۰)شناخت فایلها و برخی از function های پروژه

سوال: کاربرد کلاس SearchProblem در فایل search.py را به همراه متودهای آن توضیح دهید.

همچنین به اختصار کاربرد هر یک از کلاسهایActions, Configuration, Directions, Agent

Grid, AgentState را که در فایل game.py قرار دارند، بیان کنید.

کلاس SearchProbelm، یک کلاس انتزاعی است که چارچوب یک مسئله جستجو را مشخص میکند:

هر مسئله ی جستجو یک حالت اولیه (getStartState) دارد که حالت پکمن و روح ها و دیوار ها و غذا ها را در لحظه ی ابتدای بازی را برمیگرداند.

متود isGoalState با گرفتن حالت، به ما میگوید که ایا حالت هدف هست یا نه. این متود نیز در هر مسئله با توجه به نوع مسئله تعریف میشود.

متود getSuccessors با گرفتن هر حالت، به ما تابع تغییر و حالت های ممکن بعدی را میدهد به همراه هزینه ی هر حرکت. این متود نیز در مسائل جستجوی مختلف، میتواند پیاده سازی های مختلفی داشته باشد.

متود getCostOfActions با گرفتن دنباله ای از حرکات، هزینه ی نهایی انجام این حرکات را به ما میدهد.

کلاس Agent: عامل با توجه به حالتی که در آن قرار دارد، یک اکشن از نوع Directions برمیگرداند.

کلاس Directions: مشخص میکند که هر اکشنی که به عامل داده شود، عامل به کدام جهت باید حرکت کند.

کلاس Configuration: موقعیت و جهت حرکت یک عامل را نگهداری میکند. همچنین میتواند configuration بعدی را با توجه به موقعیت و جهت فعلی تولید کند.

کلاس Actions: حرکت ها را تعریف میکند و همچنین اطلاعاتی مربوط به حرکات ممکن تولید میکند.

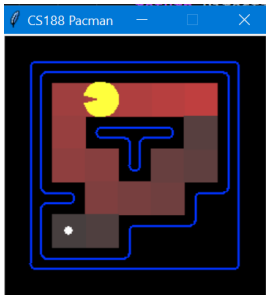
کلاس AgentState: حالت یک عامل را نگهداری میکند. شامل اطلاعاتی مثل سرعت، موقعیت، جهت، ترسیده بودن است.

کلاس Grid: یک ساختمان داده متشکل از یک لیست از لیست ها که برای نگهداری مکان دیوار ها و غذاها استفاده میشود.

Helia Akbari – 40031005 – Project 1

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

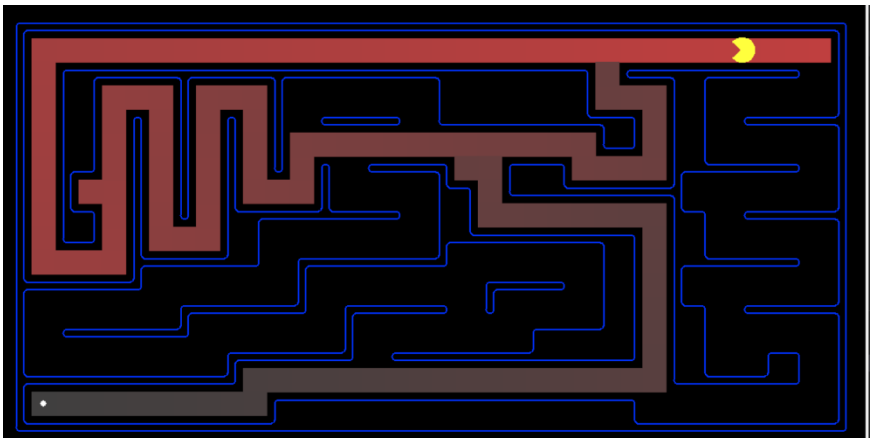
python pacman.py -l tinyMaze -p SearchAgent




```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 500
Average Score: 500.0
Scores:        500.0
Win Rate:      1/1 (1.00)
Record:        Win
PS C:\Helia\school books\term4\AI_P1\P1>
```

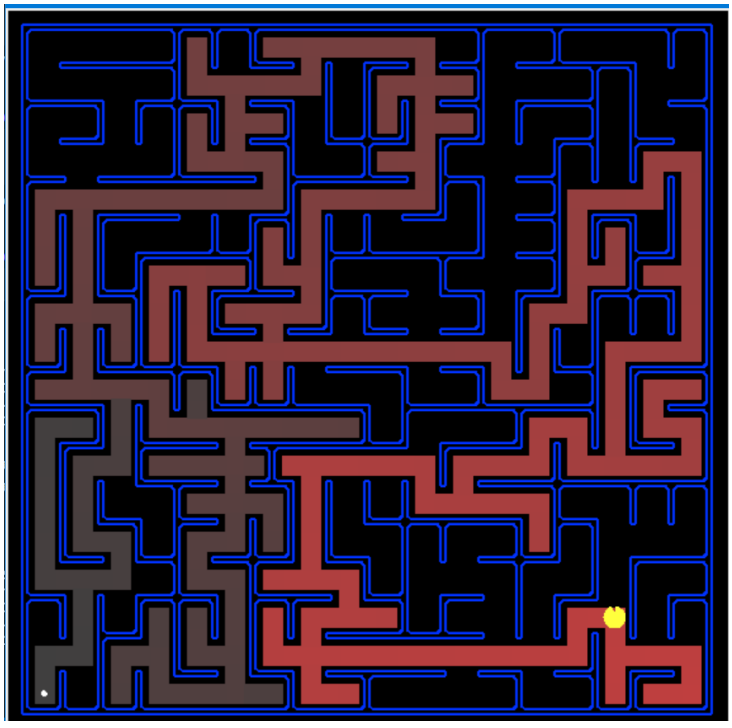
python pacman.py -l mediumMaze -p SearchAgent

```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 130 in 0.0 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:        380.0
Win Rate:      1/1 (1.00)
Record:        Win
```



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

```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```



سوال: راه حل DFS را از نظر پیچیدگی زمانی و فضایی تحلیل کنید. طبق چیزی که از حرکت عامل

مشاهده کردید به سوال زیر پاسخ دهید.

از لحاظ پیچیدگی فضایی، همسایه ی حالت هایی که از جایی که هست تا حالت اولیه را نگهمیدارد. این میتواند به اندازه ی عمق خانه ی هدف باشد.. همچنین ممکن است قبل از رسیدن به جواب، تمام خانه های دیگر را چک کرده باشیم پس پیچیدگی زمانی میتواند از اوردر تعداد خانه های بدون دیوار باشد.

آیا استفاده از الگوریتم DFS بهینه است و عامل رفتار منطقی از خود نشان میدهد؟ توضیح دهید.

خیر،عامل به کوتاه بودن راه توجه ای ندارد. به اینکه به فرزند خانه ای که در آن قرار دارد، برود، اولویت میدهد.

سوال: در غالب یک شبه کد مختصر الگوریتم IDS را توضیح دهید و حالتی را شرح دهید که این الگوریتم

عملکرد بدتری نسبت به DFS دارد.

شبه کد IDS:

```
IDS(root, goal, depthLimit){
   for depth = 0 to depthLimit
      if (DFS(root, goal, depth))
         return true
   return false
}

DFS(root, depth){
   if root == goal
      return true
   if depth == 0
      return false
   for each child in root.children
      if (DFS(child, goal, depth - 1))
         return true
   return false
}
```

IDS در واقع الگوریتم DFS را اول تا عمق یک، سپس دو و الی اخر اجرا میکند به طوری که در هر بار، گره ها بیشتر از عمق مشخص شده، بررسی نمیشوند.

اگر هدف در عمق زیادی باشد، درحالیکه در دی اف اس، نسبتا زود پیدا شود، دی اف اس عملکرد بهتری خواهد داشت.

۲)جستجوی اول سطح (BFS)

سوال: دستور زیر را اجرا کنید. آیا کد شما بدون هیچ گونه تغییری مسئله ۸-پازل را حل میکند؟ توضیح

دهید. python eightpuzzle.py

با تعریف حالت هدف جدید و حالت های جدید، از همان الگوریتمی که نوشتیم میتوان استفاده کرد. در کد داریم:

```
problem = EightPuzzleSearchProblem(puzzle)
path = search.breadthFirstSearch(problem)
```

سوال: الگوریتم BBFS را به صورت مختصر با نوشتن یک شبه کد ساده توضیح دهید و آن را با الگوریتم

BFS مقایسه کنید.

در واقع در BBFS دو BFS از ریشه و هدف شروع میکنیم و هنگامی که یک گره پیدا کردیم که توسط هر دو بسط داده شده بود، مسیر را از ریشه به آن گره و از آن گره به هدف قرار میدهیم. اینطوری گره های کمتری بسط پیدا میکنند نسبت

به بی اف اس معمولی و سریع تر نیز میباشد. اما bbfs راحت تر از bfs همچنین برای اینکه این الگوریتم کار کند، درخت باید دو جهته باشد.

سوال: الگوریتم BFS را از نظر پیچیدگی زمانی و فضایی با الگوریتم DFS مقایسه کنید و مطابق آنچه در

این دو بخش از پروژه مشاهده کردید، بیان کنید هریک از الگوریتمهای مذکور در چه مواردی عملکرد

بهتری در حرکت عامل دارد

اوردر زمانی میشود ۴ به توان عمق خانه ی هدف. از لحاظ فضایی همه ی خانه هایی که با خانه ی هدف در یک عمق قرار دارند در فرینج قرار دارد پس حدود ۴ به توان عمق خانه ی هدف، میتواند پیچیدگی فضایی ما باشد. اگر فضای محدود داشته باشیم، بهتر است از dfs استفاده کنیم اما اگر کوتاه ترین مسیر مهم باشد، از bfs بهتر است استفاده کنیم.

۳)تغییر تابع هزینه

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 پیاده سازی کد جدیدی نیست؛ صرفا تغییراتی را که باید به کد

خود اعمال کنید را ذکر نمایید).

اگر به همه ی حرکات یکی ای، هزینه ی یکسان بدهیم، یعنی successor actionCost یکسان باشد، مسئله bfs میشود. اگر اولویت هر حرکت را از یک عدد بسیار بزرگ شروع کنیم و هر بار مقداری کم کنیم، میتوانیم DFS را پیاده کنیم.

سوال: آیا الگوریتم UCS نسبت به دو الگوریتم ناآگاهانه دیگر برتری دارد؟ مزایا و معایب آن را بیان کنید.

UCS بهینه است در حالی که DFS چنین نیست. همچنین هنگامی که حرکات هزینه های متفاوتی دارند، UCS جواب بهینه را میدهد درحالی که BFS جواب بهینه را نمیدهد. اما UCS فضای بیشتری از دو الگوریتم دیگر میگیرد همچنین زمان بیشتری برای محاسبه میگیرد چون هر بار باید فرینج را برای کوچک ترین هزینه جستجو کرد.

۴) جستجوی A استار (A*)

سوال: الگوریتمهای جستجویی که تا به این مرحله پیاده سازی کرده اید را روی openMaze اجرا کنید.

و توضیح دهید چه اتفاقی میافتد (تفاوتها را شرح دهید).

```
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
PS C:\Helia\school books\term4\AI_P1\P1> python pacman.

[SearchAgent] using function dfs
[SearchAgent] using problem type PositionSearchProblem
Pacman emerges victorious! Score: 212
Average Score: 212.0
Scores:        212.0
Win Rate:      1/1 (1.00)
Record:        Win
PS C:\Helia\school books\term4\AI_P1\P1> python pacman.

[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.0 seconds
Search nodes expanded: 682
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```

UCS, BFS مانند هم حرکت کرد و مسیر نسبتا مستقیمی رو رفت اما با DFS همونطور که مشخصه، یک مسیر طولانی و مارپیچی را رفت.

سوال: ایده ی اصلی الگوریتم A* را با الگوریتم Dijkestra مقایسه کنید.

در دایجسترا، در اول مسئله تمام گره ها را داریم درحالی که در آستار، به مرور زمان و هنگام بسط دادن، گره ها اضافه میشوند.

۶)هیوریستیک برای مسئله گوشه ها

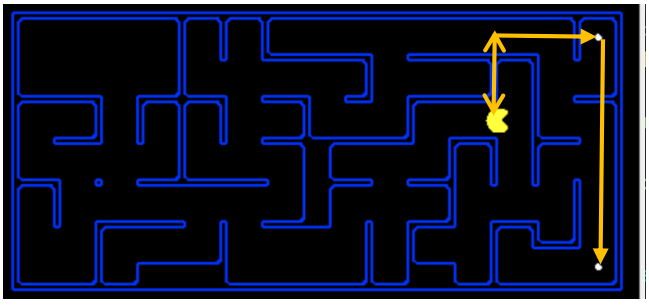
سوال: هیوریستیک خود را توضیح دهید و سازگاری آن را استدلال کنید.

هیوریستیک گوشه ها برای من به این صورت است که فرض میکنیم هیچ دیواری وجود ندارد. حال کوتاه ترین مسیر از مکان کنونی تا رسیدن به همه ی گوشه های دیده نشده به ترتیب های مختلف و انتخاب کوتاه ترین را به عنوان هیوریستیک درنظر میگیریم.

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.

فرض میکنیم تعداد حرکات مسیر X برابر با هیوریستیک ما باشد حال در حرکت بعدی، اگر همان مسیر X هیوریستیک ما باشد، اگر در مسیر X حرکت کرده باشیم، هیوریستیک یکی کمتر میشود و اگر در مسیر دیگری باشد هیوریستیک یا ثابت خواهد بود یا بعلاوه یک میشود. اگر هم مسیر هیوریستیک عوض شده باشد، از انجایی که در موقعیت پیشین مینیموم نبود، حداقل به اندازه ی هیوریستیک X بوده پس اینجا چون یک حرکت انجام شده، کمترین مقدار ممکنش X-1 خواهد بود پس همچنان سازگار است.

مثال یک هیوریستیک: طول مسیر زرد



۷)خوردن همه غذاها

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

برای هر غذا فاصله ی آن از موقعیت کنونی را با bfs محاسبه کرده و فاصله ی دورترین غذا را هیوریستیک قرار میدهیم. اینطوری وقتی از یه حالت به حالت دیگر میرویم، اگر دورترین غذا همان قبلی باشد، حداکثر یکی از هیوریستیک کم شده(در صورتی که حرکت به سمت کوچک ترین غذا بوده باشد. اگر در بین دو حالت، دور ترین غذا تغییر کرده باشد، این یعنی اگر فاصله ی دورترین غذا در حالت اول x بوده، الان فاصله از آن حداقل x-1 است پس در حالت جدید فاصله ی غذای جدید باید بزرگتر یا مساوی x-1 باشد. پس هیوریستیک سازگار است چون تغییر کوچک تر مساوی یک است.

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

در بخش قبلی از فاصله منهتن استفاده شد اما اینجا از حل مسائل bfs هیوریستیک به دست امد.

سوال: باتوجه به کدتان و مطالب درس بیان کنید که چرا به A* الگوریتم جستوجوی آگاهانه میگویند؟

چون اطلاعات اضافه ای راجع به تخمین فاصله با هدف دارد که کمک میکند، گره های درست بسط داده شوند و راه حل بهینه شود.

۸)جستجوی نیمه بهینه

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.

به عنوان مثال، در موقعیت روبه رو برای کوتاه ترین مسیر اول باید غذای بالا خورده شود و سپس غذای پایینی اما با خوردن کوچکترین نقطه، ابتدا بالایی خورده میشود و سپس پایینی