[COSE-361] 인공지능: Assignment #2

2016130927 박준영

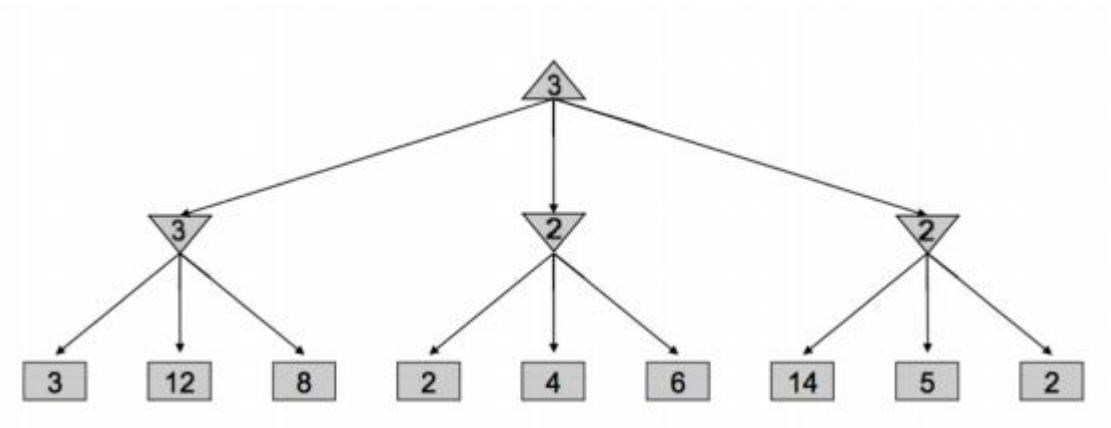
A. autograder.py 결과

```
Provisional grades
==================
Question q1: 4/4
Question q2: 5/5
Question q3: 5/5
Question q4: 0/5
Question q5: 0/6
------------------
Total: 14/25

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

B.

Q1. 알파베타 프루닝은 어떻게 미니맥스 트리를 탐색할 때 효율적으로 작동하는가?



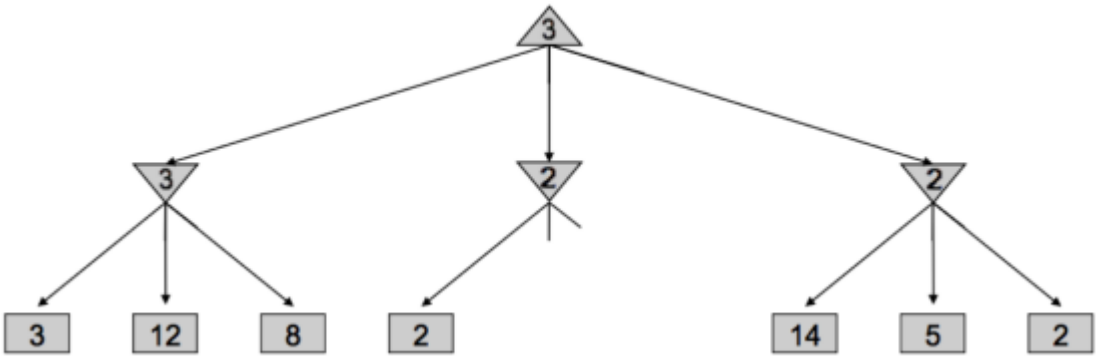
A1. 먼저 위 이미지에 나타난 미니맥스 트리가 알파베타 프루닝을 통해 어떻게 최적화되는지 확인해보자. 본 트리는 각 레벨(깊이)에서 자식 노드의 최댓값(최솟값)을 리턴해 결과적으로 루트 노드에서 최댓값을 구하는 미니맥스 트리 구조다. 루트 노드의 정삼각형을 맥시마이저, 역삼각형을 미니마이저로 생각하자.

(1). 최하단부의 하위 트리에서 각각 (3, 12, 8)의 최솟값 3, (2, 4, 6)의 최솟값 2, (14, 5, 2)의 최솟값 2가 정해진다. (2). 이후 루트 노드 차례에서 (3, 2, 2)의 최댓값 2가 정해진다.

트리의 깊이를 m, 브런칭 팩터가 b라고 할 때 시간 복잡도는 $\mathbf{O}(b^m)$이다. 미니맥스 트리는 각 레벨에 존재하는 상태 값을 구하기 위해 엄청난 양의 노드를 연산해야 한다.

알파베타 프루닝은 미니맥스 알고리즘의 **최댓값과 최솟값을 확인할 때 특정 위치의 값이 정해진다면 다른 노드 값은 확인할 필요가 없다**는 아이디어로, 미니맥스 트리의 최적화 기법이

다. 미니맥스 트리의 미니마이저(minimizer)와 맥시마이저(maximizer)가 각각 하위 트리의 최솟값, 최댓값을 찾는다는 점에서 근거해 “이미 더 좋은 선택지가 존재한다면” 트리의 다른 브런치를 자르는 프루닝 기법을 적용한다. 알바베타 프루닝은 그 이름처럼 알파와 베타라는 두 파라미터를 미니맥스 알고리즘에 추가해 구현한다. **알파**와 **베타**는 각각 맥시마이저, 미니마이저가 현재 레벨까지 탐색하면서 찾아낸 최댓값과 최솟값을 가리킨다. 위 미니맥스 트리에 알파베타 프루닝을 적용해보자.



(1). 좌측 서브트리 (3, 12, 8)에서 미니마이저1은 3을 얻는다. 현재 깊이 1의 미니마이저 값은 3이고, 맥시마이저가 가진 로컬 최댓값 알파도 3이다. (2). 중간 하부 트리에서 미나마이저2는 알파 3과 비교해 더 적은 값이 나오기 때문에 더 이상 탐색하지 않는다(프루닝). 왜냐하면 미니마이저2에서 2보다 더 큰 수가 나온다면 미니마이저의 값은 2이고, 2보다 더 작은 수가 나온다고 해도 맥시마이저가 2 (또는 더 작은 수)가 아니라 현재 알파(=3)를 고를 것이기 때문이다. (3). 세 번째 미니마이저의 하위 트리는 반면 (14, 5, 2)를 거치며 중간 노드 5까지 현재 알파 값보다 더 큰 값을 가지기 때문에 마지막 노드 2까지 탐색하였다. 결과적으로 알파베타 프루닝 기법을 통해 기존 미니맥스 알고리즘에 2개 노드를 프루닝했다.

알파베타 프루닝 기법은 미나마이저의현재 값이 알파보다 작다면 프루닝, 맥시마이저의 현재 값이 베타보다 크다면 프루닝한다. 프루닝은 **연산 결과에 아무런 영향을 미치지 않고 연산량을 줄이는 기법**이다. 중간 단계의 노드가 미니맥스에서와 달리 모든 노드를 살펴보지 않았을 수 있기 때문에 값이 다를 수도 있다. 하지만 이 값은 실제 값이 아니지만 결과 연산에는 영향을 미치지 않기 때문에 **결과가 미니맥스와 같음을 보장(guarantee)**한다.

하지만 위 알파베타 프루닝 기법 역시 다시 한번 나름의 최적화 기법을 적용할 수 있다. 만일 세 번째 자식 노드의 순서가 (12, 5, 2)가 아니라 2부터 시작했다면 맥시마이저의 알파 값 3보다 작은 수를 먼저 탐색했기 때문에 곧바로 프루닝된다. 이처럼 알파베타 프루닝 기법을 적용할 때 미니맥스 트리가 **완벽히 정렬**되었다면 시간 복잡도는 $\mathbf{O}(b^{\frac{m}{2}})$이 된다.

Q2. 반응형 에이전트(reflex agent)가 미니맥스 또는 알파베타 프루닝 알고리즘보다 성능이 좋을 때는 언제인가?

A2. 팩맨 게임의 에이전트가 **반응형**일 때 에이전트는 현재 상황을 **즉시 판단**한다. 현재 에이전트가 처한 상황("얼마나 많은 음식이 근처에 있나?", "얼마나 많은 고스트가 근처에 위치해 있나?")을, 휴리스틱 값을 리턴하는 평가 함수로 값을 측정해 유틸리티 최댓값을 가져올 것으로 생각되는 액션을 행한다.

반면 **미니맥스** (또는 알파베타 프루닝) 알고리즘은 팩맨과 고스트의 행위 모두를 트리 형태로 표현, 번갈아 행위(Tic-Tac-Toe)하면서 상태 값을 최대화/최소화한 뒤 결과적으로 팩맨 에이전트 입장에서 유틸리키가 최댓값이 되는 행동을 택한다. 결과적으로 팩맨은 현재 시점에서 최대 유틸리티를 택하기보다 고스트의 행위로 인한 피해를 최소화할 수 있다.

그렇다면 어떤 상황에서 반응형 에이전트가 미니맥스 알고리즘을 사용한 에이전트보다 더 성능이 뛰어날 수 있을까? 고스트(상대방)의 행동이 얼마나 최적화되어 있는지, 게임의 액션이 결정적/확률적인지, 또는 게임의 성향 자체를 고려할 수 있을 것이다.

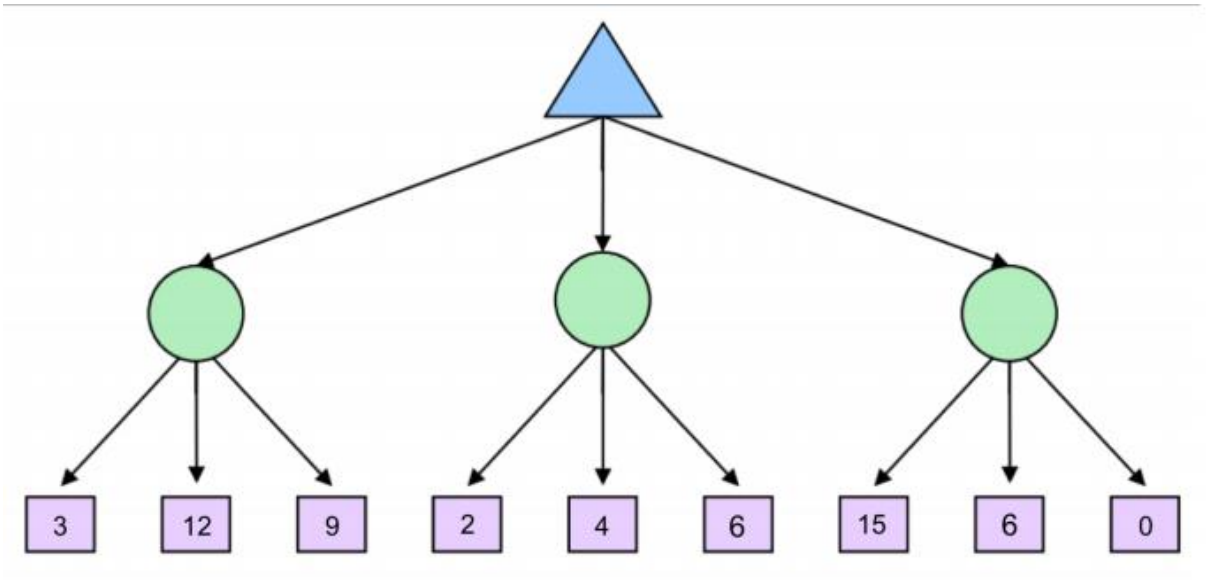
(1). 미니맥스 알고리즘은 팩맨과 고스트가 서로 간의 행동을 언제나 예측하는 합리적이고 최적의 행동을 하고 있다는 가정을 둔다. 이런 회의적(pessimistic)인 가정에서 미니맥스의 팩맨은 비록 고스트로 인해 죽는 일은 없지만, 그렇다고 유틸리티 최댓값을 언제나 얻을 수는 없다. 하지만 **상대방 고스트가 충분히 지능적이지 않다**면, 반응형 에이전트는 미니맥스보다 더 뛰어난 성과를 거둘 수 있을 것이다.

(2). 미니맥스 알고리즘의 게임 트리는 각 에이전트(팩맨 및 고스트)가 특정 액션을 선택한 결과가 필연적으로 결정되는(deterministic) 구조다. 만일 선택에서 상태값이 **확률적(stochastic)으로 결정되는 구조라면, 반응형 모델이 현 상황에서 즉각적으로 고르는 선택이 더 큰 효용으로 이어질 수도 있다**.

(3). 미니맥스 알고리즘은 멀티 에이전트가 서로 번갈아 하는 제로섬 게임(zero-sum game)에서 좋은 효용을 보인다. 만일 **일반적인 게임(general game)**이라면 미니맥스 알고리즘을 통해 걱정하는 바와 같이 에이전트가 상대방의 행위로 인해 게임이 '실패'할 일이 없다. 오히려 이 경우 **상대방보다 더 많은 유틸리티를 얻는 것만 고려**하면 되기 때문에, 현재 시점에서 즉각적으로 더 큰 유틸리티로 이어지는 행위를 택하는 반응형 모델이 더 뛰어날 수 있다.

Q3. 최대 평균(Expectimax) 알고리즘이 알파베타 프루닝 알고리즘보다 성능이 좋을 때는 언제인가? (직접 제안한 질문 및 답변)

A3. 미니맥스 알고리즘은 모든 정보를 알고 있는 완벽한(perfect) 최적(optimal)의 상대방을 가정하기 때문에 최악의 경우 얻을 수 있는 가장 좋은 해결책을 찾아낸다. 하지만 **상대방이 언제나 최적의 행동을 하지 않는다면?** 예를 들어 주사위와 같은 확률 게임을 할 때 상태 값은 고정되어 있지 않다. 이 경우 미니맥스 상태 값은 주어진 상황에서 에이전트가 택하는 확률을 통해 구해야 한다. 미니맥스가 다음 노드의 행동(successor)이 확정적(deterministic)인 게임이라면, 이 접근 방법은 **확률적(stochastic)인 게임**에 적합하다.



이때 **기회 상태(chance states)의 기댓값**을 미니맥스의 상태 값으로 활용하자. 확정적으로 다음 노드(successor)가 무엇이 될지 알 수 없기 때문에 노드를 택할 확률을 이용해 기댓값을 계산하는 것이다. 이때 최대평균(Expectimax) 알고리즘은 미니맥스와 달리 모든 successor를 확인해야 하는데, 극단값에 영향을 받을 수 있기 때문이다. 따라서 **프루닝은 적용할 수 없다**.

위 이미지의 게임 트리에서 초록색 노드가 기회 상태를 나타내는 노드로, 하위 트리의 값을 기댓값으로 가진다. (1). 각 하위 트리를 택할 확률이 동일(즉 1/3)하다고 할 때 기회 상태의 값은 좌측부터 8, 4, 7이다. (2). 루트 노드가 맥시마이저인 트리이므로 팩맨 에이전트는 기댓값이 가장 큰 좌측 노드를 택할 것이다.

비록 위 이미지의 경우 미니맥스 알고리즘을 적용해도 동일한(기회 상태를 미니마이저로 생각한다면) 결과로 이어지지만, 그렇지 않은 경우도 있다. 기댓값이 높더라도 실제로 택할 값은 확률적으로 결정되기 때문이다.

이처럼 미니맥스 알고리즘이 최악의 경우를 상정, 상대방이 최적(optimal)의 행동만을 한다고 가정하기 때문에 보수적인 접근 방법을 택하는 반면, 최대평균 알고리즘은 다소 낙관적이다. **상대방이 충분히 지능적이지 않다**면, 적어도 확률적으로는 미니맥스 알고리즘보다 더 많은 유틸리티를 얻을 수 있다. 이때 **기댓값을 사용하는 것은 물론 합리적(rational)한 상황에서 확률을 통한 평균**을 이용하는 게 가장 적절하기 때문이다. 반대로 상대방이 어떻게 행동할지 **사전 정보가 충분하지 않다**면 미니맥스 알고리즘이 보다 적절하다.