به نام خدا

دانشکده مهندسی کامپیوتر

مبانی و کاربردهای هوش مصنوعی ترم پاییز ۱۴۰۰

پروژه اول

مهلت تحویل ۷ آبان ۱۴۰۰

مقدمه

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

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

```
python autograder.py
```

ساختار پروژه بصورت زیر است و کلیه فایل‌های مورد نیاز در فایل زیپ موجود در سامانه کورسز 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 پیاده شده برای دنیای پکمن در این فایل قرار دارد.این فایل شامل چندین کلاس

[1] Pacman

	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 آن‌ها را رد 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 مصحح خودکار و فایل‌های راه‌حل
testClasses.py	کلاس‌های کلی تست خودکار
test_cases/	پوشه دربردارنده تست‌های مختلف برای هر سوال
searchTestClasses.py	کلاس‌های تست خودکار 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 بارگیری 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 در این جهان 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 پیچیدن 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.

فایل **pacman.py** چند آپشن را نیز پشتیبانی می‌کند که هر کدام را هم می‌توان به شکل بلند(--layout) یا کوتاه(-l) وارد کرد. برای دیدن لیستی از همه آپشن‌ها و مقادیر پیش‌فرض آن‌ها می‌توانید دستور زیر را وارد کنید:

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

همینطور همه دستوراتی که در این پروژه آورده شده در فایل **command.txt** نیز برای راحتی کار قرار گرفته‌اند. در سیستم‌عامل‌های مک و یونیکس می‌توانید همه این دستورات را با وارد کردن دستور زیر یکجا اجرا کنید:

```
bash commands.txt
```

۱) پیدا کردن یک نقطه ثابت غذا با استفاده از جستجوی اول عمق(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. پیاده‌سازی الگوریتم‌های جستجو 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 برای کمک به برنامه‌ریزی مسیرها توسط پکمن، پیاده‌سازی کنید.
شبه‌کد الگوریتم‌های جستجو را می‌توانید در اسلایدهای درس مشاهده کنید.
توجه کنید که یک گره جستجو باید اطلاعات مربوط به حالت و همینطور 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 کنید).
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 سرراست خواهد بود.
البته یکی از روش‌ها پیاده‌سازی تنها یک تابع جستجوی کلی است که با یک استراتژی صف‌بندی مخصوص به هر الگوریتم کانفیگ شده است(برای دریافت نمره کامل نیازی نیست که حتما این روش را پیاده‌سازی کنید).
الگوریتم جستجوی اول عمق را در تابع **depthFirstSearch** 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
```

[2] Actions

صفحه پکمن حالاتی را که کاوش شده‌اند و ترتیبی که کاوش شده‌اند را نشان می‌دهد(هرچه رنگ قرمز روشن‌تر باشد به معنای این است که زودتر کاوش شده است).

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

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

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(۳ امتیاز)

الگوریتم جستجوی اول سطح را در تابع **breadthFirstSearch** در فایل **search.py** پیاده‌سازی کنید. در اینجا نیز ورژن گرافی الگوریتم را پیاده‌سازی کنید که از گسترش حالات مشاهده‌شده جلوگیری شود. 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 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 مانند پکمن برای حل مسئله ۸-پازل کار کند.

```
python eightpuzzle.py
```

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 مناطق شامل ارواح 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** در فایل **search.py** پیاده‌سازی کنید. پیشنهاد می‌شود که به فایل **util.py** مراجعه کنید و به دنبال ساختمان داده‌هایی که می‌تواند مفید باشد بگردید.

حالا باید بتوانید رفتارهای موفقیت آمیز عامل را در سه نقشه زیر ببینید. عوامل در همه حالات عامل UCS هستند که تنها در تابع هزینه‌ای که استفاده می‌کنند متفاوت‌اند(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 تابع هزینه نمایی، به ترتیب هزینه مسیر بسیار پایین و بسیار بالایی داشته باشید(برای جزئیات بیشتر به فایل **searchAgents.py** مراجعه کنید).

۴) جستجوی A استار(۳ امتیاز)

در فایل **search.py** و در تابع خالی **aStarSearch** یک جستجوی گرافی A* پیاده سازی کنید. A* ، یک تابع heuristic به عنوان آرگومان ورودی می گیرد. تابع های heuristic دو آرگومان ورودی دارند: 1) حالت(state) فعلی در مساله جستجو 2) خود مساله ی جستجو(problem). تابع **nullHeuristic** که در فایل **search.py** قرار دار، یک نمونه اولیه و بدیهی برای تابع **heuristic** است.

شما میتوانید الگوریتم A* پیاده سازی شده خودتان را بر روی مساله ی پیدا کردن مسیر داخل ماز به نقطه ای مشخص، 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 سریع تر از الگوریتم UCS[3] پیدا می کند.

سوال: الگوریتم های جستجویی که تا به این مرحله پیاده سازی کرده اید را روی **openMaze** اجرا کنید و توضیح دهید چه اتفاقی می افتد.

[3] Uniform Cost Search

۵) پیدا کردن همه گوشه‌ها(۳ امتیاز)

قدرت واقعی الگوریتم A* تنها توسط مسائل جستجوی چالش برانگیزتر نمایان می شود. اکنون می خواهیم یک مساله جدید فرموله کنیم و یک heuristic جدی برای آن طراحی کنیم.

در گوشه های ماز چهار نقطه وجود دارد که هر کدام در یک گوشه قرار دارند. مساله جست و جوی جدید ما این است که کوتاه ترین مسیر را در ماز پیدا کنیم به طوری که مسیر پیا شده از هر چهار گوشه ماز بگذرد (بدون توجه به آنکه در گوشه ای غذا وجود دارد یا نه). توجه کنید که برا برخی از مازها مثل **tinyCorners**، کوتاه ترین همیشه به اول سمت نزدیکترین غذا نمی رود.

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** پیاده سازی کنید (این کلاس از قبل تعریف شده است قسمت های مورد نیاز را کامل کنید). شما نیاز دارید یک نمایش حالت انتخاب 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، نمایش حالتی که برای حل مساله انتخاب می کنید نباید اطلاعات نامربوط(مثل موقعیت روح ها، موقعیت غذاهای اضافه و ...) را شامل شود. در واقع از **GameState** پکمن به عنوان state برای جستجو استفاده نکنید.

راهنمایی: تنها قسمتی از GameState که نیاز دارید در پیاده سازی خود استفاده کنید موقعیت مکان شروع حرکت پکمن و موقعیت چهار گوشه است.

breadthFirstSearch پیاده سازی شده، بر روی مساله **mediumCorners** حدود ۲۰۰۰ گره را برای جستجو باز می کند. اما با استفاده از هیوریستیک و جستجوی A* میتوان این مقدار را کاهش داد.

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** در تابع **cornersHeuristic** پیاده سازی 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، هیوریستیک ها توابعی اند که یک حالت جستجو را به عنوان ورودی می گیرند و عددی را به عنوان خروجی برمی گردانند. عدد خروجی هزینه تخمین زده شده تا نزدیک ترین گره هدف را نشان می دهد. هیوریستیک های مفیدتر، مقداری نزدیکتر به هزینه واقعی تا هدف را برمی گردانند.برای آنکه یک هیوریستیک قابل قبول باشد، مقدار هیوریستیک باید از هزینه واقعی کوتاه ترین مسیر به نزدیک ترین هدف کمتر باشد (و 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 مقدار حداکثر c شود.

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، هیوریستیک های قابل قبول اکثر مواقع سازگار هم هستند. به همین منظور، معمولا آسان تر است تا از فکر کردن برای پیدا کردن یک هیوریستیک قابل قبول برای حل مساله شروع کنید. وقتی یک هیوریستیک قابل قبول پیدا کردید که خوب کار می کند، 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 گره ای که گسترش می دهید، گره های جانشین آن از نظر مقدار f برابر یا بیشتر تشخیص داده شود.علاوه بر این ، اگر UCS و A* مسیرهایی با طول های مختلف بازگردانند، هیوریستیک شما ناسازگار است.

هیوریستیک غیربدیهی: هیوریستیک های بدیهی مواردی که در همه جا صفر (UCS) و یا هیوریستیک هایی که هزینه تکمیل واقعی را محاسبه می کنند، هستند. اولی هیچ صرفه جویی در زمان برای شما نمی کند و دومی باعث به پایان رسیدن زمان autograder خواهد شد. 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).

[4]Non-trivial, consistent heuristic

[5]Admissibility

نمره دهی: هیوریستیک شما باید غیربدیهی، نامنفی و سازگار باشد تا نمره دریافت کنید. مطمئن شوید که هیوریستیک شما در هر گره هدف، مقدار صفر بازگرداند و نه مقدار منفی. با توجه به تعداد گره هایی که هیوریستیک شما باز می کند، به شما نمره داده می شود:

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 نقطه‌ها(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 پاکسازی مواد غذایی را فرموله کند، به این منظور کلاس **FoodSearchProblem** در فایل **searchAgents.py** برای شما پیاده سازی شده است.یک جواب قابل قبول، مسیری است که تمام مواد غذایی موجود در جهان پکمن را جمع آوری کند. برای پروژه فعلی، راه حل ها هیچ روح یا پلت قدرتی را در نظر نمی گیرند. جواب ها فقط به محل قرارگیری دیوارها، غذاها و پکمن وابسته است.(البته ارواح می توانند اجرای یک راه حل را خراب کنند! در پروژه بعدی به آن خواهیم رسید.) اگر متدهای سرچ کلی را در قسمت های قبل به درستی پیاده سازی کرده باشید، الگوریتم **A*** با هیوریستیک تهی[6] (برابر با جستجوی با هزینه یکسان) باید به سرعت یک راه حل بهینه با اجرای دستور زیر برای **testSearch** بدون تغییر کد از سمت شما پیدا کند (هزینه کل ۷).

```
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 جستجو با هزینه یکسان حتی برای مساله به ظاهر ساده tinySearch هم کند عمل می کند. به عنوان مرجع، در پیاده سازی ما ۲.۵ ثانیه طول می کشد 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** 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
```

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

نمره دهی: هر هیوریستیک غیربدیهی، نامنفی ۱ نمره دریافت می کند. مطمئن شوید که هیوریستیک شما در هر گره هدف، مقدار صفر بازگرداند و نه مقدار منفی. با توجه به تعداد گره هایی که هیوریستیک شما باز می کند، به شما نمره داده می شود:

[6] Null heuristic

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 نمره ای از این بخش نمی گیرید!

اگر عامل شما می تواند مساله **mediumSearch** را در زمان کوتاهی حل کند، یا ما خیلی خیلی تحت تاثیر قرار می گیریم و یا هیوریستیک شما ناسازگار است.

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

۸) جستجوی نیمه بهینه[7](۳ امتیاز)

بعضی مواقع حتی به کمک الگوریتم A* و یک هیوریستیک مناسب هم پیدا کردن مسیر بهینه از میان تمام نقطه ها سخت می شود. در این موارد ، ما هنوز دوست داریم به سرعت راه خوبی پیدا کنیم.در این بخش، عاملی می نویسید که همیشه به طور حریصانه نزدیکترین نقطه را می خورد.به این منظور کلاس **ClosestDotSearchAgent** در فایل **searchAgents.py** برای شما پیاده سازی شده است. اما تابعی که مسیر به کوتاه ترین نقطه را پیدا کند ناقص است.

تابع **findPathToClosestDot** در تابع **searchAgents.py** را پیاده سازی کنید.عامل ما این ماز را (به طور غیر بهینه!) در کمتر از یک ثانیه با هزینه مسیر 350 حل می کند.

```
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 ناکامل است.سپس مساله را با یک تابع جستجوی مناسب حل کنید. راه حل باید خیلی کوتاه باشد.

سوال: **ClosestDotSearchAgent** شما، همیشه کوتاه ترین مسیر ممکن در ماز را پیدا نخواهد کرد. مطمئن شوید که دلیل آن را درک کرده اید و سعی کنید یک مثال کوچک بیاورید که در آن رفتن مکرر به نزدیکترین نقطه منجر به یافتن کوتاهترین مسیر برای خوردن تمام نقاط نمی شود.

[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_9931099.pdf به همراه دو فایل **search.py** و **searchAgents.py** در قالب یک فایل فشرده با فرمت AI_P1_9931099.zip در سامانه کورسز آپلود کنید.
- در صورت هرگونه سوال یا ابهام از طریق ایمیل ai.aut.fall1400@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
  - @mies47
- 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 **۷ آبان ۱۴۰۰ ساعت ۲۳:۵۵** 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/