5. ЛАБОРАТОРНАЯ РАБОТА № 5 «ИССЛЕДОВАНИЕ СЕТЕЙ БАЙЕСА И СММ»

5.1. Цель работы

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

5.2. Краткие теоретические сведения

5.2.1. Неопределенность и степени уверенности высказываний

Интеллектуальные агенты почти никогда не имеют доступа ко всей информации о среде функционирования. Поэтому они действуют в условиях неопределенности. При функционировании агента в среде с неопределенностями знания агента в лучшем случае позволяют сформировать относящиеся к делу высказывания только с определенной **степенью уверенности/убежденности** (degree of belief) [1, 2, 3]. 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.

Напомним основные понятия и соотношения, используемые при построении вероятностных моделей [3, 9].

5.2.2. Совместные и условные вероятности

Совместное распределение множества случайных переменных $X_1, X_2,\ldots, X_n$ определяет вероятность каждого возможного присвоения (или исхода):

$$P(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n) = P(x_1, x_2, \ldots x_n). \qquad (5.1)$$

Свойства совместного распределения:

$$P(x_1, x_2, \ldots x_n) \geq 0,$$

$$\sum_{(x_1, x_2, \ldots x_n)} P(x_1, x_2, \ldots x_n) = 1$$

Событие – это подмножество E некоторых возможных исходов. Вероятность исходов события E равна:

$$P(E) = \sum_{(x_1 \ldots x_n) \in E} P(x_1 \ldots x_n)$$

Одна из задач, которая часто встречается, состоит в том, чтобы извлечь из совместного распределения некоторое частное распределение по нескольким или одной переменной. Например,

$$P(X_1 = x_1) = \sum_{x_2} P(X_1 = x_1, X_2 = x_2),$$

складывая вероятности по переменной X_2 при фиксированных значениях X_1, получим распределение по одной переменной $P(X_1)$ (**маргинальное распределение**). Такой процесс называется **маргинализацией,** или исключением из суммы, поскольку из суммы вероятностей исключаются прочие переменные, кроме X_1.

Условная вероятность $P(x|y)$, т.е. вероятность x при известном y, определяется на основе совместной вероятности следующим образом:

$$P(x|y) = \frac{P(x,y)}{P(y)}. \tag{5.2}$$

Иногда требуется по условной вероятности определить совместную вероятность. Тогда используют **правило произведения**

$$P(y)P(x|y) = P(x,y). \tag{5.3}$$

В общем случае можно представить любую совместную вероятность как последовательное произведение условных вероятностей (**цепочное правило**):

$$P(x_1, x_2, x_3) = P(x_1)P(x_2|x_1)P(x_3|x_1, x_2),$$

$$P(x_1, x_2, \ldots x_n) = \prod_i P(x_i|x_1 \ldots x_{i-1}) \tag{5.4}$$

Совместную вероятность 2-х переменных можно представить в виде:

$$P(x,y) = P(x|y)P(y) = P(y|x)P(x)$$

Выполнив деление, получим **правило Байеса**:

$$P(x|y) = \frac{P(y|x)}{P(y)}P(x) \tag{5.5}$$

Правило позволяет выразить одну условную вероятность через другую, которую бывает вычислить проще.

5.2.3. Независимость и условная независимость

Две случайные переменные (абсолютно) **независимы**, если [3, 9]:

$$\forall x, y : P(x,y) = P(x)P(y). \tag{5.6}$$

Совместное распределение $P(X,Y)$ независимых переменных представляется (факторизуется**)** в виде произведения двух более простых распределений. Независимость случайных переменых X и Y обозначают в виде $X \perp\!\!\!\perp Y$.

Общее определение **условной независимости** двух переменных *X* и *Y*, если дана третья переменная *Z*: *X условно не зависит от Y при заданном Z, если и только если:*

$$\forall x, y, z : P(x, y|z) = P(x|z)P(y|z) \text{, т.е. } X \perp\!\!\!\perp Y|Z \quad (5.7)$$

или, эквивалентно, если и только если

$$\forall x, y, z : P(x|z, y) = P(x|z). \quad (5.8)$$

Разработка методов декомпозиции крупных предметных областей на слабо связанные подмножества предметных переменных с помощью свойства условной независимости стало одним из наиболее важных достижений в истории искусственного интеллекта [9].

5.2.4. Байесовские сети

Байесовская сеть представляет совместную вероятность множества *n* случайных переменных X_1, X_2,…, X_n в форме направленного ациклического графа. Каждая вершина графа представляется случайной переменной, с которой связана **таблица условных вероятностей** (CPT – conditional probability table), содержащая вероятности переменной с учетом её условных родительских переменных в графе. Ребра графа обозначают взаимодействия переменных. Однако, важно помнить, что они не означают причинные связи [3, 9].

Произвольная вершина *X* сети Байеса описывается локальным условным распределением:

$$P(X \mid A_1, A_2, \ldots A_n) , \quad (5.9)$$

где $A_1, A_2, \ldots A_n$ – родительские (*parents*) переменные (вершины) для *X*. CPT переменной содержит *n*+2 столбцов: один для хранения значений каждой из *n* родитедьских переменных, один для *X* и один для значения условной вероятности *X* при заданом значении *Y*.

При известных CPT для каждой вершины сети Байеса, можно вычислить вероятности заданных совместных присваиваний всех переменных сети на основе правила:

$$P(X_1, X_2, \ldots X_n) = \prod_{i=1}^{n} X_i \mid parents(X_i) \quad (5.10)$$

Утверждения условной независимости, кодируемых сетью Байеса, можно проверить (верифицировать) путём анализа структуры сети, используя критерий, называемый **D-разделенностью** [2, 3, 9].

Отдельные элементы произведения (5.10) в ходе поиска ответов на запросы называют **факторами**. **Фактор** хранит таблицу “вероятностей”, хотя сумма записей в таблице не обязательно равна 1. Фактор в общем случае имеет форму $f(X_1,\ldots,X_m, y_1,\ldots,y_n \mid Z_1,\ldots,Z_p, w_1,\ldots,w_q)$. Напомним, что строчные буквы обозначают переменные, которые уже получили значение. Для каждого возможного значения

переменных X_i и Z_j в таблице вероятностей фактора хранится одно число. Переменные Z_j и w_k называются **условными**, а переменные X_i и y_l — **безусловными**. В сети Байеса фактор, представляемый **таблицей условных вероятностей** (CPT), обладает двумя свойствам: 1) элементы в сумме должны давать 1 для каждого назначения условных переменных; 2) фактор имеет ровно одну безусловную переменную.

Вероятностный вывод предполагает вычисление вероятностей переменных запроса через вероятности других известных переменных. Известны алгоритмы **точного вероятностного вывода** (например, вывод путем перечисления или путем исключения переменных), которые характеризуются большой вычислительной сложностью. Точный вероятностный вывод при большом числе переменных часто неосуществим. Поэтому были разработаны **методы приближенного вероятностного вывода**, основанные на формировании случайных выборок из распределений. Вероятностный вывод на основе выборок осуществляется быстрее, чем вычисление ответа на запрос, например, путем исключения переменных. Точность вывода зависит от количества формируемых выборок.

5.2.5. Вывод путем перечислений

Располагая совместным распределением, мы можем вычислить любое желаемое распределение вероятностей, представляемое **запросом** $P(Q_1,\ldots,Q_m \mid e_1,\ldots,e_n)$, где Q_i –переменные запроса, e_1 – переменные свидетельства, которые являются наблюдаемыми переменными, значения которых известны.

Вывод путем перечислений (**Inference By Enumeration**) реализуется следующим алгоритом:

1. Выбираем все строки таблицы совместного распределения, которые содержат наблюдаемые переменные свидетельств;
2. Суммируем строки, содержащие скрытые переменные (исключаем). Скрытые переменные – это те переменные, которые присутствуют в общем совместном распределении, но отсутсвуют в запросе;
3. Нормализуем таблицу так, чтобы она представляла собой распределение вероятностей (т. е. сумма значений равнялась 1).

5.2.6. Вывод путем исключения переменных

Альтернативный подход заключается в исключении скрытых переменных по одной. Чтобы исключить скрытую переменную Y (не входит в запрос) необходимо:

1. Объединить (перемножить) все **факторы**, включающие Y, например: $P(X|Y)\ P(Y)=P(X, Y)$;

2. Выполнить суммирование по Y (исключить Y): $P(X) = \sum_Y P(X, Y)$.

Важно отметить, что вывод путем исключения переменных улучшает вывод путем перечисления только в том случае, если размер наибольшего фактора ограничен разумным значением.

5.2.7. Приближенный вывод в сетях Байеса: формирование выборки

Иной подход к построению вероятностных выводов заключается в неявном вычислении вероятностей запроса путем простого подсчета выборочных значений. Это не даёт точного решения, но часто решение бывает достаточно хорошим, особенно если учитывать огромную экономию в вычислениях.

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.

Пример. Пусть задано распределение цветов (таблица 5.1), которое в памяти можно хранить в виде словаря с набором пар **{C: P(C)}**.

Таблица 5.1. – Распределение цветов

C	P(C)
red	0.6
green	0.1
blue	0.3

Введем диапазоны, ширина которых равна вероятностям цветов:

$$0 \leq u < 0.6, \rightarrow C = red$$
$$0.6 \leq u < 0.7, \rightarrow C = green$$
$$0.7 \leq u < 1, \rightarrow C = blue$$

Тогда, если **random()** возвращает $u = 0.83$, то выборное значение C = blue. После формирования большого числа выборок можно получить набор выборочных значений с вероятностями, сходящимися к заданному распределению.

5.2.8. Марковские Модели

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

Рассмотрим пример моделирования погодных условий с помощью марковской модели. Определим W_i как случайную переменную, представляющую состояниие погоды в *i*-ый день. Модель Маркова для примера погоды изображена на рисунке 5.1.

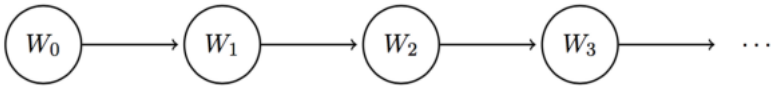


Рисунок 5.1.— Модель Маркова

Начальное состояние в примере марковской модели задано таблицей вероятностей $Pr(W_0)$. Модель перехода из состояния W_i в W_{i+1} задается условным распределением $Pr(W_{i+1}|W_i)$, т.е. погода в момент времени $t = i + 1$ удовлетворяет марковскому свойству (модель без памяти) и не зависит от погоды во все другие моменты времени, кроме $t = i$.

В общем случае на каждом временном шаге в марковских моделях делают следующее **предположение независимости** W_{i+1} _||_ $\{W_0,\ldots, W_{i-1}\}|W_i$. Это позволяет восстановить совместное распределение для первых $n + 1$ переменных с помощью цепочного правила следующим образом:

$$Pr(W_0,W_1,\ldots,W_n) = Pr(W_0)Pr(W_1|\ W_0)Pr(W_2|W_1)\ldots\ Pr(W_n|W_{n-1}). \qquad (5.11)$$

Чтобы определить распределение погодных условий в произвольный день используют алгоритм прямого распространения. В соответствии со свойством маргинализации

$$Pr(W_{i+1}) = \sum_{wi} Pr(w_i,\ W_{i+1}). \qquad (5.12)$$

Применив цепочное правило, получим выражение, определяющее **алгоритм прямого распространения** (mini-forward алгоритм):

$$Pr(W_{i+1}) = \sum_{wi}\ Pr(W_{i+1}|w_i)Pr(w_i). \qquad (5.13)$$

Алгоритм позволяет итеративно вычислять распределение W_{i+1} для произвольно заданного момента времени, начиная с априорного распределения $Pr(W_0)$.

После большого числа шагов мы приходим к **стационарному распределению**, которое слабо зависит от начального распределения. При большом числе шагов на результат оказывает доминирующее влияние переходное распределение.

5.2.9. Скрытые марковские 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 5.2 изображена скрытая модель Маркова для моделирования погоды. В отличие от обычной марковской модели, СММ содержит два типа узлов: скрытые узлы W_i , которые являются **переменными состояния** и представляют погоду в i-ый день, и наблюдаемые узлы F_i , которые представляют **переменные, называемые свидетельствами (наблюдениями).** В рассматриваемом примере свидетельства F_i представляют прогноз погоды в i-ый день.

Одна из задач, решаемая с помощью модели СММ и называемая **фильтрацией или мониторингом**, заключается в вычислении апостериорных распределений скрытых переменных состояний в текущий момент времени по значениям всех полученных к этому моменту свидетельств.

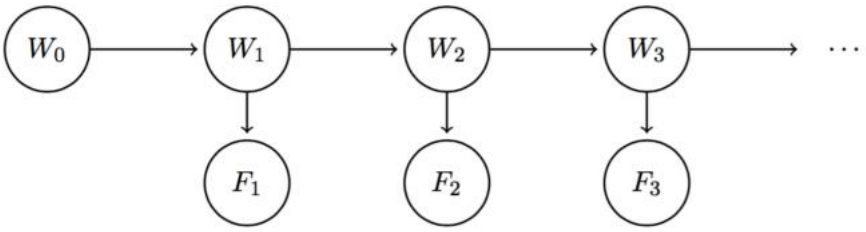


Рисунок 5.2. – Скрытая модель Маркова

СММ подразумевает такие же **условные отношения независимости**, как и для стандартной марковской модели, с дополнительным набором отношений для переменных свидетельств:

$$F_1 \_||\_ W_0|W_1 ,$$
$$\forall i = 2,\ldots,n \quad W_i \_||\_ \{ W_0,\ldots, W_{i-2}; F_1,\ldots., F_{i-1}\} \mid W_{i-1},$$
$$\forall i = 2,\ldots,n \quad F_i \_||\_ \{ W_0,\ldots, W_{i-1}; F_1,\ldots, F_{i-1} \} \mid W_i. \qquad (5.14)$$

Как и в марковских моделях, в СММ предполагается, что переходная модель $Pr(W_{i+1}|W_i)$ является стационарной. СММ делают дополнительное упрощающее предположение, что **модель восприятия (модель наблюдения, сенсорная модель)** $Pr(F_i|W_i)$ также является стационарной. *Следовательно, любая СММ может быть компактно представлена с помощью всего лишь трех таблиц вероятностей: начального распределения, модели перехода и модели наблюдения.*

Рассмотрим **алгоритм прямого распространения для СММ**. Определим *распределение степеней уверенности* (belief distribution) относительно возможных значений состояния W_i по всем свидетельствам $f_1,\ldots, f_i$, поступившим к моменту времени i как:

$$B(W_i) = Pr(W_i| f_1,\ldots, f_i). \qquad (5.15)$$

Аналогично определим через $B'(W_i)$ оценку распределения степеней уверенности (убеждений) в момент времени i по наблюдениям $f_1,\ldots, f_{i-1}$, которые поступили к моменту времени i-1, т.е. оценку $B'(W_i)$ можно рассматривать как *прогноз на один шаг вперед*:

$$B'(W_i) = Pr(W_i| f_1,\ldots, f_{i-1}) \qquad (5.16$$

Найдем соотношение между $B(W_i)$ и $B'(W_{i+1})$. Начнем с определения $B'(W_{i+1})$:

$$B'(W_{i+1}) = Pr(W_{i+1}| f_1,\ldots, f_i)= \sum_{wi} Pr(W_{i+1}, w_i \mid f_1,\ldots, f_i). \qquad (5.17)$$

Перепишем соотношение с использованием цепочного правила (5.4):

$$B'(W_{i+1}) = Pr(W_{i+1} \mid f_1, \ldots, f_i) = \sum_{wi} Pr(W_{i+1}, |w_i , f_1, \ldots, f_i)\, Pr(w_i \mid f_1, \ldots, f_i).$$

Так как $Pr(w_i|f_1, \ldots, f_i) = B(w_i)$ и W_{i+1} _||_ $\{f_1, \ldots, f_i\}|W_i$, то из последнего выражения следует **правило обновления во времени (**Time Elapse Update**),** которое распространяет распределение $B(W_i)$ с помощью модели перехода $Pr(W_{i+1}, |w_i)$ на один шаг вперед во времени и позволяет определить $B'(W_{i+1})$

$$B'(W_{i+1}) = \sum_{wi} Pr(W_{i+1}, |w_i)\, B(w_i). \qquad (5.18)$$

Найдем связь между $B'(W_{i+1})$ и $B(W_{i+1})$. Из правила Байеса (5.5) следует:

$$B(W_{i+1}) = Pr(W_{i+1}|f_1, \ldots, f_{i+1}) = \frac{Pr(W_{i+1}, f_{i+1}|f_1, \ldots, f_i)}{Pr(f_{i+1}|f_1, \ldots, f_i)}$$

Опуская операцию деления на знаменатель (операция нормализации), перепишем выражение с использованием цепочного правила:

$$B(W_{i+1}) \propto Pr(W_{i+1}, f_{i+1}|f_1, \ldots, f_i) = Pr(f_{i+1}|W_{i+1}, f_1, \ldots, f_i) Pr(W_{i+1}|f_1, \ldots, f_i).$$

В соответствии с предположениями условной независимости для СММ и определением $B'(W_{i+1})$ получим **правило обновления** $B'(W_{i+1})$ **на основе наблюдения** (Observation Update) $Pr(f_{i+1}|W_{i+1})$:

$$B(W_{i+1}) \propto Pr(f_{i+1}|W_{i+1}) B'(W_{i+1}). \qquad (5.19)$$

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 $B(W_i)$ на одном шаге во времени;
2. Обновление $B(W_i)$ на основе наблюдения, т.е. определение $B(W_{i+1})$ по $B'(W_{i+1})$.

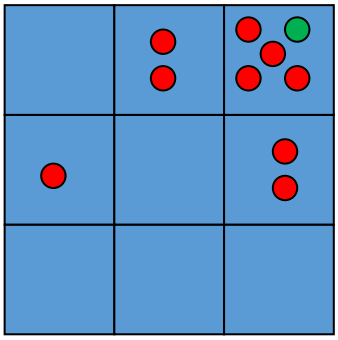
Отметим, что указанный выше трюк с нормализацией может значительно упростить вычисления. Если мы начнем с некоторого начального распределения и будем вычислять распределение степеней уверенности в момент времени *t,* то можно использовать прямой алгоритм для итеративного вычисления $B(W_1), \ldots, B(W_t)$ и выполнять нормализацию только один раз в конце, разделив каждую запись в таблице для $B(W_t)$ на сумму её записей.

5.2.10. Фильтрация частиц

Точный вывод с использованием алгоритма прямого распространения СММ характеризуется большой вычислительной сложностью. В этом случае, аналогично сетям Байеса, используют приближенные методы вывода, основанные на формировании случайных выборок из распределений [3].

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 вероятностей (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. Чем больше частиц, тем выше точность аппроксимации.

В качестве примера на рисунке 5.3. изображен **список частиц**, представленных координатами клеток, в которых они расположены. Соответственно, для частиц с координатами (3,3) эмпирическая оценка вероятности появления в списке частиц равна $B(3,3)=5/10=0.5$. Таким образом, по списку частиц можно восстановить эмпирическое распределение случайной переменной.



Список из 10 частиц:
(3,3), (2,3), (3,3), (3,2), (3,3), (3,2), (1,2), (3,3), (3,3),(2,3)

Рисунок 5.3. – Список частиц

Моделирование фильтрации частиц начинается с инициализации частиц. Например, можно выбрать частицы случайным образом из некоторого начального распределения. После того, как выбран исходный список частиц, моделирование принимает форму, аналогичную алгоритму прямого распространения в СММ с чередованием обновления распределений во времени и обновления на основе наблюдения на каждом временном шаге.

Обновление во времени (Time Elapse Update) — обновление выборочного значения каждой частицы в соответствии с моделью переходной вероятности. Для частицы в состоянии t_i выполняется случайная выборка обновленного значения из переходного распределения $Pr(T_{i+1} \mid t_i)$.

Обновление на основе наблюдения (Observation Update). Этот этап немного сложнее. Здесь используется модель сенсора $Pr(F_i|T_i)$ для взвешивания каждой частицы в соответствии с вероятностью, определяемой наблюдаемым свидетельством и состоянием частицы. В частности, частице в состоянии t_i при свидетельстве f_i , поступающим от некоторого сенсора, присваивается вес $Pr(f_i|t_i)$. Алгоритм обновления на основе наблюдений следующий:

1. Рассчитайте веса всех частиц в соответствии с $Pr(f_i|t_i)$;
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.

Давайте рассмотрим процесс фильтрации частиц (обновление во времени и обновление на основе наблюдения) на примере моделирования погоды. Пусть, например, имеется список из 10 частиц со следующими значениями температуры [15,12,12,10,18,14,12,11,11,10] из дипазона [10,20]. Соответственно, подсчитав количество различных состояний частиц и поделив эти значения на общее число частиц, получим эмпирическое распределение температуры в момент времени *i:*

T_i	10	11	12	13	14	15	16	17	18	19	20
$B(T_i)$	0.2	0.2	0.3	0	0.1	0.1	0	0	0.1	0	0

Определим модель перехода, используя температуру как случайную переменную, зависящую от времени. Будем полагать, что для определенного состояния температура может либо оставаться прежней, либо измениться на один градус в диапазоне [10, 20]. При этом пусть вероятность перехода в следующий момент времени к значению, которое ближе к 15, составляет 0.8, а остальные результирующие состояния равномерно делят оставшиеся 0.2 вероятности между собой.

Чтобы выполнить обновление во времени для первой частицы из списка частиц (T_i = 15), воспользуемся выбранной моделью перехода:

T_{i+1}	14	15	16
$Pr(T_{i+1} \mid T_i = 15)$	0.1	0.8	0.1

Для формирования выборки для частицы в состоянии T_i = 15 воспользуемся алгоритмом сэмплирования, описанным в разделе 5.2.7, в соответствии с которым просто генерируем случайное число в диапазоне [0, 1) и смотрим, в какой диапазон оно попадает. Например, если случайное число равно r = 0.467, то частица с T_i = 15 попадает в диапазон $0{,}1 \le r < 0{,}9$. Следовательно, в следующий момент времени с учетом таблицы переходных вероятностей частица будет в состоянии T_{i+1} = 15.

Допустим в ходе сэмплирования мы получили список из 10 случайных чисел в интервале [0, 1):

[0.467, 0.452, 0.583, 0.604, 0.748, 0.932, 0.609, 0.372, 0.402, 0.026]

Используя эти 10 случайных чисел для формирования выборочных значений наших 10 частиц, получим после полного обновления во времени новый список частиц:

[15,13,13,11,17,15,13,12,12,10].

Обновленный список частиц приводит к соответствующему обновленному распределению степеней уверенности $B(T_{i+1})$:

T_{i+1}	10	11	12	13	14	15	16	17	18	19	20
$B(T_{i+1})$	0.1	0.1	0.2	0.3	0	0.2	0	0.1	0	0	0

Теперь выполним обновление на основе наблюдения, предполагая, что сенсорная модель $Pr(F_i|T_i)$ утверждает, что вероятность правильного прогноза $f_i=t_i$ равна 0.8, а остальные 10 возможных значений состояний предсказываются с вероятностью 0.02. Если наблюдаемый прогноз F_{i+1} = 13, то веса частиц будут следующими:

Частица	1	2	3	4	5	6	7	8	9	10
Состояние	15	13	13	11	17	15	13	12	12	10
Вес	0.02	0.8	0.8	0.02	0.02	0.02	0.8	0.02	0.02	0.02

После суммирования весов каждого из состояний, получим

Состояние	10	11	12	13	15	17
Вес	0.02	0.02	0.04	2.4	0.04	0.02

Суммирование значений всех весов дает сумму 2.54, и мы можем нормализовать таблицу весов, чтобы получить распределение вероятностей, разделив каждую запись на эту сумму:

Состояние	10	11	12	13	15	17
Вес	0.02	0.02	0.04	2.4	0.04	0.02
Нормализованный вес	0.079	0.079	0.0157	0.9449	0.0157	0.079

Последним шагом является повторная выборка (ресэмплирование) из этого распределения вероятностей с использованием того же метода, который мы использовали для выборки во время обновления во времени. Допустим, мы генерируем 10 случайных чисел в диапазоне [0;1) со следующими значениями:

[0.315, 0.829, 0.304, 0.368, 0.459, 0.891, 0.282, 0.980, 0.898, 0.341]

Это дает новый (ресэмплированный) список частиц

[13,13,13,13,13,13,13,15,13,13]

с новым распределением степеней уверенности:

T_{i+1}	10	11	12	13	14	15	16	17	18	19	20
$B(T_{i+1})$	0	0	0	0.9	0	0.1	0	0	0	0	0

Обратите внимание, что сенсорная модель предполагает, что наш прогноз погоды весьма точен и характеризуется высокой вероятностью, равной 0,8. Соответственно, наш новый список частиц согласуется с этим: большинство частиц в результате ресэмплирования получают состояние T_{i+1} = 13.

5.3. Задания для выполнения

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 1. Структура сети Байеса

Реализуйте функцию **constructBayesNet** в файле **inference.py**. Функция должна создавать пустую байесовскую сеть со структурой, изображенной на рисунке 5.4. Сеть представляет упрощенный мир игры «Охота на привидения». Узлами сети являются следующие переменные: **Pacman, Ghost0** и **Ghost1** – два приведения, **Obs0** и **Obs1** – оценки расстояний до приведений, набдюдаемые сенсорами Пакмана. Полезно перед написанием кода функции **constructBayesNet** взглянуть на примеры в коде функции **printStarterBayesNet** в файле **bayesNet.py**.

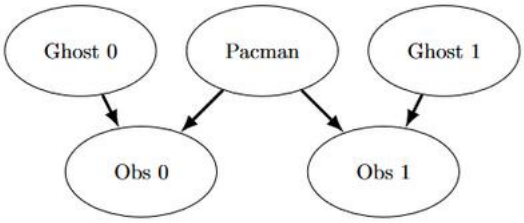


Рисунок 5.4. – Сеть Байеса игры “Охота на привидения”

Чтобы протестировать и отладить свой код, выполните команду:

python autograder.py -q q1

Задание 2. Объединение факторов

Реализуйте функцию **joinFactors** в **factorOperations.py**. Она принимает список входных факторов **factors** и возвращает новый фактор, таблица вероятностей которого вычисляется как произведение соответствующих вероятностей входных факторов. Функция **joinFactors** реализует правила произведения факторов. Например, если у нас есть фактор вида P(X|Y) и другой фактор вида P(Y), то объединение этих факторов даст P(X,Y).

Вот несколько дополнительных примеров того, что может делать функция **joinFactors**:

- **joinFactors** (P(V,W|X,Y,Z),P(X,Y|Z))=P(V,W,X,Y|Z);
- **joinFactors** (P(X|Y,Z),P(Y))=P(X,Y|Z);

- **joinFactors** (P(V|W),P(X|Y),P(Z))=P(V,X,Z|W,Y).

Не следует полагать, что **joinFactors** вызывается для таблиц вероятностей — можно вызвать **joinFactors** и для факторов, строки таблиц которых не дают в сумме 1.

Для тестирования и отладки кода выполните команду:

python autograder.py -q q2

Может быть полезно запустить определенные тесты во время отладки, чтобы увидеть только один набор факторов. Например, чтобы запустить только первый тест, выполните:

python autograder.py -t test_cases/q2/1-product-rule

Задание 3. Исключение переменных (маргинализация)

Реализовать функцию исключения переменных **eliminate** в **factorOperations.py.** Она принимает фактор **factor** и переменную для исключения **eliminationVariable** и возвращает новый фактор, который не содержит эту переменную. Это соответствует суммированию всех строк в таблице вероятностей для **factor**, которые отличаются только значением исключаемой переменной.

Вот несколько примеров того, что может делать функция **eliminate**:

- **eliminate**(P(X,Y|Z),Y)=P(X|Z)
- **eliminate**(P(X,Y|Z),X)=P(Y|Z)

Для тестирования и отладки кода запустите

python autograder.py -q q3

Может быть полезно запустить определенные тесты во время отладки. Например, чтобы запустить только первый тест, выполните:

python autograder.py -t test_cases/q3/1-simple-eliminate

Задание 4. Вывод на основе исключения переменных

Реализуйте функцию вывода для сети Байеса на основе исключения переменных **inferenceByVariableElimination** в **inference.py**. Она формирует ответ на вероятностный запрос, который представляется с помощью списка переменных запроса и свидетельств.

Для тестирования и отладки кода функции выполните команду:

python autograder.py -q q4

Может быть полезно запустить определенные тесты во время отладки. Например, чтобы запустить только первый тест, выполните команду:

python autograder.py -t test_cases/q4/1-disconnected-eliminate

Алгоритм должен перебирать скрытые переменные в порядке исключения, выполняя объединение и исключение переменных, пока не останутся только переменные запроса и свидетельств.

Сумма вероятностей в вашем выходном факторе должна быть равна 1 (чтобы это была истинная условная вероятность, обусловленная свидетельством).

Посмотрите на функцию **inferenceByEnumeration** в **inference.py** для примера того, как использовать дополнительные функции для решения задания. Напомним, что вывод путем перечисления сначала объединяет все переменные, а затем исключает все скрытые переменные. Напротив, вывод на основе исключения переменных чередует объединение и исключение путем итерации по всем скрытым переменным и выполняет объединение и исключение для одной скрытой переменной, прежде чем перейти к следующей скрытой переменной.

Вам нужно будет позаботиться об особом случае, когда фактор, который вы объединили, имеет только одну безусловную переменную (в строках документирования функции указано, что делать в этом случае).

Задание 5а. Класс DiscreteDistribution

К сожалению, наличие в игре Пакман временных шагов приводит к разрастанию графа сети Байеса, чтобы исключение переменных стало жизнеспособным. Вместо этого далее будем использовать прямой алгоритм для точного вывода в CMM и фильтрацию частиц для еще более быстрого, но приближенного вывода.

Для остальной части проекта мы будем использовать класс **DiscreteDistribution**, определенный в **inference.py**, для моделирования распределений. Класс используется для работы с дискретными распределениями. Этот класс является разновидностью словаря Python, где ключами являются значения переменных 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. 5.2.7.

Задание 5б. Вероятность наблюдения

В этой части задания 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:

$$P(\text{noisyDistance} \mid \text{pacmanPosition}, \text{ghostPosition}).$$

По сути метод реализует модель наблюдения (восприятия) СММ.

Чтобы протестировать свой код, запустите автооцениватель для этого задания:

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

Внесите код и результаты тестирования метода в отчет.

Задание 6. 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 (5.19) при получении нового наблюдения **observation** – зашумленного манхеттенского растояния до призрака. 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.

Задание 7. 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 (5.18). При этом агент имеет доступ к распределению действий призрака через **self.getPositionDistribution**.

Задание 8. 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.

Задание 9. Инициализация приближенного вывода

В нижеследующих заданиях (10, 11) 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.

Задание 10. Приближенный вывод: обновление на основе наблюдения

Необходимо реализовать метод **observUpdate** класса **ParticleFilter** в файле **inference.py**. Метод осуществляет обновление на основе наблюдения в соотвествии с алгоритом, описанным в п. 5.2.10. 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**.

Задание 11. 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. 5.2.10.

5.4. Порядок выполнения лабораторной работы

5.4.1. 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.

5.4.2. Используйте для выполнения лабораторной работы файлы из архива **МиСИИ_лаб5_2024.zip.** Разверните программный код лабораторной работы в новой папке и не смешивайте с файлами предыдущих лабораторных работ. Архив содержит следующие файлы:

Файлы для редактирования:	
bustersAgents.py	Агенты для охоты за призраками
inference.py	Код для отслеживания призраков во времени по их шумам.
factorOperations.py	Операции по вычислению новых совместных или маргинальных таблиц вероятностей.
Файлы, которые необходимо просмотреть	
bayesNet.py	Классы BayesNet и Factor
busters.py	Основной файл, из которого запускают игру Ghostbusters с охотниками за призраками (заменяет Pacman.py)
busterGhostAgents.py	Новые агенты-призраки для игры с охотниками за призраками
distanceCalculation.py	Вычисляет расстояния лабиринта
game.py	Вспомогательные классы для Pacman
ghostAgents.py	Агенты, управляющие призраками
graphicsDisplay.py	Графика Pacman
graphicsUtils.py	Графические утилиты
keyboardAgents.py	Интерфейс клавиатуры для управления игрой
layout.py	Код для чтения файлов схем и хранения их содержимого
util.py	Утилиты

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

5.4.3. В рассматриваемом варианте игры цель состоит в том, чтобы выследить невидимых призраков. Пакман оснащен сонаром (слухом), который обеспечивает оценку манхэттенского расстояния до каждого призрака. 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. Оценки расстояний, отображаемые в нижней части графического окна, всегда неотрицательны и всегда находятся в пределах 7 единиц от их истинного значения.

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, сети Байеса 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 сетей Байеса.

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 тесты по отдельности. 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-small-board

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**, который отключает графику

5.4.4. Просмотрите **bayesNet.py**, чтобы познакомиться с классами, с которыми вы будете работать – **BayesNet** и **Factor**. Вы также можете запустить этот файл, чтобы увидеть пример сети **BayesNet** и связанные с ней факторы:

python bayesNet.py

Вам следует взглянуть на функцию **printStarterBayesNet** – там есть полезные комментарии, которые могут значительно облегчить вам жизнь в дальнейшем. Сеть Байеса, созданная этой функции, является простейшей V-сетью с общим следствием: **(Raining –> Traffic <– Ballgame).**

5.4.5. В задании 1 требуется реализовать функцию **constructBayesNet** в файле **inference.py**. Функция должна создавать сеть, изображенную на рисунке 5.4. Для этого в коде **constructBayesNet** необходимо определить все переменные-узлы сети (**variables**) и ребра (**edges**). Определите каждое ребро сети Байеса, представив его в виде кортежа '**(от, до)**', например: **(GHOST1, OBS1).** Определите возможные значения переменных в словаре **variableDomainsDict[***var***],** для каждой переменной *var*, например: **variableDomainsDict[PAC] = possibleAgentPos,** где **possibleAgentPos** список из кортежей **(x, y)** – возможных позиций Пакмана (**PAC)** в пределах поля игры. Pacman и два призрака могут находиться где угодно (игнорируйте стены на схеме игры). Определите все возможные кортежи позиций призраков с учетом того, что сенсоры Пакмана не точные. Учтите, что оценки наблюдаемых расстояний **OBS** неотрицательны и равны манхэттенским расстояниям от Пакмана до призраков ± шум (**MAX_NOISE**).

5.4.6. В задании 2 требуется реализовать функцию **joinFactors** в **factorOperations.py.** Сформируйте список всех факторов **allFactors** и множества условных **conditional** и безусловных переменных **unconditional:**

```
allFactors=[factor for factor in factors]
conditional=set([])
unconditional=set([])
for factor in allFactors:
    for f in factor.conditionedVariables():
        conditional.add(f)
    for f in factor.unconditionedVariables():
        unconditional.add(f)
```

Сформируйте новый список условных переменных для объединенного результирующего фактора – это те переменные исходного списка **conditional**, которых нет в списке **unconditional**. Создайте новый объединенный фактор, передав на его вход новый список условных переменных

```
newCPT = Factor(unconditional, conditional, variableDomainsDict)
```

Объекты **Factors** хранят **variableDomainsDict**, который сопоставляет каждую переменную со списком значений, которые она может принимать (ее домен). Объект **Factor** получает свой **variableDomainsDict** из сети **BayesNet**. Он содержит все переменные сети **BayesNet**, а не только безусловные и условные переменные, используемые в объектом **Factor**. Для этой задачи вы можете предположить, что все входные факторы взяты из одной и той же сети **BayesNet**, и поэтому все их **variableDomainsDict** одинаковы, т.е.

```
variableDomainsDict=allFactors[0].variableDomainsDict()
```

После создания нового фактора, заполните его таблицу CPT, выполнив вычисления в соответствии с алгоритмом:

Для всех возможных присваиваний assignment нового фактора из newCPT.getAllPossibleAssignmentsDicts():
probability=1
Для каждого входного фактора из allFactors:
вызвать factor.getProbability(assignment) и вычислить вероятности
probability = probability * factor.getProbability(assignment)
newCPT.setProbability(assignment, probability)

5.4.7. В задании 3 требуется реализовать функцию исключения переменных **eliminate(factor: Factor, eliminationVariable: str)** в файле **factorOperations.py.** Функция возвращает новый фактор, из которого удалена переменная **eliminationVariable.**

Помните, что факторы хранят словарь **variableDomainsDict** исходной сети **BayesNet**, а не только безусловные и условные переменные, которые они используют. В результате возвращаемый фактор должен иметь тот же **variableDomainsDict**, что и входной фактор.

Псевдокод решения задания:

1. Формируем новый список безусловных переменных unconditioned без eliminationVariable;
2. Копируем в список условных переменных из фактора factor:
conditioned = factor.conditionedVariables();
3. Извлекаем из variableDomainsDict область значений eliminationVariable:
domain = variableDomainsDict[eliminationVariable];
4. Создаем новый фактор без eliminationVariable:
newFactor = Factor(unconditioned, conditioned, variableDomainsDict);
5. Для каждого возможного присваивания assignment нового фактора, извлекаемого из newFactor.getAllPossibleAssignmentsDicts():
prob = 0;
Для каждого значения elim_val исключаемой переменной из domain:
Копируем assignment в old_assignment;
Расширяем assignment на очередное значение elim_val:
old_assignment[eliminationVariable] = elim_val;
Обращаемся к строке таблицы CPT входного фактора и вычисляем:
prob += factor.getProbability(old_assignment);
Записываем в таблицу CPT нового фактора значение вероятности:
newFactor.setProbability(assignment, prob);
return newFactor

5.4.8. В задании 4 требуется реализовать функцию вывода на основе исключения переменных

inferenceByVariableElimination(bayesNet, queryVariables, evidenceDict, eliminationOrder):

Здесь:

bayesNet – сеть Байеса, на основе которой выполняется вывод;

queryVariables – список безусловных переменных запроса;

evidenceDict – словарь присваиваний **{переменная : значение}** для переменных, которые представлены как свидельства (условные) в запросе вывода;

eliminationOrder – порядок исключения переменных.

Псевдокод решения задания:

```
1. Возвращаем список факторов (с учетом свидетельств) для всех переменных в байесовской сети:
        factors = bayesNet.getAllCPTsWithEvidence(evidenceDict);
2. Для всех переменных var в порядке исключения eliminationOrder:
       #Выполнить объединение факторов по исключаемой переменной var
       factors, new_factor = joinFactorsByVariable(factors, var);
       Если new_factor содержит более одной безусловной переменной:
              Удалить из него исключаемую переменную var;
              Добавить фактор new_factor в конец списка factors;
3. Объединить факторы из factors:  final_factor = joinFactors(factors)
4. Вернуть нормализованный фактор normalize(final_factor)
```

5.4.9. В задании 5а 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 5.2.5.

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 doc-тесты. 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.

5.4.10. В задании 5б 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.

5.4.11. В задании 6 необходимо реализовать метод **observeUpdate** класса **ExactInference**. Метод обеспечивает вычисления в соответствии с правилом обновления (5.19). 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 5б), 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 (5.19) при использовании вероятности наблюдения, вычисляемой в ходе вызова:

```
self.getObservationProb(observation, pacmanPosition,
                                        possibleGhostPos, jailPosition)
```

При этом учтите, что значения $B'(W_{i+1})$ из (5.19) обновляются до значений $B(W_{i+1})$ 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 q6
```

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 q6 --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.

5.4.12. В задании 7 необходимо реализовать метод **elapseTime** класса **ExactInference**. Метод выполняет вычисления в соответствии с правилом обновления во времени (5.18), 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 (5.18):

beliefDict[newPos]=beliefDict[newPos]+self.beliefs[ghostPos]*prob,

где **beliefDict** – словарь типа **DiscreteDistribution()**. При этом значения $B(w_i)$ соответствуют **self.beliefs[ghostPos],** а переходные вероятности $Pr(W_{i+1},|w_i)$ 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 лучшего понимания задания на рисунке 5.5. изображена сеть Байеса в виде скрытой марковской модели (СММ) для 2-х приведений и 2-х моментов времени. В этом случае **getPositionDistribution** формально соответствует выражению $P(G_{t+1}$ |**gameState**, $G_t)$).

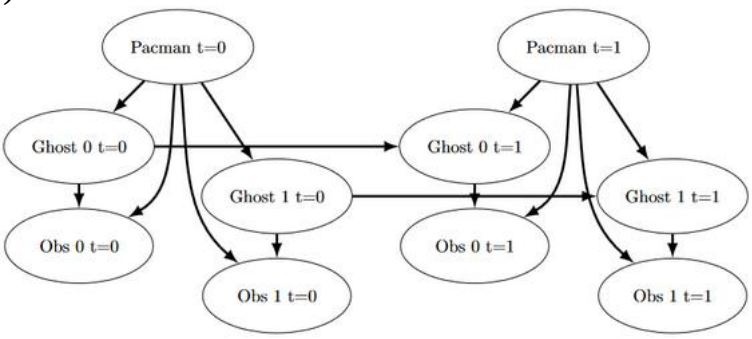


Рисунок 5.5. – Сеть Байеса в виде СММ для 2-х моментов времени

Для автооценивания этого задания и визуализации результатов используйте команду:

python autograder.py -q q7

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.

5.4.13. В начальной части кода задания 8 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
```

Мы можем представить (рисунок 5.6) как работает наш жадный агент, внеся дполнения в рисунок 5.5.

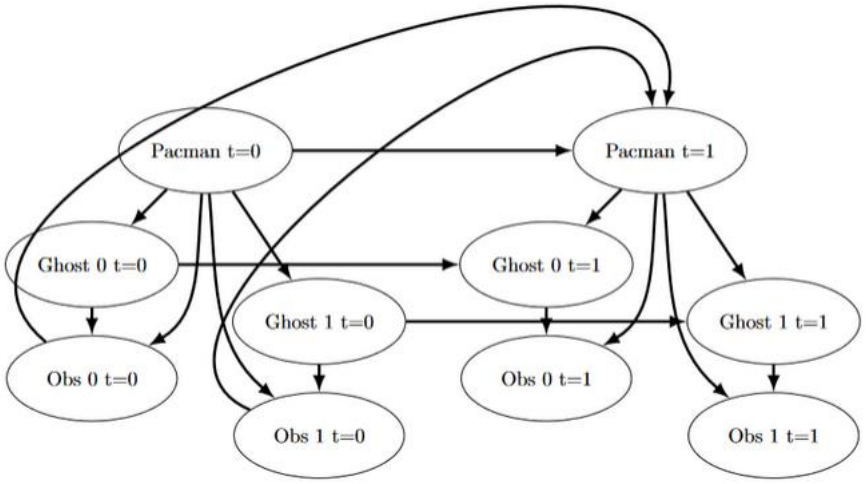


Рисунок 5.6. – Взаимодействия в СММ для 2-х моментов времени

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

python autograder.py -q q8

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

Внесите код метода **ChooseAction** и результаты тестирования в отчет.

5.4.14. При выполнении задания 9 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: **legal**=**self.legalPositions**.
Псевдоалгоритм решения задачи:

1. **Определить число частиц с помощью particle_num = self.numParticles;**
2. **Определяем допустимые позиции частиц: legal = self.legalPositions;**
3. **Повторять число раз, равное целому числу частиц в одной позиции: (int(particle_num / len(legal))):**
   Расширить список частиц на список legal: self.particles.extend(legal).

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 pos in self.particles:
    beliefDist[pos]=beliefDist[pos]+1
```

Нормализуйте полученное распределение **beliefDist** и верните его в качестве результата.

Чтобы протестировать задание выполните команду:

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

Внесите код разработанных методов и результаты тестирования в отчет.

5.4.15. В задании 10 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.5.2.10) 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 q10
```

или без графики

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

Внесите код разработанных методов и результаты тестирования в отчет.

5.4.16. В задании 11 необходимо реализовать в виде метода **elapseTime** класса **ParticleFilter** алгоритм обновления во времени, описанный в п. 5.2.10. 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 q11
```

Для запуска теста без графики добавьте параметр **--no-graphics.** Внесите код разработанных методов и результаты тестирования в отчет.

5.5. Содержание отчета

Цель работы, описание основных понятий сетей Байеса, марковских моделей, скрытых марковских моделей, описание алгоритмов прямого распространения для СММ (правил обновления во времени и на основе наблюдения), описание понятия фильтрации частиц и соотвествующих алгоритмов вывода, код реализованных

функций с комментариями в соответствии с заданиями 1-11, результаты автооценивания заданий, выводы по проведенным экспериментам с разными алгоритмами вероятностных выводов.

5.6. Контрольные вопросы

5.6.1 Объясните, что понимают под степенью уверенности высказываний?

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

5.6.3. Что понимают под событием?

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

5.6.5. Запишите формулу условной вероятности *x* при известном *y*.

5.6.6. Запишите правило произведения для 2-х переменных.

5.6.7. Запишите цепочное правило.

5.6.8. Запишите правило Байеса и объясните его.

5.6.9. Определите понятие независимости 2-х случайных переменных.

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

5.6.11. Определите понятие сети Байеса.

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

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

5.6.14. Что понимают под точным и приближенным вероятностным выводом?

5.6.15. Сформулируйте алгоритм формирования случайной выборки из заданного распределения.

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

5.6.17. Определите алгоритм прямого распространения для марковской модели.

5.6.18. Определите понятие скрытой марковской модели.

5.6.19. Сформулируйте задачу фильтрации для СММ.

5.6.20. Какие предположения независимости используют в СММ?

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

5.6.22. Сформулируйте алгоритм прямого распростанения для СММ.

5.6.23. Что такое фильтрация частиц применительно к СММ?

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