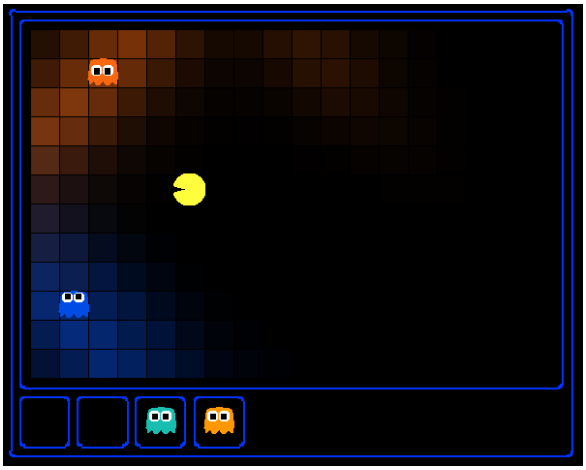
课程作业 3：捉鬼敢死队（Ghostbusters）



任务概述

在本次课程作业中，你将设计吃豆人智能体来使用传感器（声纳）定位和抓捕隐形的幽灵，包括定位单个静止的幽灵和多个移动的幽灵。传感器可以提供到每个幽灵的曼哈顿距离的噪声读数。 当吃豆人抓捕到所有幽灵时游戏结束。首先，可以尝试使用键盘自己玩游戏：

```
python busters.py
```

给定吃豆人接收到的噪声距离读数，颜色块指示每个幽灵可能位于的位置。显示屏底部的噪声距离始终为非负值，并且始终与真实距离差 7 格以内。距离读数的概率随着其与真实距离的差值呈指数下降。

本次课程作业的主要任务是实现概率推理来追踪幽灵。对于上面基于键盘的游戏，默认情况下实现了一种粗略的推理形式：所有可能存在幽灵的方块都被幽灵的颜色所标记。为了能够更好地估计幽灵的位置，我们可以使用贝叶斯网络来充分利用我们所拥有的信息。**本次课程作业将需要你去实现使用贝叶斯网络执行精确和近似推理的算法。**

对于此次任务，如果使用图形界面运行测试可能会超时。要准确确定代码是否足够高效，可以添加`--no-graphics`来运行测试。

作业的程序包位于 QQ 群文件：课程作业\tracking.zip。

所有你自己的算法实现都位于如下三个文件：`bustersAgents.py`，`inference.py`和`factorOperations.py`中相应任务的类和函数下面。

贝叶斯网络和因子

首先查看`bayesNet.py`中的`BayesNet`和`Factor`这两个后面将会用到的类。可以运行：`python bayesNet.py`
来查看这两个类的示例。也可以查看`printStarterBayesNet`函数来了解如何创建和使

用这两个类。这个函数所创建的贝叶斯网络如下：

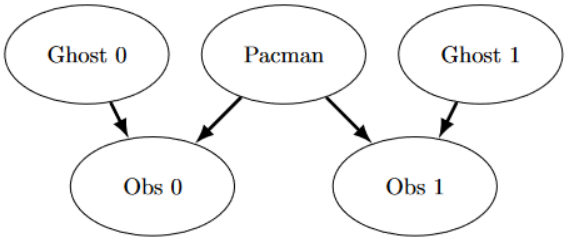
(Raining -> Traffic <- Ballgame)

- **贝叶斯网络（Bayes Net）**：一种概率模型，表示为一个有向无环图和一组条件概率表，每个变量对应一个条件概率表。
- **因子（Factor）**：存储了一个概率表，尽管表中条目的总和不一定为 1。因子的一般形式可以写为 $f(X_1,\ldots,X_m,y_1,\ldots,y_n \mid Z_1,\ldots,Z_p,w_1,\ldots,w_q)$，这里小写字母表示该变量已经被赋值。这里 Z_j 和 w_k 称之为条件变量，X_i 和 y_l 称之为无条件变量。
- **条件概率表（CPT）**：一个满足两个属性的因子：1）对于条件变量的每次赋值，其条目之和必须为 1；2）恰好存在一个无条件变量。

作业内容

任务 1：贝叶斯网络结构（Bayes Net Structure）

实现 `inference.py` 中的 `constructBayesNet` 函数。该函数构建了一个贝叶斯网络，其结构如下图所示：



正如 `constructBayesNet` 的代码中所述，我们通过列出所有变量、它们的取值以及它们之间的边来构建网络结构。上图展示了变量和边，还需要考虑它们的值域。

- 根据上图添加变量和边；
- 吃豆人和两个幽灵可以出现在网格中的任何地方（对此我们忽略墙壁）。为它们添加所有可能的位置元组；
- 这里的观测结果是非负的，等于吃豆人到幽灵的曼哈顿距离 ± 噪声。

为了测试和调试你的代码，运行：

```
python autograder.py -q q1
```

任务 2：连接因子（Join Factors）

实现 `factorOperations.py` 中的 `joinFactors` 函数。该函接受一个 `Factor` 列表并返回一个新 `Factor`，其概率条目是输入 `Factor` 的相应行的乘积。

`joinFactors` 可以用作乘积法则，例如，如果我们有一个因子 $P(X \mid Y)$ 和另一个因子 $P(Y)$，然后连接这两个因子将会得到 $P(X,Y)$。因此，`joinFactors` 允许我们合并条件变量的概率（例如这里的 Y）。`joinFactors` 不一定是在概率表上进行调用，也可以在表中条目的总和不为 1 的因子上进行调用。

为了测试和调试你的代码，运行：

```
python autograder.py -q q2
```

为了方便调试，可以运行特定测试来仅查看一组因子的打印结果。 例如，仅运行第一个测试：

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

小贴士：

- `joinFactors` 函数应该返回一个新的 `Factor`；
- 下面是一些 `joinFactors` 函数运行示例：
  - `joinFactors` $(P(X \mid Y), P(Y)) = P(X, Y)$
  - `joinFactors` $(P(V, W \mid X, Y, Z), P(X, Y \mid Z)) = P(V, W, X, Y \mid Z)$
  - `joinFactors` $(P(X \mid Y, Z), P(Y)) = P(X, Y \mid Z)$
  - `joinFactors` $(P(V \mid W), P(X \mid Y), P(Z)) = P(V, X, Z \mid W, Y)$
- `Factor` 存储了一个字典 `variableDomainsDict`，将每一个变量与它的可能取值的列表（值域）进行关联。`Factor` 包含了 `BayesNet` 的所有变量，而不仅仅是 `Factor` 中使用的无条件变量和条件变量。可以认为所有输入 `Factor` 都来自同一个 `BayesNet`，因此它们的 `variableDomainsDicts` 都是相同的。

任务 3：消元（Eliminate）

实现 `factorOperations.py` 中的 `eliminate` 函数。该函接受一个 `Factor` 和一个要消元的变量并返回一个不包含该变量的新 `Factor`。这相当于对 `Factor` 中的所有仅在被消元的变量值上不同的条目进行求和。

为了测试和调试你的代码，运行：

```
python autograder.py -q q3
```

为了方便调试，可以运行特定测试来仅查看一组因子的打印结果。 例如，仅运行第一个测试：

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

小贴士：

- `eliminate` 函数应该返回一个新的 `Factor`；
- `eliminate` 可用于边缘化概率表中的变量，例如：
  - `eliminate` $(P(X, Y \mid Z), Y) = P(X \mid Z)$
  - `eliminate` $(P(X, Y \mid Z), X) = P(Y \mid Z)$
- `Factor` 存储原始 `BayesNet` 的 `variableDomainsDict`，而不仅仅是它们用到的无条件变量和条件变量。返回的 `Factor` 应该具有与输入 `Factor` 相同的 `variableDomainsDict`。

任务 4：变量消元（Variable Elimination）

实现 `inference.py` 中的 `inferenceByVariableElimination` 函数，回答一个概率查询，该查询使用一个 `BayesNet`、查询变量列表和证据来表示。

为了测试和调试你的代码，运行：

```
python autograder.py -q q4
```

为了方便调试，可以运行特定测试来仅查看一组因子的打印结果。 例如，仅运行第一个测试：

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

小贴士：

- 算法应该按消元顺序在隐藏变量上进行迭代，对每一个隐藏变量执行连接并消除该变量，直到只剩下查询和证据变量。
- 输出因子中的概率总和应为 1（使其为一个真实的以证据为条件的条件概率分布）。
- 查看 `inference.py` 中的 `inferenceByEnumeration` 函数来了解如何使用所需的函数。（注意：通过枚举进行推断首先连接所有变量，然后消除所有隐藏变量。相反，变量消元通过在所有隐藏变量上进行迭代来交错的执行连接和消元，即在进行到下一个隐藏变量之前，对单个隐藏变量执行连接和消元。）
- 如果连接后的因子只有一个无条件变量，应该直接丢掉该因子而不应该再去对这个变量执行消元（显然对其消元的结果是 1），从而可以提高算法效率。

任务 5：

1）`DiscreteDistribution` 类的实现

时间步的加入会使我们的图模型复杂度大幅增长，使得变量消元不再可行。 因此，我们将使用隐马尔可夫模型（HMM）的前向算法进行精确推理，并使用粒子滤波进行更高效但是近似的推理。

我们将使用 `inference.py` 中定义的 `DiscreteDistribution` 类来对置信度分布和权重分布进行建模。该类是内置 Python 字典类的扩展，其中键（key）是分布中的不同离散元素，相应的值（value）与分布分配给该元素的置信度或权重成正比。这个任务要求实现该类中缺失的部分，这对于后面的任务至关重要。

首先，**实现 `normalize` 方法**，该方法将分布中的值归一化为总和为 1，但保持值的比例不变（使用 `total` 方法来获得分布中值的总和）。对于空分布或所有值均为零的分布，不执行任何操作。需要注意的是，`normalize` 方法直接修改分布，而不是返回新的分布。

其次，**实现 `sample` 方法**，该方法从分布中抽取样本，其中某个键（key）被采样的概率与其对应的值（value）成正比。 这里假设分布不为空，且不是所有值都为零。 请注意，在调用 `sample` 方法之前，分布不一定必须归一化。在实现该方法时可以考虑使用 Python 的内置 `random.random()`函数。

2）观测概率（Observation Probability）

实现 `inference.py` 中 `InferenceModule` 基类的 `getObservationProb` 方法。该方法接受一个观测（到幽灵距离的噪声读数）、吃豆人的位置、幽灵的位置和幽灵监狱的位置，并返回给定吃豆人的位置和幽灵的位置，噪声距离读数的概率，即 $P(noisyDistance \mid pacmanPosition, ghostPosition)$。

距离传感器在给定吃豆人到幽灵的真实距离的情况下，具有距离读数的概率分布。在该任务中，可以使用所提供的 `busters.getObservationProbability(noisyDistance, trueDistance)`函数来获得该分布，其返回 $P(noisyDistance \mid trueDistance)$，并可以使用所提供的 `manhattanDistance` 函数来获得吃豆人的位置和幽灵的位置之间的距离。

此外，我们还需要处理监狱这种特殊情况。当我们捕获一个幽灵并将其送到监狱所在位置时，距离传感器确定性地返回 `None`，而不返回任何其他内容（当且仅当幽灵在监狱中时，观测=`None`）。 因此，如果幽灵的位置是监狱位置，那么观测结果为 `None`，且概率为 1，而其他所有情况的概率为 0。

为了测试和调试你的代码，运行：

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

任务 6：精确推理：观测（Exact Inference Observation）

实现 `inference.py` 中 `ExactInference` 类的 `observeUpdate` 方法，从而根据传感器的观测，更新吃豆人智能体对幽灵位置的置信度分布，即对于观测新证据，实现在线置信度更新。在这个任务中，`observeUpdate` 方法应该在收到传感器读数后更新地图上每个位置的置信度。你应该在变量 `self.allPositions` 上迭代更新，其中包括所有合法位置以及监狱位置。置信度表示幽灵位于特定位置的概率，并作为 `DiscreteDistribution` 对象存储在名为 `self.beliefs` 的字段中，你需要去更新该字段。

在实现方法之前，首先写出所要解决的推理问题的方程。使用在上一个任务中编写的函数 `self.getObservationProb`，该函数返回给定吃豆人位置、潜在的幽灵位置和监狱位置的观测概率。可以使用 `gameState.getPacmanPosition()`获取吃豆人的位置，并使用 `self.getJailPosition()`获取监狱位置。

高后验置信度由明亮的颜色表示，而低置信度由暗淡的颜色表示。随着时间的推移，更多证据的积累会使置信度分布从较大范围逐渐收敛到较小范围。

为了测试你的代码并可视化输出，运行：

```
python autograder.py -q q6
```

如果想在没有图形界面的情况下进行测试，可以运行：

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

任务 7：精确推理：时间推移（Exact Inference with Time Elapse）

上一个任务根据吃豆人的观测结果实现了置信度更新。吃豆人的观测并不是他了解幽灵可能在哪里的唯一信息源。吃豆人还了解幽灵的移动方式，即幽灵不能在一个时间步内穿过一堵墙或移动多个位置。

考虑有吃豆人和一个幽灵的场景，吃豆人收到的许多观测结果表明幽灵在非常近的位置，但随后也收到了一个表明幽灵离得非常远的观察结果。这个表明幽灵离很远的读数很可能是传感器故障的结果。吃豆人关于幽灵如何移动的先验知识将减少这一读数的影响，因为吃豆人知道幽灵不能仅通过一次移动就移动到那么远的位置。

该任务需要实现 `inference.py` 中 `ExactInference` 类的 `elapseTime` 方法。该方法应该在一个时间步之后更新地图上每个位置的置信度。`self.getPositionDistribution` 可以用来访问幽灵的动作分布。给定幽灵先前的位置，为了获得其在新位置上的分布，可以使用以下代码：

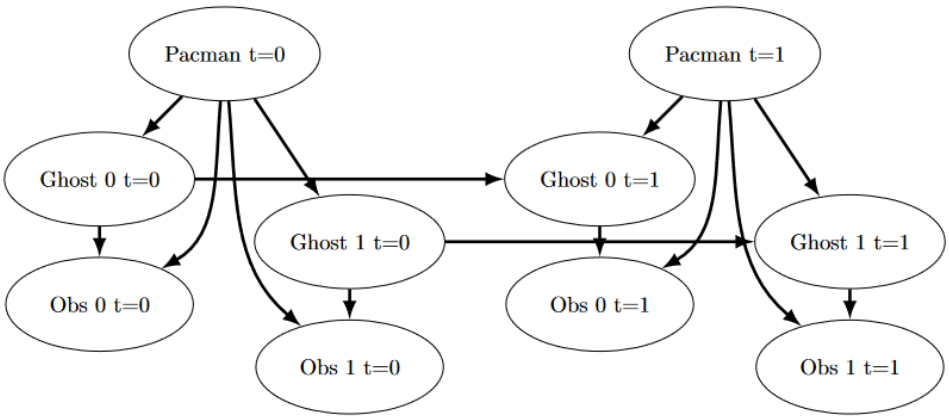
```
newPosDist = self.getPositionDistribution(gameState, oldPos)
```

其中 `oldPos` 指的是之前的幽灵位置。`newPosDist` 是一个 `DiscreteDistribution` 对象，这里对于 `self.allPositions` 中的每个位置 `p`，`newPosDist[p]`是给定幽灵在时间 `t` 位于位置 `oldPos`，其在时间 `t + 1` 时位于位置 `p` 的概率。需要注意的是，函数调用 `self.getPositionDistribution` 比较耗时，因此如果代码超时，需要考虑是否可以减少对该函数的调用次数。

为了将预测实现与上一个任务中的更新实现分开测试，该任务将不会使用你的更新实现。由于吃豆人没有观测幽灵的动作，这些动作不会影响吃豆人的置信度。给定棋盘的几何形状的位置以及幽灵可能采取的合法移动，随着时间的推移，吃豆人的置信度将反映棋盘上幽灵最有可能位于的位置。

对于该任务测试，有两种类型的幽灵，一种是随机移动的幽灵，另一种是 `GoSouthGhost`，此时幽灵倾向于向南移动，因此随着时间的推移，在没有任何观测的情况下，吃豆人的置信度分布应该开始集中在棋盘的底部。要了解每个测试用例使用哪种类型的幽灵，可以查看`.test` 文件。

下图展示了任务所对应的贝叶斯网络/隐马尔可夫模型。其中有些部分已经为你实现了（即 $P(G_{t+1} \mid gameState, G_t)$ 被实现为 `getPositionDistribution`）



为了测试你的代码并可视化输出，运行：

```
python autograder.py -q q7
```

如果想在没有图形界面的情况下进行测试，可以运行：

```
python autograder.py -q q7 --no-graphics
```

任务 8：精确推理：完整测试（Exact Inference Full Test）

现在，吃豆人知道如何利用他的先验知识和观测结果来确定幽灵在哪里，他已经准备好自己追捕幽灵了。该任务将结合你之前实现的 `observeUpdate` 和 `elapseTime` 来维持一个更新的置信度分布，并且你的简单的贪婪智能体将根据每个时间步的最新分布进行动作选择。在简单的贪婪策略中，吃豆人根据自己的置信度假设每个幽灵都处于最可能的位置，然后向最近的幽灵移动。

实现 `bustersAgents.py` 中 `GreedyBustersAgent` 的 `chooseAction` 方法。吃豆人智能体应该首先找到每个剩余的未捕获幽灵最可能的位置，然后选择一个动作来最小化到最近幽灵的迷宫距离。

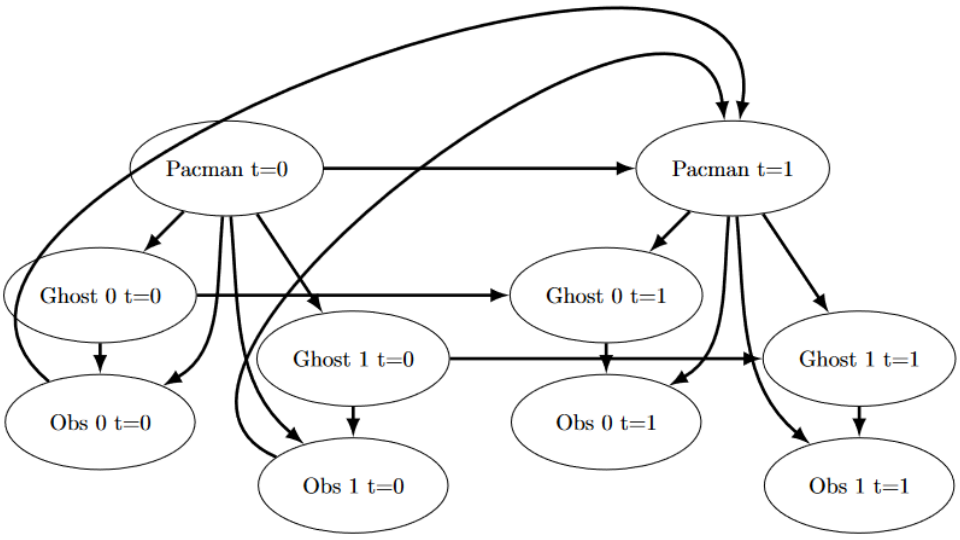
可以使用 `self.distancer.getDistance(pos1, pos2)`来查找任意两个位置 `pos1` 和 `pos2` 之间的迷宫距离。要查找执行某个动作后，一个位置的后继位置：

```
successorPosition = Actions.getSuccessor(position, action)
```

一个 `DiscreteDistribution` 对象列表 `livingGhostPositionDistributions` 可供使用，表示每个尚未捕获的幽灵的位置置信度分布。

如果实现正确，你的吃豆人智能体应该在 `q8/3-gameScoreTest` 中赢得比赛，且 10 次中至少有 8 次得分大于 700。

我们可以通过对刚才的贝叶斯网络图进行以下修改来表示贪婪智能体的工作方式：



为了测试你的代码并可视化输出，运行：

```
python autograder.py -q q8
```

如果想在没有图形界面的情况下进行测试，可以运行：

```
python autograder.py -q q8 --no-graphics
```

任务 9：近似推理：初始化与置信度（Approximate Inference Initialization and Beliefs）

接下来的三个任务将实现用于跟踪单个幽灵的粒子滤波算法。

首先，**实现 `inference.py` 中 `ParticleFilter` 类的函数 `initializeUniformly` 和 `getBeliefDistribution`。** 这里的粒子（样本）是该推理任务中的幽灵位置。需要注意的是，对于初始化，粒子应均匀（而不是随机）分布在合法位置上，以确保先验一致。建议在实现 `initializeUniformly` 时考虑使用 mod 运算符。

注意需要使用列表来存储粒子。列表仅是未加权变量（在本任务中为位置）的集合。 将粒子存储为任何其他数据类型（例如字典）会使程序报错。`getBeliefDistribution` 方法获取粒子列表并将其转换为 `DiscreteDistribution` 对象。

为了测试和调试你的代码，运行：

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

任务 10：近似推理：观测（Approximate Inference Observation）

接下来，需要**实现 `inference.py` 中 `ParticleFilter` 类的 `observeUpdate` 方法。**该方法在 `self.particles` 上构造一个权重分布，其中粒子的权重是给定吃豆人位置和该粒子位置的观测概率。然后，我们从这个加权分布中重采样以构建新的粒子列表。

使用函数 `self.getObservationProb` 来查找给定吃豆人位置、潜在幽灵位置和监狱位置的观测概率。`DiscreteDistribution` 类的 `sample` 方法也会对实现有帮助。
使用 `gameState.getPacmanPosition()`获取吃豆人的位置， `self.getJailPosition()` 获取监狱位置。

需要注意的是，当所有粒子的权重为零时，应通过调用 `initializeUniformly` 重新初始化粒子列表。此时可以考虑使用 `DiscreteDistribution` 的 `total` 方法。

为了测试你的代码并可视化输出，运行：

```
python autograder.py -q q10
```

如果想在没有图形界面的情况下进行测试，可以运行：

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

任务 11：近似推理：时间推移（Approximate Inference with Time Elapse）

实现 `inference.py` 中 `ParticleFilter` 类的 `elapseTime` 函数。该函数应该构造一个新的粒子列表，该列表对应于 `self.particles` 中前进一个时间步的现有粒子，然后将此新列表分配回 `self.particles`。完成该实现之后，应该能够像精确推理一样有效地跟踪幽灵。

请注意，在这个任务中，我们将单独测试 `elapseTime` 函数，以及结合 `elapseTime` 和 `observeUpdate` 的粒子滤波的完整实现。

与 `ExactInference` 类的 `elapseTime` 方法一样，可以使用：

```
newPosDist = self.getPositionDistribution(gameState, oldPos)
```

来获取幽灵在给定其先前位置(`oldPos`)的情况下在新位置上的分布。
`DiscreteDistribution` 类的 `sample` 方法也会对实现有帮助。

为了测试你的代码并可视化输出，运行：

```
python autograder.py -q q11
```

如果想在没有图形界面的情况下进行测试，可以运行：

```
python autograder.py -q q11 --no-graphics
```

注意即使没有图形界面，该测试可能也需要花费数分钟来运行。

各个任务的更详细描述可以参考：
https://inst.eecs.berkeley.edu/~cs188/sp23/projects/proj5/

作业报告

本次作业需要提交报告和代码。对于以上任务，**报告需分别详细介绍代码的实现并分析实验结果**。使用 QQ 群文件中的报告模版（课程作业\报告模版.doc）撰写实验报告。

作业提交

将作业报告存储为 PDF 文件，用学号命名，例如 221900001.pdf，并与相应的源码打包为学号命名的.zip 文件，例如 221900001.zip。

上传到百度网盘：
https://pan.baidu.com/disk/main#/transfer/send?surl=AB0AAAAAABF54w
注意：与之前的课程作业是不同的网址。

提交截止日期：11 月 23 日 23:59:59

学术诚信

允许同学之间的相互讨论，但是**署你名字的工作必须由你自己独立完成**。

如果发现作业之间高度相似将被判定为互相抄袭行为，**抄袭和被抄袭双方的成绩都将被取消**。

应项目开发者的要求，**严禁将作业答案发布在网上**。