
Microchess AI Competition Documentation

출시 0.1.0

Hyunsoo Park

2018년 05월 02일

Contents

1 빠른 시작	3
1.1 설치방법	3
1.2 AI 구현	3
1.3 AI끼리 게임 플레이	4
1.4 AI와 게임 플레이	5
1.5 게시판	5
2 Tutorial	7
2.1 Microchess AI 경진대회	7
2.1.1 경진대회 규칙	7
2.1.2 점수계산	9
2.1.3 주의사항	10
2.1.4 운영 게시판	10
2.1.5 최종 우승자 결정	11
2.2 Microchess AI 플랫폼	12
2.2.1 개요	12
2.2.2 디렉토리 구조	12
2.2.3 설치방법	12
2.2.4 실행 방법	13
2.2.5 인간 플레이어 vs. AI 인터페이스	17
2.3 기본 AI 예제	18
2.3.1 Random AI	18
2.3.1.1 가장 간단한 AI 구현	18
2.3.1.2 act 메소드의 입력과 출력 요약	19
2.3.2 Search 기반 AI	19
2.3.2.1 One Step Search AI	20
2.3.2.2 Two Step Search AI	21
2.3.2.3 Two Step Search AI 구현	22
2.3.2.4 Negamax Search AI	23
2.3.2.5 Negamax Search AI + $\alpha - \beta$ pruning	24
2.3.2.6 Microchess 상태공간 규모	24
2.3.2.7 평가함수	26
2.3.2.8 Stockfish	27
2.4 Monte Carlo Tree Search	29
2.4.1 MCTS 개요	29
2.4.2 Multi-Armed Bandit	29

2.4.3	Upper Confidence Tree	31
2.4.4	MCTS 예제	32
2.4.5	MCTS 시각화	33
2.5	Self Learning AI	34
2.5.1	Self Learning 알고리즘	34
2.5.2	인공신경망	35
2.5.3	MCTS	36
2.5.4	결론 및 한계	37
3	API	39
3.1	scripts package	39
3.1.1	Submodules	39
3.1.2	scripts.chess_board module	39
3.1.3	scripts.run_game module	40
3.1.4	scripts.utils module	42
3.1.5	Module contents	44
3.2	agents package	44
3.2.1	Module contents	44
3.3	agents.basic package	44
3.3.1	Submodules	44
3.3.2	agents.basic.debug_agent module	44
3.3.3	agents.basic.human module	45
3.3.4	agents.basic.random_agent module	46
3.3.5	Module contents	46
3.4	agents.search package	46
3.4.1	Submodules	46
3.4.2	Module contents	46
3.4.3	agents.search.one_step_search_agent module	46
3.4.4	agents.search.two_step_search_agent module	47
3.4.5	agents.search.negamax_search_agent module	47
3.4.6	agents.search.abp_negamax_search_agent module	48
3.4.7	agents.search.mcts_agent module	49
3.5	agents.self_learning package	51
3.5.1	Submodules	51
3.5.2	Module contents	51
3.5.3	agents.self_learning.agent module	52
3.5.4	agents.self_learning.models module	54
3.5.5	agents.self_learning.memory module	54
3.5.6	agents.self_learning.mcts module	55
3.5.7	agents.self_learning.utils module	56
3.6	agents.stockfish package	58
3.6.1	Submodules	58
3.6.2	Module contents	58
3.6.3	agents.stockfish.agent module	58
Python 모듈 목록	61	

- 이 플랫폼은 Microchess를 플레이하는 AI 개발 경진대회에 사용하기 위한 목적으로 개발되었다.
- 이 플랫폼은 크게 Microchess 게임과 예제 AI들 그리고 실행 스크립트로 구성되어 있으며, Windows 10 환경에서 python 3.5 (anaconda)를 사용하여 개발 테스트 되어있다.
- 플랫폼이 python으로 구현되어 있기 때문에 python으로 AI를 구현하는 것이 권장되지만, 다른 언어로 구현한 AI를 사용할 수도 있다.

CHAPTER 1

빠른시작

1.1 설치방법

이 문서에서는 플랫폼 개발환경과 같은 Windows 10 + python 3.5 (anaconda)를 기준으로 설명한다. 하지만, 대부분의 구성요소들이 순수 python 으로 구현되어 있기 때문에, 다른 운영체제에서 큰 문제없이 실행 가능하다.

설치 순서

1. 플랫폼 다운로드 및 압축해제
2. anaconda 다운로드 및 설치 (<https://www.anaconda.com/download/>)
3. 가상환경 생성 및 활성화:

```
(base) C:\Users\user\MicrochessAICompetition> conda create -n mchess python=3.5
(base) C:\Users\user\MicrochessAICompetition> activate mchess
(mchess) C:\Users\user\MicrochessAICompetition>
```

4. 주요 모듈 설치:

```
(mchess) C:\Users\user\MicrochessAICompetition> conda install numpy scipy ipython_
˓→tqdm pyyaml mkl matplotlib
(mchess) C:\Users\user\MicrochessAICompetition> conda install -c CogSci pygame
(mchess) C:\Users\user\MicrochessAICompetition> pip install sqlitedict
(mchess) C:\Users\user\MicrochessAICompetition> pip install visdom # visdom 설치
(mchess) C:\Users\user\MicrochessAICompetition> conda install -c peterjc123_
˓→pytorch-cpu=0.3.1 # pytorch 설치 (CPU, Windows 용)
```

1.2 AI 구현

AI를 구현할 때는 BaseAgent (agents/__init__.py)를 상속받아, reset, act, close 세 가지 메소드를 구현한다. 가장 간단한 예인 Random AI (agents/basic/random_agent.py)는 다음과 같이 구현한다.

```

import random
from agents import BaseAgent

class RandomAgent(BaseAgent):
    """
    Random AI

    - 무작위로 행동을 결정하는 예제
    """

    def reset(self):
        pass

    def act(self, state):
        moves = list(state.legal_moves)
        return random.choice(moves)

    def close(self):
        pass

```

이 세 가지 메소드는 플랫폼에 의해 호출된다.

reset와 close는 AI의 초기화와 종료를 처리한다. 게임을 시작할 때와 끝낼 때 한번씩만 실행된다. act는 AI의 턴마다 한번씩 실행된다. 현재 게임상태(state)를 입력받아, 다음 수(move)를 출력해야 한다.

1.3 AI끼리 게임 플레이

AI끼리 게임을 플레이할 때는 scripts/run_game.py 을 사용한다. Random AI (white)와 One Step Search AI (black)끼리 플레이할 때는 다음과 같이 실행한다.:

```

python scripts/run_game.py --white=agents.basic.random_agent.RandomAgent --
                           ↪black=agents.search.one_step_search_agent.OneStepSearchAgent

```

white와 black에 AI의 경로를 지정해 주면 게임을 바로 실행하고, 결과를 출력해 준다.

One Step AI는 다음 수의 결과를 고려하는 간단한 AI이다. 강력한 AI는 아니지만, Random AI는 쉽게 이길 수 있다. 실행한 결과는 다음과 같이 나타난다.

```

1 [2018-04-23 15:40:07,079 DEBUG] White: agents.basic.random_agent.RandomAgent-True
2 [2018-04-23 15:40:07,079 DEBUG] White: board value 0.500
3 <IPython.core.display.SVG object>
4 k n b r
5 p . . .
6 . . . .
7 . . . p
8 R B N K
9 [2018-04-23 15:40:07,093 DEBUG] White: 400.0 sec. remain
10 [2018-04-23 15:40:07,093 DEBUG] White: move b1d3
11 [2018-04-23 15:40:07,093 DEBUG] 8/8/8/knbr4/p7/3B4/3P4/R1NK4 b Kk - 1 1
12 ... (생략)
13 [2018-04-23 18:05:27,443 DEBUG] White: move a1a2
14 [2018-04-23 18:05:27,443 DEBUG] 8/8/8/8/k7/8/K7/2b5 b - - 0 14
15 [2018-04-23 18:05:27,443 DEBUG] [0, 1]
16 [2018-04-23 18:05:27,443 DEBUG] Black Win

```

(continues on next page)

(이전 페이지에서 계속)

```

17 <IPython.core.display.SVG object>
18 . . .
19 k . .
20 . . .
21 K . .
22 . . b .
23 [2018-04-23 18:05:27,449 DEBUG] Score: [0.000 1.000]
24 [2018-04-23 18:05:27,449 DEBUG] turns: 26
25 [2018-04-23 18:05:27,449 DEBUG] Game end: True
26 [2018-04-23 18:05:27,449 DEBUG] checkmate: False
27 [2018-04-23 18:05:27,449 DEBUG] Stalemate: False
28 [2018-04-23 18:05:27,449 DEBUG] Insufficient material: True
29 [2018-04-23 18:05:27,449 DEBUG] 57 moves: False
30 [2018-04-23 18:05:27,449 DEBUG] 5-fold: False
31 [2018-04-23 18:05:27,450 DEBUG] Black win: [0, 1]

```

4~8번째 줄처럼 매 턴마다, 게임 상태를 텍스트 상태로 출력한다. white는 대문자로 표시하고, black은 소문자로 표시한다.

23번째 줄부터 게임 결과를 출력한 것이다. Score의 첫번째 숫자는 white의 점수를 나타내고, 두 번째 숫자는 black의 점수를 나타낸다. (특별한 경우가 아니면 플랫폼의 나머지 부분에서도 white, black 순서로 출력함) 게임에 승리하면 1.0, 패배하면 0.0, 그리고 비기면 0.5점을 얻는다.

일반 Chess (Microchess 포함)와 달리 이 플랫폼에서는 승패가 결정되지 않으면, 게임이 종료되었을 때 남아있는 기물의 점수를 계산하여 최종 승패를 판단한다. 따라서, 기존 무승부 조건을 만족한 상태에서 기물의 점수도 동일한 경우만 무승부가 가능하다.

turns는 전체 게임의 턴 수를 보여주고, 그 다음 줄의 game end는 게임이 정상적으로 종료되었는지 여부를 알려준다. 그 다음부터는 게임이 종료된 이유(checkmate, stalemate, 등)를 알려준다.

AI 끼리의 성능을 평가할 때는 benchmark 옵션을 사용한다. benchmark 옵션을 사용하면, 지정된 white와 black 옵션과 상관없이 총 20게임 (지정된 white와 black으로 10게임, white와 black을 뒤집어서 10게임)을 플레이하고 전체 승률을 출력해준다.

1.4 AI와 게임 플레이

개발한 AI의 성능을 평가하기 위해 직접 게임을 플레이해봐야 할 필요가 있다. 다음과 같이 플랫폼을 실행하면 직접 게임을 플레이 해 볼 수 있다.:

```
python scripts/run_game.py --white=human --black=one_step_search
```

일반 AI대신 human (agents.basic.human.Player)을 인자로 주면, 게임을 직접 플레이 할 수 있는 인터페이스 게임 플레이 인터페이스 가 활성화 된다.

1.5 게시판

- 경진대회 운영 게시판

- https://groups.google.com/forum/#!forum/microchess_ai

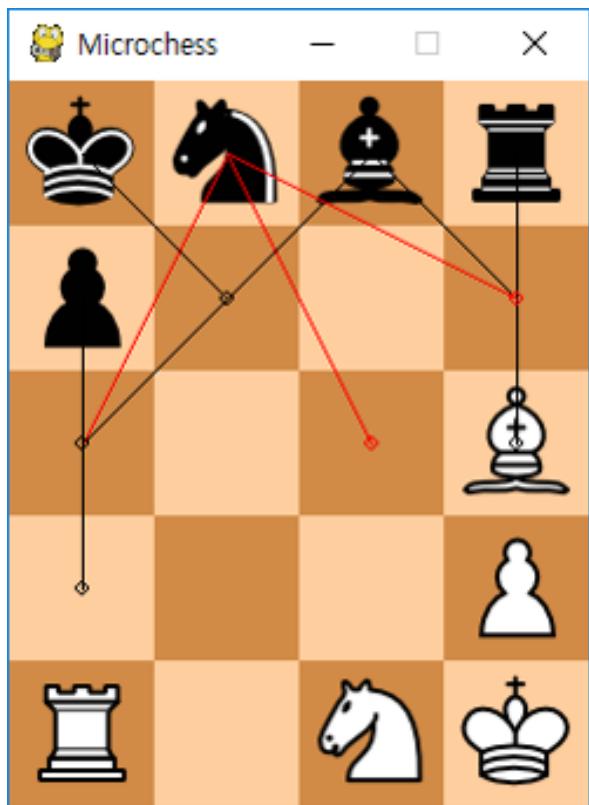


Fig. 1: 게임 플레이 인터페이스

CHAPTER 2

Tutorial

2.1 Microchess AI 경진대회

본 경진대회는 Game AI에 대한 학생들의 흥미를 유발하기 위해 기획되었다. 따라서, 학생들에게 주어진 각종 제약(시간, 컴퓨팅 자원)안에서 AI를 구현하고 실험하기에 너무 어렵지 않도록, 규모가 작을 필요가 있었다. 하지만 너무 쉽게 해결할 수 있도록 너무 단순하지는 않아야 했다.

*Microchess*는 일반 체스의 일부분 만을 사용하는 미니 체스 중 하나로써, 기존 체스에 비해 작은 탐색 공간을 가지고 있으면서도, 모든 경우의 수를 따져보는 것은 거의 불가능했기 때문에 경진대회 종목으로 선정되었다. 기존 체스는 대략 35^{80} 가지 경우의 수를 가지고 있는 것으로 알려져 있다. 반면 *Microchess*는 훨씬 작은 10^{20} 규모로 추정된다. 비록 문제의 규모는 매우 축소되었지만, 모든 경우의 수를 따지는 것은 천문학적인 규모이기 때문에, 다음 수를 결정하기 위해서는 보다 효율적인 방법이 필요하다.

*Microchess*는 일반 체스가 가지는 특징(완전 정보공개, 불확실성 없음)을 가지지만, 일부 다른 점이 있다.

- 4×5 미니체스
- 탐색공간이 매우 크지는 않지만, 쉽게 해결은 어려움
- 플레이어는 퀸을 제외한 기물을 하나씩 가지고 있음
- 기물의 움직임은 일반 체스와 동일
- Castling 가능

2.1.1 경진대회 규칙

이 경진대회에서는 대부분 일반 체스의 규칙을 따르지만, 게임 시간을 절약하고, 무승부¹를 줄이기 위해, AI 경진 대회에 적합하게 몇 가지 규칙을 변경하였다.

일반 체스는 무승부가 전체 게임의 30%가 넘을 정도로 무승부가 많은 게임이다. (Black의 승률보다 높음) *Microchess*에서 무승부 비율은 명확하게 조사된 바가 없지만, 무승부 확률이 동일하거나, 높은 것으로 추정된다. 일정 수준의 성능을 보이는 AI들끼리의 게임은 대부분 무승부로 끝나는 경향이 있다. 따라서, 이 경진대회에서는 보다 쉽게 승패를 가릴 수 있도록 무승부 조건을 강화하였다.

¹ 일반 체스는 무승부 비율이 높음 약 30~35 %

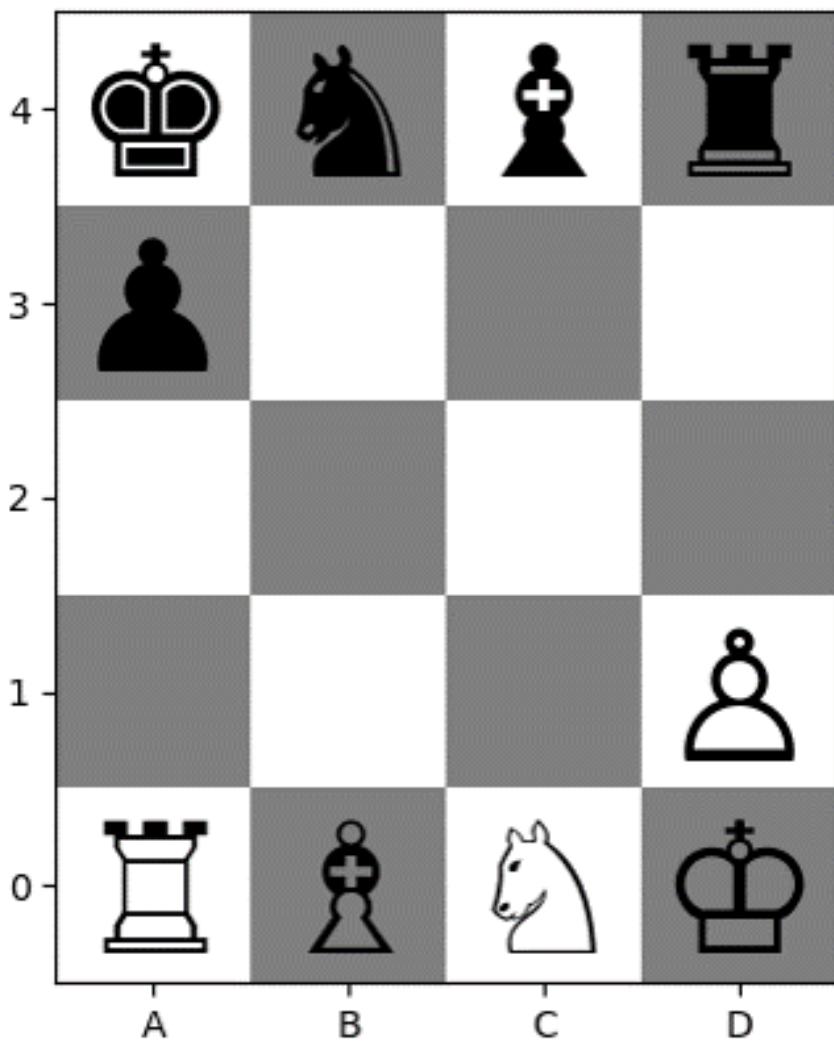


Fig. 1: Microchess

또한, 게임 플레이 데이터로 의사결정 모델을 학습할 때, 무승부가 되었을 때는 유용한 정보를 알아내기 어렵기 때문에, 무승부 비율이 줄어들면 기계학습 알고리즘을 사용하여 AI를 학습할 때도 도움이된다.

- **최대 80턴으로 제한**

- 대부분의 게임은 50턴 이내에서 종료됨
- AI들의 경우 무의미한 수를 반복하면서 지나치게 오랫동안 게임을 할 때도 있기 때문에 최대 턴에 제한을 둠

- **플레이어마다 전체 게임 400초 시간제한을 둠**

- 턴마다 시간제한은 없지만, 한 게임에서 400초를 모두 소모하면 패배
- 플레이어가 한 턴에 약 10초를 소모할 것이라고 가정해서 400초 시간제한을 둔 것
- **전체 400초를 초과하지 않는 범위내에서 플레이어가 임의로 조정가능함**
 - * 최대 80턴까지 게임하면 전체 게임 시간은 800초
 - * $400\text{초} = (80\text{턴} \times 10\text{초}) / 2\text{플레이어}$
 - * 한 턴에 10초 가정

- **승리/패배 조건**

- 기존 체스의 승리조건을 만족하면 승리(예. 상대 킹을 체크메이트)
- 최대 턴을 초과하면, 남은 기물에 따라 점수를 계산하여 높은 쪽이 승리함
- 기존 무승부 규칙²에 따라 무승부가 되면, 남은 기물의 점수를 계산하여 승패를 판단함
- 최대 턴 초과 또는 무승부로 게임 종료 후, 기물 점수도 동일한 경우에만 무승부로 판단함
- 한 플레이어가 먼저 400초 시간제한을 초과하면 패배로 판단함

2.1.2 점수계산

기물	점수
♔ (K) ♚ (k)	4
♕ (Q) ♛ (q)	10
♖ (R) ♜ (r)	5
♘ (N) ♞ (n) ♗ (B) ♖ (b)	3
♙ (P) ♖ (p)	1

Fig. 2: 점수 표

² 무승부 규칙: 기물이 부족해 체크메이트 불가, 어떤 기물도 이동 불가, 50턴 동안 폰을 이동하지 않거나, 기물이 잡히지 않음, 5번 이상 똑같은 수 반복, <https://en.wikipedia.org/wiki/Chess#Draw>

무승부로 게임이 종료되면, 점수 표³ 을 참조하여 남은 기물의 점수를 계산하여 게임의 승패를 판단한다.

다음은 점수 계산 예를 보여준다.

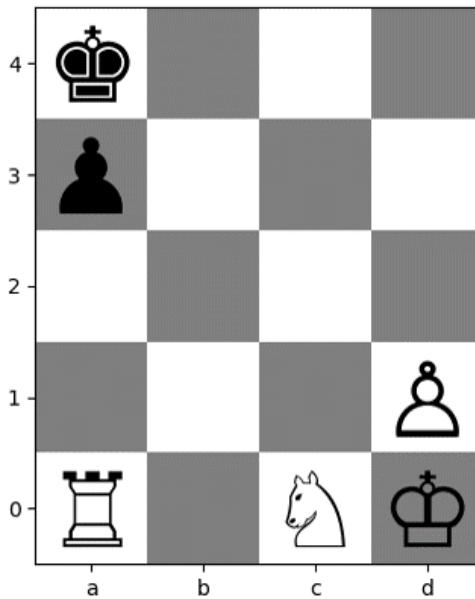


Fig. 3: 점수 계산 예

- 백: $(K)+(R)+(N)+(P)= 4+5+3+1 = 13$
- 黑: $(k)+(p)= 4+1 = 5$

2.1.3 주의사항

- 실격패 조건
 - 400초를 모두 모소하는 경우
 - 프로그램이 종료될 수준의 예외/에러 발생
 - 유효하지 않은 수를 둘 경우
 - 부정행위를 할 경우
 - * 파일 시스템에서 자신의 AI에게 허용되지 않은 경로에 접근하는 경우
 - * 네트워크에 연결하는 경우
 - * 상대 AI 혹은 경진대회 시스템에 영향을 주기 위해 별도의 thread, process 를 실행하는 경우

2.1.4 운영 게시판

- Google Groups를 이용
 - https://groups.google.com/forum/#!forum/microchess_ai
- 운영목적
 - 공지사항 전달

³ 사용하는 전략에 따라, 기물의 점수에 이견이 있지만, 많은 Chess AI들이 이 테이블의 점수를 따름

- 질의응답 게시판
- 과제제출용

2.1.5 최종 우승자 결정

Double Round-Robin Tournament (Full League)

- 모든 AI는 서로 흑과 백으로 여러 번⁴ 경기를 진행
- 가장 많은 점수를 획득한 AI가 우승
 - 점수: 승(+1), 패(0), 무승부(+1/2)

⁴ 정확한 경기횟수는 미정, 일반 체스는 두 번 이지만 AI 체스에서는 두 번만으로 평가가 어려울 수 있음

2.2 Microchess AI 플랫폼

2.2.1 개요

- Microchess 플랫폼 구현
 - 일반 체스 플랫폼인 python-chess¹ 을 수정하여 구현
- Python으로 AI를 작성할 수 있는 기반을 제공함
- 주요 AI 예제 네 가지 제공
 - Random AI, 탐색 기반 AI, MCTS AI, Self Learning AI
- 사람과 AI 체스 플레이 가능한 인터페이스 제공
- 개발 및 테스트 환경
 - python 3.5 (anaconda)
 - Windows 10

다른 python 버전이나, 배포판, 다른 OS에서 심각한 문제가 발생할 가능성은 적지만, 이 문서는 위 환경을 기준으로 작성되었다.

2.2.2 디렉토리 구조

```
MicrochessAICompetition
└── chess: Microchess 용으로 수정한 python-chess
└── scripts: 실행 스크립트 및 기타 파일
└── agents
    ├── basic: 기본 AI 예제
    ├── search: 탐색 기반 AI, MCTS 예제
    ├── self_learning: 학습 AI 예제
    └── stockfish: 다른 언어로 구현된 AI 사용 예제
```

2.2.3 설치방법

1. 플랫폼 다운로드 및 압축해제
2. anaconda 다운로드 및 설치 (<https://www.anaconda.com/download/>)
3. 가상환경 생성 및 활성화:

```
(base) C:\Users\user\MicrochessAICompetition> conda create -n mchess python=3.5
(base) C:\Users\user\MicrochessAICompetition> activate mchess
(mchess) C:\Users\user\MicrochessAICompetition>
```

4. 주요 모듈 설치:

```
(mchess) C:\Users\user\MicrochessAICompetition> conda install numpy scipy ipython_
˓→tqdm pyyaml mkl matplotlib
(mchess) C:\Users\user\MicrochessAICompetition> conda install -c CogSci pygame
(mchess) C:\Users\user\MicrochessAICompetition> pip install sqlitedict
(mchess) C:\Users\user\MicrochessAICompetition> pip install visdom # visdom 설치
(mchess) C:\Users\user\MicrochessAICompetition> conda install -c peterjc123_
˓→pytorch-cpu=0.3.0 # pytorch 설치 (CPU, Windows 용) (continues on next page)
```

¹ <https://github.com/niklasf/python-chess>

(이전 페이지에서 계속)

- visdom은 AI 디버깅 및 데이터 시각화 용으로 사용함
 - MCTS AI와 Self Learning AI에서 사용
- pytorch는 비공식 Windows + CPU 버전(<https://github.com/peterjc123/pytorch-scripts>) 사용
 - 최신 버전인 0.4.0 버전은 정식으로 Windows를 지원하지만, 개발에 사용한 0.3.1 버전은 정식으로 Windows를 지원되지 않음
 - 0.3.1 버전과 0.4.0 버전의 대부분은 호환되지만, 0.4.0을 이 플랫폼에서 검증하지 못했음
 - CUDA 지원버전을 사용할 수 있지만, CUDA 사용가능한 부분이 적어서 속도향상이 크지 않고, 학습 성능 검증이 부족함

2.2.4 실행 방법

예제 AI 실행

- Random AI vs. One Step Search AI

- Random AI 위치: agents/basic/random_agent.py
- One Step Search AI: agents/search;bruteforce_search_agent.py

```
(mchess) C:\Users\user\MicrochessAICompetition> python scripts/run_game.py --
→white=agents.basic.random_agent.RandomAgent --black=agents.basic.bruteforce_search_
→agent.OneStepSearchAgent
```

```
C:\WINDOWS\system32#cmd.exe
[2018-03-23 14:48:13,256 DEBUG] Agents.Basic.Greedy.OneStepGreedyAgent-False
[2018-03-23 14:48:13,257 DEBUG] Black board value 0.743
<IPython.core.display.SVG object>
[2018-03-23 14:48:13,694 DEBUG] move: g4e2
[2018-03-23 14:48:13,695 DEBUG] rn2kb1r/ppp1nppp/8/8/3q1P1P/6K1/q3b1B1/6NR w kq - 0 15
[2018-03-23 14:48:13,695 DEBUG] False move g4e2
[2018-03-23 14:48:13,695 DEBUG] Agents.Basic.Random.RandomAgent-True
[2018-03-23 14:48:13,697 DEBUG] White board value 0.246
<IPython.core.display.SVG object>
[2018-03-23 14:48:13,712 DEBUG] move: h4h5
[2018-03-23 14:48:13,713 DEBUG] rn2kb1r/ppp1nppp/8/7P/3q1P2/6K1/q3b1B1/6NR b kq - 0 15
[2018-03-23 14:48:13,715 DEBUG] True move h4h5
[2018-03-23 14:48:13,715 DEBUG] Agents.Basic.Greedy.OneStepGreedyAgent-False
[2018-03-23 14:48:13,716 DEBUG] Black board value 0.754
<IPython.core.display.SVG object>
[2018-03-23 14:48:14,179 DEBUG] move: e2h5
[2018-03-23 14:48:14,180 DEBUG] rn2kb1r/ppp1nppp/8/7b/3q1P2/6K1/q5B1/6NR w kq - 0 16
[2018-03-23 14:48:14,181 DEBUG] False move e2h5
<IPython.core.display.SVG object>
[2018-03-23 14:48:14,197 DEBUG] Score: [0.235 0.765]
[2018-03-23 14:48:14,198 DEBUG] Win or Lose: [0.5 0.5]

(drl) D:\desktop\MicroChessCompetition>
```

Fig. 4: fen² 표기법으로 보드 상태 출력

- Jupyter qtconsole에서 실행

```
(mchess) C:\Users\user\MicrochessAICompetition> jupyter qtconsole
In [1]: %run scripts/run_game.py --white=agents.basic.random_agent.RandomAgent --
→black=agents.search;bruteforce_search_agent.OneStepSearchAgent
```

² https://en.wikipedia.org/wiki/Forsyth-Edwards_Notation

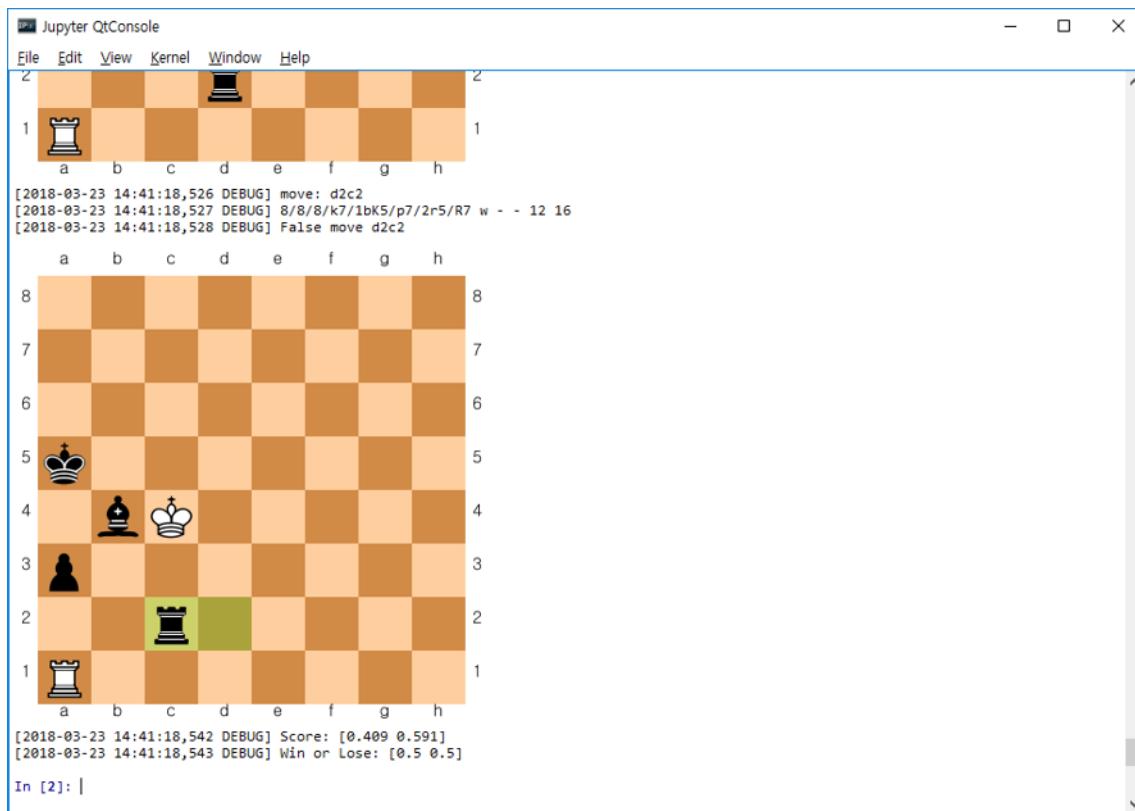


Fig. 5: Qt Console 화면

Random AI vs. Random AI

AI끼리 게임을 플레이할 때는 scripts/run_game.py 을 사용한다. 두 Random AI끼리 플레이할 때는 다음과 같이 실행한다.:

```
python scripts/run_game.py --white=agents.basic.random_agent.RandomAgent --
→black=agents.basic.random_agent.RandomAgent
```

White와 black에 AI의 경로를 지정해 주면 게임을 실행하고, 결과를 출력해 준다.

예제 AI는 AI의 경로를 run_game.py의 agents에 미리 저장해 두었기 때문에, 전체 경로를 지정할 필요없이 다음과 같이 짧은 이름으로 실행 가능하다.:

```
python scripts/run_game.py --white=random --black=random
```

Listing 1: 예제 AI의 이름과 경로

```
agents = dict(
    human='agents.basic.human.Player',
    malfunction='agents.basic.debug_agent.MalfunctionAgent',
    first_move='agents.basic.debug_agent.FirstMoveAgent',
    random='agents.basic.random_agent.RandomAgent',
    one_step_search='agents.search.one_step_search_agent.OneStepSearchAgent',
    two_step_search='agents.search.two_step_search_agent.TwoStepSearchAgent',
    greedy='agents.search.two_step_search_agent.TwoStepSearchAgent',
    negamax='agents.search.negamax_search_agent.NegamaxSearchAgent',
    abp_negamax='agents.search.abp_negamax_search_agent.ABPNegamaxSearchAgent',
    mcts='agents.search.mcts_agent.MCTSAgent',
    mcts_dev='agents.search.mcts_agent.MCTSAgentDev',
    self_learning='agents.self_learning.agent.Agent',
    stockfish='agents.stockfish.agent.Stockfish',
)
```

Random AI와 One Step Search AI의 게임은 다음과 같이 실행 가능하다.:

```
python scripts/run_game.py --white=random --black=one_step_search
```

One Step AI는 다음 수의 결과를 고려하는 간단한 AI이다. 강력하지는 않지만, Random AI는 쉽게 이길 수 있다. 실행한 결과는 다음과 같이 나타난다.

```
1 [2018-04-23 15:40:07,079 DEBUG] White: agents.basic.random_agent.RandomAgent-True
2 [2018-04-23 15:40:07,079 DEBUG] White: board value 0.500
3 <IPython.core.display.SVG object>
4 k n b r
5 p . . .
6 . . . .
7 . . . p
8 R B N K
9 [2018-04-23 15:40:07,093 DEBUG] White: 400.0 sec. remain
10 [2018-04-23 15:40:07,093 DEBUG] White: move b1d3
11 [2018-04-23 15:40:07,093 DEBUG] 8/8/8/knbr4/p7/3B4/3P4/R1NK4 b Kk - 1 1
12 ... (생략)
13 [2018-04-23 15:30:05,171 DEBUG] White: move d2c1
14 [2018-04-23 15:30:05,172 DEBUG] 8/8/8/8/3k4/8/8/2K5 b - - 0 37
15 [2018-04-23 15:30:05,172 DEBUG] [0.5, 0.5]
16 [2018-04-23 15:30:05,172 DEBUG] draw
17 <IPython.core.display.SVG object>
18 . . .
```

(continues on next page)

(이전 페이지에서 계속)

```

19 . . . k
20 . . .
21 . . .
22 . . K .
23 [2018-04-23 15:30:05,178 DEBUG] Score: [0.500 0.500]
24 [2018-04-23 15:30:05,178 DEBUG] turns: 72
25 [2018-04-23 15:30:05,178 DEBUG] Game end: True
26 [2018-04-23 15:30:05,179 DEBUG] checkmate: False
27 [2018-04-23 15:30:05,179 DEBUG] Stalemate: False
28 [2018-04-23 15:30:05,179 DEBUG] Insufficient material: True
29 [2018-04-23 15:30:05,179 DEBUG] 57 moves: False
30 [2018-04-23 15:30:05,179 DEBUG] 5-fold: False
31 [2018-04-23 15:30:05,179 DEBUG] Draw: [0.5, 0.5]

```

4~8번째 줄처럼 매 턴마다, 게임 상태를 텍스트 상태로 출력한다. white는 대문자로 표시하고, black은 소문자로 표시한다.

23번째 줄부터 게임 결과를 출력한 것이다. Score의 첫번째 숫자는 white의 점수를 나타내고, 두 번째 숫자는 black의 점수를 나타낸다. (특별한 경우가 아니면 플랫폼의 나머지 부분에서도 white를 black보다 먼저 출력함) 게임에 승리하면 1.0, 패배하면 0.0, 그리고 비기면 0.5점을 얻는다.

일반 Chess (Microchess 포함)와 달리 이 플랫폼에서는 승패가 결정되지 않으면, 게임이 종료되었을 때 남아있는 기물의 점수를 계산하여 최종 승패를 판단한다. 따라서, 기존 무승부 조건을 만족한 상태에서 기물의 점수도 동일한 경우만 무승부가 가능하다.

turns는 전체 게임의 턴 수를 보여주고, 그 다음 줄의 game end는 게임이 정상적으로 종료되었는지 여부를 알려준다. 그 다음부터는 게임이 종료된 이유(checkmate, stalemate, 등)를 알려준다.

AI 끼리의 성능을 평가할 때는 benchmark 옵션을 사용한다. benchmark 옵션을 사용하면, 지정된 white와 black 옵션과 상관없이 총 20게임 (지정된 white와 black으로 10게임, white와 black을 뒤집어서 10게임)을 플레이하고 전체 승률을 출력해준다.

Random AI vs. One Step Search AI

Random AI와 One Step Search AI의 성능을 비교하려면 다음과 같이 실행한다.:

```
python scripts/run_game.py --white=random --black=one_step_search --benchmark
```

실행 결과는 다음과 같이 나타난다.

	random		one_step_search
[2018-04-23 15:52:19,247 INFO]			
[2018-04-23 15:52:19,247 INFO]	W	0.000 B	1.000
[2018-04-23 15:52:19,247 INFO]	W	0.500 B	0.500
... (생략)			
[2018-04-23 15:52:19,248 INFO]	B	1.000 W	0.000
[2018-04-23 15:52:19,248 INFO]	B	0.000 W	1.000
[2018-04-23 15:52:19,248 INFO]	random (White): 0.300 vs. one_step_search (Black): 0. ↳ 700		
[2018-04-23 15:52:19,249 INFO]	random (Black): 0.350 vs. one_step_search (White): 0. ↳ 650		
[2018-04-23 15:52:19,250 INFO]	random: 0.325 vs. one_step_search: 0.675		

7번째 줄 까지는 두 AI가 얻은 점수를 보여주고, 마지막 세 줄은 그 평균을 보여준다. 두 AI가 각각 White/Black 그리고 Black/White인 경우의 평균을 따로 보여주고, 마지막으로 둘을 통합한 평균을 보여준다. 이 결과에서는 One Step Search AI가 평균 0.675점을 얻어 더 좋은 성능을 보여주고 있다.

2.2.5 인간 플레이어 vs. AI 인터페이스

AI 성능 테스트 용으로 게임 플레이 인터페이스 제공한다. AI의 경로를 입력하는 부분에 `human`이라고 쓰면, AI와 동일한 방식으로 실행할 수 있다. 인간 플레이어로 게임을 실행하면, [인간 플레이어 인터페이스](#) 가 나타나고, 플레이어의 순서에 마우스로 조작이 가능해진다.

```
(mchess) C:\Users\user\MicrochessAICompetition> python scripts/run_game.py --  
--white=agents.basic.random_agent.RandomAgent --black=agents.basic.human.Player
```

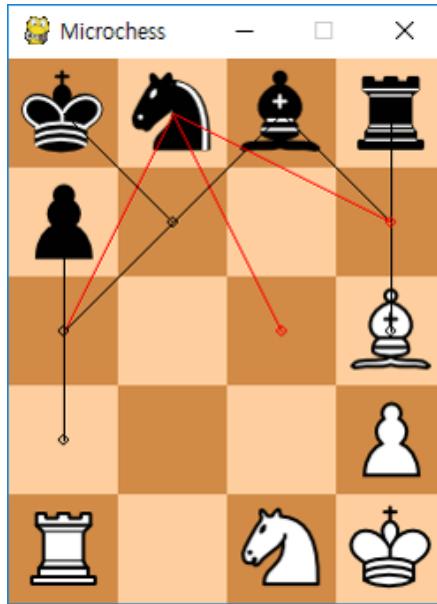


Fig. 6: 인간 플레이어 인터페이스

2.3 기본 AI 예제

2.3.1 Random AI

Listing 2: Random AI

```

1 import random
2 from agents import BaseAgent
3
4
5 class RandomAgent(BaseAgent):
6     """
7         Random AI
8
9         - 무작위로 행동을 결정하는 예제
10    """
11
12     def reset(self):
13         pass
14
15     def act(self, state):
16         moves = list(state.legal_moves)
17         return random.choice(moves)
18
19     def close(self):

```

2.3.1.1 가장 간단한 AI 구현

*Random AI*는 이 플랫폼을 이용해 구현된 가장 간단한 AI 중 하나이다. 이 AI의 목적은 플랫폼 사용자들에게 AI를 구현하는 가장 기초적인 방법에 대해 설명하고, AI 성능의 최저 기준선을 제시하는 것이다. 새로 구현한 AI가 Random AI에 대비하여 명확하게 좋은 성능을 보이지 않는다면, 새로 구현한 AI 제대로 작동하고 있지 않을 가능성이 매우 높다.

AI를 구현할 때는 Random AI처럼 *BaseAgent*를 상속받아 *reset*, *act*, *close* 메소드를 구현하면 된다. 반드시 필수는 아니지만, 모든 예제 AI는 "agents/{package}/{agent_name}.py"에 저장하고 있다. package와 agent_name은 AI의 특징을 나타낼 수 있도록 임의로 정하여 사용한다. 게임 시작할 때 AI 클래스의 위치를 "agents.{package}.{agent_name}.{agent_class_name}" 형식으로 전달하면, 플랫폼이 AI 클래스를 import하여 실행한다.

reset() 메소드는 새로운 게임이 시작하기 전에 한번 호출된다. 보통 이 메소드에서는 AI 초기화 작업을 진행한다. 예를 들면 학습한 모델이나 데이터를 메모리로 읽어들이는 작업을 이때 할 수 있다.

close() 메소드는 반대로 게임이 종료되었을 때, 한번 호출된다. 여기서는 아직 열려있는 파일을 닫거나, 메모리에 아직 남아있는 데이터를 저장하는 등의 작업을 할 수 있다.

*reset()*과 *close()* 메소드는 필요하지 않다면, 반드시 구현할 필요는 없다. 구현하지 않으면 *BaseAgent*에 있는 아무 것도 하지 않는 *reest()*과 *close()*가 사용된다.

하지만, *act(state)* 메소드는 반드시 구현해야 한다. 이 메소드는 해당 AI 턴마다 실행된다. 인자로 전달되는 *state*는 현재 게임 상태정보를 가지고 있다. AI는 이 정보를 이용해서 다음에 둘 수를 반환해야한다. AI를 구현한다는 것은 기본적으로 이 부분에 특정 게임상태(*state*)와 최선의 수(*move*)를 매핑해주는 함수를 정의하는 일이다.

state 인스턴스는 *State* 객체이다. *State* 객체는 *python-chess*의 *Board* 객체를 상속받아 시뮬레이션 기능을 지원하는 *forward(move)* 메소드를 추가한 것이기 때문에, 기본적으로 *chess.Board* 객체에서 지원하는 기능을 모두 가지고 있다. 향후 부정행위에 이용할 만한 기능을 발견되면 일부 기능이 제한될 수 있다.

Board 객체에서 지원하는 기능 중에 하나는 주어진 게임상태에서 둘 수 있는 수를 알려주는 것이다. 15번째 줄의 `state.legal_moves`는 현재 둘 수 있는 수들을 반환하는 python generator이다. Random AI는 이것을 list (`moves`)에 담은 뒤, 그 중에서 무작위로 하나를 선택(`random.choice`)하여 반환한다.

2.3.1.2 `act` 메소드의 입력과 출력 요약

`act(state)` 메소드의 입력과 출력을 핵심을 정리하면 다음과 같다.

- **입력: state**

- python-chess의 Board 객체 + 다음 상태 시뮬레이션(forward)
- 현재 보드 상태에 대한 정보

- **출력: move**

- python-chess의 Move 객체
- 기물의 현재 위치, 다음 위치 저장
- [0, 20] 정수 x [0, 20] 정수 = 400 가지 조합 가능함
- 특정 상태에서 선택가능한 수는 평균 10개 정도

2.3.2 Search 기본 AI

AI가 하는 가장 기본적인 작업은 특정 상태(s ; state)가 주어졌을 때, 최선의 행동(a ; action)을 결정하는 일이다. 즉 우리는 AI를 상태와 행동을 매핑하는 일종의 함수라고 볼 수 있다($f(s) \rightarrow a$). 일반적으로는 행동(a ; action)으로 표현하지만, 체스같은 보드게임에서는 수(m ; move)라고 표현하므로, 이 문서에는 필요에 따라 행동(a)과 수(m)를 혼용해서 한다.

현재 적합한 행동을 결정하는 가장 간단한 방법은 실제 그 행동을 했을 때, 어떤 결과가 나타날지 시뮬레이션을 해보고, 가장 좋은 결과가 나타나는 행동으로 다음 행동을 결정하는 것이다. 시뮬레이션은 현재 상태(s_t)와 행동(a)이 주어졌을 때, 다음 상태(s_{t+1})를 반환해주는 함수 ($f(s_t, a) \rightarrow s_{t+1}$)로 볼 수 있는데, 이 플랫폼에서는 `State.forward(move)` 함수가 그 역할을 한다. `forward` 함수는 게임으로직과 동일하게 작동하고, 상대방의 미래 행동을 제외하면 게임에는 불확실성이 없기 때문에 상대방의 미래 행동을 정확하게 예측할 수 있다면 정확한 미래 상태를 예측할 수 있다.

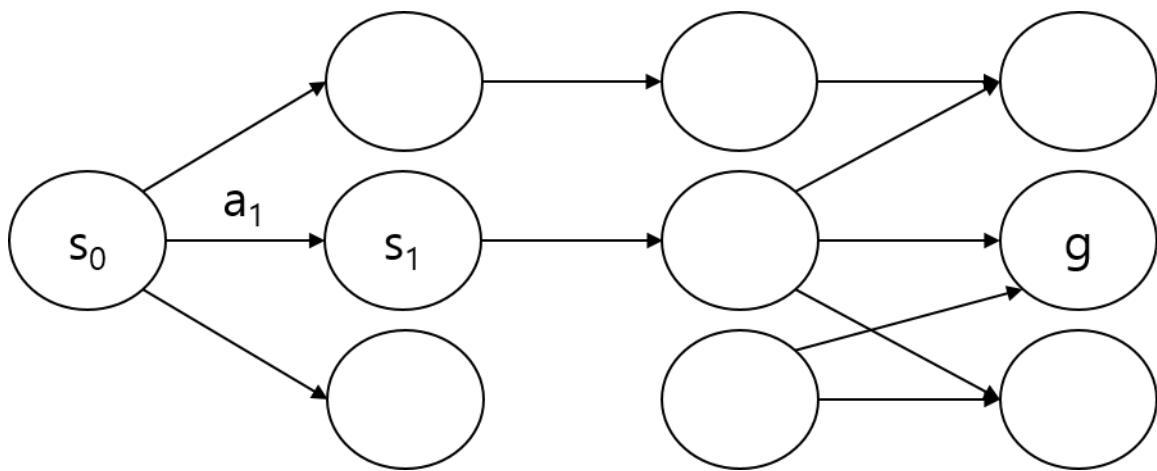


Fig. 7: 상태공간

Simulation 함수가 정의되고 나면, 그래프 구조로 만들어진 [상태공간](#)을 만들 수 있다. 각각의 노드는 특정 상태(예. 특정한 체스기물의 배치상태)를 나타내고, 노드와 노드를 연결하는 링크는 행동을 의미한다. AI의 궁극적인

목적은 상태공간의 시작 상태(s_0)에서 시작하여, 목표상태(g , 게임에 승리한 상태)까지 도달하기 위해 현재 어떤 행동을 해야하는지 알아내는 것이다. 이것은 그래프 구조에서 경로를 탐색하는 문제로 취급할 수 있기 때문에 순회(traverse), 탐색(search) 알고리즘을 사용해 문제를 해결한다. 따라서 이런 방식의 AI를 탐색(Search)기반 AI라고 한다.

실제 상태공간은 순환(cycle)이 존재하는 그래프지만 문제를 단순화하기 위해 일반적으로는 순환을 없애고(실제 동일한 상태를 새로운 상태로 취급) 상태공간을 트리 구조로 모델링한다.

2.3.2.1 One Step Search AI

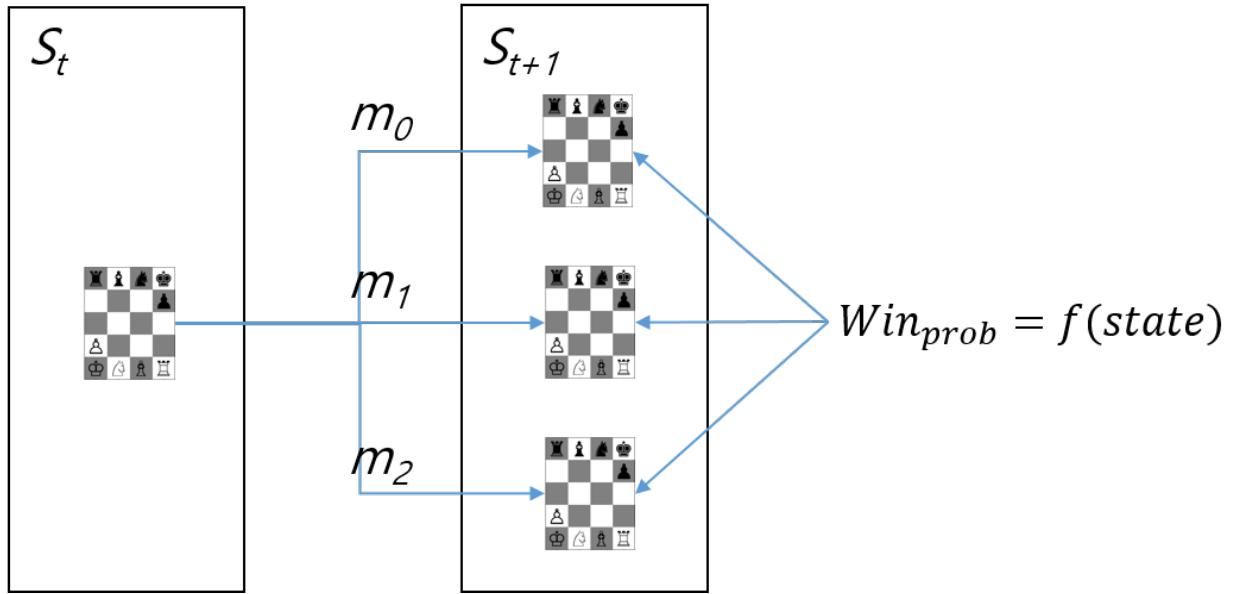


Fig. 8: One Step Search AI

One Step Search AI 는 바로 다음 수의 결과만을 예측하여 현재 행동을 결정하는 가장 단순한 Search 기반 AI이다. 현재 상태 S_t 에서 가능한 수를 모두 시뮬레이션 하여, 다음 상태 S_{t+1} 들을 알아낸 뒤, 목표상태와 가장 가까운 상태를 찾아, 그와 연관된 수를 다음 수로 결정한다.

여기서 목표상태와 얼마나 가까운지 판단하기 위해 평가함수를 사용한다. 평가함수는 현재 상태(s)를 입력받아, 이것이 목표상태(g)에 얼마나 가까운지 정량적으로 측정하는 함수이다($eval(s) \rightarrow v$).

$$v = \frac{score_{my}}{score_{my} + score_{opponent}} \quad (2.1)$$

One Step Search AI가 사용하는 평가함수 (2.1)는 얼마나 승리에 가까운지 평가하여 [0, 1] 실수로 반환한다. 자신과 상대방의 점수($score_{my}$, $score_{opponent}$)는 점수 표의 기률점수를 사용한다. 패배에 가까울 수록 0에 가깝고, 승리에 가까울 수록 1에 가까운 값을 반환한다.

가장 간단한 탐색 방법을 사용하는 One Step Search AI는 Random AI 보다는 좋은 성적을 보이지만, 한 수 이상을 고려하지 않기 때문에, 여러가지 문제를 가지고 있다. 상대방의 대응을 예상하지 못하기 때문에 쉽게 반격당할 수 있고, 다음 상태에서, 다른 상태에 비해 더 좋은 상태가 없는 경우 의사결정이 어렵다. 이 문제의 원인은 One Step Search AI가 매우 단기적인 이익 만을 추구하기 때문에, 장기적인 계획(연속된 행동)이 필요한 목표를 추구할 수 없기 때문이다.

따라서 이 문제의 근본적인 해결 방법은 더 먼 미래까지 탐색하는 것이 된다.

2.3.2.2 Two Step Search AI

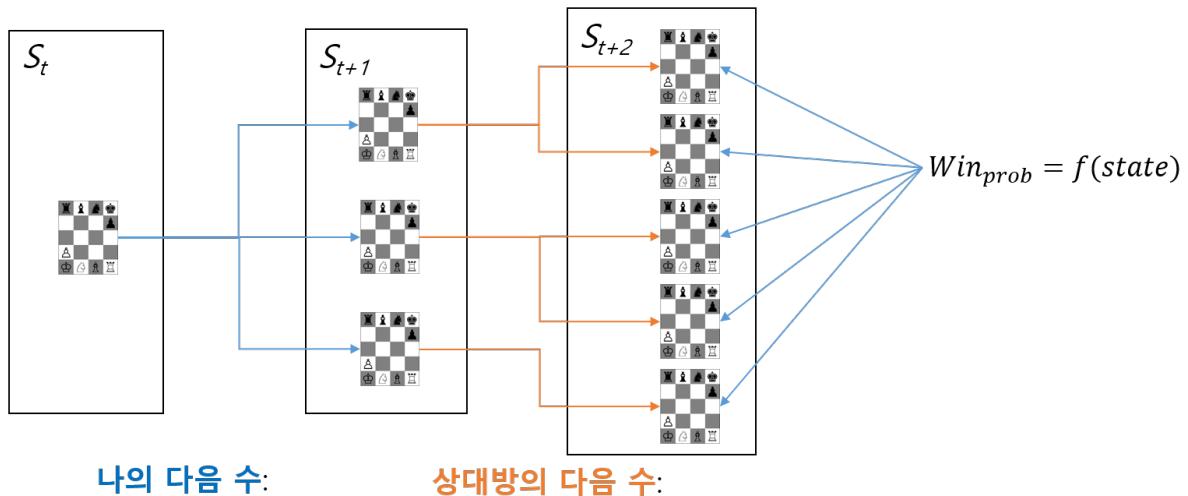


Fig. 9: Two Step Search AI

*Two Step Search AI*는 이 방향에서 One Step Search AI를 개선한 것이다. *Two Step Search AI*는 이름 그대로 두 단계 미래 상태 (S_{t+2})까지 탐색하고 가장 좋은 상태로 연결되는 수를 찾아내는 AI이다.

*Two Step Search*에서 두 번째 수는 내가 아닌 상대방의 수이기 때문에, 미래 상태를 예측 하려면 다음에 상대방이 어떤 수를 둘 것인지를 알아야 한다. 실제로 상대방이 다음에 어떤 수를 둘지는 알 수 없지만 상대방도 나와 마찬가지로 현재 상태를 목표상태에 가깝게 바꿔나가고자 한다고 가정하면, 상대방의 수를 예측할 수 있다.

Chess 같은 zero-sum 게임¹에서는 상태의 평가 점수가 낮을 수록 상대방의 입장에서는 유리한 것이기 때문에, 내가 더 좋은 상태로 연결되는 수를 선택해야하는 것과는 반대로 상대방은 (나에게) 더 나쁜 상태로 연결되는 수를 선택할 것이다.

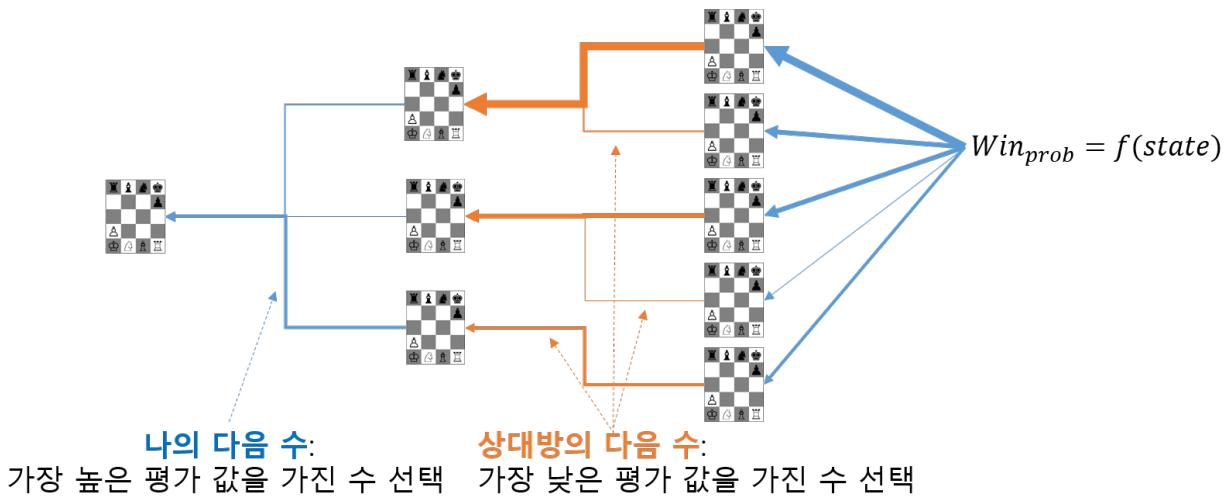


Fig. 10: Two Step Search AI의 평가 과정

*Two Step Search AI의 평가 과정*은 그림처럼 마지막 상태들을 평가한 뒤에, 꺼꾸로 가장 먼 미래로부터 현재 상태까지 어떤 행동들을 선택해야 가장 높은 평가를 받은 마지막 상태에 도달할 수 있는지 알아낸다. 차례대로

¹ 한쪽의 승리가 다른쪽에서는 패배가 되는 게임

가장 낮은 평가값을 받은 상태로 연결되는 수(상대순서), 가장 높은 평가를 받은 상태로 연결되는 수(내 순서)를 선택하여 현재 상태에서 두어야 할 수를 결정한다.

2.3.2.3 Two Step Search AI 구현

Listing 3: Two Step Search AI (Greedy AI)

```

1  def act(self, state):
2      my_action_evals = list()
3      for my_action in state.legal_moves:
4          next_state = state.forward(my_action)
5          if next_state.is_game_over():
6              # 한 수 뒤에 게임이 종료될 경우 상대방의 수를 보지 않고 바로 평가
7              # add_nodise: 평가값이 동일한 경우 그 중에 하나를
8              # 무작위로 선택하도록 하기 위해 작은 노이즈 추가
9              score = Evaluator.eval(next_state, self.turn) + 1e-6 * random.random()
10         else:
11             score = self.opponent_act(next_state)
12             my_action_evals.append((my_action, score))
13
14     # 가장 평가값이 높은 수를 반환함
15     best_action, score = max(my_action_evals, key=itemgetter(1))
16     return best_action
17
18 def opponent_act(self, state):
19     """
20         상대방의 행동 시뮬레이션
21         이 상태의 평가는 상대방의 그 다음 행동에 따라 달라짐
22
23         :param State state: 현재 상태
24         :return: (float) -- 평가값 [0, 1] 범위
25     """
26
27     opp_action_evals = list()
28     for opp_action in state.legal_moves:
29         next_state = state.forward(opp_action)
30         score = Evaluator.eval(next_state, self.turn) + 1e-6 * random.random()
31         opp_action_evals.append((opp_action, score))
32     _, score = min(opp_action_evals, key=itemgetter(1))
33     return score

```

*Two Step Search AI (Greedy AI)*은 Two Step AI의 핵심인 `act` 와 `opponent_act` 메소드이다. `act` 메소드는 s_t 를 입력받아, 다음 상태(s_{t+1})를 평가할 때, 상대방이 대응하고 난 뒤의 상태(s_{t+2})를 먼저 평가한 뒤에 그 결과를 이용한다.

매우 많은 약점을 노출하는 One Step Search AI와 달리, Two Step Search AI는 쉽게 반격당하는 무모한 공격을 하지 않기 때문에 상당히 그럴듯하게 작동한다. 하지만, 실수하지 않고 주의깊게 공략하면 쉽게 이길 수 있다.

기본적으로 Two Step Search AI는 One Step Search AI와 동일한 한계를 가진다. Two Step Search AI도 두 수 이상의 미래는 고려하지 않기 때문에, 두 수 이내에서 현재보다 좋은 상황이 없을 경우 다음 수를 결정하기 어렵고, 상대방이 두 수 이상을 고려한 전술을 사용할 경우 효과적으로 대처가 불가능하다.

이것을 해결하는 직접적인 방법은 보다 깊게(먼 미래) 상태공간을 탐색하는 것이다. 두 수만이 아니라, 더 많은 미래의 경우의 수를 계산하면 쉽게 성능을 높일 수 있다. Microchess는 모든 정보가 공개된, 불확실성이 없는 게임이기 때문에, 상대방의 행동만 예상할 수 있다면 게임 시작부터 끝날 때 까지 모든 상태를 탐색할 수는 있다.

Two Step Search AI과 같은 방식의 AI를 보다 일반적으로 구현한 것이 Negamax Search이다. Negamax Search는 Two Step Search AI와 달리 임의의 미래까지 탐색이 가능하다.

2.3.2.4 Negamax Search AI

Listing 4: Negamax Search 함수

```

1  def negamax(self, state, depth, color):
2      """
3          Negamax 탐색
4
5          :param State state: 현재 상태
6          :param int depth: 남은 탐색 깊이, self.max_depth - 현재 깊이
7          :param float color: white = 1., black = -1
8          :return: (chess.Move, float) -- best_move, 평가값
9
10         - chess.Move: 가장 좋은 행동
11         - float: 가장 좋은 행동을 했을 때 얻을 수 있는 미래 보상
12     """
13
14     if depth == 0 or state.is_game_over():
15         # 입력 받은 상태가 최대 탐색 깊이거나, 게임이 종료된 상태라면, 상태를 평가함
16         heuristic_value = Evaluator.eval_2(state, True) + 1e-6 * random.random()
17         # white는 1.0, black은 -1.0이 가장 좋은 점수
18         return None, color * heuristic_value
19
20     # 최저값으로 기본값 결정
21     best_move, best_value = None, -1000
22     for move in state.legal_moves:
23         next_state = state.forward(move)
24         # 재귀적으로 미래상태를 시뮬레이션함
25         # 다음 상태는 상대방 순서이므로, color를 바꾸고,
26         # 반환받은 평가값(value)도 부호를 변경함
27         _, value = self.negamax(next_state, depth-1, -color)
28         value = -value
29         if value > best_value:
30             best_move, best_value = move, value
31
32     self.n_nodes += 1
33     return best_move, best_value

```

Negamax Search² 는 Minimax Search의 일종으로, 구현을 단순하게 하기 위해, 평가값 구간을 [0, 1] 사이 실수로 하는 대신, [-1, 1] 사이 실수를 사용한다. 한 단계 미래상태를 탐색할 때마다, 보상값에 -1을 곱하여 상대방의 입장에서도 최대값(-를 곱했으므로 나에게는 최소값)을 찾도록 구현하였다. 재귀적으로 Negamax Search 함수를 호출하면서, 깊이 우선탐색(depth-first search)로 정해진 최대 깊이(미래 상태)까지 탐색하고, 그 중에 가장 좋은 상태로 연결되는 현재 수와 평가값을 반환한다.

여기서 가장 중요한 파라미터는 최대 탐색깊이(depth 인자)이다. Microchess를 포함한 대부분의 문제에서 하나의 행동으로 목표를 달성하는 경우는 드물고, 보통 연속된 행동의 결과로 목표 상태에 도달할 수 있다. 따라서 시작상태부터 목표상태까지 경로를 찾아내기 위해서는 가능한 먼 미래 상태까지 탐색할 필요가 있다. 만약 탐색깊이가 부족하면, 모든 상태를 탐색했음에도 불구하고, 목표상태를 발견하지 못할 수 있다. 그러나, 탐색 깊이를 증가시키면 탐색공간이 지수적으로 증가하기 때문에 매우 많은 계산비용이 필요할 수 있다. 따라서, 목표까지의 거리와 탐색 비용은 상충관계(trade-off)에 해당하고, 이것을 고려해서 최대 탐색 깊이를 신중히 결정해야 한다. 탐색깊이가 부족해서 목표상태를 발견하지 못하는 현상을 수평선 효과(horizon-effect³)라고 하며, AI 분야에서 대표적인 문제 중의 하나이다.

이 플랫폼에서는 기물점수를 계산해서 승패에 중요한 요소로 삼기 때문에, 수평선 효과가 상대적으로 덜 발생할 수 있다. 기존 체스(Microchess 포함)에서는 게임 종료상태까지 탐색하지 못하면(약 80수), 현재 수의 좋고 나쁨을

² <https://en.wikipedia.org/wiki/Negamax>

³ https://en.wikipedia.org/wiki/Horizon_effect

판단할 수 없기 때문에 충분히 먼 미래를 탐색하지 않으면 수평선효과가 강하게 나타날 수 밖에 없다. 하지만 이 플랫폼의 기본 규칙에서는 내 기물을 지키고 상대방의 기물을 제거하기만 하면(5수 이하) 더 좋은 상태로 판단할 수 있기 때문에 얇은(가까운 미래) 탐색만으로도 상태의 좋고 나쁨을 쉽게 판단할 수 있다.

2.3.2.5 Negamax Search AI + $\alpha - \beta$ pruning

$\alpha - \beta$ pruning⁴은 Negamax 알고리즘의 탐색공간을 줄여주는 대표적인 기법이다. 기본 아이디어는 탐색하는 도중에 상대방 순서에서 찾아낸 최소 평가값(β)이 그 이전의 내 순서에서 찾아낸 최대 평가값(α) 보다 작거나 같으면 ($\beta \leq \alpha$), 나머지 수를 탐색하지 않고, 현재 평가값을 최종 평가값으로 반환하는 것이다. 이 경우 나머지 수에서 어떤 작은 평가값이 나와도, 그 위 단계에서 나는 그것을 선택하지 않고 α 을 선택할 것이기 때문에 더 이상 탐색할 필요가 없다.

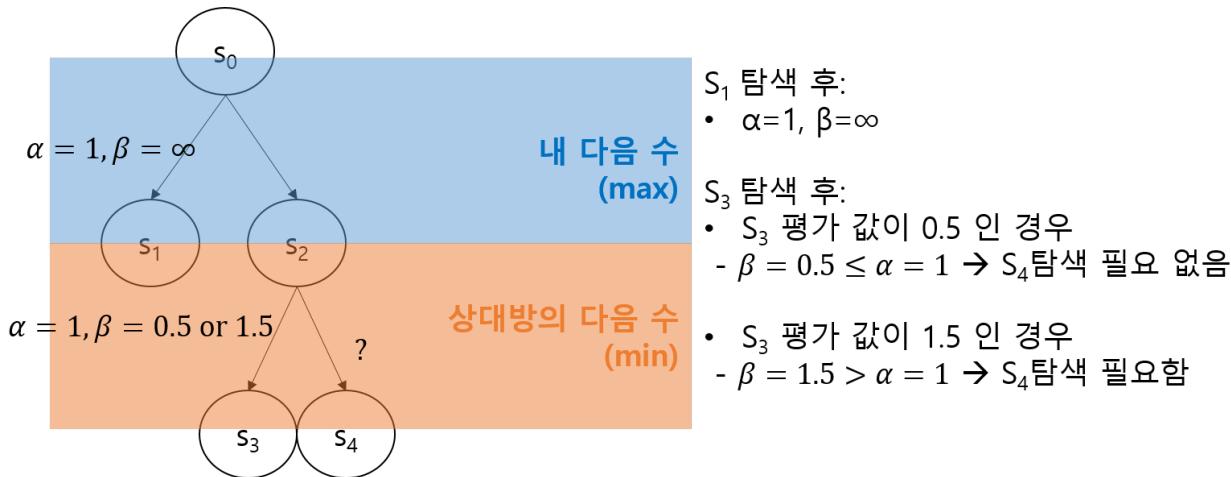


Fig. 11: $\alpha - \beta$ pruning 예

간단한 $\alpha - \beta$ pruning 예⁵를 보면 S_1 에서 평가값 1을 얻었을 때, 다음 평가 대상인 S_3 의 평가값이 1보다 낮거나 높거나 두 가지 가능성 있다. 만약 이 값이 1보다 작은 0.5가 나오다면 S_4 는 탐색이 필요없다. S_4 에서 0.5보다 큰 값이 나온다면 상대방이 이것을 선택하지 않을 것이고, 0.5보다 작은 값이 나온다면 상대방은 선택하겠지만 내가 그것을 선택하지 않을 것이기 때문이다.

$\alpha - \beta$ pruning을 사용한 기본적인 Negamax Search는 ABPNegamaxSearchAgent에 구현되어 있다. 기존 Negamax Search AI는 탐색 깊이 4이상부터 급격히 탐색시간이 증가하여 경진대회에서 가정한 10초를 거의 다 사용한다. 반면 ABP Negamax Search AI는 탐색깊이 6까지는 어렵지 않게 탐색할 수 있다. 계산비용을 절약한 만큼 더 많은 공간을 탐색할 수 있기 때문에, 결과적으로 성능을 향상시킬 수 있다. 이 알고리즘은 일반적인 체스 AI를 구현하는 가장 기본적인 접근방법이다⁶. 많은 유명한 AI들도 기본적으로 이런 접근방법을 사용한다.

2.3.2.6 Microchess 상태공간 규모

Microchess 처럼 간단한 문제에서도 모든 탐색공간을 탐색하는 것은 현실적으로 매우 어렵다고 볼 수 있다. 한 상태에서 선택 가능한 수가 평균적으로 10개이고, 게임 길이가 평균 20턴이라고 가정하면, b (branch) 가 10이고, d (depth)가 20인 상태공간 트리를 순회해야 하고 가장 좋은 상태를 찾아야 하는데, 총 순회해야 할 노드의 개수는 약 $b^d = 10^{20} = 100,000,000,000,000,000,000$ 개⁶ 정도 된다. 매 초마다, 백만개씩 순회 가능하다고 하더라도, 한 수를 두는데 100,000,000,000초가 소모된다. 비록 $\alpha - \beta$ pruning 같은 기법을 사용하여 불필요한 탐색공간을 줄이더라도, 문제는 크게 완화되지 않는다. 만약 전체 탐색공간의 99%를 제외할 수 있다고 하더라도, 전체 탐색공간에서 0이 두 개 줄어들 뿐이다.

⁴ https://en.wikipedia.org/wiki/Alpha–beta_pruning

⁵ <https://medium.freecodecamp.org/simple-chess-ai-step-by-step-1d55a9266977>

⁶ https://en.wikipedia.org/wiki/Endgame_tablebase

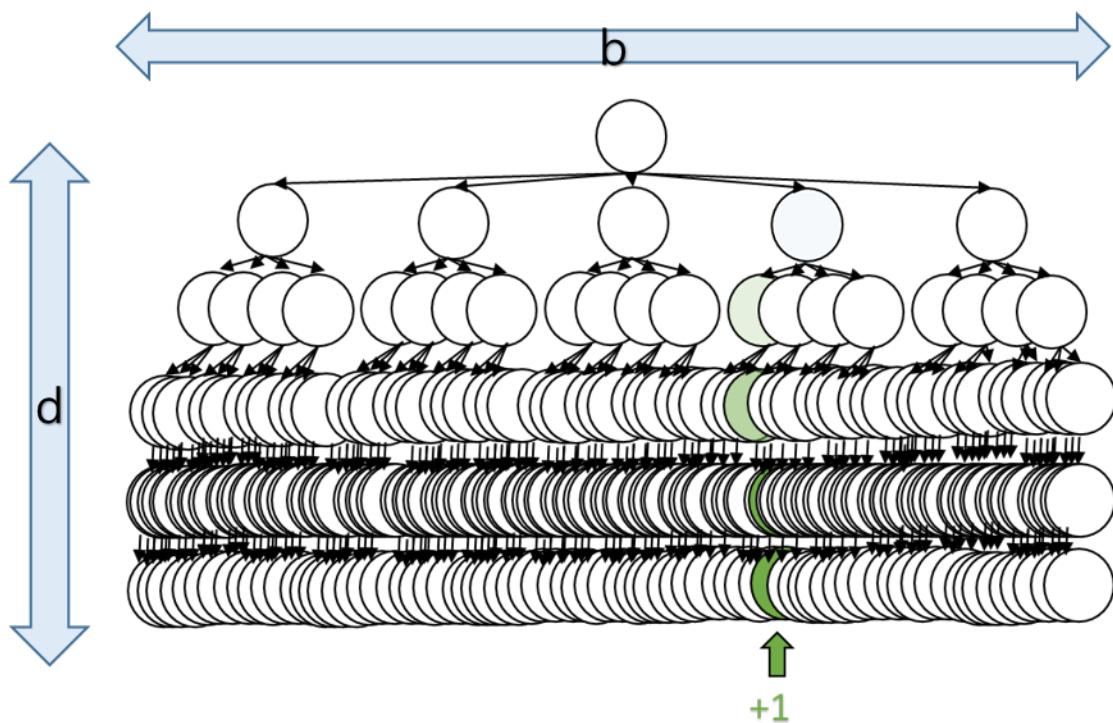


Fig. 12: 상태공간 트리

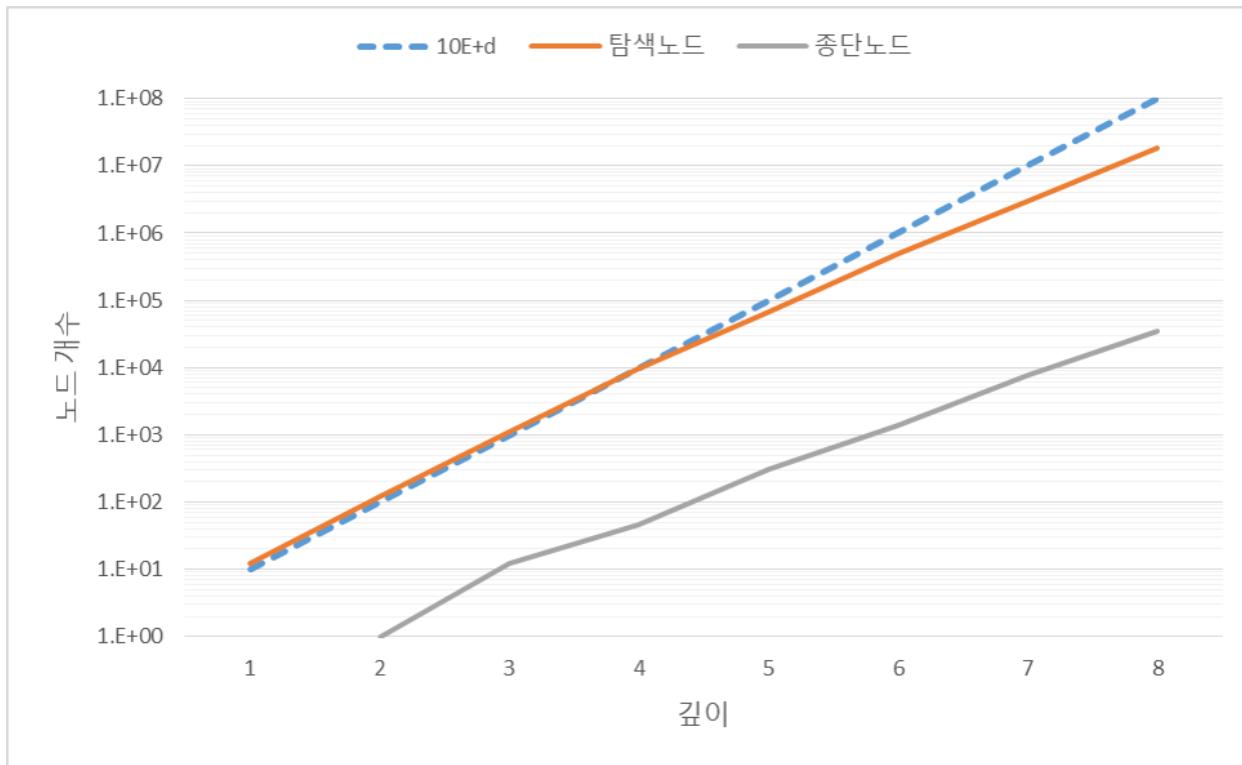


Fig. 13: 탐색공간의 규모

게임 시작상태부터 실제로 너비우선 탐색으로(breadth-first search)로 가능한 모든 상태를 순회하면 전체 탐색공간의 규모⁷를 예상해 볼 수 있다. 깊이 8까지 순회한 결과 상태공간의 개수는 예측한 결과와 유사하게 지수적으로 증가하는 것을 볼 수 있다. 깊이가 깊어질 수록 점차 예상보다는 줄어들기는 하지만, 깊이 8에서 증가폭은 여전히 매우 크다. 또한 순회한 상태 중에서 게임이 종료된 상태(종단노드)에 도달하는 수는 전체 공간의 0.01~0.001에 불과하다는 점도 알 수 있다. 때문에 아무런 정보도 없이 탐색 기법만으로 게임이 종료될 상태를 찾아낼 확률은 매우 낮다.

탐색공간이 크기 때문에 게임 도중 탐색하는 대신에, 사전에 미리 탐색하여 데이터를 테이블 형태로 정리해놓고, 탐색을 대체할 수도 있는데, Microchess 모든 경우의 수를 모두 조사하여 디스크에 저장하려 한다면, 한 상태에 1 Byte씩 소모된다고 해도(실제로는 최소 24bit 필요) 100,000,000TB의 용량이 필요하다.

따라서, 특별한 기술이나 지식없이 게임 도중에 실시간으로 탐색하거나, 사전에 모두 탐색하여 테이블 형태로 정리해 놓거나, 모든 경우의 수를 탐색하는 것은 상당한 시간과 메모리/디스크 용량을 필요로 한다.

일반 체스에서는 기물이 7~5개 남은 상태에서 모든 경우의 수를 계산해 놓은 Endgame tablebase⁷ 이 있는데, 이 테이블을 참조하면 게임의 종반부에서 더 이상 게임을 진행할 필요 없이 언제 승패가 결정될지를 알 수 있다. 일반 체스에서 7개 기물을 대상으로 한 Syzygy endgame tablebase의 용량은 약 140TB에 달한다고 알려져 있다. 그러나 Microchess는 기물의 개수가 더 많은데다가, 승리규칙도 조금 다르기 때문에 이 테이블을 직접 사용할 수도 없다.

2.3.2.7 평가함수

One Step Search 부터 ABP Negamax Search 까지 목표상태로 가는 방향을 알아내기 위해 더 깊이 탐색하는 방법을 사용했다. 더 깊이 탐색할 수록 수평선 효과를 완화할 수 있고, 결과적으로 더 높은 성능을 기대할 수 있기 때문이다. 그러나, 깊이 탐색할 수록 지수적으로 늘어나는 상태공간을 탐색해야하고 일정시점을 넘어가면 더 이상 감당할 수 없는 수준이 된다.

때문에 많은 AI들은 여러가지 다른 접근방법을 동시에 사용한다. 그 중 하나가 정교한 평가함수를 사용하는 것이다. 지금까지 예제 AI들은 단순히 기물 점수를 이용해서 현재 상태가 얼마나 좋은지 나쁜지를 판단한다. 그러나, 좋은 성능을 보이는 유명한 AI들은 그보다 훨씬 복잡한 평가함수를 사용한다. 기물의 절대적인 위치, 다른 기물과의 상대적인 위치 등 다양한 정보를 이용하면, 간단한 평가함수로 잡아낼 수 없는 좋고 나쁨에 대한 신호를 잡아낼 수 있다. 비록 탐색깊이를 계산 가능한 수준까지 줄이더라도 평가함수를 이용하면 수평선 효과를 완화할 수 있다.

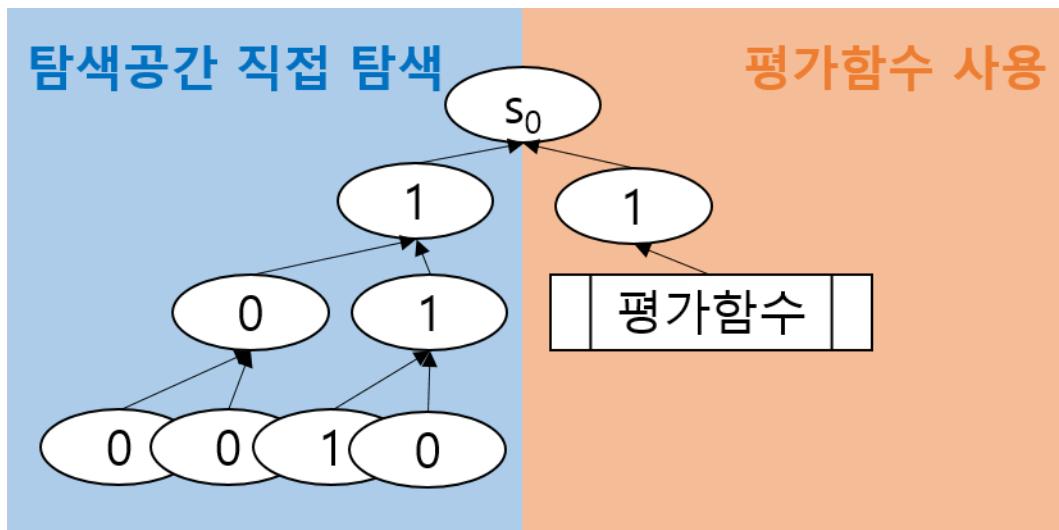


Fig. 14: 평가함수

평가함수 예에서 왼쪽은 단순한 탐색기법으로 상태공간을 탐색했는데, 총 7개 노드를 탐색했다. 그러나, 평가함수를 사용한 오른쪽은 단 한 개의 노드만을 탐색하고 평가함수로 그것이 얼마나 좋은지 예측하였다. 만약 평가함수가 충분히 정확하다면, 1개 노드 탐색 + 평가함수 실행으로 7개 노드 탐색을 대체하여 계산비용을 대폭 줄일 수 있다.

⁷ 일반 Chess 상태의 수 $b^d = 35^{80}$

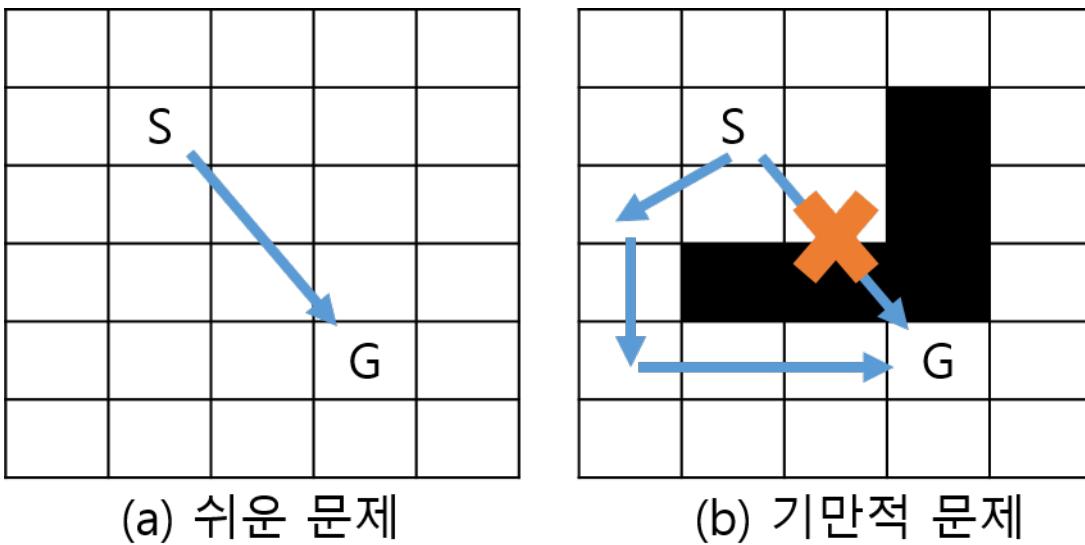


Fig. 15: 경로탐색 문제

대부분 평가함수는 문제(게임)에 대한 전문지식을 가진 전문가에 의해 설계되며, 매우 많은 노력이 필요하다. 잘못된 평가함수는 탐색공간을 왜곡하여 탐색알고리즘이 최종 목표상태를 찾는 것을 방해할 위험성이 있다. 출발지점 S 부터 목표지점 G까지 경로탐색 문제 (a)에서, 각 상태에서 선택할 수 있는 행동이 4가지(상, 하, 좌, 우 이동)라고하면, S 부터 G까지 최소거리는 5이기 때문에, 첫 번째 이동방향을 결정하기 위해 기본적인 트리 탐색으로는 최소한 깊이 5를 탐색할 필요가 있다. 따라서, 최소 $4^5 = 1,024$ 개 노드를 탐색하지 않으면 수평선 효과 때문에, 시작 지점에서 어느 위치로 움직여야 하는지 판단할 수 없다.

경로탐색 문제에서 가장 쉽게 사용할 수 있는 평가함수는 현재 위치와 목표지점사이의 거리가 얼마나 가까운지를 이용하는 것이다. 현재 위치와 목표지점이 가까울 수록 큰 평가를 받는 함수(예. 1/거리; $1/(Gx - Sx) + (Gy - Sy)$)를 설계하고 탐색에 사용하면, 트리 탐색으로 목표지점에 도달하지 못하더라도, 어느 방향으로 이동하는것이 좀 더 가까운지를 판단할 수 있기 때문에, 수평선 효과를 완화할 수 있다.

그러나, 경로탐색 문제 (b) 같은 환경에서는 똑같은 평가함수가 경로탐색에 오히려 방해가 된다. 목표지점까지 절대적인 거리가 가까운 최단경로에는 큰 장애물이 있기 때문에, AI는 절대 목표지점에 도착하지 못한다. 이 경우에는 오히려 일시적으로 목표지점으로부터 멀어지는 장애물을 우회하는 경로가 더 가까운 경로지만, 평가함수는 그것을 반영하지 않았다. 이런 문제를 기만적 문제(deceptive problem)⁸이라고 한다.

경로탐색 문제 (b) 예에서처럼 문제의 규모가 매우 작고 장애물이 명확하게 보인다면 평가함수를 개선할 수 있지만, 대부분의 문제는 이렇게 쉽게 해결되지 않기 때문에 좋은 평가함수를 설계하는 것은 매우 어려운 작업이다.

이론적으로는, 매우 정교한 평가 함수를 정의할 수 있어서 모든 상태에서 각 상태가 얼마나 상대적으로 좋은지 알 아낼 수 있다면, 언제나 현재 상태보다 더 좋은 상태로 연결되는 행동을 선택함으로써, 목표상태에 도달할 수 있다. 이 경우 One Step Search 만으로도 목표상태를 찾아갈 수 있다. 그러나 실질적으로 우리는 각 상태가 목표지점에 얼마나 정확한지 정확하게 알 수 없기 때문에, 우리가 신뢰할만한 평가값을 알 수 있는 상태까지 깊이 탐색할 수 있는 효율적인 탐색알고리즘이 도움이 된다.

2.3.2.8 Stockfish

실제로 강력한 체스 AI들은 빠르고 효율적인 탐색알고리즘(Minimax Search + $\alpha - \beta$ pruning)과 정교한 평가함수를 결합한 형태로 구현되었다고 알려져 있다. 계산 자원이 한정된 상황에서 탐색능력이 비슷하다면 그들 사이에 성능을 결정짓는 것은 얼마나 정교하고 정확한 평가 함수를 사용하느냐에 달려있다.

⁸ Chen, Yang, et al. "Solving deceptive problems using a genetic algorithm with reserve selection." Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on. IEEE, 2008.

Stockfish⁹ 는 그 중에서도 가장 성능이 좋은 체스 AI로 알려져 있다. 개인용 데스크톱이나 스마트폰에서 작동할 수 있을 정도로 가볍지만, 체스 AI 중에서 가장 높은 성능을 보여준다. 이 플랫폼에서도 *Stockfish* 래퍼로 Stockfish 8을 사용할 수 있다.

경진대회 플랫폼에서 직접 지원하지는 않지만, 원래 체스 AI 분야에서는 여러가지 방식으로 구현된 체스 AI(일반적으로는 체스 엔진이라고 표현)들이 서로 게임을 플레이할 수 있도록 UCI(Universal Chess Interface)¹⁰ 라는 공통 인터페이스를 정의했다. 이 플랫폼에서는 python-chess의 UCI 지원기능¹¹ 을 이용해 다른 언어(C++)로 구현된 Stockfish를 간접적으로 지원한다. UCI는 PIPE를 이용해 서로 인터페이스 하므로, 이것을 구현하기만 하면 python 이외에 다른 언어로도 AI를 구현할 수 있다.

Stockfish는 매우 효율적인 탐색알고리즘과 정교한 평가함수를 가지고 있기 때문에 Microchess에서도 뛰어난 성능을 보여준다. 다만, 경진대회의 평가기준과 달리 일반 체스의 규칙을 따르기 때문에 일반 체스에서 Stockfish의 실력보다 이 플랫폼의 Stockfish는 낮은 성능을 보이고 있을 가능성이 높다. 예를 들어, 경진대회 규칙으로는 패배지만, 일반 체스의 규칙으로는 무승부라면 Stockfish는 무승부 상태라고 잘못 판단하기 때문에, 실제 게임에서는 패배할 것이다. 이런 제약에도 불구하고, Stockfish는 상당히 강력해서, 지금까지 소개한 예제 AI들은 Stockfish를 이기기 매우 어렵다.

하지만, 경진대회에서 규칙에서 기준 Stockfish의 성능을 더 높이는 작업은 매우 어려울 가능성이 높다. 빠르고 효율적인 탐색 알고리즘 구현은 체스와 관련된 전문지식과 거의 상관없이 때문에 쉽게 재활용 가능하지만, 일반 체스에 최적화된 평가함수를 Microchess에 맞게 수정하는 것은, 매우 많은 시행착오가 필요할 것이다.

⁹ <https://stockfishchess.org/>

¹⁰ https://it.wikipedia.org/wiki/Universal_Chess_Interface

¹¹ <http://python-chess.readthedocs.io/en/latest/uci.html>

2.4 Monte Carlo Tree Search

지금까지는 상태공간 탐색에 대한 기본적인 사항을 알아보았고, 탐색기반 예제 AI들과 일반 체스에서 가장 강력한 AI인 Stockfish를 살펴보았다. Microchess가 간단한 게임에도 불구하고, 탐색공간이 매우 커질 수 있기 때문에, 간단한 예제 AI들의 성능은 충분하지 않았다. 심지어 기존에 가장 강력한 체스 AI인 Stockfish라고 하더라도 평가기준이 달라진 경진대회에서는 상당히 정밀한 수정이 없이는 기존 성능을 보여주지 못하기 때문에, 이 이상의 성능을 위해서는 새로운 방식으로 AI를 구현해야만 한다. 지금부터는 MCTS (Monte-Carlo Tree Search)와 Self Learning으로 Stockfish 이상의 성능을 달성하는 방법을 보인다.

2.4.1 MCTS 개요

MCTS (Monte-Carlo Tree Search)¹의 기본 아이디어는 탐색 공간이 거대하여 모두 탐색이 불가능하다면, 무작위로 상태공간을 표본추출(sampling)하여 통계적인 근사치를 구하고, 그 근사치에 근거한 의사결정을 하겠다는 것이다. 만약 충분히 많은 표본추출을 할 수 있다면 어떤 행동을 하는 것이 가장 승률이 높은지를 추정할 수 있다. 하나의 표본을 추출하는 과정은 현재 상태부터 마지막(예. 게임종료)까지 무작위로 행동을 결정하여 첫 번째 행동의 보상(평가값)을 알아내는 것이며, 이것을 시뮬레이션이라고 한다.

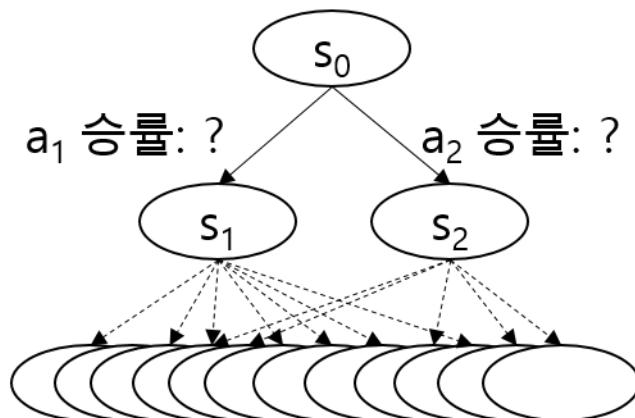


Fig. 16: 간단한 MCTS

간단한 MCTS는 현재 상태(S_0)에서 여러 번 무작위로 시뮬레이션(점선)을 하고, 현재 선택 할 수 있는 행동들의 기대 보상을 추정하는 것이다. 무작위 시뮬레이션을 충분히 많이 해볼 수 있다면, 특정 상태에서 행동마다 기대 보상을 추정할 수 있기 때문에, 전체 공간을 탐색하지 않더라도, 현재 상태에서 의사결정을 할 수 있다. 이 방법의 성능은 충분한 시뮬레이션에 달려있기 때문에, 기본 계산비용이 큰 경우가 많다.

단순 MCTS를 개선한 것이 UCT (Upper Confidence bound Tree search)¹ 알고리즘이다. 이 알고리즘은 시뮬레이션을 할 때 완전히 무작위로 행동을 결정하는 대신, 이전에 시뮬레이션한 정보를 이용하여 시뮬레이션의 효율을 개선한 것이다. 요즘은 보통 UCT를 그냥 MCTS라고 부른다.

2.4.2 Multi-Armed Bandit

MCTS에서 특정 한 상태를 떼어놓고 보면(*MCTS → Multi-Armed Bandit*), 이것은 비결정적인 보상을 반환하는 환경에서 특정 상태에서 어떤 행동을 하는것이 더 좋은지를 알아내는 문제이다. 지금까지 살펴본 탐색 기반 AI에서 취하는 기본 접근방법은 결정적인 환경을 가정하고, 특정 상태에서 가능한 행동들을 모두 시뮬레이션 해본다음, 그 결과를 바탕으로 어떤 행동이 더 좋은지 알아내는 것이다. 그러나, 만약 시뮬레이션 결과가 (MCTS의 무작위 탐색

¹ Browne, Cameron B., et al. "A survey of monte carlo tree search methods." IEEE Transactions on Computational Intelligence and AI in games 4.1 (2012): 1-43.

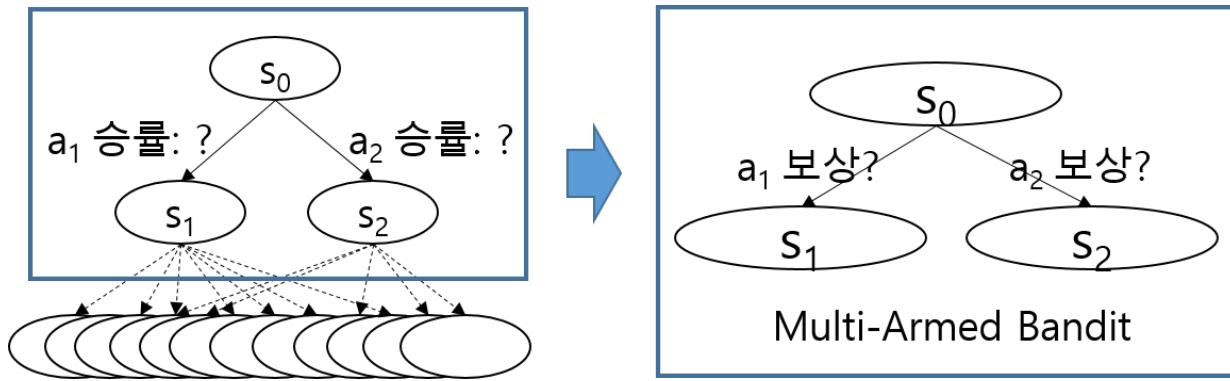


Fig. 17: MCTS → Multi-Armed Bandit

때문에) 시뮬레이션을 수행 할 때마다 달라진다면, 가능한 행동들 중에서 더 좋은 행동을 찾기위해서는 모든 행동들을 여러 번 반복하여 시뮬레이션 해보고, 통계적인 추정을 할 필요가 있다. 시뮬레이션을 하면 할 수록 추정이 정확해 지겠지만, 계산비용을 절약하기 위해서는 최소한의 시뮬레이션 만으로 정확한 추정을 할 필요가 있다.

이런 문제를 탐색과 활용 딜레마(exploration-exploitation dilemma)라고 한다. 탐색과 활용 딜레마가 발생하는 가장 간단한 문제는 MAB (Multi-Armed Bandit) 문제이다. 여기서 Bandit란 슬롯머신을 의미한다. 만약 상금이 나올 확률이 서로 다른 여러 대의 슬롯머신이 있다면, 가장 높은 상금을 타는 방법은 어떤 슬롯머신이 더 확률이 높은지를 알아내고, 그 슬롯머신에서 게임을 계속 시도하는 것이다. 그러나 문제는 슬롯머신에서 상금을 탈 확률을 알아내기 위해서는 충분한 게임(실험)을 해서, 통계적인 추정치를 알아내야만 한다는 것이다. 투자할 수 있는 총 자금에는 한계가 있고, 실험에는 비용이 들기 때문에 지나친 게임은 최종 기대이익을 낮출 수 있다.

이 문제에 대한 가장 간단한 해법 중 하나는 ϵ - greedy 알고리즘이다. 이 알고리즘은 전체 자원 중 일정 비율(ϵ)을 실험을 위해 사용하는 알고리즘이다. 예를 들어 현재 가진 돈으로 1,000게임을 할 수 있다면, 그 중 500게임을 실험을 위해 사용하는 식이다. 2대의 슬롯머신이 있다면, 500게임을 250게임씩 균등하게 나눠서 실험해보고, 그 중에서 상금이 더 높은 슬롯머신에서 나머지 500게임을 하는 것이다. 흔히 처음에 실험을 위해 하는 500게임을 탐색(exploration)이라고 하고, 나중에 상금을 얻기 위해 하는 행동을 활용(exploitation)이라고 한다.

이 방법은 직관적으로 알 수 있다시피, 탐색과 활용의 비율을 잘 조절하는 것이 기대이익을 높이는 핵심이다. 탐색에 너무 많은 게임을 하면, 이익을 극대화 시키기 위한 활동인 활용의 비율이 낮아져 최종 상금이 낮아진다. 반면, 탐색에 너무 적은 게임을 할당하면, 부정확한 추정에 근거해 활용을 하기 때문에, 최종 기대보상이 낮아진다. 따라서, 탐색과 활용의 비율을 적절히 조정하는 것이 이 문제를 잘 해결하는 방법이다. 이 문제, 탐색과 활용 딜레마(dilemma)는 강화학습에서 중요한 문제 중 하나이다.

UCB (Upper Confidence Bound)는 ϵ 보다 좀 더 정교한 방식으로 탐색과 활용을 조정한다. 반복해서 시뮬레이션을 하면서, 얻은 예측의 신뢰도(upper confidence)를 이용해 현재 추정치가 얼마나 신뢰할 만한지를 판단한다. ϵ 처럼 명확하게 탐색과 활용을 분리하지 않고, 언제나 각 행동에 대해 현재 보상과 신뢰도에 의해 결정된 UCB 값이 가장 높은 행동을 결정한다. 기본적으로 기대 보상이 높으며, 신뢰도가 낮은 행동일 수록 선택될 가능성이 높다. (2.2)은 가장 대표적인 UCB 공식이다. 첫 번째 항은 활용에 대응되는 기대보상(평균)이며, 두 번째 항은 탐색에 대응되는 값이다. 두 번째 항의 N 은 전체 시뮬레이션 횟수이고, n_a 는 그 중에서 특정 행동 a 를 시뮬레이션한 횟수이다. 따라서, 전체 시뮬레이션 횟수에 비해서 a 를 시뮬레이션 한 횟수가 클 수록 두 번째 항은 자연스럽게 줄어든다.

$$UCB_a = \frac{v_a}{n_a} + \sqrt{\frac{2\log N}{n_a}} \quad (2.2)$$

ϵ -greedy는 적절한 ϵ 를 알고 있다면, 좋은 결과를 얻을 수 있지만, 좋은 ϵ 값을 모를 경우 UCB가 보다 좋은 성능을 보이는 경우가 많다. 다만, 끝까지 탐색을 멈추지 않기 때문에 충분히 탐색을 한 뒤에도 예상과 달리 전혀 영뚱한 행동을 선택하기도 한다.

2.4.3 Upper Confidence Tree

MCTS의 시뮬레이션은 결국 각 단계에서 MAB 문제를 연속적으로 풀고 있는 것으로 볼 수 있다. 각 깊이에서 아직 충분히 시도해 보지 않은 행동을 실험(탐색) 할 것인지, 아니면 지금까지 시뮬레이션 한 결과 가장 좋았던 행동에서 연결되는 미래상태를 더 시뮬레이션하여 추정치를 정교하게 개선(활용)할 것인지를 끊임없이 결정하는 것이다. UCT (Upper Confidence Tree)는 이 과정에 UCB를 사용하여 MCTS의 성능을 높인 것이다.

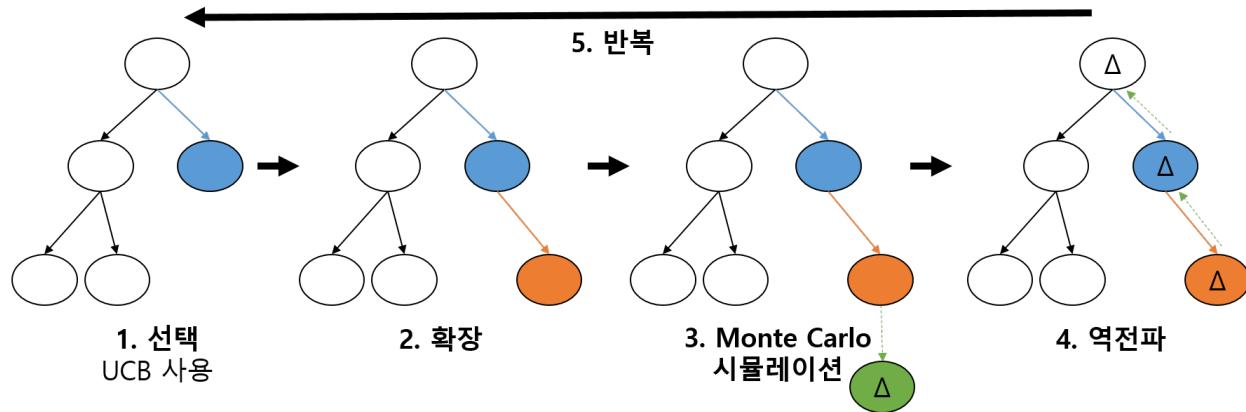


Fig. 18: UCT 알고리즘

무작위 탐색(Monte-Carlo simulation)만을 하는 기존 MCTS와 달리 UCT는 UCB를 사용하는 Tree Policy 와 Default Policy (무작위 탐색) 단계로 구분된다. [UCT 알고리즘](#)에 UCT의 네 단계를 설명하였다. 첫 번째 선택 단계에서는 (2.2)을 이용하여 현재 상태에서 UCB값이 가장 큰 행동을 선택한다. 제일 처음에 시뮬레이션을 시작할 때나 현재 까지 탐색을 해본 상태를 따라 깊게 내려가다보면, 아직 한번도 시도해보지 않은 행동이 나타난다. 이때, 두 번째 확장 단계로 넘어간다. 이 단계에서는 아직 시도해보지 않은 행동들 중에서 무작위로 하나를 골라 현재까지 탐색 해본 트리의 끝부분에 상태 노드를 하나 추가한다. 이 두 단계를 Tree Policy라고 한다. 세 번째 단계에서는 기존 MCTS 처럼 게임이 종료되는 상태까지 무작위로 트리를 탐색한다. 이것을 Default Policy라고 한다. 마지막 네 번째 단계에서는 게임이 종료되었을 때의 결과를 역전파하여 트리 노드의 기대 보상값을 개선한다. 개선된 보상값은 다음 반복(iteration)때, Tree Policy에서 사용된다.

Listing 5: MCTS 기본코드

```
v0 = Node(self.turn, state)
start_time = time.perf_counter()
simulation_count = 0
while True:
    if simulation_count > n_simulations or time.perf_counter() - start_time > 0.99 * timeout:
        # 제약조건(시뮬레이션 횟수와 시간제한)을 초과하면 바로 시뮬레이션 종료
        break

    self._depth = 0
    # Tree Policy: 선택 및 확장
    v1 = self.tree_policy(v0)
    # Default Policy: Monte-Carlo 시뮬레이션
    delta = self.default_policy(v1.state)
    # Backup: 보상 역전파
    self.backup(v1, delta)
    simulation_count += 1
```

MCTS의 전체 흐름은 [MCTS 기본코드](#)에서 볼 수 있다. 주어진 자원(시뮬레이션 횟수 또는 시간제한)동안 계속 시뮬레이션을 반복하면서, Tree를 확장해 간다.

이 네 단계를 한번 수행하는 것을 시뮬레이션이라고 하고, 이 과정을 충분히 반복하면 현재 상태(root node)에서 각 행동들의 기대 보상값을 추정할 수 있다. 최종 의사결정을 할 때는 행동의 기대보상값(보상 / 실험 횟수)가 가장 높은 행동을 고르기도 하지만, 시도 횟수가 가장 많은 것을 고르는 경우가 많다.

Minimax 같은 기존 탐색 알고리즘은 기본적으로 수평선 효과를 완화하기 위해 깊은 탐색이 필요하다. 탐색의 깊이가 깊어질 수록 탐색해야 할 상태가 지수적으로 증가하기 때문에, 계산복잡도를 대강 $O(b^d)$ 로 생각할 수 있다 (b : 평균 선택가능한 행동, d : 탐색깊이). 반면에 MCTS는 시뮬레이션 횟수(Monte-Carlo 샘플링 횟수)에 따라 계산 비용이 증가한다. 대강의 계산복잡도는 $O(nd)$ 로 볼 수 있다(n : 시뮬레이션 횟수, d : 탐색깊이). 따라서 상태공간의 규모가 커질 수록 상대적으로 적은 계산 비용만을 필요로 한다.

주의할 점은, MCTS로 충분한 성능을 얻으려면, 충분한 시뮬레이션이 필요하다는 것이다.. 매우 오랜시간이 걸리지만 최적해를 찾는 것이 모장된 기존 탐색알고리즘과 달리, MCTS는 근사해를 구하는 알고리즘이기 때문에 최적해를 찾는다고 보장할 수 없다. 이것을 해결하는 직접적인 방법은 충분한 시뮬레이션 횟수를 확보하는 것 뿐이다. 그러나, 너무 큰 시뮬레이션 횟수도 불필요하다. 비록 이론적으로는 MCTS의 시간복잡도가 선형으로 증가하기는 하지만, 실제구현에서는 그 외 요소도 있기 때문에 너무 큰 시뮬레이션 횟수는 큰 부담이 되는 경우가 많다. 때문에 대부분의 구현에서는 시뮬레이션을 몇 번 할 것인지 미리 지정해두기보다는, 주어진 시간에 맞춰 최대한 많이 시뮬레이션을 하는 방식을 사용한다. 이 경우 만약 주어진 시간이 변경되더라도 그에 맞춰 언제나 최선의 성능(근사해)을 보일 수 있다.

2.4.4 MCTS 예제

MCTS 예제는 *MCTSAgent*에 구현되어 있다. 기본 MCTS 구현은 Microchess 정도의 문제이에서도 충분한 성능을 보이지 못하기 때문에, 두 가지를 수정 했다.

탐색깊이 제한

원래 MCTS는 게임의 종료상태까지 탐색을 시도하고, 종료상태에서 승리했는지 패배했는지에 따라 보상을 받고, 그 정보를 이용해 의사결정을 하는 것이지만, 실질적으로는 너무 깊은 공간을 탐색하는 것은 큰 계산비용이 소모된다. 뿐만아니라, 탐색할 공간이 커질 수록 충분히 정확한 결과를 얻기 위해 시뮬레이션의 횟수가 많이 필요하기 때문에 탐색깊이를 최대 6으로 제한하고, 마지막에서 평가함수를 사용하여 계산비용을 절감하였다. 비록 정교하지 못한 평가함수로 인해 보상의 추정이 왜곡될 위험이 있기는 하지만, 주어진 시뮬레이션 횟수에서 탐색깊이를 제한하지 않으면, 부정확한 추정으로 인해 명백히 나쁜 수를 둘 확률이 높았다. 체스 같은 턴제 게임에서는 한번의 나쁜 수가 승패를 가를 정도로 중요하기 때문에, 이것을 막기위해 탐색깊이를 제한했다.

보상신호 강화

탐색깊이를 제한했지만, 깊이가 깊어질 수록 노드의 개수가 10배씩 증가하고, 보상신호는 1/10씩 감소하기 때문에, 깊이 6정도에서도 보상신호는 매우 약해졌다. 즉 현재 상태에서 좋은 행동과 나쁜 행동의 차이를 구분하기 어려워졌다. 게다가, 탐색깊이를 제한하고, 평가함수를 사용하면서 가장 좋은 상태와 가장 나쁜 상태의 차이는 더 작아졌다. 이 문제를 완화하기 위해 보상을 평가함수의 출력을 그대로 사용하지 않고, 마지막 상태가 현재 상태보다 좋은을 때는 보상 1, 나쁠 때는 보상 0으로 사용하였다. 이 변경으로 인해 상태적인 보상의 크기가 훨씬 커지게 되었고, AI는 보다 그럴듯하게 작동하게 되었다. 다만, 마지막 상태가 현재 상태보다 좋기만하다면 모두 동일하게 보상 1을 받기 때문에 더 좋은 수와 덜 좋은 수를 구분하지는 못하게 되었다.

탐색깊이 제한과 보상신호를 강화하기 전에 MCTS는 너무 낮은 보상신호 때문에, 종종 매우 나쁜 수를 두고는 했다. 한 게임 동안에 반드시 몇 번을 그런 수를 두기 때문에 승률을 높이기 어려웠다. 하지만, 개선한 MCTS는 훨씬 안정적이고 그럴듯하게 작동했다.

구현한 MCTS 예제의 성능을 평가하기 위해 Stockfish와 비교하였다. Stockfish는 비록 Microchess 용으로 개발된 AI도 아니고, 승리/패배 규칙도 경진대회 규칙과 다르지만, 상당히 강력한 성능을 보인다. 다른 예제 AI는 Stockfish를 상대로 0.3 이상의 승률을 보이기 어렵다. 게임을 잘 플레이한 경우에도 Stockfish를 상대로 승리하기는 어렵고, 겪우 비기는 경우였다.

하지만, MCTS는 Stockfish를 상대로 평균(40게임) 0.562의 승률을 보였다. White로 20게임 Black으로 20게임을 했는데, White일 때는 0.85, Black일 때는 0.275를 기록했다. 일정 수준의 실력을 보이는 AI끼리는 누가봐도 명백한 실수를 하지 않기 때문에 먼저시작하는 White로 할 때 이기고, 나중에 두는 Black일 때 진다. 따라서 평균 승률을 높이기 위해서는 White일때 확실히 이겨야 하고, Black일 때 비기거나 쳐야 한다. MCTS도 Black일 때 이겼 경우는 20게임 중 불과 2게임에 불과하다.

Stockfish는 체스에 대한 전문적인 지식을 가지고 설계된 매우 정교한 평가함수와 매우 빠른 탐색 알고리즘을 사용하기 때문에 매우 가벼우면서 높은 성능을 가지고 있다. 하지만, 이 경진대회처럼 게임의 규칙이 바뀐 경우, 여기에 대처하도록 개선하는 것은 쉽지 않다. Microchess는 일반 체스와 유사하기 때문에, 어느 정도 성능을 보일 수 있었지만, 만약 전혀 다른 보드 게임이라면 Stockfish가 사용하는 정교한 평가함수는 전혀 사용할 수 없을 것이기 때문에, 일정 규모 이상의 문제에서는 빠르고 효율적인 탐색 알고리즘으로도 해결하기 어려울 것이다.

반면 MCTS는 전문적인 지식에 거의 의존하지 않는다. 비록 예제 MCTS에서는 문제의 규모를 축소하기 위해 간단한 평가함수를 도입했지만, 이것은 매우 간단하기 때문에 다른 문제에서도 쉽게 이 수준의 평가함수를 구현할 수 있다. 그 대신에 MCTS는 수 많은 시뮬레이션으로 근사해를 찾아낸다. 실제로 예제 MCTS가 높은 성능을 내기 위해서는 사용할 수 있는 모든 시간동안 계속 시뮬레이션을 해야만 했다. 경진대회에서 한 턴에 약 10초를 가정했기 때문에, MCTS는 10초동안 약 7,000~12,000번의 시뮬레이션을 수행했다(Intel i7-7700). 만약 시간이 반 이하로 주어지거나, 실행하는 PC의 사양이 낮다면, MCTS가 Stockfish를 상대로 높은 승률을 보이기는 어려울 것이다.

2.4.5 MCTS 시각화

MCTS AI는 10초 시간제한으로 `agents.search.mcts_agent.MCTSAgent` 와 5000회 시뮬레이션 횟수 제한으로 실행되는 `agents.search.mcts_agent.MCTSAgentDev`가 있다. MCTSAgentDev가 작동하기 전에 visdom server를 실행시켜두면 MCTS 의사결정과정을 볼 수 있다. 시각화 기능에도 무시하기 어려운 계산비용이 필요하기 때문에, 기본적으로 개발용 버전인 MCTSAgentDev만 기능이 활성화 되어있다.

```
# visdom 서버 실행
(mchess) ~/ python -m visdom.server

# 다른 콘솔 창에서
(mchess) ~/ python scripts/run_game.py --white=mcts_dev --black=mcts
```

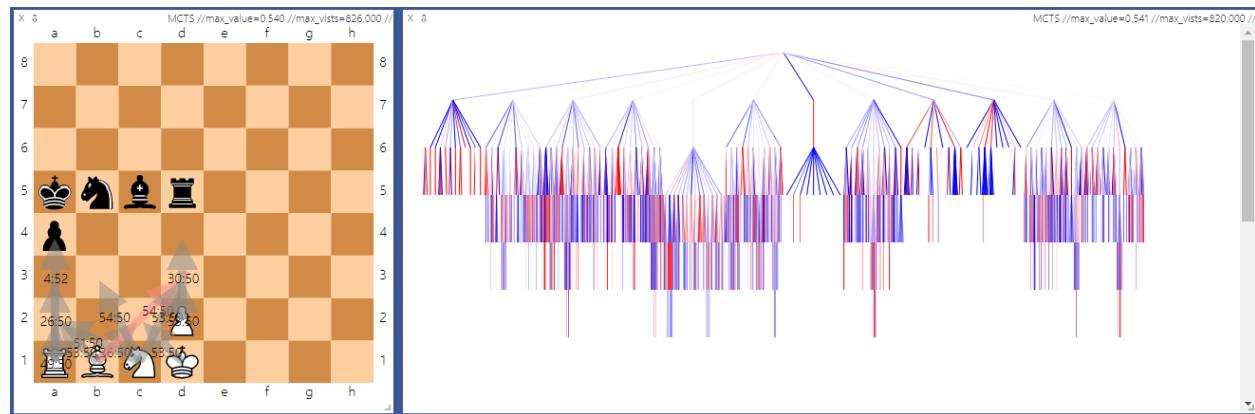


Fig. 19: MCTS 의사결정과정

2.5 Self Learning AI

MCTS를 포함해 트리 탐색 알고리즘의 성능을 개선하는 방법은 크게 두 가지가 있다. 첫째, 탐색할 가치가 있는 상태를 선별적으로 탐색한다. 둘째, 정교한 평가함수를 사용한다. 첫 번째 방법은 흔히 가지치기(pruning)라고 불리는 방법으로, 여러 행동들 중에서 실제로 탐색해보지 않더라도, 결과가 나쁠 것으로 예측 되는 것을 무시하는 기법이다. $\alpha - \beta$ pruning도 이런 기법의 일종이다. 이런 기법들은 $O(b^d)$ 에서 b 를 줄이는 효과가 있다. 두 번째 방법은 높은 성능을 보이는 많은 AI에서 사용하는 방법이다. 전문가에 의해 혹은 데이터에 기반해 설계된 정교한 평가함수를 이용해 실제 시뮬레이션을 해보지 않고 미래를 예측하는 것이다. 실제 결과와 차이가 있을 가능성이 있지만, 잘 설계한다면 $O(b^d)$ 에서 d 를 크게 줄일 수 있다. $\alpha - \beta$ pruning은 특정 게임에 대한 전문지식에 의존하지 않지만, 이런 기법들의 대부분은 특정 게임에 강하게 종속되어 있다. 때문에 Stockfish가 아무리 강력한 체스 AI라고 해도, 장기나 바둑같은 다른 보드게임을 전해 플레이 할 수 없는 것이다. 반면 MCTS는 전문지식에 거의 의존하지 않기 때문에 최소한의 변경만으로도 다른 게임에 쉽게 적용가능하다. 하지만, 기본 계산비용이 크기 때문에 계산비용을 줄이기 위한 방법이 필요하고, 결국 가지치기나 평가함수를 도입할 필요가 있다. 단, 가지치기나 평가함수가 전문지식에 의존하지 않아야 MCTS의 장점을 유지할 수 있다.

Alpha Go Zero¹는 이 문제를 해결한 가장 유명한 예이다. 기본 의사결정은 Monte-Carlo Tree Search (MCTS)를 사용하면서, 인공신경망을 이용해 가지치기와 평가를 수행한다. 기존 Alpha Go²도 이것과 동일한 접근방법이지만, 인공신경망을 학습할 때, 사람들이 플레이한 기보를 이용해 기본학습을 진행했다. 아무것도 모르는 처음(scratch)부터 학습하는 것은 어렵기 때문에 기보데이터를 이용해 기본적인 플레이방법을 학습하고, 그 이후 강화학습으로 성능을 향상시키는 방법을 사용했다. 반면 Alpha Go Zero는 처음부터 기보 데이터를 전혀 사용하지 않고, AI 스스로 게임을 플레이(Self Play)하면서 게임 데이터를 수집하고 학습 하였다. 따라서, Alpha Go Zero는 전혀 탐색알고리즘, 가지치기, 그리고 평가함수까지 알고리즘의 거의 모든 부분이 전문지식(도메인 지식)에 독립적이다. 그 덕분에 Alpha Go Zero가 발표된지 불과 몇 달 뒤에 바둑 뿐 아니라 다른 보드 게임을 모두 플레이 할 수 있는 Alpha Zero³가 공개되었다. Alpha Zero는 학습을 시작한지 불과 4시간만에 기존에 가장 강력한 AI였던 Stockfish를 뛰어넘는 성능에 도달할 수 있었다.

Self Learning AI 예제는 Alpha Go Zero와 유사한 방식으로 구현되었다. 실제 Alpha Zero는 매우 큰 계산비용을 요구하기 때문에, 최종 학습성능은 부족하더라도, 보다 빠르게 학습 결과를 확인할 수 있도록, 보다 간단하게 구현했다. 평가방식으로는 MCTS AI와 마찬가지로 Stockfish를 대상으로 승률을 측정했다. MCTS AI는 약 7,000~12,000번의 시뮬레이션(Intel i7-7700에서 10초 제한)에서 근소한 차이로 Stockfish를 이기는 수준이었지만, 약 4일을 학습한 인공신경망으로 성능을 강화한 Self Learning AI는 동일한 실험에서 승률 0.7을 기록했다. 특히 Self Learning AI가 White 일 때는 거의 모든 게임을 이겨서 승률 0.925를 기록했으며, Black일 때도 대부분 게임을 무승부로 끝내서 0.475를 기록했다.

2.5.1 Self Learning 알고리즘

Listing 6: Self Learning 알고리즘

```
memory = ReplayMemory()
best_model = create_mode()
current_model = copy(best_model)

while True:
    # self play: best_model 끼리 게임 플레이하고, 학습 데이터 생성
    for n in range(n_self_play):
        state, pi, legal_moves, win = play_game(best_model, best_model, ↵
        ↵exploration=True)
        reward = 2 * win - 1
        memory.add(state, pi, legal_moves, reward)
```

(continues on next page)

¹ Silver, David, et al. "Mastering the game of go without human knowledge." Nature 550.7676 (2017): 354.² Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." nature 529.7587 (2016): 484-489³ Silver, David, et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm." arXiv preprint arXiv:1712.01815 (2017).

(이전 페이지에서 계속)

```
# training: 학습 데이터로 current_model 학습
train_data = memory.get_minibatch()
for n in range(n_train):
    current_model.fit(train_data)

# evaluation: current_model이 best_model보다 좋아지면
#   current_model을 best_model로 교체
total_wins = 0
for n in range(n_evaluations):
    state, pi, legal_moves, win = play_game(current_model, best_model,
exploration=False):
    total_wins += win

if total_wins / n_evaluations > 0.6:
    best_model = copy(current_model)
```

Self Learning 알고리즘은 전체 알고리즘의 의사코드이다. 이 코드는 실제 예제 코드가 아니라, 설명하기 위해 단순화 시킨 코드이다. 이 알고리즘은 기본적으로 현재 학습한 가장 좋은 모델(인공신경망)끼리 게임을 플레이한 데이터를 이용해서 현재 모델을 학습하는 것이다. 한번 반복할 때마다, 현재 모델이 기존의 가장 좋은 모델보다 좋아졌는지 평가하고, 더 좋다고 판단될 경우 교체하기 때문에, 점진적으로 성능을 향상시켜 나갈 수 있다.

2.5.2 인공신경망

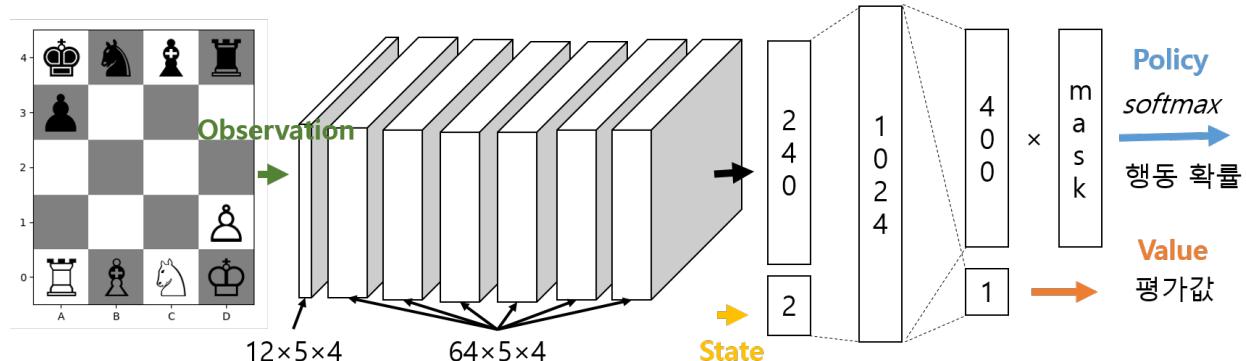


Fig. 20: 인공신경망 구조

Best model과 Current model은 현재 게임상태를 입력받아, 다음에 둘 수의 확률과 평가값을 출력하는 인공신경망이다. 예제 AI가 기본으로 사용하는 **인공신경망 구조**는 Convolution layer 6개 층에 Linear layer를 결합한 간단한 구조를 가지고 있다. Observation과 State 노드가 입력 노드이고, Policy와 Value 노드가 출력노드이다. 시각적으로 보이는 정보는 Observation 노드로 입력되어 64개의 필터(필터 크기: 3×3)를 가지고 있는 6개의 Convolution layer를 지난다. 그 뒤에 비시각적인 정보가 저장된 2차원 State노드와 결합하여 Linear layer를 거치고 최종적으로는 Policy와 Value 노드로 예측 값이 출력된다.

Observation은 $12 \times 5 \times 4$ 차원의 배열로 되어있다. Observation에 입력할 정보는 현재 보드 상태 수치연산에 적합하도록 배열 형태로 변형해야한다. 첫 번째 차원은 12종류의 기물마다 할당되어 있고, 두 번째와 세 번째 차원은 기물의 위치를 의미한다. 특정 기물이 특정 위치에 있으면 배열에서 해당하는 부분의 값은 1로 할당하고, 나머지 부분은 0으로 한다. 그 외 시각적으로 보이는 게임 상태 외의 정보를 입력하기 위한 것이 State이다. 여기에는 캐슬링 가능여부가 기록되어 있다. 캐슬링이 가능하면 1, 불가능하면 0으로 설정한다.

학습옵션 mirror에 따라 차원이 할당되는 부분이 달라진다. 기본적으로는 mirror 옵션이 켜져 있는데, 이 상태에서는 내 기물이 Observation의 1~6 채널을 사용하고 상대방의 기물이 6~12 채널을 사용한다. State에서도 내가 0

차원을 상대방이 1차원을 사용한다. 하지만, mirror 옵션을 끄면 White가 Observation의 1~6차원, State의 0차원을 사용한다.

출력부에는 평가값 출력노드(Value) 한 개와 정책 출력노드(Policy) 400개가 있다. Microchess 보드에 기물을 놓을 수 있는 위치는 총 20개 (5×4)이기 때문에, 현재 기물의 위치와 기물의 다음 위치의 모든 경우의 수는 총 400개가 된다. 하지만, 그 중에 대부분은 현재 기물이 없거나 기물이 움직일 수 없는 곳이기 때문에 현재 게임 상태에서는 유효하지 않다. 유효하지 않은 수와 관련된 출력을 mask를 사용해 제거한다. 유효하지 않은 수를 -100으로 덮어쓰면, 마지막 softmax 출력에서 0에 가까운 값이 된다. 학습의 결과 인공신경망의 Policy 출력은 현재 게임 상태에서 다음에 들 수의 확률(p ; probability)이 되어야 하며, Value 출력은 게임 상태가 얼마나 승리상태에 가까운지 [-1, 1] 사이 실수(v ; value)로 평가할 수 있어야 한다.

학습 데이터 샘플에는 게임상태(s ; state), MCTS의 탐색 빈도(π , 전체 합이 1.이 되도록 변환), 유효한 수 목록(legal moves), 그리고 게임의 승패(r ; reward)가 있어야 한다. 여기서 π 는 다음에 들 수의 확률 p 로, 게임의 승패 r 은 평가값 v 로 볼 수 있다. 그래서, loss 함수는 (2.3) 같이 정의된다. 첫 번째 항은 Policy 출력의 loss이고, 두 번째 항은 Value 출력의 loss이다. 마지막 항은 과적합을 막기위한 l_2 값이다.

$$L = -\pi \log p + c_1(r - v)^2 + c_2 \|\theta\|^2 \quad (2.3)$$

Listing 7: 신경망 학습

```

1 # optimizer 초기화
2 c1 = 0.02
3 c2 = 0.0001
4 optimizer = optim.SGD(current_model.parameters(), 0.01, momentum=0.9, weight_decay=c2)
5
6 # 중간 생략 ...
7
8 while True:
9
10    # 중간 생략 ...
11
12    # 입력 상태: ob, s
13    prob, value = current_model(ob, s)
14
15    # 실제 가능한 수에 마스킹을 함
16    prob = ((1 - legal_moves) * -100) + (legal_moves * prob)
17
18    # 신경망 가중치 갱신
19    optimizer.zero_grad()
20    log_prob = F.log_softmax(prob, dim=1)
21    policy_loss = -(pi * log_prob).mean()
22    value_loss = (z - value_pred).pow(2).mean()
23    loss = policy_loss + value_loss_coef * value_loss
24    loss.backward()

```

이것을 실제 코드로 작성하면 신경망 학습처럼된다(예제 AI와 다를 수 있음). 많은 인공신경망 라이브러리는 l_2 항을 따로 사용하지 않고, optimizer에 인자를 추가하는 식으로 처리한다(4번째 줄 weight_decay). Value loss 값은 [-1, 1] 사이 실수이고, 실제 더 커질 가능성도 충분히 있지만, Policy loss 값은 [0, 1] 범위인데다가 대부분이 0이므로, 평균적으로 Value loss 값이 Policy loss 값에 비해 약 50배 정도 크다. 이것을 보정하기위해 c_2 값을 0.02를 사용했다.

2.5.3 MCTS

Self Learning 구조 를 다시 살펴보면 MCTS와 인공신경망이 결합되어 있다. 게임을 플레이할 때, MCTS는 현재 게임상태에서 어떤 수를 탐색(search)해야할지 결정할 때, 선택과 Default Policy에서 인공신경망의 예측을 사용한다.

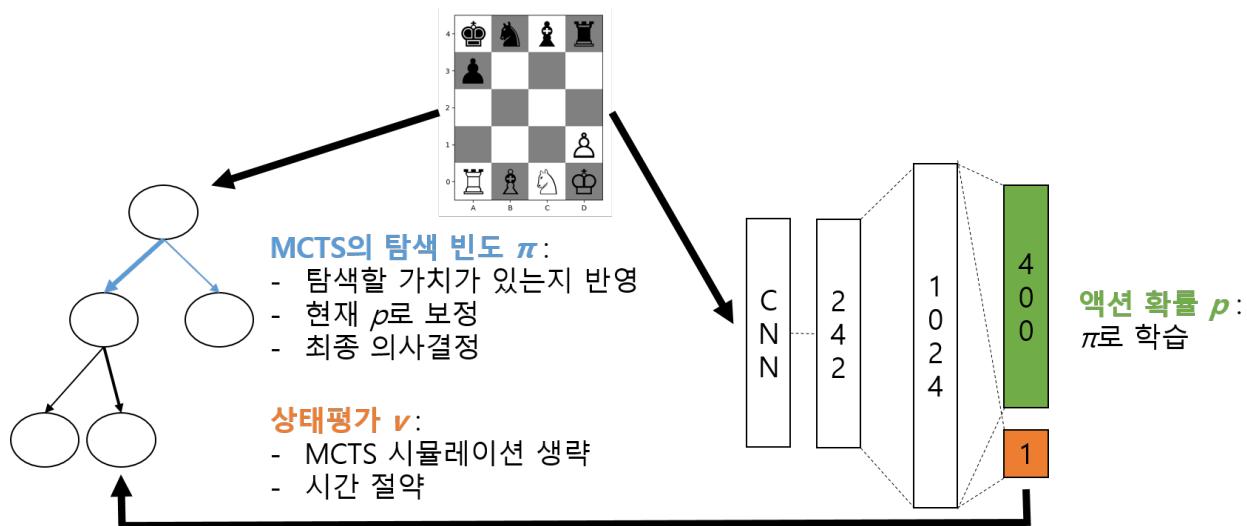


Fig. 21: Self Learning 구조

기존 MCTS에서는 선택 단계에서 UCB 알고리즘을 사용했는데, Self Learning에서는 그 변종인 PUCT 알고리즘을 사용한다. 기본적인 사항은 기존 UCB 알고리즘과 같지만, 두 번째 항(탐색)에서 신경망의 Policy 출력($P(s, a)$)를 사용한다. 신경망의 출력은 과거 MCTS에서 탐색했던 결과를 반영하고 있으므로, 과거에 유망했던 것으로 판단했던 수는 현재에도 더 탐색을 하게된다.

$$a_t = \text{argmax}(Q(s_t, a) + c_{puct} P(s_t, a) \frac{\sqrt{N}}{1 + N_a}) \quad (2.4)$$

Self Learning에서는 Default policy는 무작위 샘플링으로 상태의 좋고 나쁨을 판단하기 위한 것이었기 때문에, 이것 대신하여 인공 신경망의 Value 출력을 사용한다. Default policy는 시뮬레이션 횟수와 탐색 깊이에 비례하여 시간복잡도가 증가할 뿐만 아니라, 실제로는 상태노드를 생성하기 위해 메모리 할당과 해제가 빈번하게 발생하므로, 많은 경우 꽤 큰 계산비용을 요구한다. 반면, 인공신경망은 계산비용이 크기는 하지만, 병렬처리가 더 쉽고 고정되어 있기 때문에 훨씬 빠르다.

MCTS를 사용할 때 시뮬레이션 횟수를 주의해서 정해야 한다. MCTS 시뮬레이션 횟수가 크면 학습할 때도 더 높은 성능을 보일 가능성이 있지만, 학습시간이 오래 걸리기 때문에 제한된 시간동안 성능을 충분히 높이기 어렵다. 반면 시뮬레이션 횟수가 너무 부족하면 MCTS가 충분히 일관성있는 탐색이 어렵기 때문에 실제 성능이 향상되기 어렵다. 예제 AI에서는 학습할 때 시뮬레이션 횟수를 500번으로 하였다.

2.5.4 결론 및 한계

Self Learning으로 학습한 AI는 4일 정도 학습으로 두 Stockfish 보다 좋은 성능을 보일 수 있었다. 단순히 더 오랜 시간 학습을 하면 성능을 높일 수 있을 가능성은 있지만, 쉽지는 않다. 최종 학습결과가 초기 조건(신경망 초기값, Replay Memory 초기 데이터, 등)에 민감하기 때문에 일정수준 이상의 성능향상을 위해서는 더욱 개선이 필요하다. 현재 예제 AI는 Stockfish Level 5~6 정도까지는 하루 이틀 사이에 안정적으로 학습 가능하고, 8~10정도는 학습을 더 진행하면(약 4일) 학습이 가능하지만, 그 이상의 성능은 안정적으로 학습하지 못한다.

3.1 scripts package

3.1.1 Submodules

3.1.2 scripts.chess_board module

인간 플레이어 인터페이스와 관련된 클래스 모음

```
class scripts.chess_board.ChessBoard  
Bases: object
```

인간 플레이어 인터페이스 시작화 클래스

속성 **block_size** int, 체스 한 칸의 픽셀 크기

속성 **width** int, 마이크로 체스 가로 칸 수

속성 **height** int, 마이크로 체스 세로 칸 수

```
block_size = 62
```

```
height = 310
```

```
step(board, move=None)
```

board 객체와 move 객체를 입력받아, pygame 모듈을 사용해서 상태를 시작화 함

- AI 플레이어가 Move 를 입력하면, 단순히 현재 상태를 시작화 시킴고 그대로 Move를 반환
- 인간플레이어는 Move 객체대신 None을 입력하고, 이 함수에서 마우스 입력을 받아, 게임 상태를 변경할 Move 객체를 생성해서 반환함
- 폰을 마지막 칸으로 보내면 무조건 퀸으로 승급하도록 했음 (AI는 어떤 기물로든 승급가능)

매개 변수

- **board** (`State`) -

- **move** (*chess.Move*) –
반환 (*chess.Move*) –

width = 248

```
scripts.chess_board.micro_to_std = {0: 0, 1: 1, 2: 2, 3: 3, 4: 8, 5: 9, 6: 10, 7: 11, 8: 12, 9: 5, 10: 6, 11: 7, 12: 13, 13: 14, 14: 15, 15: 16}
```

체스 기물 이미지 파일 경로

```
scripts.chess_board.std_to_micro = {0: 0, 1: 1, 2: 2, 3: 3, 4: 8, 5: 9, 6: 10, 7: 11, 8: 12, 9: 5, 10: 6, 11: 7, 12: 13, 13: 14, 14: 15, 15: 16}
```

마이크로 체스 좌표와 일반 체스 좌표를 매핑

3.1.3 scripts.run_game module

scripts/run_game.py

Microchess AI 플랫폼에서 사용하는 체스와 관련된 기본 구성 요소 모음

```
class scripts.run_game.Environment
    Bases: object

    마이크로 체스 게임 환경의 기본 객체

    • 게임의 초기화, 진행, 종료에 관한 기능은 담당
    • 문자를 이용한 기초적인 시각화 가능
    • 그 이상의 시각화 기능은 chess_board.py (pygame)와 visdom에서 담당

    close()
    render()
        시각화
        • 언제나 jupyter console에 출력 가능한 svg 객체를 출력함 (jupyter console에서 시각화 됨)
        • 문자열을 이용해 체스 상태를 출력함 (대문자: white, 소문자: black)
        • 인간 플레이어가 한명이라도 게임을 할 때는 pygame을 이용함

    reset(fen=None, max_turns=80)
        환경 초기화

        매개 변수
        • fen (str) – fen 표기법으로 체스 보드 상태를 초기화 함, None 으로 전달할 경우 기본 초기화 상태로 세팅함
        • max_turns (int) – 최대 게임 턴, 기본값 80

    반환 (State) – 현재 게임 상태 반환

    step(move)
        현재 상태에 move를 적용하여 다음 턴으로 게임을 진행

        매개 변수 move – chess.Move, 현재 플레이어의 다음 수

        반환
            (State, float, bool, dict) –
            • State, move가 적용된 다음 게임 상태
            • float, 보상, 이번 수로 게임이 승리하면 1., 패배하면 -1., 그 외에는 0.5
            • bool, 이번 수로 게임이 종료되었는지 여부
```

- dict, 그 외 기타 정보 전달 용

```
class scripts.run_game.Evaluator
Bases: object
```

기본 예제에서 사용하는 평가함수 모음

현재 상태(board.Chess)와 현재 턴(bool, white=True, black=False)를 입력으로 받아, 평가 값을 [0, 1] 실수로 반환함

```
static eval(turn)
```

기물 점수를 계산하는 평가함수

- 현재 판위에 남아있는 기물의 종류와 개수에 따라 [0, 1] 실수값으로 평가함
- 게임의 승패가 결정되어 있다면, Evaluator.win_or_lose로 평가함
- 게임이 종료되지 않았다면 기물의 종류와 개수에 따라 점수를 계산한 뒤 [0, 1]값으로 정규화 시킴
- 절대적인, 상대적인 기물의 위치, 캐슬링 여부등 다른 요소는 고려하지 않음

기물의 점수

- Pawn: 1점
- Knight, Bissop: 3점
- Rook: 5점
- Queen: 10점
- King: 4점

매개 변수

- **board (State)** – 평가할 상태
- **turn (bool)** – chess.WHITE 또는 chess.BLACK

반환 (float) – [0, 1] 범위 실수

```
static eval_2(turn)
```

Evaluator.eval 결과([0, 1] 실수)를 [-1, 1] 실수로 변환

매개 변수

- **board (State)** – 평가할 상태
- **turn (bool)** – chess.WHITE 또는 chess.BLACK

반환 (float) – [-1, 1] 실수

```
static win_or_lose(turn)
```

승/패만 판단하는 간단한 평가함수

- 간단한 평가함수, [0., 0.5, 1.] 중 하나의 값을 반환
- 승리하면 1., 패배하면 0.을 반환함
- 나머지 경우에는 0.5 반환 (무승부, 게임이 진행 중)

매개 변수

- **board (State)** – 평가할 게임 상태
- **turn (bool)** – chess.WHITE 또는 chess.BLACK

반환 (float) – [0, 1] 범위 실수

```
class scripts.run_game.Move (from_square, to_square, promotion=None, drop=None)
Bases: chess.Move

class scripts.run_game.State (fen='8/8/8/knbr4/p7/8/3P4/RBNK4 w Kk - 0 1', chess960=False)
Bases: chess.Board

마이크로 체스 AI 플랫폼에서 게임 상태를 전달하기 위해 사용


- chess.Board + forward 기능
- chess.Board에서 제공하는 대부분의 기능은 그대로 사용 가능
- 향후 부정행위에 사용할 수 있는 기능이 발견되는 경우 일부 기능이 제한될 수 있음

black_remain_sec = 400
(int) – black 플레이어의 남은 시간

color

forward(move)
시뮬레이션 - 현재 State에 move를 적용한 다음 상태를 반환
    매개 변수 move (Move) –
    반환 (State) – move가 적용된 다음 상태

white_remain_sec = 400
(int) – white 플레이어의 남은 시간

scripts.run_game.TIME_LIMIT = 400
한 플레이어에게 주어진 시간제한, 한턴에 10초씩 80턴으로 계산해서 한 플레이어에 400초 할당

scripts.run_game.game (white, black, human_play, initial_fen, max_turns, timelimit)
게임을 1회 실행하는 함수
    매개 변수
        

- white (str) – white 플레이어의 객체(BaseAgent 상속한 클래스) 경로
- black (str) – black 플레이어의 객체(BaseAgent 상속한 클래스) 경로
- human_play (bool) – 인간 플레이어용 인터페이스를 사용여부
- initial_fen (str) – 게임의 초기상태를 fen으로 전달
- max_turns (int) – 최대 게임 턴, 경진대회 기본은 80
- timelimit (int) – 최대 게임 시간, 경진대회 기본은 400

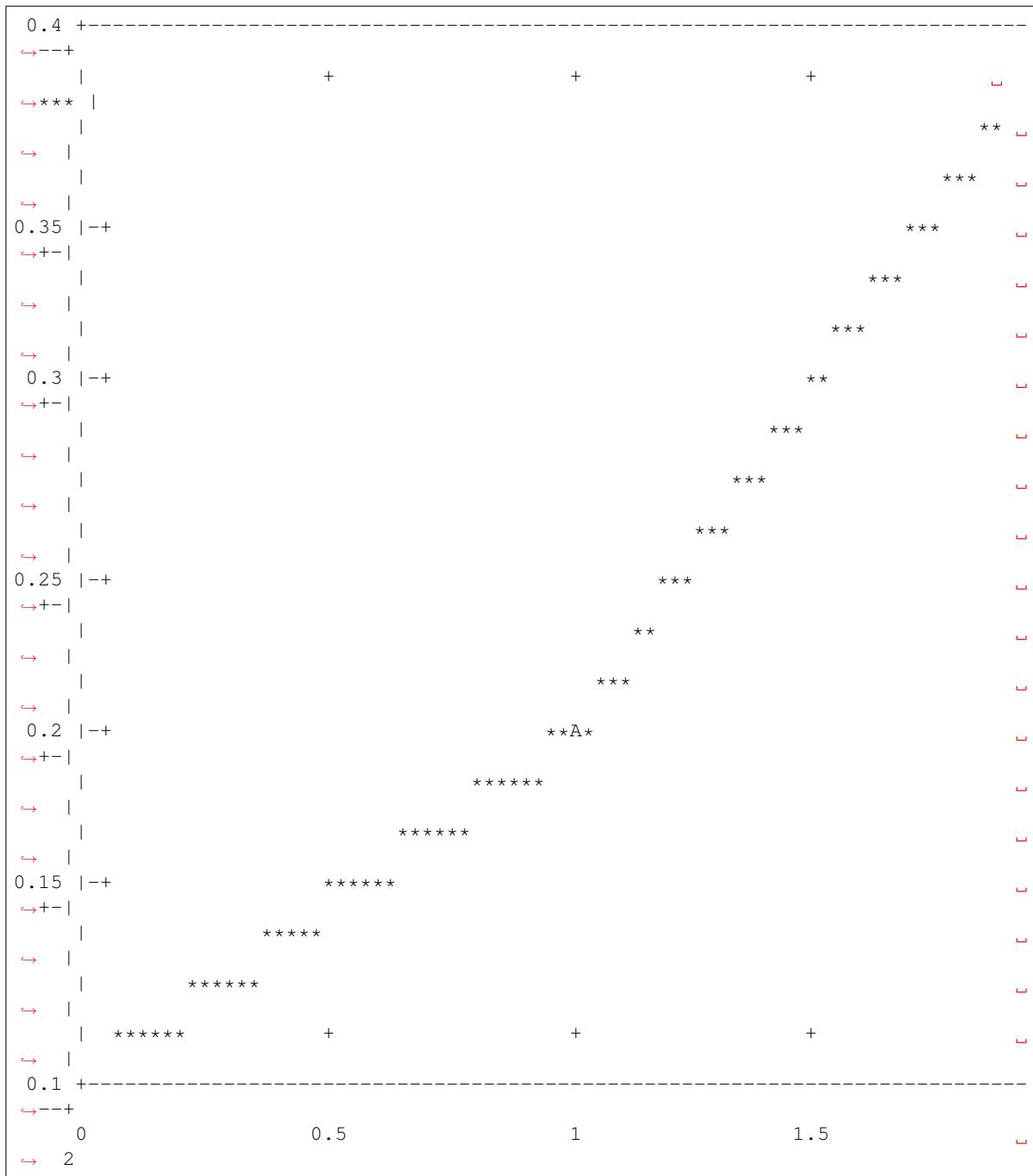

    반환 (State, int, float) – state, turn, reward
```

3.1.4 scripts.utils module

마이크로 체스와 관계없는 기타 코드 모음

```
scripts.utils.ascii_plot (xs, ys, title=None, print_out=False)
gnuplot을 이용해서 ascii 문자로作出선 그래프를 그려줌 GNU plot을 별도로 설치해야 함 - Windows
(C:/Program Files/gnuplot/bin/gnuplot.exe) - Linux (/usr/bin/gnuplot) 설치가 되어있지 않으면 경고 메시지만
출력함
```

```
>>> from scripts.utils import ascii_plot
>>> fig = ascii_plot([0, 1, 2], [0.1, 0.2, 0.4])
>>> print(fig)
```



매개 변수

- **xs** – list of {int, float}, 숫자 리스트
- **ys** – list of {int, float}, 숫자 리스트
- **title** – str, 그래프 위에 표시할 문자열
- **print_out** – bool, True일 때는 반환하는 것과 별개로 그래프를 print 함

반환 str, ascii 문자로 만든 그래프

3.1.5 Module contents

3.2 agents package

3.2.1 Module contents

class agents.**BaseAgent** (*name, color*)

Bases: object

AI의 기반 클래스 AI를 구현할 때는 이 클래스를 상속받아 reset, act, close를 구현해야 함

act (*state*)

AI 초기화

매개 변수 **state** (State) – 현재 게임 상태

반환

다음 수 반환

- chess.Move – AI의 다음수
- None – 다음 수를 반환할 필요가 없는 경우

close ()

AI 종료 마지막에 한 번 실행

opponent_color

상대의 턴

reset ()

AI 초기화 게임 초기에 한 번 실행

3.3 agents.basic package

3.3.1 Submodules

3.3.2 agents.basic.debug_agent module

플랫폼 디버깅 및 테스트 용으로 만들어진 AI 모음

class agents.basic.debug_agent.**FirstMoveAgent** (*name, color*)

Bases: agents.**BaseAgent**

First Move Agent

가능한 수 중에서 가장 첫번째 수를 선택함

- 플랫폼 테스트용

act (*state*)

AI 초기화

매개 변수 **state** (State) – 현재 게임 상태

반환

다음 수 반환

- chess.Move – AI의 다음수

- None – 다음 수를 반환할 필요가 없는 경우

close()
AI 종료 마지막에 한 번 실행

reset()
AI 초기화 게임 초기에 한 번 실행

class agents.basic.debug_agent.**MalfunctionAgent** (*name, color*)
Bases: *agents.BaseAgent*

Malfunction Agent

한 수를 둘 때마다 50%의 확률로 예외를 발생함

- 플랫폼 테스트용
- 예외를 발생시키면 반드시 패배해야함

act (state)
AI 초기화

매개 변수 **state** (*State*) – 현재 게임 상태

반환

다음 수 반환

- chess.Move – AI의 다음수
- None – 다음 수를 반환할 필요가 없는 경우

close()
AI 종료 마지막에 한 번 실행

reset()
AI 초기화 게임 초기에 한 번 실행

3.3.3 agents.basic.human module

인간 플레이어 인터페이스를 작동시키기 위한 더미 AI

class agents.basic.human.**Player** (*name, color*)
Bases: *agents.BaseAgent*

인간 플레이어를 위한 더미 AI

act (state)
AI 초기화

매개 변수 **state** (*State*) – 현재 게임 상태

반환

다음 수 반환

- chess.Move – AI의 다음수
- None – 다음 수를 반환할 필요가 없는 경우

close()
AI 종료 마지막에 한 번 실행

reset()
AI 초기화 게임 초기에 한 번 실행

3.3.4 agents.basic.random_agent module

Random AI 구현

class agents.basic.random_agent.**RandomAgent** (*name, color*)

Bases: *agents.BaseAgent*

Random AI

- 무작위로 행동을 결정하는 예제

act (*state*)

AI 초기화

매개 변수 **state** (*State*) – 현재 게임 상태

반환

다음 수 반환

- chess.Move – AI의 다음수
- None – 다음 수를 반환할 필요가 없는 경우

close()

AI 종료 마지막에 한 번 실행

reset()

AI 초기화 게임 초기에 한 번 실행

3.3.5 Module contents

3.4 agents.search package

3.4.1 Submodules

3.4.2 Module contents

3.4.3 agents.search.one_step_search_agent module

One Step Search AI 구현

class agents.search.one_step_search_agent.**OneStepSearchAgent** (*name, color*)

Bases: *agents.BaseAgent*

One Step Search Agent

- 현재 가능한 모든 수를 시뮬레이션 하여 다음 결과를 예상하고, 가장 좋은 수를 결정하는 AI
- 바로 다음 단계만을 고려하는 가장 단순한 탐색 AI
- 예제에 포함된 가장 약한 탐색 AI

act (*state*)

AI 초기화

매개 변수 **state** (*State*) – 현재 게임 상태

반환

다음 수 반환

- chess.Move – AI의 다음수
- None – 다음 수를 반환할 필요가 없는 경우

close()
AI 종료 마지막에 한 번 실행

reset()
AI 초기화 게임 초기에 한 번 실행

3.4.4 agents.search.two_step_search_agent module

Two Step Search AI 구현

class agents.search.two_step_search_agent.TwoStepSearchAgent (*name, color*)
Bases: *agents.BaseAgent*

Two Step Search Agent

- 나의 다음 수와 상대방의 그 다음 수, 두 수를 시뮬레이션하고 의사결정을 하는 AI
- 게임 승패를 판단하는 것과 동일한 평가함수 사용
- 플랫폼의 다른 부분에서는 greedy AI로 부르기도 함
- 예제에 포함된 두 번째로 약한 텁색기반 AI

act (state)
AI 초기화

매개 변수 **state** (*State*) – 현재 게임 상태

반환

다음 수 반환

- chess.Move – AI의 다음수
- None – 다음 수를 반환할 필요가 없는 경우

close()
AI 종료 마지막에 한 번 실행

opponent_act (state)
상대방의 행동 시뮬레이션 이 상태의 평가는 상대방의 그 다음 행동에 따라 달라짐

매개 변수 **state** (*State*) – 현재 상태

반환 (float) – 평가값 [0, 1] 범위

reset()
AI 초기화 게임 초기에 한 번 실행

3.4.5 agents.search.negamax_search_agent module

Negamax Search AI 구현

class agents.search.negamax_search_agent.NegamaxSearchAgent (*name, color*)
Bases: *agents.BaseAgent*

Negamax Search AI

- Minimax AI의 일종인 Negamax AI 기본 구현

- <https://en.wikipedia.org/wiki/Negamax> 참조
- Negamax가 Minimax에 비해 코드가 간결함
- Two Search AI의 일반적인 형태, Two Search AI와 달리 임의의 깊이를 탐색 가능함
- 탐색 깊이가 깊어질 수록 탐색 공간이 10배씩 증가하기 때문에 최대 탐색 깊이를 적절히 선택해야함

act (state)

AI 초기화

매개 변수 **state** (State) – 현재 게임 상태

반환

다음 수 반환

- chess.Move – AI의 다음수
- None – 다음 수를 반환할 필요가 없는 경우

close()

AI 종료 마지막에 한 번 실행

negamax (state, depth, color)

Negamax 탐색

매개 변수

- **state** (State) – 현재 상태
- **depth** (int) – 남은 탐색 깊이, self.max_depth - 현재 깊이
- **color** (float) – white = 1., black = -1

반환 (chess.Move, float) – best_move, 평가값

- chess.Move: 가장 좋은 행동

- float: 가장 좋은 행동을 했을 때 얻을 수 있는 미래 보상

reset()

AI 초기화 게임 초기에 한 번 실행

3.4.6 agents.search.abp_negamax_search_agent module

Negamax Search AI + alpha-beta pruning 구현

class agents.search.abp_negamax_search_agent.**ABPNegamaxSearchAgent** (*name, color*)
Bases: *agents.BaseAgent*

Negamax Search AI + alpha-beta pruning

- 기본 Negamax 알고리즘의 탐색 성능 향상을 위해 alpha-beta pruning 사용
- alpha-beta pruning 은 불필요한 상태공간을 탐색하지 않기 때문에 같은 시간동안 더 깊은 탐색이 가능함

act (state)

AI 초기화

매개 변수 **state** (State) – 현재 게임 상태

반환

다음 수 반환

- chess.Move – AI의 다음수
- None – 다음 수를 반환할 필요가 없는 경우

```
close()
    AI 종료 마지막에 한 번 실행

negamax(state, depth, alpha, beta, color)

reset()
    AI 초기화 게임 초기에 한 번 실행
```

3.4.7 agents.search.mcts_agent module

Monte-Carlo Tree Search 구현

```
class agents.search.mcts_agent.Arrow
    Bases: agents.search.mcts_agent.Arrow
```

체스 보드 시각화에 사용

```
class agents.search.mcts_agent.MCTSAgent(name, color)
    Bases: agents.BaseAgent
```

MCTS AI

평가용 MCTS AI, 시뮬레이션 횟수를 최대한으로 하고, 탐색시간을 제한함, 주어진 시간을 최대한 활용하여 최선의 성능을 보임,

```
act(state)
    AI 초기화
```

매개 변수 **state** (State) – 현재 게임 상태

반환

다음 수 반환

- chess.Move – AI의 다음수
- None – 다음 수를 반환할 필요가 없는 경우

```
close()
    AI 종료 마지막에 한 번 실행
```

```
max_depth = 6
```

```
n_simulations = 1000000
```

```
planner = None
```

```
reset()
    AI 초기화 게임 초기에 한 번 실행
```

```
search_time = 10
```

```
class agents.search.mcts_agent.MCTSAgentDev(name, color)
    Bases: agents.BaseAgent
```

개발용 MCTS AI

시뮬레이션 횟수를 고정하고, 탐색 시간은 매우 크게함, PC 사양에 관계없이 균일한 성능을 보임

```
act(state)
    AI 초기화
```

매개 변수 `state` (`State`) – 현재 게임 상태
반환

- 다음 수 반환
 - `chess.Move` – AI의 다음수
 - `None` – 다음 수를 반환할 필요가 없는 경우

close()
 AI 종료 마지막에 한 번 실행

max_depth = 6

n_simulations = 5000

planner = None

reset()
 AI 초기화 게임 초기에 한 번 실행

search_time = 10000

class `agents.search.mcts_agent.MCTSPlanner` (`color`, `max_depth=9223372036854775807`,
`action_score='visits'`, `eval_func=<function Evaluator.eval_2>`, `exploration_coef=1.0`,
`reward_amplify=True`, `debug=False`)

Bases: `object`

MCTS 알고리즘 구현

backup(node, delta)
 시뮬레이션 결과 업데이트 턴마다 Negamax 스타일로 보상의 부호를 바꿈, 보상구간 [-1, 1]을 가정함

매개 변수

- `node` (`agents.search.macts_agent.Node`) – 탐색한 트리의 종단노드
- `delta` (`float`) – 시뮬레이션으로 알아낸 보상

best_child(node)
 현재 노드에서 가장 좋은 행동을 결정하여 반환함

매개 변수 `node` (`agents.search.macts_agent.Node`) – 현재 상태 노드 v0
반환 (`chess.Move`) – 가장 좋은 행동

default_policy(state)
 Monte Carlo 시뮬레이션

매개 변수 `state` (`State`) – 탐색한 트리의 종단노드에 저장되어 있는 state
반환 (`float`) – 시뮬레이션 결과 얻은 최종 보상

search(state, n_simulations=1000, timeout=10)
 주어진 상태와 제약조건(시뮬레이션 횟수와 시간제한)동안 시뮬레이션을 하고 찾아낸 가장 좋은 수을 반환함

매개 변수

- `state` (`State`) – 현재 게임 상태
- `n_simulations` (`int`) – 최대 시뮬레이션 횟수
- `timeout` (`int`) – 시간제한 (sec.)

반환 (`chess.Move`) – 가장 좋은 수

tree_policy (node)
선택 및 확장 단계

매개 변수 **node** (`agents.search.macts_agent.Node`) – 현재 노드 v0
반환 (Node) – 생성한 트리의 종단노드 v1

class `agents.search.mcts_agent.Node (turn, state)`
Bases: `object`

상태공간 노드

Attrs:

- (bool) – color, white=True, black=False
- (`scripts.run_game.State`) – state
- (`agents.search.macts_agent.Node`) – parent 이전 상태를 저장하고 있는 노드
- (dict) – children, dict(key-> chess.Move, value-> Node), 특정 수를 두고 난 뒤의 상태
- (int) – visits, 이 노드를 방문한 횟수
- (float) – wins, 이 노드와 자식 노드에서 받은 보상의 총합
- (float) – ucb 이 노드의 ucb 값

children

key

next_turn

parent

state

turn

ucb

visits

wins

3.5 agents.self_learning package

3.5.1 Submodules

3.5.2 Module contents

OBSERVATION_SHAPE:

12 channels: 말 종류마다 하나의 채널을 사용 함(P, R, N, B, Q, K, p, r, n, b, q, k) 백은 대문자, 흑은 소문자로 표시함

5 rows: 세로

4 columns: 가로

특정 위치에 기물이 위치하면 1., 기물이 없으면 0.으로 표시

LEN_STATE 지금이 백의 차례인가?, 흑의 차례인가?, 현재 플레이어에게 castling 권리가 있는가?, 다음 플레이어에게 castling 권리가 있는가? 정보를 1., 과 0.로 표시

N_ACTIONS 가능한 모든 행동의 개수, 실제 기물의 존재 여부와 관계없이 가능한 모든 출발 지점 20개와 목표 지점 20개를 곱한 400개를 가능한 모든 행동으로 함

3.5.3 agents.self_learning.agent module

Self Learning AI

```
class agents.self_learning.agent (name, color)
    Bases: agents.BaseAgent
```

Self Learning AI

초기 속성은 eval 모드에서 사용할 값으로 설정되어 있음

```
act (state, include_pi=False, tau=0)
```

매개 변수

- **state** (scripts.run_game.State) – 현재 게임 상태
- **include_pi** (bool) – 다음 수와 pi를 같이 반환할지 여부 True일 때는 pi를 같이 반환함
- **tau** (int) – 몇 번째 수 까지 탐색을 할 것인지 [0, n] 0 일 때는 탐색을 하지 않음 (평가, 테스트) 0 이상일 때는 정해진 확률에 따라 무작위로 다음 수를 선택함 (학습)

반환

```
(int, np.array[1, 400]) – move, pi
```

- move: [0, 399] 사이의 정수, 탐색을 하지 않는다면, 가장 pi값이 높은 행동을 선택함
- pi: 1 x 400 실수 행렬, 모든 행동에 대한 MCTS의 탐색 빈도를 정규화

```
alpha = 1.0
```

```
close()
```

AI 종료 마지막에 한 번 실행

```
cputc = 1.0
```

```
epsilon = -1.0
```

```
max_depth = 32
```

```
mirror = True
```

```
model = None
```

```
model_cls
```

alias of `agents.self_learning.models.C64r6L1024r`

```
model_file = 'models/C64r6L1024r/v1/best.pt'
```

```
n_simulations = 100000
```

```
planner = None
```

```
reset (load_model=True, visualize_tree=False)
```

매개 변수

- **load_model** (bool) – 학습한 모델을 로드할지 여부, 학습할 때는 False로 하고, 평가할 때는 기본 값 True 사용함
- **visualize_tree** (bool) – MCTS 시각화 기능을 사용하는지 여부, 학습할 때는 False, 평가할 때는 True

set_params (*model*, *max_depth*, *n_simulations*, *cputc*, *epsilon*, *alpha*, *mirror*)

AI의 초기화 작업은 reset에서 실행하지만, 편의 목적으로 일부 값은 set_params에서 설정함 set_params는 reset 이전에 호출되어야 함

매개 변수

- **model** (*torch.nn*) – 인공신경망(policy + value net)
- **max_depth** (*int*) – MCTS 최대 탐색 깊이
- **n_simulations** (*int*) – MCTS 시뮬레이션 횟수
- **cputc** (*float*) – MCTS, UCB 공식에서 U의 가중치 c (cU+Q)S
- **epsilon** (*float*) – MCTS, UCB 공식에서 탐색 가중치
- **alpha** (*float*) – MCTS, UCB 공식에서 활용 가중치
- **mirror** (*bool*) – True일 때는 학습 AI가 흑과 백의 데이터를 모두 학습에 사용함 현재 자신과 다른 색깔일 때는 판을 180도 회전하여 학습 데이터로 사용

```
class agents.self_learning.agent.Sample (observation, state, move, pi, reward, done, legal_moves)
```

Bases: object

데이터 샘플 저장 용

done**legal_moves****move****observation****pi****reward****state**

```
agents.self_learning.agent.play_game (inqueue, outqueue, seed)
```

종료되지 않고 무한 루프로 실행, inqueue에서 다음 인자를 받아서, outqueue로 반환

매개 변수

- **inqueue** –
 - (Agent) – white, white AI
 - (Agent) – black, black AI
 - (float) – turns_until_tau0, 탐색 횟수 (tau)
 - (int) – max_turn, 최대 턴 (default: 80)
 - (bool) – mirror
 - (bool) – reversed_reward, 학습 AI가 black 일 때 True 설정
- **outqueue** –
 - (float) – reward
 - (bool) – turn
 - (list) – short_term_memory
- **seed** (*int*) – worker 를 위한 seed 값으로 초기화, worker는 모두 다른 seed 값으로 초기화 해야 함

```
agents.self_learning.agent.train()  
모델 학습 함수
```

3.5.4 agents.self_learning.models module

Self Learning AI 가 사용할 수 있는 인공신경망

```
class agents.self_learning.models.C64r6L1024r(observation_shape, len_state, n_actions)  
Bases: torch.nn.modules.module.Module
```

CNN 6층 + MLP 1층 구조

```
forward(observations, states)
```

Defines the computation performed at every call.

Should be overriden by all subclasses.

주석: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

```
class agents.self_learning.models.L1024r(observation_shape, len_state, n_actions)  
Bases: torch.nn.modules.module.Module
```

1024개의 은닉노드를 가진 간단한 MLP

```
forward(observations, states)
```

Defines the computation performed at every call.

Should be overriden by all subclasses.

주석: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

```
agents.self_learning.models.weights_init(m)
```

3.5.5 agents.self_learning.memory module

Replay Memory

```
class agents.self_learning.memory.ReplayMemory(capacity, memory_path)  
Bases: object
```

게임 플레이 데이터를 저장하는 용도

```
add_samples(samples)
```

```
capacity
```

```
get_dataset(train_set_size, validation_set_size)
```

학습 데이터와, 검증 데이터의 개수를 입력함

매개 변수

- **train_set_size** (int) – 학습 데이터 개수
- **validation_set_size** (int) – 검증 데이터 개수

반환 (torch.Tensor, torch.Tensor) – train_set, validation_set - train_set: 학습 데이터 세트 - validation_set: 검증 데이터 세트

pos

size

3.5.6 agents.self_learning.mcts module

Self Learning용 Monte Carlo Tree Search 구현

```
class agents.self_learning.mcts.MCTSPlanner(turn, max_depth=9223372036854775807,
                                             action_score='visits', cputc=1.0, epsilon=0.2, alpha=0.8, model=None, mirror=True)
```

Bases: object

MCTS 알고리즘 구현

backup (node, delta)

시뮬레이션 결과 업데이트 턴마다 Negamax 스타일로 보상의 부호를 바꿈, 보상구간 [-1, 1]을 가정함

매개 변수

- **node** (agents.self_learning.mcts.Node) – 탐색한 트리의 종단노드
- **delta** (float) – 시뮬레이션으로 알아낸 보상

best_action (node, deterministic)

현재 노드에서 가장 좋은 행동을 결정하여 반환함

매개 변수

- **node** (agents.self_learning.mcts.Node) – 현재 상태 노드 v0
- **deterministic** (bool) – 결정적으로 행동을 선택(True)할 것인지 확률적으로 선택 (False)할 것인지 여부

반환 (chess.Move) – 가장 좋은 행동

pi (node, tau)

현재 상태 노드에서 pi 값을 계산함

매개 변수

- **node** (agents.self_learning.mcts.Node) –
- **tau** (int) –

반환 (torch.Tensor[400]) – 현재 노드에서 시뮬레이션했던 모든 행동들을 확률 값으로 변환함

predict (board_state)

인공신경망을 이용해 현재 상태에서 가능한 수들의 선택 확률과 가치를 예측함

매개 변수 board_state (State) –

반환 (torch.Tensor, torch.Tensor) – move_prob, value - move_prob: torch.Tensor, 가능한 수들의 선택 확률 - value: torch.Tensor, 현재 상태의 가치

search (state, n_simulations=1000, tau=0.0, timeout=10)

주어진 상태와 제약조건(시뮬레이션 횟수와 시간제한)동안 시뮬레이션을 하고 찾아낸 가장 좋은 수를 반환함

매개 변수

- **state** (`State`) – 현재 게임 상태
- **n_simulations** (`int`) – 최대 시뮬레이션 횟수
- **tau** (`int`) –
- **timeout** (`int`) – 시간제한 (sec.)

반환 (`chess.Move`) – 가장 좋은 수

tree_policy (`node`)
선택 및 확장 단계

 매개 변수 **node** (`agents.self_learning.mcts.Node`) – 현재 노드 v0

 반환 (`Node`) – 생성한 트리의 종단노드 v1

class `agents.self_learning.mcts.Node` (`turn, state`)
Bases: `object`

상태공간 노드

Attrs:

- (`bool`) – turn, white=True, black=False
- (`scripts.run_game.State`) – state
- (`agents.self_learning.mcts.Node`) – parent, 이전 상태를 저장하고 있는 노드
- (`dict`) – children, dict(key-> `chess.Move`, value-> `Node`), 특정 수를 두고 난 뒤의 상태
- (`int`) – visits, 이 노드를 방문한 횟수
- (`float`) – wins, 이 노드와 자식 노드에서 받은 보상의 총합
- (`float`) – ucb, 이 노드의 ucb 값

children

key

next_turn

parent

print_tree (`move=None, depth=0`)
 콘솔에 트리 출력

prob

state

turn

ucb

visits

wins

3.5.7 `agents.self_learning.utils` module

Self Learning AI에 필요한 기타 파일

```
class agents.self_learning.utils.Buffer(args)
Bases: object

학습에 필요한 메모리 공간을 미리 할당해두고 재사용하기 위해 사용

class agents.self_learning.utils.Operators
Bases: object

학습 도중 실험 조건을 변경하기 위해 사용

static lr(log, current_model, best_model, arguments=[])
static n_simulations(log, current_model, best_model, arguments=[])

class agents.self_learning.utils.Record(args)
Bases: object
```

학습 동안에 저장되는 데이터를 모아두는 객체

```
agents.self_learning.utils.decode_move(board, move_code, turn, mirror)
[0, 399] 정수로 된 수를 chess.Move로 변환, encode_move의 반대
```

매개 변수

- **board** (State) –
- **move_code** (chess.Move) –
- **turn** (bool) –
- **mirror** (bool) –

반환 (chess.Move) –

```
agents.self_learning.utils.encode_move(move, turn, mirror)
chess.Move를 [0, 399] 정수로 변환, decode_move의 반대
```

매개 변수

- **move** (chess.Move) –
- **turn** (bool) – white=True, black=False
- **mirror** (bool) –

반환 (int) – 정수 [0, 399] 범위

```
agents.self_learning.utils.encode_observation(state, turn, mirror)
```

fen 표기법으로 된 보드 상태를 입력받아서, numpy array 형태로 변환 각 채널은 특정 말에 대응되며, 해당 말이 존재하는 위치에 1로 표시 나머지 공간은 0.

매개 변수

- **state** (str) – fen 표기법
- **turn** (bool) – white=True, black=False
- **mirror** (bool) – 미러 옵션 사용여부

반환 (torch.Tensor[6x8x8]) – 게임 상태를 tensor로 반환

```
agents.self_learning.utils.get_logger()
로거 반환
```

반환 (logging.logger) –

```
agents.self_learning.utils.is_castling(state, turn, mirror=True)
castling 가능 여부 검사하여 실수 list로 반환, 게임 상태로 사용
```

매개 변수

- **state** (`State`) –
- **turn** (`bool`) –
- **mirror** (`bool`) –

반환 ([float, float]) – 길이 2

`agents.self_learning.utils.make_piece_index()`
알파벳으로 표시된 체스 기물을 정수로 변환하는 dict를 반환
반환 (dict) –

3.6 agents.stockfish package

3.6.1 Submodules

3.6.2 Module contents

3.6.3 agents.stockfish.agent module

Stockfish Wrapper

`class agents.stockfish.agent.Stockfish(name, color)`
Bases: `agents.BaseAgent`

UCI 인터페이스를 이용해 Stockfish를 래핑함, 실제 구현은 stockfish_8_*.exe 입

Microchees에 맞게 Stockfish의 기능을 제한함

- 탐색범위 제한: 5x4 범위 이외의 공간을 탐색하지 못하게 함
- castling 제한: castling 위치가 다르기 때문에 Stockfish에서는 에러 발생

TODO: 간헐적으로 에러가 발생함, 원인은 아직 불명

`act(state)`
AI 초기화

매개 변수 `state` (`State`) – 현재 게임 상태

반환

다음 수 반환

- `chess.Move` – AI의 다음수
- `None` – 다음 수를 반환할 필요가 없는 경우

`available_squares = [32, 33, 34, 35, 24, 25, 26, 27, 16, 17, 18, 19, 8, 9, 10, 11, 0,`

`close()`

AI 종료 마지막에 한 번 실행

`engine = None`

`engine_path = 'stockfish'`

`level = 20`

`reset()`

AI 초기화 게임 초기에 한 번 실행

- genindex
- modindex
- search

Python 모듈 목록

a

agents, 44
agents.basic, 46
agents.basic.debug_agent, 44
agents.basic.human, 45
agents.basic.random_agent, 46
agents.search, 46
agents.search.abp_negamax_search_agent,
 48
agents.search.mcts_agent, 49
agents.search.negamax_search_agent, 47
agents.search.one_step_search_agent, 46
agents.search.two_step_search_agent, 47
agents.self_learning, 51
agents.self_learning.agent, 52
agents.self_learning.mcts, 55
agents.self_learning.memory, 54
agents.self_learning.models, 54
agents.self_learning.utils, 56
agents.stockfish, 58
agents.stockfish.agent, 58

s

scripts, 44
scripts.chess_board, 39
scripts.run_game, 40
scripts.utils, 42