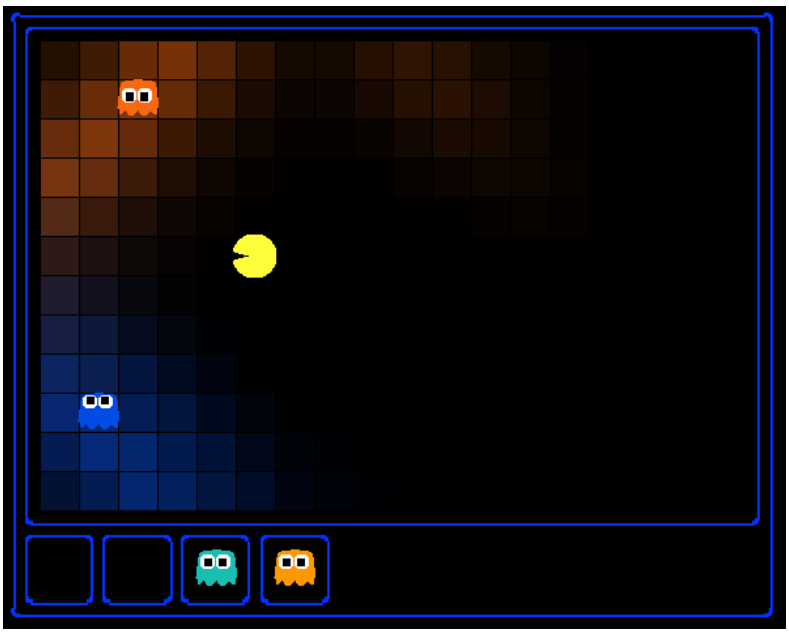
PJ 2：捉鬼敢死队与贝叶斯网络

引言

风水轮流转，一直在躲避幽灵的吃豆人（Pacman）终于从它曾祖父Grandpac那里学习了捕猎幽灵的能力，然而由于它技艺不精，无法看到幽灵，只能通过声音来判断幽灵的位置。

在这个项目中，你将设计完成Pacman代理，实现根据传感器定位并捕猎幽灵。Pacman将从定位单个静止的幽灵开始，逐步发展到高效地狩猎成群移动的幽灵。

Pacman的目标是追捕隐形但会发出声音的幽灵，Pacman拥有一个不精确的传感器，能够将幽灵发出的声音转换成幽灵距离Pacman的曼哈顿距离，其读数与真实距离之间有7以内的误差并始终非负，且距离读数的概率随着与真实距离的差异呈指数级下降，即误差大的读数出现概率更小。



项目文件

项目文件下载见elearning中的附件。该项目中重要文件与对应功能如下：

要编辑的文件	
bustersAgents.py	吃豆人Agent类
inference.py	实现根据声音追踪幽灵
要查看的文件	
factorOperations.py	计算联合或边缘概率表等操作
bayesNet.py	BayesNet类与Factor类
其他可忽略文件	
busters.py	入口文件（取代Pacman.py）
bustersGhostAgents.py	用于捉幽灵敢死队的幽灵agents
distanceCalculator.py	计算迷宫距离，缓存结果以避免重复计算
game.py	Pacman内部运行与支持类
ghostAgents.py	Agents to control ghosts.
graphicsDisplay.py	Pacman图形
graphicsUtils.py	Pacman图形支持
keyboardAgents.py	控制Pacman的键盘接口
layout.py	用于读取布局文件并存储其内容
util.py	实用函数

需要编辑和提交的文件：本项目中，你将填写 `bustersAgents.py`、`inference.py` 的部分内容。 将完成后的文件与其他支持文件打包成zip文件并与报告一同提交，报告中应包括实现思路与评估结果。

本地测试：该项目包括一个自动评分器，用于在本地对所完成的代理进行评分。运行以下代码来对所有测试样例进行评估：

```
1 python autograder.py
```

也可以通过如下命令运行特定测试样例，例如q2-1：

```
1 python autograder.py -q q2-1
```

还可以通过以下命令运行特定测试，如 `test_cases/q2-1/1-ExactUpdate`：

```
1 python autograder.py -t test_cases/q2-1/1-ExactUpdate
```

注意：对于该项目，有时如果使用图形运行测试，自动评分器可能会超时。为了准确地确定你的代码是否足够高效，你可以添加 `--no-graphics` 选项来运行测试：

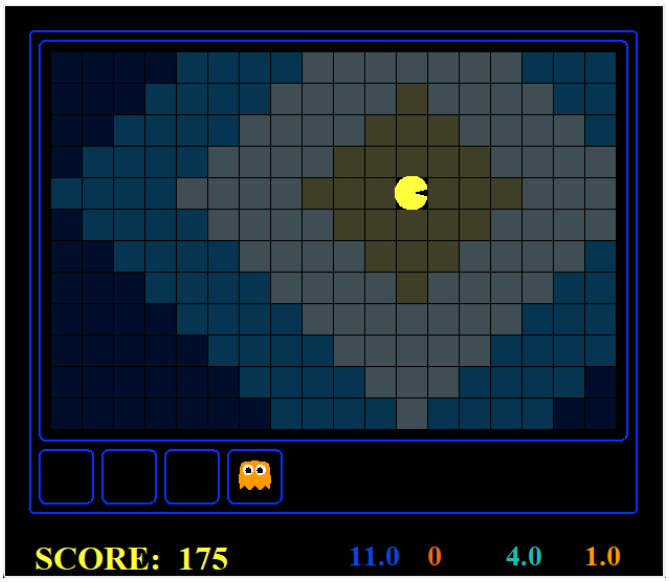
```
python autograder.py --no-graphics
```

测试样例类型: 在使用自动评分器评估所实现的代理时，测试样例分为 `DoubleInferenceAgentTest` 和 `GameScoreTest` 两类

- `DoubleInferenceAgentTest`：此类测试样例中除了地图、幽灵位置和Pacman位置外，还包括一组动作序列和执行对应动作后每个坐标的幽灵出现分布，你所实现的代理将执行所给的动作序列，并计算执行动作后所估计的幽灵分布，与样例中所给分布进行比较计算差异。
- `GameScoreTest`：此类测试样例中只包括地图、幽灵位置和Pacman位置，你所实现的代理将自行决定要执行的动作，根据是否完成游戏和得分进行评估。

在开始项目前，你可以先运行以下代码了解体验该游戏：

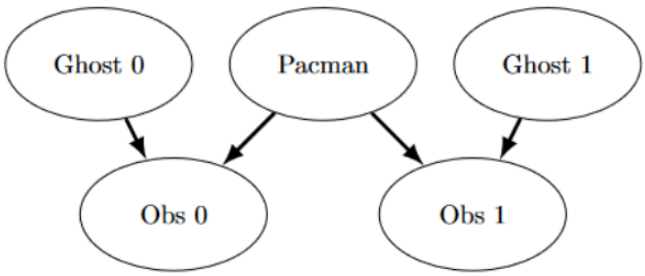
```
python busters.py
```



界面右下方的不同颜色的数字代表传感器显示的对应颜色的幽灵的噪声距离。我们希望通过贝叶斯网络来实现更加精确的位置推理。在这个项目中，你将实现使用贝叶斯网络执行精确推理和近似推理。

Part 1: 一维空间下的捉鬼敢死队——基本数据结构和操作（4分）

考虑简单情况，在一个一维整数轴上，有一个Pacman和两个Ghost在不同的坐标位置。Pacman可以估计与每个Ghost之间的距离，但这些估计存在一定的误差。估计的距离可能与真实距离之间有最多7单位的误差，并且这些误差的大小与真实距离之间的差异呈指数递减的概率分布。问题形式化如下：



- `Pacman`：Pacman的位置。
- `Ghost0` 和 `Ghost1`：两个Ghost的位置。
- `Obs0` 和 `Obs1`：Pacman与两个Ghost的噪声距离。

记噪声为 v ，噪声的概率分布 ☒(☒)可以表达为：

$$P(v) = \frac{2^{7-|v|}}{NORM}, v \in [-7, 7]$$

其中 `NORM` 为归一化因子。

Q1：完成该问题的条件概率表CPT（2分）

Q2：概率计算（2分）

1. $P(Obs = 6|Pacman = 5, Ghost = 12)$
2. $P(Obs1 = 9, Obs2 = 0|Pacman = 5, Ghost1 = 12, Ghost2 = 4)$
3. $P(Ghost = 3|Pacman = 4, Obs = 0)$
4. $P(Ghost1 = 3|Pacman = 4, Obs1 = 9, Obs2 = -1)$
5. $P(Ghost1 = 3, Ghost2 = 3|Pacman = 4, Obs1 = 9, Obs2 = -1)$
6. $P(\text{at least one Ghost at } 3|Pacman = 4, Obs1 = 9, Obs2 = -1)$

注意：Part1的两问请以笔答形式完成！后续提交时扫描成pdf与报告一并提交。

Part 2：精确推理实现（8分）

Pacman 可以被视为一个观测者，而幽灵的状态（位置）是隐含的，即我们无法直接观察到幽灵的确切位置，只能通过Pacman的传感器（观测）来推断。该任务的一些关键组件可以被如下形式化表述：

状态集合 对应于幽灵所有可能的位置以及一个特殊的“监狱”状态，表示幽灵被捕获。

观测集合 对应于Pacman的传感器读数，即Pacman到幽灵的曼哈顿距离的噪声读数。

初始状态概率 对应于游戏开始时，幽灵处于每个可能位置的概率。

状态转移概率 对应于幽灵从一个时间步到下一个时间步可能的移动方式，例如，幽灵不能穿过墙壁或一次移动超过一个空间。

观测概率 对应于给定Pacman的位置和幽灵的某个位置，Pacman的传感器读数的概率分布。

在当前部分，精确推理的目标是基于Pacman的观测和幽灵的可能状态转移，计算幽灵在每个时间步处于每个可能位置的概率。

本次作业中，同学们需要填写 `inference.py` 中的 `ExactInference` 类和 `bustersAgents.py` 中的 `GreedyBustersAgent` 类来实现该目标。

Step 1：基于观测的信念分布更新（3分）

实现一个函数来更新信念分布，即在接收到新的观测（Pacman的传感器读数）后，重新计算幽灵可能位置的概率。

此步需要填写 `ExactInference` 类的 `observeUpdate` 方法。

遍历变量 `self.allPositions` 以进行更新，它包含了所有可能的位置以及特殊的监狱位置。信念分布表示幽灵在特定位置出现的概率，并且以 `DiscreteDistribution` 对象的形式存储在一个名为 `self.beliefs` 的字段中，需要对其进行更新。

如下代码将运行这个问题的自动评分器并可视化输出：

```
python autograder.py -q q2-1
```

如果想在没有图形界面的情况下运行这个测试（或下文的任何其他测试），可以在命令行中添加关键词 `--no-graphics`：

```
python autograder.py -q q2-1 --no-graphics
```

Step 2：基于时间的信念分布更新（3分）

实现一个函数来模拟时间流逝对信念分布的影响，即在没有新的观测的情况下，根据幽灵的可能移动更新其位置的概率。

此步需要填写 `ExactInference` 类的 `elapseTime` 方法。

在这个问题中， `elapseTime` 步骤应该在每个时间步之后更新地图上每个位置的信念分布。你的代理可以通过 `self.getPositionDistribution` 来访问幽灵的转移概率分布。要获取幽灵从先前位置 `oldPos` 移动到新位置的分布，可以使用以下代码：

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

其中 `oldPos` 指的是幽灵的先前位置。 `newPosDist` 是一个 `DiscreteDistribution` 对象，它对于 `self.allPositions` 中的每个位置 `p` ， `newPosDist[p]` 表示在时间 `t + 1` 时幽灵在位置 `p` 的概率，假设在时间 `t` 时幽灵位于位置 `oldPos` 。请注意，这个调用可能相当耗时，因此如果你的代码运行超时，一个需要考虑的问题是是否能够减少对 `self.getPositionDistribution` 的调用次数。

如下代码将运行这个问题的自动评分器并可视化输出：

```
1 python autograder.py -q q2-2
```

Step 3：实现完整的精确推理（2分）

结合观测更新与时间更新，实现一个完整的精确推理过程，并指导Pacman选择最优动作来捕捉幽灵。

此步需要填写 `GreedyBustersAgent` 类的 `chooseAction` 方法。

要计算迷宫中任意两个位置 `pos1` 和 `pos2` 之间的距离，请使用以下方法：

```
1 distance = self.distancer.getDistance(pos1, pos2)
```

要确定一个位置在执行某个动作后的后继位置，可以使用以下代码：

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

你已经获得了一个名为 `livingGhostPositionDistributions` 的列表，它是一个 `DiscreteDistribution` 对象的集合，表示每个未被捕获的幽灵位置的信念分布。

如果代理正确实现，它应该能够在 `q2-3/3-gameScoreTest` 这个样例的10次测试中至少有8次超过700分。

如下代码将运行这个问题的自动评分器并可视化输出：

```
python autograder.py -q q2-3
```

Part3：近似推理实现（8分）

该部分的目标是实现一个粒子滤波算法来估计幽灵的位置。

本次作业中，同学们需要填写 `inference.py` 中的 `ParticleFilter` 类来实现该目标。

Step 1：粒子初始化和信念分布转化（2分）

在游戏开始时，创建一组代表幽灵可能初始位置的粒子。这些粒子应该均匀分布在所有可能的位置上，以形成一个关于幽灵位置的先验信念分布。还要实现粒子信息向信念分布的转化，这将粒子集合转换为一个离散分布，其中每个粒子的位置及其计数表示该位置的概率。

此步需要填写 `ParticleFilter` 类的 `initializeUniformly` 方法和 `getBeliefDistribution` 方法。

请注意，存储粒子的变量必须是列表形式。列表本质上是一组未加权的变量（在这个场景中指的是位置）的集合。如果你尝试将粒子以其他数据类型存储（如字典）将会报错。
`getBeliefDistribution` 方法随后会获取这个粒子列表，并将其转换成 `DiscreteDistribution` 对象。这意味着你需要确保粒子数据以列表的形式提供给这个方法，以便它能够正确地执行转换和计算信念分布。

如下代码将运行这个问题的自动评分器并可视化输出：

```
python autograder.py -q q3-1
```

Step 2：基于观测的近似推理（3分）

根据Pacman的观测和每个粒子的位置，使用观测概率来更新每个粒子的权重。这反映了新的观测数据对幽灵位置信念分布的影响。从加权粒子集合中进行重采样，以创建新的粒子集合。重采样过程倾向于保留那些具有更高权重的粒子，这有助于算法集中于更有信息量的区域。

此步需要填写 `ParticleFilter` 类的 `observeUpdate` 方法。

你需要使用 `self.getObservationProb` 函数来计算给定 Pacman 的位置、一个潜在的幽灵位置以及监狱位置的观察概率。此外， `DiscreteDistribution` 类的 `sample` 方法也将非常有用，它可以帮助我们进行随机抽样。

你可以使用 `gameState.getPacmanPosition()` 方法获取 Pacman 的位置，并用 `self.getJailPosition()` 方法获取监狱位置。此时的一种特殊情况是，当所有粒子获得的权重为零时，应调用 `initializeUniformly()` 重新初始化粒子列表。 `DiscreteDistribution` 类的 `total` 方法也有可能被用到。

如下代码将运行这个问题的自动评分器并可视化输出：

```
python autograder.py -q q3-2
```

Step 3：基于时间的近似推理（3分）

在没有新的观测数据的情况下，根据幽灵的状态转移概率来更新粒子的位置。这模拟了时间流逝对幽灵位置的影响。通过粒子滤波算法预测幽灵在下一时间步的可能位置，为Pacman提供追逐幽灵的策略。

此步需要填写 `ParticleFilter` 类的 `elapseTime` 方法。该函数应创建一个新的粒子列表，该列表表示 `self.particles` 中每个现有粒子向前移动一个时间步长，然后将这个新列表重新赋值给 `self.particles` 。实现后，你将能够像精确推理过程一样有效地追踪幽灵。

请注意，在此问题中，我们将独立测试 `elapseTime` 函数，以及结合 `elapseTime` 和 `observeUpdate` 的粒子滤波器的完整实现。与 `ExactInference` 类的 `elapseTime` 方法类似，可以使用以下代码行来获取新位置的分布，基于幽灵的先前位置（ `oldPos` ）：

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

`DiscreteDistribution` 类的 `sample` 方法在此过程中也将非常有用。

如下代码将运行这个问题的自动评分器并可视化输出：

```
python autograder.py -q q3-3
```