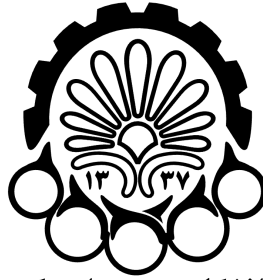


به نام خدا



دانشگاه صنعتی امیرکبیر  
( پلی تکنیک تهران )

دانشکده مهندسی کامپیوتر

مبانی و کاربردهای هوش مصنوعی، پاییز ۱۴۰۳

پروژه یک: پک من

مهلت تحویل: ۲۵ آبان ماه ۱۴۰۳

---

ابتدا فایل پروژه رو از حالت زیپ خارج کنید.

با زدن دستور زیر تو فولدر پروژه، می‌تونین بازی پک من رو اجرا کنید و شروع به بازی کنید.

```
python pacman.py
```

حالا ما می‌خواهیم به کمک الگوریتم‌های جستجویی که یاد گرفتیم به پک من کمک کنیم تا هوشمندتر بازی کنه. ساده‌ترین نوع ایجنت تو فایل `searchAgents.py` ایجنت `GoWestAgent` هست که همیشه به سمت چپ میره.

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

اما وقتی نیاز به چرخیدن باشه به مشکل می‌خوره.

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

اگه پک من گیر کرد می‌تونین با زدن `ctrl-c` تو ترمینال بازی رو متوقف کنین.

فایل `pacman.py` کلی آپشن برای انتخاب داره که می‌تونین به صورت کامل یا به اختصار موقع اجرا بهش بدین. مثلا `--layout` یا `-l`. آپشن‌های موجود رو می‌تونین با دستور زیر ببینین.

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

## بخش یک:

پیدا کردن یه نقطه ثابت با الگوریتم DFS

تو فایل `searchAgents.py` یه پیاده‌سازی کامل از `SearchAgent` می‌تونین ببینین که که به پک من یه برنامه می‌ده و اون رو قدم به قدم اجرا می‌کنه. حالا الگوریتم‌های جستجو رو برای برنامه‌ریزی ما باید پیاده‌سازی بکنیم. برا اطمینان از این‌که `SearchAgent` درست کار می‌کنه می‌تونین از دستور زیر استفاده کنین.

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

دستور بالا به `SearchAgent` ما می‌گه از الگوریتم `tinyMazeSearch` برای جستجو استفاده کنه. تو کد زیر هم دیده می‌شه که این الگوریتم یه لیست با ترتیب از اکشن‌هایی که پک من باید انجام بده رو برمی‌گردونه.

```
def tinyMazeSearch(problem: SearchProblem) -> List[Directions]:
    """
    Returns a sequence of moves that solves tinyMaze. For any other maze, the
    sequence of moves will be incorrect, so only use this for tinyMaze.
    """
    s = Directions.SOUTH
    w = Directions.WEST
    return [s, s, w, s, w, s, w]
```

برای این بخش ما باید تابع `depthFirstSearch` تو فایل `search.py` رو پیاده‌سازی کنیم. توضیحات داخل هر بخش رو با دقت بخونین (:

```
def depthFirstSearch(problem: SearchProblem) -> List[Directions]:
```

نکته: برا `Data Structure` های `Stack`, `Queue`, `PriorityQueue` حتما از پیاده‌سازی‌هایی که تو فایل `util.py` هست استفاده کنید.

راهنمایی: اکثر الگوریتم‌های جستجوی این پروژه شبیه هم هستند. اگر DFS رو درست پیاده سازی کنین بقیه خیلی نباید سخت باشن. فقط یادتون باشه DFS رو برای Graph Search پیاده‌سازی کنید. بعد پیاده‌سازی می‌تونین ببینین که پک من برا هزارتوهای دیگه هم می‌تونه راه‌حل پیدا کنه.

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

برای چک کردن پیاده سازی تون می‌تونین از کد زیر استفاده کنین.

```
python autograder.py -q q1
```

## بخش دو:

جستجوی عمق اول BFS

برای این بخش تابع breadthFirstSearch رو تو فایل search.py کامل می‌کنیم. فقط یادتون باشه از نسخه جستجوی گرافی استفاده کنید تا از اکسپند کردن حالت‌های دیده شده جلوگیری بشه.

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

اگه پک‌من موقع اجرا کند بود، از 0 frameTime - استفاده کنین. اگه کدتون رو جنریک پیاده‌سازی کرده باشین، کدتون باید eightpuzzle رو هم بتونه حل کنه.

```
python eightpuzzle.py
```

برای چک کردن پیاده سازی تون می‌تونین از کد زیر استفاده کنین.

```
python autograder.py -q q2
```

## بخش سه:

با اینکه BFS می‌تونه مسیرهایی که از نظر تعداد حرکت بهینه هستند را برای هزارتو پیدا کنه، ممکنه ما در نظر داشته باشیم که مسیری را انتخاب کنیم از جنبه‌های دیگه بهینه است. به همین دلیل از شما می‌خواهیم در این مرحله الگوریتم `UniformCostSearch` رو پیاده سازی کنید. برای پیاده‌سازی این الگوریتم می‌توانید از `Data Structure` های داخل فایل `util.py` استفاده کنید. پس از پیاده سازی باید ران کردن سه `command` زیر نتیجه مطلوب رو به شما بده.

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

برای تست اتوماتیک نمره خود می‌توانید از `command` زیر استفاده نمایید.

```
python autograder.py -q q3
```

## بخش چهار:

حالا که UCS رو هم پیاده‌سازی کردیم و می‌تونیم مسیر دلخواهمون رو توی هزارتوهای کوچیک به دست بیاریم، حالا می‌خوایم کارو یکم بیشتر جلو ببریم و `A*` رو پیاده سازی کنیم که بتونیم تو هزارتو های بزرگتر هم بتونیم مسیر دلخواهمون رو پیدا کنیم. برای پیاده سازی `A*` کد داخل تابع `aStarSearch` رو توی فایل `search.py` پیاده‌سازی کنید. کدی که این مرحله ران می‌کنید از هیوریستیک `manhattan` استفاده می‌کنه. بعد پیاده‌سازی با `command` زیر می‌تونید مطمئن شید کدتون کار میکنه.

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

در نهایت امتیاز این مرحلتون رو هم می‌تونید با ران کردن `command` زیر نمرتون رو ببینید.

```
python autograder.py -q q4
```

## بخش پنج:

حالا که A\* رو هم پیاده سازی کردید قراره یه heuristic متفاوت براش پیاده سازی کنید. تو این هیوریستیک قراره مسیرمون از چهار نقطه مختلف که داخل ۴ گوشه هزارتومون قرار داره بگذره. برای این کار کلاس CornerProblems رو داخل کلاس searchAgents.py پیاده سازی کنید. برای تست این قسمت هم می تونید از دستورات زیر استفاده کنید.

```
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 اینکه نمره کامل بگیری، باید یک نمایشی از وضعیت تعریف کنی که اطلاعات غیرضروری رو شامل نشه (مثل مکان ارواح یا جاهایی که غذای اضافه هست). مخصوصاً نباید از GameState مخصوص پک من به عنوان وضعیت جستجو استفاده کنی، چون این کار کد تو رو خیلی کند و اشتباه می کنه.

یک نمونه از کلاس CornersProblem کل مسئله جستجو رو نشون می ده، نه یک وضعیت خاص رو. وضعیت های خاص توسط تابع هایی که می نویسی برمی گردند، و این تابع ها باید ساختاری مثل tuple یا set که نمایانگر وضعیته، برگردونن.

همچنین وقتی برنامه در حال اجراست، به خاطر داشته باش که وضعیت های زیادی به صورت همزمان وجود دارند و همشون در صف الگوریتم جستجو هستن و باید مستقل از هم باشن. یعنی نباید فقط یک وضعیت برای کل CornersProblem داشته باشی؛ کلاس باید بتونه وضعیت های مختلفی برای الگوریتم جستجو تولید کنه.

نکته ۱: تو پیاده سازی، فقط لازمه که به موقعیت اولیه پک من و مکان چهار گوشه توجه کنی.

نکته ۲: وقتی getSuccessors رو کدنویسی می کنی، مطمئن شو که بچه ها رو با هزینه ۱ به لیست جانشین ها اضافه کنی.

در پیاده سازی ما، breadthFirstSearch حدود ۲۰۰۰ گره جستجو رو برای mediumCorners گسترش می ده. با این حال، استفاده از heuristics (در جستجوی A\*) می تونه مقدار جستجوی لازم رو کم کنه.

مثل بخشای قبل می تونید نمره تون رو برای این بخش با استفاده از تابع زیر تست کنید.

```
python autograder.py -q q5
```

## بخش شش (امتیازی):

نکته: حتماً سوال ۴ رو کامل کن قبل از اینکه روی سوال ۶ کار کنی، چون سوال ۶ بر اساس پاسخ سوال ۴ ساخته شده.

یک `heuristic` غیرساده برای `CornerProblems` در تابع `CornerHeuristic` پیاده‌سازی کن.

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

توجه: `AStarCornersAgent` یک راه میانبر برای اینه:

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

هیوریستیک غیرساده: `heuristic` های ساده همون‌هایی هستند که همه جا صفر برمی‌گردونن (مثل `UCS`) و یا `heuristics` که هزینه واقعی تکمیل رو محاسبه می‌کنه. تو باید یک `heuristics` طراحی کنی که کل زمان محاسباتی رو کاهش بده.

## توضیحات تکمیلی

- پاسخ به پروژه باید به صورت فردی انجام بشه.
- پاسخ خودتون رو به صورت زیپ شده پوشه پروژه آپلود کنین.
- فرمت نامگذاری پاسختون هم به صورت `AI4031_P1_StudentNumber.zip` هست.
- در صورت ابهام یا مشکل می‌تونین از طریق آیدی‌های تلگرام زیر، با ما در ارتباط باشین.

[t.me/your\\_ai\\_ta](https://t.me/your_ai_ta)

[t.me/kiank1382](https://t.me/kiank1382)