn에서 Pacman의 action은? west or east?
  - Minimax case
  - Expectimax case
- `python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3`

Codes

- `pacman.py`
  - -p <Agent 타입>: ReflexAgent, MinimaxAgent, AlphaBetaAgent, ExpectimaxAgent
  - -l <Map 타입>: mediumClassic, openClassic
  - -a depth=<숫자>
  - 예. `python pacman.py -p ReflexAgent -l mediumClassic`
  - 예. `python pacman.py -p MinimaxAgent -l mediumClassic -a depth=2`
- `submission.py`
  - Agents
    - ReflexAgent
    - MinimaxAgent: minimax
    - AlphaBetaAgent: minimax + alpha-beta pruning
    - ExpectimaxAgent: expectimax
  - Agent의 getAction(gameState) 함수로 action 결정
  - 모든 agent는 기본적으로 depth-limited search

제공되는 변수 및 함수

- `gameState.getLegalActions()`
  - Returns the legal actions for the agent specified
  - Subset of [NORTH, SOUTH, EAST, WEST]
- `gameState.generateSuccessor(agentIndex, action)`
  - Returns the successor state after the specified agent takes the action
  - Pac-Man is always agent 0
- `gameState.isWin()` and `gameState.isLose()`
- `gameState.getNumAgents()`
  - Returns the total number of agents in the game
- `gameState.getScore()`
  - Returns the score corresponding to the current state of the game
- `self.evaluationFunction(gameState)`
  - Returns the game score of gameState
- `self.index`: Player index
- `self.depth`: Maximum depth

예. ReflexAgent


```
class ReflexAgent(Agent):
  def getAction(self, gameState):
    # Collect legal moves and successor states
    legalMoves = gameState.getLegalActions()

    # Choose one of the best actions
    scores = [self.evaluationFunction(gameState, action) for action in legalMoves]
    bestScore = max(scores)
    bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
    chosenIndex = random.choice(bestIndices) # Pick randomly among the best

    return legalMoves[chosenIndex]
```

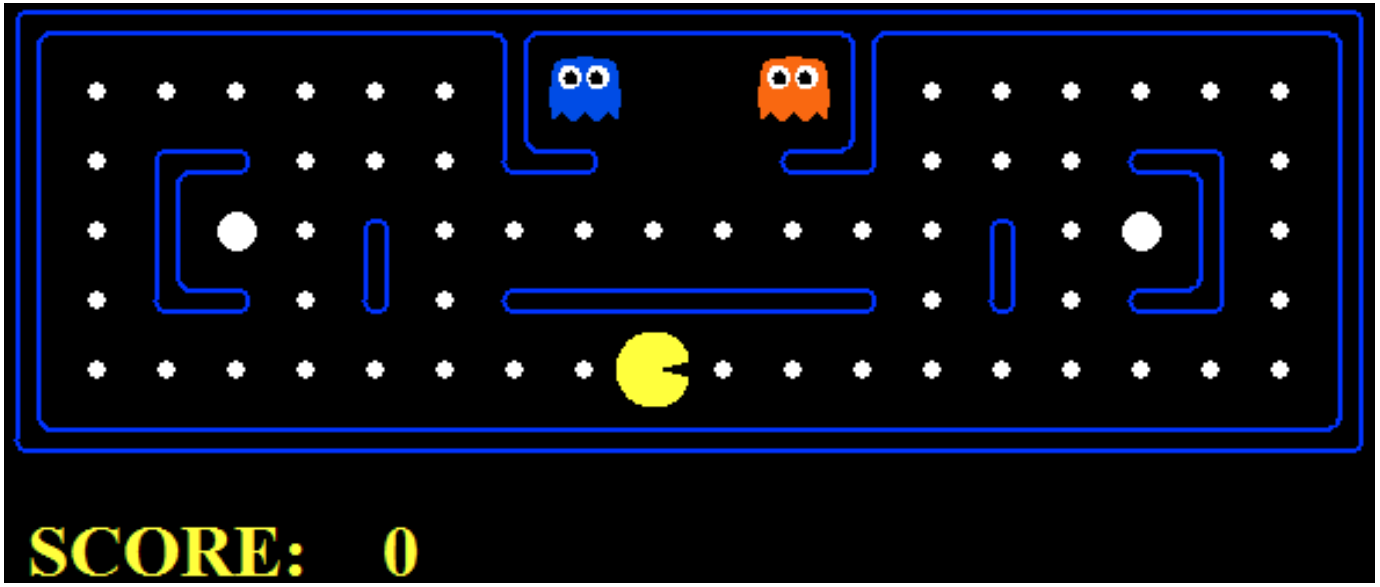
- getAction 함수: gameState에서 최선의 action을 return

Multi-player Game 구현

- **MinimaxAgent의 getAction 함수 구현**
  - `python pacman.py -p MinimaxAgent -l smallClassic`
- **AlphaBetaAgent의 getAction 함수 구현**
  - `python pacman.py -p AlphaBetaAgent -l smallClassic -a depth=3`
- **ExpectimaxAgent의 getAction 함수 구현**
  - `python pacman.py -p ExpectimaxAgent -l smallClassic`

Pacman Competition

- Pacman (d = 2) with **better** evaluation function
  - 추가 feature: Ghost와의 거리, Food와의 거리, 등등...



- `python pacman.py -l smallClassic -p ExpectimaxAgent -a evalFn=better -q -n 20`