Project-4: Ghostbusters

2015136107 이현진

컴퓨터공학부
2sguswls2s@koreatech.ac.kr

1 Introduction

1.1 GhostBusters

이번 프로젝트에서는 저번 프로젝트에서 만들었던 단순히 유령을 피하는 프로젝트가 아닌 직접적으로 유령을 잡아먹는 알고리즘을 이용할 생각이다. 이 때, 센서를 사용하여 보이지 않는 유령의 위치를 찾아 먹는 프로젝트를 찾는 것이고, 유령을 찾는데 있어서 유추하여 효율적인 탐색을 통해 여러 개의 고스트를 사냥하는 방법을 알아볼 것이다.

GhostBusters에서의 목표는 보이지 않는 유령을 사냥하는 것이다. Pacman은 항상 풍부한 센서정보를 얻으며, 각 유령은 소리를 내어 Pacman에게 유령이 있을 법한 위치, 맨해튼거리를 제공하게 됩니다. 이 때, 확률분포를 이용하여 다양한 법칙들을 이용하여 가장 유령이 있을법한 곳에 Pacman이 이동 할 것이고, 목표를 이룰 것이다.

2 Questions

2.1 DiscreteDistribution Classgent

해결방법에 대한 소스 코드

```
def normalize(self):
    """
    Normalize the distribution such that the total value of all keys sums
    to 1. The ratio of values for all keys will remain the same. In the case
    where the total value of the distribution is 0, do nothing.

    >>> dist = DiscreteDistribution()
    >>> dist['a'] = 1
    >>> dist['b'] = 2
    >>> dist['c'] = 2
    >>> dist['d'] = 0
    >>> dist.normalize()
    >>> list(sorted(dist.items()))
    [('a', 0.2), ('b', 0.4), ('c', 0.4), ('d', 0.0)]
    >>> dist['e'] = 4
    >>> list(sorted(dist.items()))
    [('a', 0.2), ('b', 0.4), ('c', 0.4), ('d', 0.0), ('e', 4)]
    >>> empty = DiscreteDistribution()
    >>> empty.normalize()
    >>> empty
    {}
    """
    "*** YOUR CODE HERE ***"
    tot = sum([abs(self.get(k))for k in self.keys()])    #확률값의 키를 모두 더한 값을 구함

    if tot < 0: #음수인 경우, 양수로 전환시킴
        tot = -tot
    elif tot == 0:  #0인 경우, 1로 전환, 절대로 일어나지 않는 사건은 없음.
        tot = 1

    for k in self.keys():   #확률 키 개수만큼 반복
        self.__setitem__(k, self.get(k)/tot)     #해당 키에서 전체합친 키값을 나눔으로써 정규화 시켜 item에 저장
```

Fig. 1. DiscreteDistribution의 normalize 함수 소스코드

```
def sample(self):
    """
    Draw a random sample from the distribution and return the key, weighted
    by the values associated with each key.

    >>> dist = DiscreteDistribution()
    >>> dist['a'] = 1
    >>> dist['b'] = 2
    >>> dist['c'] = 2
    >>> dist['d'] = 0
    >>> N = 100000.0
    >>> samples = [dist.sample() for _ in range(int(N))]
    >>> round(samples.count('a') * 1.0/N, 1)  # proportion of 'a'
    0.2
    >>> round(samples.count('b') * 1.0/N, 1)
    0.4
    >>> round(samples.count('c') * 1.0/N, 1)
    0.4
    >>> round(samples.count('d') * 1.0/N, 1)
    0.0
    """
    "*** YOUR CODE HERE ***"
    self.normalize()    #확률 키값들을 정규화 시킴

    keys_arr = [k for k in self.keys()] #키값들에 대해서 따로 리스트로 저장해놓음
    items_arr = np.array([self.get(k) for k in keys_arr])    #아이템, 확률에 대한 리스트를 넘파이를 이용하여 리스트화시켜 정리해놓음.
    cum_sum = np.cumsum(items_arr)  #각 원소들의 누적합을 저장하여 계산한 값을 cum_sum에 넣음.

    r = random.random() #랜덤값을 하나 지정함. 랜덤값이 샘플분포를 할 때 역할을 수행할 수 있음.

    ctr = 0 #카운트 값
    for i in range(len(items_arr)): #아이템의 수만큼 반복
        if r > cum_sum[i]:  #랜덤한 값이 누적합보다 크다면 카운트 증가
            ctr += 1
        else:   #아니면 종료시킴
            break

    return keys_arr[ctr]    #랜덤한 값보다 커지는 순간의 누적합을 출력함
```

Fig. 2. DiscreteDistribution의 sample함수 소스코드

Grading code 결과

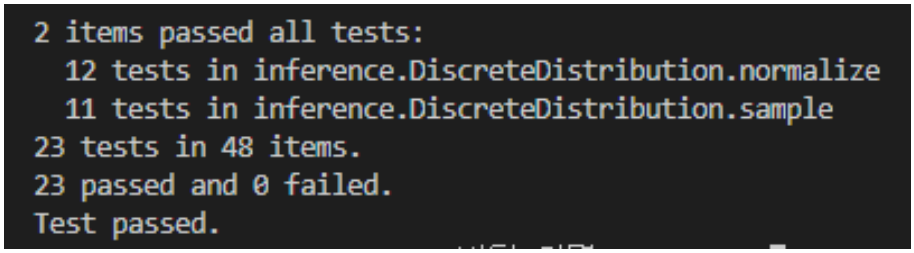

Fig. 3. DiscreteDistribution Grading code 결과

해결방법을 실행 했을 때의 결과 및 분석

우선 주석의 예제로 주었던 문제를 바탕으로 정규화를 수행하기 전, 리스트에 'a'는 1이 저장되어 있고, 'b', 'c'는 2가 저장되고, 'd'의 부분에서는 0이 저장되었다고 하였을 때, 전체의 개수는 총 5개이고, 이를 바탕으로 정규화 함수를 실행 한다면, 'a'는 1/5가 되고, 'b', 'c'는 2/5가 된다. 또한, 'd'의 경우는 0이기 때문에 0이 된다. 이를 튜플로 변환하여 리스트에 집어넣는 것을 수행하고, 혹시 새로운 'e'가 추가가 된다면, 정규화를 실행하기 전까지는 전체 리스트에서 4의 값을 유지하고 있는다. 즉, 정규화는 조인트 시킬 리스트 중 하나의 키값에서 전체의 키를 합친 값을 나눠주면 정규화가 된다고 볼 수 있다.

두번째로 샘플에 대해서 얘기를 하자면, 위의 조건을 그대로 가져와 리스트에 'a'는 1이 저장되어 있고, 'b', 'c'는 2가 저장되고, 'd'의 부분에서는 0이 저장되었다고 하였을 때, 정규화를 통해서 각각의 값을 구한 후, 그 값들에 대해서 누적합을 하여 누적분포함수를 만든다. 그리고 나서 샘플값 N을 이용하여 어느 부분에 해당 사건이 일어났는지에 대해서 검사를 수행한 후, 해당 구간에 도달하게 되면, 미리 정규화 된 값을 호출하는 방법을 이용하였다.

만약 해당 자료들에 대한 차트를 그리자면 다음과 같아진다.

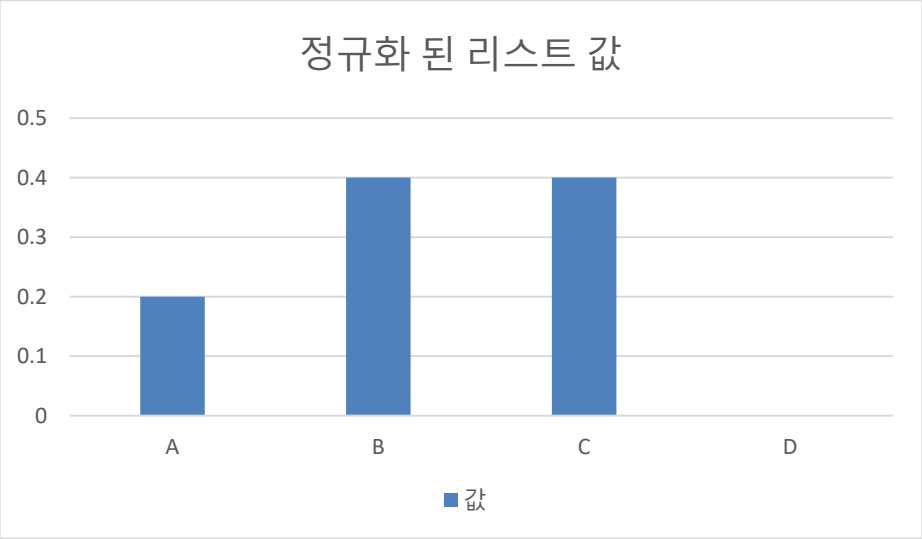

Fig. 4. DiscreteDistribution의 normalize 함수 예제 정규화 결과

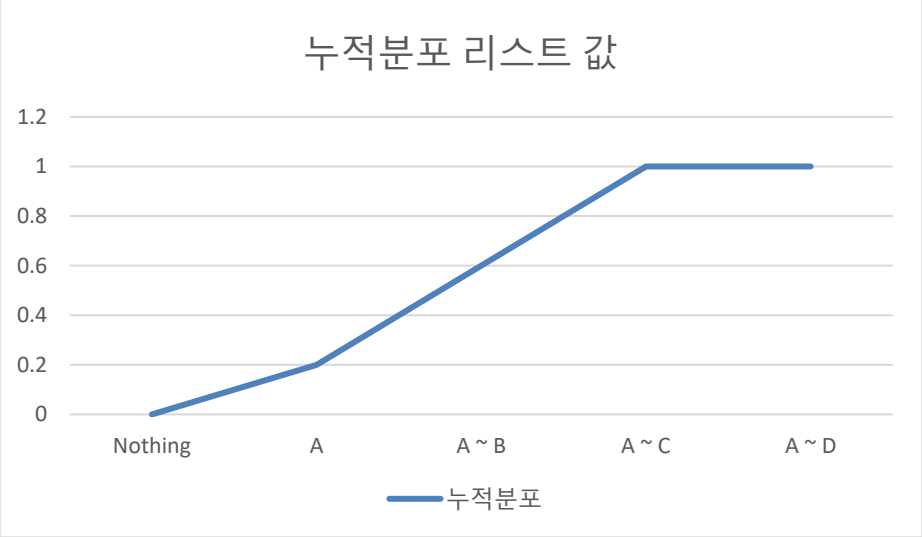

Fig. 5. DiscreteDistribution의 sample함수 내 누적분포함수 예제 결과

느낀점 및 제안점, 고찰 등

이번 과제를 통해 알 수 있었던 점은 확률을 계산할 때 다양한 방법이 있다는 것을 알 수 있었다. 확률을 통해서 내가 어떠한 곳을 갔을 때, 노이즈를

감수하고도 가장 좋은 길인지 선택하는데 있어서 가장 좋은 방법 중 하나인 확률을 이용하는 것을 알 수 있었다.

2.2 Observation Probability

해결방법에 대한 소스 코드

```
def getObservationProb(self, noisyDistance, pacmanPosition, ghostPosition, jailPosition):
    """
    Return the probability P(noisyDistance | pacmanPosition, ghostPosition).
    """
    "*** YOUR CODE HERE ***"
    if noisyDistance == None:   #귀신 소리가 탐지되지 않으면
        if ghostPosition == jailPosition:   #귀신의 위치가 감옥이라면
            return 1    #확률 1 반환, 이미 감옥에 갇혀 귀신은 없는 것임.
        else:
            return 0    #확률 0 반환, 감옥에 귀신이 있지 않음.


    if ghostPosition == jailPosition:   #귀신의 위치가 감옥이라면
        if noisyDistance == None:   #귀신 소리가 탐지되지 않으면
            return 1    #확률 1 반환, 감옥에 갇힘.
        else:
            return 0    #확률 0 반환, 감옥에 있지 않음.

    true_dist = manhattanDistance(pacmanPosition, ghostPosition)    #실제 거리는 팩맨과 귀신위치의 맨하탄 거리로 구한다.
    return busters.getObservationProbability(noisyDistance, true_dist)
    #위의 사례가 없다면 노이즈가 있는 관측된 귀신 거리와 실제 거리에 대한 확률을 반환함.
```

Fig. 6. Observation Probability의 getObservationProb 함수 소스 코드

Grading code 결과

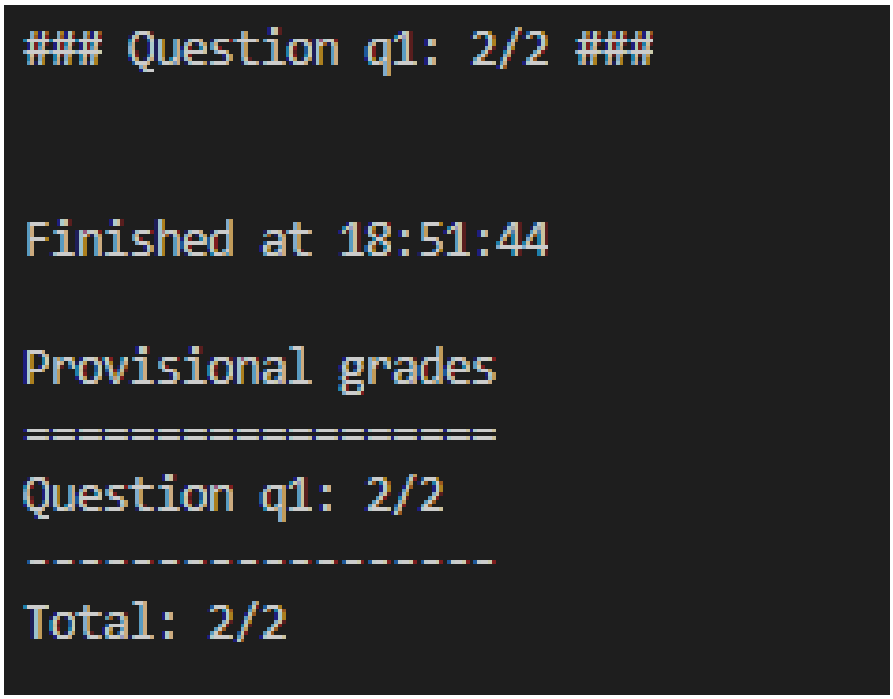

Fig. 7. Observation Probability Grading code 결과

해결방법을 실행 했을 때의 결과 및 분석

Fig. 8. Observation Probability 실행 결과

이 방법을 해결하기 위해서는 getObservationProb 함수의 역할에 대해서 이해할 필요가 있었다. 이 함수는 귀신이 움직일 때 내는 노이즈를 읽는 것과, 팩맨의 위치, 귀신의 위치, 귀신이 감옥에 있는지를 감지하여 귀신이 내는 소리와 실제위치를 알아내어 그것들에 대한 확률을 반환하는 역할을 한다고 나와 있었다. 그 확률은 팩맨과 유령의 실제거리를 감안하여 귀신이 내는 소리에 대한 확률 분포를 가지며, 이는 buster에 있는 함수 getObservationProbability로 이용하여 계산한다는 것을 알 수 있었다. 이 함수는 관측된 거리 데이터를 바탕으로 해당 위치에 있을 확률을 구하는 것으로, 관측된 데이터가 관측분포도 안에 들어가 있지 않으면, 미리 지정된 공식을 이용하여 오류를 고쳐서

출력하고, 그렇지 않으면 저장된 분포도에서 리스트 내 값을 출력하는 것을 수행한다. 이를 이용하여 해당 함수를 반환하기만 하면 끝난다는 것을 알 수 있었다. 하지만, 귀신 소리가 탐지되지 않거나, 귀신의 위치가 감옥이라면 감옥에 갇혔다고 판단해야하기 때문에 확률을 1을 줌으로써 이를 표현하였다. 이 예외처리를 반드시 해야한다고 되어 있어 예외처리를 하였다. 하지만 거리가 감지되는 경우는 귀신이 감옥에 없기 때문에 확률을 0을 주었다.

느낀점 및 제안점, 고찰 등

이번 과제를 통해 알 수 있었던 점은 귀신이 해당 위치에 있는지에 대해서 판단하는 방법으로 확률을 이용했다는 점을 알 수 있었는데, 귀신의 소리가 안들리고, 감옥에 있거나, 감옥에 있으면서 소리가 안들릴 경우 귀신이 감옥에 있다고 판단하고, 그렇지 않으면 귀신이 감옥에 있지 않다고 판단하는 것을 처리한 후, 나머지의 감옥에 있지 않으면서 소리가 들리는 귀신들에 대한 처리 방법을 수행 할 수 있었다. 1 또는 0으로 확정적으로 확률을 줌으로써 귀신에 대한 예외처리를 수행하였음을 알 수 있었던 코드였다.

2.3 Exact Inference Observation

해결방법에 대한 소스 코드

```
def observeUpdate(self, observation, gameState):
    """
    Update beliefs based on the distance observation and Pacman's position.

    The observation is the noisy Manhattan distance to the ghost you are
    tracking.

    self.allPositions is a list of the possible ghost positions, including
    the jail position. You should only consider positions that are in
    self.allPositions.

    The update model is not entirely stationary: it may depend on Pacman's
    current position. However, this is not a problem, as Pacman's current
    position is known.
    """
    "*** YOUR CODE HERE ***"
    self.beliefs.normalize()     #초기 믿음에 의한 확률값들을 정규화 시킴

    for pos in self.allPositions:   #모든 귀신의 위치의 수만큼 반복
        self.beliefs[pos] = self.beliefs[pos] * self.getObservationProb(
            noisyDistance=observation,  #관측된 거리
            pacmanPosition=gameState.getPacmanPosition(),   #팩맨의 위치
            ghostPosition=pos,  #귀신의 위치
            jailPosition=self.getJailPosition() #감옥의 위치
        )   #이전에 구현하였던 getObservationProb 함수를 이용하여 필요한 데이터 값들을 채워 넣음.
    self.beliefs.normalize()     #이 후, belief 리스트(믿음에 의한)를 정규화 시킴.
```

Fig. 9. Exact Inference Observation의 observeUpdate 소스코드

Grading code 결과

```
### Question q2: 3/3 ###


Finished at 21:02:34

Provisional grades
==================
Question q2: 3/3
------------------
Total: 3/3
```

Fig. 10. Exact Inference Observation의 Grading code 결과

해결방법을 실행 했을 때의 결과 및 분석

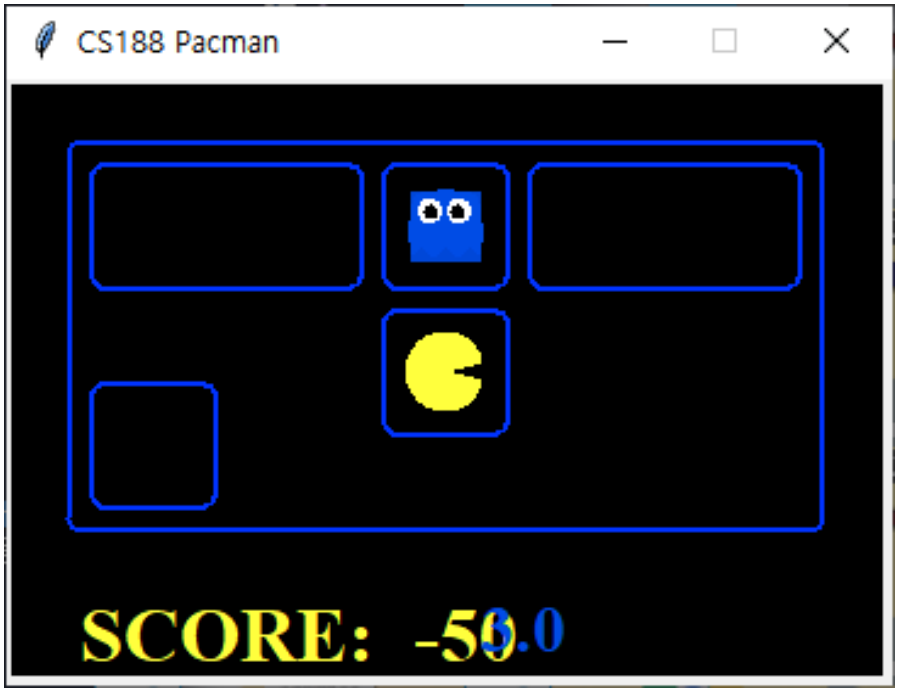

Fig. 11. Exact Inference Observation 결과

여기서는 확률에서 새로운 개념인 믿음이 나오게 되는데, 이 믿음은 팩맨의 위치와 팩맨에서의 관측거리를 기반으로 한 확률 리스트를 의미한다. 해당 확률 리스트를 만드는데, 팩맨에서의 관측거리는 유령의 위치에 대한 맨해튼 거리, 노이즈(소리)가 낀 관측되는 거리를 의미한다. 이 코드에서는 팩맨의 위치가 바뀌더라도 지속적인 업데이트가 되어야 하므로 지속적으로 리스트를 수정하고, 정규화를 하는 것이 목표이다. 따라서 이전에 작성하였던 getObservationProb 함수에 지속적으로 변환되는 관측값, 팩맨 위치, 귀신 위치, 감옥위치 등 다양한 자료를 넣어서 지속적으로 믿음에 의한 확률을 업데이트 해주고, 해당 리스트를 정규화 해줌으로써 확률분포를 만든다. 여기서 믿음에 의한 확률의 뜻은 베이즈 이론에 의거하는 것으로, 확률은 믿음에서 나오는 것이라는 뜻에서 유래된 것이다. 이로 인해서 팩맨이 탐지하였을 때의 확률이 높다면 거기에 귀신이 있을 것 같다고 믿는것과 같은 것이라고 본다.

느낀점 및 제안점, 고찰 등

이번 과제를 통해 알 수 있었던 점은 확률은 결국 믿음에서 나오는 것이라는 베이즈 이론을 바탕으로 코드를 짜고 있다는 점을 알 수 있었다. 여기서 중요한 점으로는 관측된 값을 바탕으로 지속적으로 확률을 업데이트 해줘야한다는 점인데, 이는 팩맨과 귀신은 항상 같은 위치에서 존재하는 것이 아니기 때문에 지속적인 업데이트가 필요하다는 것을 알 수 있었다.

2.4 Exact Inference with Time Elapse

해결방법에 대한 소스 코드

```
def elapseTime(self, gameState):
    """
    Predict beliefs in response to a time step passing from the current
    state.

    The transition model is not entirely stationary: it may depend on
    Pacman's current position. However, this is not a problem, as Pacman's
    current position is known.
    """
    "*** YOUR CODE HERE ***"
    new_beleifs = DiscreteDistribution()     #새로운 믿음에 의한 확률을 새로 정의
    new_beleifs.normalize() #새로 정의하였던 확률들을 정규화 시킴. P(a)
    for pos in self.allPositions:   #모든 귀신이 움직일 수 있는 위치의 수만큼 반복
        new_pos_dist = self.getPositionDistribution(gameState, pos) #새로운 위치에 의한 거리의 확률들을 얻어옴. P(b|a)
        for new_pos in self.allPositions:   #모든 귀신이 움직일 수 있는 위치의 수만큼 반복
            new_beleifs[new_pos] += new_pos_dist[new_pos]*self.beliefs[pos]
            #새로운 믿음에 의한 확률을 갱신시킴, 갱신 시 이후의 이동분포(확률)와 기존의 확률을 곱함.
            #베이즈 추론(이론), 확률적 평균값으로 기대값을 계산함. 확률의 확률을 사용하여 추정함. P(b|a) * P(a) = P(a,b)
    new_beleifs.normalize() #갱신을 통해서 새로운 확률을 정규화 시킴.

    self.beliefs = new_beleifs  #위에서 구한 확률을 갱신시킴.
```

Fig. 12. Exact Inference with Time Elapse의 ElapseTime 함수 소스코드

Grading code 결과

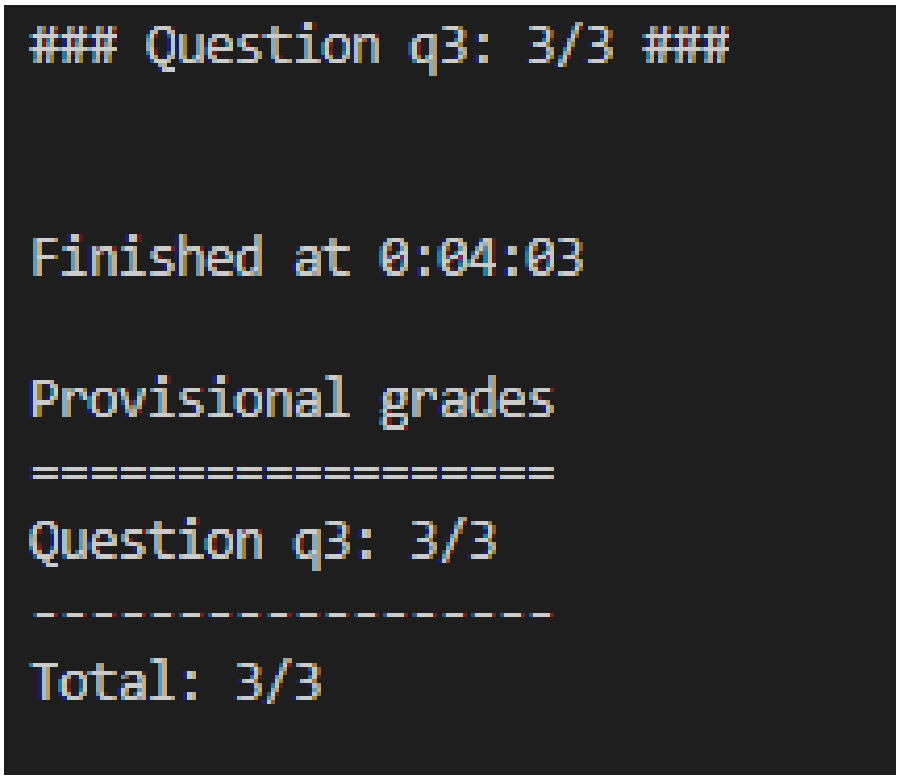

Fig. 13. Exact Inference with Time Elapse Grading code 결과

해결방법을 실행 했을 때의 결과 및 분석

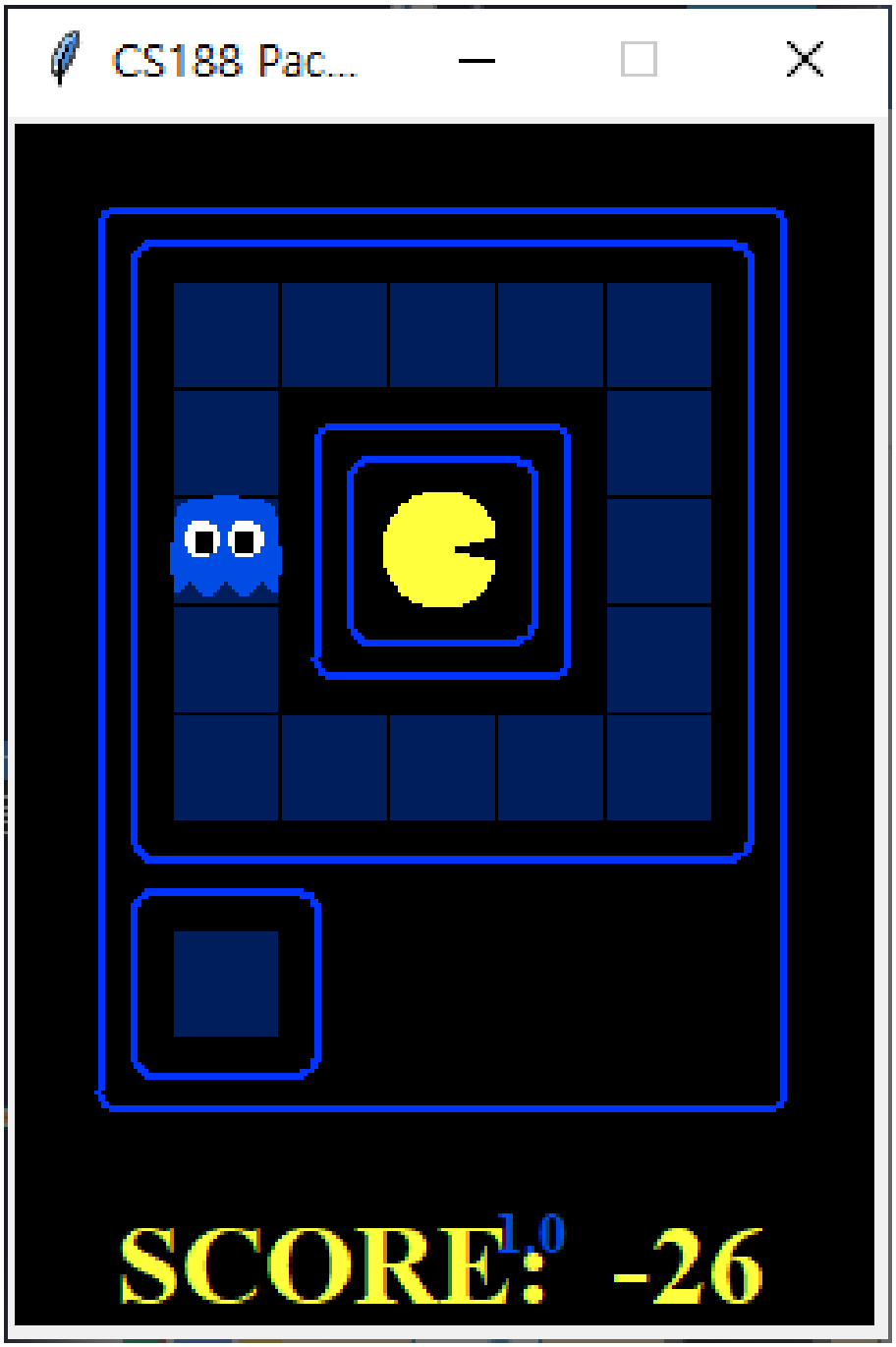

Fig. 14. Exact Inference with Time Elapse 실행 결과

위 문제를 해결하기 위해서는 베이즈 추론에 대해서 이해할 필요가 있다. 사전의 확률을 이용하여 사후의 확률을 계산하는 것으로, 확률의 확률을 이용한다. 이전의 확률을 얻어온 후, 정규화를 하여 나오지 않을 확률의 가짓수를 줄이고, 그리고 베이즈 이론을 통해서 이전에 이러한 값이 나왔는데 다시 이러한 값일 수 있는지에 대한 기대값(사후확률)을 얻어온 후, 이를 믿음에 의한 확률로 사용하는 것이다. 이를 통해서 다시 새롭게 업데이트를 수행 할 수 있는 것이다. 수식으로 표현하자면 다음과 같다. 이전의 확률 a, 이후의 확률 b, P(b|a) = a가 제공되었을때의 b의 확률, 베이즈 추론 : P(b|a) * P(a) = P(a,b)

느낀점 및 제안점, 고찰 등

이번 과제를 통해 알 수 있었던 점은 베이즈 이론은 상당히 많은 곳에서 사용하는 것을 알 수 있었는데, 일상속에서 우리가 쉽게 예상하고자 하는 것들이 전부 베이즈 이론이라는 것에 많이 놀랐다.

2.5 Exact Inference Full Test

해결방법에 대한 소스 코드

```
def chooseAction(self, gameState):
    """
    First computes the most likely position of each ghost that has
    not yet been captured, then chooses an action that brings
    Pacman closest to the closest ghost (according to mazeDistance!).
    """
    pacmanPosition = gameState.getPacmanPosition()
    legal = [a for a in gameState.getLegalPacmanActions()]
    livingGhosts = gameState.getLivingGhosts()
    livingGhostPositionDistributions = \
        [beliefs for i, beliefs in enumerate(self.ghostBeliefs)
         if livingGhosts[i+1]]
    "*** YOUR CODE HERE ***"
    #살아있는 귀신이 있을법한 위치분포 beliefs의 리스트의 키(확률) 값 중 가장 큰 값의 키(가장 큰 확률)의 분포리스트를 저장함.
    max_prob_locations = [max(beliefs.keys(), key = lambda x: beliefs[x]) for beliefs in livingGhostPositionDistributions]
    #넘파이의 최소 색인값을 찾는 함수를 이용하여 여러개의 위치 중 가장 가까운 곳을 가까운 유령 위치로 지정함.
    #getDistance를 이용하여 두 위치의 사이의 거리를 얻어올 수 있음. 팩맨과 해당 확률이 있는 위치의 인덱스.
    closest_ghost_loc_index = np.argmin([self.distancer.getDistance(pacmanPosition, pos) for pos in max_prob_locations])
    #해당 부분의 키값(확률)을 가져옴. 해당 값은 가장 가까운것 같은 귀신의 위치가 됨.
    closest_ghost_loc = max_prob_locations[closest_ghost_loc_index]
    #새로운 거리를 구함, 팩맨의 거리와 팩맨이 이동할 수 있는 거리와 가장 가까운것 같은 귀신의 위치를 비교하여 거리를 구함.
    new_gs = [self.distancer.getDistance(Actions.getSuccessor(pacmanPosition, a), closest_ghost_loc) for a in legal]
    result = legal[np.argmin(new_gs)]   #가장 가까운 귀신의 위치를 결과로 나타냄.

    return result   #가까운 귀신의 맨해튼 거리 반환
```

Fig. 15. Exact Inference Full Test 중 chooseAction 소스코드

Grading code 결과

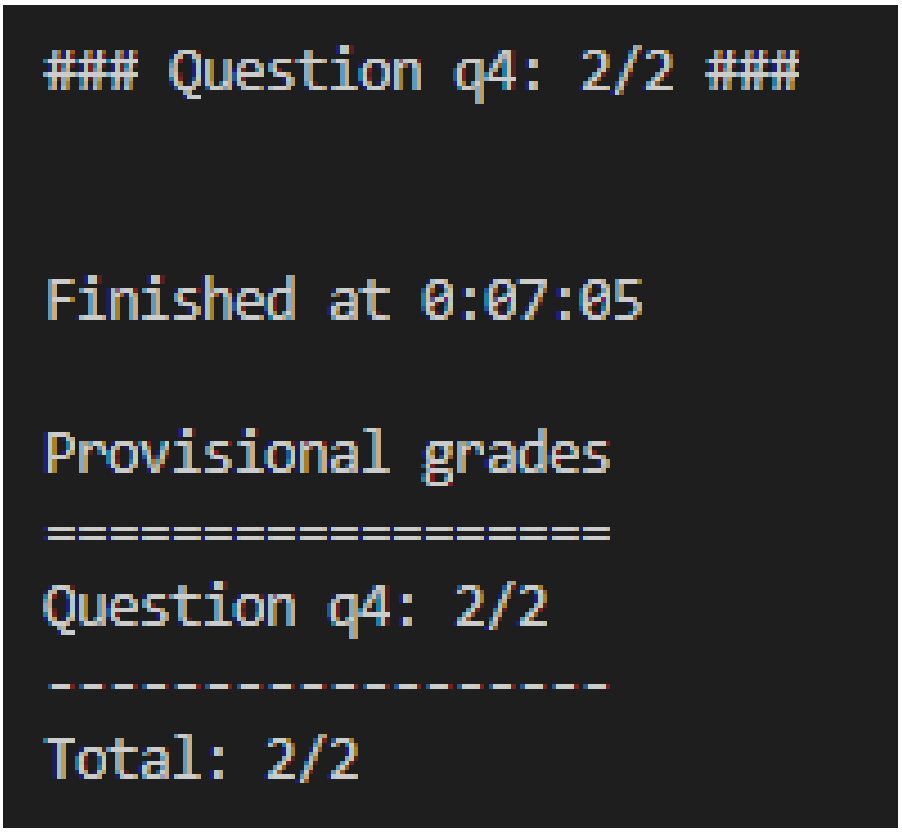

Fig. 16. Exact Inference Full Test Grading code 결과

해결방법을 실행 했을 때의 결과 및 분석

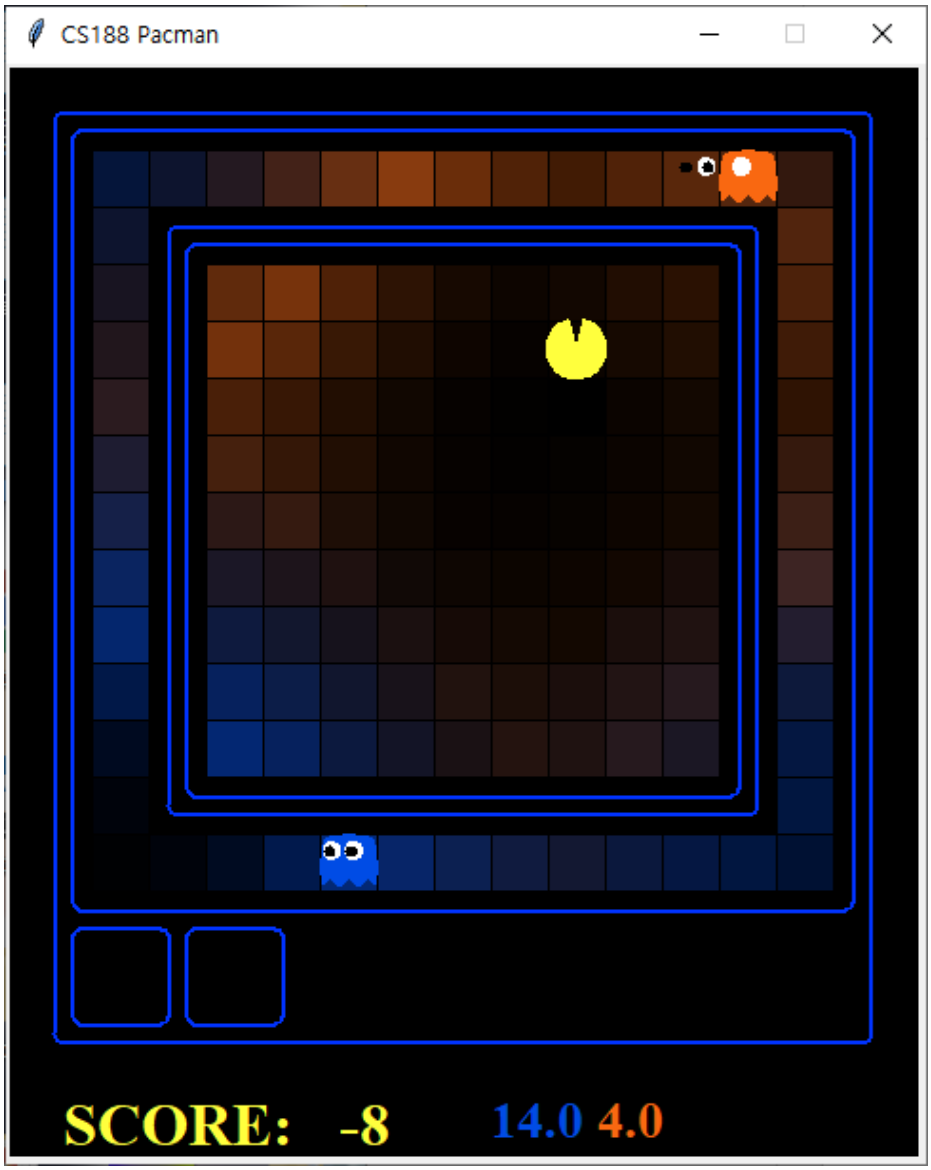

Fig. 17. Exact Inference Full Test 실행 결과

해당 문제를 해결하기 위해서 가장 짧은 거리에 예상되는 유령쪽으로 가야하기 때문에 여러가지의 확률 중에서 가장 큰 확률을 가지면서 가장 짧은 거리를 가지는 위치를 찾아서 해당 위치에 대한 맨해튼 거리를 반환해줌으로서 문제를 해결하였다. 그리드 알고리즘을 그대로 적용한 것으로, 유령을

먹기위해서 가장 탐욕스러운 방향으로 이동하도록 방향을 결정하도록 코드를 작성하였다.

느낀점 및 제안점, 고찰 등

이번 과제를 통해 알 수 있었던 점은 확률을 계산할 때 다양한 방법이 있다는 것을 알 수 있었다. 확률을 통해서 내가 어떠한 곳을 갔을 때, 노이즈를 감수하고도 가장 좋은 길인지 선택하는데 있어서 가장 좋은 방법 중 하나인 확률을 이용하는 것을 알 수 있었다.

3 Conclusion

느낀점

이번 과제를 하면서 알 수 있었던 점은 확률을 이용하여 다음 활동을 예측하는데 있어서 도움을 줄 수 있었다는 사실을 알 수 있었다. 이번에 배운 확률 또한 인공지능의 판단에 있어서 도움을 많이 준다는 사실에 새삼 놀랐던 점이 있었다.

어려웠던 점

이번에는 역시 확률이 나오다 보니 생각보다 어려웠다. 확률이 수학적인 면모를 띄고 있기 때문에 수학에 약한 나로써는 이해하는데 많은 시간이 들었다. 하지만 다시 한 번 확률의 힘을 느끼고 이를 다시 한 번 공부 할 수 있었다.

제안점

이번 과제와 같이 소스코드를 올리는 방안이 수도코드를 작성하는 것보다는 편해서 좋았다.

References

1. Bayesian Infernece, https://forensics.tistory.com/45?category=821535
2. Bayes arrangement, https://j1w2k3.tistory.com/1009
3. Bayesian Inference, https://sumniya.tistory.com/29