과목 명: 기초인공지능

<<Assignment02 : Pacman>>

서강대학교 컴퓨터학과

[20171600]

[강주언]

목 차

1. 프로그램 설명

Adversarial Search 를 활용한 Pacman 구현을 하였다.

초기 배포된 hw02.py 내부에 구현되어 있는 ReflexAgent 함수를 참고하여, 3 가지 Algorithm 을 구현하였다.

1. Minimax Algorithm

   최소극대화 알고리즘(Minimax algorithm)은 결정이론, 게임이론, 통계학, 철학에서 사용하는 개념으로 최악의 경우 발생 가능한 손실을 최소화 한다는 규칙이다. 손실이 아니라 이익이 기준이라면 최소 이익을 극대화 한다는 의미에서 maximin 이라고 부르기도 한다. Complete 하고 Optimal 하지만, 시간복잡도가 exponential 하기에, 문제가 된다.

2. Alpha-Beta Pruning Algorithm

   알파-베타 가지치기(Alpha-Beta Pruning Algorithm)는 탐색 트리 (Tree Seach) 에서 최소극대화 (Minimax) 알고리즘을 적용할 때 평가(evaluate)하는 노드의 수를 줄이기 위한 알고리즘이다. 이 알고리즘은 적대탐색 알고리즘 이라고도 한다. 이 알고리즘은 이전에 평가한 노드 보다 현재 평가하는 노드가 더 좋지 않을 가능성이 있으면 평가를 중단한다. 이 노드의 남은 형제(sibling) 노드와 모든 후손 노드는 가지치기 되어 평가하지 않는다. 이 알고리즘을 일반적인 Minimax 에 적용하면 동일한 결과를 얻게 된다. 최종 결정에 영향을 미치지 않는 가지들을 쳐낼 뿐이기 때문이다.

3. Expectimax Algorithmt

   예상극대화 알고리즘(Expectimax Algorithm)은 Minimax 알고리즘의 변형으로, 결과가 플레이어의 기술과 주사위 굴림과 같은 찬스 요소의 조합에 따라 달라진다. 통상적인 Minimax 트리의 최소 최대 노드 이외에도, 확률 노드가 있다. 확률 노드는 임의 이벤트 발생의 예상값을 취한다.

2. 코드 설명

2.1 Minimax Agent

코드는 아래와 같다.

```
class MinimaxAgent(AdversialSearchAgent):
  """
    [문제 01] MiniMax의 Action을 구현하시오. (20점)
    (depth와 evaluation function은 위에서 정의한 self.depth and self.evaluationFunction을 사용할 것.)
  """
  def Action(self, gameState):
    ###################### Write Your Code Here ################################

    move_candidate = gameState.getLegalActions()

    list_nextstates = [gameState.generateSuccessor(0,action) for action in move_candidate]
    n_agents = gameState.getNumAgents()
    scores = [self.MinimaxValue(n_agents,1,nextstate,(n_agents * self.depth)-1) for nextstate in list_nextstates]

    bestScore = max(scores)
    Index = [index for index in range(len(scores)) if scores[index] == bestScore]

    get_index = random.choice(Index)

    return move_candidate[get_index]

  def MinimaxValue(self, n_agents, i_agent, gameState, depth):

    if (depth == 0 or gameState.isLose() or gameState.isWin()): # terminal tet
      return self.evaluationFunction(gameState)

    depth -= 1
    move_candidate = gameState.getLegalActions(i_agent)
    list_nextstates = [gameState.generateSuccessor(i_agent,action) for action in move_candidate]

    if (i_agent != 0): # Ghost
      val = min([self.MinimaxValue(n_agents, (i_agent+1)%n_agents, nextstate, depth) for nextstate in list_nextstates])

    else: # Pacman
      val = max([self.MinimaxValue(n_agents, 1%n_agents, nextstate, depth) for nextstate in list_nextstates])

    return val
```

코드의 앞부분은, 미리 구현되어 있는 ReflexAgent code 에서 gameState.getNumAgents() 를 추가하여 구현하였다. 우선, 이동 가능한 Candidate 들을 추출한 뒤에, 해당 Candidate 들로부터 다음 상태를 gameState.generateSuccessor 함수로 구한 뒤에, 그 상태의 점수들을 리스트로 만든 뒤에 가장 큰 값을 max 함수로 찾아주었다. 동률일 경우에는 랜덤 하게 return 해 주었다.
이 앞부분의 코드는 세가지 agent 가 거의 유사하다. 다른점이 있다면, 각각의 evaluation 함수일 것이다. 따라서, 앞으로 evaluation 부분만 설명 하겠다.

MinimaxAgent 의 evaluation 함수는 MinimaxValue 이다. Terminal test 를 한 뒤에 만약 terminal 이 아니라면 본격적으로 evaluate 한다. 일단 depth 를 하나 낮춰준다. i_agent 는 agent 의 종류를 구별하기 위함인데, 0 이외의 숫자는 전부 Ghost 이다. getLegalActions 에서 Pacman 인 경우엔 PacmanRule 에 따라서 return 을 해주고, Ghost 인 경우엔 GhostRule 에 따라서 LegalAction 을 return 해준다. 그 이후, i_agent 에 따라 연산을 다르게 진행하는데, Ghost 는 min 으로 recursion 을 하고, 가장 마지막이 Pacman 의 연산인데 Pacman 만이 max 로 최종 return 을 해준다. 이것이 minimax evaluation 의 핵심이다.

2.2 AlphaBeta Agent

코드는 아래와 같다.

```
class AlphaBetaAgent(AdversialSearchAgent):
  """
    [문제 02] AlphaBeta의 Action을 구현하시오. (25점)
    (depth와 evaluation function은 위에서 정의한 self.depth and self.evaluationFunction을 사용할 것.)
  """
  def Action(self, gameState):
    ###################### Write Your Code Here ############################

    move_candidate = gameState.getLegalActions()

    list_nextstates = [gameState.generateSuccessor(0,action) for action in move_candidate]
    n_agents = gameState.getNumAgents()
    scores = [self.AlphaBetaValue(n_agents,1,nextstate, (n_agents * self.depth)-1, MIN_, MAX_) for nextstate in list_nextstates]

    bestScore = max(scores)
    Index = [index for index in range(len(scores)) if scores[index] == bestScore]

    get_index = random.choice(Index)

    return move_candidate[get_index]

  def AlphaBetaValue(self, n_agents, i_agent, gameState, depth, alpha, beta):

    if (depth==0 or gameState.isLose() or gameState.isWin()):  # terminal tet
      return self.evaluationFunction(gameState)

    depth -= 1
    move_candidate=gameState.getLegalActions(i_agent)
    list_nextstates=[gameState.generateSuccessor(i_agent,action) for action in move_candidate]
    if (i_agent != 0): # Ghost
      val = MAX_
      for nextstate in list_nextstates:
        val = min(self.AlphaBetaValue(n_agents, (i_agent+1)%n_agents, nextstate, depth, alpha, beta), val)
        if (val < alpha):
          return val
        beta = min(beta, val)

    else: # Pacman
      val = MIN_
      for nextstate in list_nextstates:
        val = max(self.AlphaBetaValue(n_agents, 1%n_agents, nextstate, depth, alpha, beta), val)
        if (val > beta):
          return val
        alpha = max(alpha, val)

    return val
```

코드의 앞부분은 앞서 Minimax 에서 말한 부분과 거의 유사하다. 다만 Evaluation Function call 만 다를 뿐이다.
AlphaBetaAgent 의 evaluation 함수는 AlphaBetaValue 이다. Terminal test 를 한 뒤에 만약 terminal 이 아니라면 본격적으로 evaluate 한다. 기본적인 구조는 Minimax 와 거의 동일하다. 다만 개념 부분에서 설명했듯이, 탐색이 필요없는 노드를 search 하지 않기 위해서 약간의 장치만 추가해 준 것이다. i_agent 가 0 이 아닐 때 즉, Ghost 일 때는, 해당 value(점수)가 alpha 값보다 작다면 굳이 더 탐색하지 않고 바로 그 값을 return 해준다. 즉 pruning 해주는 것이다. 반대로 i_agent 가 0 일 때 즉, Pacman 일 때에는, 해당 value(점수)가 beta 값보다 크다면 pruning 해준다. 이것이 Alpha-Beta pruning algorithm 의 핵심이다.

2.3 Expectimax Agent

코드는 아래와 같다.

```
class ExpectimaxAgent(AdversialSearchAgent):
  """
    [문제 03] Expectimax의 Action을 구현하시오. (25점)
    (depth와 evaluation function은 위에서 정의한 self.depth and self.evaluationFunction을 사용할 것.)
  """
  def Action(self, gameState):
    ###################### Write Your Code Here ############################

    move_candidate = gameState.getLegalActions()

    list_nextstates = [gameState.generateSuccessor(0,action) for action in move_candidate]
    n_agents = gameState.getNumAgents()
    scores = [self.ExpectimaxValue(n_agents,1,nextstate,(n_agents * self.depth)-1) for nextstate in list_nextstates]

    bestScore = max(scores)
    Index = [index for index in range(len(scores)) if scores[index] == bestScore]

    get_index = random.choice(Index)

    return move_candidate[get_index]

  def ExpectimaxValue(self, n_agents, i_agent, gameState, depth):
    if (depth==0 or gameState.isLose() or gameState.isWin()):  # terminal tet
      return self.evaluationFunction(gameState)

    depth -= 1
    move_candidate=gameState.getLegalActions(i_agent)
    list_nextstates=[gameState.generateSuccessor(i_agent,action) for action in move_candidate]
    if (i_agent != 0): # Ghost
      val = sum([self.ExpectimaxValue(n_agents, (i_agent+1)%n_agents, nextstate, depth) for nextstate in list_nextstates])  / len(list_nextstates)

    else: #pacman
      val =  max([self.ExpectimaxValue(n_agents, 1%n_agents, nextstate, depth) for nextstate in list_nextstates])

    return val
```

이 알고리즘의 코드의 앞부분 또한 앞서 Minimax 에서 말한 부분과 거의 유사하다. 다만 Evaluation Function call 이 다를 뿐이다.
ExpectimaxAgent 의 evaluation 함수는 ExpectimaxValue 이다. Terminal test 를 한 뒤에 만약 terminal 이 아니라면 본격적으로 evaluate 한다. 여기서 또한 이 함수가 recursive 될 때마다 depth 를 하나 줄여준다.
Minimax 랑 구조가 흡사하지만, 여기서는 중요한 요소가 random 에 있다. Ghost 에 해당될 때, 모든 ExpectimaxValue 값들의 평균을 return 해준다. 이때 min 이나 max 를 쓰지 않아서 random 한 성향을 띈다.
마지막에 Pacman 일 때, max 를 해줌으로써, Expect 된 것들의 Max 를 최종적으로 return 한다.

3. 실행 결과

3.1 Minimax Agent 명령어의 승률 출력 화면

```
jk$ python pacman.py -p MinimaxAgent -m minimaxmap -a depth=4 -n 1000 -q
```

```
Win Rate: 68% (680/1000)
Total Time: 78.91477251052856
Average Time: 0.07891477251052856
========================================
```

승률이 68% (50% ~70%) 나왔다.

3.2 time_check.py 에서 출력된 실행시간 캡처 화면

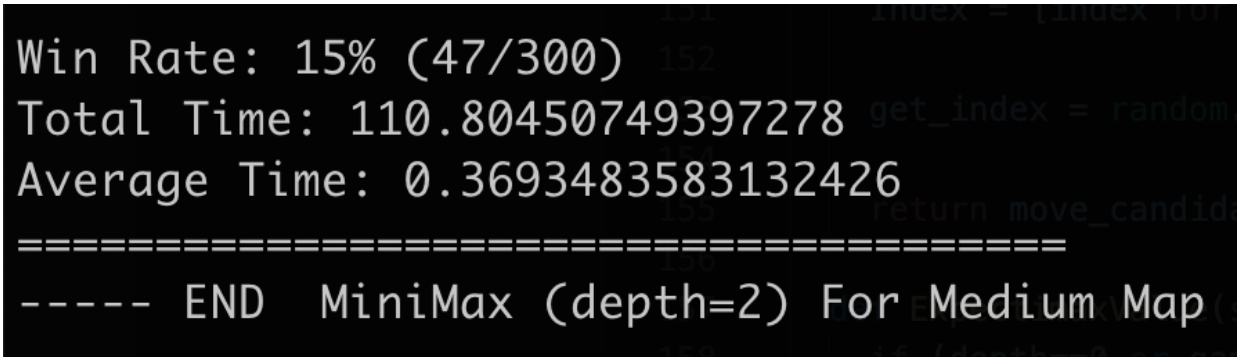
- Small Map (depth = 2)

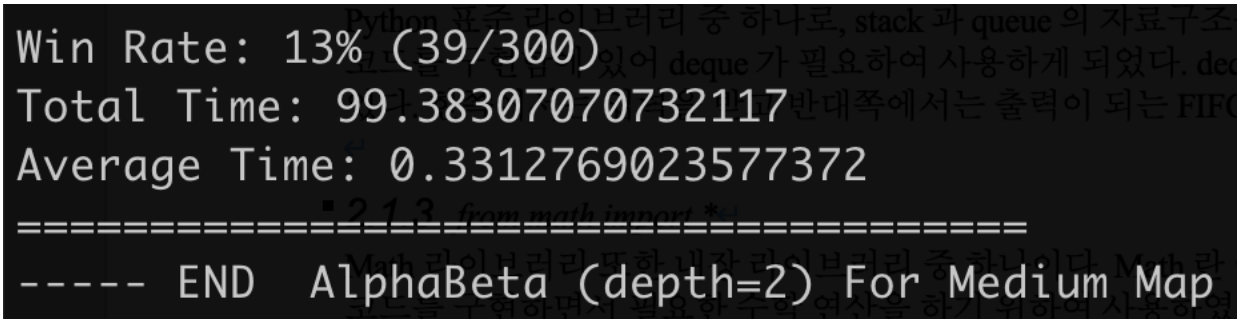
```
Win Rate: 10% (32/300)
Total Time: 45.86706519126892
Average Time: 0.1528902173042297
========================================
----- END  MiniMax (depth=2) For Small Map
```

```
Win Rate: 8% (26/300)
Total Time: 40.58717632293701
Average Time: 0.1352905877431233
========================================
----- END  AlphaBeta (depth=2) For Small Map
```

Small Map 에서 depth 가 2 일 때, AlphaBeta 가 Minimax 보다 약간 더 빠르다는 것을 알 수 있다.

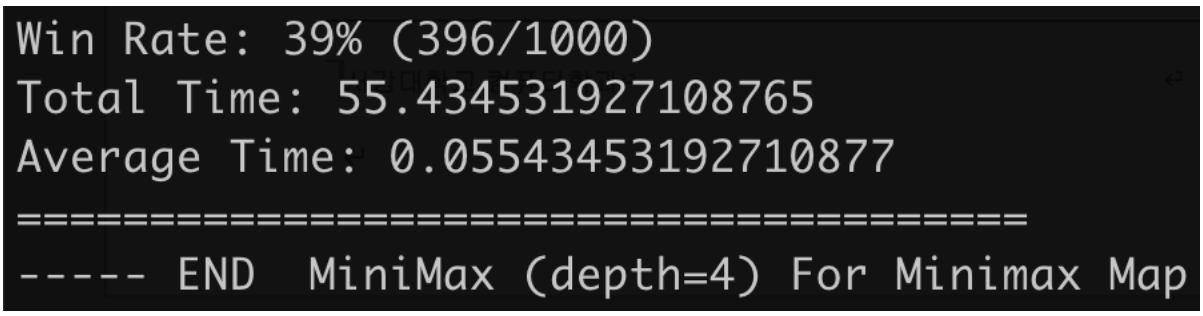
- Medium Map (depth = 2)

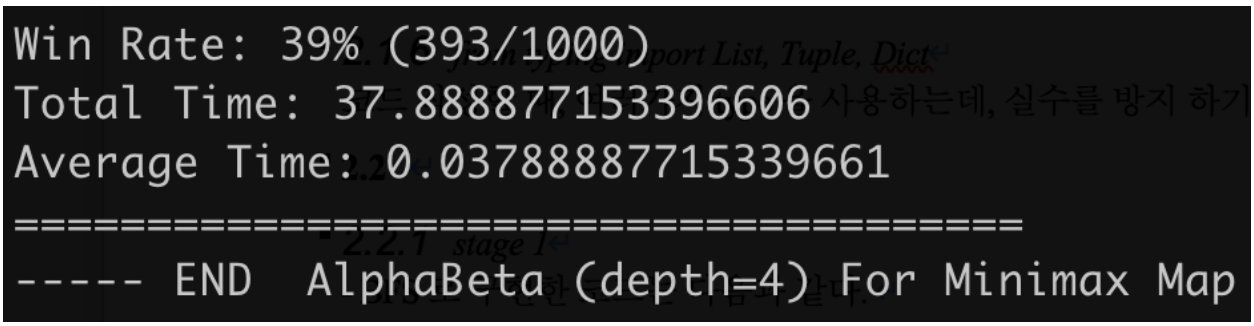




Medium Map 에서 depth 가 2 일 때, AlphaBeta 가 Minimax 보다 약간 더 빠르다는 것을 알 수 있다.

- Minimax Map (depth = 4)





Minimax Map 에서 depth 가 4 일 때, AlphaBeta 가 Minimax 보다 확실히 더 빠르다는 것을 알 수 있다.

- Small Map (depth = 3)

추가로 직접 depth = 3 일때 속도를 비교해 보았다.

Minimax Agent

```
jk$ python pacman.py -p MinimaxAgent -m Smallmap -a depth=3 -n 100 -q
Win Rate: 52% (52/100)
Total Time: 226.38447093963623
Average Time: 2.2638447093963623
========================================
```

Alpha-Beta Agent

```
python pacman.py -p AlphaBetaAgent -m Smallmap -a depth=3 -n 100 -q
Win Rate: 57% (58/100)
Total Time: 198.52105593681335
Average Time: 1.9852105593681335
========================================
```

Small Map 에서 depth 가 3 일 때, AlphaBeta 가 Minimax 보다 더 빠르다는 것을 알 수 있다.

3.3 Expectimax Agent 명령어의 승률 및 Score 출력 화면

```
(base) gangjueon-ui-MacBook-Pro:AI_Assignment02_update jk$ python pacman.py -p AlphaBetaAgent -m minimaxmap -a depth=4 -n 100 -q
```

```
----------Game Results----------
Average Score: 15.0
Score Results: -502, 532, -502, 532, -502, 532, 532, -502, -502, -502, -502, -502, -502, 532,
532, 532, 532, -502, 532, -502, -502, 532, 532, -502, -502, 532, -502, 532, -502, 532, -502, -
502, -502, 532, 532, -502, -502, -502, -502, -502, 532, -502, -502, 532, -502, 532, -502, 532,
 -502, 532, -502, 532, -502, 532, 532, 532, 532, 532, 532, -502, -502, 532, 532, 532, 532, 532
, -502, 532, 532, -502, 532, 532, -502, -502, -502, -502, -502, 532, -502, 532, 532, 532, 532,
 532, 532, -502, 532, -502, -502, 532, -502, -502, -502, 532, 532, -502, -502, -502, 532, 532
Record: Lose, Win, Lose, Win, Lose, Win, Win, Lose, Lose, Lose, Lose, Lose, Lose, Lose, Win, Win, Wi
n, Win, Lose, Win, Lose, Lose, Win, Win, Lose, Lose, Win, Lose, Win, Lose, Win, Lose, Lose, Lo
se, Win, Win, Lose, Lose, Lose, Lose, Lose, Win, Lose, Lose, Win, Lose, Win, Lose, Win, Lose,
Win, Lose, Win, Lose, Win, Win, Win, Win, Win, Win, Lose, Lose, Win, Win, Win, Win, Win, Lose,
 Win, Win, Lose, Win, Win, Lose, Lose, Lose, Lose, Lose, Win, Lose, Win, Win, Win, Win, Win, W
in, Lose, Win, Lose, Lose, Win, Lose, Lose, Lose, Win, Win, Lose, Lose, Lose, Win, Win

Win Rate: 50% (50/100)
Total Time: 0.3876965045928955
Average Time: 0.003876965045928955
========================================
```