기초인공지능 과제 2번 보고서

20181650 안도현

1. Minimax agent

```python
class MinimaxAgent(AdversialSearchAgent):
    """
    [문제 01] MiniMaxAgent의 Action을 구현하시오.
    (depth와 evaluation function은 위에서 정의한 self.depth and self.evaluationFunction을 사용할 것.)
    """
    def Action(self, gameState):
        ######################## Write Your Code Here ################################
        return self.minimax_agent_evaluationFunc(gameState,0,1,0)

    def minimax_agent_evaluationFunc(self, currentGameState, agent, maxnode, depth):
        if maxnode:
            retval = -1_000_000_000
            tmp = retval
        else:
            retval = 1_000_000_000
        if currentGameState.isWin() or currentGameState.isLose() or depth == self.depth:
            return self.evaluationFunction(currentGameState)
        for i, newaction in enumerate(currentGameState.getLegalActions(agent)):
            successorGameState = currentGameState.generateSuccessor(agent, newaction)

            if maxnode:
                scores = self.minimax_agent_evaluationFunc(successorGameState, 1, 1 - maxnode, depth)
                if depth==0:
                    if tmp < scores:
                        tmp = scores
                        retval = newaction
                else:
                    retval=max(retval,scores)

            elif agent < currentGameState.getNumAgents()-1:
                scores = self.minimax_agent_evaluationFunc(successorGameState, agent + 1, maxnode, depth)
                retval=min(retval,scores)
            else:
                scores = self.minimax_agent_evaluationFunc(successorGameState, 0, 1 - maxnode, depth + 1)
                retval=min(retval,scores)
        return retval
```

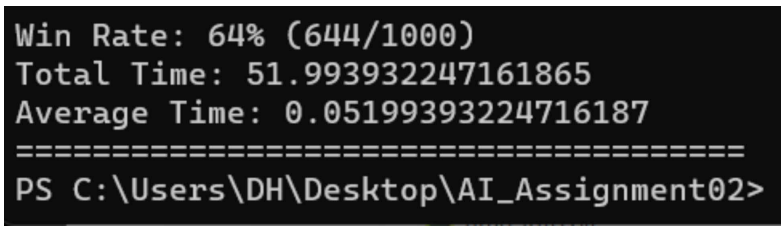
-minimax_agent_evaluationFunc()을 구현하여 사용했다. 강의자료의 pseudocode를 많이 참고했지만, 다른 점은 기본적으로 재귀에서 리턴하는 값이 오직 score값이고 depth가 0인 maxnode에서만 움직여야 하는 node를 리턴하도록 만들었다.

-현재가 max node일 경우 retval을 매우 작은 값, min node일 경우 매우 큰 값으로 설정하고 시작한다. 터미널 노드일 경우 미리 정의되어 있던 evaluationFunction()을 통해 유틸리티 값을 얻어 리턴한다. 그게 아니라면 현재 노드에서 움직일 수 있는 모든 노드에 대해 루프를 도는데, 여기서 그 노드들에 대해 state를 먼저 생성한다. max node라면 재귀 호출 후 score를 받게 되는데 최종적으로 그 중 최댓값을 리턴하고, min node라면 반대로 최솟값을 리턴한다. 위에서 언급했듯 특이한 depth가 0인 max

node에서만 score가 아닌 action을 리턴하도록 했다.

– 재귀 호출의 인자가 minimax의 특징을 잘 나타낸다. max node라면 agent가 0이므로 호출 시 agent 인자에 1을 주고, maxnode에 1–maxnode로서 0을 집어넣으며, depth를 그대로 유지한다. min node중 마지막 agent가 아니라면 agent 인자에 agent+1을 주고, max node 여부를 그대로 유지하며, depth도 그대로 유지한다. 마지막 agent인 min node는 agent에 0을 주어 다시 maxnode로 돌아가게 하고, maxnode도 set하며, depth를 증가시킨다.

요구 캡쳐 화면)



64%의 확률이 나온다. 정의되어 있던 evaluation함수의 영향인지 실행할 때 마다 50~70%의 범위 내에서 승률이 바뀌어가며 출력되었다.

2. AlphaBeta Agent

```
def alphabeta_agent_evaluationFunc(self, currentGameState, agent, maxnode, depth, alpha, beta):
    if maxnode:
        retval = -1_000_000_000
        tmp = retval
    else:
        retval = 1_000_000_000
    if currentGameState.isWin() or currentGameState.isLose() or depth == self.depth:
        return self.evaluationFunction(currentGameState)
    for i, newaction in enumerate(currentGameState.getLegalActions(agent)):
        successorGameState = currentGameState.generateSuccessor(agent, newaction)

        if maxnode:
            scores = self.alphabeta_agent_evaluationFunc(successorGameState, 1, 1 - maxnode, depth, alpha, beta)
            if depth == 0:
                if tmp < scores:
                    tmp = scores
                    retval = newaction
                    alpha = scores
            else:
                retval = max(retval, scores)
                alpha = retval

            if scores >= beta:
                break

        elif agent < currentGameState.getNumAgents() - 1:
            scores = self.alphabeta_agent_evaluationFunc(successorGameState, agent + 1, maxnode, depth, alpha, beta)
            retval = min(retval, scores)
            beta = retval
            if scores <= alpha:
                break
        else:
            scores = self.alphabeta_agent_evaluationFunc(successorGameState, 0, 1 - maxnode, depth + 1, alpha, beta)
            retval = min(retval, scores)
            beta = retval
            if scores <= alpha:
                break
    return retval
```

-alphabeta agent는 minimax에서 속도 향상을 위해 pruning을 도입한 것이다. 강의자료의 수도코드처럼 재귀 호출되는 함수에 alpha, beta 인자를 추가한다. 그리고 maxnode의 statement에서 scores가 beta보다 크면 break를 걸어주는데, 이로써 for문에서 빠져나간 뒤 직전에 저장했던 value를 바로 리턴한다. 어차피 부모 min node에서는 이 자식 max node를 채택할 일이 없기 때문에 값은 중요하지가 않기 때문이다. 마찬가지로 min node statement에서는 scores가 alpha보다 작으면 바로 break해서 저장해둔 retval을 바로 리턴해버린다. 부모 max node에서는 alpha값보다 작은 값을 채택하지 않기 때문이다.

-alpha와 beta값은 인자로 계속 전달되어 재귀 과정에서도 부모 노드의 alpha 값이나 beta 값을 적절히 받아와 비교할 수 있도록 했다.

요구 캡처 화면)

```
Win Rate: 15% (45/300)
Total Time: 253.76851606369019
Average Time: 0.845895053545634
======================================
----- END  MiniMax (depth=3) For Medium Map
```

```
Win Rate: 17% (53/300)
Total Time: 121.57190012931824
Average Time: 0.40523966709772746
======================================
----- END  AlphaBeta (depth=3) For Medium Map
```

minimax와 alphabeta는 medium map에 대해 비슷한 승률을 보이지만, pruning을 도입한 alphabeta쪽이 거의 절반의 수행 시간을 보이는 것을 확인할 수 있다.

```
Win Rate: 39% (395/1000)
Total Time: 32.953768253326416
Average Time: 0.032953768253326416
======================================
----- END  MiniMax (depth=4) For Minimax Map
```

```
Win Rate: 35% (359/1000)
Total Time: 16.143091917037964
Average Time: 0.016143091917037965
======================================
----- END  AlphaBeta (depth=4) For Minimax Map
```

minimax map에 대해서도 역시 비슷한 결과로 alphabeta가 절반에 가까운 수행 시간을 가지고 있다.

3. Expectimax Agent

```
def expectimax_agent_evaluationFunc(self, currentGameState, agent, maxnode, depth):
    if maxnode:
        retval = 0
        tmp = -1_000_000_000
    else:
        retval = 0
    if currentGameState.isWin() or currentGameState.isLose() or depth == self.depth:
        return self.evaluationFunction(currentGameState)
    p = 1/len(currentGameState.getLegalActions(agent))
    for i, newaction in enumerate(currentGameState.getLegalActions(agent)):
        successorGameState = currentGameState.generateSuccessor(agent, newaction)

        if maxnode:
            scores = self.expectimax_agent_evaluationFunc(successorGameState, 1, 1 - maxnode, depth)
            if depth==0:
                if tmp < scores:
                    tmp = scores
                    retval = newaction
            else:
                retval+=p*scores

        elif agent < currentGameState.getNumAgents()-1:
            scores = self.expectimax_agent_evaluationFunc(successorGameState, agent + 1, maxnode, depth)
            retval+=p*scores
        else:
            scores = self.expectimax_agent_evaluationFunc(successorGameState, 0, 1 - maxnode, depth + 1)
            retval+=p*scores
    return retval
```

–다음은 상대가 항상 optimal 한 결정만을 하지 않을 때 유용한 expectimax agent이다. 이는 각 자식 노드에 대해 확률과 value를 곱한 것을 모두 더한 값을 사용하는 것이다. 이를 위해 p값을 도입해 1을 자식 노드의 개수로 나눈 값으로 정했다. 그리고 for문 내에서 p*scores값을 retval에 쌓고 리턴함으로써 각 노드의 value를 공평하게 고려하여 부모 노드의 value를 주었다. 그리고 depth가 0인 max node에서 자식 중 최대 scores를 내는 값을 받아 minimax agent때와 같이 그 위치를 리턴해줌으로써 코드를 완성했다.

요구 캡처 화면)

```
-----------Game Results--------------
Average Score: -5.68
Score Results: -502, 532, -502, 532, -502, 532, -502, 532, 532, 532, -502, -502, 532, -502, -502, -502, -502, -502, -502
, 532, -502, 532, 532, 532, 532, 532, -502, -502, 532, 532, -502, 532, -502, 532, 532, 532, 532, 532, -502, -502, 532, 5
32, -502, 532, -502, -502, -502, -502, -502, -502, -502, 532, -502, -502, -502, 532, 532, -502, -502, -502, -502, -502,
-502, 532, 532, -502, 532, -502, -502, 532, 532, -502, 532, -502, 532, 532, 532, -502, 532, -502, 532, 532, 532, -502, -
502, -502, -502, -502, 532, 532, -502, 532, 532, -502, 532, 532, -502, 532, 532, -502, -502
Record: Lose, Win, Lose, Win, Lose, Win, Lose, Win, Win, Win, Lose, Lose, Win, Lose, Lose, Lose, Lose, Lose, Lose, Win,
Lose, Win, Win, Win, Win, Win, Lose, Lose, Win, Win, Lose, Win, Lose, Win, Win, Win, Win, Win, Lose, Lose, Win, Win, Los
e, Win, Lose, Lose, Lose, Lose, Lose, Lose, Lose, Win, Lose, Lose, Lose, Win, Win, Lose, Lose, Lose, Lose, Lose, Lose, W
in, Win, Lose, Win, Lose, Lose, Win, Win, Lose, Win, Lose, Win, Win, Win, Lose, Win, Lose, Win, Win, Win, Lose, Lose, Lo
se, Lose, Win, Win, Lose, Win, Win, Lose, Win, Win, Lose, Win, Win, Lose, Lose

Win Rate: 48% (48/100)
Total Time: 0.3102457523345947
Average Time: 0.003102457523345947
====================================
PS C:\Users\DH\Desktop\AI_Assignment02>
```