دانشگاه علم و صنعت
دانشکده‌ی مهندسی کامپیوتر

# تمرین سری اول

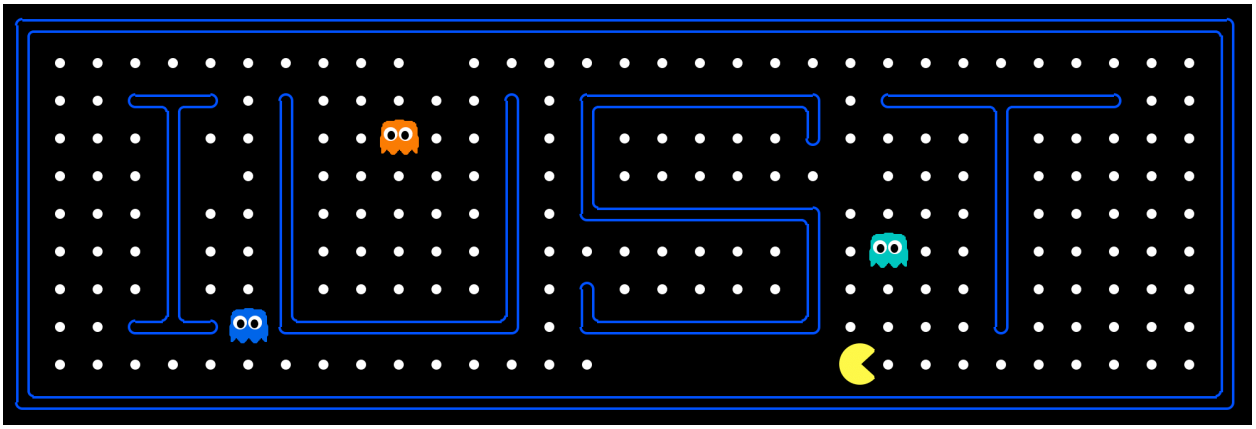
لطفا به نکات زیر توجه کنید:

- در صورتی که به اطلاعات بیشتری نیاز دارید می‌توانید به صفحه‌ی تمرین در وب‌سایت درس مراجعه کنید.
- این تمرین شامل سوال‌های برنامه‌نویسی می‌باشد، بنابراین توجه کنید که حتماً موارد خواسته‌شده در سوال را رعایت کنید. در صورتی که به هر دلیلی سامانه‌ی داوری نتواند آن را اجرا کند مسئولیت آن تنها به عهده‌ی شماست.
- ما همواره هم‌فکری و هم‌کاری را برای حل تمرین‌ها به دانشجویان توصیه می‌کنیم. اما هر فرد باید تمامی سوالات را به تنهایی تمام کند و پاسخ ارسالی حتماً باید توسط خود دانش‌جو نوشته‌شده باشد. لطفاً اگر با کسی هم‌فکری کردید نام او را ذکر کنید. در صورتی که سامانه‌ی تطبیق، تقلبی را تشخیص دهد متأسفانه هیچ مسئولیتی بر عهده‌ی گروه تمرین نخواهد بود.
- لطفاً برای ارسال پاسخ‌های خود از راهنمای موجود در صفحه‌ی تمرین استفاده کنید.
- هر سوالی درباره‌ی این تمرین را می‌توانید در گروه درس مطرح کنید و یا از دستیاران حلِ تمرین بپرسید.

- آدرس صفحه‌ی تمرین: https://iust-courses.github.io/ai982/assignments/01_search_problems
- آدرس گروه درس: https://groups.google.com/forum/ - !forum/ai982

موفق باشید

دنیای پَک‌من



فریم‌ورکی[1] که در این سری از تمرین‌ها با آن کار می‌کنید یک نسخه‌ی ساده و البته کامل از بازی معروف پک‌من است. هدف از ایجاد این چهارچوب، پیاده‌سازی و یادگیری مفاهیم و تکنیک‌های پایه در هوش مصنوعی مانند جست‌و جو در فضای حالات، یادگیری تقویتی و استنتاج احتمالی است.

قبل از اینکه به اولین سوال بپردازیم، ابتدا باید کمی با نحوه‌ی کارکرد این فریم‌ورک آشنا شویم.

۱. نحوه‌ی اجرا:

فایل زیپ را از صفحه‌ی تمرین دانلود کنید و آن را از حالت فشرده خارج کرده، سپس دستورات زیر را اجرا کنید:

```
$ cd assignment01
$ python pacman.py
```

می‌توانید زمین بازی را به نقشه‌ی دلخواهتان تغییر دهید (سایر نقشه‌ها را در پوشه‌ی layouts می‌توانید پیدا کنید):

```
$ python pacman.py --layout powerClassic
```

می‌توانید عامل[2] کنترل‌کننده‌ی پک‌من و حتی روح‌ها را هم عوض کنید:

```
$ python pacman.py --pacman GreedyAgent --ghost DirectionalGhost
```

برای مشاهده‌ی تمام قابلیت‌های بازی می‌توانید از دستور زیر استفاده کنید:

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

۲. ساختار فایل‌ها:

[1] این فریم‌ورک ابتدا در دانشگاه برکلی توسعه یافته و سپس برای این درس شخصی‌سازی شده است.

[2] Agent

نکته: این فرم‌ورک با زبان پایتون نوشته شده است. بنابراین برای انجام تمرین‌ها نیاز به کمی آشنایی با زبان پایتون دارید. در صورت نیاز می‌توانید از این‌جا استفاده کنید.

اطلاعاتی که برای انجام این تمرین نیاز دارید کاملاً در قسمت بعد آمده است بنابراین این قسمت مستقیماً مورد سوال نیست اما مطالعه‌ی آن دیدِ بهتری از ساختار فریم‌ورک به شما می‌دهد.

| **ماژول‌های اصلی، بهتر است نگاهی به آن‌ها بیندازید.** | |
|---|---|
| این فایل، نقطه‌ی شروع برنامه است و جزئیات مخصوص به بازی پک‌من مانند سیاست‌های برد و باخت، نحوه‌ی حرکت شخصیت‌های بازی و تعاملات آن‌ها با یک‌دیگر را مدل می‌کند. | paceman.py |
| موتور اصلی بازی و نحوه‌ی کنترل آن در این فایل قرار دارد. داده‌ساختارهای AgentState(وضعیت شخصیت)، Agent(شخصیت‌ها) و Grid(نقشه‌ی بازی) در آن پیاده‌سازی شده‌اند. | game.py |
| چند مورد از عامل‌های کنترل‌کننده‌ی پک‌من در این ماژول پیاده‌سازی شده‌اند. | pacmanAgents.py |
| چند مورد از عامل‌های کنترل‌کننده‌ی روح‌ها در این ماژول پیاده‌سازی شده‌اند. | ghostAgents.py |
| عامل کنترل‌کننده که دستورات‌آن از صفحه‌کلید گرفته می‌شود. | keyboardAgents.py |
| ابزارها و داده‌ساختارهای کمکی که می‌توانید در تمرین‌ها از آن‌ها استفاده کنید. | util.py |

**سایر فایل‌ها که صرفا برای پیاده‌سازی بازی هستند. می‌توانید آن‌ها را رد کنید.**

graphicDisplay.py, graphicUtils.py, layout.py, projectParams.py, test*.py

زمین بازی یک صفحه‌ی دوبعدی است که هر خانه‌ی آن یا دیوار است یا خالی و طبیعتاً تنها در صورتی که آن خانه خالی باشد می‌توان وارد آن شد. ممکن است در هر خانه‌ی زمین یک غذا و یا یک کپسول موجود باشد. همچنین همه‌ی عامل‌های بازی می‌توانند به وضعیت تمام زمین از جمله غذاها، دیوارها، کپسول‌ها و هم‌چنین محل و جهت سایر عامل‌ها دسترسی داشته باشند.

در این فریم‌ورک تقریباً تمام بازی پیاده‌سازی شده است؛ وظیفه‌ی شما تنها پیاده‌سازی یک عامل هوشمند است که کنترل شخصیت پک‌من یا یکی از روح‌ها را بر عهده می‌گیرد. کلاس Agent به همین منظور تعبیه شده است.

در هر مرحله موتور بازی وضعیت همه‌ی المان‌های بازی را محاسبه می‌کند و سپس با فراخوانی متد getAction از این کلاس و هم‌چنین پاس‌دادن وضعیت زمین به آن، حرکت بعدی عامل را درخواست می‌کند. این روند تا پایان بازی تکرار خواهد شد.

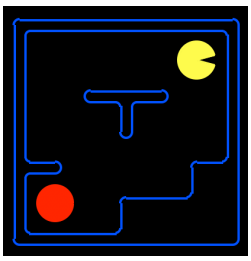
۳. حالت‌های بازی:

این فریم‌ورک دو حالت مختلف را در خود دارد. حالت اول همان پک‌من کلاسیک است که شخصیت پک‌من باید غذاهای روی زمین را بخورد و هم‌چنین از روح‌ها باید فرار کند. حالت دیگر، «حالت جست‌وجو» است به این صورت که پک‌من باید از نقطه‌ای شروع کند و به هدف مشخصی برسد. حال ممکن است این هدف صرفاً مکان خاصی در زمین باشد یا گذشتن از ۴ گوشه‌ی زمین و یا حتی خوردن همه غذاها و یا حتی همه‌ی این‌ها با هم. کاملاً می‌توان مسائل و فضای جست‌وجوی دلخواهی را برای آن تهیه کرد.

```
class SearchProblem:
    def getStartState(self)
    def isGoalState(self, state)
    def getNextStates(self, state)
    def getCostOfActions(self, actions)
```

این سری از تمرین‌ها فقط در مورد حالت دوم است. در ادامه برای حل سوالات مربوط به جست‌وجو نیاز نیست عامل را از اول پیاده‌سازی کنید. فریم‌ورک این را برای شما فراهم کرده است. کلاس SearchAgentبرای این حالت طراحی شده‌است. این کلاس دو ورودی می‌گیرد ۱- الگوریتم جست‌وجو ۲- مساله جست‌وجو. در این تمرین مسائل جست‌وجو مختلفی را خواهید دید، بعضی از آن‌ها برای شما پیاده‌سازی شده‌اند و بعضی هم به عهده‌ی شماست. هم‌چنین در سوالات ابتدایی شما چند الگوریتم جست‌وجو را نیز پیاده‌سازی خواهید کرد.

برای الگوریتم جست‌وجو، کافی است تابعی را پیاده‌سازی کنید که مساله‌ی جست‌وجو را به عنوان ورودی گرفته و دنباله‌ای از حرکاتی که پک‌من باید انجام دهد تا به هدف مساله برسد را به عنوان خروجی برگرداند. حرکت‌هایی که پک‌من می‌تواند انجام دهد شامل حرکت به سمت شمال، جنوب، شرق، غرب و یا ایست است.



```
from game import Directions

def search_algorithm(problem):
    s = Directions.SOUTH
    w = Directions.WEST
    return  [s, s, w, s, w, w, s, w]
```

همان‌طور که در مثال ساده‌ی بالا مشاهده می کنیم، این تابع برای رسیدن پک‌من به مقصد (نقطه‌ی قرمز) دنباله‌ای شامل ۸ حرکت را خروجی می‌دهد( در مثال بالا از پارامتر problemاستفاده نشده است اما در ادامه به این پارامتر نیاز خواهید داشت).

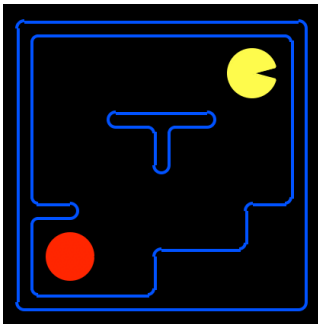
پارامتر problemمتغیری از جنس کلاس SearchProblemاست. این کلاس، کلاسی انتزاعی و عمومیست که بیان‌گر و مدل‌کننده‌ی هر نوع مساله‌ی جست‌وجو و فضای مربوط به آن است. بنابراین هر مساله‌ای باید جداگانه آن را پیاده‌سازی بکند. این کلاس به شما حالت شروع، حالت هدف (پایان)، حرکت‌های مجاز از یک حالت خاص و هم‌چنین هزینه‌ی هر دنباله‌ی دلخواهی از حرکات را برمی‌گرداند.

- getStartState(): این تابع stateشروع جست‌وجو را به شما می‌دهد؛ به طور مثال در سوالات ابتدایی که مساله جست‌وجو فقط براساس مکان است، خروجی این تابع مختصات نقطه‌ی شروع پک‌من در نقشه است.
- isGoalState(state): این تابع یک stateمی‌گیرد و اگر آن حالت هدف(مقصد) جست‌وجو باشد مقدار Trueبرمی‌گرداند و در غیر این صورت False.

- getNextStates(state): این تابع با گرفتن یک state، حالت‌های بعدی را که می‌توان با حرکات مجاز رفت، خروجی می‌دهد. هر آیتم از این لیست یک سه‌تایی است که به ترتیب: حالت جدید، حرکت لازم برای رسیدن به آن وهزینه‌ی انجام این حرکت.
- getCostOfActions(actions): این تابع لیستی از حرکت‌ها را می‌گیرد و هزینه‌ی این دنباله را حساب می‌کند. طبیعتاً همه‌ی حرکات باید مجاز باشند.

حال هر مساله‌ی جست‌وجویی با توجه به شرایطش باید این توابع را پیاده‌سازی کند. به طور مثال اگر مساله، جست و جو در گراف باشد، این کلاس باید ریشه، گره‌ی هدف، بچه‌های هر گره و هزینه‌ی حرکات را خروجی دهد (در این‌جا هزینه‌ی همه‌ی حرکات برابر یک است).

یک مساله‌ی جست‌وجوی دیگر، مساله‌ی پیداکردن یک نقطه‌ی خاص در نقشه‌ی بازی پک‌من است. این کلاس به صورت پیشفرض برای شما پیاده‌سازی شده است (کلاس PositionSearchProblem در فایل searchAgents.py) به طور مثال در این مساله، کلاس ذکر شده باید مختصات نقطه‌ی شروع، مختصات نقطه‌ی پایان و مکان‌هایی را که با هر حرکت به آن می‌رسیم خروجی دهد. در صفحه‌ی بعد عملکرد این کلاس را می‌توان مشاهده کرد، توجه کنید که شکل سمت چپ تصویری از نقشه را نشان می‌دهد .(مبدا مختصات پایین سمت چپ است)



```
def search_algorithm(problem):
    # problem is an instance of PositionSearchProblem
    print problem.getStartState()
    print problem.isGoalState((5, 5))
    print problem.isGoalState((1, 1))
    print problem.getNextStates(problem.getStartState())
    return [...]
```

خروجی

```
(5, 5)
False
True
[ ( (5, 4), 'South', 1 ), ( (4, 5), 'West', 1 ) ]
```

برای اجرای حالت جست‌وجو، از دستور زیر می‌توانید استفاده کنید:

```
$ python pacman.py -p SearchAgent -a fn=<search_fn>,prob=<search_problem> -
l=<search_map>
```

- <search_fn>: نام تابعی است که الگوریتم جست‌وجو را پیاده‌سازی می‌کند و حتماً باید در فایل searchFunctions.pyموجود باشد.
- <search_problem>: نام کلاسی است که مساله‌ی جست‌وجو را پیاده سازی می‌کند و حتماً باید در فایل searchProblems.pyموجود باشد. اگر این پارامتر را مقدار ندهید، به صورت پیش‌فرض مساله‌ی جست‌وجوی مکانی بارگذاری می‌شود.
- <search_map>: نام نقشه‌ای است که از این حالت پشتیبانی می‌کند(در هر سوال نقشه‌ی مورد نظر به شما گفته می‌شود)

به طور مثال:

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

نکته: در سوال‌های پیش‌رو مساله جست‌وجوی مورد بحث ، جست‌وجوی مکانیست (مگر خلاف آن گفته شود).

**سوال‌های عملی**

- برای اجرای فایل ها از پایتون ۲ استفاده کنید.
- خروجی توابع searchFunctions.py باید دنباله ای از حرکت ها باشد. برای درک بیشتر نگاهی به تابع tinyMazeSearch(problem) انداخته و نحوه عملکرد آن را به صورت زیر تست کنید:

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

- بازی به ازای هر مکانی در نقشه که الگوریتم شما بررسی میکند، رنگ قرمزی روی آن میکشد. هرچه رنگ قرمز روشن تر باشد یعنی این مکان زودتر بررسی شده است و هرچه تیره تر، دیرتر.
- پاسخ به یکی از دو سوال ۴ و ۵ کافی میباشد. درصورت پاسخگویی به هر دو نمره امتیازی منظور میشود.

**۱. هزارتو (۱۰ نمره)**

یکی از ساده ترین روش های یافتن مسیر خروج در یک هزارتو (Maze) گرفتن یکی از دست ها به دیوار و شروع به حرکت میباشد. اگر راه خروجی موجود باشد, بعد از طی کردن مسافتی متناهی میتوان به آن رسید. (اگر ابتدا دست راست را به دیوار گرفته باشیم تا زمانی که به خروجی میرسیم باید این کار را ادامه دهیم.) حال فرض کنید پکمن قصه ما هم در هزارتویی قرار دارد. تابع الگوریتم جستجو را طوری پیاده سازی کنید که با استفاده از این روش به نقطه ی خروج هزارتو (Goal Dot) برسد.

برای پاسخ به این سوال باید بدنه ی تابع simpleMazeSearch(problem) را در فایل searchFunctions.py کامل کنید. خروجی تابع باید دنباله ای از حرکتها باشد. برای تست کد خود میتوانید از دستورات زیر استفاده کنید:

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

```
$ python pacman.py -l rhmsMaze -p SearchAgent -a fn=simpleMazeSearch
```

آیا این روش همیشه جواب میدهد؟ در قسمت کامنت موجود در بدنه ی تابع پاسخ خود را توضیح دهید.

**۲. جستجو اول عمق - DFS (۱۰ نمره)**

همانطور که مشاهده شد روش ساده روش بهینه ای برای یافتن هدف نیست. حال وقت آن است که با الگوریتم بهتری به دنبال هدف بگردیم. درکلاس، الگوریتم جستجوی DFS را یادگرفتیم. در این سوال شما باید این الگوریتم را پیاده سازی کنید. توجه داشته باشید پیاده سازی الگوریتم باید کاملا عمومی باشد تا هر نوع مساله جستجویی که با استفاده از کلاس SearchProblem (در فایل searchProblems.py پیاده سازی شده است.) پیاده سازی شده باشد را حل کند.

برای پاسخ به این سوال باید بدنه ی تابع dfs(problem) را که در فایل searchFunctions.py قرار دارد را کامل کنید. خروجی تابع، دنباله ای از حرکتهاست. برای پیادهسازی خود میتوانید از ساختار داده های پیاده سازی شده در فایل utils استفاده کنید. همچنین درستی کد خود را با دستورات زیر تست کنید:

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

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

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

آیا روند بررسی خانه های نقشه همانیست که انتظار داشتید؟آ یا پکمن برای رسیدن به جواب، تمامی مکانها را بررسی میکند؟ (در قسمت کامنت موجود در بدنه ی تابع پاسخ خود را توضیح دهید.)
حال فرض کنید برای مساله ی جست و جو، جوابی وجود نداشته باشد. به طور مثال پکمن در مکان بسته قرار بگیرد. پاسخ پیاده سازی شما چه خواهد بود؟ برای تکمیل پیادهسازی خود، آن را طوری تغییر دهید که اگر جوابی وجود نداشت یک لیست فقط شامل اینام (Enum) STOP.Directions برگرداند. برای تست کد خود میتوانید از دستورات زیر استفاده کنید:

```
$ python pacman.py -l trappedPacman -p SearchAgent -a fn=dfs

$ python pacman.py -l unreachableGoal -p SearchAgent -a fn=dfs
```

### ۳. جستجو اول سطح - BFS (۱۰ نمره)

در این سوال باید الگوریتم BFS را پیاده سازی کنید. یکی از ویژگی های این الگوریتم تضمین رسیدن به پاسخ بهینه (از لحاظ هزینه) است. برای همین اگردر تست های زیر برنامه ی شما به پاسخ بهینه نرسید باید دوباره آن را بررسی کنید. برای پاسخ به این سوال باید بدنه ی تابع bfs(problem) را که در فایل searchFunctions قرار دارد را کامل کنید. خروجی تابع، دنباله ای از حرکتهاست. برای پیاده سازی خود میتوانید از ساختار داده های پیاده سازی شده در فایل utils استفاده کنید. درستی کد خود را با دستورات زیر تست کنید:

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

$ python pacman.py -l bigMaze -p SearchAgent -a fn= bfs -z 0.5
```

گفته شد که این الگوریتم پاسخ بهینه را تضمین میکند، اما آیا همیشه همینطور است؟ چه چیزی در این مساله باعث میشود BFS بهترین پاسخ را بیابد؟میتوانید مساله ی را مثال بزنید که نتواند بهترین پاسخ را بیابد؟ (در قسمت کامنت موجود در بدنه ی تابع پاسخ خود را توضیح دهید.)

### ۴. غذا خوردن به ترتیب (۱۰ نمره)

در سوالات قبل روشهایی برای گشتن در فضای جستجوپیاده سازی کردیم و آنها را در مساله ی PositionSearchProblem تست کردیم. از آنجایی که قرار بود پیاده سازی مان عمومی باشد بنابراین باید بتوانند در هرمساله ی جستجویی کار بکنند. در این سوال میخواهیم مساله ی دیگری

را پیاده‌سازی کنیم. مساله ی جستجو این است که پکمن باید قبل از رسیدن به مقصد مشخصش، ابتدا تمامی غذاهای روی زمین را بخورد با این شرط که روش حرکت پکمن پاد ساعتگرد باشد.

بنابراین برای پیاده سازی این مساله، شما باید کلاسی بنویسید که از SearchProblem ارث بری کرده باشد و متدهای آن را با توجه به این مساله ی خاص پر شده باشد. برای راحتی کار بهتر است ابتدا فضای حالت را برای این مساله ی خاص در نظربگیرد و جزئیات آن را به دست آورید. توجه کنید فضای حالتی که در نظرمیگیرید نباید اطلاعات غیر ضروری در خود داشته باشد چراکه باعث میشود اندازه ی فضای حالات شما بیهوده بزرگ شود.

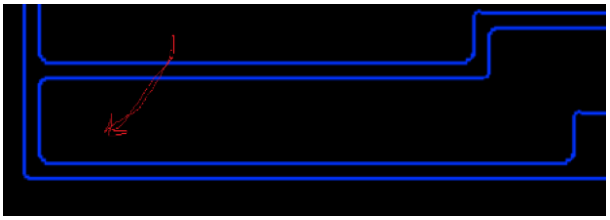
برای پاسخ به این سوال شما باید کلاس CounterClockwiseFoodProblem که در فایل psearchProblems قرار داد را کامل کنید. برای تست برنامه خود میتوانید از دستورات زیر استفاده کنید:

```
$ python pacman.py -l openSearch -p SearchAgent -a fn=bfs,prob=
CounterClockwiseFoodProblem


$ python pacman.py -l mediumCFoodMaze -p SearchAgent -a fn=bfs,prob=
CounterClockwiseFoodProblem

$ python pacman.py -l bigCFoodMaze -p SearchAgent -a fn=bfs,prob=
CounterClockwiseFoodProblem  -z 0.5
```

راهنمایی: برای مثال میتوانید کلاس PositionSearchProblem را مرور کنید. پیاده‌سازی شما در این سوال، بیشتر در توابع getNextStates و isGoalState تفاوت خواهد داشت.

**۵. خوردیم به بن بست ! (۱۰ نمره)**

در این سوال عامل پکمن باید ابتدا در کمترین زمان تمامی بن بست ها را پیدا کرده و سپس به سمت هدف حرکت کند.



برای پاسخ به این سوال باید بدنه ی تابع deadend(problem) را که در فایل py.searchFunctions قرار دارد را کامل کنید. خروجی تابع، دنباله ای از حرکت‌ها است. برای پیاده سازی خود میتوانید از ساختار داده های پیاده سازی شده در فایل utils استفاده کنید. درستی کد خود را با دستورات زیر تست کنید.

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

$ python pacman.py -l bigMaze -p SearchAgent -a fn= deadend -z 0.5
```

### ۶. مناطق سمی – cost_function (۱۰ نمره)

در الگوریتم هایی که تا به اینجا پیاده سازی کردید هزینه حرکت از یک خانه به خانه در نظر گرفته نشده بود. حال در این سوال قصد داریم عامل پکمن جدیدی را پیاده سازی کنیم که از مبدا به مقصد در کمترین زمان ممکن رسیده و همچنین از مناطقی که روح ها در آنها هستند (منطقه سمی) دوری کند. توجه کنید که منطقه ی سمی، مربعی ۳در۳ است که یک روح در مرکز آن قرار دارد.

برای اعمال این شرایط تغییر جزئیات فضای حالت و یا هدف مساله کمکی به ما نمیکند. اما اگر هزینه ی ورود به مناطق سمی را بیشتر از حالتهای دیگر در نظر بگیریم میتوانیم الگوریتم جستجویی طراحی کنیم که مسیرهای بهینه تری را انتخاب کند.

بنابراین فقط کافیست نحوه ی محاسبه ی هزینه را طوری به روزرسانی کنیم که هزینه ی ورود به مناطق سمی بیشتر از مناطق امن باشد.

برای سوال بعدی میخواهیم جستجوی هزینه یکنواخت (ucs) را پیاده سازی کنیم و از کلاس ScaryProblem (در فایل searchProblems) استفاده کنیم که تابع محاسبه هزینه در آن پیاده سازی نشده است.

تابع cost_function در کلاس ScaryProblem را به شکلی کامل کنید که هزینه وارد شدن به مناطق سمی زیاد شود. درستی برنامه خود را میتوانید پس از حل سوال بعدی تست کنید.

مطمئن شوید پیاده‌سازی شما از برخورد با روحها جلوگیری میکند زیرا در صورت برخورد با آنها میبازید. در صورت نیاز میتوانید تابع getNextStates را نیزتغییر دهید.

### ۷. هزینه یکنواخت - UCS (۱۰ نمره)

در این سوال پکمن تنها عامل بازی نیست و روح های ساکنی نیز در نقشه بازی حضور دارند.(مناطق سمی که حرکت یه سمت آنها هزینه داشته و در سوال قبل پیاده سازی شده است.) همانطورکه میدانید bfs هیچ اطلاعی از هزینه ی حرکتها ندارد بنابراین نمیتواند کمکی به ما بکند. اما الگوریتم Search-Cost-Unifrom میتواند این قابلیت را برای ما فراهم کند. در این سوال، این الگوریتم را پیاده‌سازی خواهیم کرد. برای پاسخ به این سوال باید بدنه‌ی تابع problem(ucs) را در فایل searchFunctions.py پرکنید. خروجی تابع، دنباله هایی از حرکت ها است.

برای پاسخ به این سوال باید بدنه ی تابع ucs(problem) را که در فایل searchFunctions قرار دارد را کامل کنید. خروجی تابع، دنباله ای از حرکت ها است. برای پیاده‌سازی خود میتوانید از ساختار داده های پیاده سازی شده در فایل utils استفاده کنید. درستی کد خود را با دستورات زیر تست کنید:

```
$python pacman.py -l dangMaze -p SearchAgent -a fn=ucs, prob= ScaryProblem

$python pacman.py -l bigDangMaze -p SearchAgent -a fn=ucs, prob= ScaryProblem -
z 0.5
```

در چه صورتی نتیجه الگوریتم های bfs و ucs یکی خواهد شد؟ (در قسمت کامنت موجود در بدنه ی تابع پاسخ خود را توضیح دهید.)

## سوال‌های تئوری

### ۱. جست و جوی ۲ طرفه(۵ نمره)

الگوریتم جست و جوی ۲ طرفه را به کمک یک مثال شرح دهید و با دلیل بگویید آیا این الگوریتم همواره بهترین جواب را می دهد یا خیر .

### ۲. پیمایش گراف(۱۰نمره)

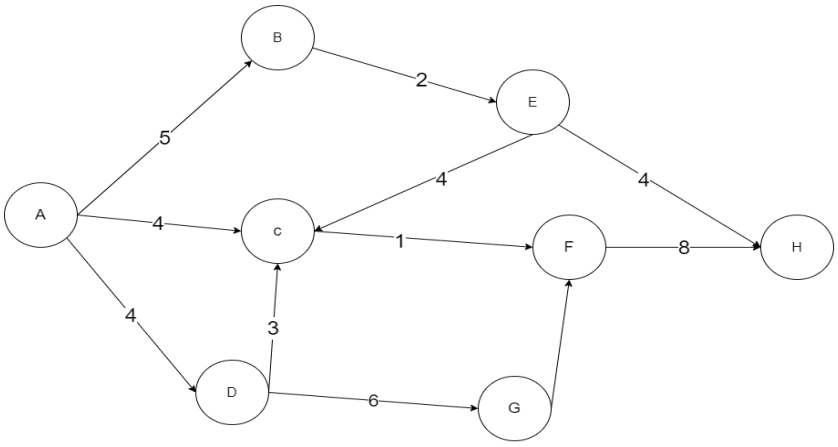
در گراف زیر جستجو از گره A شروع میشود و با رسیدن به گره پایانی H تمام میشود. با استفاده از هر یک از الگوریتم های ذکر شده گراف را پیمایش کنید. برای هر یک از الگوریتم ها موارد زیر را بنویسید.

۱. ترتیب گره هایی که بعد از پایان جستجو برای رسیدن به هدف طی خواهیم کرد.

۲. ترتیب گره هایی که در مراحل الگوریتم مشاهده میشوند.

۳. در هر مرحله از اجرا الگوریتم محتویات فرینج را نمایش دهید و مشخص کنید چه گره‌ای در مرحله بعد بسط داده میشود.

- DFS
- BFS
- UCS

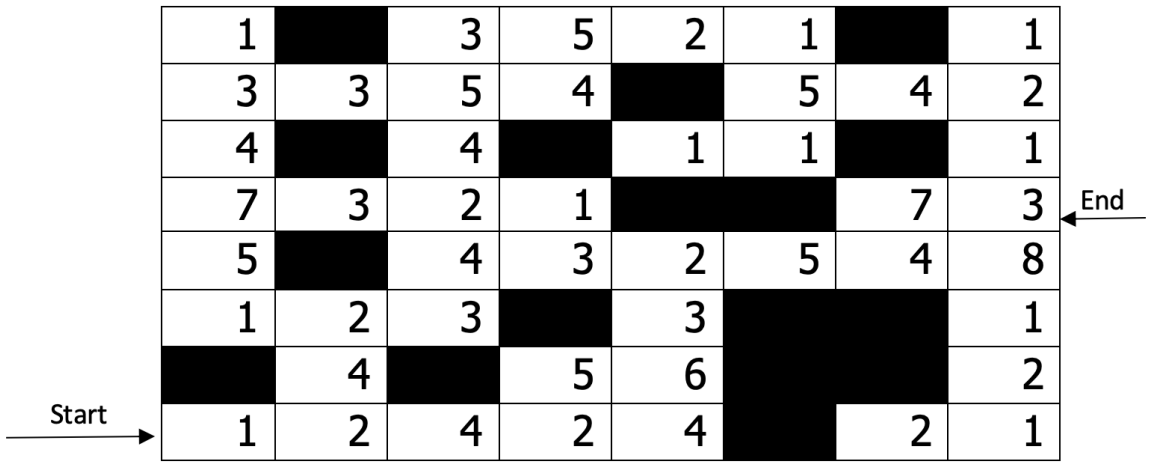



### ۳. مدل سازی(۵ نمره)

حال اپلیکشن waze را در نظر بگیرید و آن را مدل سازی کنید . به فرض مثال محیط و عوامل موثر بر آن را پیدا کنید و جستوجوی اول عمق را صرفا روی آن توضیح دهید.

**۴.فرار از ماز(۱۰نمره):**

با استفاده از الگوریتم A Star راه بهینه خروج از ماز زیر را پیدا کنید.(در هر خانه مقدار پتانسیل را بنویسید)




**۵. متغیر های محدود کننده(۱۰نمره):**

CSP با ۱۵ متغیر $X_1, X_2, \ldots, X_{15}$ و $D = \{۱,۲,۳,۴,۵\}$ را در نظر بگیرید.

محدودیت‌های زیر در بین متغیرها برقرار است:

$X_i > X_{2i}, X_i > X_{2i+1}, \forall i = 1, \ldots n$.

-گراف محدودیت را رسم کنید.

-متغیرها را با اعمال binary arc consistency هرس کرده و روی گراف مشخص کنید.

-با استفاده از گراف محدودیت، راه‌حلی با استفاده از backtracking پیدا کنید. مراحل طی شده و فرضیات خود برای انتخاب متغیرها و مقادیر را بنویسید.