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.

Порядок выполнения лабораторной работы:

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, СММ 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

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 docтесты. 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.

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.

8. В начальной части кода задания 4 определяется позиция Пакмана pacmanPosition, формируются списки допустимых действий Пакмана legal и распределений степеней уверенности о позициях ещё непойманных призраков livingGhostPositionDistributions

**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]]**

Вам следует с помощью списка livingGhostPositionDistributions найти наиболее вероятные позиции каждого непойманного призрака и сохранить их, например, в списке ghostMaxProb.

Затем в цикле для всех допустимых действий Пакмана с помощью вызова

**successorPosition = Actions.getSuccessor(pacmanPosition, action)**

определите следующую позицию после действия action и для всех наиболее вероятных позиций призраков из ghostMaxProb создайте список minDist из пар в виде кортежей (действие, расстояние), где расстояние – это расстояние от Пакмана до призрака:

**minDist.append((action, self.distancer.getDistance(successorPosition, ghostPos)))**

Анализируя этот список, найдите минимальное расстояние до призрака

**minGhostDist = min([d for act, d in minDist])**

и верните действие act, ведущее в сторону ближайшего призрака:

**for act, d in minDist:**

**if d==minGhostDist:**

**return act**

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

**python autograder.py -q q4**

Если вы хотите запустить этот тест без графики, вы можете добавить параметр -- no-graphics.

Внесите код метода ChooseAction и результаты тестирования 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 результаты используйте команду:

**python autograder.py -q q7**

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

В лабораторной работе необходимо создать Пакман-агента, который используют сенсоры для обнаружения невидимых призраков. Такой агент, кроме поиска одиночных призраков сможет охотиться с высокой эффективностью на группы из нескольких движущихся призраков.

**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

Листинг 1 – Код класса DiscreteDistribution

class DiscreteDistribution(dict):

"""

Класс для работы с распределением,

представляемым в виде словаря c

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 = self.total()

if total != 0:

for k, v in self.items():

self[k] = v / 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 \*\*\*"

u = random.random()

alpha = 0.0

for key, value in self.items():

if alpha <= u < alpha + (value / self.total()):

return key

alpha += value / self.total()

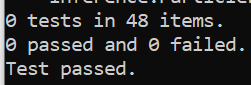


Рисунок 1 – Прохождение тестов

**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.

Листинг 2 – Код метода getObservationProb

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

else:

if noisyDistance == None:

return 0.0

else:

trueDistance = manhattanDistance(pacmanPosition, ghostPosition)

return busters.getObservationProbability(noisyDistance, trueDistance)

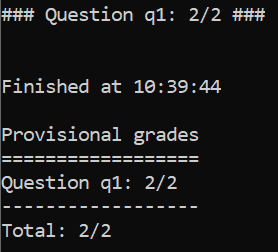


Рисунок 2 – Прохождение тестов 2/2

**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.

Листинг 3 – Код метода observeUpdate

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

"""

oldPD = self.beliefs

pacmanPosition = gameState.getPacmanPosition()

jailPosition = self.getJailPosition()

newPD = DiscreteDistribution()

for ghostPosition in self.allPositions:

newPD[ghostPosition] = self.getObservationProb(observation,

pacmanPosition, ghostPosition, jailPosition) \* oldPD[ghostPosition]

self.beliefs = newPD

self.beliefs.normalize()

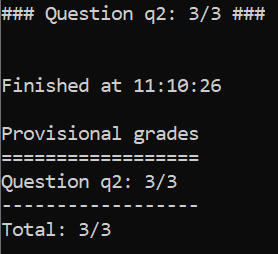


Рисунок 3 – Прохождение тестов 3/3

**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.

Листинг 4 – Код метода elapseTime

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 \*\*\*"

oldPD = self.beliefs

newPD = DiscreteDistribution()

for oldPos in self.allPositions:

newPosDist = self.getPositionDistribution(gameState, oldPos)

for newPos in self.allPositions:

if newPosDist[newPos] > 0:

newPD[newPos] += newPosDist[newPos] \* oldPD[oldPos]

self.beliefs = newPD

self.beliefs.normalize()

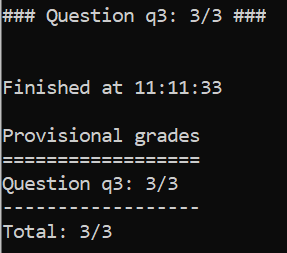


Рисунок 4 – Прохождение тестов 3/3

**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.

Листинг 5 – Код метода

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\*\*\*"

GhostPositions = [LGPD.argMax() for LGPD in livingGhostPositionDistributions]

nearestGhost = min(GhostPositions, key=lambda ghostPos: self.distancer.getDistance(pacmanPosition, ghostPos))

action = min(legal,

key=lambda a: self.distancer.getDistance(Actions.getSuccessor(pacmanPosition, a), nearestGhost))

return action

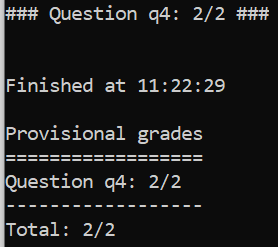


Рисунок 5 – Прохождение тестов 2/2

**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.

Листинг 6 – Код класса ParticleFilter

class ParticleFilter(InferenceModule):

def \_\_init\_\_(self, ghostAgent, numParticles=300):

InferenceModule.\_\_init\_\_(self, ghostAgent)

self.setNumParticles(numParticles)

def setNumParticles(self, numParticles):

self.numParticles = numParticles

def initializeUniformly(self, gameState):

self.particles = []

self.particles += self.legalPositions \* (self.numParticles // len(self.legalPositions))

self.particles += self.legalPositions[:(self.numParticles % len(self.legalPositions))]

def observeUpdate(self, observation, gameState):

raiseNotDefined()

def elapseTime(self, gameState):

raiseNotDefined()

def getBeliefDistribution(self):

belief = DiscreteDistribution()

for particle in self.particles:

belief[particle] += 1

belief.normalize()

return belief

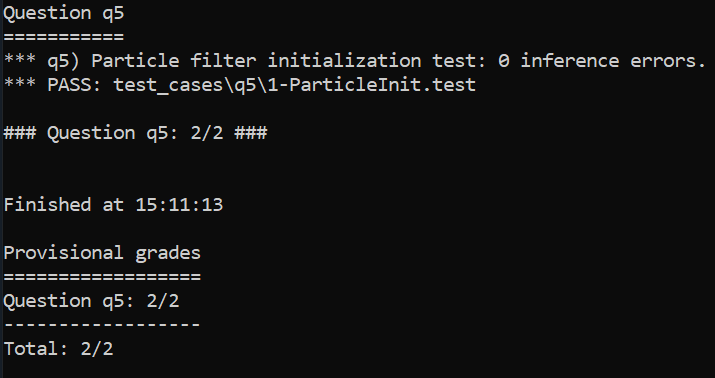


Рисунок 6 – Прохождение тестов 2/2

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

Необходимо реализовать метод **observUpdate** класса **ParticleFilter** в файле **inference.py**. Метод осуществляет обновление на основе наблюдения в соответствии с алгоритмом, 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.

Листинг 7 – Код метода observUpdate

def observeUpdate(self, observation, gameState):

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))]

#raiseNotDefined()

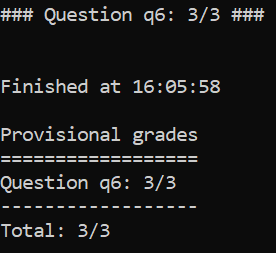


Рисунок 7 – Прохождение тестов 3/3

**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.

Листинг 8 – Код метода elapseTime

def elapseTime(self, gameState):

elapseDist=DiscreteDistribution()

# определяем выборку в виде списка позиций частиц

sample=self.particles

# для каждой позиции частицы

for pos in sample:

# находим распределение в следующей возможной позиции

newPosDist = self.getPositionDistribution(gameState, pos)

# для всех элементов распределения newPosDist

for newPos, prob in newPosDist.items():

# обновляем степени доверия возможных новых позиций

elapseDist[newPos]+=prob

# нормализуем распределение

elapseDist.normalize()

# формируем новый список частиц путем выборки из распределния

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

#raiseNotDefined()

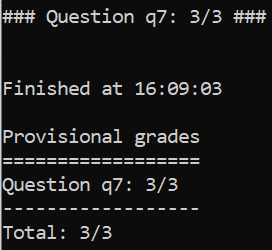


Рисунок 8 – Прохождение тестов 3/3

**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.

Листинг 9 – Код метода initializeUniformly

def initializeUniformly(self, gameState):

#self.particles = []

self.particles = []

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

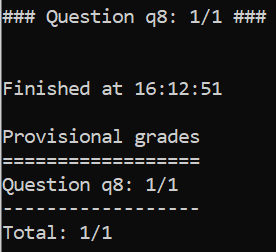


Рисунок 9 – Прохождение тестов 1/1

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

В задании необходимо реализовать метод **оbservUpdate** 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.

Листинг 10 – Код метода оbservUpdate

def observeUpdate(self, observation, gameState):

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)]

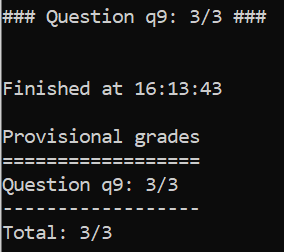


Рисунок 10 – Прохождение тестов 3/3

**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.

Листинг 11 – Код метода elapseTime

def elapseTime(self, gameState):

"""

Sample each particle's next state based on its current state and the

gameState.

"""

newParticles = []

for oldParticle in self.particles:

newParticle = list(oldParticle) # A list of ghost positions

# now loop through and update each entry in newParticle...

"\*\*\* YOUR CODE HERE \*\*\*"

for i in range(self.numGhosts):

newPosDist = self.getPositionDistribution(gameState, newParticle, i, self.ghostAgents[i])

newParticle[i] = newPosDist.sample()

"""\*\*\* END YOUR CODE HERE \*\*\*"""

newParticles.append(tuple(newParticle))

self.particles = newParticles

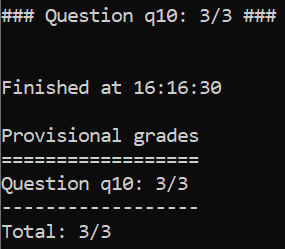


Рисунок 11 – Прохождение тестов 3/3

Выводы:

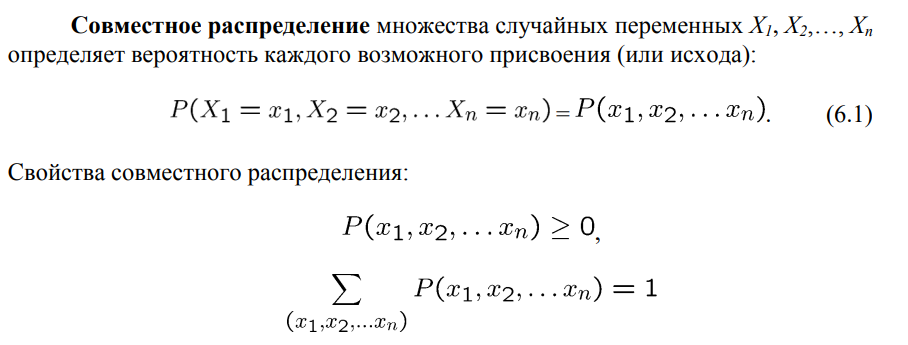
В ходе выполнения лабораторной работы были исследованы методы точного и приближенного вероятностного вывода с использованием динамических сетей Байеса, приобретены навыки программирования интеллектуальных агентов, знания которых представляются условными высказываниями с определенной степенью уверенности.

6.6.1 Объясните, что понимают под степенью уверенности высказываний?

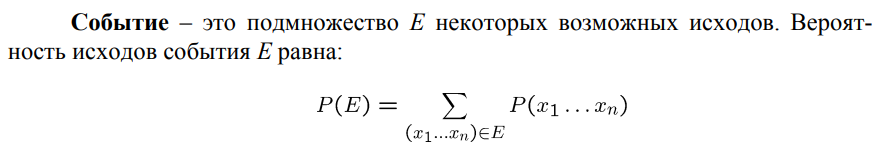
**Степень уверенности высказывания** указывает на уровень доверия или вероятность того, что данное высказывание является истинным или верным. Она представляет собой меру убежденности в правильности или достоверности информации, выраженной в высказывании.

Основным инструментом, применяемым для обработки степеней уверенности таких высказываний и осуществления вывода, является теория вероятностей, в которой каждому высказыванию присваивается числовое значение степени уверенности в диапазоне от 0 до 1

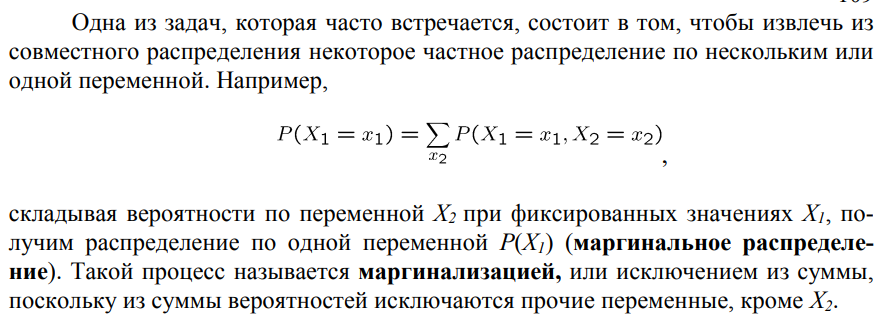
6.6.2. Что понимают под совместным распределением случайных переменных? Свойства совместного распределения?



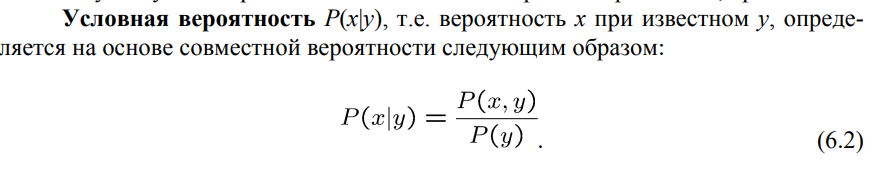
6.6.3. Что понимают под событием?



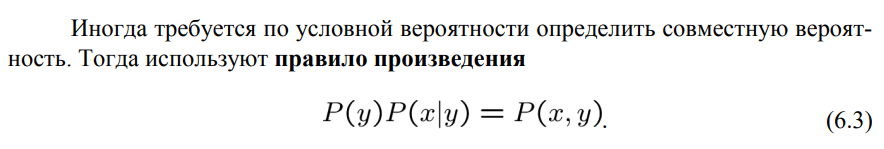
6.6.4. Что такое маргинальное распределение? Как его получить из совместного распределения случайных переменных?



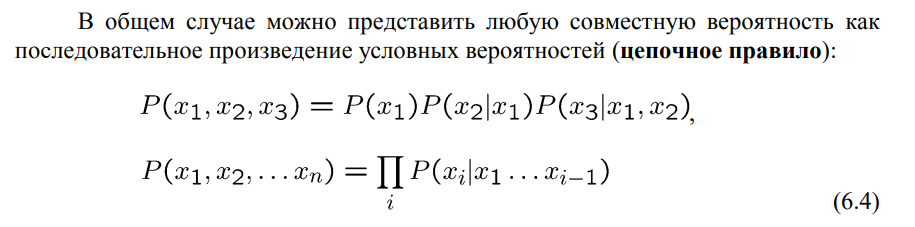
6.6.5. Запишите формулу условной вероятности x при известном y.



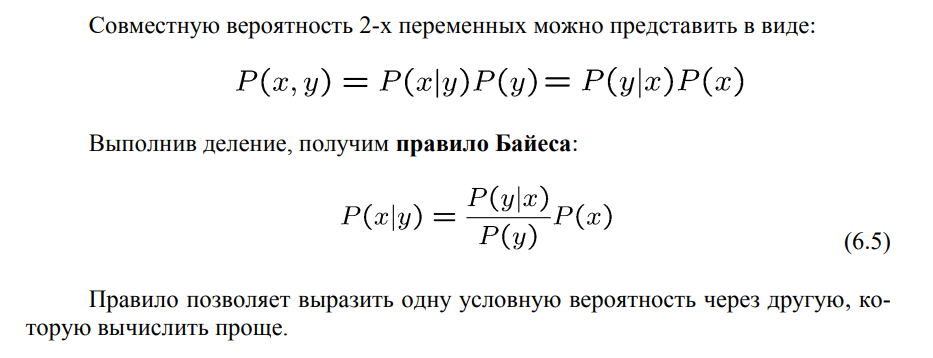
6.6.6. Запишите правило произведения для 2-х переменных.



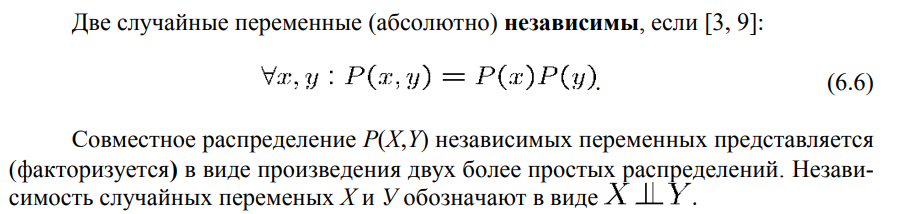
6.6.7. Запишите цепочное правило.



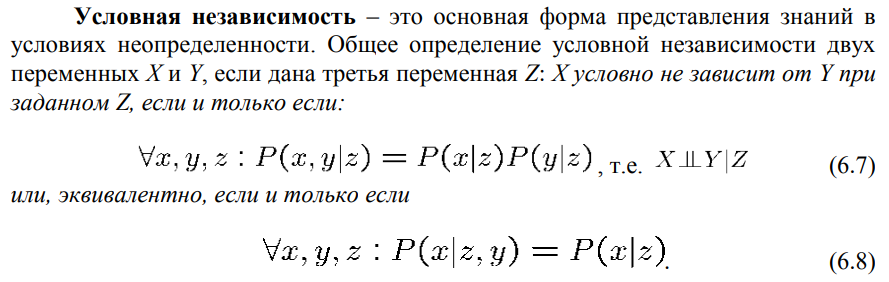
6.6.8. Запишите правило Байеса и объясните его.



6.6.9. Определите понятие независимости 2-х случайных переменных.



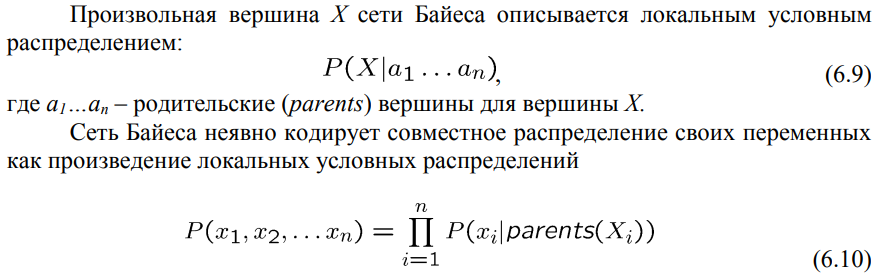
6.6.10. Определите понятие условной независимости 2-х случайных переменных при заданной третьей переменной.



6.6.11. Определите понятие сети Байеса.

**Байесовская сеть** представляет совместную вероятность множества n дискретных случайных переменных Х1, Х2,…, Хn в форме направленного ациклического графа. Каждая вершина графа представляется случайной переменной, с которой связана таблица условных вероятностей, содержащая вероятность каждого состояния переменной с учетом её родителей в графе. Ребра графа обозначают взаимодействия переменных. Интуитивный смысл стрелки вдоль ребра сети Байеса указывает на то, что вершина X оказывает непосредственное влияние на вершину Y [3, 9].

6.6.12. Запишите выражение полного совместного распределения для сети Байеса и объясните его.



6.6.13. Сформулируйте критерии D-разделенности для различных триплетов подсетей Байеса.

Утверждения локальной условной независимости можно проверить (верифицировать) непосредственно с помощью структуры сети, используя критерий, называемый D-разделенностью [2, 3, 9].

D-разделенность - это концепция, связанная с графическими моделями, такими как сети Байеса. Она указывает на независимость между наборами переменных в графе, при условии некоторого набора других переменных.

Для различных триплетов подсетей Байеса (три переменные, связанные между собой), критерии D-разделенности могут быть сформулированы следующим образом:

1. X независимо от Z при условии Y:  
   Это означает, что переменная X статистически независима от переменной Z, при условии, что переменная Y известна. Формально, это можно записать как P(X, Z | Y) = P(X | Y) \* P(Z | Y), где \* обозначает произведение и P(A | B) обозначает условную вероятность A при условии B.
2. X независимо от Y при условии Z:  
   Это означает, что переменная X статистически независима от переменной Y, при условии, что переменная Z известна. Формально, это можно записать как P(X, Y | Z) = P(X | Z) \* P(Y | Z).
3. Y независимо от Z при условии X:  
   Это означает, что переменная Y статистически независима от переменной Z, при условии, что переменная X известна. Формально, это можно записать как P(Y, Z | X) = P(Y | X) \* P(Z | X).

6.6.14. Что понимают под точным и приближенным вероятностным выводом?

Представление предметной области в виде модели байесовской сети позволяет осуществлять вероятностные логические выводы. **Вероятностный вывод** в сетях Байеса предполагает вычисление вероятностей переменных запроса через вероятности других известных переменных. Известны алгоритмы **точного вероятностного вывода** (например, вывод с использованием метода прямого (полного) перебора или метода исключения переменных), которые характеризуются большой вычислительной сложностью. Точный вероятностный вывод в больших многосвязных сетях, часто является неосуществимым. Поэтому были разработаны **методы приближенного вероятностного вывода**, основанные на формировании случайных выборок из распределений. Вероятностный вывод на основе выборок происходит быстрее, чем вычисление ответа на запрос, например, методом исключения переменных. Точность вывода зависит от количества формируемых выборок.

6.6.15. Сформулируйте алгоритм формирования случайной выборки из заданного распределения.

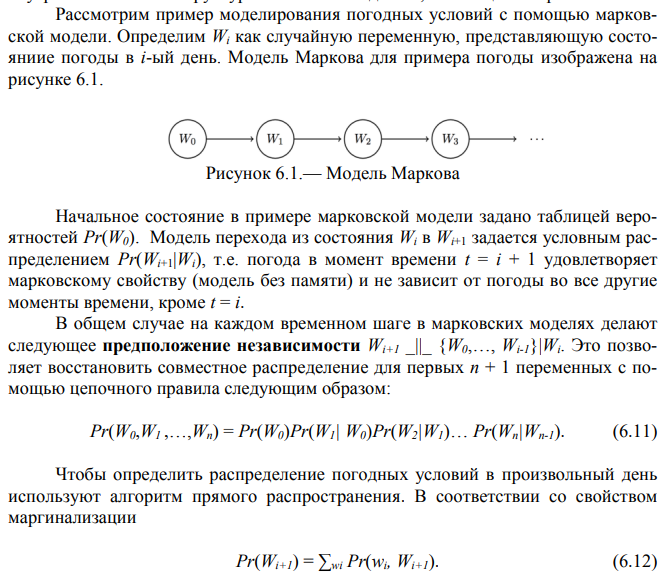
Алгоритм формирования выборки из заданного дискретного распределения можно представить в виде 2-х шагов:

**1.** Получить случайное число u из равномерного распределения в интервале [0, 1). Например, можно использовать функцию random() языка Пайтон;

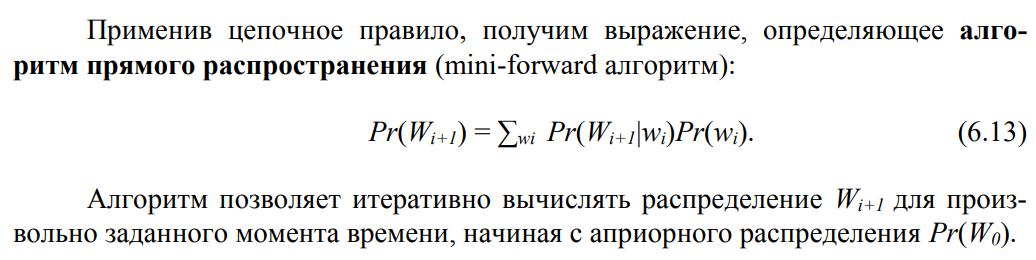
**2.** Преобразовать это значение u в выборочное значение дискретной случайной переменной с учетом заданного распределения, связав u с некоторым диапазоном, ширина которого равна задаваемой распределением вероятности.

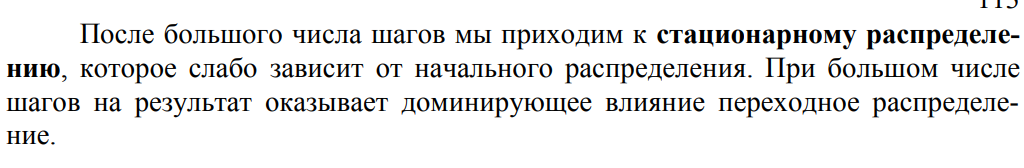
6.6.16. Приведите пример марковской модели. Какие предположения независимости используют в марковской модели?

Сети Байеса представляют собой универсальную структуру, используемую для компактного представления отношений между случайными величинами. **Марковскую мод**ель можно рассматривать как аналог байесовской сети в виде внутренне связанной структуры бесконечной длины, зависящей от времени.



6.6.17. Определите алгоритм прямого распространения для марковской модели.





6.6.18. Определите понятие скрытой марковской модели.

**Скрытые марковские модели** (СММ) описываются с помощью двух вероятностных процессов: скрытого процесса смены состояний цепи Маркова и наблюдаемых значений свидетельств, формируемых при смене состояний.

В качестве примера на рисунке 6.2 изображена скрытая модель Маркова для моделирования погоды. В отличие от обычной марковской модели, СММ содержит два типа узлов: скрытые узлы Wi , которые являются **переменными состояния** и представляют погоду в i-ый день, и наблюдаемые узлы Fi , которые представляют **переменные, называемые свидетельствами (наблюдениями)**. В рассматриваемом примере свидетельства Fi представляют прогноз погоды в i-ый день.

6.6.19. Сформулируйте задачу фильтрации для СММ.

Примение к СММ процедур, аналогичных байесовскому сэмплированию (взятию выборок), называется **фильтрацией частиц** и включает в себя моделирование движения набора частиц через граф состояний для аппроксимации распределения вероятности (доверий) рассматриваемой случайной величины в требуемый момент времени. Частица в этом случает представляет возможное выборочное значение случайной величины. При этом вместо хранения полных таблиц вероятностей, отображающих каждое состояние в вероятность, хранят список из n частиц, в котором каждая частица может находиться в одном из d состояний.

6.6.20. Какие предположения независимости используют в СММ?

6.6.21. Сформулируйте правило обновления во времени и правило на основе наблюдений для СММ.

**Обновление во времени** (Time Elapse Update) — обновление выборочного значения каждой частицы в соответствии с моделью переходной вероятности. Для частицы в состоянии ti выполняется случайная выборка обновленного значения из переходного распределения Pr(Ti+1 | ti).

**Обновление на основе наблюдения** (Observation Update). Этот этап немного сложнее. Здесь используется модель сенсора Pr(Fi |Ti) для взвешивания каждой частицы в соответствии с вероятностью, определяемой наблюдаемым свидетельством и состоянием частицы. В частности, частице в состоянии ti при свидетельстве fi , поступающим от некоторого сенсора, присваивается вес Pr(fi |ti).

Алгоритм обновления на основе наблюдений следующий:

1. Рассчитайте веса всех частиц в соответствии с Pr(fi |ti);

2. Вычислите суммарный вес каждого состояния;

3. Если сумма всех весов во всех состояниях равна 0, повторно инициализируйте все частицы;

4. Иначе нормализуйте распределение по отношению к суммарному весу и выполните выборки из этого распределения.

6.6.22. Сформулируйте алгоритм прямого распростанения для СММ.

Объединение полученных правил дает итерационный алгоритм, известный как алгоритм прямого распространения для СММ (аналог mini-forward алгоритма для обычной марковской модели). Алгоритм включает два отдельных шага:

1. Обновление B’(W i+1) по B(Wi) на одном шаге во времени;

2. Обновление B(Wi) на основе наблюдения, т.е. определение B(Wi+1) по B’(W i+1).

6.6.23. Что такое фильтрация частиц применительно к СММ?

6.6.24. Сформулируйте правило обновления во времени и правило на основе наблюдений, применяемые при фильтрации частиц.