

Project-3: Reinforcement Learning

2015136107 이현진

컴퓨터공학부
2sguswls2s@koreatech.ac.kr

1 Introduction

1.1 Reinforcement Learning

이번 프로젝트는 Reinforcement Learning이다. Reinforcement Learning란 에이전트와 실제 상호작용을 통해서 주변 환경으로부터 행동을 학습하는 것으로, 주변 환경으로부터 영향을 받아 자신이 어떠한 행동을 할지 판단할 수 있도록 학습하는 머신러닝 알고리즘 중 하나이다. 이 알고리즘은 다른 알고리즘과 다르게 실제 환경에서만 학습하기 때문에 컴퓨터로 시뮬레이션 하기 힘든 매개변수들에 대한 처리를 유기적으로 수행할 수 있다. 여기에서는 Q러닝 기법을 이용하여 강화학습을 수행한다는 점이 있다.

해당 문제를 그림으로 표현하면 다음과 같다.

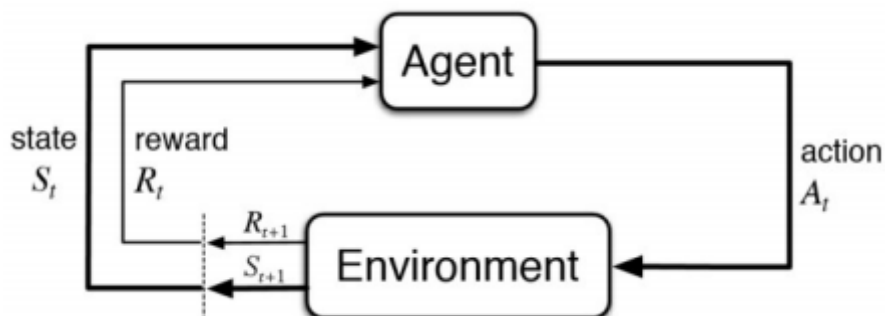


Fig. 1. Reinforcement Learning에서 가장 주로 사용하는 MDP 모델

여기서 MDP 모델이란 의사결정 과정을 확률과 그래프를 이용하여 모델링하는 것으로, 시간 t 에서의 상태는 $t-1$ 의 상태에서만 영향을 받는다는 것으로, 미래를 결정하는 것은 과거가 아닌 현재라는 것을 의미한다.

이번 프로젝트에서 총 10가지의 문제가 존재한다.

- Value Iteration
- Bridge Crossing Analysis
- Policies
- Asynchronous Value Iteration
- Prioritized Sweeping Value Iteration
- Q-Learning
- Epsilon Greedy
- Bridge Crossing Revisited
- Q-Learning and Pacman
- Approximate Q-Learning

위의 문제를 해결하기 위해서 어떠한 해결방안을 사용하고, 수도코드 및 Grading 결과, 결과 및 분석, 느낀점을 기록한 보고서이다.

```
python autograder.py
```

해당 코드를 실행 할 경우 점수를 볼 수 있다.

2 Questions

2.1 Value Iteration

해결방법

이 문제를 해결하기 위해서는 Value Iteration에 대한 이해가 필요하다. Value Iteration는 최적의 MDP의 Policy 및 최적의 value를 구하는 방법(알고리즘)이라고 할 수 있다. 즉, 자신이 현재 상태에서 어떠한 행동을 하느냐에 따라서 리워드가 결정되는데, 해당 리워드가 가장 좋게 나오는 액션을 수행하는 것이다. 결론적으로는 MDP의 최적의 정책(optimal policy)를 구하는데 있어서 가장 대표적인 알고리즘이라고 할 수 있는데, 초기의 상태에서 다음의 상태까지의 최적의 Policy를 구하는 것을 반복함으로써 문제를 해결하는 알고리즘이고, 초기 상태가 변하지 않으며, 반복되는 동안에 값을 구하는

함수로 구한 값이 변하지 않는다. 구한 값이 모일수록 우리가 원하는 최적의 정책을 구할 수 있는 것을 알 수 있다.

좀 더 쉽게 설명하자면, 이전 반복에서의 state값은 다음 반복에서의 state값을 변환시키는데 사용하고, State의 유틸리티(Bellman Equation)는 최적의 Policy가 수행될 때 해당 State에서 시작하여 예상되는 보상을 의미한다. 따라서 각각의 유틸리티를 통해 해당 액션에서의 미래의 보상을 알 수 있는 것이다.

마지막으로 Value Iteration의 방정식은 Bellman Equation에서의 optimal value function의 관계식을 반복적으로 찾아가는 알고리즘인데, Bellman Equation은 Reinforce 알고리즘의 기본 매커니즘인데, 가장 큰 이득값으로 액션을 수행 시 예상되는 전체 가치는 얼마인지 매기는 것을 의미한다.

수도코드

(해결한 알고리즘에 대한 설명 또는 수도코드 작성)

(코드를 캡처해서 첨부하지 말 것)

수도코드 예시)

Value Iteration

Algorithm RunValueIteration

States X

Actions A

Integer iterations

```

For I in range(iterations) do
  Get counter in map
  for state in X do
    max_value = -infinity
    for action in getPossibleAction(A) do
      Compute q_value from Value Iteration Equation(state, action)
      if q_value > max_val then
        max_value = q_value
      endif
    counter[state] = max_value
  endfor
  values = counter
endfor
endfor

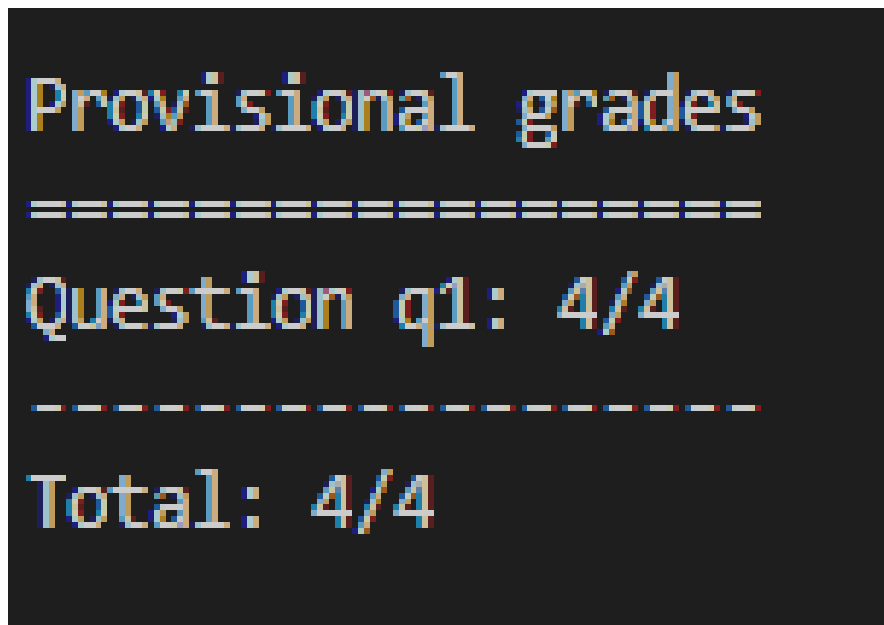
```

수도코드 설명

1. for 문

-
1. Value를 구하는데 몇번 반복할지 결정함.
 2. map에 있는 초기 상태값을 가지고 옴.
 3. for 문
 1. 움직일 수 있는 state 수 만큼 반복 수행함.
 2. max_value 를 구하기 전 초기화를 함. Max_value는 최적의 policy 값임.
 3. for 문
 1. 움직일 수 있는 액션 수만큼 반복을 수행함
 2. Value Iteration Equation에 현재 상태와 움직일 수 있는 액션을 넣음으로써 얻어지는 값을 q_value라고 정의함.
 3. if-then-else 문
 1. q_value가 max_value 값을 넘는다면 q_value값을 max_value로 지정
 4. 최종적인 상태값에 max_value를 집어넣음.
 4. map의 상태값을 갱신함.
-

Grading code 결과



```

Provisional grades
=====
Question q1: 4/4
-----
Total: 4/4
  
```

Fig. 2. Value Iteration Grading code 결과

결과 및 분석

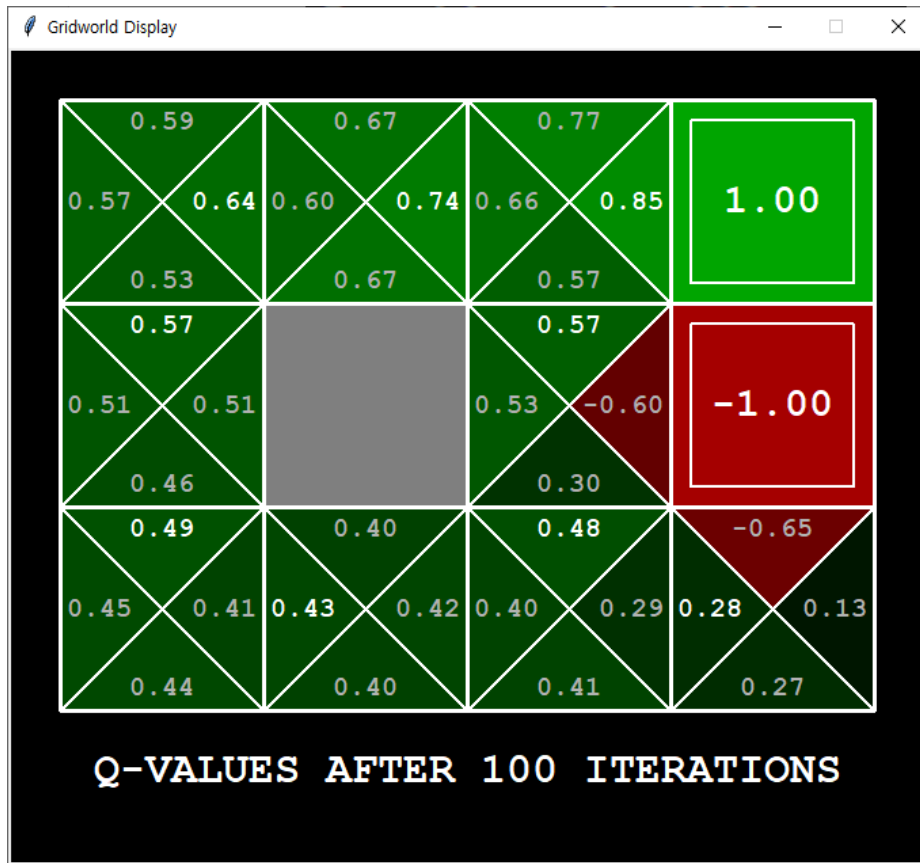


Fig. 3. Value Iteration Q-Values를 100회 수행 결과



Fig. 4. Value Iteration Values를 5회 수행 결과

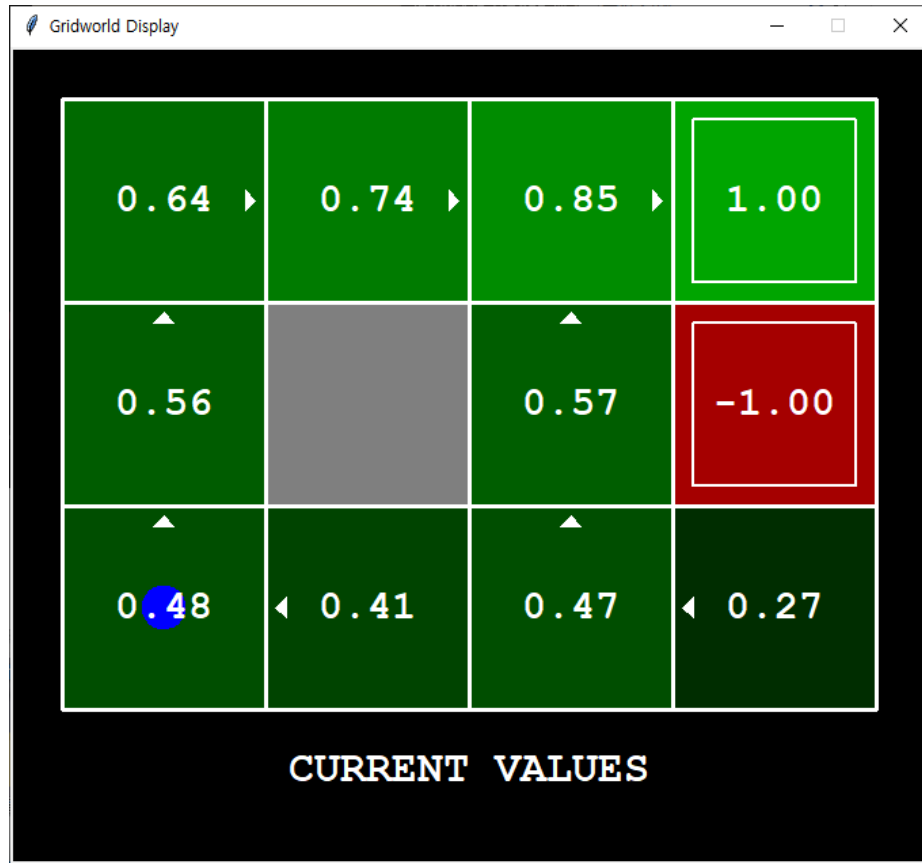


Fig. 5. Value Iteration Values를 10회 수행 결과

이 과제는 1.0에 들어가야하고, -1.0에 들어가면 안되는데 있어서에 대한 판단하는데 있어서 Value Iteration 방정식의 내용에서의 리워드 및 Discount에 대해서 적절하게 설정되어 있기 때문에, 방정식을 적절하게 구현하는 것을 통해서 잘 수행될 수 있음을 볼 수 있었다. 1행 3열을 볼 때, 오른쪽으로 갈 때가 가장 최적의 정책이라고 판단되기 때문에 오른쪽을 선택함을 볼 수 있었다. 그리고 -1.0 주변으로 가는 것은 최악의 선택이기 때문에 가는 방향이 전부 음수가 나옴을 볼 수 있었다. 그리고 반복 횟수에 따라서 달라지기도 하는데, 최적의 Policy를 찾기 위해서 수많은 반복을 수행해야하는 것을 알 수 있었다. 마지막으로 여기서 Q-Values와 Values가 있는데, Values는 가장 큰 유틸리티, 가장 큰 노드를 선택하는 반면에, Q-Value는 state가 아닌 Q-state를 통해서

계산하기 때문에 자신이 가고자 하는 방향,노드가 바뀔 수 있다는 점을 알 수 있었다.

느낀 점

이번과제를 하면서 알 수 있었던 점은 Value Iteration의 원리와 MDP의 원리 등 Reinforcement Learning에서 가장 기본이 되는 것들을 배울 수 있었다. 해당 원리를 통해 밑에 나온 Q러닝을 배워서 다른 문제를 해결 할 수 있어서 공부가 잘 되었다.

2.2 Bridge Crossing Analysis

해결방법

해당 문제를 해결하기 위해서는 map에서 어디로 가야하는지에 대해서 알아야 한다. Map의 모형을 본 후, Discount라는 개념과 Noise에 대한 개념을 알아야하는데, Discount는 보상으로부터 멀어질수록 일정한 수치를 곱함으로써 보상에 대한 기대값을 점점 줄여서 가만히 있지 않도록 만드는 것이다. 그리고 Noise도 있는데, Noise는 해당 액션으로 가지 않고 다른곳으로 움직일 확률을 의미한다. 이 두개의 개념을 이해하여 Map을 보고 문제를 해결하면 된다. 위 map은 맨 오른쪽에 보상이 있지만, 길 옆마다 최악의 보상이 주어져 있기 때문에 옆으로 가지 않도록 하게 만들고, Discount 를 적절하게 사용하여 기대보상을 많이 깎지 않도록 하는 것이 이번 과제의 문제다.

수도코드

Bridge Crossing Analysis	
Discount = 0.9	
Noise = 0.0	
수도코드 설명	
1. 최종 보상구간에서부터 멀어질수록 약간의 할인을 주어 제자리에 안 머물수 있도록 한다.	
2. 다른 방향으로 빠질 확률을 아예 0으로 두어 정확하게 계산된 값으로 갈 수 있도록 한다.	

Grading code 결과

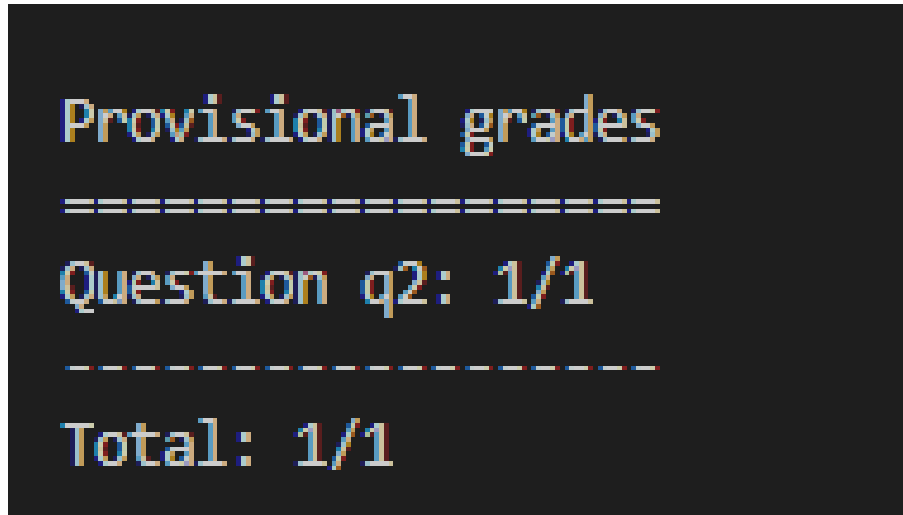


Fig. 6. Bridge Crossing Analysis Grading code 결과

결과 및 분석

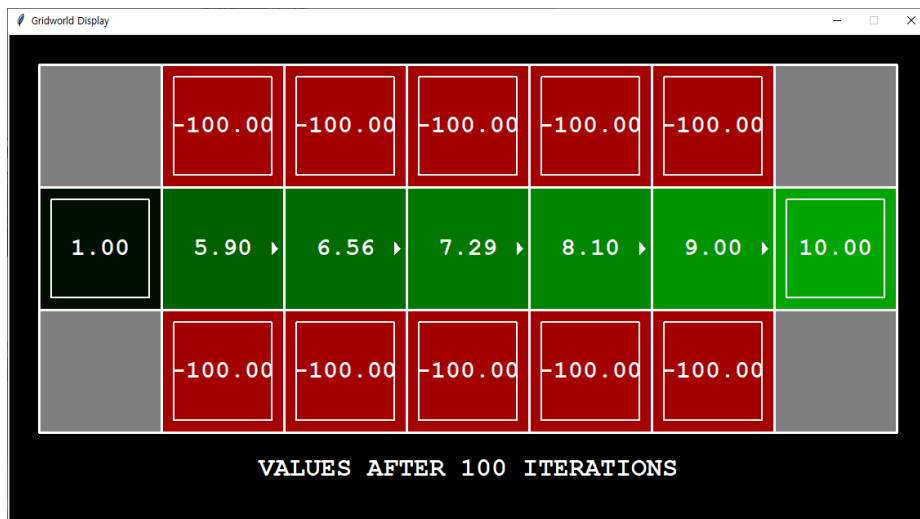


Fig. 7. Bridge Crossing Analysis 결과

해당 문제를 보고 알 수 있었던 점은 적절한 Discount와 Noise 조절을 통해서 최적의 보상을 갈 수 있게 만드는 방법에 대해서 배울 수 있었다. 너무 많은 Dis-

count를 하면 출발값에 있는 1.0보다 적어지게 된다면 1.0으로 다시 되돌아가는 상황이 만들어 질 수 있으므로 적절한 할인이 있어야 하고, -100.0으로 빠지지 않게 하기 위해서는 엉뚱하게 가는 확률을 없앴으로써 확실하게 10.0으로 갈 수 있게 만들었다.

느낀 점

이번 문제를 풀면서 알 수 있었던 점은 할인과 노이즈의 개념을 이해할 수 있었고, 이 개념이 곧 최적의 정책을 만든다는 것을 알 수 있었다.

2.3 Policies

해결방법

2.2에서 배웠던 Discount와 Noise 에서 LivingReward를 추가하여 최적의 정책을 만드는 것이 2.3의 과제이다. 이를 해결하기 위해서는 LivingReward의 개념을 이해해야 하는데, LivingReward는 살아 있을때의 어드벤처지를 주는것으로, 살아있기만 해도 보상을 주거나 보상을 뺀 것을 의미한다. 이를 이용하여 A부터 E까지의 상황을 통해 최적의 정책을 찾는 방법을 수행하는 것이 이번 과제이다.

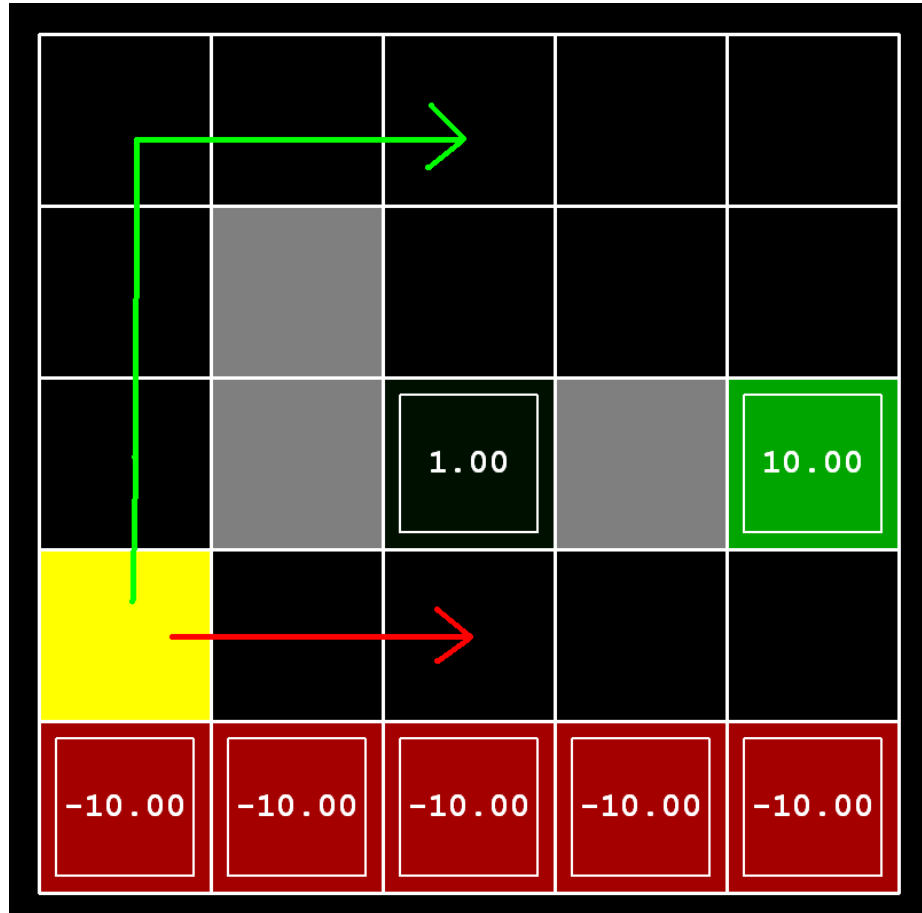


Fig. 8. 이번 과제에서 최적의 정책을 수행해야하는 그림

A부터 E까지의 조건은 다음과 같다.

- A : 절벽(-10)을 무릅쓰는 폐쇄 출구(+1)를 선호한다.
- B : 닫힌 출구(+1)를 선호하지만 절벽(-10)은 피하십시오.
- C : 절벽(-10)을 무릅쓰는 먼 출구(+10)를 선호한다.
- D : 절벽(-10)을 피해 먼 출구(+10)를 선호한다.
- E : 두 출구와 절벽 모두 피하십시오(일화가 종료되지 않도록).

수도코드

Value Iteration

Question A
Discount = 0.4
Noise = 0.0

LivingReward = -0.5

Question B
Discount = 0.4
Noise = 0.2
LivingReward = -0.5

Question C
Discount = 1.0
Noise = 0.0
LivingReward = -0.5

Question D
Discount = 1.0
Noise = 0.2
LivingReward = -0.5

Question E
Discount = 0.0
Noise = 0.0
LivingReward = infinity

수도코드 설명

1. A의 질문의 경우는 “절벽(-10)을 무릅쓰는 폐쇄 출구(+1)를 선호한다.”이기 때문에, 오른쪽으로 가는데 절벽에 빠지면 안되기 때문에 Noise는 있어서는 안되고, 거리가 가까운 곳을 가는 루트이기 때문에 거리에 관해서 할인을 세게 주어 윗길이 아닌 아랫길을 선택할 수 있도록 한다.
 2. B의 질문의 경우는 “닫힌 출구(+1)를 선호하지만 절벽(-10)은 피하십시오.”이기 때문에, 절벽을 회피하기는 하는데 적당히 회피하면 됨으로 약간의 Noise를 추가함으로써 오른쪽으로 가는데 아닌 윗쪽으로도 갈 수 있도록 만든다. 또한 오른쪽으로 가다가 절벽에 떨어지는 변수가 생기기 때문에 윗쪽으로 가게 만듦.
 3. C의 질문의 경우는 “절벽(-10)을 무릅쓰는 먼 출구(+10)를 선호한다.”이기 때문에 절벽 아래 길을 가야하므로 Noise는 없어야하고, 먼 출구를 선호하여야함으로 할인의 수준을 1과 유사하게 하여 1.0과 10.0 부분을 결정하는 부분에서 10.0 방향으로 갈 수 있도록 Discount를 설정한다.
 4. D의 경우 “절벽(-10)을 피해 먼 출구(+10)를 선호한다.” 이므로 윗길을 갈 수
-

있도록 노이즈를 적절히 주고, Discount를 1과 유사하게 주어 10.0으로 갈 수 있도록 만든다. 또한 오른쪽으로 가다가 절벽에 떨어지는 변수가 생기기 때문에 왼쪽으로 가게 만들.

5. E의 경우에는 “두 출구와 절벽 모두 피하십시오(일화가 종료되지 않도록)”의 조건이기 때문에 10.0 과 1.0 보다 살아있을 때의 보상을 크게 주면 문제가 해결된다.

Grading code 결과

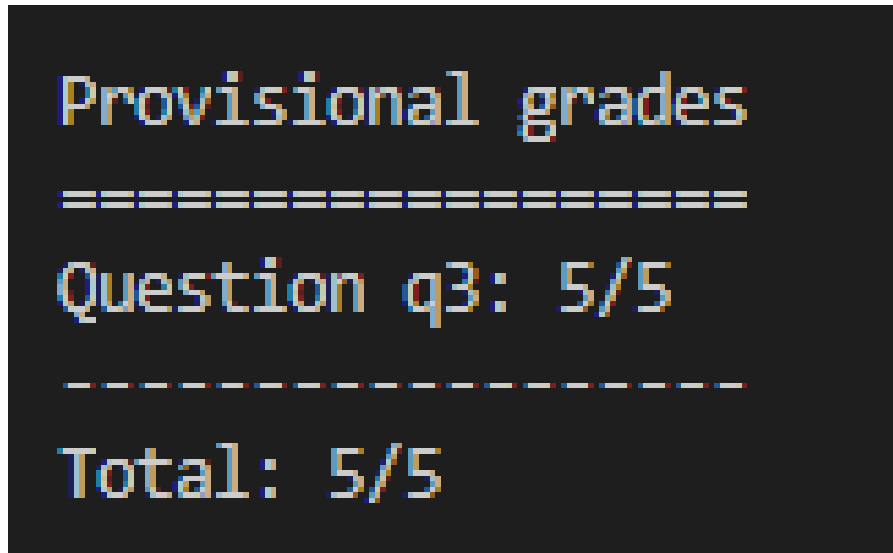


Fig. 9. Policies Grading code 결과

결과 및 분석

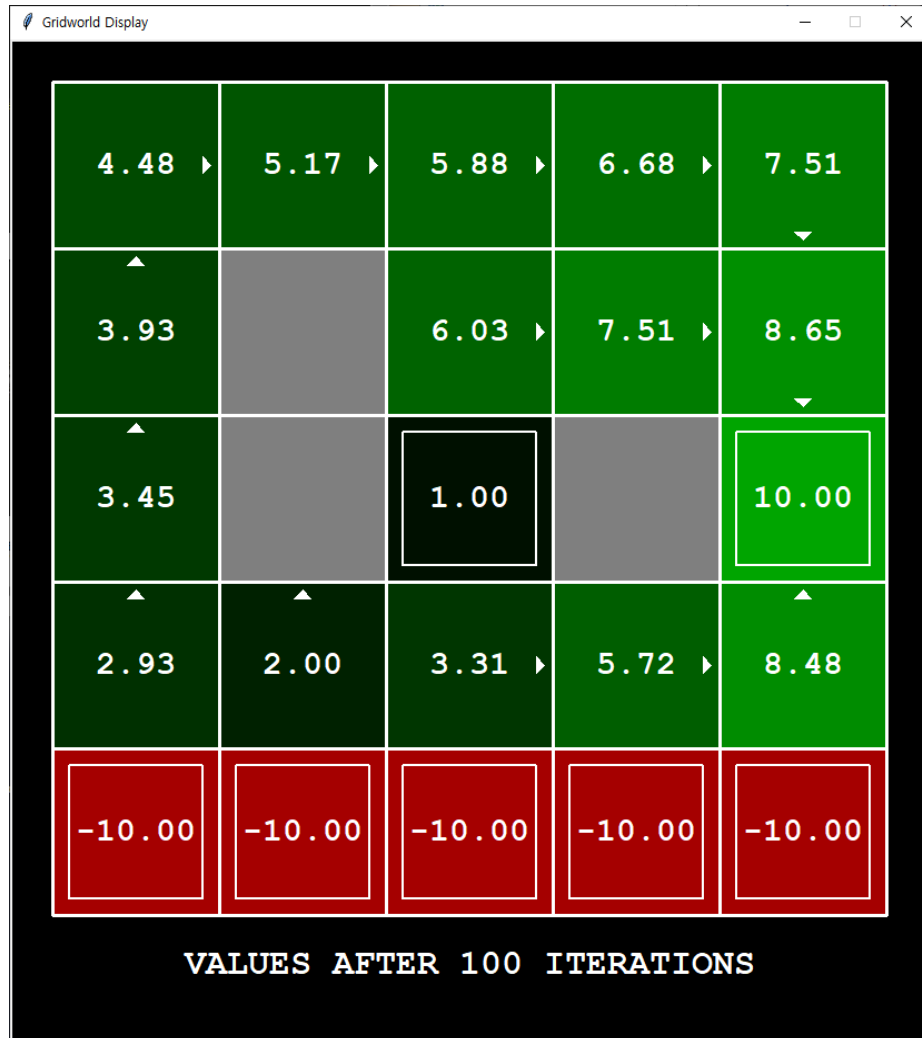


Fig. 10. Policies 결과

위 결과를 보고 알 수 있는 점은 수도코드에서 설명했던 것처럼 최적의 정책을 수행하기 위해서는 할인, 노이즈, 살아있을때의 보상을 줌으로써 적절한 정책을 수행할 수 있음을 알 수 있었다. 위의 사진의 경우는 D에 대한 결과임을 알 수 있다. 새로운 개념인 살아있을때의 보상이라는 개념을 추가하면 다양한 결과가 나올 수 있음을 알 수 있었다.

느낀 점

조금 더 좋은 정책을 위해서 다양한 경우에 대한 처리를 해야한다는 점을 알 수 있었고, 이를 통해 MDP에 대한 개념을 이해하는데 도움이 되었던 과제인 것 같다.

2.4 Asynchronous Value Iteration

해결방법

이 문제는 2.1에 나온 Value Iteration 에서의 확장판으로, 이름에서 볼 수 있듯이 비동기식으로 돌리는 것이기 때문에 해당 리워드가 다음 유틸리티를 구할 때 갱신되는 것이 아니라 직접 구하는 것이기 때문에 수많은 시도가 필요하다는 것을 알 수 있다. 하지만 Value Iteration 보다 데이터를 적게 사용 할 수 있고, 빠르게 계산을 할 수 있다는 장점이 있다.

쉽게 설명하자면, 2.1의 경우는 모든 상태에 대해서 Value를 구하고, 그 값들을 한번에 update 하는 반면, 2.4는 한 상태에 대해서 value를 구해 update를 수행하는 방법이라고 보면 된다. 하지만 2.4에서 사용한 방법의 단점으로는 error가 발생하였을 때에 대한 처리가 부재하다는 점이 있다.

비동기식 Value Iteration을 구현하는데 방법이 3가지가 존재하는데, in-place value iteration, prioritized sweeping, real-time 방법이 존재한다. 여기서는 in-place value iteration을 구현하였다.

마지막으로 중요한 점은 선택된 state 가 터미널, 즉 보상구간이라면 아무 작업도 수행하지 않고 끝내는 것이 중요하다. 왜냐하면 하나하나의 state에 대해서만 계산하기 때문에 끝내주어야 하는 점이 중요하다.

수도코드

Asynchronous Value Iteration

Algorithm In-place value iteration

States X
Map Value
Integer iterations

```
For I in range(iterations) do
  for state[i%length(X)] in X do
    if state is Terminal then
      pass
    else
```

```

    Get Action from computeActionFromValue(state)
    Value[state] = Value Iteration Equation(state, action)
  Endif
Endfor
Endfor

```

수도코드 설명

1. for 문

1. 테스트를 몇번 반복할지 결정함.

2. for 문

1. 각각의 하나의 state에 대해서 반복을 수행함

2. for 문

1. if-then-else문

1. 현재 상태가 터미널이면 모든 것을 끝냄

2. 그게 아니라면 각 정책을 얻어옴.

3. 그리고 Value Iteration Equation을 실행하여 최적의 정책 값을 저장함.

Grading code 결과

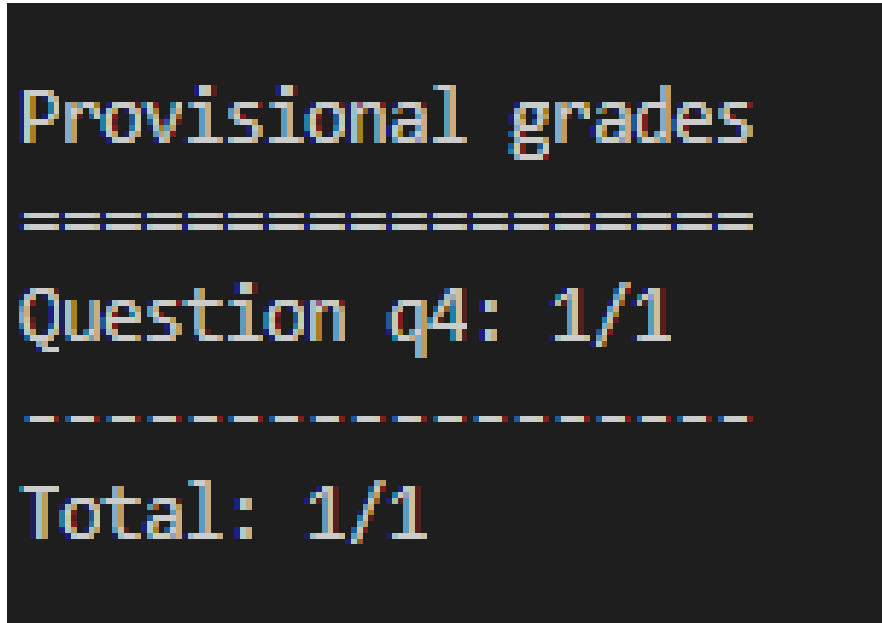


Fig. 11. Asynchronous Value Iteration Grading code 결과

결과 및 분석

(결과 및 분석 내용)



Fig. 12. Asynchronous Value Iteration 결과

이번 결과를 통해 알 수 있었던 점은 2.1의 경우 많은 시간이 걸렸지만, 2.4의 경우는 많은 처리에도 불과하고 2.1과 속도가 유사하게 나와 계산을 수행한다는 점을 알 수 있었다. 하지만 반복횟수가 적은 경우에는 제대로 Value가 정해지지 않아서 실행에 있어서 어려움이 있었음을 알 수 있었다. 하지만 noise로 인해서 제대로 갈 때도 있었다는 점도 알 수 있었다.

느낀 점

이번 과제를 하면서 알 수 있었던 점은 Asynchronous Value Iteration은 Value Iteration에서도 2가지가 존재하는데, 동기적인 방법과 비동기적인 방법이 있었음을 알 수 있었고, 이로 인한 각각의 장점을 알 수 있었다. 그리고 이 두가지의 방법이 결론적으로는 최적의 정책을 보여줄 수 있었다.

2.5 Prioritized Sweeping Value Iteration

해결방법

이번 과제는 Prioritized Sweeping을 구현하는 것인데 이는 비동기식 Value Iteration의 방법 중 하나로, Bellman error가 크게 남아 있는 states를 백업하여 큐에 집어넣고, 우선순위를 주어 처리하도록 하는 방법이다. 즉, 각각의 state에 방문횟수가 많아질수록 업데이트가 최적화가 되어 최적의 정책을 내올 수 있지만, 방문횟수가 낮으면 error가 커지기 때문에 더 많이 업데이트를 해주는 것이다.

이를 수행하기 위해서는 error가 발생한 state들을 저장하기 위한 큐가 존재해야 하고, 그리고 error가 많이 나온 state를 임시 저장할 배열이 하나 존재해야 한다. 이를 통해서 error가 발생한 state에 대해서 한 번 더 계산을 함으로써 2.4에서의 단점이었던 error 처리에 대한 부재를 해결함으로써 더 빠르게 계산할 수 있는 장점이 존재한다.

수도코드

Prioritized Sweeping Value Iteration

Algorithm Prioritized Sweeping Value Iteration

States X

Queue Q

Map Value

State List L

Actions A

Integer iterations

Exponential theta

for state in X **do**

for action in A **do**

for success in getTransitionStateAndProb(state, action)

if success in L **then**

 L[success].add(state)

Else

 L[success] = {state}

Endif

Endfor

Endfor

Endfor

```

for state in X do
  if state is Terminal then
    pass
  else
    action = computeActionFromValues(state)
    maximum = Value Iteration Equation(state, action)
    diff = abs(maximum - value[state])
    Q.push(state, -diff)
  endif
Endfor

```

```

For I in range(iterations) do
  If Q is empty then
    return
  endif
  state = Q.pop()
  if state is Terminal then
    pass
  else
    action = computeActionFromValues(state)
    values[state] = Value Iteration Equation(state, action)
  endif

  for list in L do
    if state is not Terminal then
      action = computeActionFromValues(state)
      maximum = Value Iteration Equation(state, action)
      diff = abs(maximum - value[state])
      if diff < theta then
        Q.update(list, -diff)
      Endif
    Endif
  endfor
Endfor

```

수도코드 설명

1. **for** 문
 1. state의 개수만큼 반복함
 2. **for** 문
 1. 액션의 개수만큼 반복함
 2. **for** 문
 1. 현재 상태에서 해당 액션을 하였을때의 다음 상태의 확률 개수만큼
-

반복

2. if-then-else 문

1. 만약 현재 상태가 리스트에 있으면 $L[\text{success}]$ 에 현재상태를 추가함.
2. 그렇지 않다면 현재상태 1개만 $L[\text{success}]$ 에 있도록 변화시킴.

2. for 문

1. state의 개수만큼 반복함.

2. if-then-else 문

1. 만약 현재 state가 터미널이면 코드 종료
2. 그렇지 않다면 이동할 수 있는 action을 구하고, 해당 액션을 이용해 방정식을 사용하여 maximum 값을 구하고, $\text{abs}(\text{maximum} - \text{value}[\text{state}])$ 의 값을 구하여 큐에 해당값에 음수를 취해 현재상태와 함께 집어넣음.

3. for문

1. 사용자가 정해진 개수만큼 반복수행

2. if-then 문

1. 큐가 비어있으면 error가 없는 것으로 종료시킨다.

3. state는 큐에 있는 상태로 가져옴.

4. if – then – else 문

1. state가 터미널이면 코드 종료
2. 그렇지 않다면 액션을 구하여 방정식을 사용하여 나온 값을 values에 집어넣음.

5. for문

1. 리스트 L의 개수만큼 반복 수행

2. if – then – else 문

1. 현재 상태가 터미널이 아니라면 이동할 수 있는 action을 구하고, 해당 액션을 이용해 방정식을 사용하여 maximum 값을 구하고, $\text{abs}(\text{maximum} - \text{value}[\text{state}])$ 의 값을 구하여 큐의 내용을 list와 -diff로 갱신함.
-

Grading code 결과

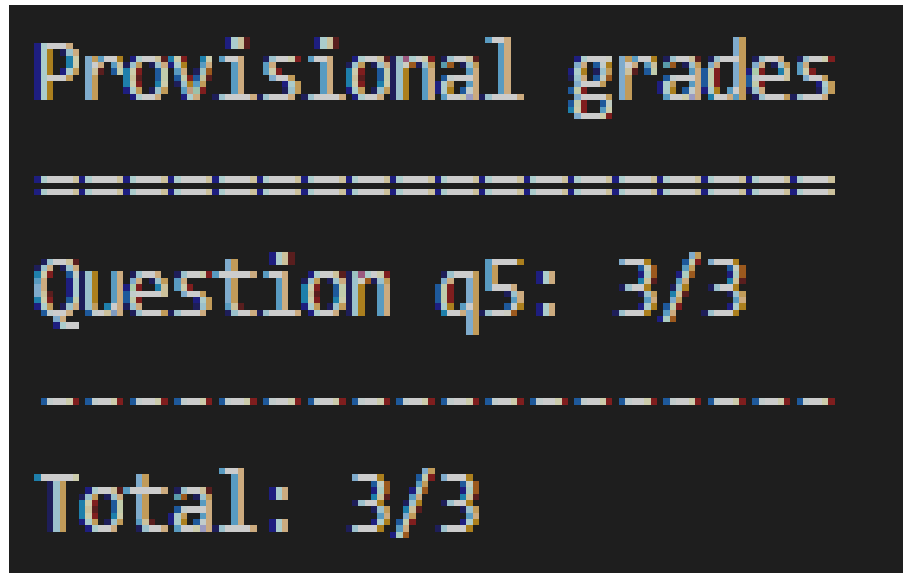


Fig. 13. Prioritized Sweeping Value Iteration Grading code 결과

결과 및 분석

(결과 및 분석 내용)

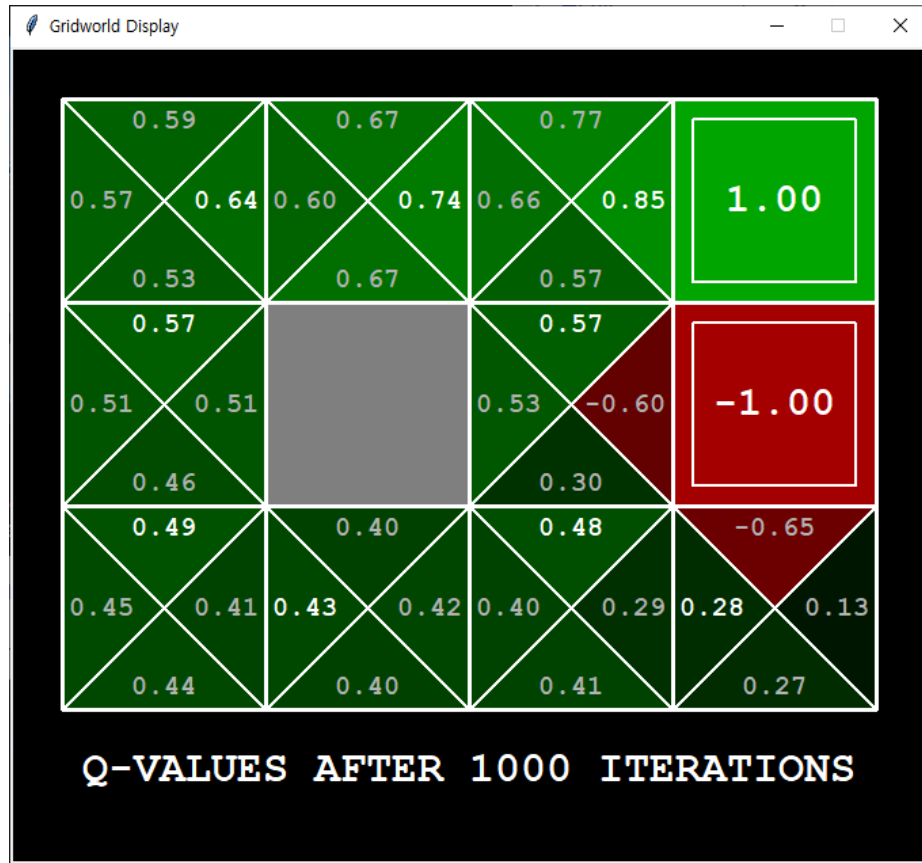


Fig. 14. Prioritized Sweeping Value Iteration 결과

해당 결과를 보고 알 수 있었던 점은 2.4에서 구현하였던 코드보다는 복잡하였지만 속도면에서 매우 빨라진 것을 알 수 있었다. 1000번을 바로바로 하는 것을 보아 엄청난 속도개선이 있었던 것 같다. 결과도 보면 2.4와 동일함을 알 수 있었는데, 이는 알고리즘의 차이일뿐, Value Iteration의 결과는 동일함을 알 수 있었다.

느낀 점

이번 과제를 하면서 알 수 있었던 점은 다양한 Value Iteration이 존재함을 알 수 있었다. 동기식과 비동기식 외에도 비동기식에서도 다양한 방법이 존재함을 알고 놀랐던 과제였다.

2.6 Q-learning

해결방법

이문제를 해결하기 위해서는 Q-learning에 대한 이해가 필요하다. 이전의 Value Iteration은 실제 환경에서 돌리는 것이 아니다. 환경에 적응하여 결과를 내는 것 같지만 결국엔 환경과 상호작용하기 전에 최적의 정책이 나오기 때문에 실제 환경과 상호작용을 수행할 때 별도의 Reflex Agent로 구분을 시킨다. 이는 시뮬레이션에서는 별 차이가 없어도, 실제에서는 MDP를 사용할 수 없기 때문에 실제환경에서의 별도의 구분이 필요하다. 결론적으로는 모델을 기반으로 수행하는 알고리즘이었다.

하지만 Q-learning은 모델 없이 환경만으로 학습하는 강화학습 알고리즘이다. Agent가 특정 상황에서 특정 행동을 하라는 최적의 정책을 배우는 것으로, 현재 상태에서부터 시작되어 차근차근 단계를 거치다보면 예측 보상의 예측값이 극대화된다. 따라서 다른 상태에 대한 Discount 및 Noise가 변하는 환경에서도 변형없이 적용이 가능하다는 점이 있다.

결론적으로는 환경이 바뀌더라도(직접 조작을 통해서 나오는 데이터를 기반으로) Agent가 정상적으로 움직일 수 있도록 구현하는 것이 이번 목표이다.

수도코드

Q-learning

Algorithm Q-learning

```

States X
Queue Q
Map Value
Actions A
Integer maximum = -infinity
Action next_action

Function computeActionFromQValues(state) {
  For action in A do
    If q_value > maximum then
      Maximum, next_action = q_value, action
    Endif
  Endfor
  Return next_action
}
```

```

Function computeValueFromQvalues(state) {
```

If not action then

Return 0.0

Endif

Return max([getQValue(state, action) for action in A])

}

Function update(state,action, nextState, reward) {

Sample = reward + discount * getValue(nextState)

q_value = (1-alpha) * getQValue(state, action) + alpha * sample)

}

수도코드 설명

1. computeActionFromQValue 설명

1. for 문

1. 현재 수행가능한 액션수만큼 반복

2. if - then - else 문

1. 만약 가져온 q_value가 maximum보다 크면 maximum과 nextState는 각각 q_value와 현재 state 가 됨

3. 다음에 수행할 액션 nextState를 반환

2. computeValueFromQvalues 설명

1. if-then 문

1. 만약 수행가능한 액션이 없으면 0.0 반환

2. 수행 가능한 액션들 중 가장 큰 QValue 값을 가지는 값을 반환

3. update 설명

1. Sample 값을 지정해줌. Q-Value 식을 이용한 계산식임.

2. Sample 값을 이용해서 q_value 값을 구하여 업데이트 수행.

Grading code 결과

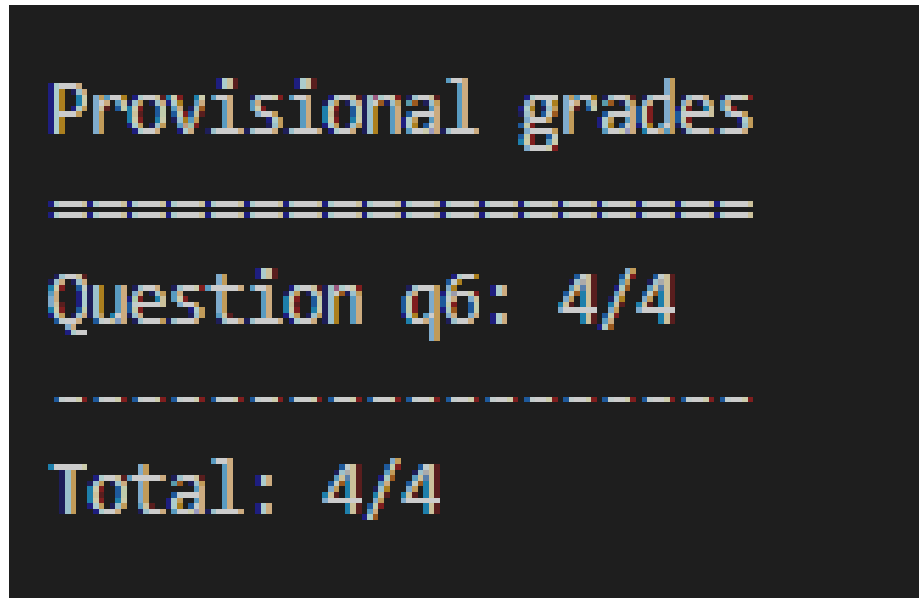


Fig. 15. Q-learning Grading code 결과

결과 및 분석

(결과 및 분석 내용)

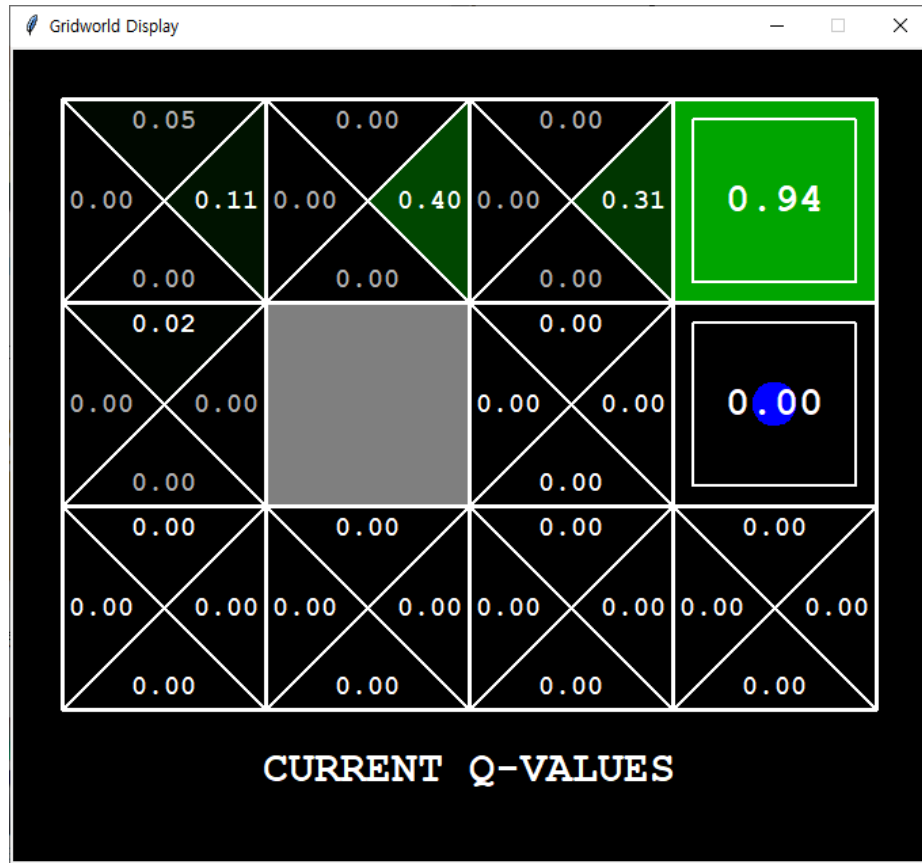


Fig. 16. Q-learning 결과

실행해본 결과 처음부분에는 사용자가 직접 움직여줌으로써 보상에
다다르게 되는데 Agent는 해당 보상이 좋은 보상인지 나쁜보상인지 모르는
상태이기 때문에 아무리 사용자가 좋은 보상쪽으로 가려고 해도 위의 사진과
같이 탐험을 하는 Agent를 볼 수 있었다. 이를 통해서 환경이 어떤지 탐험하고,
좋은 길인지 나쁜길인지 서서히 배우는 Agent를 볼 수 있었다.

느낀 점

이번 과제를 하면서 다양한 알고리즘이 존재한다는 것을 알 수 있었고,
MDP로는 환경에 적응하지 못한다는 점은 좀 충격이 컸다. 왜냐하면 MDP로 할

때는 모델을 정의한다는 생각을 하지 못했지만 Discount와 Noise 등이 환경을 설정해주는 모델이라는 것을 깨달았을 때는 조금 충격이었다.

2.7 Epsilon Greedy

해결방법

이 문제 또한 해결하기 위해서는 Epsilon Greedy 알고리즘을 이해해야 한다. 기존의 Greedy 알고리즘에서 더욱 강화한 것으로, Greedy 알고리즘에서 부족하였던 탐색이 충분하게 이뤄지지 않는 문제를 개선시켜서 입실론이라는 Hyper-Parameter를 통해 행동을 결정하는 것이다. 입실론은 Random으로 됨으로써 탐색을 활용할 수 있다. 즉, 가장 좋은것만 선택하던 Greedy 선택을 줄이고 나머지 확률을 랜덤으로 정함으로써 이 문제를 해결한다.

수도코드

Epsilon Greedy

Algorithm Epsilon Greedy

States X

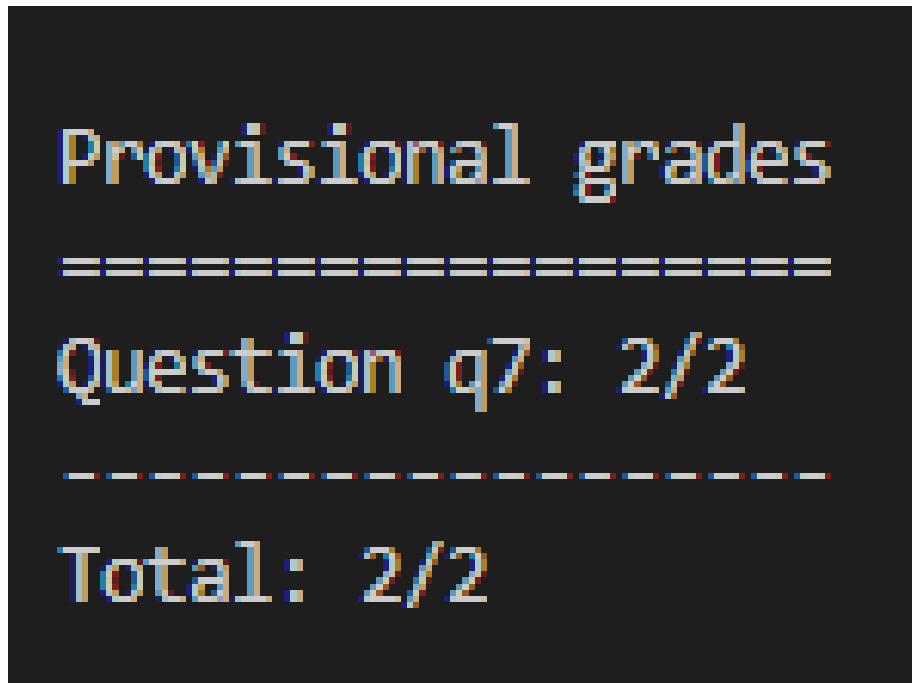
Actions A

Probability epsilon

```
Function getAction(state) {
  Action in Actions
  If epsilon is up then
    Action = Rand(A)
  Else
    Action = getPolicy(Action)
  Endif
  Return Action
```

수도코드 설명

1. Action의 값은 실행 가능한 Actions의 값 중 하나임
 2. if then else 문
 1. 만약 epsilon이 앞면 확률인 경우 랜덤으로 액션을 갱신시킴
 2. 그렇지 않다면 최적의 정책을 가지고 옴.
 3. 액션을 반환함.
-

Grading code 결과**Fig. 17.** Epsilon Greedy Grading code 결과**결과 및 분석**

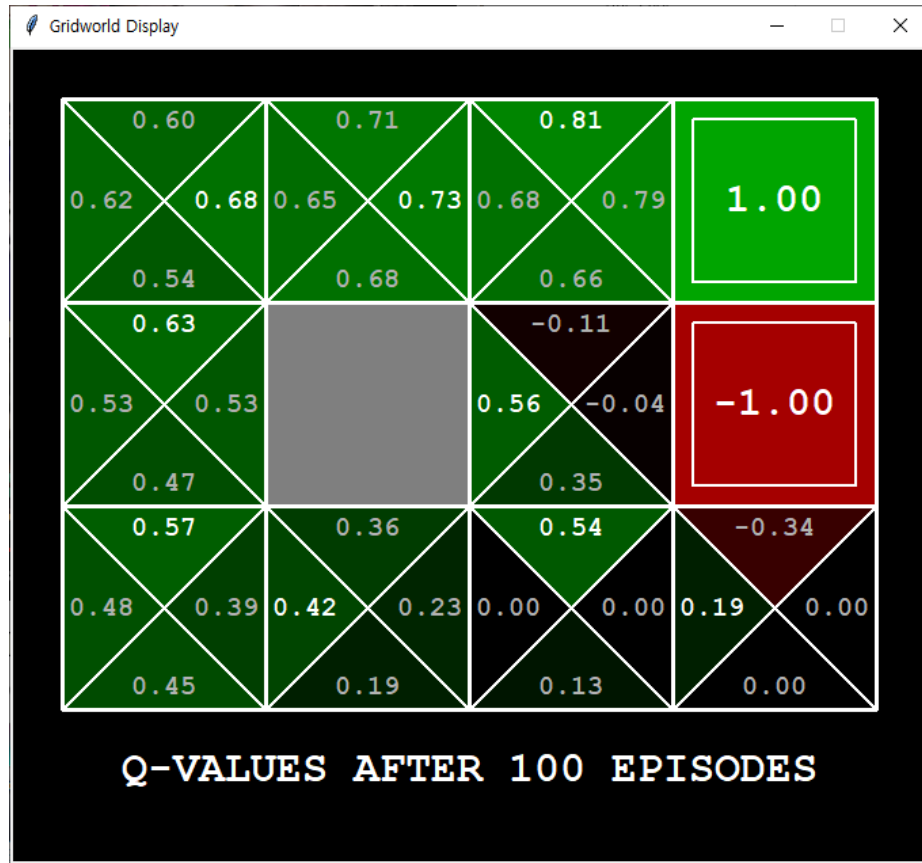


Fig. 18. Epsilon Greedy 결과

해당 프로그램을 실행했을 때 알 수 있었던 점은 평소에 나오던 Q-Value 값이 나오지 않고 다른 방향을 제시해주는 모습을 보였다. 이러한 결과가 나온 이유는 최적의 값만 찾는 것이 아니라 다른 방향으로도 가는 선택을 수행했기 때문에 각각에 대해서 예측보상값이 달라짐을 보여주었다. 이 프로그램을 다시 실행하였을 때, 엡실론이라는 확률을 기반으로 random 하게 액션을 수행하는 경우가 생기기 때문에 어느쪽으로 갈지 모르는 것이다. 하지만 이러한 선택이 최적의 정책보다 좋을 때가 있는데, 보통 우리가 미로에서 어떻게 이동해야할지 모를 때에는 최적의 정책보다는 우연적인 이동이 이득을 챙겨줄 때가 있다는 뜻이다.

느낀 점

이 문제를 해결하면서 알 수 있었던 점은 무조건 탐욕적인 검색보다는 적절한 운에 따른 이동이 탐욕적으로 찾는 것 보다 이득이 생길 수 있다는 것을 배울 수 있었다. 역시 운은 대단하다고 생각이 들었다.

2.8 Bridge Crossing Revisited

해결방법

일단 해당 문제를 풀기 이전에 해당 맵을 관찰 하고 나서 답을 결정해야한다. 해당 맵을 보았을 때, 아무것도 알 수 없고, 어느 길로 가야 제일 큰 이득인지 확인할 수 없는 상황에서 학습을 수행하여 정답을 찾는 것이 이번 목표이다. LearningRate 와 epsilon이 주어졌는데, epsilon을 통해서 확률을 적절하게 조정하고, learningRate를 통해서 얼마나 학습을 수행할지에 대해서 결정하는데, 너무 많지도 않고, 너무 적지도 않게 학습을 시켜야한다.

수도코드

Bridge Crossing Revisited	
Epsilon = none	
LearningRate = none	
Return 'NOT POSSIBLE'	
수도코드 설명	
1. 해당 문제는 해결할 수 없는 문제이다.	

Grading code 결과

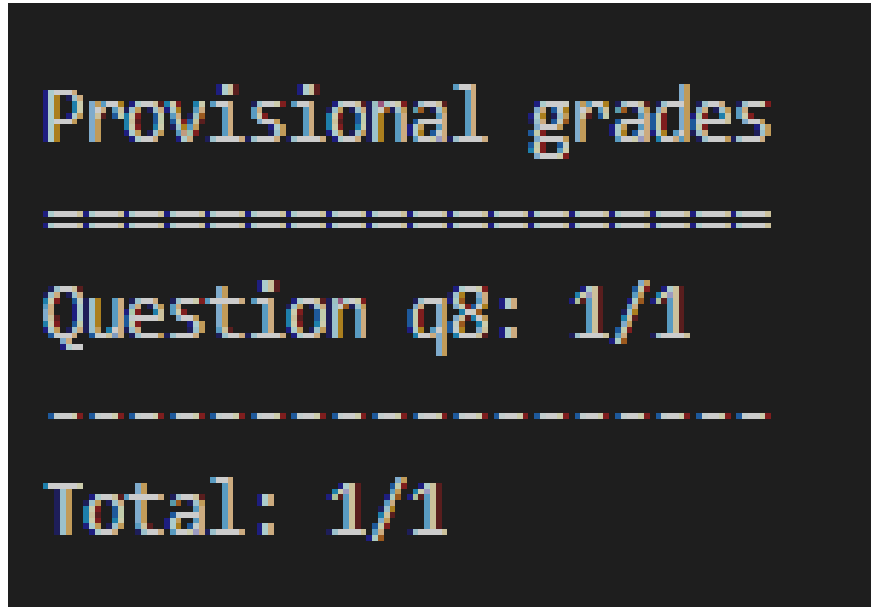


Fig. 19. Bridge Crossing Revisited Grading code 결과

결과 및 분석



Fig. 20. Bridge Crossing Revisited 50회 수행 결과

출발점에도 어느정도의 보상이 있는데, 이로 인해서 출발점과 그 주변의 최악의 보상들이 존재하기 때문에 맨 끝에 있는 최고의 보상에 도달하기 전에 적절한 보상에 만족을 하고 다시 출발지점으로 돌아가는 현상이 발견되었다.

이번 코드에서는 문제를 해결할 수 없는데 그 이유는 아무리 학습률을 높여봤자 적절한 학습률이 최고의 정책을 내놓는데, 횟수가 많아질수록 더 많이 배우게 됨으로 점점 성능이 안좋아지게 된다. 그리고 Epsilon으로 해결하려고 해도 너무 낮은 확률로 최고의 보상에 도달하기 때문에 결국엔 절벽에 떨어지거나 처음부분에 도달하게 되는 결과가 나오게 된다.

느낀 점

이번 과제를 하면서 알 수 있었던 점은 아무리 작은 보상이 초기위치와 가까이 존재한다면 어떻게 학습시켜도 결국에는 초기위치로 돌아가버린다는 것을 알 수 있었다. 이를 통해 학습시킬 때 어떠한 알고리즘을 써야할지 고민을 많이 해야겠다고 생각했다.

2.9 Q-learning and Pacman

해결방법

이번 과제는 실제 팩맨에서의 Q-learning을 실행시켜보는 과제이다. 이 과제를 해결하기 위해서는 QlearningAgent의 getAction 부분을 수정하는 것으로 문제를 해결해야한다. 이 문제는 위에서 2.7의 Epsilon Greedy를 해결하기위해 이미 코드를 작성했으므로 간단하게 해결 할 수 있다.

수도코드

Q-learning and Pacman

Algorithm Q-learning and Pacman

States X

Actions A

Probability epsilon

Function getAction(state) {

 Action in Actions

If epsilon is up **then**

 Action = Rand(A)

Else

```

    Action = getPolicy(Action)
Endif
    Return Action
}

```

수도코드 설명

1. Action의 값은 실행 가능한 Actions의 값 중 하나임
 2. if then else 문
 1. 만약 epsilon이 앞면 확률인 경우 랜덤으로 액션을 갱신시킴
 2. 그렇지 않다면 최적의 정책을 가지고 옴.
 3. 액션을 반환함.
-

Grading code 결과

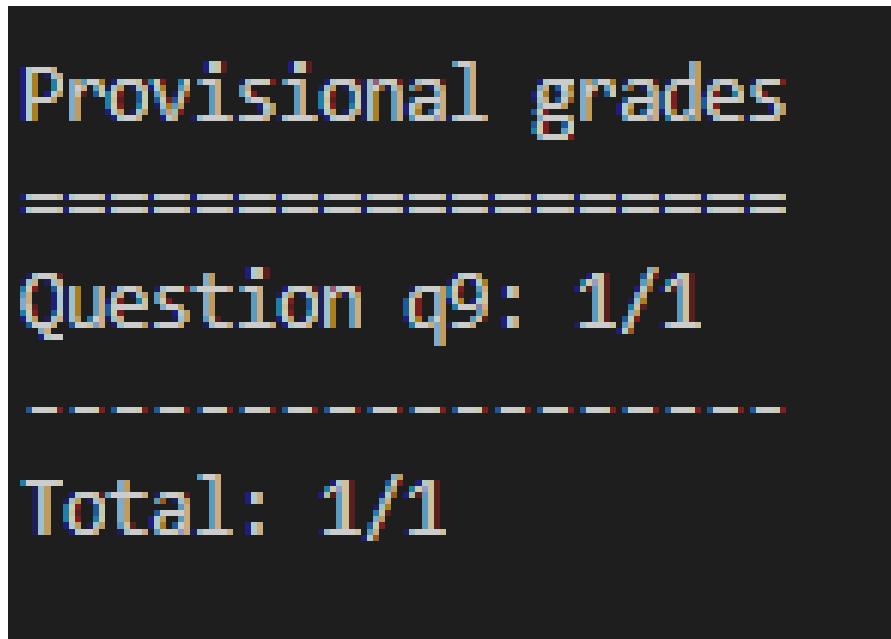


Fig. 21. Q-learning and Pacman Grading code 결과

결과 및 분석

(결과 및 분석 내용)

```
Reinforcement Learning Status:
    Completed 100 test episodes
    Average Rewards over testing: 500.60
    Average Rewards for last 100 episodes: 500.60
    Episode took 1.05 seconds
Average Score: 500.6
Scores:      495.0, 503.0, 503.0, 503.0, 503.0, 503.0, 503.0,
503.0, 503.0, 503.0, 499.0, 495.0, 499.0, 503.0, 503.0, 503.0,
.0, 495.0, 503.0, 499.0, 495.0, 503.0, 503.0, 495.0, 499.0, 49
, 503.0, 503.0, 499.0, 499.0, 503.0, 495.0, 503.0, 503.0, 503
Win Rate:      100/100 (1.00)
Record:      Win, Win, Win, Win, Win, Win, Win, Win, Win, Win,
in, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
n, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\q9\grade-agent.test (1 of 1 points)
***      Grading agent using command: python pacman.py -p Pac
***      100 wins (1 of 1 points)
***      Grading scheme:
***      < 70: 0 points
***      >= 70: 1 points
```

해당 결과를 보고 알 수 있었던 점은 2000번의 훈련 끝에 평균 -37점 정도로 점점 준수해진 훈련결과를 볼 수 있었고, 실제로 100번의 에피소드 결과 모두 잘 통과함을 볼 수 있었다. 이를 통해서 여러 에피소드를 돌려본 후, 가장 최적의 에피소드를 기억함으로써 다음의 학습의 경험을 쌓아서 좋은 결과를 출력함을 알 수 있었다.

느낀 점

이번 문제는 풀다보니깐 자연스럽게 풀린 문제이다. Q-learning을 짤 때, 구체적이고 어려운 코드보다 단순하고 일반적인 코드를 작성하여 문제를 해결하였다. 이 때문인지 처음에는 바로 죽는 모습을 보였지만 학습하면 할수록 열심히 움직이는 팩맨을 보니 기분이 좋았다.

2.10 Approximate Q-learning

해결방법

해당 문제를 해결하기 위해서는 사이트에 나와있는 내용을 잘 읽어야 문제를 해결 할 수 있다. 사이트에서 제시해준 방정식들을 이용하여 해당 방정식이 어느 부분에 들어가야하는지에 대해서 고민한 후, 문제를 해결하면 된다.

다음 방정식은 다음과 같다.

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

Fig. 24. 대략적인 Q-Value 함수 방정식

$$\begin{aligned} w_i &\leftarrow w_i + \alpha \cdot difference \cdot f_i(s, a) \\ difference &= (r + \gamma \max_{a'} Q(s', a')) - Q(s, a) \end{aligned}$$

Fig. 25. Q-Value를 구하는 함수 방정식

수도코드

Approximate Q-learning

Algorithm Q-learning

States X
Queue Q
Map Value
Features F

Function getQValue(state,action) {
For feature in F **then**
 Total += Weight[feature] * features[feature]
Endfor
Return Total
}

Function update(state,action, nextState, reward) {
diff = (reward + discount * getValue(nextState) – getQValue(state, action))
For feature in F **then**
 Weight[feature] += alpha * diff * features[feature]
endfor
}

수도코드 설명

1. getQValue 설명

1. 해당 팩맨의 벡터값을 가져와 가중치와 곱하고 그 값들은 시그마한 값을 반환해준다.

2. update 설명

1. 사이트에 나와있는 방정식을 구현하여 diff를 구한 후, 각각의 가중치를 업데이트 해준다.

Grading code 결과

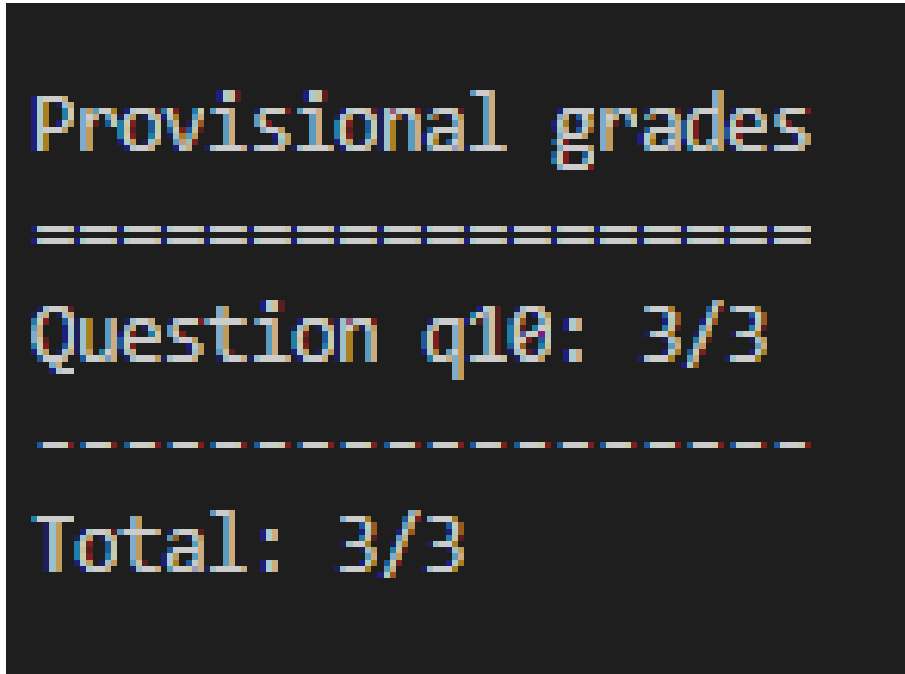


Fig. 26. Approximate Q-learning Grading code 결과

결과 및 분석

```
Pacman died! Score: -510
Pacman died! Score: -510
Pacman died! Score: -506
Pacman died! Score: -506
Pacman died! Score: -514
Pacman died! Score: -506
Pacman died! Score: -510
Pacman died! Score: -506
Pacman died! Score: -514
Pacman died! Score: -506
Average Score: -508.8
Scores:      -510.0, -510.0, -506.0, -506.0, -514.0, -506.0, -510.0, -506.0, -514.0, -506.0
Win Rate:    0/10 (0.00)
Record:      Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

Fig. 27. Approximate Q-learning 결과

해당 문제를 보고 알 수 있었던 점은 Q러닝과 유사하게 돌아가긴 했지만, 대략적인 Q러닝이다 보니 제대로 학습이 되지 않아서 모두 패배를 한 모습을 볼

수 있었다. 뭔가 코드를 잘못짚건가 솔직히 잘 모르겠다. Grader는 점수가 잘 나왔는데 학습시켰을 때에는 실패하여서 이유를 모르겠다.

느낀 점

이번 문제는 솔직히 왜 있는지 솔직히 몰랐었다. 하지만 완전하게 해당 시나리오대로 돌아가는 Q러닝대로 짤 수 없기 때문에 해당 Q러닝의 방정식을 이용한 다른 알고리즘을 구현하라는 의미로 받아들여졌다.

3 Conclusion

느낀 점

이번 과제를 하면서 알 수 있었던 점은 인공지능은 너무 어렵다는 생각을 많이 하였다. 솔직히 수업을 들으면서 이해하면서 하는 경향이 있었지만, 수도코드를 짤 수 없는 과제 내용들도 있어서 어떻게 짜야할지 고민을 매우 많이 했었다. 이번에도 역시 열심히 공부해야겠다는 생각밖에 들지 않았던 과제다.

어려웠던 점

마지막 문제가 학습이 제대로 안되는 이유가 모르겠다. 사이트에 나온 내용대로 짰는데 사이트에서 해보라는 코드를 실행해보아도 잘 되지 않았다.

제안점

이번과제에서 문제를 푸는 구간에서는 수도코드가 필요하지 않은 것 같다.

References

1. Exploration and Exploitation, <https://brunch.co.kr/@chris-song/62>
2. Learning rate, <https://bioinformaticsandme.tistory.com/130>
3. Q-learning wikipedia, <https://en.wikipedia.org/wiki/Q-learning>