ЛАБОРАТОРНАЯ РАБОТА № 6
«ИССЛЕДОВАНИЕ ДИНАМИЧЕСКИХ СЕТЕЙ БАЙЕСА»

Цель работы

Исследование методов точного и приближенного вероятностного вывода с использованием динамических сетей Байеса, приобретение навыков программирования интеллектуальных агентов, знания которых представляются условными высказываниями с определенной степенью уверенности.

Постановка задачи

1. Изучите по лекционному материалу и учебным пособиям [1-3, 9] основные понятия вероятностного вывода, понятие сетей Байеса, марковских моделей, методы и алгоритмы точного и приближенного вероятностного вывода в скрытых марковских моделях. Ответьте на контрольные вопросы.

2. Используйте для выполнения лабораторной работы файлы из архива МИСИИ_лаб_6.zip. Разверните программный код лабораторной работы в новой папке и не смешивайте с файлами предыдущих лабораторных работ. Архив содержит следующие файлы:

Ваш код будет автоматически проверяться автооценивателем. Поэтому не меняйте имена каких-либо функций или классов в коде, иначе вы внесете ошибку в работу автооценивателя.

3. В рассматриваемом варианте игры цель состоит в том, чтобы выследить невидимых призраков. Пакман оснащен сонаром (слухом), который обеспечивает оценку манхэттенского расстояния до каждого призрака по его шумам.

Игра заканчивается, когда Пакман выследит и съест всех призраков. Для начала попробуйте сыграть в игру, используя клавиатуру:

python busters.py

Для выхода из игры просто закройте графическое окно.

Цвет позиции поля игры указывает, где может находиться каждый из призраков с учетом оценок расстояний, вычисляемых по их шуму. Оценки расстояний, отображаемые в нижней части графического окна, всегда неотрицательны и всегда находятся в пределах 7 единиц от их истинного значения. Вероятность нахождения призрака в соответствующей позиции экспоненциально уменьшается при увеличении отличия от истинного расстояния.

Ваша основная задача — реализовать вероятностный вывод для отслеживания призраков. В случае игры, осуществляемой с помощью клавиатуры, по умолчанию реализуется грубая форма вывода: все квадраты, в которых может быть призрак, закрашиваются цветом призрака. Естественно, нам нужна более точная оценка положения призрака. К счастью, CMM представляют мощный инструмент для максимально эффективного использования имеющейся у нас информации.

Вы должны будете реализовать алгоритмы для выполнения как точного, так и приближенного вывода с использованием динамических байесовских сетей.

При отладке кода будет полезно иметь некоторое представление о том, что делает автооцениватель. Автооцениватель использует 2 типа тестов, различающихся файлами .test, которые находятся в подкаталогах папки test_cases. Для тестов класса DoubleInferenceAgentTest вы увидите визуализации распределений, сгенерированных в ходе построения выводов вашим кодом, но все действия Пакмана будут предварительно выбираться в соответствии с ранее заложенной реализацией. Второй тип теста — GameScoreTest, в котором действия выбирает созданный вами агент BustersAgent. Вы будете наблюдать, как играет Пакман и как он выигрывает.

По мере реализации и отладки кода может оказаться полезным запускать по одному тесту за раз. Для этого вам нужно будет использовать флаг -t при вызове автооценивателя. Например, если вы хотите запустить только первый тест задания 1, используйте команду:

python autograder.py -t test_cases/q1/1-ObsProb

Как правило, все тестовые примеры можно найти внутри test_cases/q*.

Иногда автооцениватель может не сработать при выполнении тестов с графикой. Чтобы определить, эффективен ли ваш код, используйте в этом случае дополнительно при вызове автооценивателя параметр --no-graphics.

4. В задании 0 требуется реализовать следующие методы класса DiscreteDistribution: normalize и sample. Класс является разновидностью словаря языка Python и представляется в виде дискретных ключей и значений пропорциональных вероятностям.

Определите метод normalize, который нормализует значения распределения, таким образом, чтобы сумма всех значений была равна единице. Используйте метод total, чтобы найти сумму значений распределения. Для распределения, в котором все значения равны нулю, ничего делать не требуется:

summa = self.total()

if summa == 0:

return

Иначе необходимо все значения распределения нормализовать по отношению к значению переменной summa. Метод должен изменять значения распределения в памяти напрямую, а не возвращать новое распределение.

Определите метод sample, который формирует выборку из распределения, в которой вероятность значения ключа пропорциональна соответствующему хранящемуся значению. Предполагается, что распределение не пустое и не все значения равны нулю. Обратите внимание, что обрабатываемое распределение не обязательно нормализовано. Для выполнения этого задания полезной будет встроенная функция Python

random.random(). Способ формирования выборки из распределения описан в разделе 6.2.5.

Тестов автооценивателя на это задание нет, но правильность реализации

можно легко проверить. Для этого можно использовать Python doctests, которые включаются в комментарии определяемых методов. Можно свободно добавлять новые и реализовывать другие собственные тесты. Чтобы запустить doctest и выполнить проверку, используйте вызов:

python -m doctest -v inference.py

Обратите внимание, что в зависимости от деталей реализации метода sample, некоторые правильные реализации могут не пройти предоставленные doctесты. Чтобы полностью проверить правильность метода sample, необходимо сделать много выборок и посмотреть, сходится ли частота каждого ключа к соответствующему значению вероятности.

Внесите код и результаты тестирования расзработанных методов в отчет.

6.4.5. В задании 1 необходимо реализовать метод getObservationProb(self, noisyDistance, pacmanPosition, ghostPosition, jailPosition), который возвращает вероятность наблюдения noisyDistance для заданных позиций Пакмана и призрака.

Данный метод соответсвует модели наблюдения (восприятия), которой оснащён Пакман.

Значения, возвращаемые датчиком расстояния, характеризуются распределением вероятностей, которое учитывает истинное расстояние от Пакмана до призрака. Это распределение вычисляется в модуле busters функцией busters.getObservationProbability(noisyDistance, trueDistance), которая возвращает вероятности P(noisyDistance | trueDistance). Для выполнения задания вы должны использовать эту функцию совместно с функцией manhattanDistance, которая вычисляет истинное расстояние trueDistance между местоположением Пакмана и местоположением призрака:

trueDistance=manhattanDistance(pacmanPosition, ghostPosition)

P=busters.getObservationProbability(noisyDistance, trueDistance)

Кроме этого, необходимо учесть особый случай, связанный с арестом призрака. Когда призрак попадает в тюрьму, то датчик расстояния возвращает значение None. Если при этом позиция призрака — это позиция тюрьмы, т.е. ghostPosition == jailPosition, то датчик расстояния возвращает — None с вероятностью P=1.

И наоборот, если оценка расстояния не None, то вероятность нахождения призрака в тюрьме (ghostPosition == jailPosition) равна нулю. Соответственно для указанных условий метод getObservationProb должен возвращать 1 или 0.

Чтобы протестировать свой код, запустите автооцениватель для этого задания:

python autograder.py -q q1

Внесите код и результаты тестирования метода в отчет.

6. В задании 2 необходимо реализовать метод observeUpdate класса ExactInference. Метод обеспечивает вычисления в соответствии с правилом обновления (6.19). В данном случае обновляется распределение степеней уверенности агента в отношении позиций призрака, оцениваемых на основе наблюдений, поступающих от датчика расстояний Пакмана. Степени уверенности представляются вероятностями того, что призрак находится в определенной позиции, и хранятся в виде объекта DiscreteDistribution в поле с именем self.beliefs, которое необходимо обновлять для каждой возможной позиции призрака после получения наблюдения.

Для выполнения задания необходимо использовать функцию self.getObservationProb (была определена в задании 1), которая возвращает вероятность наблюдения с учетом положения Пакмана, потенциального положения призрака и локации тюрьмы. Получить значение позиции Пакмана можно с помощью gameState.getPacmanPosition(), позицию тюрьмы с помощью --self.getJailPosition(), а список возможных позиций призрака с помощью self.allPositions Для выполнения задания вам необходимо реализовать цикл по всем возможным позициям призрака possibleGhostPos:

for possibleGhostPos in self.allPositions:

…

В цикле должно выполняться обновление степеней уверенности для каждого состояния self.beliefs[possibleGhostPos] в соответствии с (6.19) при использовании вероятности наблюдения, вычисляемой в ходе вызова:

self.getObservationProb(noisyDistance, pacmanPosition, possibleGhostPos, jailPosition)

При этом учтите, что значения B’(Wi+1) из (6.19) обновляются до значений B(Wi+1) и все эти значения хранятся в одной и той же области памяти self.beliefs[possibleGhostPos]. Не забудьте в конце выполнить нормализацию распределения self.beliefs.normalize().

Чтобы запустить автооцениватель для этого задания и визуализировать результат используйте команду:

python autograder.py -q q2

На экране высокие апостериорные степени уверенности представляются более ярким цветом. Если вы хотите запустить этот тест без графики, то используйте вызов с параметром --no-graphics:

python autograder.py -q q2 --no-graphics

Примечание: у агентов-охотников есть отдельный модуль вероятностного вывода для каждого призрака. Вот почему, если печатать наблюдение внутри функции ObserveUpdate, то отображается только одно число, даже если имеется несколько призраков.

Внесите код и результаты тестирования метода в отчет.

7. В задании 3 необходимо реализовать метод elapseTime класса ExactInference. Метод выполняет вычисления в соответствии с правилом обновления во времени (6.18), где состояния представляются позициями призрака.

Чтобы получить распределение степеней уверенности по новым позициям призрака, учитывая его текущую позицию, используйте следующую строку кода:

newPosDist = self.getPositionDistribution(gameState, ghostPos),

где ghostPos — текущая позиция призрака, newPosDist — это объект типа DiscreteDistribution, в котором для каждой позиции p призрака (из self.allPositions) newPosDist[p] — это вероятность того, что призрак будет находиться в момент времени t + 1 в позиции p, если в предыдущий момент времени t призрак находился в позиции ghostPos.

В ходе реализции метода для каждой позиции призрака ghostPos организуйте цикл по всем новым возможным позициям

for newPos, prob in newPosDist.items():

…,

в котором выполните обновление степеней уверенности нахождения призрака в новых позициях beliefDict[newPos] в соответствии с (6.18), где beliefDict – словарь типа DiscreteDistribution(). При этом значения B(wi) соответствуют self.beliefs[ghostPos], а переходные вероятности Pr(Wi+1,|wi) хранятся в prob.

По окончании циклов наобходимо нормализовать распределение beliefDict и сохранить его в self.beliefs=beliefDict.

Обратите внимание, что этот вызов self.getPositionDistribution может быть довольно затратным. Поэтому, если время ожидания исполнения вашего кода истечет, то стоит подумать об уменьшении количества вызовов self.getPositionDistribution.

При автооценивании правильности выполнения этого задания иногда используется призрак со случайными перемещениями, а иногда используется GoSouthGhost. Последний призрак имеет тенденцию двигаться на юг, поэтому со временем распределение степеней уверенности Пакмана должно начать фокусироваться в нижней части игрового поля. Чтобы увидеть, какой призрак используется для каждого теста, просмотрите файлы .test.

Для автооценивания этого задания и визуализации результатов используйте команду:

python autograder.py -q q3

Если вы хотите запустить этот тест без графики, то добавьте в предыдущий вызов параметр --no-graphics.

Наблюдая за результатами автооценивания, помните, что более светлые квадраты указывают на то, что призрак с большей вероятностью будет находиться в них. Подготовьте ответы на вопросы: В каком из тестовых случаев вы заметили различия в закраске квадратов? Можете ли вы объяснить, почему некоторые квадраты становятся светлее, а некоторые темнее?

Внесите код и результаты тестирования метода в отчет.

9. При выполнении задания 5 в методе initializeUniformly переменная, в которой хранятся частицы, представляется в виде списка self.particles, элементами которого являются позиции частиц. Хранение частиц в виде другого типа данных, например, словаря, является неправильным и приведет к ошибкам.

Число инициализирумых частиц задается конструктором класса ParticleFilter и по умолчанию равно self.numParticles=300. Допустимые позиции частиц positions определяются следующим образом: positions=self.legalPositions. Прежде чем сформировать список частиц, определите целое число частиц, приходящихся на одну позицию

numP=self.numParticles // len(positions)

Затем в цикле, пока длина списка self.particles не станет равной числу частиц self.numParticles, для каждой допустимой позиции частиц pos из positions необходимо тиражировать позицию pos

sublistP=[pos]*numP

и расширить общий список частиц на sublistP , т.е. self.particles.extend(sublistP).

Метод getBeliefDistribution получает список частиц и формирует соответсвующее распределение. Поэтому в начале создайте переменную-распределение beliefDist, которая является экземпляром класса DiscreteDistribution. Затем посчитайте число частиц в каждой позиции:

for particle in self.particles:

beliefDist[particle]=beliefDist[particle]+1

Нормализуйте полученное распределение beliefDist и верните его в качестве результата.

Чтобы протестировать задание выполните команду:

python autograder.py -q q5

Внесите код разработанных методов и результаты тестирования в отчет.

10. В задании 6 необходимо сформировать выборку из распределения с

учетом весов наблюдений. Поэтому создайте в начале экземпляр распределения, например weightsDist, путем вызова конструктора класса DiscreteDistribution().

Чтобы найти вероятности наблюдений с учетом положения Пакмана, потенциального положения призрака и положения тюрьмы, используйте ранее определенную функцию self.getObservationProb. В соотвествии с алгоритмом обновления на основе наблюдения (см. п.6.2.8) найдите сумму весов для каждой позиции for pos in self.particles:

weightsDist[pos]+= self.getObservationProb(observation, pacmanPosition, pos, jailPosition)

Нормализуйте полученное распределение weightsDist и сформируйте новый список частиц, сделав выборки из weightsDist:

self.particles = [weightsDist.sample() for _ in range(int(self.numParticles))]

Не забудьте учесть особый случай, когда все частицы получают нулевой вес. В этом случае следует повторно инициализировать список частиц, вызвав метод initializeUniformly(gameState).

Метод возвращает обновленный список частиц self.particles.

Чтобы запустить автооцениватель для этого задания и визуализировать результаты тестирования, выполните команду

python autograder.py -q q6

или без графики

python autograder.py -q q6 --no-graphics

Внесите код разработанных методов и результаты тестирования в отчет.

11. В задании 7 необходимо реализовать в виде метода elapseTime класса ParticleFilter алгоритм обновления во времени, описанный в п. 6.2.8. Так как в итоге метод должен формировать новый список частиц как выборку из распределения, то создайте в начале экземпляр распределения, например

elapseDist=DiscreteDistribution().

Аналогично методу elapseTime класса ExactInference для определения следующей возможной позиции частицы по предыдущей позиции pos следует использовать функцию self.getPositionDistribution(). Тогда распределение новых позиций частиц можно определить так:

for pos in self.particles:

newPosDist = self.getPositionDistribution(gameState, pos)

Распределение newPosDist представляется словарем с парами занчений { newPos, prob }, где newPos – новая позиция частицы, а prob – вероятность нахождения частицы в этой позиции. Для каждой позиции из списка частиц self.particles посчитайте сумму вероятностей нахождения частиц в соответствующей новой позиции и сохраните значения в elapseDist[newPos]:

for newPos, prob in newPosDist.items():

elapseDist[newPos]+=prob

Нормализуйте полученное распределение elapseDist и сформируйте с его помощью новый список частиц

self.particles=[elapseDist.sample() for _ in range(int(self.numParticles))]

Данный вариант метода elapseTime позволяет отслеживать призраков почти

так же эффективно, как и в случае точного вывода.

Обратите внимание, что для этого задания автооцениватель тестирует как

метод elapseTime, так и полную реализацию фильтра частиц, сочетающую

elapseTime и обработку наблюдений.

Чтобы запустить автооцениватель для этого задания и визуализировать ре-

зультаты используйте команду:

python autograder.py -q q7

Для запуска теста без графики добавьте параметр --no-graphics. Внесите код раз-

работанных методов и результаты тестирования в отчет.

12. В начале выполнения задания 8 определите допустимые позиции

призраков с помощью positions=self.legalPositions. Затем необходимо будет сформировать кортежи из позиций, которые соответствуют одномоментным возможным расположениям разных призраков (в одной позиции может находиться только один призрак). При этом полезны будут возможности модуля Python itertools, который обеспечивает реализацию различных итераторов. В частности, используйте функцию itertools.product, чтобы сформировать список декартовых произведений (перестановок) из возможных допустимых позиций для

self.numGhosts призраков:

cartProduct=[i for i in itertools.product(positions, repeat=self.numGhosts)]

Список cartProduct будет состоять из кортежей в виде перестановок позиций призраков. Однако кортежи не будут следовать в случайном порядке. Чтобы обеспечить их случайный выбор, заполняйте список частиц, выбирая элементы из списка cartProd с помощью вызова random.choice():

self.particles.append(random.choice(cartProduct))

Также, как и в задании 5, заполнение списка частиц выполняйте в цикле пока выполняется условие len(self.particles) <= self.numParticles.

Чтобы запустить автооцениватель для этого задания и визуализировать результаты используйте команду (или с необязательным параметром --no-graphics):

python autograder.py -q q8

Внесите код разработанных методов и результаты тестирования в отчет.

13. При выполнении задания 9 создайте в начале экземпляр распределения weightsDist путем вызова конструктора класса DiscreteDistribution().

По-прежнему позицию Пакмана можно определить с помощью метода gameState.getPacmanPosition(), но для определения позиции тюрьмы призрака, используйте self.getJailPosition(i), так как теперь есть несколько призраков, у каждого из которых есть своя позиция тюрьмы.

При реализации метода необходимо для каждой частицы (представляемой кортежем позиций) формировать свой список наблюдений obs, в котором следует размещать вероятности наблюдений для каждого из призраков. Поэтому необходимо организовать два вложенных цикла:

```
# для каждой из частиц и для каждого из призраков
for p in self.particles:
obs=[]
for i in range(self.numGhosts):
…
```

Как и в аналогичном методе для класса ParticleFilter, здесь необходимо использовать функцию self.getObservationProb, чтобы найти вероятность наблюдения с учетом положения Пакмана, потенциального положения призрака и положения тюрьмы:

```
probObs=self.getObservationProb(observation[i],
pacmanPosition, p[i], self.getJailPosition(i))
obs.append(probObs)
```

В соотвествии с алгоритмом обновления на основе наблюдения (см. п.6.2.8) находим сумму произведений весов совместных наблюдений для каждой комбинации позиций призраков

```
weightsDist[p]+=prod(obs)
```

Нормализуйте полученное распределение weightsDist и сформируйте новый список частиц, сделав выборки из weightsDist аналогично заданию 6 (см. п.6.4.10).

Не забудьте учесть особый случай, когда все частицы получают нулевой вес. В этом случае следует повторно инициализировать список частиц, вызвав метод initializeUniformly(gameState).

Метод возвращает обновленный список частиц self.particles.

Чтобы запустить автооцениватель для этого задания и визуализировать результаты тестирования, выполните команду (возможно с необязательным параметром --no-graphics) python autograder.py -q q9

Внесите код разработанных методов и результаты тестирования в отчет.

14. При выполнении задания 10, как и в задании 9 следует организовать два вложенных цикла по всем частицам и по всем призракам. Вам предлагается следующий шаблон кода, который нужно дополнить:

```
newParticles = []
for oldParticle in self.particles:
newParticle = list(oldParticle) # Список позиций призраков
# цикл обновления всех значений в newParticle
"*** ВСТАВЬТЕ ВАШ КОД СЮДА ***"
…
"""*** КОНЕЦ ВАШЕГО КОДА ***"""
newParticles.append(tuple(newParticle))
self.particles = newParticles
```

Вам необходимо в цикле обновить все значения позиций в newParticle. Поэтому в цикле

```
for i in range(self.numGhosts):
…
```

полагая, что i ссылается на индекс призрака, чтобы получить распределение новых позиций этого призрака, учитывая список (list(oldParticle) предыдущих позиций всех призраков, используйте вызов self.getPositionDistribution(gameState, list(oldParticle), i, self.ghostAgents[i]). Получив распределение новых позиций призрака, сформируйте выборку из

распределения с помощью метода sample() и сохраните результат в newParticle[i].

Обратите внимание, что выполнение этого задания включает в себя автооценивание как задания 9, так и задания 10. Поскольку эти задания связаны с вычислением совместного распределения, для их выполнения потребуется больше времени.

При запуске автооценивателя учтите, что тесты q10/1-JointParticlePredict и q10/2-JointParticlePredict проверяют только реализации обновления во времени, а q10/3-JointParticleFull выполняет полное тестирование. Обратите внимание на разницу между тестом 1 и тестом 3. В обоих тестах Пакман знает, что призраки будут двигаться по сторонам игрового поля. Чем отличаются тесты и почему?

Чтобы запустить автооцениватель для этого задания и визуализировать результаты, используйте команды:

python autograder.py -q q10

или

python autograder.py -q q10 --no-graphics

Внесите код разработанных методов и результаты тестирования в отчет

Ход работы

Задание 0 (0 баллов). Класс DiscreteDistribution

Класс DiscreteDistribution, определенный в inference.py, используется для работы с дискретными распределениями. Этот класс является разновидностью словаря Python, где ключами являются дискретные элементы распределения, а значения ключей равны вероятностям (степени уверенности в возможном значении ключа).

В задании необходимо дописать недостающие методы этого класса: normalize и sample. Метод normalize нормализует значения распределения,

таким образом, чтобы сумма всех значений была равна единице. Метод sample формирует случайную выборку из распределения в соответствии с алгоритмом, описанным п. 6.2.5.

Разработанный код:

```
class DiscreteDistribution(dict):
    """
    Класс для работы с распределением,
    представляемым в виде словаря с
    набором значений ключей и соответсвующих вероятностей
    """

    def __getitem__(self, key):
        self.setdefault(key, 0)
        return dict.__getitem__(self, key)

    def copy(self):
        """
        Возвращает копию распределения
        """
        return DiscreteDistribution(dict.copy(self))

    def argMax(self):
        """
        Возвращает ключ с наибольшим значением
        """
        if len(self.keys()) == 0:
            return None
        all = list(self.items())
        values = [x[1] for x in all]
        maxIndex = values.index(max(values))
        return all[maxIndex][0]

    def total(self):
        """
        Возвращает сумму всех значений вероятностей
        """
        return float(sum(self.values()))

    def normalize(self):
        """
      Нормализуйте распределение таким образом, чтобы суммарное значение
       всех вероятностей ключей равнялось 1. Сотношение значений для всех
       ключей должно остаться прежним. В случае, когда суммарное значение
         равно 0, ничего не делайте.

         Тесты:

         >>> 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
        {}
        """
        "*** ВСТАВЬТЕ ВАШ КОД СЮДА ***"

        total = float(self.total())
        if total != 0:
            for key in self.keys():
                self[key] = self[key] / total

    def sample(self):
        """
        Формирует случайную выборку по распределению, представляемому
        в виде словаря, и возвращает ключ,
        соответствующий  случайной выборке.

        Тесты:

        >>> 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
        """
        "*** ВСТАВЬТЕ ВАШ КОД СЮДА ***"

        items = sorted(self.items())

        distribution = [i[1] for i in items]

        item_values = [i[0] for i in items]
        random_choice = random.random()

        i, total = 0, distribution[0]
        if self.total() != 1:
            self.normalize()
```

```
        while random_choice > total:
            i += 1
            total += distribution[i]

        return item_values[i]
```

Задание 1 (2 балла). Вероятность наблюдения

В этом задании необходимо реализовать метод getObservationProb базового класса InferenceModule, определяемого в файле inference.py. Метод должен принимать на вход наблюдение (зашумленное значение расстояния до призрака noisyDistance), позицию Пакмана pacmanPosition, позицию призрака ghostPosition, позицию тюремной камеры для призрака jailPosition и возвращать вероятность наблюдения noisyDistance для заданных положений Пакмана и призрака: P(noisyDistance | pacmanPosition, ghostPosition).

По сути метод реализует модель наблюдения (восприятия) СММ.

Разработанный код:

```
    def getObservationProb(self, noisyDistance, pacmanPosition, ghostPosition, jailPosition):
        """
        Возвращает вероятность P(noisyDistance | pacmanPosition, ghostPosition).
        """
        "*** ВСТАВЬТЕ ВАШ КОД СЮДА ***"

        if ghostPosition == jailPosition:
            if noisyDistance == None:
                return 1.0
            else:
                return 0.0

        if noisyDistance == None:
            return 0.0

        return busters.getObservationProbability(noisyDistance, manhattanDistance(pacmanPosition, ghostPosition))
```

Задание 2 (3 балла). Точный вывод на основе наблюдений

В этом задании необходимо реализовать метод observeUpdate класса ExactInference, определяемого в файле inference.py. Метод обновляет

распределение степеней уверенности агента в отношении позиций призрака, оцениваемых на основе данных, поступающих от сенсоров Пакмана. Необходимо реализовать онлайн-обновление степеней уверенности в соответствии с (6.19) при получении нового наблюдения observation. Метод observeUpdate должен обновлять степени уверенности для каждой возможной позиции призрака после получения наблюдения. Необходимо циклически выполнять обновления для всех значений переменной self.allPositions, которая включает в себя все легальные позиции призрака, а также специальную тюремную позицию. Степени уверенности представляются вероятностями того, что призрак находится в определенной позиции, и хранятся в виде объекта DiscreteDistribution в поле с именем self.beliefs, которое необходимо обновлять.

Разработанный код:

```
def observeUpdate(self, observation, gameState):
        """
      Обновляет степени уверенности агента в отношении позиций призраков
        на основе наблюдания observation и позиции Пакмана.
        observation – это зашумленное манхеттенское расстояние до
        отслеживаемого призрака.

      self.allPositions  - список возможных позиций призрака, включающий
        позицию тюрьмы. Вам необходимо рассматривать только те позиции,
        которые есть в self.allPositions.

        Модель обновления не является полностью стационарной: она может
        зависисеть от текущей позиции Пакмана. Это не проблема, если
текущая
        позиция Пакмана известна
        """

        "*** ВСТАВЬТЕ ВАШ КОД СЮДА ***"

        pacmanPosition = gameState.getPacmanPosition()
        jailPosition = self.getJailPosition()
        positions = self.allPositions
        noisyDistance = observation

        for possibleGhostPos in positions:
            self.beliefs[possibleGhostPos] = self.getObservationProb(
                noisyDistance,    pacmanPosition,    possibleGhostPos,
jailPosition)*self.beliefs[possibleGhostPos]
            self.beliefs.normalize()
```

```
        self.beliefs.normalize()
```

Задание 3 (3 балла). Точный вывод во времени

В предыдущем задании было реализовано обновление распределения степеней доверия на основе наблюдений. К счастью, наблюдения Пакмана — не единственный источник информации о том, где может быть призрак. Пакман также знает, как может двигаться призрак, а именно, что призрак не может пройти сквозь стену или более чем через одну ячейку за один временной шаг.

Представим следующий сценарий, в котором имеется один призрак. Пакман получает серию наблюдений, которые указывают на то, что призрак очень близко, но затем поступает одно наблюдение, которое указывает, что призрак очень далеко. Наблюдение, указывающие на то, что призрак находится очень далеко, вероятно, является результатом сбоя сенсора. Предварительное знание Пакманом правил движения призрака может снизить влияние этого наблюдения, поскольку Пакман знает, что призрак не может далеко переместиться за один шаг.

В этом задании необходимо реализовать метод elapseTime класса ExactInference. Метод elapseTime должен обновлять степени доверия для каждой возможной новой позиции призрака по истечении одного временного шага в соответствии с (6.18). При этом агент имеет доступ к распределению действий призрака через self.getPositionDistribution.

Разработанный код:

```
    def elapseTime(self, gameState):
        """
        Предсказывает степени уверенности агента в отношении позиций призраков
        в ответ на один шаг призрака, совершаемый из текущего состояния

        Модель перехода не обязательно стационарна: она может зависеть
        от текущей позиции Пакмана. Однако, это не проблема, т.к.
        позиция Пакмана известна.
        """
        "*** ВСТАВЬТЕ ВАШ КОД СЮДА ***"
```

```
# определяем возможные позции призрака
positions = self.allPositions
# создаем экземпляр распределения
beliefDict = DiscreteDistribution()

# выполняем итерации по всем возможным позициям призрака
for ghostPos in positions:
 # определяем распределение новых позиций призрака
 # по предыдущей позиции ghostPos
 newPosDist = self.getPositionDistribution(gameState, ghostPos)
 # для всех элементов распределения newPosDist
    for newPos, prob in newPosDist.items():
     # обновляем степени доверия возможных новых позиций
        beliefDict[newPos] = beliefDict[newPos] + \
            self.beliefs[ghostPos]*prob
 # нормализуем распределение
beliefDict.normalize
# сохраняем обновленное распределение
self.beliefs = beliefDict
```

Задание 4 (2 балла). Полное тестирование точного вывода

Теперь, когда Пакман знает, как использовать свои априорные знания о поведении призраков и свои наблюдения, он готов эффективно выслеживать призраков. В задании необходимо будет совместно использовать разработанные методы observUpdate и elapseTime, а также реализовать простую стратегию жадной охоты. В простой стратегии жадной охоты Пакман предполагает, что призрак находится в наиболее вероятной позиции поля игры в соответствии с его степенью уверенности, и поэтому он движется к ближайшему призраку. До этого момента Пакман выбирал допустимое действие случайно.

Реализуйте метод ChooseAction класса GreedyBustersAgent в файле bustersAgents.py. Ваш агент должен сначала найти наиболее вероятную позицию каждого непойманного призрака, а затем выбрать действие, которое ведет к ближайшему призраку. Чтобы найти расстояние между любыми двумя позициями pos1 и pos2, используйте метод self.distancer.getDistance(pos1, pos2). Чтобы найти следующую позицию после выполнения действия используйте вызов:

successorPosition = Actions.getSuccessor(position, action)

Вам предоставляется список LivingGhostPositionDistributions, элементы которого представляют распределения степеней уверенности о позициях каждого из еще непойманных призраков.

При правильной реализации ваш агент должен выиграть игру в тесте q4/3 gameScoreTest со счетом выше 700 очков как минимум в 8 из 10 раз.

Разработанный код:

```
def chooseAction(self, gameState):
        """
          Сначала вычисляет наиболее вероятную позицию каждого призрака,
        который еще не был пойман. Затем выбирает действие, перемещающее
          Пакмана к ближайшему призраку (в соответствии с mazeDistance).
          """
          # определяем позицию Пакмана
          pacmanPosition = gameState.getPacmanPosition()
          # формируем список допустимых действий Пакмана
          legal = [a for a in gameState.getLegalPacmanActions()]
          # формируем список распределений степеней уверенностей
          # о положении каждого из еще непойманных призраков
          livingGhosts = gameState.getLivingGhosts()
          livingGhostPositionDistributions = [
              beliefs for i, beliefs in enumerate(self.ghostBeliefs) if
livingGhosts[i+1]]

          "*** ВСТАВЬТЕ ВАШ КОД СЮДА***"
          ghostMaxProb = []
          # итерации по всем призракам
          for g in livingGhostPositionDistributions:
              # создаем список наиболее вероятных позиций призраков
              ghostMaxProb.append(g.argMax())

          minDist = []
          # итерации по всем допустимым действиям Пакмана

          for action in legal:
              # находим следующую позицию после действия action
              successorPosition   =   Actions.getSuccessor(pacmanPosition,
action)
           # для всех наиболее вероятных позиций призраков из ghostMaxProb
              for ghostPos in ghostMaxProb:
                  # создаем список расстояний от Пакмана до призрака
                  minDist.append(
                 (action,   self.distancer.getDistance(successorPosition,
ghostPos)))
          # находим минимальное расстояние до призрака
          minGhostDist = min([d for act, d in minDist])
          # находим действие act ведущее в сторону ближайшего призрака
          for act, d in minDist:
              if d == minGhostDist:
                  return act
```

Задание 5 (2 балла). Инициализация приближенного вывода

В нижеследующих заданиях (5,6 и 7) необходимо реализовать приближенный вероятностный вывод, основанный на алгоритме фильтрации частиц для отслеживания одного призрака.

В данном задании реализуйте методы initializeUniformly и getBeliefDistribution класса ParticleFilter в файле inference.py. Частица представляется позицией призрака. В результате применения метода initializeUniformly частицы должны быть равномерно (не случайным образом) распределены по допустимым позициям.

Метод getBeliefDistribution получает список частиц и отображает позиции частиц в соответствующее распределение вероятностей, представляемое в виде объекта DiscreteDistribution. Метод должен возвращать нормализованное распределение.

Разработанный код:

```
 def initializeUniformly(self, gameState):
        """
       Инициализирует список частиц self.particles .Частицы должны быть
         равномерно (не случайно) распределены по допустимым позициям.
         Использует self.numParticles для хранения числа частиц,
         а self.legalPositions для хранения допустимых позиций частиц.
         """
         self.particles = []
         "*** ВСТАВЬТЕ ВАШ КОД СЮДА ***"

         num = 0
         while num < self.numParticles:
             for p in self.legalPositions:
                 if num >= self.numParticles:
                     break
                 self.particles.append(p)
                 num += 1
...
def getBeliefDistribution(self):
        """
        Метод преобразует список частиц в соответствующее
        распределение степеней уверенности. Метод должен возвращать
        нормализованное распределение типа DiscreteDistribution.
        """
        "*** ВСТАВЬТЕ ВАШ КОД СЮДА ***"

        answer = util.Counter()
        for particle in self.particles:
```

```
            answer[particle] += 1
        answer.normalize()
        return answer
```

Задание 6 (3 балла). Приближенный вывод: обновление на основе наблюдения

Необходимо реализовать метод observeUpdate класса ParticleFilter в файле inference.py. Метод осуществляет обновление на основе наблюдения в соотвествии с алгоритом, описанным в п. 6.2.8. Наблюдение — это зашумленное манхеттенское расстояние до отслеживаемого призрака. Метод должен выполнять выборку из нормализованного распределения весов частиц и формировать новый список частиц self.particles. Вес частицы — это вероятность наблюдения с учетом положения Пакмана и местоположения частицы.

Имеется специальный случай, который необходимо учесть. Когда все частицы получают нулевой вес, список частиц следует повторно инициализировать, вызвав initializeUniformly.

Разработанный код:

```
def observeUpdate(self, observation, gameState):
        """
        Обновление списка частиц с учетом весов наблюдений.
        Наблюдение - это зашумленное манхеттенское расстояние
        до отслеживаемого призрака.
       Имеется специальный случай, который необходимо учесть. Когда все
        частицы получают нулевой вес, список частиц слудует повторно
        инициализировать, вызвав initializeUniformly.
        """
        "*** ВСТАВЬТЕ ВАШ КОД СЮДА ***"

        weights = DiscreteDistribution()
        resample = []
        # определяем позиции Пакмана и тюрьмы

        pacmanPosition = gameState.getPacmanPosition()
        jailPosition = self.getJailPosition()
        # для каждой позиции частицы
        for pos in self.particles:
            # определяем степень уверенности наблюдения при заданных
          # pacmanPosition, pos, jailPosition и аккумулируем в виде веса
            weights[pos] += self.getObservationProb(
                observation, pacmanPosition, pos, jailPosition)
```

```
            # если частицы получают нулевой вес
        if weights.total() == 0:
            # то инициализаируем повторно список частиц
            self.initializeUniformly(gameState)
        # иначе
        else:
            # нормазизуем распределение весов
            weights.normalize()
         # формируем список частиц путем выборки из распределния весов
            self.particles = [weights.sample()
                              for _ in range(int(self.numParticles))]
```

Задание 7 (3 балла). Приближенный вывод: обновление во времени

Реализуйте метод elapseTime класса ParticleFilter в файле inference.py. Метод должен сформировать новый список частиц self.particles с учетом изменения состояний игры во времени. Используйте алгоритм обновления во времени, описанный в п. 6.2.8.

Разработанный код:

```
def elapseTime(self, gameState):
        """
        Выполняет выборку следующего состояния каждой частицы на основе
        её текущего состояния и состояния игры
        """
        "*** ВСТАВЬТЕ ВАШ КОД СЮДА ***"

        elapseDist = DiscreteDistribution()
        sample = self.particles
        for pos in sample:
            newPosDist = self.getPositionDistribution(gameState, pos)
            for newPos, prob in newPosDist.items():
                elapseDist[newPos] += prob
        elapseDist.normalize()
        self.particles = [elapseDist.sample()
                          for _ in range(int(self.numParticles))]
```

Задание 8 (1 балл). Инициализация при совместной фильтрации частиц

В задании рассматривается случай, когда имеется несколько призраков. Поскольку модели перехода призраков больше не являются независимыми, все призраки должны отслеживаться совместно с использованием динамической сети Байеса (ДСБ), которая является обобщением СММ.

ДСБ с двумя призраками (a и b) изображена на рисунке 6.4. На рисунке скрытые переменные G представляют положения призраков, а переменные

свидетельств E представляют собой зашумленные расстояния до каждого из призраков. Представленную структуру ДСБ можно распространить на большее количество призраков.

В заданиях ниже необходимо реализовать алгоритм вывода, основанный на фильтрации частиц, который одновременно отслеживает несколько призраков.

Каждая частица (полная выборка на временном шаге) представляется кортежем позиций призраков, показывающим, где призраки находятся в данный момент.

Предоставляемый вам программный код уже подготовлен для извлечения маргинальных распределений по каждому призраку с помощью алгоритма совместного отслеживания призраков, который вы реализуете.

В данном задании завершите определение метода initializeUniformly класса JointParticleFilter в файле inference.py. Метод должен обеспечить начальное равномерное распределение частиц. Как и в задании 5, частицы хранятся в списке частиц self.particles.

Разработанный код:

```
def initializeUniformly(self, gameState):
        """
        Инициализирует частицы равномерным априорным распределением.
        Частицы должны быть равномерно распредены по позициям, чтобы
        гарантировать равномерное априорное распреление.
        """
        self.particles = []
        "*** ВСТАВЬТЕ ВАШ КОД СЮДА ***"

        permutations = list(itertools.product(
            self.legalPositions, repeat=self.numGhosts))
        random.shuffle(permutations)
        size = self.numParticles
        i = 0
        while i < size:
            for particle in permutations:
                self.particles.append(particle)
                i += 1
```

Задание 9 (3 балла). Обновление на основе наблюдения при совместной фильтрации частиц

В задании необходимо реализовать метод observUpdate класса JointParticleFilter файла inference.py. Метод должен обеспечивать взвешивание и повторное сэмплирование всех частиц с учетом правдоподобия наблюдаемого расстояния до каждого из призраков. Метод аналогичен одноименному методу класса ParticleFilter, но обеспечивает обработку наблюдений для нескольких призраков.

Также реализация метода должна обрабатывать особый случай, когда все частицы получают нулевой вес. В этом случае список частиц self.particles следует воссоздать из априорного распределения, вызвав initializeUniformly.

Разработанный код:

```
    def observeUpdate(self, observation, gameState):
        """
        Обновляет степени доверия на основе позцции Пакмана и
        наблюдений расстояний.

        Наблюдения - это зашумленное манхеттенское расстояние до всех
        отслеживаемых призраков

      Есть особый случай, который необходимо учесть при реализции метода.
       Когда все частицы получают нулевой вес, список частиц должен быть
        переинициализирован заново путем вызова initializeUniformly. При
этом
        могут использоваться общие методы класса DiscreteDistribution
        """
        "*** ВСТАВЬТЕ ВАШ КОД СЮДА ***"

        pacmanPosition = gameState.getPacmanPosition()
        newPD = DiscreteDistribution()

        for ghostPositions in self.particles:

            prob = 1
            for i in range(self.numGhosts):
                prob *= self.getObservationProb(
                    observation[i], pacmanPosition, ghostPositions[i],
self.getJailPosition(i))
            newPD[ghostPositions] += prob

        if newPD.total() == 0:
            self.initializeUniformly(gameState)
        else:
```

```
                newPD.normalize()

                self.particles = [newPD.sample() for _ in
range(self.numParticles)]
```

Задание 10 (3 балла). Обновление во времени при совместной фильтрации частиц

В задании необходимо завершить определение метода elapseTime класса JointParticleFilter в файле inference.py, чтобы корректно выполнять ресэмплирование частиц в ДСБ совместного отслеживания призраков. В частности, необходимо учитывать, что каждый призрак перемещается в новую позицию, обусловленную позициями всех призраков на предыдущем временном шаге.

Разработанный код:

```
    def elapseTime(self, gameState):
            """
            Формирует выборку следующего состояния частицы на основе её
текущего
            состояния и состояния игры
            """
            newParticles = []
            for oldParticle in self.particles:
                newParticle = list(oldParticle)  # Список позиций призраков

                # цикл обновления всех частиц
                "*** ВСТАВЬТЕ ВАШ КОД СЮДА ***"

                for i in range(self.numGhosts):
                    newPosDist = self.getPositionDistribution(
                        gameState, newParticle, i, self.ghostAgents[i])
                    newParticle[i] = newPosDist.sample()

                """*** КОНЕЦ ВАШЕГО КОДА ***"""
                newParticles.append(tuple(newParticle))
            self.particles = newParticles
```

ВЫВОДЫ

В ходе выполнения лабораторной работы были исследованы методы точного и приближенного вероятностного вывода с использованием динамических сетей Байеса, приобретены навыки программирования интеллектуальных агентов, знания которых представляются условными высказываниями с определенной степенью уверенности.