قوانین تمرین‌ها (چه نظری – چه عملی)

۱. کپی یعنی صفر 😐 – با هیییییچ احدی هم تعارف نداریم!
۲. کمک گرفتن از دیگر دوستان مشکلی نداره به شرطی که خودتون مفهوم رو درک کرده باشید.
۳. برای تحویل تمرین‌های عملی، در روز مشخص شده، کد رو آنلاین از گیت‌هاب توضیح می‌دید.
۴. برای تحویل تمرین‌های نظری، با فرمت زیر فایل زیپ می‌سازید و در تلگرام به امید محمدی کیا ارسال می‌کنید. (@Saha۳۳۳)

فرمت فایل (به جای # شماره تمرین – به جای؟ ها شماره دانشجویی):

HW#_۹????????

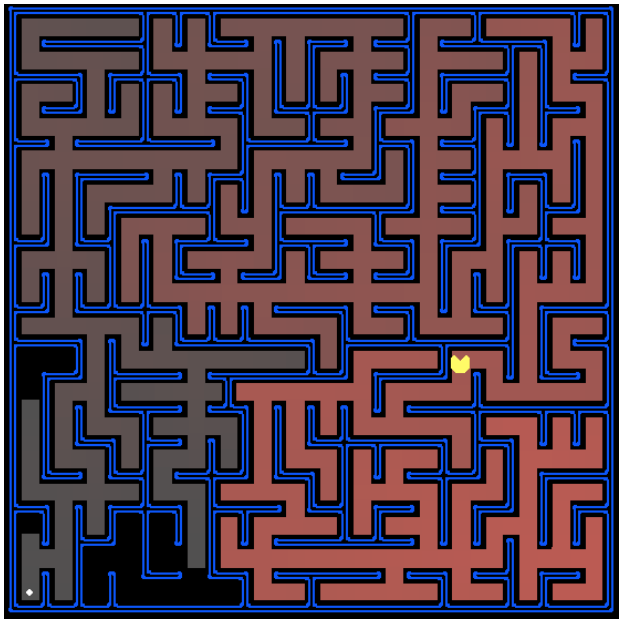
نمونه فردی: HW۱_۹۴۲۱۷۱۰۰۱ نمونه گروهی: HW۱_۹۴۲۱۷۱۰۰۱_۹۴۲۱۷۱۰۰۲

۵. تأخیر و عدم ارسال تمرین و پروژه به این صورت محاسبه میشه:

تأخیر تا ۲۴ ساعت	کسر ۵۰ درصد
تأخیر تا ۷۲ ساعت	کسر ۸۰ درصد
بعد از ۷۲ ساعت - عدم ارسال	صفر

۶. تمامی تمرین‌های نظری به صورت فردی انجام میشه؛ در مورد تمرین‌های عملی، هر تمرین رو جداگانه معلوم خواهیم کرد.

تمرین سری اول برنامه نویسی: سرچ در پکمن



معرفی:

در این پروژه، عامل پکمن شما از بین جهان هزارتوی خود هم برای رسیدن به مکانی خاص و هم برای جمع آوری بهینه غذاها مسیری پیدا خواهد کرد. شما باید الگوریتم‌های عمومی جست و جو را پیاده سازی و آن‌ها را به جهان پکمن اعمال کنید.

این پروژه شامل یک سیستم نمره دهی خودکار است تا بتوانید بعد از اتمام کد خود نمره خود را مشاهده کنید. این عمل با دستور زیر قابل اجراست:

```
python autograder.py
```

نکته: اگر در ترمینال pycharm کد می‌زنید در تمامی دستورهای گفته شده کلمه python اول دستور را حذف کنید

کد این پروژه از چندین فایل پایتون تشکیل شده است، بعضی از آن‌ها را باید بخوانید و متوجه شوید و از بعضی از آن‌ها می‌توانید چشم پوشی کنید. شما می‌توانید همهٔ کدها و فایل‌های کمکی را به صورت یک فایل زیپ از اینجا دانلود کنید.

فایل‌هایی که باید ویرایش کنید:	
search.py	همه‌ی الگوریتم‌های جست و جوی شما اینجا خواهند بود
searchAgents.py	همه‌ی عامل‌های جست و جوی شما اینجا خواهند بود
فایل‌هایی که باید بخوانید ولی ویرایش نکنید:	
pacman.py	فایل اصلی که بازی پکمن را اجرا می‌کند.
game.py	قانونی که پشت نحوه کار دنیای پکمن است.
util.py	ساختار داده‌ای های مفید برای پیاده سازی الگوریتم‌های جست و جو
فایل‌های کمکی که می‌توانید از خواندن آن‌ها چشم پوشی کنید:	
graphicsDisplay.py	گرافیک پکمن
graphicsUtils.py	فایل کمکی برای گرافیک پکمن
textDisplay.py	گرافیک ASCII برای پکمن
ghostAgents.py	عامل‌هایی برای کنترل روح‌ها
keyboardAgents.py	واسط کیبورد برای کنترل پکمن
layout.py	کدهایی برای خواندن فایل‌های لایوت و ذخیره‌ی محتوای آن‌ها
autograder.py	سیستم نمره دهی خودکار پروژه
testParser.py	تجزیه آزمون نمره دهی خودکار و فایل‌های راه حل
testClasses.py	آزمون نمره دهی عمومی کلاس‌ها
test_cases/	مسیری که شامل موارد آزمون برای هر سؤال است
searchTestClasses.py	آزمون نمره دهی کلاس‌های مخصوص پروژه اول

فایل‌هایی که باید ویرایش و ارسال کنید: شما بخش‌هایی از search.py و searchAgents.py را که مشخص شده است در طول این تمرین پر خواهید کرد. لطفاً سایر قسمت‌ها را دست نخورده باقی بگذارید. در طول روند تکمیل پروژه، کد خود را بر روی GitHub قرار دهید!

ارزیابی: کد شما برای ارزیابی صحت فنی توسط سیستم نمره دهی خودکار بررسی خواهد شد. لطفاً نام هیچ کدام از توابع و یا کلاس‌های داخل کدها را تغییر ندهید. در غیر این صورت سیستم نمره دهی خودکار کد شما را رد خواهد کرد. طبق نمرات حاصل شده از سیستم نمره دهی خودکار نمره‌ی سؤال ۵ شما به عنوان نمره‌ی سؤال ۸ درج شده و سؤال ۵ و ٦ و ۷ این سیستم برای شما حذف شده است. با این حال نمره‌ی حاصله از سیستم نمره دهی خودکار، نمره نهایی شما نخواهد بود و صحت پیاده سازی کدها پایه و اساس نمره دهیست.

و حالا:

بعد از دانلود کد (search.zip) خارج کردن آن از حالت فشرده و باز کردن آن با نرم افزار pycharm شما باید بتوانید پکمن را با تایپ کردن دستور زیر در ترمینال pycharm اجرا کنید:

```
python pacman.py
```

پکمن در دنیایی شمال راهروهای تو در تو و میوه‌های خوش مزه زندگی می‌کند. هدایت کارامد پکمن در این جهان قدم اول در تسلط به آن خواهد بود.

ساده‌ترین عامل در searchAgents.py عامل GoWestAgent است که همیشه به سمت غرب می‌رود. این عامل هر از گاهی می‌تواند برنده شود:

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

اما وقتی نیاز به چرخش باشد اوضاع برای این عامل سخت می‌شود:

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

اگر پکمن جایی گیر کرد شما می‌توانید با تایپ کردن CTRL-c در ترمینال خود از بازی خارج شوید.

به زودی عامل شما قادر خواهد بود نه تنها tinyMaze بلکه هر مازی را حل کند.

نکته اینکه pacman.py از چندین گزینه پشتیبانی می‌کند که هر کدام می‌توانند به روش طولانی (layout--) و کوتاه (l-) بیان شوند. شما می‌توانید لیست همه‌ی این گزینه‌ها و مقادیر پیش فرضشان را با دستور زیر مشاهده کنید:

```
python pacman.py –h
```

همچنین، همه‌ی دستوراتی که در این پروژه به کار گرفته می‌شوند در فایل commands.txt موجودند که می‌توانید به راحتی آن‌ها را کپی و پیست کنید.

سؤال ۱: پیدا کردن یک نقطه ثابت غذا با استفاده از DFS

در فایل searchAgents.py شما یک SearchAgent که به طور کامل پیاده سازی شده را پیدا خواهید کرد که مسیری در دنیای پکمن پیدا می‌کند و سپس قدم به قدم آن را طی می‌کند اما الگوریتم‌های جست و جو برای کوتاه کردن یک مسیر پیاده سازی نشده‌اند که این کار شماست.

ابتدا با دستور زیر امتحان کنید که آیا SearchAgent به درستی کار می‌کند یا خیر:

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

دستور بالا به SearchAgent می‌گوید که از tinyMazeSearch به عنوان الگوریتم جست و جوی خود استفاده کند که در فایل search.py پیاده سازی شده است. پکمن باید با موفقیت در هزارتو حرکت کند. حالا زمان آن رسیده که تابع عمومی جست و جوی تکامل یافته را برای کمک به مسیر یابی پکمن بنویسید! الگوریتم کلی جست و جو برای درس مطابق زیر است:

```
function GRAPH-SEARCH(problem, fringe, strategy) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe, strategy)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

نکته مهم: به یاد داشته باشید که یک گره جست و جو باید علاوه بر حالت خود اطلاعات لازم برای بازسازی مسیری که به آن حالت ختم می‌شود را داشته باشد.

نکته مهم: همه‌ی توابع جست و جوی شما باید لیستی از اعمالی که عامل را از نقطه‌ی شروع به نقطه‌ی هدف می‌رسانند برگردانند. این اعمال همگی باید حرکات قانونی باشند (جهت دهی‌های صحیح نه حرکت از بین دیوارها)

نکته مهم: حتماً از ساختار داده‌های Stack، Queue و PriorityQueue که در فایل util.py برای شما آماده شده است استفاده کنید! این ساختار داده‌های پیاده سازی شده مشخصه‌های ویژه‌ای دارند که جهت سازگاری با سیستم نمره دهی خودکار نیاز اند.

راهنمایی: الگوریتم‌های مربوط به DFS, BFS, UCS و A* خیلی به هم شبیه‌اند و از شبه کد اگوریتم جست و جوی کلی بالا مشتق شده‌اند. پس فقط روی پیاده سازی صحیح DFS تمرکز کنید و بعد از آن قادر خواهید بود سایر توابع را به سادگی بنویسید.

الگوریتم depth-first search (DFS) را در تابع depthFirstSearch موجود در فایل search.py پیاده سازی کنید. برای کامل کردن الگوریتم خود نسخه‌ی جست و جوی گرافی (graph search) این الگوریتم را بنویسید که مانع از بسط گره‌هایی می‌شود که قبلاً مشاهده شده‌اند.

کد شما باید به سرعت راه حلی برای دستورهای زیر پیدا کند:

```
python pacman.py -l tinyMaze -p SearchAgent
python pacman.py -l mediumMaze -p SearchAgent
python pacman.py -l bigMaze -z.۵ -p SearchAgent
```

صفحه‌ی پکمن گره‌های بسط داده شده و ترتیب بسط دادن آن‌ها را نشان خواهد داد (قرمز روشن‌تر به معنی بسط دادن زودتر آن گره است). آیا ترتیب بسط گره‌ها همانی بود که انتظار داشتید؟ آیا پکمن واقعاً به تمام گره‌های بسط داده شده در مسیرش تا هدف می‌رود؟

راهنمایی: اگر شما از Stack به عنوان ساختار داده‌ای خود استفاده کردید، راه حل پیدا شده با الگوریتم DFS برای mediumMaze باید طولی برابر با ۱۳۰ داشته باشد. آیا این کم هزینه‌ترین راه حل است؟ اگر نه درباره اینکه چرا جست و جوی DFS جواب بهینه را پیدا نمی‌کند فکر کنید.

سؤال ۲: پیدا کردن یک نقطه ثابت غذا با استفاده از BFS

الگوریتم breadth-first search (BFS) را در تابع breadthFirstSearch موجود در فایل search.py پیاده سازی کنید. مجدد الگوریتم گراف سرچ آن را بنویسید که بسط گره‌های دیده شده خودداری می‌کند. کد خود را به همان روشی که در سؤال قبل بود تست کنید.

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z.۵
```

آیا BFS کم هزینه‌ترین راه حل را پیدا می‌کند؟ اگر نه باید پیاده سازی خود را مجدداً چک کنید.

راهنمایی: اگر پکمن شما خیلی آرام حرکت می‌کند گزینه‌ی frameTime-- را امتحان کنید.

نکته: اگر شما کد جست و جوی خود را به صورت کلی نوشته‌اید، کد شما باید برای مساله ی جست و جوی eight-puzzle هم به خوبی کار کند و نیازی به هیچگونه تغییر نخواهد داشت.

```
python eightpuzzle.py
```

سؤال ۳: تغییر تابع هزینه

در حالی که BFS کوتاه‌ترین مسیر را تا هدف به دست می‌آورد، ممکن است ما به دنبال پیدا کردن مسیرهایی باشیم که از جنبه‌های دیگر بهترین‌اند. ماز های mediumDottedMaze و mediumScaryMaze را در نظر بگیرید.

با عوض کردن تابع هزینه ما می‌توانیم پکمن را ترغیب کنیم تا مسیرهای متفاوتی را پیدا کند. برای مثال، ما می‌توانیم هزینه‌ی بیشتری برای گام‌های خطرناک در مناطقی که روح‌ها حرکت می‌کنند و یا هزینه‌ی کمتر برای گام‌هایی در مناطق نزدیک به غذا ایجاد کنیم و یک عامل هوشمند پکمن باید رفتار خود را در پاسخ به آن تنظیم کند.

الگوریتم گراف جست و جوی UCS را در تابع uniformCostSearch موجود در فایل search.py پیاده سازی کنید. ما به شما توصیه می‌کنیم تا با نگاه به فایل util.py به دنبال ساختار داده‌ای بگردید که در کدتان مؤثر خواهد بود. اکنون شما باید رفتاری موفق را در هر سه لایه مشاهده کنید، جایی که عامل تمامی آن‌ها عامل UCS است که تنها در تابع هزینه‌ای که استفاده می‌کنند متفاوت‌اند. (عامل‌ها و توابع هزینه برای شما نوشته شده‌اند):

```
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 مسیرهای خیلی کم هزینه و بسیار پر هزینه‌ای را به ترتیب به ازای StayEastSearchAgent و StayWestSearchAgent طی توابع هزینه‌ی نمایی آن‌ها به دست اورید (برای جزئیات بیشتر فایل searchAgents.py را ببینید)

سؤال ٤: جست و جوی A*

جست و جوی گراف A* را در تابع خالی aStarSearch در فایل search.py پیاده سازی کنید. A* یک تابع هیوریستیک را به عنوان ورودی خود می‌گیرد. توابع هیوریستیک دو ورودی می‌گیرند: یک حالت در مساله ی جست و جو (آرگومان اصلی) و خود مساله (به عنوان مرجع اطلاعات). تابع هیوریستیک nullHeuristic در فایل search.py یک مثال جزئی از این نوع توابع است.

شما میتوانید پیاده سازی A* خود را روی مساله ی اصلی یافتن مسیری در بین هزارتو تا یک موقعیت ثابت با استفاده از هیوریستیک فاصله‌های منهتن (Manhattan distance heuristic) که با عنوان manhattanHeuristic در فایل searchAgents.py موجود است، امتحان کنید.

```
python pacman.py -l bigMaze -z.۵ -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
```

پس از اجرای آن باید مشاهده کنید که A* راه حل بهینه را کمی سریع‌تر از UCS پیدا می‌کند. برای استراتژی‌های جست و جوی گوناگون چه اتفاقی روی openMaze می افتد؟

سؤال ۵: جست و جوی نیمه بهینه

گاهی اوقات، حتی با A* و یک هیوریستک خوب، یافتن مسیر بهینه در بین همه نقاط سخت است. در این موارد، ما هنوز به دنبال یافتن مسیری خوب و منطقی هستیم. در این بخش شما عاملی خواهید نوشت که همیشه حریصانه نزدیک‌ترین نقطه به خود را بخورد. ClosestDotSearchAgent برای شما در فایل searchAgents.py پیاده سازی شده است اما یک تابع کلیدی برای پیدا کردن مسیری به نزدیک‌ترین نقطه در آن جا مانده است.

تابع findPathToClosestDot را در فایل searchAgents.py پیاده سازی کنید. عامل ما این هزارتو را به صورت نیمه بهینه زیر یک ثانیه با مسیری به هزینه‌ی ۳۵۰ حل خواهد کرد:

```
python pacman.py -l bigSearch –p ClosestDotSearchAgent -z.۵
```

راهنمایی: سریع‌ترین راه برای کامل کردن findPathToClosestDot این است که AnyFoodSearchProblem را کامل کنید که که هدف آن جا افتاده است. سپس مساله را با یک تابع مناسب جست و جو حل کنید. جواب کوتاه خواهد بود.

ClosestDotSearchAgent همیشه کوتاه‌ترین مسیر ممکن در هزارتو را پیدا نخواهد کرد. بررسی کنید که چرا این اتفاق می افتد و مثال ساده‌ای از جایی که خوردن نزدیک‌ترین نقاط به کوتاه‌ترین مسیر ختم نمی‌شود ارائه کنید.