인공지능 과제 #2 (2025년도 1학기)

이번 과제에서는 minimax, expectimax 탐색 알고리즘 등을 이용하여 유령이 포함된 PacMan 게임을 수행하는 agent를 구현합니다. 제공되는 압축파일(ai-sp25_hw2_code.zip)을 풀면, multiagent라는 이름을 가진 디렉토리를 확인할 수 있습니다. 이 디렉토리에 대해서 주어진 문제에 맞는 요구사항을 구현하면 됩니다.

과제 진행 시 주의 사항은 다음과 같습니다.

1. 구현해야 하는 파일과 코드가 아닌 게임 그래픽 관련된 코드나 다른 코드들은 수정하지 않습니다.
2. 모든 채점은 autograder.py 파일을 통해서 이루어집니다.
3. 과제 제출
   A. 기한: **2025년 5월 9일(금) 23:59까지**
   B. 방법: 첫번째 과제와 동일합니다.
4. **주의 사항**
   A. zip 형식 이외의 다른 압축 형식으로 제출하거나, Windows 10이나 Windows 11 환경에서 압축이 풀리지 않은 zip 파일들은 0점 처리합니다.
   B. 파일 이름이 예시 형식과 맞지 않을 경우 이 과제 만점의 10%를 감점합니다.
   C. 다른 사람의 코드를 그대로 사용하거나 인터넷에 공개된 코드나 로직을 베껴서 사용한 것이 적발되면 해당 과제는 수업 첫 시간의 안내에 따라 처리합니다. 여러분들이 인터넷에서 찾을 수 있을만한 자료들은 이미 조교들이 모두 확보하고 있다고 생각하시면 됩니다. 스스로의 힘으로 과제를 풀어보세요.

1. 과제 소개

이 과제에서 minimax, expectimax 탐색 알고리즘과 직접 설계한 평가함수를 이용해 유령을 포함한 팩맨 agent를 구현하게 됩니다. **과제의 코드는 과제#1과 크게 다르지 않지만, 과제#1의 코드를 재활용하지 말고 새로 본 과제의 코드를 다운로드하여 시작하세요.**

이 과제의 채점을 위한 자동 채점기 명령어는 아래와 같습니다.

```
python autograder.py
```

이 과제의 코드는 여러 개의 Python 파일들로 구성되어 있으며, 과제를 완료하기 위해 일부는 읽고 이해해야 하며 일부는 무시해도 됩니다. 모든 코드와 지원 파일은 ai-sp25_hw2_code.zip에 포함되어 있습니다.

[구현해야 하는 파일]

multiAgents.py	모든 multi-agent의 탐색 agent를 포함하는 파일

[참고할 파일]

pacman.py	팩맨 게임을 실행하는 메인 파일입니다. 이 파일은 이 과제에서 사용하는 팩맨 GameState type에 대해 설명합니다.
game.py	팩맨 세계를 작동시키는 agent 상태, agent, 방향, 그리드 등 여러 지원 유형에 대해 설명합니다.
util.py	탐색 알고리즘 구현에 유용한 데이터 구조.

[수정 금지 파일]

graphicsDisplay.py	팩맨용 그래픽
graphicsUtils.py	팩맨 그래픽 지원

textDisplay.py	팩맨의 ASCII 기반 그래픽
ghostAgents.py	유령 제어용 agent
keyboardAgents.py	팩맨 제어를 위한 키보드 인터페이스
layout.py	레이아웃 파일을 읽고 내용을 저장하는 코드
autograder.py	과제 자동 채점기
testParser.py	자동채점 테스트 및 솔루션 파일 파싱
testClasses.py	기본 자동채점 테스트 클래스
test_cases/	각 질문에 대한 테스트 케이스가 포함된 디렉터리

2. Multi-agent 팩맨에 오신 것을 환영합니다.

코드를 다운로드하고 압축을 푼 다음 디렉터리로 변경한 후 명령줄에 다음을 입력하면 팩맨 게임을 플레이할 수 있습니다.

`python3 pacman.py`

`multiAgents.py` 에 구현된 `ReflexAgent` 의 실행 명령어는 아래와 같습니다.

`python3 pacman.py -p ReflexAgent`

`ReflexAgent` 는 아래와 같이 실행하며 간단한 구조의 미로에서도 느리게 실행됩니다.

`python3 pacman.py -p ReflexAgent -l testClassic`

`multiAgents.py` 파일을 살펴보고, 어떻게 동작하는지 확인해 보시기 바랍니다.

3. Q1: Reflex Agent

`multiAgents.py`에 구현된 `ReflexAgent`가 동작하도록 개선해 봅시다. 제공된 `ReflexAgent` 코드는 `GameState`의 정보를 조회하는 방식들에 대한 몇 가지 예시를 제공합니다. Reflex agent를 제대로 동작시키기 위해서는 음식의 위치와 유령의 위치를 모두 고려해야합니다. 구현한 agent는 아래 명령어를 통해 `testClassic` 미로를 쉽고 안정적으로 클리어해야 합니다.

`python3 pacman.py -p ReflexAgent -l testClassic`

하나 또는 두 마리의 유령을 포함한 `mediumClassic` 미로에서 여러분이 구현한 reflex agent를 테스트해보세요. (애니메이션을 끄면 화면 표시 속도를 향상시킬 수 있습니다.)

```
python3 pacman.py --frameTime 0 -p ReflexAgent -k 1
python3 pacman.py --frameTime 0 -p ReflexAgent -k 2
```

Agent가 어떻게 작동하나요? 여러분이 만든 평가 함수가 아주 뛰어나지 않는 한 기본 미로에 두 마리의 유령이 때는 죽는 경우가 많을 것입니다.
[참고] `newFood`에는 `asList()` 함수가 있습니다.
[참고] 특징으로 값들을 그대로 사용하지 말고 중요한 값(예: 음식까지의 거리)의 역수를 사용해보세요.
[참고] 여러분이 작성하고 있는 평가함수는 상태-행동 쌍을 평가하며, 과제의 후반부에서는 상태를 평가하게 됩니다.
[참고] 디버깅을 위해 다양한 객체의 내부 내용을 참고하는 것이 좋습니다. 아래 예시와 같이 `print()`를 통해 객체 내부의 내용을 확인할 수 있습니다.

`print(newGhostStates)`

[옵션] 기본적으로 유령은 랜덤하게 움직입니다. 게임을 더 재미있게 플레이하고 싶다면 `-g DirectionalGhost` 옵션을 사용해서 더 똑똑한 방향성을 가지는 유령을 추가할 수 있습니다. 무작위성 때문에 여러분의 agent가 향상되었는지 알 수 없다면 `-f` 옵션을 사용해 랜덤 seed를 고정할 수 있습니다. 그리고 `-n` 옵션을 이용해 여러 게임을 연속으로 플레이할 수 있습니다. 만약 많은 게임을 빠르게 실행하고 싶다면 `-q` 옵션을 이용해 그래픽을 꺼보세요.

[채점] 구현한 agent 를 `openClassic` 미로에서 10 회 실행한 후, 다음과 같은 조건에 따라 점수를 부여합니다

- Agent가 시간을 초과하거나, 한 번도 이기지 못하면 0점을 받게 됩니다.
- Agent가 5회 이상 승리하면 1점, agent가 10게임 모두 승리하면 2점을 받게 됩니다.
- Agent의 평균 점수가 500점 이상이면 1점을, 1,000점 이상이면 2점을 추가로 받게 됩니다.

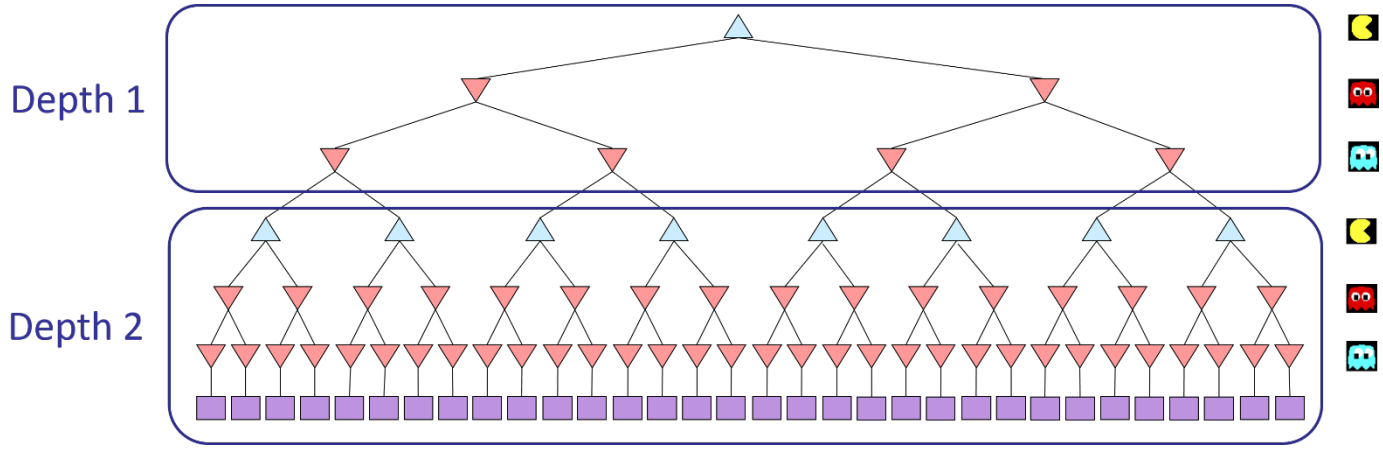
위의 시험 조건은 모두 자동 채점기에 구현되어 있습니다. 아래 명령을 실행하여 작성한 코드가 자동 채점기를 통과하는지 확인하세요.

`python3 autograder.py -q q1`

그래픽을 사용하지 않고 실행하고 싶다면 아래 명령어를 사용하세요.
`python3 autograder.py -q q1 --no-graphics`

4. Q2: minimax

`multiAgents.py`의 `MinimaxAgent` 클래스에 적대적 탐색(adversarial search) agent를 구현합니다. 여러분의 minimax agent는 유령 수에 상관없이 작동해야 하므로 수업시간에 배운 것보다 좀더 일반적으로 구현해야 합니다. 특히 minimax 트리는 모든 max 레이어마다 여러 개의 min 레이어(유령 당 하나씩)를 가지게 됩니다.
여러분이 작성한 코드는 게임 트리를 임의의 깊이까지 확장할 수 있어야 합니다. 제공된 `self.evaluationFunction`를 이용해 여러분의 minimax 트리의 리프 노드들에 점수를 매겨보세요. `MinimaxAgent`는 `MultiAgentSearchAgent`를 확장해 `self.depth` 및 `self.evaluationFunction`에 대한 접근을 제공합니다. `self.depth`와 `self.evaluationFunction`는 커맨드 라인의 옵션에 따라 설정되고, minimax 코드에서 적절히 이 두 변수를 참조하세요.
[중요!] 하나의 탐색 깊이는 팩맨의 움직임과 모든 유령의 응답으로 구성됩니다. 즉 깊이가 2인 탐색에서는 팩맨과 유령들이 각각 두 번씩 이동하게 됩니다. (아래 그림 참고)



[채점] 여러분이 작성한 코드가 올바른 갯수의 게임 상태를 탐색하는지 확인합니다. 이는 minimax 알고리즘에서 미묘한 오류를 감지할 수 있는 유일한 방법입니다. 따라서 자동 채점기는 `GameState.generateSuccessor`를 호출하는 횟수를 매우 까다롭게 판단합니다. 만약에 여러분의 코드가 `GameState.generateSuccessor`를 필요 이상으로 많이 호출하거나 적게 호출하면 자동 채점기가 이를 체크합니다. 아래 명령을 실행하여 작성한 코드를 테스트하고 디버깅해보세요.

`python3 autograder.py -q q2`

이 명령어는 팩맨 게임뿐만 아니라 여러 개의 작은 트리에서 알고리즘이 수행되는 것을 보여줍니다.
그래픽 없이 코드를 실행하고 싶다면 아래 명령을 실행하세요.

`python3 autograder.py -q q2 --no-graphics`

[힌트]

- Helper 함수를 이용해 재귀적으로 알고리즘을 구현하세요.
- minimax 알고리즘을 올바르게 구현해도 일부 테스트에서 팩맨이 패배할 수 있습니다. 이는 정상적인 현상이며 테스트에 통과할 수 있습니다.
- 본 문제의 팩맨 테스트를 위한 평가 함수 `self.evaluationFunction`는 이미 구현되어 있습니다. 이 함수를 변경하지 않도록 주의하세요. 구현되어 있는 평가 함수는 동작이 아닌 상태를 평가하고 있습니다. Look-ahead agent는 미래의 상태를 평가하지만 reflex agent는 현재 상태로부터 동작을 평가합니다.
- `minimaxClassic` 미로 초기 상태에서의 minimax 값은 1부터 4까지의 깊이에서 각각 9, 8, 7, -492입니다. 깊이 4의 minimax 값이 매우 안 좋지만, 이런 경우에도 665/1,000 정도의 승률로 성공합니다.

`python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4`

- 팩맨은 항상 agent 0이며, agent는 인덱스가 증가하는 순서대로 움직입니다.
- minimax의 모든 상태는 `GameStates`여야 하며, `getAction`에 전달되거나 `GameState.generateSuccessor`를 통해 생성되어야 합니다. State들을 임의로 단순하게 초기화시키지 마세요.
- `openClassic`, `mediumClassic`와 같은 큰 미로에서는 팩맨이 잘 죽지는 않지만, 잘 이기지는 못한다는 것을 알게 될 것입니다. 이리저리 뛰어다니기만 하거나 점을 먹은 후 어디로 가야 할 지 몰라서 점을 먹지 않고 바로 옆을 맴돌기도 합니다. 지금은 신경쓰지 마세요. Q5에서 이 문제를 해결할 것입니다.
- 팩맨은 자신이 죽음을 피할 수 없다고 생각하면, 패널티를 줄이기 위해 가능한 빨리 게임을 끝냅니다. 무작위로 움직이는 유령이 있을 때 이렇게 행동하는 것이 잘못된 일일지도 모르지만, minimax 에이전트는 항상 최악의 경우를 가정합니다. 아래 명령을 실행했을 때 팩맨이 가장 가까운 유령에게 달려드는 이유를 이해할 수 있어야 합니다.

`python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3`

5. Q3: 알파-베타 가지치기

Minimax 트리를 더 효율적으로 탐색하기 위해 알파-베타 가지치기를 사용하는 새로운 agent를 만드세요. 이 agent는 `AlphaBetaAgent`로 지정됩니다. 알파-베타 가지치기 로직을 여러 개의 min agent로 확장할 수 있도록 구현해야 합니다.
알파-베타를 사용하여 알고리즘을 실행할 경우 속도 향상을 볼 수 있을 것입니다. (깊이 2의 minimax와 깊이 3의 알파-베타 속도가 거의 같을 것입니다.) 다음 명령어를 실행했을 때, 각 움직임 당 몇 초 안에 실행되어야 합니다.

`python3 pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic`

`AlphaBetaAgent`의 minimax 값은 `MinimaxAgent`의 minimax 값과 동일해야 합니다. 단, 동점일 때의 행동이 서로 다르기 때문에 선택된 작업이 다를 수 있습니다. (예: `minimaxClassic` 미로 초기 상태에서의 minimax 값은 1부터 4까지의 깊이에서 각각 9, 8, 7, -492이므로, `AlphaBetaAgent`에서의 값도 이와 같아야 합니다.)

[채점] 정확히 몇 번의 탐색이 일어나는지 확인할 것입니다. 따라서, 자식 노드를 재정렬하지 않고 알파-베타 가지치기를 수행해야 합니다. 즉, 자식 노드(혹은 후속 상태, successor)는 항상 `GameState.getLegalActions`가 반환한 순서대로 처리되어야 합니다. `GameState.generateSuccessor`를 필요 이상으로 호출하지 마세요.
자동 채점기에서 탐색하는 상태 집합과 여러분의 답을 똑같이 만들기 위해서 같은 상태의 노드를 가지치기 하지 마세요. 같은 상태를 가지는 노드를 가지치기 하여 실행 속도를 높일 수 있겠지만, 자동 채점기로부터 좋은 점수를 받을 수 있는 방법이 아닙니다.
아래 의사 코드는 알파-베타 알고리즘을 나타냅니다.

Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v > β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v < α return v
        β = min(β, v)
    return v
```

코드를 테스트하고 디버그 하려면 다음을 실행하세요.

`python3 autograder.py -q q3`

이 명령어는 팩맨 게임 뿐만 아니라 여러 개의 작은 트리에서 알고리즘이 수행되는 것을 보여줍니다.
만약 그래픽 없이 코드를 실행하고 싶다면 아래 명령을 실행하세요.

`python3 autograder.py -q q3 --no-graphics`

알파-베타 알고리즘을 올바르게 구현하였더라도 일부 테스트에서 팩맨이 패배할 수 있습니다. 이는 정상적인 현상이며 채점시 테스트를 통과할 수 있습니다.

6. Q4: Expectimax

Minimax 와 알파-베타 알고리즘은 최적의 결정을 내리는 상대와 게임을 하고 있다고 가정합니다. 하지만 실제로는 항상 최적의 결정을 내리는 상대와 게임을 하는 것은 아니라는 걸 알 것입니다. 이 문제에서는 차선의 선택을 할 수 있는 agent 의 확률적 행동을 모델링하는 `ExpectimaxAgent` 를 구현해 봅니다.
이러한 알고리즘의 장점은 일반적으로 적용할 수 있다는 점입니다. 자동 채점기에는 여러분의 개발을 가속화할 수 있도록 generic 트리에 기반한 몇 가지의 테스트 사례들이 제공됩니다. 다음 명령어를 사용하여 작은 게임 트리에서 여러분의 코드를 디버깅해보세요.

`python3 autograder.py -q q4`

작고 관리하기 쉬운 테스트 사례를 사용하면 버그를 빨리 찾을 수 있기 때문에 이를 활용해 디버깅 하는 것을 권합니다.
여러분이 구현한 알고리즘이 작은 트리에서 동작한다면 팩맨에서도 잘 동작할 것입니다. 그러나 랜덤하게 움직이는 유령들은 최적의 minimax agent 가 아니므로 minimax 탐색으로 랜덤하게 움직이는 유령들을 모델링하는 것은 적절하지 않을 수 있습니다. `ExpectimaxAgent` 는 더 이상 모든 유령 행동들에 대한 최소값을 구하지 않고 유령 행동 방식에 대한 agent 모델에 따른 기대값을 구합니다. 여러분의 코드를 단순화하기 위해, 무작위로 `getLegalActions` 을 균일하게 선택하는 상대와 게임을 하고 있다고 생각해보세요.
팩맨에서 ExpectimaxAgent 를 작동시키려면 아래와 같이 실행하세요.

`python3 pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3`

유령과 가까운 곳에서의 움직임은 좀 더 신중해야 합니다. 특히, 팩맨이 유령에 잡힐 수도 있지만 음식 몇 개를 더 얻고 탈출할 수 있다고 판단되면 팩맨은 시도해 볼 수도 있습니다. 다음 두 가지 시나리오의 결과를 조사해 보세요.

`python3 pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10`
`python3 pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10`

`ExpectimaxAgent` 가 절반 정도 이기는 반면 `AlphaBetaAgent` 는 항상 지는 것을 확인해야 합니다. minimax 사례와 동작이 다른 이유를 명확히 이해해야 합니다.
Expectimax 알고리즘을 올바르게 구현하였더라도 일부 테스트에서 팩맨이 패배할 수 있습니다. 이는 정상적인 현상입니다.

7. Q5: 평가 함수

마지막 문제에서는 `betterEvaluationFunction` 에 팩맨에 대한 더 나은 평가 함수를 작성합니다. 평가 함수는 reflex agent 평가 함수처럼 행동이 아닌 상태를 평가해야 합니다. 깊이가 2 인 탐색을 사용했을 때, 여러분이 구현한 평가 함수는 `smallClassic` 미로에서 랜덤하게 움직이는 유령 하나를 절반 이상 클리어하면서 합리적인 속도로 실행되어야 합니다 (만점을 받으려면 팩맨이 승리할 때 평균 1000 점 정도는 받아야 합니다).

[채점] `smallClassic` 미로에서 여러분의 agent 를 10 회 실행시켜, 아래와 같이 평가 함수에 점수를 부여합니다.

- 자동 채점기의 시간 초과 없이 단 한 번이라도 이긴다면 1 점을 받습니다. 만약 이 기준을 충족시키지 못한다면 0 점을 받습니다.
- 5 회 이상 승리하면 1 점을, 10 회 이상 승리하면 2 점을 추가로 받습니다.
- 평균 점수가 500 점 이상이라면 1 점을, 평균 점수가 1000 점 이상이라면 2 점을 추가로 받습니다(패배한 게임의 점수도 포함합니다).
- `--no-graphics` 옵션과 함께 실행할 때 자동 채점 시 게임에 평균 30 초 미만이 소요될 경우 1 점을 추가로 받습니다.
- 평균 점수와 계산 시간에 대한 추가적인 점수들은 최소 5 회 이상 승리한 경우에만 부여되는 것을 주의하길 바랍니다.
- 첫 번째 과제로부터 어떠한 파일들도 복사해서 사용하시지 마시길 바랍니다. 만약, 그럴 경우 자동 채점기를 통과하지 못합니다.

아래 명령을 실행하여 여러분이 작성한 코드가 자동 채점기를 통과하는지 확인하세요.

`python3 autograder.py -q q5`

그래픽 없이 코드를 실행하고 싶다면 아래 명령을 실행하세요.

`python3 autograder.py -q q5 --no-graphics`