به‌نام خدا

دانشکده مهندسی کامپیوتر

مبانی و کاربردهای هوش مصنوعی ترم بهار ۱۴۰۴

پروژه اول

مهلت تحویل ۱۷ اسفند ساعت ۲۳:۵۹

مقدمه

احتمالا تاکنون بازی پکمن[۱] را انجام داده‌اید یا اگر هم نه، در طول درس با آن آشنا شده‌اید. در این پروژه قصد داریم پکمن را هدایت کنیم تا تمام غذاها را بخورد و به مکانی که هدف نهایی است برسد. ولی با یک تفاوت؛ به جای اینکه شما پکمن را هدایت کنید، پکمن باید خودش این مسیر را پیدا کند. برای اینکار از الگوریتم‌های جستجو کمک میگیریم. در ادامه با ساختار پروژه بیشتر آشنا می‌شویم.

ساختار پروژه

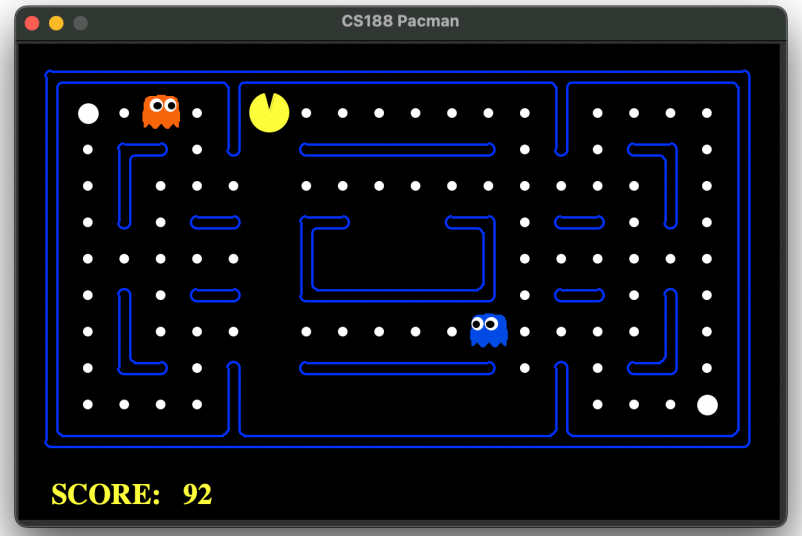
برای اینکه بازی پکمن را اجرا کنید و خودتان پکمن را هدایت کنید، از دستور زیر استفاده کنید:

```
python pacman.py
```

بازی پکمن به صورت زیر اجرا می‌شود و می‌توانید پکمن را با کلیدهای ↑، ↓، ← و → هدایت کنید.

---

[۱] Pacman



ساده‌ترین عامل، عامل `GoWestAgent` است که در فایل `searchAgents.py` تعریف شده است. برای اجرای آن می‌توانید از دستور زیر استفاده کنید:

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

این عامل همیشه به چپ حرکت می‌کند. در مارپیچ `testMaze` این عامل به خوبی عمل می‌کند ولی در مارپیچ‌هایی مانند `tinyMaze` که نیاز به چرخش است، به هدف نمی‌رسد. برای دیدن دستورات بیشتر این پروژه دستور زیر را اجرا کنید:

```
python pacman.py -h
```

همچنین برای دیباگ و تست الگوریتم‌های خود می‌توانید دستور زیر را اجرا کنید و جزئیات آن را ببینید:

```
python autograder.py
```

فایل‌های داخل این پروژه به سه دسته‌ی زیر تقسیم می‌شوند:

- فایل‌هایی که باید ویرایش کنید و ارسال کنید.
- فایل‌هایی که خواندن آن‌ها به درک بهتر شما از پروژه کمک می‌کند.
- فایل‌هایی که منطق بازی، گرافیک و ... را پیاده‌سازی کرده‌اند و نیازی به خواندن آن‌ها نیست.[۲]

[۲] در صورت علاقه می‌توانید آن‌ها را نیز بخوانید.

فایل‌هایی که باید ویرایش کنید:

`search.py`: الگوریتم‌های جستجوی خود را در این فایل باید بنویسید.

`searchAgent.py` عامل‌های جستجو در این فایل قرار دارند.

`util.py` ساختمان‌داده‌های کاربردی برای پیاده‌سازی الگوریتم‌های جستجوی شما در این فایل قرار دارند. ممکن است بخواهید بعضی از آن‌ها را تغییر دهید.

فایل‌هایی که خواندن آن‌ها پیشنهاد می‌شود:

`pacman.py`: فایل اصلی که بازی‌های پکمن را اجرا می‌کند. کلاس `GameState` در این فایل تعریف شده است که وضعیت کنونی بازی مانند مکان پکمن، غذاها و روح‌ها را مشخص می‌کند.

`game.py`: منطق جهان پکمن در این فایل پیاده‌سازی شده است که شامل کلاس‌های `AgentState`، `Agent`، `Grid` و `Direction` می‌شود.

فایل‌هایی که نیازی به خواندن آن‌ها نیست:

`graphicsDisplay.py`: گرافیک بازی پکمن

`graphicsUtils.py.py`: ابزار کمکی گرافیک بازی

`textDisplay.py`: گرافیک کاراکترهای ASCII بازی

`ghostAgents.py`: منطق عامل‌های روح

`keyboardAgents.py`: رابط کیبرد برای کنترل پکمن

`layout.py`: برنامه برای خواندن فایل‌های نقشه و ذخیره‌ی اطلاعات آن‌ها

`autograder.py`: مصحح خودکار پروژه

`testParser.py`: فایل‌های تست و راه‌حل را پردازش می‌کند.

`testClasses.py`: کلاس‌های مربوط به `autograder` در این فایل قرار دارند.

`test_cases/`: testcase های هر سوال در این پوشه هستند.

`searchTestClasses.py`: کلاس‌های `autograder` پروژه‌ی اول

شما باید بخش‌هایی از فایل‌های `search.py`، `searchAgent.py` و `util.py` را پر کنید و تغییر دهید و صرفا همین فایل‌ها را ارسال کنید. **لطفا تغییری در سایر فایل‌ها ندهید.**

توجه: پاسخ کامل به سوالات تشریحی و ارائه توضیحات به همراه اسکرین‌شات برای بخش‌های پیاده‌سازی در این پروژه الزامی می‌باشد و بخش قابل‌توجهی از نمره را تعیین می‌کند.

پیدا کردن یک نقطه‌ی ثابت غذا با استفاده از DFS

در فایل `searchAgents.py` عامل‌های مختلفی تعریف شده‌اند و برخی از عامل‌ها را باید خودتان کامل کنید. عامل `searchAgent` یک مسیر را مشخص می‌کند و آن را قدم به قدم پیمایش می‌کند. این مسیر توسط الگوریتم‌های مسیریابی ایجاد می‌شود. الگوریتم `tinyMazeSearch` از قبل در فایل `search.py` نوشته شده است. این الگوریتم برای مارپیچ `tinyMaze` به خوبی کار می‌کند ولی برای مارپیچ‌های دیگر مناسب نیست. می‌توانید با دستور زیر آن را اجرا کنید:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

اکنون شما باید الگوریتم‌های دیگر که خالی هستند را پر کنید. برای مشاهده‌ی شبه کد هر الگوریتم به اسلایدهای درس مراجعه کنید.

توجه ۱: یک گره‌ی جستجو علاوه بر اطلاعات وضعیت کنونی‌اش، باید شامل اطلاعاتی شود که بتوان از روی آن مسیر جستجو شده را بازسازی کرد.

توجه ۲: توابع جستجوی شما باید لیستی از `action` ها را برگردانند که باعث شوند عامل شما به هدف خود برسد. این `action` ها باید عملیات معتبر باشند (مثلا عامل نباید از دیوار عبور کند).

توجه ۳: ترجیحا از ساختمان داده‌های موجود در فایل `util.py` مانند پشته، صف و صف اولویت استفاده کنید.

از آنجایی که الگوریتم‌های جستجویی که در درس خواندیم همه ساختاری مشابه دارند و فقط مدیریت fringe آن‌ها متفاوت است، سعی کنید الگوریتم DFS را تا حد امکان درست پیاده‌سازی کنید (پیاده‌سازی سایر الگوریتم‌ها مشابه DFS می‌شود).

یکی از روش‌های نوشتن الگوریتم جستجو پیاده‌سازی تنها یک تابع کلی جستجو است که صف‌بندی آن متناسب با هر الگوریتم تنظیم می‌شود. اگر نام این الگوریتم جستجو را GENERAL-SEARCH[۳] بگذاریم که تابعی به عنوان ورودی جهت مقایسه‌ی اولویت دو گره می‌گیرد، به سوال زیر پاسخ دهید (تابع GENERAL-SEARCH از ساختمان داده‌ی صف اولویت استفاده می‌کند).

سوال ۱: فرض کنید الگوریتم DFS را یکبار با تابع GENERAL-SEARCH پیاده‌سازی کنیم (آن را DFS1 بنامیم) و باری دیگر با پشته پیاده‌سازی کنیم (آن را DFS2 بنامیم). آیا زمان اجرای DFS1 و DFS2 با هم برابر خواهد شد؟ چرا؟

[۳] نیازی به این پیاده‌سازی نیست.

بعد از پیاده‌سازی تابع `depthFirstSearch` می‌توانید دستورات زیر را اجرا کنید و عامل خود را تست کنید:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=dfs
python pacman.py -l mediumMaze -p SearchAgent -a fn=dfs
python pacman.py -l bigMaze -z 0.5 -p SearchAgent -a fn=dfs
```

توجه ۴: برای اینکه الگوریتم DFS شما دچار حلقه‌ی بینهایت نشود، از جستجوی گرافی (و نه درختی) استفاده کنید.

بعد از اجرا، میبینید که برخی خانه‌ها قرمزتر از بقیه می‌شوند.

سوال ۲: خانه‌های قرمزتر چه ویژگی دارند؟ در کجای پروژه رنگ خانه‌ها تنظیم می‌شود؟

جستجوی اول سطح (BFS)

در فایل `search.py` تابع `breadthFirstSearch` را پیاده‌سازی کنید. در اینجا نیز نسخه‌ی گرافی الگوریتم را پیاده‌سازی کنید که از گسترش حالات مشاهده‌شده جلوگیری می‌کند. کد خود را مشابه الگوریتم جستجوی اول عمق تست کنید.

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

```
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

آیا الگوریتم جستجوی اول سطح راه‌حل را با کمترین هزینه پیدا می‌کند؟ اگر نه پیاده‌سازی خود را چک کنید.

راهنمایی: اگر پکمن به شدت آهسته حرکت می‌کند از آپشن زیر استفاده کنید:

```
--frameTime 0
```

سوال ۳: دستور زیر را اجرا کنید. آیا کد شما بدون هیچ‌گونه تغییری مسئله‌ی ۸-پازل را حل می‌کند؟ توضیح دهید.

```
python eightpuzzle.py
```

سوال ۴: اگر فضای حالت بسیار بزرگ باشد، BFS چه مشکلاتی خواهد داشت؟

سوال ۵: اگر حافظه محدود باشد، BFS چگونه می‌تواند بهینه‌تر اجرا شود؟

جستجوی UCS

در بخش قبل دیدیم که الگوریتم جستجوی اول سطح (BFS) در مارپیچ‌هایی که هزینه‌ی هر خانه یکسان بود بهترین مسیر را می‌داد. ولی آیا همیشه شرط برابر بودن هزینه‌ها برقرار است؟ مارپیچ‌های `mediumDottedMaze` و `mediumScaryMaze` نمونه‌هایی از این نوع مارپیچ هستند.

در مارپیچ `mediumDottedMaze` برخی خانه‌ها غذا دارند که هزینه‌ی کمتری نسبت به سایر خانه‌ها دارند و می‌خواهیم پکمن را تشویق به رفتن به این خانه‌ها کنیم. یا در `mediumScaryMaze` خانه‌هایی که شامل ارواح هستند هزینه‌ی بیشتری دارند و باید پکمن را از رفتن به آن‌ها منع کنیم.

برای اینکه پکمن بتواند در این جهان‌ها نیز مسیر بهینه را پیدا کند تابع `uniformCostSearch` را پیاده‌سازی کنید. پس از پیاده سازی الگوریتم خود را با دستورات زیر در مارپیچ‌های مختلف تست کنید:

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -z 0.5 -p StayWestSearchAgent
```

توجه ۱: عامل‌های `StayEastSearchAgent` و `StayWestSearchAgent` هردو از تابع `uniformCostSearch` برای مسیریابی استفاده می‌کنند.

توجه ۲: به دلیل اینکه عامل‌های `StayEastSearchAgent` و `StayWestSearchAgent` توابع هزینه با رشد نمایی دارند، باید به ترتیب هزینه‌ی بسیار پایین و هزینه‌ی بسیار بالا برای مسیر دریافت کنید. (برای دیدن جزئیات بیشتر به فایل `searchAgents.py` مراجعه کنید.)

سوال ۶: در این بخش دیدیم که الگوریتم UCS مسیر بهینه را پیدا می‌کرد. پس دلیل اینکه از الگوریتم‌هایی مانند A^* استفاده می‌شود چیست؟

سوال ۷: فرض کنید پکمن تصمیم بگیرد تا مدتی رژیم غذایی بگیرد و کمتر غذا بخورد. چه تغییری در الگوریتم خود باید ایجاد کنید تا پکمن رژیم خود را رعایت کند؟ به صورت شبه کد بنویسید (نیازی به نوشتن کد آن در پروژه نیست و صرفا نوشتن شبه کد در فایل گزارش کافیست).

جستجوی A^*

در فایل `search.py` در تابع `aStarSearch` یک جستجوی گرافی A^* پیدا سازی کنید. الگوریتم جستجوی A^* یک تابع `heuristic` به عنوان آرگومان ورودی می‌گیرد. تابع‌های `heuristic` دو آرگومان ورودی دارند:

۱. حالت (state) فعلی در مسئله جستجو

۲. خود مسئله جستجو (problem)

تابع `heuristic` که در فایل `search.py` قرار دارد، یک نمونه اولیه و بدیهی برای تابع `nullHeuristic` است.

پیاده سازی: متود دو heuristic منهتن و اقلیدسی را که در توابع `manhattanHeuristic` و `euclideanHeuristic` در فایل `searchAgents.py` قرار دارند، کامل کنید.

شما می‌توانید الگوریتم A^* پیاده‌سازی شده توسط خودتان را برای یک مسئله مسیریابی، به کمک هیوریستیک `manhattan distance` تست کنید.

برای این منظور می‌توانید به کمک دستور زیر کد را اجرا کنید:

```
python pacman.py -l bigMaze -z 0.5 -p SearchAgent \
-a fn=astar,heuristic=manhattanHeuristic
```

پس از اجرای این کد خواهید دید که الگوریتم A^*، جواب بهینه را تا حدی سریعتر از الگوریتم UCS پیدا می‌کند.

سوال ۸: الگوریتم‌های جستجو که تا به این مرحله پیاده‌سازی کرده‌اید را روی `openMaze` اجرا کنید و توضیح دهید چه اتفاقی می‌افتد (تفاوت‌ها را شرح دهید).

سوال ۹: آیا می‌توان الگوریتم A^* را برای محیط‌های تغییرپذیر و پویا تنظیم کرد؟ چگونه؟

سوال ۱۰: در این قسمت، شما هیوریستیک های منهتن و اقلیدسی را پیاده سازی کردید. حال تحقیق کنید چه هیوریستیک های دیگری برای A^* استفاده می‌شود. یکی از این هیوریستیک‌ها را به انتخاب خود پیاده سازی کنید و با هیوریستیک‌های منهتن و اقلیدسی مقایسه کنید.

پیدا کردن همه گوشه‌ها

قدرت واقعی الگوریتم A^* تنها توسط مسائل جستجوی چالش‌برانگیزتر نمایان می‌شود. اکنون می‌خواهیم یک مسئله جدید طراحی کنیم و یک هیوریستیک برای آن طراحی کنیم. در هر مارپیچ که دارای گوشه می‌باشد (از مارپیچ‌های متفاوتی برای این بخش استفاده می‌کنیم)، به ازای هر گوشه یک نقطه در نظر گرفته شده است. مسئله جستجوی جدید ما این است که کوتاه‌ترین مسیری که از هر چهار گوشه بگذرد را در مارپیچ پیدا کنیم (بدون توجه به اینکه در گوشه‌ای غذا وجود دارد یا نه). توجه کنید که در برخی از مارپیچ‌ها مثل `tinyCorners`، کوتاه‌ترین مسیر همیشه اول سمت نزدیک‌ترین غذا نمی‌رود.

راهنمایی: کوتاه‌ترین مسیر در `tinyCorners` به اندازه ۲۸ قدم است.

توجه: حتماً پیش از حل این بخش، بخش اول را به طور کامل حل کنید.

کلاس `CornersProblem` در فایل `searchAgents.py` پیاده‌سازی کنید (این کلاس از قبل تعریف شده است، نیاز است که شما قسمت‌های مورد نیاز را کامل کنید). شما نیاز دارید که یک حالت طراحی کنید که بتواند تمام اطلاعات مورد نیاز برای تشخیص این که آیا مسیر به هر چهار گوشه رفته است یا نه، را مشخص کنید. حال عامل هوشمند شما می‌تواند دو مسئله زیر را حل کند:

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=dfs,prob=CornersProblem
```

```
python pacman.py -l mediumCorners -p SearchAgent -a fn=dfs,prob=CornersProblem
```

برای دریافت نمره کامل این قسمت، حالتی که برای حل مسئله طراحی می‌کنید نباید اطلاعات نامربوط (مثل موقعیت روح‌ها، موقعیت غذاهای اضافه و ...) را شامل شود. مشخصاً `GameState` پکمن به عنوان state برای جستجو استفاده نکنید. خروجی‌های مربوط به دستورات بالا را تحلیل کنید.

راهنمایی: تنها قسمتی از `GameState` که نیاز دارید در پیاده‌سازی خود از آن استفاده کنید، موقعیت اولیه پکمن، موقعیت چهار گوشه و دیوارها است.

هیوریستیک برای مسئله گوشه‌ها

توجه: حتماً پیش از حل این بخش، بخش اول را به طور کامل حل کنید.

یک هیوریستیک غیربدیهی سازگار `cornersHeuristic` برای `CornersProblem` موجود در تابع پیاده‌سازی کنید. کد شما باید بتواند مسئله زیر را حل کند:

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

توجه: یک `AStarCornersAgent` برای shortcut دستور زیر است:

```
-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
```

قابل قبول بودن و سازگار بودن: همان‌طور که به یاد دارید، هیوریستیک‌ها توابعی‌اند که یک حالت جستجو را به عنوان ورودی می‌گیرند و عددی را به عنوان هزینه تخمینی تا نزدیک‌ترین هدف برمی‌گردانند. هیوریستیک‌های مفیدتر، مقدار نزدیک‌تر به هزینه واقعی رسیدن به هدف را برمی‌گردانند.

برای آنکه یک هیوریستیک قابل قبول باشد، مقدار هیوریستیک باید از هزینه واقعی کوتاه‌ترین مسیر به نزدیک‌ترین هدف کمتر باشد (همچنین نامنفی باشد).

برای آنکه یک هیوریستیک سازگار باشد، علاوه بر قابل قبول بودن باید اگر عملی هزینه c داشته باشد، انجام آن عمل تنها باعث کاهش مقدار هیوریستیک به مقداری کمتر یا مساوی با c برسد.

به خاطر داشته باشید که قابل قبول بودن، درست بودن یک جستجو گرافی را تضمین نمی‌کند (شما به مسیری قوی‌تر برای سازگار بودن نیاز دارید). با این حال، هیوریستیک‌های قابل قبول اکثر مواقع سازگار هم هستند. به همین منظور، معمولاً آسان‌تر است تا از پیدا کردن یک هیوریستیک قابل قبول برای حل مسئله شروع کنید. وقتی یک هیوریستیک قابل قبول پیدا کردید که خوب کار می‌کند، سازگاری آن را بررسی کنید. تنها راه تضمین سازگاری، اثبات کردن آن است. با این حال، اغلب می‌توان ناسازگاری را با تایید اینکه برای هر گره‌ای که گسترش می‌دهید، گره‌های جانشین آن از نظر مقدار f یا بیش‌تر تشخیص داده شود. علاوه بر این، اگر UCS و A^* مسیرهایی با طول متفاوت بازگردانند، هیوریستیک شما ناسازگار است.

هیوریستیک غیربدیهی: هیوریستیک‌های بدیهی مواردی که همیشه صفر (UCS) و یا هیوریستیک‌هایی که هزینه تکمیل واقعی را محاسبه می‌کنند، هستند. اولی هیچ صرفه‌جویی در زمان برای شما نمی‌کند و دومی باعث به پایان رسیدن زمان `autograder` خواهد شد

(timeout دریافت خطا). شما هیوریستیکی نیاز دارید که کل زمان محاسبه را کاهش دهد. اگرچه برای این تمرین، `autograder` فقط تعداد گره‌ها را بررسی می‌کند (صرف‌نظر از اعمال محدودیت زمانی).

نمره‌دهی: هیوریستیک شما باید غیربدیهی، نامنفی و سازگار باشد تا نمره دریافت کنید. مطمئن شوید که هیوریستیک شما برای حالت‌های هدف مقدار صفر را برگرداند (مقدار منفی نباید برگردانده شود). با توجه به تعداد گره‌هایی که هیوریستیک شما باز می‌کند، به شما نمره داده می‌شود:

نمره	تعداد گره های باز شده
۰/۳	بیش از ۲۰۰۰
۱/۳	حداکثر ۲۰۰۰
۲/۳	حداکثر ۱۶۰۰
۳/۳	حداکثر ۱۲۰۰

توجه: اگر هیوریستیک شما ناسازگار باشد هیچ نمره‌ای از این بخش نمی‌گیرید!

سوال ۱۱: هیوریستیک خود را توضیح دهید و سازگاری آن را استدلال کنید.

سوال ۱۲: هیوریستیک شما چه پارامترهایی (مانند فاصله، موانع، نزدیکی به هدف و غیره) را در نظر می‌گیرد؟ این پارامترها را چگونه در محاسبات هیوریستیک خود ادغام می‌کنید؟ چه پارامترهای دیگری می‌توانند برای هیوریستیک شما مفید باشند؟ نظریه خود را با ادغام کردن دیگر پارامترها آزمایش کنید.

خوردن همه غذاها

در این قسمت قرار است یک مسئله جستجوی سخت را حل کنیم: خوردن همه غذاهای پکمن با کمترین تعداد قدم ممکن. پس ما به تعریف مسئله جستجوی جدیدی نیاز داریم که مسئله دریافت تمام غذاها را پیاده‌سازی کند. به این منظور کلاس `FoodSearchProblem` در فایل `searchAgents.py` برای شما پیاده‌سازی شده است. یک جواب قابل قبول، مسیری است که تمام غذاهای موجود در جهان پکمن را جمع‌آوری کند. برای پروژه فعلی، ارواح هیچ روح یا قدرتی را در نظر نمی‌گیرند. جواب‌ها فقط به محل قرارگیری دیوارها، غذاها و پکمن وابسته است. (البته ارواح می‌توانند اجرای یک راه‌حل را خراب کنند! در پروژه بعدی به آن خواهیم رسید). اگر راه‌های جستجوی کلی را در قسمت‌های قبل به درستی پیاده‌سازی کرده باشید، الگوریتم A^* با هیوریستیک تهی (برابر با UCS) باید به سرعت یک راه‌حل بهینه را به اجرای دستور زیر برای `testSearch` بدون تغییر در کد پیدا کند (هزینه کل برابر با ۷ می‌باشد).

```
python pacman.py -l testSearch -p AStarFoodSearchAgent
```

توجه: `AStarFoodSearchAgent` یک میانبر برای دستور زیر است:

```
-p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic
```

باید توجه کرده باشید که الگوریتم UCS حتی برای مسئله به ظاهر ساده `tinySearch` هم کند عمل می‌کند. به عنوان مرجع، در پیاده‌سازی ما ۲.۵ ثانیه طول می‌کشد تا مسیری به طول ۲۷ را پس از گسترش ۵۰۵۷ گره جستجو پیدا کند.

توجه: حتما پیش از حل این بخش، بخش جستجوی A^* را به طور کامل حل کنید.

توجه: تابع `foodHeuristic` موجود در فایل `searchAgents.py` را با یک هیوریستیک سازگار برای `FoodSearchProblem` تکمیل کنید. سپس عامل خود را با استفاده از دستور زیر امتحان کنید:

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

عامل UCS ما با کاوش در بیش از ۱۶۰۰۰ گره، راه حل مطلوب را در حدود ۱۳ ثانیه پیدا می‌کند.

نمره‌دهی: هر هیوریستیک غیربدیهی، نامنفی و سازگار ۱ نمره دریافت می‌کند. مطمئن شوید که هیوریستیک شما در هر حالت هدف، مقدار صفر بازگرداند و مقدار منفی برنگرداند.

با توجه به تعداد حالت‌هایی که هیوریستیک شما بررسی می‌کند، به شما نمره داده می‌شود:

نمره	تعداد گره‌های باز شده
۱/۴	بیش از ۱۵۰۰۰
۲/۴	حداکثر ۱۵۰۰۰
۳/۴	حداکثر ۱۲۰۰۰
۴/۴	حداکثر ۹۰۰۰
۵/۴	حداکثر ۷۰۰۰

توجه: اگر هیوریستیک شما ناسازگار باشد هیچ نمره‌ای از این بخش نمی‌گیرید!

اگر عامل شما می‌تواند مسئله `mediumSearch` را در زمان کوتاهی حل کند، یا ما خیلی خیلی تحت تأثیر قرار می‌گیریم و یا هیوریستیک شما ناسازگار است.

سوال ۱۳: هیوریستیک خود را توضیح دهید و سازگاری آن را استدلال کنید.

سوال ۱۴: پیاده‌سازی هیوریستیک خودتان در این بخش و در بخش قبلی را با یکدیگر مقایسه و تفاوت‌ها را بیان کنید.

سوال ۱۵: نسخه‌ای از هیوریستیک خود را پیاده‌سازی کنید که جمع‌آوری غذاهای نزدیک‌ترین به موقعیت شروع را اولویت دهد و آن را با نسخه‌ای که غذاهای دورترین به موقعیت شروع را اولویت می‌دهد مقایسه کنید. این تغییرات چگونه بر کارایی جستجو از نظر تعداد گره‌های گسترش یافته و زمان صرف شده تأثیر می‌گذارد؟

جستجوی نیمه بهینه

بعضی مواقع حتی به کمک الگوریتم A^* و یک هیوریستیک مناسب، پیدا کردن مسیر بهینه‌ای که از تمام نقاط عبور کند سخت می‌شود. در این موارد، ما هنوز دوست داریم به سرعت یک راه خوب و منطقی پیدا کنیم. در این بخش، عملی می‌نویسید که همیشه به طور حریصانه نزدیک‌ترین نقطه را می‌خورد، به منظور کلاس `ClosestDotSearchAgent` در فایل `searchAgents.py` برای شما پیاده‌سازی شده است. اما تابعی که مسیر به نزدیک‌ترین نقطه را پیدا کند ناقص است.

تابع `findPathToClosestDot` موجود در تابع `searchAgents.py` را پیاده‌سازی کنید. عامل ما این مارپیچ را (به طول غیر بهینه!)، که از یک طرف با هزینه مسیر ۳۵۰ حل می‌کند.

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z 0.5
```

راهنمایی: اولین کار برای کامل کردن تابع `findPathToClosestDot` کامل کردن تابع `AnyFoodSearchProblem` است که آزمون هدفش کامل نمی‌باشد. سپس مسئله را با یک تابع جستجوی مناسب حل کنید. برای این بخش الگوریتم IDS را مانند بخش یک و دو با استفاده از `problem` که در `findPathToClosestDot` فراهم شده است پیاده کنید (می‌توانید از شبه کدی که در بخش DFS از شما خواسته شده است و پیاده‌سازی DFS (برای ایده گرفتن) استفاده کنید).

توجه: حداکثر عمق برای الگوریتم IDS معادل با ۱۰۰ در نظر گرفته می‌شود.

سوال ۱۶: `ClosestDotSearchAgent` شما، همیشه کوتاه‌ترین مسیر ممکن در مارپیچ را پیدا نخواهد کرد. مطمئن شوید که دلیل آن را درک کرده‌اید و سعی کنید یک مثال کوچک بیاورید که در آن رفتن مکرر به نزدیک‌ترین نقطه منجر به یافتن کوتاه‌ترین مسیر برای خوردن تمام نقاط نمی‌شود.

توضیحات تکمیلی

- این پروژه مشابه پروژه اول دانشگاه برکلی است که تغییرات زیادی در آن اعمال شده است. برای خواندن نسخه‌ی اصلی به این صفحه مراجعه کنید.
- پاسخ به تمرین‌ها باید به صورت انفرادی انجام شود. در صورت استفاده مستقیم از کدهای موجود در اینترنت و مشاهده تقلب، نمره صفر لحاظ خواهد شد.
- پاسخ خود به سوالات که به صورت **سوال** مشخص شده‌اند را در قالب یک فایل PDF به صورت تایپ شده به فرمت `AI_P1Q_student-number.pdf` به همراه سه فایل `search.py`، `searchAgents.py` و `util.py` در قالب یک فایل فشرده با فرمت `AI_P1_student-number.zip` در سامانه کورسز آپلود کنید (توجه کنید نوشتن گزارش الزامی است).
- در صورت هرگونه سوال یا ابهام از طریق ایمیل parsa2201@aut.ac.ir یا seyyedsina.ngh@aut.ac.ir با تدریس یاران در ارتباط باشید. همچنین خواهشمند است در متن ایمیل به شماره دانشجویی خود نیز اشاره کنید.
- همچنین می توانید از طریق تلگرام نیز با آیدی‌های زیر در تماس باشید و سوالاتتان را مطرح کنید:

  – `@parsa22000`

  – `@ssina_ngh`

- این پروژه **تحویل آنلاین یا حضوری از تمام دانشجویان** خواهد داشت و تسلط کافی به سورس کد برنامه ضروری است. بخشی از نمره به صورت ضریب به تسلط شما وابسته است.
- ددلاین این پروژه **۱۷ اسفند ۱۴۰۳ ساعت ۲۳:۵۹** است.
- بودجه‌ی تاخیر همه‌ی پروژه‌ها در کل ۷ روز می‌باشد و به ازای هر روز دیرتر تحویل دادن ۲۰ درصد از نمره‌ی **کل پروژه** کم خواهد شد. بنابراین بودجه‌ی تاخیر خود را مدیریت کنید.