دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر

پروژه چهارم مبانی و کاربردهای هوش مصنوعی

نگارش
مهدیه سادات بنیس

استاد درس
دکتر مهدی جوانمردی

نیم سال دوم ۱۴۰۱

• بخش ۱: احتمال مشاهده

توضیح پیاده سازی کد:

ابتدا حالت خاص گفته شده را بررسی میکنیم یعنی زمانی که سنسور None میدهد اگر روح در زندان باشد ۱ و در غیر این صورت صفر برمیگردانیم (با توجه به توضیحات دستور کار)

اما در غیر این حالت ابتدا بررسی میکنیم اگر روح در زندان باشد که فاصله درست نیست و احتمال آن صفر است چون باید None میبود. در غیر اینصورت کافیست به کمک تابع getObservationProbability توزیع احتمال را یا در واقع:

P(noisyDistance | manhattanDistance(pacmanPosition, ghostPosition))

به عنوان جواب باز میگردانیم(true distance را با فاصله منهتنی بین پکمن و روح جایگزین میکنم).

```
def getObservationProb(self, noisyDistance, pacmanPosition, ghostPosition,
jailPosition):
        """
        Return the probability P(noisyDistance | pacmanPosition, ghostPosition).
        """
        "*** YOUR CODE HERE ***"
        if noisyDistance == None:
            if jailPosition == ghostPosition:
                return 1
            elif jailPosition != ghostPosition:
                return 0
        else:
            if jailPosition == ghostPosition:
                return 0
        return busters.getObservationProbability(noisyDistance,
                                            manhattanDistance(pacmanPosition, ghostPosition))
```

• بخش ۲: مشاهده استنتاج دقیق

توضیح پیاده سازی کد:

ابتدا لازم است تابع نرمالایز را پیاده کنیم چون میخواهیم در ادامه از آن استفاده کنیم. برای این کار کفایت مقدار متناظر هر کلید بر مجموع تقسیم کنیم:

```
def normalize(self):
        total = self.total()
        if total == 0 or len(self.copy()) == 0:
            return
        else:
            for key in self.keys():
                self[key] = self[key] / total
```

در نهایت تابع آپدیت را پیاده سازی میکنیم به این صورت که برای تمام مکان های ممکن روح و زندان باور جدید ما میشود توزیع احتمالی که در مرحله قبل پیاده سازی کردیم با توجه به observation داده شده و موقعیت روح و پکمن و زندان ضرب در باور قبلیمان. و در نهایت این مقدار را نرمال میکنیم:

```
def observeUpdate(self, observation, gameState):
        for ghost_pos in self.allPositions:
            self.beliefs[ghost_pos] *= self.getObservationProb(observation, \
                                                    gameState.getPacmanPosition(),\
                                                    ghost_pos, self.getJailPosition())
       self.beliefs.normalize()
```

سوال: چرا باید تابع normalize را در این بخش فراخوانی کنیم؟ لزوم استفاده از آن را شرح دهید.

باید توزیع را به گونه ای normalize کنید که مجموع مقدار کل باورها به ۱ برسد. نسبت مقادیر برای همه کلیدها یکسان خواهد ماند. در موردی که مقدار کل توزیع ۰ است، هیچ کاری انجام نمیدهیم.

سوال: توضیح دهید مقدار beliefs در گذر زمان چگونه تغییر میکند؟

باتوجه به توضیح پیاده سازی کد دیدم که هر بار مقدار چگونه اپدیت میشود و چون هر بار به نتیجه های بهتری میرسیم مقدار beliefs همگرا میشود.

• بخش ۳: استنتاج دقیق با گذشت زمان

توضیح پیاده سازی کد:

در این تابع میخواهیم باور را پس از سپری شدن یک مرحله زمانی به روز کنیم برای این کار توزیع احتمالاتی عملی که موقعیت قبلی روح را از ghost_pos به gameState برساند را پیدا میکنیم و آن ها را با وزنی برابر با باورمان برای موقعیت روح با هم جمع میکنیم تا باور جدید را بسازیم.

```
def elapseTime(self, gameState):

        temp_dist = DiscreteDistribution()
        for pos in self.allPositions:
            temp_dist[pos] = 0
        for ghost_pos in self.allPositions:
            new_pos_dist = self.getPositionDistribution(gameState, ghost_pos)
            for new_pos in new_pos_dist:
                temp_dist[new_pos] += new_pos_dist[new_pos]* self.beliefs[ghost_pos]
        self.beliefs = temp_dist
```

سوال: کاربرد کلاس DiscreteDistribution را به همراه متدهای آن توضیح دهید.

کلاس DiscreteDistribution توزیع باورها و توزیع وزن را بر روی مجموعه محدودی از کلیدهای گسسته مدل می کند.

def __getitem__(self, key)

این تابع مقادیر کلاس مورد نظر را به صورت دیکشنری ایجاد میکند و کمک میکند خواندن مقادیر از دیکشنری باشند.

def copy(self)

یک کپی از توزیع بر میگرداند

def argMax(self)

کلید ماکسیمم مقدار در دیکشنری را برمیگرداند

def total(self)

جمع مقدار های value کل کلید ها را برمیگرداند

def normalize(self)

این متد هم در بخش یک توضیح داده شد که فرآیند نرمال کردن روی دیکشنری انجام میشود

def sample(self)

این نمونه رندم از دیکشنری برمیدارد و کلید مقدار وزن دهی شده آن را برمیگرداند

سوال: چگونه ساختار شبکه بیزین در پروژه وابستگی‌های بین متغیرها را نشان میدهد؟

یک شبکه بیزی برای نشان دادن وابستگی بین متغیرها استفاده می کند. ساختار شبکه، با گره‌هایی که متغیرها را نشان می‌دهند و یال‌های هدایت‌شده که وابستگی‌ها را نشان می‌دهند، این روابط را به‌طور بصری نشان می‌دهند. جهت یال ها روابط علی یا شرطی بین متغیرها را نشان می دهد.

وابستگی ها با استفاده از توزیع های احتمال شرطی بدست می آیند. هر گره در شبکه دارای یک CPD مرتبط است که توزیع احتمال آن گره را با توجه به والدینش توصیف می کند. CPD ها مشخص می کنند که چگونه مقادیر گره های والد بر توزیع احتمال گره فرزند تأثیر می گذارد.

با بررسی ساختار شبکه بیزی می توان وابستگی مستقیم و غیرمستقیم بین متغیرها را تعیین کرد. گره هایی که مستقیماً به هم متصل نیستند، اگر والدین مشترکی داشته باشند یا اگر به طور غیرمستقیم از طریق متغیرهای دیگر به هم متصل شوند، ممکن است همچنان

وابستگی داشته باشند. اگر هیچ لبه مستقیمی بین دو گره وجود نداشته باشد، به معنای استقلال شرطی است، به این معنی که متغیرها با توجه به والدین مشترک خود مستقل از یکدیگر هستند.

به طور خلاصه، ساختار شبکه بیزی به صورت بصری وابستگی‌های بین متغیرها را از طریق لبه‌های جهت‌دار نمایش می‌دهد و CPD های مرتبط با هر گره به‌طور کمی روابط احتمالی بین متغیرها را نشان می‌دهند.

- **بخش ۴: استنتاج دقیق با تست کامل**

توضیح پیاده سازی کد:

ابتدا یک صف اولویت برای پیدا کردن نزدیک ترین روح پیدا میکنیم و برای پر کردن این صف برای تمامی روح هایی که زنده اند دروترین موقعیت محتمل بر اساس باور را پیدا میکنیم و این موقعیت را با اولویت فاصله روح تا پکمن در صف قرار میدهیم. سپس با استفاده از این صف اولویت نزدیک ترین روح را پیدا میکنیم فاصله اش تا پکمن را پیدا کرده و هر ۴ جهت ممکن برای حرکت پکمن را امتحان میکنیم و هر کدام ما را به روح نزدیک کرد بر میگزینیم اگر حرکت مجاز نبود نیز آن را در نظر نمیگیریم.

```
def chooseAction(self, gameState):
        """
        First computes the most likely position of each ghost that has
        not yet been captured, then chooses an action that brings
        Pacman closest to the closest ghost (according to mazeDistance!).
        """
        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]]
        "*** YOUR CODE HERE ***"
        ghost_queue = util.PriorityQueue()
        for ghost in livingGhostPositionDistributions:
            ghost_position = ghost.argMax()
            ghost_queue.push(ghost_position, self.distancer.getDistance(pacmanPosition,
                                    ghost_position))
        nearest_ghost = ghost_queue.pop()
        current = self.distancer.getDistance(pacmanPosition, nearest_ghost)
        for step in [Directions.WEST, Directions.EAST, Directions.NORTH,
                                    Directions.SOUTH]:
            try:
                next = self.distancer.getDistance(nearest_ghost,
                                            Actions.getSuccessor(pacmanPosition, step))
                if next < current:
                    return step
            except Exception:
                pass

        return Directions.STOP
```