به نام خدا

دانشکده مهندسی کامپیوتر

مبانی و کاربردهای هوش مصنوعی ترم بهار ۱۴۰۰- ۱۴۰۱

پروژه اول

مهلت تحویل ۲۳ فروردین ۱۴۰۱

مقدمه

در این پروژه، عامل پکمن[1] مسیر خود را در مارپیچ به صورت بهینه برای رسیدن به یک مکان خاص و جمع‌آوری غذا پیدا می‌کند. شما الگوریتم‌های جستجوی کلی را پیاده‌سازی می‌کنید و آن‌ها را در سناریوهای بازی پکمن بکار می‌برید.

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

```
python autograder.py
```

[1] Pacman

ساختار پروژه بصورت زیر است و کلیه فایل‌ها در قالب یک فایل زیپ از سامانه کورسز قابل دریافت خواهد بود:

فایل‌هایی که باید ویرایش کنید:	
search.py	فایلی که همه الگوریتم‌های جستجوی شما در آن قرار می‌گیرند.
searchAgents.py	فایلی که همه عامل‌های جستجو در آن قرار می‌گیرند.
فایل‌هایی که شاید بخواهید آن‌ها را ببینید:	
pacman.py	فایل اصلی که بازی‌های پکمن را اجرا می‌کند. این فایل کلاس GameState را برای بازی پکمن توصیف می‌کند که در این پروژه از آن استفاده می‌کنید.
game.py	منطق جهان بازی پکمن در این فایل پیاده‌سازی شده است. این فایل شامل چندین کلاس مانند AgentState(وضعیت عامل)، Agent(عامل)، Grid(نقشه بازی) و Direction(جهت) می‌شود.
util.py	ساختمان داده‌های مفید برای پیاده‌سازی الگوریتم‌های جستجو در این فایل قرار دارند.
فایل‌هایی که می‌توانید از آن‌ها عبور کنید:	
graphicsDisplay.py	گرافیک‌های پیاده‌سازی شده برای بازی پکمن
graphicsUtils.py	پشتیبانی برای گرافیک بازی
textDisplay.py	گرافیک ASCII برای پکمن
ghostAgents.py	عامل‌های کنترل‌کننده ارواح
keyboardAgents.py	رابط صفحه‌کلید برای کنترل پکمن

layout.py	برنامه برای خواندن فایل‌های نقشه و ذخیره اطلاعات آن‌ها
autograder.py	تصحیح‌کننده خودکار پروژه
testParser.py	parse کردن تست های autograder و فایل‌های راه‌حل
testClasses.py	کلاس‌های کلی autograding
test_cases/	پوشه دربردارنده تست‌های مختلف برای هر سوال
searchTestClasses.py	کلاس‌های autograding پروژه ۱

شما باید بخش‌هایی از دو فایل search.py و searchAgents.py را پر کنید. لطفا سایر فایل‌ها را تغییر ندهید.

به پکمن خوش آمدید!

پس از دریافت کد پروژه از سامانه کورسز و خارج کردن آن از حالت فشرده، با تایپ کردن فرمان‌های زیر می‌توانید بازی پکمن را اجرا کنید:

```
cd P1
python pacman.py
```

پکمن در دنیای آبی براقی که پر از راهروهای پیچ در پیچ و غذاهای لذیذ است، زندگی می‌کند. حرکت بهینه در این جهان اولین قدم پکمن برای موفقیت است.
ساده‌ترین عامل در فایل searchAgents.py عاملی با نام GoWestAgent است که همیشه به سمت غرب حرکت می‌کند (یک عامل واکنشی ساده). این عامل گاهی می‌تواند برنده شود:

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

اما وقتی تغییر مسیر نیاز باشد این عامل به خوبی عمل نمی‌کند:

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

اگر پکمن در جایی از بازی گیر کرد با وارد کردن Ctrl+c در ترمینال خود می‌توانید از بازی خارج شوید.
به زودی پکمن شما نه تنها tinyMaze بلکه هر مارپیچ دیگری که بخواهید را نیز می‌تواند حل کند.
توجه: فایل pacman.py از چند option نیز پشتیبانی می‌کند که می‌توان هر کدام را هم به شکل بلند (layout--) و هم به شکل کوتاه (l-) وارد کرد. برای دیدن لیستی از همه optionها و مقادیر پیش‌فرض آن‌ها می‌توانید دستور زیر را وارد کنید:

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

همچنین همه دستوراتی که در این پروژه آورده شده جهت استفاده سریع و راحت در فایل command.txt قرار گرفته‌اند. در سیستم‌عامل‌های مک و یونیکس می‌توانید همه این دستورات را یکجا با وارد کردن دستور زیر اجرا کنید:

```
bash commands.txt
```

۱) پیدا کردن یک نقطه ثابت غذا با استفاده از جستجوی اول عمق (DFS) (۳ امتیاز)

در فایل searchAgents.py می‌توانید کلاس SearchAgent را که بطور کامل پیاده‌سازی شده است پیدا کنید. این عامل یک مسیر را مشخص می‌کند و قدم به قدم آن را طی می‌کند. الگوریتم‌های جستجو پیاده نشده‌اند و پیاده‌سازی آن‌ها بر عهده شماست.

در ابتدا با اجرای دستور زیر مطمئن شوید که SearchAgent به درستی کار می‌کند:

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

دستور بالا به عامل جستجو (SearchAgent) می‌گوید که برای الگوریتم جستجو، از tinyMazeSearch که در فایل search.py پیاده سازی شده است استفاده کند. پکمن باید بتواند با موفقیت در این مارپیچ حرکت کند.

اکنون زمان آن است که توابع جستجوی کامل و کلی را پیاده‌سازی کنید. شبه‌کد الگوریتم‌های جستجو را می‌توانید در اسلایدهای درس مشاهده کنید.

توجه کنید که یک گره جستجو باید علاوه بر اطلاعات مربوط به حالت، اطلاعات لازم برای بازسازی مسیری که به آن حالت می‌رسد را در خود داشته باشد.

نکته مهم: همه توابع جستجوی شما باید یک لیست از اعمالی[2] که عامل را از حالت شروع به هدف می‌رساند را برگردانند. همه این اعمال باید حرکت‌های مجاز باشند (جهت‌های مجاز، نباید از دیوارها عبور کند).

نکته مهم: حتما از ساختمان داده‌های صف، پشته و صف اولویت که در فایل util.py به شکل آماده در اختیارتان قرار داده شده است، استفاده کنید. این ساختمان داده‌ها ویژگی‌های مشخصی دارند که برای سازگاری با autograder لازم هستند.

راهنمایی: الگوریتم‌ها بسیار مشابه‌اند. الگوریتم‌های DFS, BFS, A*, UCS تنها در جزئیات مدیریت fringe تفاوت دارند. پس تلاش کنید ابتدا الگوریتم DFS را به درستی پیاده‌سازی کنید و پیاده‌سازی بقیه الگوریتم‌ها مشابه با پیاده‌سازی DFS خواهد بود.

البته یکی از روش‌ها پیاده‌سازی تنها شامل یک تابع جستجوی کلی است که با یک صف‌بندی مخصوص به هر الگوریتم تنظیم شده است؛ به شکلی که تنها بخش متفاوت بین الگوریتم‌ها بخش مربوط به صف‌بندی

[2] Actions

می‌باشد و جستجوی کلی یکسان است (برای دریافت نمره کامل نیازی نیست که حتما این روش را پیاده‌سازی کنید).

الگوریتم جستجوی اول عمق را در تابع depthFirstSearch موجود در فایل search.py پیاده‌سازی کنید. برای اینکه الگوریتم شما کامل باشد، ورژن جستجوی گرافی DFS را پیاده‌سازی کنید که حالاتی که قبلا مشاهده شده‌اند را گسترش نمی‌دهد.

کد شما باید به سرعت راه‌حل را برای حالات زیر بیابد:

```
python pacman.py -l tinyMaze -p SearchAgent
```

```
python pacman.py -l mediumMaze -p SearchAgent
```

```
python pacman.py -l bigMaze -z .5 -p SearchAgent
```

صفحه پکمن حالاتی را که کاوش شده‌اند را به ترتیب نشان می‌دهد (هرچه رنگ قرمز روشن‌تر باشد به معنای این است که اخیراً کاوش شده است و به مرور زمان کم‌رنگ‌تر می‌شود).

سوال: آیا ترتیب کاوش همان ترتیبی بود که انتظار داشتید؟ آیا پکمن در راه رسیدن به هدف، از همه مربع‌های کاوش شده می‌گذرد؟

راهنمایی: اگر از ساختمان داده پشته استفاده می‌کنید، مسیر به دست آمده از الگوریتم DFS برای mediumMaze باید طولی برابر ۱۳۰ داشته باشد (با فرض اینکه successorها را با ترتیبی که getSuccessors مشخص می‌کند به انتهای fringe اضافه کنید. اگر این کار را با ترتیب عکس انجام دهید ممکن است طولی برابر با ۲۴۶ داشته باشد).

سوال: آیا این راه‌حل کمترین هزینه را دارد؟ اگر نه فکر کنید که جستجوی اول عمق چه کاری را اشتباه انجام می‌دهد.

۲) جستجوی اول سطح (BFS) (۳ امتیاز)

الگوریتم جستجوی اول سطح را در تابع breadthFirstSearch موجود در فایل search.py پیاده‌سازی کنید. در اینجا نیز ورژن 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 را پیدا می‌کند؟ اگر نه پیاده‌سازی خود را چک کنید.

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

```
--frameTime 0
```

نکته: اگر کد جستجوی خود را به صورت کلی نوشته باشید، کد شما باید بدون تغییر به خوبی برای حل مسئله ۸-پازل نیز کار کند.

```
python eightpuzzle.py
```

سوال: تفاوت در تعداد گره‌های کاوش شده در دو پیاده‌سازی BFS و DFS را برای حالت‌های mediumMaze و bigMaze توضیح دهید. (توجه نمایید که این توضیح باید با اشاره به بخش‌های متفاوت این دو پیاده‌سازی در کد خودتان می‌باشد و در صورت عدم اشاره به آن‌ها نمره‌ای را برای این سوال دریافت نخواهید کرد)

۳) تغییر تابع هزینه (۳ امتیاز)

در حالیکه BFS مسیر با کمترین اعمال مورد نیاز برای رسیدن به هدف را می‌یابد، شاید بخواهیم مسیرهایی را بیابیم که از جهات دیگری بهترین هستند. دو مارپیچ mediumDottedMaze و mediumScaryMaze را در نظر بگیرید.

با تغییر تابع هزینه می‌توانیم پکمن را برای پیدا کردن مسیرهای متفاوت تشویق کنیم. به عنوان مثال، می‌توانیم هزینه بیشتری برای حرکت‌های خطرناک در مناطقی که شامل ارواح هستند یا هزینه کمتری برای حرکت در مناطقی که غذا در آن زیاد است در نظر بگیریم و یک عامل پکمن منطقی باید رفتارهایش را با توجه به این هزینه‌ها تنظیم کند.

الگوریتم جستجوی گرافی UCS را در تابع uniformCostSearch موجود در فایل search.py پیاده‌سازی کنید. پیشنهاد می‌شود که به فایل util.py مراجعه کنید و از ساختمان‌داده‌هایی که می‌توانند مفید باشند استفاده کنید.

حالا باید بتوانید رفتارهای موفقیت‌آمیز عامل را در سه نقشه زیر ببینید. عامل‌ها در تمام حالت‌های زیر از الگوریتم UCS استفاده می‌کنند که تنها در تابع هزینه‌ای که استفاده می‌کنند تفاوت دارند (عوامل و توابع هزینه برای شما نوشته شده است).

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

```
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
```

```
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

نکته: شما باید برای StayEastSearchAgent و StayWestSearchAgent به دلیل اینکه تابع هزینه‌ای با رشد نمایی دارند، به ترتیب هزینه‌ای بسیار پایین و بسیار بالایی را برای مسیر داشته باشید (برای جزئیات بیشتر به فایل searchAgents.py مراجعه کنید).

سوال: آیا ممکن است که با مشخص کردن یک تابع هزینه مشخص برای الگوریتم UCS، به الگوریتم BFS و یا DFS برسیم؟ در صورت امکان برای هرکدام از الگوریتم‌های BFS و یا DFS، تابع هزینه مشخص شده را با تغییر کد خود توضیح دهید (نیاز به پیاده‌سازی کد جدیدی نیست؛ صرفا تغییراتی را که باید به کد خود اعمال کنید را ذکر نمایید).

۴) جستجوی A استار (A*) (۳ امتیاز)

در فایل search.py و در تابع خالی aStarSearch یک جستجوی گرافی A* پیاده سازی کنید. الگوریتم جستجوی A* یک تابع heuristic به عنوان آرگومان ورودی می‌گیرد. تابع های heuristic دو آرگومان ورودی دارند:

1) حالت (state) فعلی در مسئله جستجو

2) خود مسئله جستجو (problem)

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

شما می‌توانید الگوریتم A* پیاده‌سازی شده توسط خودتان را برای یک مسئله مسیریابی، به کمک هیوریستیک manhattan distance تست کنید. (این heuristic در تابعی به نام manhattanHeuristic در فایل searchAgents.py پیاده سازی شده است)

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

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a
fn=astar,heuristic=manhattanHeuristic
```

پس از اجرای کد با این الگوریتم خواهید دید که الگوریتم A*، جواب بهینه را تا حدی سریع‌تر از الگوریتم UCS[3] پیدا می کند.

سوال: الگوریتم‌های جستجویی که تا به این مرحله پیاده سازی کرده‌اید را روی openMaze اجرا کنید و توضیح دهید چه اتفاقی می‌افتد (تفاوت‌ها را شرح دهید).

[3] Uniform Cost Search

۵) پیدا کردن همه گوشه‌ها (۳ امتیاز)

قدرت واقعی الگوریتم A* تنها توسط مسائل جستجوی چالش‌برانگیزتر نمایان می‌شود. اکنون می‌خواهیم یک مساله جدید طراحی کنیم و یک heuristic 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 آنکه در گوشه ای 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: حتما پیش از حل بخش ۵، بخش ۲ را به طور کامل حل کنید.

کلاس CornersProblem را 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 کند. حال عامل هوشمند شما می‌تواند دو مساله زیر را حل کند:

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

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

برای دریافت نمره کامل این قسمت، حالتی که برای حل مساله طراحی می‌کنید نباید اطلاعات نامربوط (مثل موقعیت روح‌ها، موقعیت غذاهای اضافه و ...) را شامل شود. مشخصاً از GameState پکمن به عنوان state برای جستجو استفاده نکنید. breadthFirstSearch پیاده‌سازی شده، بر روی مساله mediumCorners حدود ۲۰۰۰ گره را برای جستجو بررسی می‌کند؛ اما با استفاده از هیوریستیک و جستجوی A*، می‌توان این مقدار را کاهش داد.

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

۶) هیوریستیک برای مسئله گوشه‌ها (۳ امتیاز)

توجه: حتما پیش از حل بخش ۶، بخش ۴ را به طور کامل حل کنید.

یک هیوریستیک غیربدیهی سازگار[4] برای CornersProblem موجود در تابع cornersHeuristic پیاده‌سازی کنید. کد شما باید بتواند مساله زیر را حل کند:

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

توجه: AStarCornersAgent یک shortcut برای دستور زیر است:

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

قابل قبول بودن[5] **و سازگار بودن:** همان طور 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، مقداری 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 نمی کند (شما به شرط قوی تری برای سازگار بودن نیاز دارید). با این حال، هیوریستیک‌های قابل قبول اکثر مواقع سازگار هم هستند. به همین منظور، معمولا آسان‌تر است تا از پیدا کردن یک هیوریستیک قابل قبول برای حل مساله شروع کنید. وقتی یک هیوریستیک قابل قبول پیدا کردید که خوب کار می‌کند، سازگاری آن را چک کنید. تنها راه تضمین سازگاری، اثبات کردن آن است. با این حال، اغلب می توان ناسازگاری را با تأیید اینکه برای هر گره‌ای که گسترش می دهید، گره‌های جانشین آن از نظر مقدار f برابر یا بیشتر

[4] Non-trivial, consistent heuristic

[5] Admissibility

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 می کنند، هستند. اولی هیچ صرفه‌جویی در زمان برای شما نمی‌کند و دومی باعث به پایان رسیدن زمان autograder خواهد شد (دریافت خطا timeout). شما هیوریستیکی نیاز دارید که کل زمان محاسبه را کاهش دهد. اگرچه برای این تمرین، autograder فقط تعداد گره ها را بررسی می کند (صرف نظر از اعمال محدودیت زمانی).

نمره دهی: 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 نمره ای از این بخش نمی گیرید!

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

۷) خوردن همه غذاها (۴ امتیاز)

در این قسمت قرار است یک مساله جستجوی سخت را حل کنیم: خوردن همه غذاهای پکمن با کم‌ترین تعداد قدم ممکن. پس به تعریف مساله جستجوی جدیدی نیاز داریم که مساله دریافت تمام غذاها را پیاده 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، راه حل‌ها 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[6] (برابر با UCS) باید به سرعت یک راه‌حل بهینه را با اجرای دستور زیر برای testSearch بدون تغییری در کد پیدا کند (هزینه کل برابر با ۷ می‌باشد).

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

توجه:AStarFoodSearchAgent یک shortcut 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 پیاده سازی 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 بخش ۷، بخش ۴ را به طور کامل حل کنید.

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

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

[6] Null heuristic

عامل UCS ما با کاوش در بیش از ۱۶۰۰۰گره، 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 می شود:

نمره	تعداد گره های 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 مساله mediumSearch را در زمان کوتاهی حل کند، یا ما خیلی خیلی تحت تاثیر قرار می‌گیریم و یا هیوریستیک شما ناسازگار است.

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

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

۸) جستجوی نیمه بهینه[7] (۳ امتیاز)

بعضی مواقع حتی به کمک الگوریتم A* و یک هیوریستیک مناسب، پیدا کردن مسیر بهینه‌ای که از تمام نقطه‌ها عبور کند سخت می‌شود. در این موارد، ما هنوز دوست داریم به سرعت یک راه خوب و منطقی پیدا کنیم. در این بخش، عاملی می‌نویسید که همیشه به طور حریصانه نزدیک‌ترین نقطه را می‌خورد. به این 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 طور غیر بهینه!) در کمتر از یک ثانیه با هزینه مسیر 350 حل می‌کند.

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

راهنمایی:سریع‌ترین راه برای کامل کردن تابع findPathToClosestDot کامل کردن AnyFoodSearchProblem است که آزمون هدف[8] اش کامل نمی‌باشد. سپس مسئله را با یک تابع جستجوی مناسب حل کنید. راه حل باید خیلی کوتاه باشد.

سوال: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.

[7] Suboptimal
[8] Goal test

توضیحات تکمیلی

- برای آشنایی با یونیکس و پایتون می‌توانید به این صفحه[9] مراجعه کنید.
- این پروژه نسخه ترجمه شده پروژه اول دانشگاه برکلی است که برای راحتی شما تدوین شده است. برای خواندن نسخه اصلی می‌توانید به این صفحه[10] مراجعه کنید.
- پاسخ به تمرین ها باید به صورت فردی انجام شود. در صورت استفاده مستقیم از کدهای موجود در اینترنت و مشاهده تقلب، برای همه‌ی افراد نمره صفر لحاظ خواهد شد.
- پاسخ خود به سوالات که در فایل به شکل سوال مشخص شده‌اند را در قالب یک فایل PDF بصورت تایپ شده با فرمت AI_P1Q_StdNum.pdf به همراه دو فایل **search.py** و **searchAgents.py** در قالب یک فایل فشرده با فرمت AI_P1_StdNum.zip در سامانه کورسز آپلود کنید.
- در صورت هرگونه سوال یا ابهام از طریق ایمیل ai.aut.spring1401@gmail.com با تدریسیاران در تماس باشید، همچنین خواهشمند است در متن ایمیل به شماره دانشجویی خود اشاره کنید.
- همچنین می‌توانید از طریق تلگرام نیز با آیدی‌های زیر در تماس باشید و سوالاتتان را مطرح کنید:
  - @lilhedi
  - @koroshroohi
  - @Sarvenaz_srv
- این پروژه تحویل آنلاین نیز خواهد داشت و تسلط کافی به سورس کد برنامه ضروری است. بخشی از نمره به‌صورت ضریب به تسلط شما وابسته است.
- ددلاین این پروژه ۲۳ فروردین ۱۴۰۱ است و امکان ارسال با تاخیر وجود ندارد، بنابراین بهتر است انجام تمرین را به روز های پایانی موکول نکنید.

[9] https://inst.eecs.berkeley.edu/~cs188/su21/project0/#unix-basics
[10] https://inst.eecs.berkeley.edu/~cs188/su21/project1/