인공지능 과제 #3 (2024년도 1학기)

이번 과제에서는 value iteration과 Q-learning을 구현합니다. 제공되는 압축파일(ai-sp24_hw3_code.zip)을 풀면, reinforcement라는 이름을 가진 디렉토리를 확인할 수 있습니다. 이 디렉토리에 대해서 주어진 문제에 맞는 요구사항을 구현하면 됩니다.

과제 진행 시 주의 사항은 다음과 같습니다.

- 구현해야 하는 파일과 코드가 아닌 게임 그래픽 관련된 코드나 다른 코드들은 수정하지 않습니다.
- 모든 채점은 `autograder.py` 파일을 통해서 이루어집니다.
- 과제 제출
  A. 기한: **6월 3일 23:59**까지
  B. 방법: LMS 인공지능 과목의 레포트 제출에서 **구현이 포함되어 있는 파일들을 하나의 zip 파일로 압축해서 제출하시기 바랍니다.** (파일 이름 예시, 홍길동_20000000.zip)
- **주의 사항**
  A. zip 형식 이외의 다른 압축형식으로 제출했거나, zip 형식 파일이어도 윈도우 PC에서 압축이 풀리지 않은 파일들은 채점에서 제외됩니다.
  B. 다른 사람의 코드를 그대로 사용한 것이 적발되면 보여준 사람, 베낀 사람 모두 0점 처리합니다.
  C. 인터넷에 공개된 코드나 로직을 베껴서 사용하지 마세요. 과제에 사용한 것이 적발되면 해당과제는 0점 처리합니다. 여러분들이 인터넷에서 찾을 수 있을만한 자료들은 이미 조교들이 모두 확보하고 있다고 생각하시면 됩니다. 스스로의 힘으로 과제를 풀어보세요.

1. 과제 소개

이 과제에서는 value iteration과 Q-learning을 구현합니다. 여러분이 구현한 agent는 먼저 GridWorld에서 테스트된 후, 시뮬레이션 된 로봇 컨트롤러와 팩맨에 적용됩니다.

이전의 과제와 마찬가지로 이 과제에도 컴퓨터에서 답을 채점할 수 있는 자동 채점기가 포함되어 있습니다. 이 자동 채점기는 아래 명령어로 실행할 수 있습니다:

```
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-sp24_hw3_code.zip에 포함되어 있습니다.

[구현해야 하는 파일]

valueIterationAgents.py	Known MDPs를 해결하기 위한 value iteration agent
qlearningAgents.py	Gridworld, Crawler, 팩맨을 위한 Q-learning agent
analysis.py	과제에서 주어진 질문에 대한 답을 입력하는 파일

[살펴볼 만한 파일]

mdp.py	General MDPs에 대한 메서드 정의
learningAgents.py	Agent가 확장할 ValueEstimationAgent 및 QLearningAgent와 같은 기본 클래스 정의
util.py	Q-learner에게 유용한 util.Counter와 같은 유틸리티
gridworld.py	Gridworld 구현
featureExtractors.py	(state, action)쌍에서 특징을 추출하기 위한 클래스 Approximate Q-learning agent에 사용됨(qlearningAgents.py)

[지원 파일]

environment.py	일반적인 강화학습 환경을 위한 abstract class(gridworld.py에서 사용됨)
graphicsGridworldDisplay.py	Gridworld 그래픽 디스플레이
graphicsUtils.py	그래픽 유틸리티
textGridworldDisplay.py	Gridworld 텍스트 인터페이스를 위한 Plug-in
crawler.py	Crawler 코드와 테스트 하네스 (코드 수정하지 말 것)
graphicsCrawlerDisplay.py	Crawler 로봇을 위한 GUI
autograder.py	자동 채점기
testParser.py	자동 채점기 테스트 및 솔루션 파일 파싱
testClasses.py	일반적인 자동 채점기 테스트 클래스
test_cases/	각 문제에 대한 테스트 케이스가 포함된 디렉토리
reinforcementTestClasses.py	3번 과제 자동 채점기 테스트 클래스

2. MDPs

아래 명령어를 통해 화살표 키를 사용하는 수동 제어 모드로 Gridworld를 실행해봅시다.

`python gridworld.py -m`

위의 명령어를 실행하면 두 개의 출구가 있는 layout을 볼 수 있습니다. 파란색 점은 agent를 나타냅니다. 위쪽 방향키를 누르면 agent는 80%의 확률로 북쪽으로 이동합니다.

시뮬레이션의 여러 부분을 직접 제어할 수 있습니다. 아래 명령어를 통해 옵션 목록을 확인할 수 있습니다.

`python gridworld.py -h`

기본 agent는 무작위로 움직입니다. 아래 명령어로 확인해보세요.

`python gridworld.py -g MazeGrid`

Agent가 출구에 도달할 때까지 무작위로 그리드를 돌아다니는 것을 볼 수 있습니다. 우리가 기대하는 AI agent스러운 움직임이 아닙니다.

Gridworld MDP는 먼저 GUI에 표시된 이중 상자로 들어가 사전 종료 state가 된 다음 에피소드가 실제로 종료되기 전에 특별한 'exit' action을 수행해야 합니다. 실제 종료 state인 `TERMINAL_STATE`는 GUI에 표시되지 않습니다. 에피소드를 수동으로 실행하는 경우 `-d` 옵션으로 설정하는 discount rate(기본값 0.9)으로 인해 총 return 값이 생각했던 것보다 작을 수 있습니다.

그래픽 출력과 함께 제공되는 콘솔 출력에 agent가 경험하는 각 전이가 표시됩니다. 그래픽 출력과 콘솔 출력을 사용하지 않으려면 `-q` 옵션을 사용하세요.

팩맨과 마찬가지로 (x, y) Cartesian 좌표계로 위치가 표현되며, 모든 배열은 `[x][y]`로 인덱싱 됩니다. 북쪽은 y가 증가하는 방향입니다. 기본적으로 대부분의 전이는 0의 보상을 받습니다. `-r` 옵션을 통해 이를 변경할 수 있습니다.

Q1. Value iteration (5점)

다음은 value iteration state 업데이트 방정식 입니다.

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

valueIterationAgents.py에 구현이 완료되지 않은 ValueIterationAgent가 있습니다. 이번 문제는 ValueIterationAgent에 value iteration agent를 작성하는 것입니다. Value iteration agent는 강화학습 agent가 아닌 오프라인 플래너입니다. 따라서 초기 계획 단계에서 실행할 value iteration의 반복 횟수를 -i 옵션을 통해 설정해야 합니다. ValueIterationAgent는 생성 시 MDP를 가져오고 생성자가 객체를 반환하기 전에 지정된 반복 횟수 동안 value iteration을 실행합니다.

Value iteration은 최적의 value들의 k단계 추정치인 V_k를 계산합니다. runValueIteration과 V_k를 이용해서 다음과 같은 ValueIterationAgent의 메소드들을 구현해야 합니다.

- computeActionFromValues(state): self.values에 의해 제공된 가치 함수(value function)에 따라 최적 action을 계산합니다.
- computeQValueFromValues(state, action): self.values에 의해 제공된 가치 함수에 따라 (state, action) 쌍의 Q-value를 리턴합니다.

위의 값은 모두 GUI에 표시됩니다. 값은 정사각형 안의 숫자, Q-value는 사분면 안의 숫자, 정책(policy)은 각 정사각형에서 나오는 화살표로 표시됩니다.

중요한 점은 "online"이 아닌 "batch" 버전의 value iteration을 사용해야 한다는 것입니다. "batch" 버전 value iteration의 각 벡터 V_k는 고정된 벡터 V_{k-1}로부터 계산됩니다. 즉, k 번째의 state의 value가 successor state의 value를 기반으로 업데이트될 때, 업데이트 계산에 사용되는 successor state의 값은 $k-1$ 번째의 값이어야 합니다. 이는 일부 successor state가 k 번째에서 이미 업데이트된 경우에도 해당됩니다.

[참고사항]

- 깊이 k 에서 생성된 정책은 원래 time step k 에서의 보상(reward)을 반영해야 하지만, 실제로는 time step $k+1$ 에서의 보상을 반영합니다(즉 π_{k+1}). 마찬가지로, Q-value도 Q_{k+1}를 리턴합니다.
  정책 π_{k+1}을 리턴해야 합니다.
- MDP에서 사용 가능한 action이 없는 state도 처리해야 합니다. (이 부분이 향후 보상 계산에 어떤 의미가 있는지 생각해 보세요.)
- 필요하다면 util.py의 util.Counter 클래스를 사용할 수 있습니다. 이 클래스는 기본값이 0인 dictionary입니다. 실제 원하는 argMax가 객체에 없는 키일 수 있음에 유의하세요.

아래 명령어로 구현한 value iteration agent를 테스트하세요.

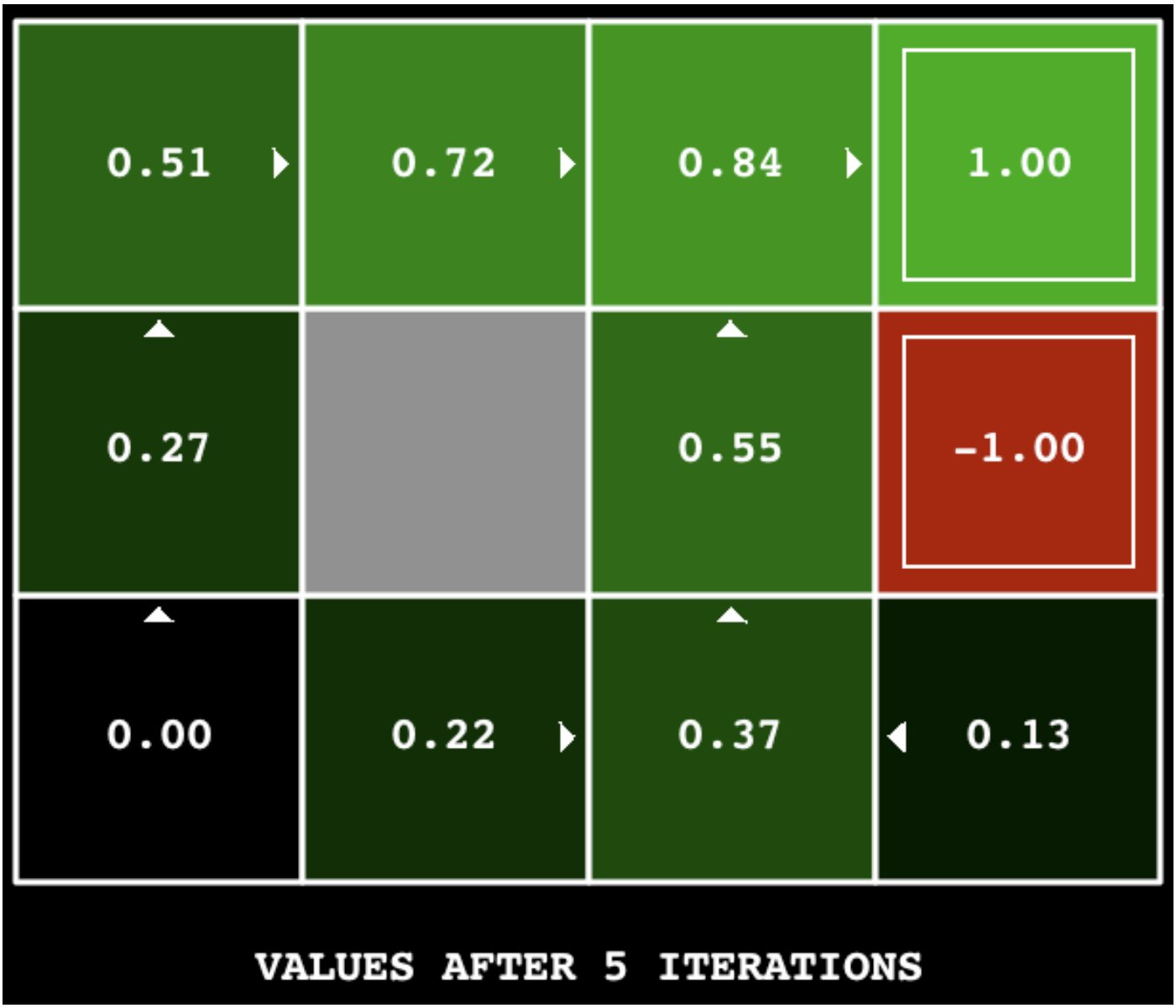
```
python autograder.py -q q1
```

아래 명령어는 여러분이 구현한 ValueIterationAgent를 불러오고 해당 정책을 10번 실행합니다. 키를 누르면 value 와 Q-value, 시뮬레이션이 순환됩니다. GUI에서 V(start)로 표시되어 있는 시작 state의 value와 10번의 실행 후 출력되는 경험적으로 얻은 평균 보상이 아주 비슷해야 합니다.

`python gridworld.py -a value -i 100 -k 10`

[힌트] 기본 세팅인 `BookGrid` 에서 5 번의 반복을 거쳐서 value iteration 을 실행하면 다음과 같은 출력이 생성되어야 합니다:

`python gridworld.py -a value -i 5`



여러분이 구현한 value iteration agent 는 새로운 그리드 환경에서 채점되며, 일정 수만큼 고정된 반복 횟수를 거쳐서 value 값이 수렴했을 때 출력된 value 그리고 Q-values, 정책들을 확인합니다.

Q2. Bridge Crossing Analysis (1점)

BridgeGrid 는 보상이 낮은 종료 state 와 보상이 높은 종료 state 가 양쪽에 높은 음수의 보상을 가진 좁은 "다리"로 분리된 Gridworld 맵입니다. 이 환경에서 agent 는 낮은 보상 state 근처에서 시작합니다.
Discount 매개변수와 noise 의 기본 설정 값인 0.9 와 0.2 로 만들어진 최적 정책은 다리를 건너지 못합니다. 여러분은 discount 매개변수와 noise 중 하나만 변경해서 최적의 정책으로 하여금 agent 가 다리를 건널 수 있게 해야 합니다.
analysis.py 의 question2()를 구현하세요. 여기서 noise 는 agent 가 action 을 수행할 때 의도하지 않은 state 로 전이하는 확률입니다. 기본값은 다음과 같습니다.

```
python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2
```

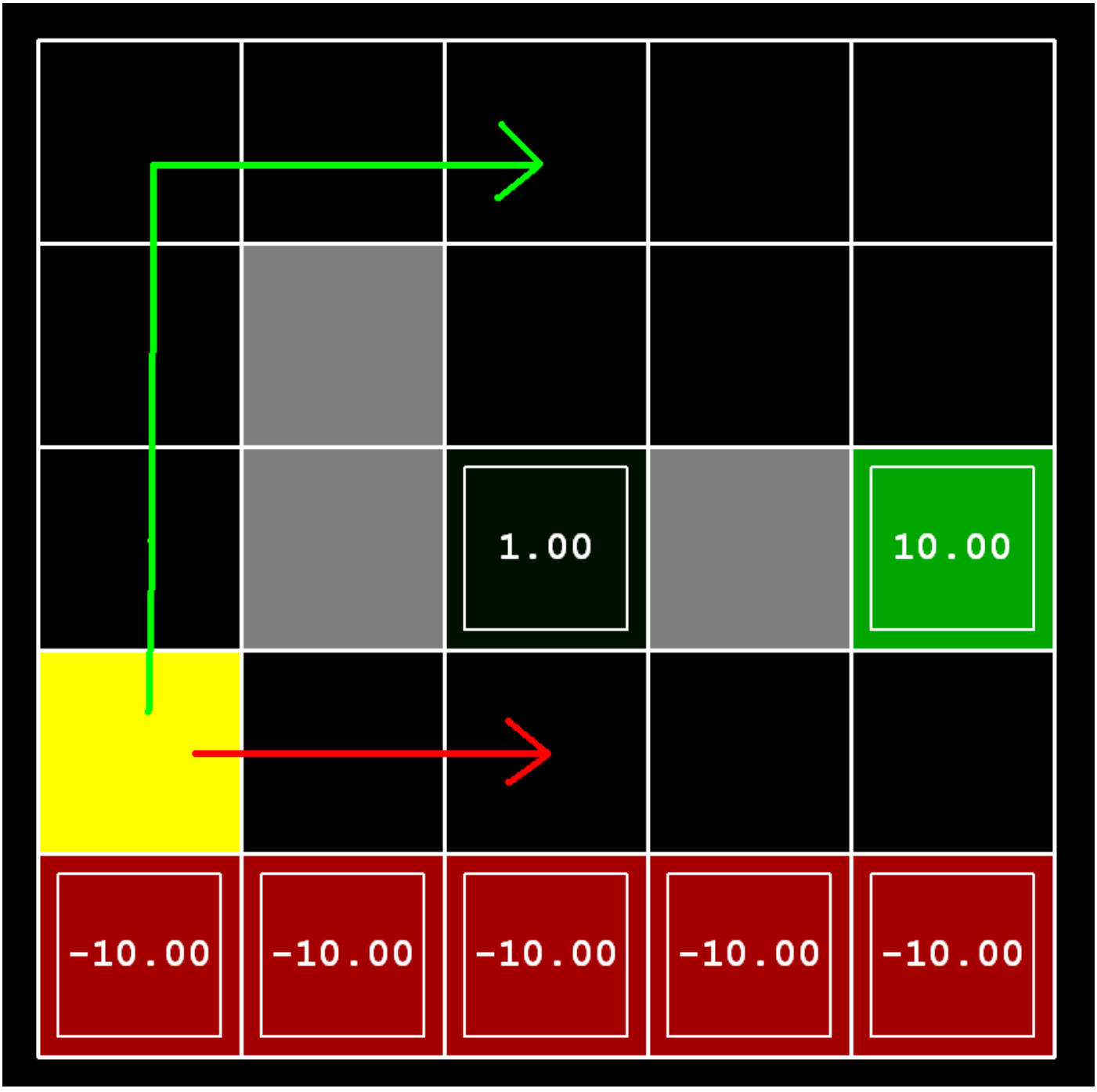
채점에서는 주어진 매개 변수 중 하나만 변경했는지 그리고 이 변경으로 인해 value iteration agent 가 다리를 잘 건너는 지를 확인합니다. 아래 명령어로 작성한 코드가 자동 채점기를 통과하는지 확인하세요.

```
python autograder.py -q q2
```

Q3. Policies (6점)

이 문제에서는 아래 그림과 같은 DiscountGrid 레이아웃을 고려합니다. 이 그리드에는 중간행에 양수 값의 보상을 가진 두 개의 종료 state 가 있습니다. 하나는 +1 의 보상이 있는 가까운 출구이며, 다른 하나는 +10 의 보상이 있는 먼 출구입니다. 또한 맨 아래 행은 -10 의 보상을 가진 종료 state 로 구성되어 있습니다. 빨간색으로 표시된 맨 아래 행을 "절벽" 영역이라고 하겠습니다. 이 환경에서의 시작 state 는 노란색 정사각형입니다.

이 Gridworld 에서는 두 가지 유형의 경로를 생각할 수 있습니다. 첫 번째는 "절벽 위험을 감수하는 경로"로 그리드의 맨 아래 행 근처를 따라 이동하는 경로입니다. 이러한 경로는 짧지만 큰 음수의 보상을 얻을 위험이 있으며 아래 그림에서 빨간색 화살표로 표시되어 있습니다. 두 번째로 "절벽을 피하는 경로"는 그리드의 위쪽 가장자리를 따라 이동하는 경로로, 이 경로는 길지만 큰 음수의 보상이 발생할 가능성이 적습니다. 이 경로는 아래 그림에서 녹색 화살표로 표시되어 있습니다.



이 문제에서 여러분은 이 MDP 에 대한 discount 및 noise, living reward 매개변수들을 설정하여 다양한 유형의 최적 정책을 만들어야 합니다. 각 매개변수 값을 설정할 때 agent 가 어떠한 noise 에도 영향을 받지 않고 최적의 정책을 따른다면 아래 주어진 특정 행동을 달성할 수 있어야 합니다. 만약 어떤 매개변수 설정에 따라 특정 행동이 달성될 수 없다면 'NOT POSSIBLE' 문자열을 반환하여 정책을 적용할 수 없음을 보여야 합니다.

다음은 여러분이 생성하려고 시도 해야 하는 최적의 정책 유형입니다.

- 절벽(-10)의 위험을 감수하면서 가까운 출구(+1)를 선호하는 유형
- 절벽(-10)을 피하는 동시에 가까운 출구(+1)를 선호하는 유형
- 절벽(-10)의 위험을 감수하면서 먼 출구(+10)를 선호하는 유형
- 절벽(-10)을 피하는 동시에 먼 출구(+10)를 선호하는 유형
- 출구와 절벽을 모두 피하는 유형(즉, 에피소드가 종료되지 않아야 함)

각 매개변수의 집합이 어떻게 동작하는지 확인하려면 아래 명령어를 실행하세요.

`python gridworld.py -g DiscountGrid -a value --discount [YOUR_DISCOUNT] --noise [YOUR_NOISE] --livingReward [YOUR_LIVING_REWARD]`

아래 명령어를 실행하여 작성한 코드가 자동 채점기를 통과하는지 확인하세요. 각 경우에 원하는 정책이 반환되는지 확인합니다.

`python autograder.py -q q3`

`analysis.py` 파일에 있는 `question2a()`부터 `question2e()`는 각각 `(discount, noise, living reward)`의 3 개 항목으로 이루어진 튜플을 반환해야 합니다.

[참고사항]

- 여러분은 GUI를 통해 정책을 확인할 수 있습니다. 예를 들어, 3(a)에 대한 올바른 답을 사용하면 (0,1)의 화살표는 동쪽을 가리켜야 하며, (1,1)의 화살표도 동쪽을 가리켜야 하고, (2,1)의 화살표는 북쪽을 가리켜야 합니다.
- 일부 기기에서는 화살표를 볼 수 없을 수 있습니다. 이 경우 키보드의 버튼을 눌러 qValue 화면으로 전환하고 각 state의 사용 가능한 Q-value의 argmax를 취하여 정책을 계산하세요.

Q4. Prioritized Sweeping Value Iteration[Extra Credit] (1점)

이 문제는 추가점수를 위한 문제입니다. 이 문제에서는 valueIterationAgents.py에 부분적으로 작성된 PrioritizedSweepingValueIterationAgent를 구현합니다. Prioritized sweeping은 정책을 변화시킬 가능성이 높은 방식으로 state 값을 업데이트하는데 초점을 맞추려고 시도합니다.

이 문제에서 여러분은 'Memory-based Reinforcement Learning: Efficient Computation with Prioritized Sweeping(NIPS1992)' 논문에서 설명된 standard prioritized sweeping 알고리즘의 간소화된 버전을 구현합니다. (논문을 직접 읽을 필요는 없습니다) 과제에서는 다음과 같이 단순화된 알고리즘을 사용합니다. 먼저 state s의 predecessors를 정의해야 합니다. 이는 어떤 action a를 수행하여 state s에 도달할 확률이 0이 아닌 모든 state로 정의됩니다. 또한, 매개변수로 전달되는 theta는 state 값을 업데이트할 때 오차를 허용하는 정도를 나타냅니다

구현할 때 따라야 할 알고리즘은 아래와 같습니다.

```
1.  Compute predecessors of all states
2.  Q ← Initialize an empty priority queue. // 자동 채점기가 잘 작동하기 위해서는
3.  for each non-terminal state s, do:                  // self.mdp.getStates()가 반환하는 순서대로
4.      s_current ← self.values[s]              // 반복문을 실행
5.      s_highest ← highest Q-value across all possible actions from s
6.      diff ← |s_current – s_highest|                  // self.values[s] 여기서 업데이트하지 마세요.
7.      Push s into Q with priority -diff
8.      for iteration in 0,1,⋯,self.iteration-1, do:
9.          if Q.isEmpty: terminate
10.         s ← Q.pop()
11.         if not isTerminalState(): update the value of s in self.values
12.         for each predecessor p in s, do:
13.             p_current ← current value of p
14.             p_highest ← highest Q-value across all possible actions from p
15.             diff ← |p_current – p_highest|// self.values[p] 여기서 업데이트하지 마세요.
16.             if diff > theta:
```

구현 시 참고할 사항은 다음과 같습니다.

- state의 predecessors를 계산할 때에는 중복을 피하기 위해서 list가 아닌 set에 저장해야 합니다.
- 구현 시, util.PriorityQueue의 update 메서드를 사용하면 편리합니다. 해당 메서드의 공식 설명을 참고하세요.

구현을 테스트하려면 아래 명령어를 통해 자동 채점기를 실행하세요. 실행에 1초 정도 걸립니다. 시간이 오래 걸리면 과제 후반에 문제가 발생할 수 있으므로 지금부터 효율적으로 구현해야 합니다.
구현한 prioritized sweeping value iteration agent는 새로운 그리드에서 고정된 반복횟수 (1000 iteration) 후 수렴시의 value, Q-value, 정책을 확인해 채점됩니다.

```
python autograder.py -q q4
```

아래 명령어를 사용하면 Gridworld 환경에서 PrioritizedSweepingValueIterationAgent를 실행할 수 있습니다.

```
python gridworld.py -a priosweepvalue -i 1000
```

Q5. Q-Learning. (5점)

Value iteration agent는 실제로 경험을 통해 학습하지 않습니다. 대신에 실제 환경과 상호작용하기 전에 MDP 모델을 고려하여 완전한 정책에 도달합니다. Value iteration agent는 환경과 상호작용할 때, 단순히 미리 계산된 정책을 따르기만 하면 됩니다(reflex agent가 됩니다). 이 차이는 Gridworld와 같은 시뮬레이션 환경에서는 중요하지 않을 수 있지만, MDP 사용이 불가능한 실제 세계에서는 매우 중요합니다.

이 문제에서는 Q-learning agent를 구현합니다. 이 agent는 객체가 생성됐을 때 거의 아무것도 하지 않고, 대신 update(state, action, nextState, reward)를 이용한 환경과의 상호 작용을 통해 시행착오를 거치며 학습합니다. Q-learner의 기본구조는 qlearningAgents.py의 QLearningAgent에 작성되어 있으며, '-a q' 옵션으로 선택할 수 있습니다. 이 문제에서 여러분은 update, computeValueFromQValues, getQValue 및 computeActionFromQValues 메서드를 구현해야 합니다.

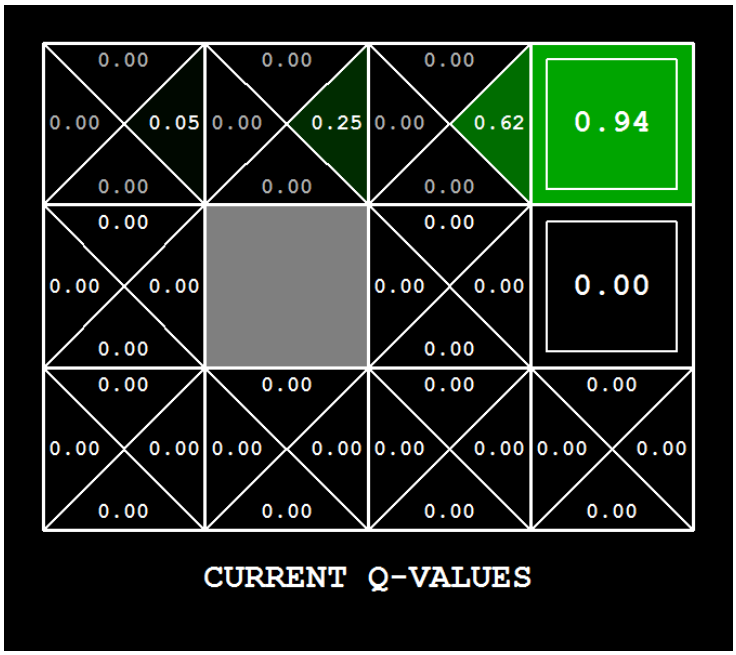
[참고사항]

- computeActionFromQValues에서는 더 좋은 행동을 위해 무작위로 action을 선택해야 하며, random.choice() 함수를 이용할 수 있습니다. Agent가 이전에 탐험하지 않은 state과 action에 대한 Q-value는 0입니다. 만약, agent가 경험한 모든 action들에 대한 Q-value가 음수라면 탐험하지 않은 action이 최적의 action 일 수 있습니다.
- **[중요]** ComputeValueFromQValue 및 ComputeActionFromQValue 메서드에서 Q-value에 접근할 때는 반드시 getQValue 메서드를 사용해야 합니다.

Q-learning update을 구현했다면, 아래 명령어로 키보드를 사용하여 수동으로 제어하면서 Q-learner가 학습하는 것을 확인할 수 있습니다.

```
python gridworld.py -a q -k 5 -m
```

-k는 agent가 학습할 수 있는 에피소드 수를 제어하기 위한 옵션입니다. 올바르게 코드를 작성했다면, 수동으로 최적의 경로를 따라 팩맨을 북쪽으로 이동한 다음 동쪽으로 이동하는 것을 4개의 에피소드 동안 반복하면 아래 그림과 같은 Q-value를 얻을 수 있습니다.



`--noise 0.0` 옵션을 사용해 noise를 제거하면 디버깅에 도움이 될 수 있습니다.

아래 명령어를 실행하여 작성한 코드가 자동 채점기를 통과하는지 확인하세요. 주어진 예제에서 Q-learning agent가 reference Q-learning agent와 동일한 Q-value와 정책을 학습하는지 확인합니다.

```
python autograder.py -q q5
```

Q6. Epsilon Greedy (2점)

`getAction`에서 epsilon-greedy action selection을 구현해서 Q-learning agent를 완성하세요. 이는 epsilon 비율만큼 무작위로 action을 선택하고, 그 외의 경우에는 현재 최적의 Q-value를 따르는 것입니다. 무작위로 선택한 action이 최적의 action 일 수도 있으므로, best action을 제외한 action 중 임의로 action이 선택되는 것이 아니라, action들 중 임의로 하나가 선택되어야 합니다.

리스트에서 원소를 균일하게 임의로 선택하기 위해서 `random.choice` 함수를 이용합니다. `p`의 확률로 `True`를 반환하고 `1-p`의 확률로 `False`를 반환하는 `util.flipCoin(p)`를 사용하면 성공 확률이 `p`인 이진 변수를 시뮬레이션 할 수 있습니다.

`getAction`를 구현한 후, 아래 명령어로 `GridWorld`에서 agent의 행동을 관찰해보세요. (이때, epsilon은 0.3입니다.)

```
python gridworld.py -a q -k 100
```

최종적으로 얻은 Q-value는 특히 잘 탐험(exploration)된 경로에서 value iteration agent와 유사해야 합니다. 그러나 무작위 action과 초기 학습 단계로 인해 평균 반환값은 Q-value가 예측하는 것보다 낮습니다.

이제, 다양한 epsilon 값에 대해 시뮬레이션해 봅시다. Agent 행동이 예상과 일치하는지 확인해보세요.

```
python gridworld.py -a q -k 100 --noise 0.0 -e 0.1
```

```
python gridworld.py -a q -k 100 --noise 0.0 -e 0.9
```

아래 명령을 실행하여 여러분이 구현한 코드를 테스트해보세요.

```
python autograder.py -q q6
```

또한, 추가로 코드를 작성하지 않아도 Q-learning crawler robot을 수행할 수 있습니다.

```
python crawler.py
```

만약 작동하지 않는다면, 여러분이 작성한 코드가 `GridWorld` 문제에만 최적화되어 작성된 것일 수 있으므로, 모든 MDP에 적용할 수 있도록 더 일반적인 코드로 작성해야 합니다.

Q7. Bridge Crossing Revisited (1점)

먼저, 아래 명령어를 통해 기본으로 설정 되어있는 학습률을 가지고 완전한 랜덤 Q-learner를 50 에피소드 동안 noise 없는 BridgeGrid에서 훈련시키고 그 결과 최적 정책을 찾는지 살펴보세요.

```
python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1
```

이제 epsilon을 0으로 설정하여 같은 실험을 시도하세요. 이제 50번의 반복 후 최적 정책을 학습할 가능성이 99% 이상인 epsilon과 학습률이 있는지 찾아보세요.
analysis.py의 question8( ) 함수는 튜플 (epsilon, 학습률)을 반환하거나 문자열 'NOT POSSIBLE'을 반환해야 합니다. Epsilon과 학습률은 각각 -e, -l 로 제어할 수 있습니다.

구현이 action 선택을 위해 동점 해결 방식에 의존하면 안 됩니다. 예를 들어, 전체 그리드 월드를 90도 회전시켜도 답이 올바르게 나와야 합니다.

아래 명령어를 실행하여 자동 채점기를 실행하세요.

```
python autograder.py -q q7
```

Q8. Q-Learning과 팩맨 (1점)

이제 팩맨 게임을 해볼 시간입니다. 팩맨은 훈련과 테스트 두 단계로 게임을 진행합니다. 첫 번째 훈련 단계에서는 위치와 action들의 value들에 대해 학습합니다. 아주 작은 그리드에서도 정확한 Q-value들을 학습하는데 매우 오랜 시간이 걸리기 때문에 기본 모드에서는 훈련과정이 GUI나 콘솔로 보이지 않습니다.
훈련이 끝나면 테스트 모드로 전환됩니다. 테스트시 팩맨의 self.epsilon과 self.alpha는 0.0으로 설정되어 Q-learning과 탐험이 중단되고 학습한 정책에 따라 행동하게 됩니다. 테스트 게임은 훈련과정과 달리 기본적으로 GUI에 표시됩니다.
코드 변경 없이 아래 코드를 실행하여 아주 작은 그리드에서 Q-learning 팩맨을 실행할 수 있어야 합니다.

```
python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid
```

PacmanQAgent는 여러분이 구현한 QLearningAgent를 기반으로 작성되어 있습니다. 대신에 PacmanQAgent는 팩맨 문제에 더 효과적인 기본 학습 매개변수로 구현되어 있습니다(epsilon=0.05, alpha=0.2, gamma=0.8).
위 명령어가 예외 없이 작동하고 agent가 80% 이상 승리하면 이 문제에서 만점을 받을 수 있습니다. 자동 채점기는 2000번의 훈련 게임과 100번의 테스트 게임을 적용합니다.

[힌트] 만약 QLearningAgent가 gridworld.py 및 crawler.py에 대해 작동하지만 smallGrid에서 팩맨이 좋은 정책을 학습하지 못하는 것 같다면 이는 getAction 및 computeActionFromQValues 메서드가 이전에 탐험하지 않은 action을 잘 활용하지 못해서일 수도 있습니다.
특히, 이전에 탐험하지 않은 action이 0으로 설정되어 있고 탐험한 action들이 -1로 설정되어 있다면 탐험하지 않은 action이 최적일 수 있습니다. util.Counter의 argMax 함수에 주의하세요.

아래 명령어를 실행하여 자동 채점기를 실행하세요.

```
python autograder.py -q q8
```

[참고사항]

- 학습 매개변수에 대한 실험을 하고 싶다면 -a 옵션을 사용할 수 있습니다. (-a epsilon=0.1,alpha=0.3, gamma=0.7). 이 값들은 agent 내에서 self.epsilon, self.gamma 및 self.alpha로 읽을 수 있습니다.
- 총 2010개의 게임이 실행될 것이지만, 처음 2000개의 게임은 -x 2000 옵션으로 인해 훈련용으로 지정되어 있으므로 출력이 표시되지 않습니다. 훈련 게임 수도 옵션 numTraining으로 agent에 전달됩니다.
- 만약 10번의 훈련과정을 직접 관찰하고 확인하고 싶다면 아래와 같은 명령어를 실행하세요.
  python pacman.py -p PacmanQAgent -n 10 -l smallGrid -a numTraining=10

훈련 중에는 100 게임마다 팩맨의 상황 통계가 출력됩니다. 훈련 중에는 epsilon이 양수이므로 팩맨은 좋은 정책을 학습한 후에도 무작위 탐험을 하기 때문에 유령 쪽으로 이동할 수 있고, 따라서 성능이 안 좋을 수 있습니다. 보통 팩맨이 100 에피소드 세그먼트에 대한 보상이 양수가 되는 데는 1000-1400 게임 정도가 소요됩니다. 훈련이 끝날 때까지 이 보상은 양수로 유지되어야 하며, 100-350 정도로 높은 값이여야 합니다.

팩맨의 훈련이 끝나면 학습한 정책을 활용하기 때문에 테스트 게임에서 매우 안정적으로(약90% 이상) 승리할 수 있습니다.

하지만 보기에는 단순한 mediumGrid에서 동일한 agent를 훈련시키면 잘 동작하지 않습니다. 훈련 내내 보상이 음수를 유지하며, 시간도 굉장히 오래 걸리는데, 모든 테스트 게임에서 패배할 정도로 나쁜 결과를 보여줍니다.

이는 각 보드 구성이 별도의 Q-value를 가진 별도의 state이기 때문입니다. 즉 현재의 접근 방식은, 어떠한 위치에 있더라도 '유령에게 접근하는 것이 나쁘다'는 것을 일반화할 수 있는 방법이 없습니다.

Q9. Approximate Q-Learning (4점)

State 특징에 대한 가중치를 학습하는 근사 Q-learning agent를 구현하세요. 이는 여러 state가 동일한 특징을 공유할 수 있습니다. qlearningAgents.py 파일에서 PacmanQAgent의 서브 클래스인 ApproximateQAgent 클래스를 구현하세요.

근사 Q-learning은 (state, action)에 대한 함수 $f(s,a)$ 가 있다는 것을 전제로 하며, 이는 특징 값 벡터 $[f_1(s,a),\cdots,f_i(s,a),\cdots,f_n(s,a)]$를 생성합니다.
함수 $f(s,a)$ 는 featureExtractors.py에 구현되어 있습니다. 특징 벡터는 Python의 dictionary 클래스와 유사한 util.Counter 객체입니다. 이는 0이 아닌 특징들과 값들의 쌍으로 구성되어 있으며, 생략된 특징의 값은 0입니다. 따라서 각 특징을 나타내는 것은 인덱스가 아니라 dictionary의 키가 됩니다.

근사 Q함수는 다음과 같이 정의 되어있습니다.

$$Q(s,a)=\sum_{i=1}^{n} f_i(s,a)w_i$$

각 가중치 w_i는 특정 특징 $f_i(s,a)$와 관련이 있습니다. 코드에서는 가중치 벡터를 구현할 때 특징을 가중치 값에 매핑하는 dictionary로 만들어야 합니다.
가중치 벡터를 업데이트 하는 방법은 다음과 같이 Q-value를 업데이트하는 방법과 유사합니다.

$$w_i \leftarrow w_i + \alpha \cdot \text{difference} \cdot f_i(s,a)$$
$$\text{difference} = \left(r + \gamma \max_{a'} Q(s',a')\right) - Q(s,a)$$

difference 항은 일반적인 Q-learning과 동일하며, r은 경험한 보상입니다.

기본적으로 ApproximateQAgent는 IdentityExtractor를 사용합니다. 이는 모든 (state, action) 쌍에 하나의 특징을 할당합니다. 즉 IdentityExtractor 특징 추출기를 사용하면 근사 Q-learning agent가 PacmanQAgent와 동일하게 작동합니다. 아래 명령어를 실행하여 테스트해보세요.

```
python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid
```

ApproximateQAgent는 QLearningAgent의 하위 클래스이므로 getAction과 같은 메서드들을 공유합니다. 따라서 QLearningAgent의 메서드에서는 Q-value를 직접 접근하는 대신 getQValue를 호출하여야 합니다. 이렇게 하면 근사 agent에서 getQValue를 재정의할 때 새로운 근사 Q-value가 사용되어 동작합니다.

구현이 완료되었다면 커스텀 특징 추출기를 사용하여 근사 Q-learning agent를 실행하세요. 아래 명령어로 확인해볼 수 있습니다.

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid
```

ApproximateQAgent를 사용하면 훨씬 큰 레이아웃도 해결할 수 있습니다. 아래 명령어를 실행해보세요. 대신 훈련은 몇 분 정도 소요될 수 있습니다.

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumClassic
```

오류가 없다면 여러분이 구현한 Q-learning agent는 간단한 특징 추출기로 50게임만 학습해도 거의 이깁니다.

채점은 여러분이 구현한 근사 Q-learning agent를 실행하고, 동일한 예제가 주어질 때 기준이 되는 참조 구현과 동일한 Q-value와 특징 가중치를 학습하는지 확인할 것입니다. 아래 명령어를 실행하여 자동 채점기를 실행해보세요.

```
python autograder.py -q q9
```