«باسمه تعالی»

درس مبانی و کاربردهای هوش مصنوعی

گزارش پرژه اول

طراحی و تدوین:

مهدی رحمانی

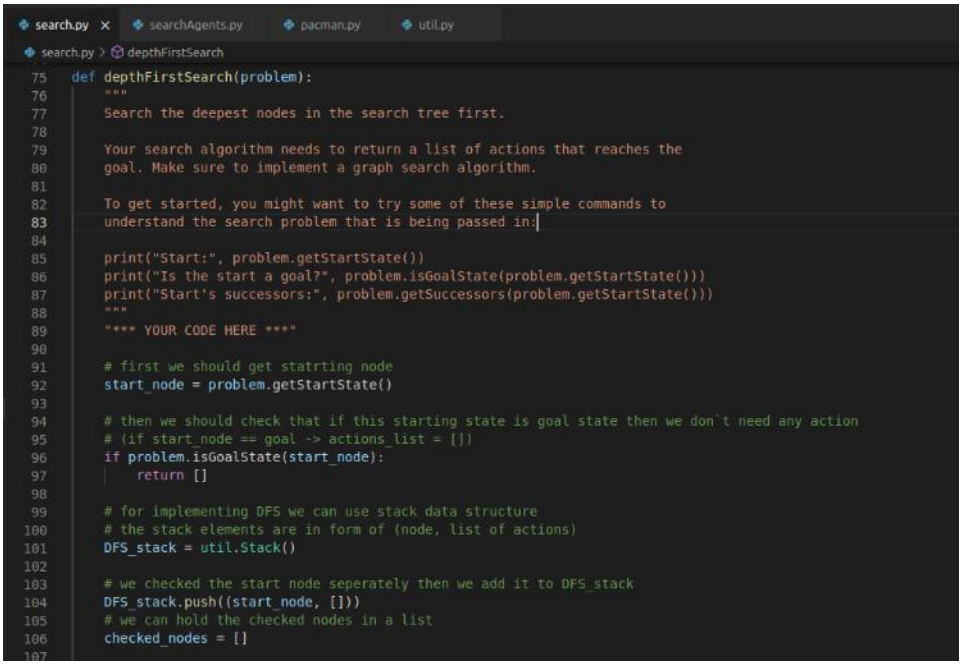
۹۷۳۱۷۰۱

بخش اول

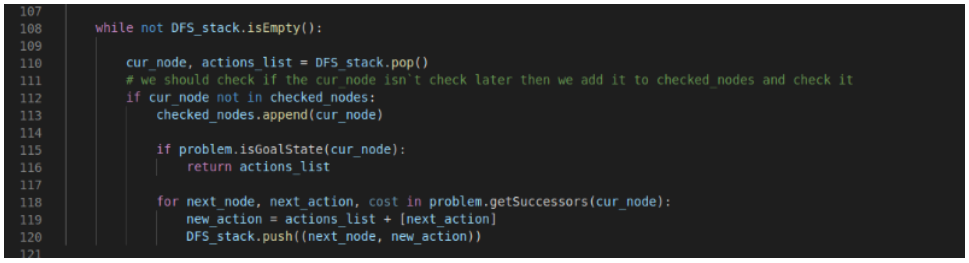
پیدا کردن یک نقطه ثابت با استفاده از جست و جوی اول عمق

کد نوشته شده

در فایل search.py میتوان کد مربوط به آن را نوشت:



```python
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    print("Start's successors:", problem.getSuccessors(problem.getStartState()))
    """
    "*** YOUR CODE HERE ***"

    # first we should get statrting node
    start_node = problem.getStartState()

    # then we should check that if this starting state is goal state then we don't need any action
    # (if start_node == goal -> actions_list = [])
    if problem.isGoalState(start_node):
        return []

    # for implementing DFS we can use stack data structure
    # the stack elements are in form of (node, list of actions)
    DFS_stack = util.Stack()

    # we checked the start node seperately then we add it to DFS_stack
    DFS_stack.push((start_node, []))
    # we can hold the checked nodes in a list
    checked_nodes = []
```



```python
    while not DFS_stack.isEmpty():

        cur_node, actions_list = DFS_stack.pop()
        # we should check if the cur node isn't check later then we add it to checked_nodes and check it
        if cur_node not in checked_nodes:
            checked_nodes.append(cur_node)

            if problem.isGoalState(cur_node):
                return actions_list

            for next_node, next_action, cost in problem.getSuccessors(cur_node):
                new_action = actions_list + [next_action]
                DFS_stack.push((next_node, new_action))
```

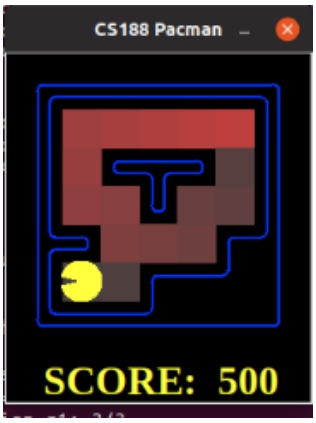
برای implement کردن الگوریتم DFS از stack استفاده کردم که در util.py پیاده سازی شده است.

همچنین یک لیست به نام checked_nodes داریم تا حالاتی که قبلا دیده شده اند را در آن نگه داریم تا در چک کنیم که اگر نودی در این لیست بود دیگر آن را گسترش ندهیم.

نمونه‌هایی از اجرای کد:

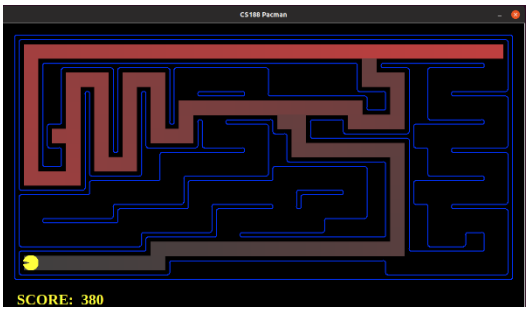
Python3 pacman.py -l tinyMaze -p SearchAgent

```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 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
```



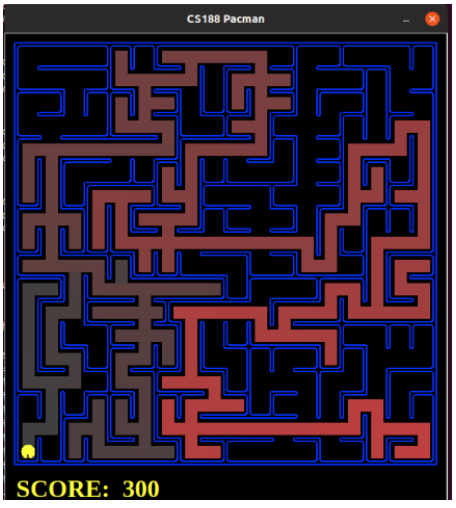
python3 pacman.py -l mediumMaze -p SearchAgent

```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 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
mahdi@OSlab:~/Desktop/AI_P1/search$
```



python3 pacman.py -l bigMaze -z .5 -p SearchAgent

```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 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
```



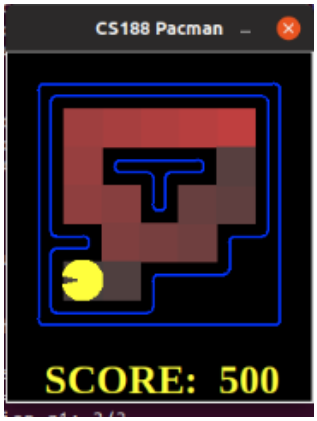
سوال) آیا ترتیب کاوش همان ترتیبی بود که انتظار داشتید؟ آیا پکمن در راه رسیدن به هدف، به همه مربع‌های کاوش شده میرود؟

بله باتوجه به کدی که نوشته بودیم همین انتظار را داشتیم. چرا که ما در پیاده سازی fringe از پشته استفاده کردیم که به صورت LIFO میباشد. یعنی میدانیم که میتوان به چپ یا راست یا بالا و پایین رفت.حال اگر فرض کنیم در یک مکان نودهای مختلفی که در ۴ طرف نود فعلی هستن داخل فرینج قرار بگیرند ما براساس DFS چپ ترین گره را برمیداریم و ادامه میدهیم. اگر تابع getSuccessor را ببینیم ترتیب وارد شدن استیت‌ها بالا و پایین و راست و چپ است. پس آخرین استیت سمت چپ است و به همین دلیل در استک هم آخرین استیت push شده اول pop میشود. اینجا هم به همین دلیل قرمز پر رنگ تر که سمت چپ استیت اولیه‌ی پکمن است زودتر کاوش شده است.

خیر- به همه مربع های کاوش شده نمیرود. برخی مربع ها کاوش شده اند ولی در مسیر نهایی که pacman از آن میرود نیست.

سوال) آیا این راه حل کمترین هزینه را دارد؟ اگر نه فکر کنید که جستجوی اول عمق چه کاری را اشتباه انجام میدهد.

خیر- اگر برای مثال در شکل زیر هم نگاه کنیم مسیری که از پایین به مقصد میرسد بهینه تر است. این جست و جو یک جست و جوی نا آگاهانه هست و به این صورت کار میکند که گره با عمق بیشتر براش اولویت بیشتری دارد و ممکن است با توجه به اولویتش همینطوری تا انتهای یک مسیر را بررسی کند ولی در انتها به بن بست برسد. پس در عمق بیشتر میرود تا به هدف برسد ولی لزوما مسیری که برمیگرداند بهترین مسیر نیست.

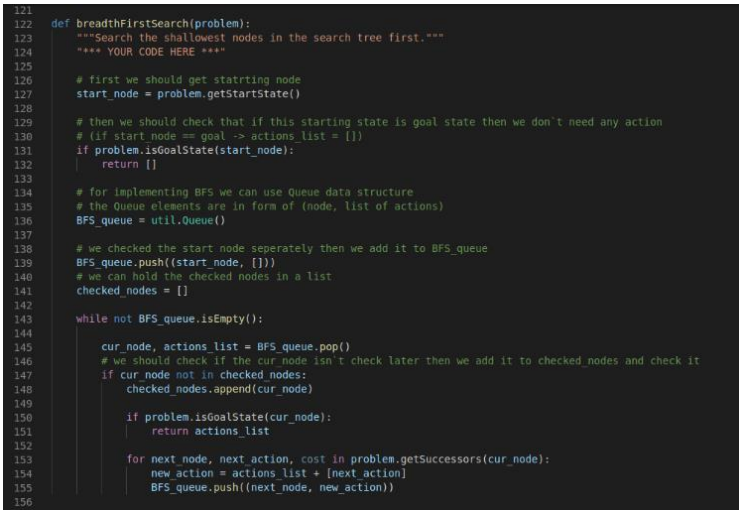


بخش دوم

جست و جوی اول سطح

کد نوشته شده

در فایل search.py لازم است تا تابع breadthFirstSearch را بنویسیم:



```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"

    # first we should get statrting node
    start_node = problem.getStartState()

    # then we should check that if this starting state is goal state then we don't need any action
    # (if start_node == goal -> actions_list = [])
    if problem.isGoalState(start_node):
        return []

    # for implementing BFS we can use Queue data structure
    # the Queue elements are in form of (node, list of actions)
    BFS_queue = util.Queue()

    # we checked the start node seperately then we add it to BFS_queue
    BFS_queue.push((start_node, []))
    # we can hold the checked nodes in a list
    checked_nodes = []

    while not BFS_queue.isEmpty():

        cur_node, actions_list = BFS_queue.pop()
        # we should check if the cur_node isn't check later then we add it to checked_nodes and check it
        if cur_node not in checked_nodes:
            checked_nodes.append(cur_node)

            if problem.isGoalState(cur_node):
                return actions_list

            for next_node, next_action, cost in problem.getSuccessors(cur_node):
                new_action = actions_list + [next_action]
                BFS_queue.push((next_node, new_action))
```

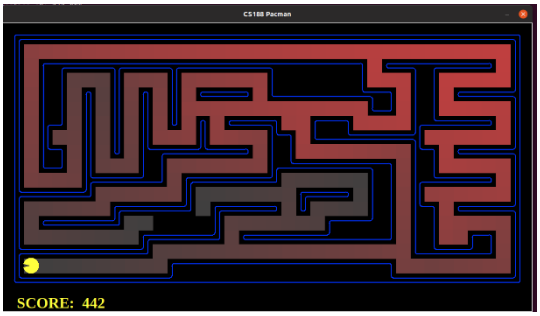
این تابع نیز خیلی شبیه به DFS پیاده سازی میشود منتها لازم است تا از ساختمان داده Queue به جای پشته استفاده کنیم. برای استفاده از صف میتوان از صف پیاده سازی شده در utils.py کمک گرفت. چون صف به صورت FIFO میباشد درواقع نودهایی که زودتر در fringe قرار گرفتند زودتر بررسی میشوند. به این ترتیب اولویت ما پیشروی در عرض میباشد به جای عمق و درواقع نودهای هم عمق ابتدا باید بررسی شوند و بعد سراغ عمق بعدی برویم.

بنابراین با این تغییر کوچک از کد مربوط به DFS میتوان BFS را پیاده سازی کرد.

نمونه‌هایی از اجرای کد:

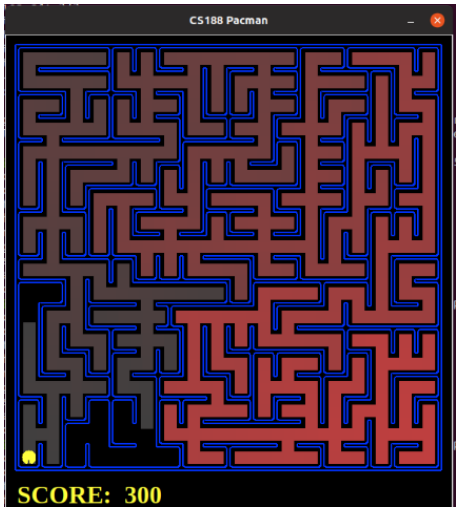
python3 pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.1 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```



python3 pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

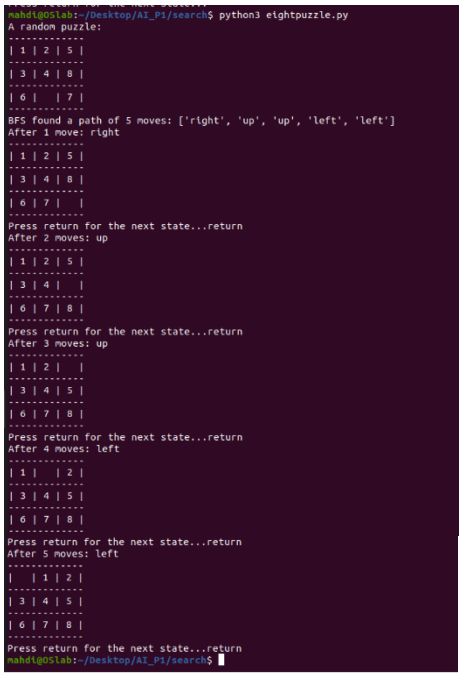
```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
mahdi@OSlab:~/Desktop/AI_P1/search$
```



سوال) آیا الگوریتم جستجوی اول سطح راه‌حل با کمترین هزینه را پیدا میکند؟ اگر نه پیاده‌سازی خود را چک کنید.

بله- چون هزینه برابر ۱ میباشد بنابراین در این روش الگوریتم bfs مسیر بهینه را برمیگرداند.

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.



بخش سوم

تغییر تابع هزینه

کد نوشته شده

```python
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    # first we should get statrting node
    start_node = problem.getStartState()
    # then we should check that if this starting state is goal state then we don't need any action
    # (if start_node == goal -> actions_list = [])
    if problem.isGoalState(start_node):
        return []

    # for implementing BFS we can use Queue data structure
    # the Queue elements are in form of ((node, list of actions to cur_node, total cost to cur_node), priority = total cost to cur_node)
    UCS_priority_queue = util.PriorityQueue()

    # we checked the start_node seperately then we add it to UCS_priority_queue
    UCS_priority_queue.push((start_node, [], 0), 0)
    # we can hold the checked nodes in a list
    checked_nodes = []

    while not UCS_priority_queue.isEmpty():

        cur_node, actions_list, prev_cost = UCS_priority_queue.pop()
        # we should check if the cur_node isn't check later then we add it to checked_nodes and check it
        if cur_node not in checked_nodes:
            checked_nodes.append(cur_node)

            if problem.isGoalState(cur_node):
                return actions_list

            for next_node, next_action, cost in problem.getSuccessors(cur_node):
                new_cost = prev_cost + cost
                new_action = actions_list + [next_action]
                UCS_priority_queue.push((next_node, new_action, new_cost),new_cost)

```

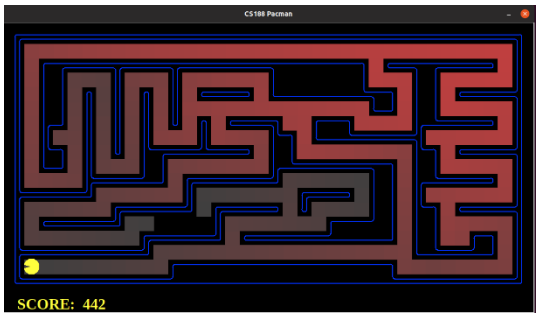
الگوریتمی که پیاده سازی کردیم بسیار شبیه به دو الگوریتم قبلی میباشد ولی تفاوت‌های کمی دارد. اولین تفاوت این هست که برای پیاده سازی و نگهداری fringe از ساختمان داده priority queue استفاده کردیم. این ساختمان داده در util.py پیاده سازی شده و ما فقط اینجا از آن استفاده میکنیم. اولویت در اینجا هزینه تجمعی میباشد. در UCS به این صورت عمل میکنیم که نودهای ارزان‌تر باید زودتر expand شوند.

در الگوریتم‌های قبلی هزینه را در نظر نمیگرفتیم. در اینجا وقتی problem.getSuccessors را صدا میزنیم به ما هزینه را هم برمیگرداند که لازم است با هزینه‌هایی که تا این نود داشتیم جمع کنیم و آن را به همراه نود و action بعدی در صف اولویت قرار دهیم.

نمونه‌هایی از اجرای کد:

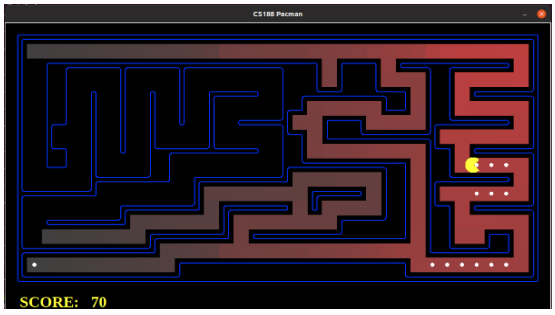
python3 pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

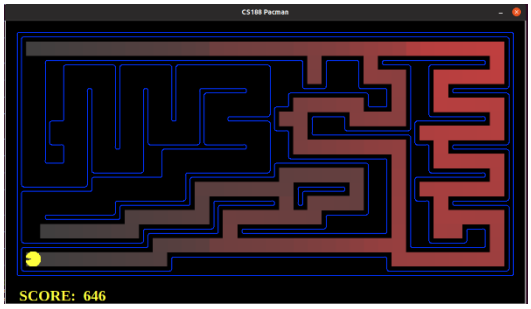
```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.1 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
mahdi@OSlab:~/Desktop/AI_P1/search$
```



python3 pacman.py -l mediumDottedMaze -p StayEastSearchAgent

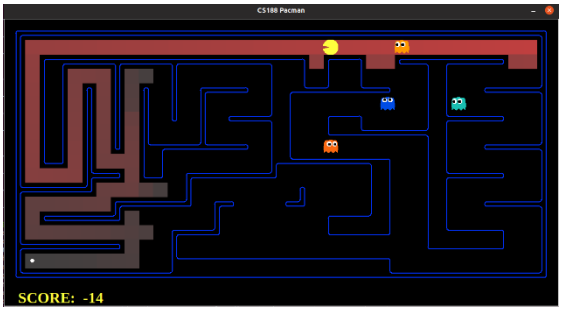
```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 pacman.py -l mediumDottedMaze -p StayEastSearchAgent
Path found with total cost of 1 in 0.1 seconds
Search nodes expanded: 186
Pacman emerges victorious! Score: 646
Average Score: 646.0
Scores:        646.0
Win Rate:      1/1 (1.00)
Record:        Win
mahdi@OSlab:~/Desktop/AI_P1/search$
```

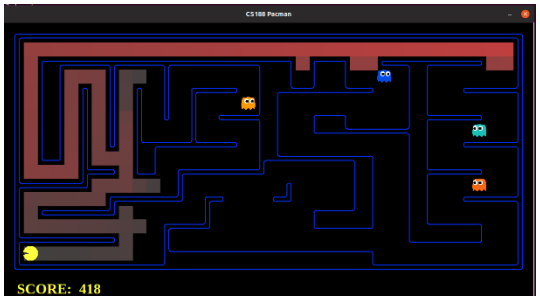




python3 pacman.py -l mediumScaryMaze -p StayWestSearchAgent

```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 pacman.py -l mediumScaryMaze -p StayWestSearchAgent
Path found with total cost of 68719479864 in 0.1 seconds
Search nodes expanded: 108
Pacman emerges victorious! Score: 418
Average Score: 418.0
Scores:        418.0
Win Rate:      1/1 (1.00)
Record:        Win
mahdi@OSlab:~/Desktop/AI_P1/search$
```



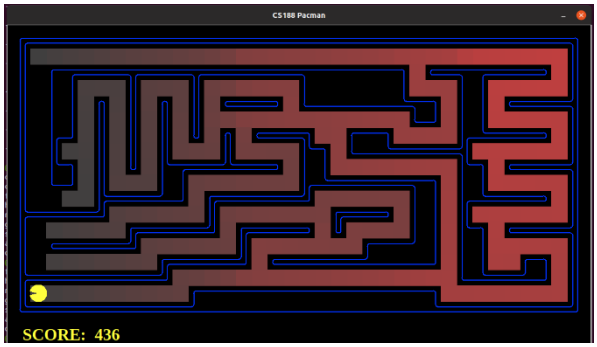


نکته: شما باید برای StayEastSearchAgent و StayWestSearchAgent به دلیل تابع هزینه نمایی، به ترتیب هزینه مسیر بسیار پایین و بسیار بالایی داشته باشید(برای جزئیات بیشتر به فایل py.searchAgents مراجعه کنید).

ابتدا به کمک دستور زیر حالت StayEastSearchAgent را اجرا میکنیم:

```
python3 pacman.py -l mediumMaze -p StayEastSearchAgent
```

```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 pacman.py -l mediumMaze -p StayEastSearchAgent
Path found with total cost of 1 in 0.1 seconds
Search nodes expanded: 260
Pacman emerges victorious! Score: 436
Average Score: 436.0
Scores:        436.0
Win Rate:      1/1 (1.00)
Record:        Win
mahdi@OSlab:~/Desktop/AI_P1/search$
```

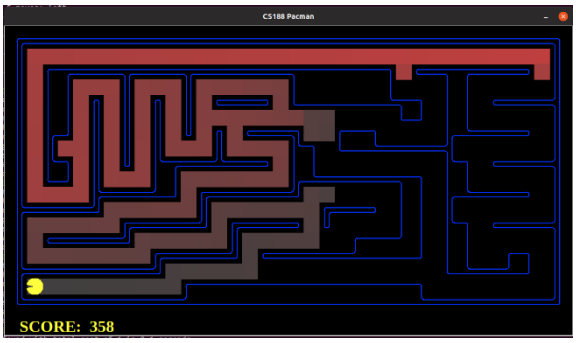


در این حالت اگر در شکل سمت چپ نگاه کنید هزینه برابر ۱ شده است.

حال به کمک دستور زیر حالت StayWestSearchAgent را اجرا میکنیم:

```
python3 pacman.py -l mediumMaze -p StayWestSearchAgent
```

```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 pacman.py -l mediumMaze -p StayWestSearchAgent
Path found with total cost of 17183280440 in 0.1 seconds
Search nodes expanded: 173
Pacman emerges victorious! Score: 358
Average Score: 358.0
Scores:        358.0
Win Rate:      1/1 (1.00)
Record:        Win
mahdi@OSlab:~/Desktop/AI_P1/search$
```



در این حالت اگر در شکل سمت چپ نگاه کنید هزینه برابر ۱۷۱۸۳۲۸۰۴۴۰ شده است.

بخش چهارم

جست و جوی A استار

کد نوشته شده:

در فایل search.py لازم است تا تابع aStarSearch را کامل کنیم. کد مربوطه به صورت زیر است:

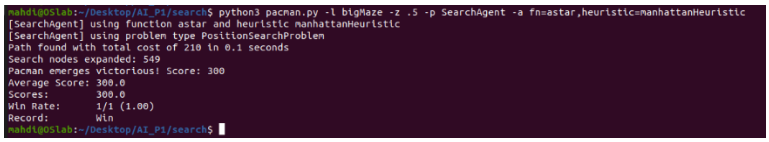
```python
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    # first we should get statrting node
    start_node = problem.getStartState()

    # then we should check that if this starting state is goal state then we don't need any action
    # (if start_node == goal -> actions_list = [])
    if problem.isGoalState(start_node):
        return []

    # for implementing BFS we can use Queue data structure
    # the Queue elements are in form of ((node, list of actions to cur_node, total cost to cur_node(=g_n)), priority (=f(n)=g(n)+h(n)))
    Astar_priority_queue = util.PriorityQueue()

    # we checked the start node seperately then we add it to Astar_priority_queue
    Astar_priority_queue.push((start_node, [], 0), 0)
    # we can hold the checked nodes in a list
    checked_nodes = []

    while not Astar_priority_queue.isEmpty():

        cur_node, actions_list, prev_cost = Astar_priority_queue.pop()
        # we should check if the cur_node isn t check later then we add it to checked_nodes and check it
        if cur_node not in checked_nodes:
            checked_nodes.append(cur_node)

            if problem.isGoalState(cur_node):
                return actions_list

            for next_node, next_action, cost in problem.getSuccessors(cur_node):
                g_n = prev_cost + cost
                # now we should use f(n) = g(n) + h(n)
                f_n = g_n + heuristic(next_node, problem)
                new_action = actions_list + [next_action]
                Astar_priority_queue.push((next_node, new_action, g_n),f_n)
```

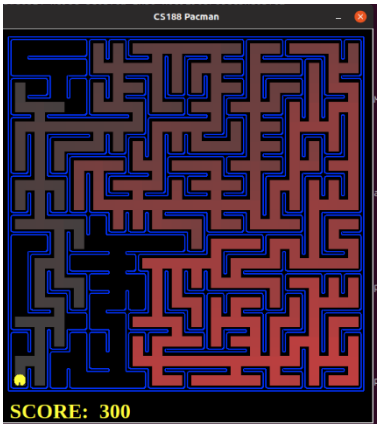
پیاده سازی این تابع نیز تا حد زیادی شبیه به UCS میباشد ولی با این تفاوت که در اینجا الگوریتم جست و جوی ما آگاهانه میباشد و یک تابع heuristic نیز به عنوان ورودی میپذیرد. در اینجا نیز برای پیاده سازی و نگهداری fringe از ساختمان داده priority queue استفاده میکنیم.

همچنین چون heuristic داریم لازم است تا هزینه تخمینی تا نود بعدی را به کمک $f(n) = g(n)+h(n)$ حساب کنیم. مقدار $g(n)$ که برابر هزینه واقعی تا نود و state فعلی است و $h(n)$ هم هزینه پیشبینی شده تا نود بعدی توسط تابع heuristic میباشد. لازم است در اینجا $f(n)$ به عنوان اولویت در نظر گرفته شود و هرگره که $f(n)$ کمتر داشته باشد باید برای گسترش یافتن در اولویت باشد. پس لازم است تا این مقدار را نیز به همراه مقادیر مربوط به هزینه واقعی تا اینجا، استیت بعدی، و action لازم برای رسیدن به آن را نگه داریم.

نمونه‌هایی از اجرای کد:

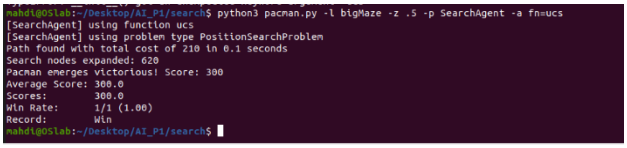
python3 pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

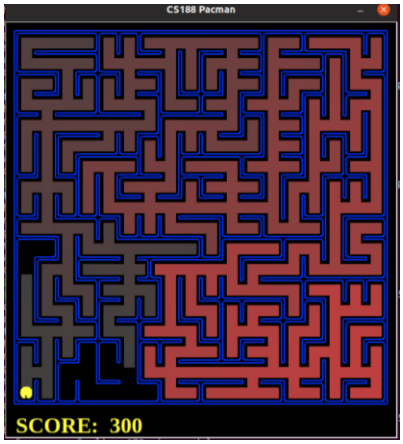




اگر بخواهیم با الگوریتم USC این قسمت را مقایسه کنیم، همین دستور را با الگوریتم جست‌وجوی UCS اجرا میکنیم و داریم:

python3 pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=ucs





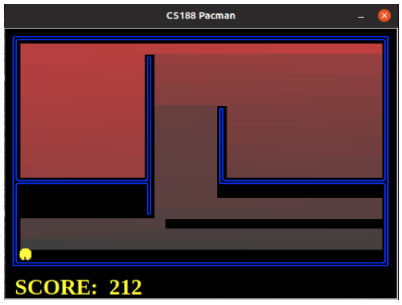
همانطور که مشاهده میشود در ucs تعداد نودهای بیشتری expand شده است و تاحدی الگوریتم A* سریع‌تر میباشد.

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.

الگوریتم DFS :

python3 pacman.py -l openMaze -z .5 -p SearchAgent

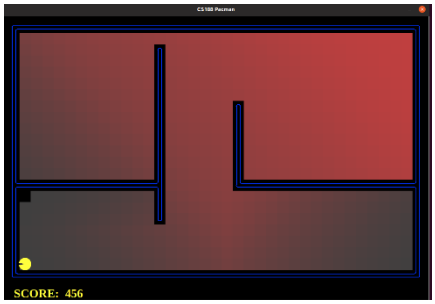
```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 pacman.py -l openMaze -z .5 -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 298 in 0.1 seconds
Search nodes expanded: 576
Pacman emerges victorious! Score: 212
Average Score: 212.0
Scores:        212.0
Win Rate:      1/1 (1.00)
Record:        Win
mahdi@OSlab:~/Desktop/AI_P1/search$
```



الگوریتم BFS :

python3 pacman.py -l openMaze -p SearchAgent -a fn=bfs

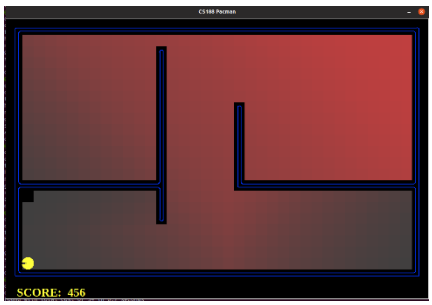
```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 pacman.py -l openMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.2 seconds
Search nodes expanded: 682
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
mahdi@OSlab:~/Desktop/AI_P1/search$
```



الگوریتم UCS :

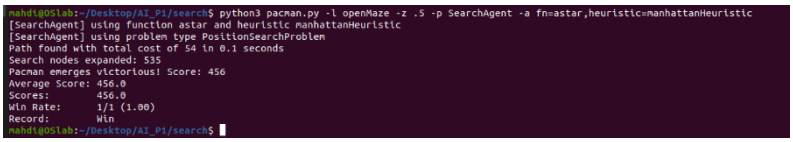
python3 pacman.py -l openMaze -p SearchAgent -a fn=ucs

```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 pacman.py -l openMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.2 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
mahdi@OSlab:~/Desktop/AI_P1/search$
```



الگوریتم A* :

python3 pacman.py -l openMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic



توضیح DFS :

همانطور که قبل تر توضیح دادیم در پیاده سازی از استک که به صورت LIFO میباشد استفاده کردیم. به ترتیب گره‌های سمت بالا و پایین و راست و چپ داخل fringe وارد میشوند و در جهت برعکس یعنی چپ و بعد راست و بعد پایین و بعد بالا از فرینج pop میشوند و expand میشوند. در این شک هم اگر دقت شود سمت چپ و بالای شکل پررنگ تر است که یعنی زودتر کاوش شده اند. این روش بهینه نیست ولی کامل و همچنین ناآگاهانه است.

توضیح BFS :

میدانیم که برای پیاده سازی فرینج از queue استفاده کردیم که FIFO میباشد. نودها به ترتیبی که وارد میشوند خارج میشوند یعنی سمت بالا و پایین و راست و چپ. همانطور که در تصویر مشخص است سمت بالا و نزدیک به نقطه شروع ما پر رنگ تر ات یعنی زودتر بررسی شدند. هرچه جلوتر میرویم چون عمق گراف زیاد تر میشود و ما هم در اینجا اول در پهنای گراف پیش میرویم پس پایین ها کم رنگ تر است. چون در اینجا هزینه ۱ است مسیر بهینه را به ما میدهد. این نیز یک جست و جوی نا آگاهانه است و کامل میباشد ولی لزوما همیشه بهینه نیست.

توضیح UCS :

در این روش میدانیم برای نگهداری fringe از priority queue استفاده میکنیم. اولویت ما هزینه میباشد که یعنی هرچه فاصله تا هدف کمتر باشد اولویت بیشتری دارد. در اینجا چون هزینه هر نود تا نود بعدی ۱ میباشد رسما مانند BFS عمل میکند و در عرض گراف تشکیل شده پیش میرویم. به همین دلیل نواحی قرمز رنگ شبیه BFS است و ابتدا نواحی نزدیک تر به مبدا یعنی سمت راست بالا کاوش شده و پر رنگ ترند.

توضیح *A :

در این روش نیز برای نگهداری fringeها، از صف اولویت استفاده میکنیم. در اینجا اولویت ما f(n) میباشد. درواقع دراینجا الگوریتم ما یک الگوریتم جست‌وجوی آگاهانه میباشد و از یک heuristic برای تخمین هزینه تا هدف استفاده میشود. این هیوریستیک فاصله منهتن میباشد. به همین دلیل یک دیدی از هدف داریم و اگر دقت شود دیگر نیازی نیست نودهای زیادی کاوش شوند بلکه به تابع هیوریستیکمان نگاه میکنیم و براساس آن پیش میرویم. بین نودهایی که در fringe داریم اونی که کمترین هزینه هست اولویت بیشتری در صف دارد و اول آن را انتخاب میکنیم. به این ترتیب سمت راست بالا چون به استیت آغازین نزدیک تر است پر رنگ تر میباشد. چون ما قبل از اینکه همه نودها را نیازی باشد expand کنیم به هدف برسیم برخی از آن ها اصلا کاوش نمیشوند و محدوده مشکی رنگ را تشکیل میدهند.

اگر مقایسه کنیم متوجه میشویم که در الگوریتم DFS کمترین امتیاز یعنی ۲۱۲ را گرفتیم ولی در ۳ الگوریتم دیگر همگی امتیاز 456 گرفتیم.

در الگوریتم DFS در مجموع ۵۷۶ نود و در الگوریتم‌های BFS و UCS مقدار ۶۸۲ نود و در الگوریتم *A تعداد ۵۳۵ نود expand شده است.

مقدار هزینه در الگوریتم DFS برابر ۲۹۸ و در مدت زمان ۰,۱ ثانیه و در BFS و UCS برابر ۵۴ در ۰,۲ ثانیه میباشد و برای *A برابر ۵۴ و در ۰,۱ ثانیه میباشد.

با این اوصاف میتوان بهترین و بهینه‌ترین الگوریتم *A بوده که هم سریع بوده و هم امتیاز بالا گرفته و هزینه‌ی کم داشته است. DFS نیز الگوریتم بدتری بوده چراکه کمترین امتیاز را گرفته و دلخواه ما نمیباشد. از معایب دیگر آن هزینه‌ی بالای آن میباشد.البته نسبت به BFS و UCS زمان کمتری طول کشیده.

دو الگوریتم UCS و BFS در این مسئله شبیه یک دیگر بودند. و اگر بخواهیم رتبه بندی کنیم با درنظر گرفتن همه‌ی ویژگی‌هایی که داشتند در مقام دوم قرار میگیرند.

بخش پنجم

پیدا کردن همه گوشه‌ها

کد نوشته شده

برای این قسمت لازم است تا در کلاس CornersProblem در فایل searchAgents.py تغییرات لازم را بدهیم:

ابتدا در بخشی از تابع __init__ که خالی است را باید پر کنیم:

```
def __init__(self, startingGameState):
    """
    Stores the walls, pacman's starting position and corners.
    """
    self.walls = startingGameState.getWalls()
    self.startingPosition = startingGameState.getPacmanPosition()
    top, right = self.walls.height-2, self.walls.width-2
    self.corners = ((1,1), (1,top), (right, 1), (right, top))
    for corner in self.corners:
        if not startingGameState.hasFood(*corner):
            print('Warning: no food in corner ' + str(corner))
    self._expanded = 0 # DO NOT CHANGE; Number of search nodes expanded
    # Please add any code here which you would like to use
    # in initializing the problem
    "*** YOUR CODE HERE ***"

    # initialized checked corners list
    self.checked_corners_list = [False,False,False,False]
```

سپس تابع بعدی getStartState میباشد:

```
def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    "*** YOUR CODE HERE ***"

    # return the start state and the list of checked corners
    # here list of checked corners is [False,False,False,False] because in initial state we don't visit any corner
    return (self.startingPosition, self.checked_corners_list)
```

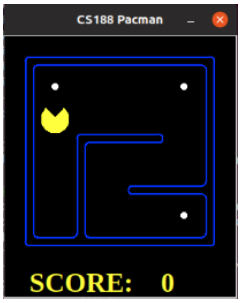
در نهایت به سراغ تابع getSuccessors میرویم:

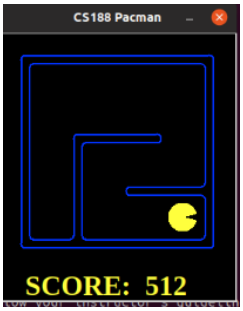
```
def getSuccessors(self, state):
    """
    Returns successor states, the actions they require, and a cost of 1.

     As noted in search.py:
        For a given state, this should return a list of triples, (successor,
        action, stepCost), where 'successor' is a successor to the current
        state, 'action' is the action required to get there, and 'stepCost'
        is the incremental cost of expanding to that successor
    """
    successors = []
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
        # Add a successor state to the successor list if the action is legal
        # Here's a code snippet for figuring out whether a new position hits a wall:
        #   x,y = currentPosition
        #   dx, dy = Actions.directionToVector(action)
        #   nextx, nexty = int(x + dx), int(y + dy)
        #   hitsWall = self.walls[nextx][nexty]

        "*** YOUR CODE HERE ***"
        # in the above sentences tell us the cost value is 1
        cost = 1
        # visited corners until this position
        checked_corners = state[1]
        # current position
        x,y = state[0]
        # hold the direction in two variables
        dx,dy = Actions.directionToVector(action)
        # new position is created from current position and direction
        x_new,y_new = int(x + dx),int(y + dy)
        # copy the checked_corners list to a new variable
        checked_corners_new = checked_corners[:]
        hit_wall = self.walls[x_new][y_new]
        if not hit_wall:
            checked_corners_index = 0
            # check if this position is one of the corners then we upadte the corner visited list
            for corner in self.corners:
                if (x_new,y_new) == corner:
                    checked_corners_new[checked_corners_index] = True
                    break
                else:
                    checked_corners_index += 1
            new_state = ((x_new,y_new),checked_corners_new)
            successors.append((new_state,action,cost))

    self._expanded += 1 # DO NOT CHANGE
    return successors
```

نمونه هایی از اجرای کد:

python3 pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

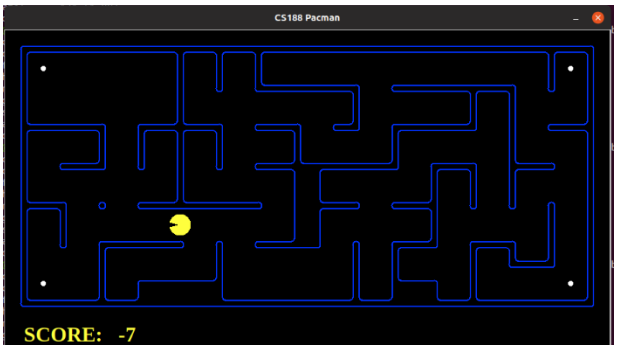
```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 252
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
mahdi@OSlab:~/Desktop/AI_P1/search$
```

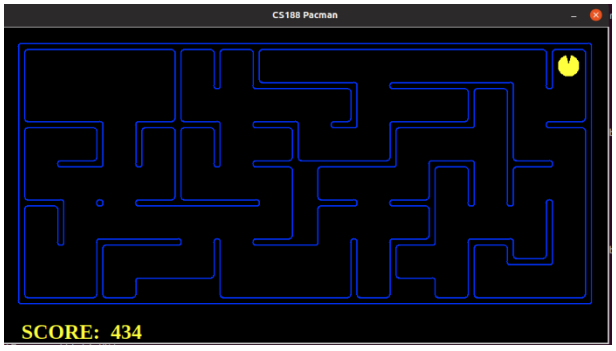




python3 pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.5 seconds
Search nodes expanded: 1966
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
mahdi@OSlab:~/Desktop/AI_P1/search$
```





بخش ششم

هیوریستیک برای مسئله گوشه‌ها

کد نوشته شده:

باید تابع cornersHeuristic در کلاس CornersProblem در فایل searchAgents.py را پیاده سازی کنیم. کد نوشته شده به صورت زیر است:

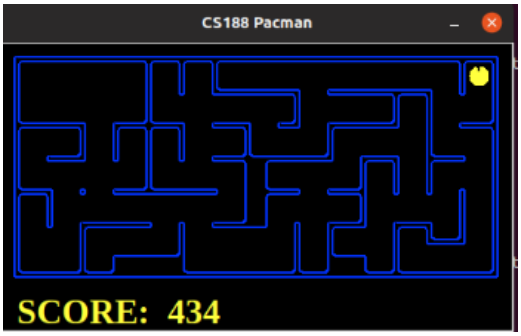
```python
def cornersHeuristic(state, problem):
    """
    A heuristic for the CornersProblem that you defined.

      state:   The current search state
               (a data structure you chose in your search problem)

      problem: The CornersProblem instance for this layout.

    This function should always return a number that is a lower bound on the
    shortest path from the state to a goal of the problem; i.e.  it should be
    admissible (as well as consistent).
    """
    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

    "*** YOUR CODE HERE ***"
    # I intialize the heuristic value with 0
    heuristic_value = 0
    # hold current position and checked coners list in seperate variables
    cur_position = state[0][:]
    checked_corners = state[1][:]
    # if we are at goal state then we should return 0 and we dont need to cintinue
    if checked_corners == [True,True,True,True]:
        return heuristic_value

    # there remains at most 4 corners unvisited, so we need to go for four times
    for i in range(4):
        corners_distance = [0,0,0,0]
        # calculate the distance between the current position and each of the corners
        for i in range(4):
            corners_distance[i] = util.manhattanDistance(cur_position,corners[i])

        # first we should initilize min_distance
        # for doing this we add the first unvisited corner distance to min_distance variable
        # also if there isn't any distace less than this we can say this index is closest_corner_index
        min_distance = 0
        closest_corner_index = 0
        for i in range(4):
            if not checked_corners[i]:
                min_distance = corners_distance[i]
                closest_corner_index = i
                break
        # we check if any distance is less than min distance and the corner is unvisited
        # then we should update min_distance and closest_index_corner
        corner_index = 0
        for distance in corners_distance:
            if(not checked_corners[corner_index]) and (distance < min_distance):
                min_distance = distance
                closest_corner_index = corner_index
            corner_index+=1
        # if this closest_corner that we find isn't unvisited
        # then we should go to that corner and add the heuristic value to that
        # also because we go to that corner we should make it True in checked_corners
        if (not checked_corners[closest_corner_index]):
            heuristic_value += corners_distance[closest_corner_index]
            cur_position = corners[closest_corner_index][:]
            checked_corners[closest_corner_index]=True
        else:
            break
    # after all we should return the total heurist value.
    # it estimates the value if we want go the all unvisited corners from this position
    return heuristic_value
    #return 0 # Default to trivial solution
```

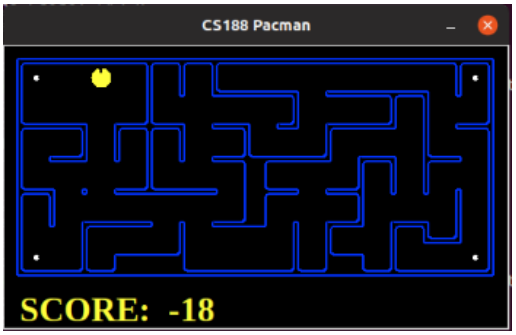
توضیحات مربوط به پیاده سازی در قسمت جواب به سوال نوشته شده است.

نمونه‌هایی از اجرای کد:

python3 pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.1 seconds
Search nodes expanded: 692
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
mahdi@OSlab:~/Desktop/AI_P1/search$
```





سوال) **هیوریستیک خود را توضیح دهید و سازگاری آن را استدلال کنید**

میدانیم که هدف ما این است که هر ۴ گوشه را بازدید کنیم. در هیوریستیکی که تعریف کردیم به این صورت عمل میکند که ابتدا مکان فعلی را که با cur_position = state[0][:] به دست می‌آوریم. سپس فاصله مکان فعلی را تا نزدیک ترین گوشه با حساب کردن فاصله manhatan به دست می‌آوریم. بعد باید باز به کمک manhatan فاصله نزدیک ترین گوشه که پیدا کردیم را تا نزدیک ترین گوشه به آن گوشه بیابیم. همین کار را ادامه میدهیم تا جایی که هر چهارگوشه بازدید شده باشند. لازم به ذکر است که همه این فاصله‌ها را با هم جمع میکنیم و به عنوان heuristic برمیگردانیم.

خب حال لازم است تا شرط سازگاری را نشان دهیم. اگر سازگار باشد admissible نیز میباشد.

البته منهتن به این دلیل که ما مسئله را ریلکس کردیم واضح است که admissible کردیم چون که دیوارها را درنظر نگرفتیم و در بهترین حالت هزینه واقعی همان هزینه منهتن است و اگرنه هزینه واقعی بیشتر است پس میتوان گفت:

$$0 \le h(A) \le h^*(A) \rightarrow admissible$$

حال برای اثبات سازگاری، دو نود A وB را درصفحه بازی نظر بگیریم. طبق بالا میتوان برای هریک گفت:

$$\begin{cases} h(A) \le h^*(A) \\ h(B) \le h^*(B) \end{cases} \xrightarrow{\text{از ریاضیات داریم}} h(A) - h(B) \le |\, h^*(A) - h^*(B)| \quad (1)$$

همچنین اگر فرض کنیم از A به B راهی وجود داشته باشد و نقطه شروع A باشد میتوان گفت:

$$h^*(A) \le h^*(B) + cost(A\ to\ B) \rightarrow h^*(A) - h^*(B)\ \le cost(A\ to\ B)\ \ (2)$$

همچنین اگر از A به B راهی باشد قطعا از Bبه A هم مسیر وجود دارد چراکه در این مسئله مسیرهای ما یک طرفه نیستند. حال اگر فرض کنیم ابتدا در B باشیم مثل بالا میتوان گفت:

$$h^*(B) \le h^*(A) + cost(B\ to\ A) \rightarrow h^*(B) - h^*(A)\ \le cost(B\ to\ A)\ \ (3)$$

در توضیح روابط ۲ و ۳ میتوان گفت که هزینه‌ی واقعی رفتن از A به هدف قطعا کمتر مساوی مجموع هزینه‌های واقعی رفتن از A به B و سپس از B به هدف میباشد. چرا که در بهترین حالت B در مسیر A تا هدف است و مجموع هزینه گفته شده برابر هزینه A تا هدف است. اگر B در این مسیر نباشد پس اگر از هر مسیر دیگری برویم هزینه بیشتر خواهد شد.

از آنجایی که در این مسئله هزینه‌ها برحسب فاصله میباشد و هزینه‌ی هر خانه تا خانه‌ی مجاورش ۱ میباشد پس اگر راهی از Aبه B باشد هزینه اش با راهی که از B به A هست برابر میباشد:

$$cost(A\ to\ B) = cost(B\ to\ A)\ \ (4)$$

حال باتوجه به رابطه۳ پیش میرویم:

$$\overset{3}{\Rightarrow} h^*(B) - h^*(A)\ \leq cost(B\ to\ A)\ \rightarrow -\big(h^*(A) - h^*(B)\big) \leq cost(B\ to\ A)$$

$$\overset{4}{\Rightarrow} -\big(h^*(A) - h^*(B)\big) \leq cost(A\ to\ B) \xrightarrow{\text{ضرب طرفین در } -1} h^*(A) - h^*(B) \geq -cost(A\ to\ B) \quad (5)$$

حال باتوجه به روابط ۵ و ۲ داریم:

$$\xrightarrow{2\ و\ 5} -\ cost(A\ to\ B) \leq h^*(A) - h^*(B)\ \leq cost(A\ to\ B)$$

$$\rightarrow |h^*(A) - h^*(B)|\ \leq cost(A\ to\ B)\ \ (6)$$

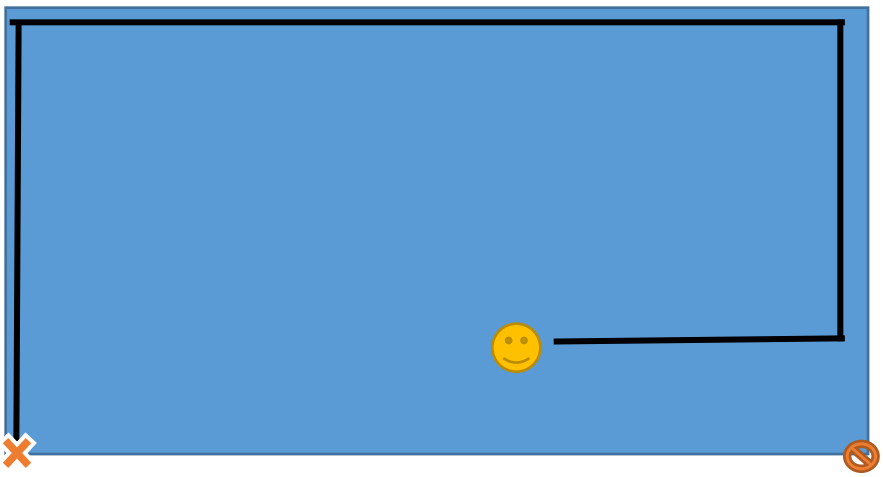
حال با این اوصاف میتوان گفت:

$$\begin{cases} \overset{6}{\rightarrow} |h^*(A) - h^*(B)|\ \leq cost(A\ to\ B) \\ \overset{1}{\rightarrow} h(A) - h(B) \leq |\ h^*(A) - h^*(B)| \end{cases} \rightarrow h(A) - h(B)\ \leq cost(A\ to\ B)$$

بنابراین ثابت کردیم که سازگار نیز میباشد.

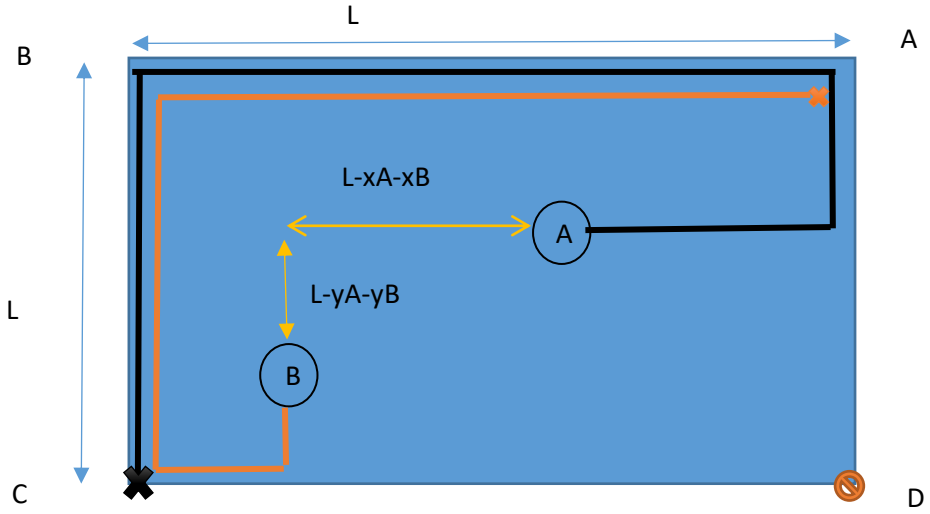
روش دوم اثبات:

در این سوال استیت ما مکان فعلی و لیست ۴ گوشه میباشد که آن هایی که مشاهده نشده اند false میباشند. اگر همه گوشه بازبینی شوند هر چهارگوشه true میشود که یعنی به هدف رسیدیم. حال برای توضیح شکل زیر را در نظر بگیرید:



برای مثال اگر در موقعیت ☺ باشیم و گوشه سمت راست پایین قبلا مشاهده شده باشد ابتدا به باید گوشه سمت راست بالا برویم که هنوز visit نشده و نزدیک ترین گوشه به ماست. طول این مسیر را ذخیره میکنیم. بعد باید به دو گوشه دیگر نیز برویم و فاصله رفتن به آن‌ها را نیز جمع میکنیم. این مقدار را به عنوان heuristic برمیگردانیم. چون در عمل ممکن است دیوارها باشند و این حالت ریلکس شده است پس هیوریستیک ما admissible است.

حال برای سازگاری دو نود A و B را به صورت زیر بگیرید:



طبقشکل میتوان فهمید که نود A ابتدا به گوشه A و بعد B و C رفته و D هم ازقبل مشاهده شده. همچنین نود B هم ابتدا به C رفته و سپس به B و A رفته است.

حال داریم:

$$\begin{cases} h(A) = xA + yA + 2L \\ h(B) = xB + yB + 2L \end{cases} \rightarrow h(A) - h(B) = (xA + yA) - (xB + yB)$$

همچنین فاصله منهتن بین A و B برابر $2L - (xA + yA) - (xB + yB)$ میباشد.چون از A زودتر به A رسیدیم داریم:

$$\begin{cases} xA + yA < L - xA + yA \quad \rightarrow \quad xA < \dfrac{L}{2} \\ xA + yA < L - xA + L - yB \quad \rightarrow \quad xA + yB < L \end{cases} \quad (1)$$

برای این که هیوریستیک سازگار باشد باید $h(A) - h(B) \leq cost(A\ to\ B)$

این cost(A to B) قطعا از فاصله منهتن بین A و B بیشتره.

$$h(A) - h(B) = (xA + yA) - (xB + yB)$$

$$manhatna(A, B) = 2L - (xA + yA) - (xB + yB)$$

$$h(A) - h(B) \overset{?}{\leqq} manhatna(A, B) \rightarrow (xA + yA) - (xB + yB) \overset{?}{\leqq} 2L - (xA + yA) - (xB + yB)$$

$$\rightarrow xA + yB \overset{?}{\leqq} L$$

این رابطه را قبلا در (۱) اثبات کردیم پس :

$$h(A) - h(B) \leq manhatna(A, B) \leq cost(A\ to\ B)$$

پس سازگار است.

بخش هفتم

خوردن همه نقطه ها

کد نوشته شده:

تابع *foodHeuristic* در کلاس *FoodSearchProblem* در فایل *searchAgents.py* را باید تکمیل کنیم:

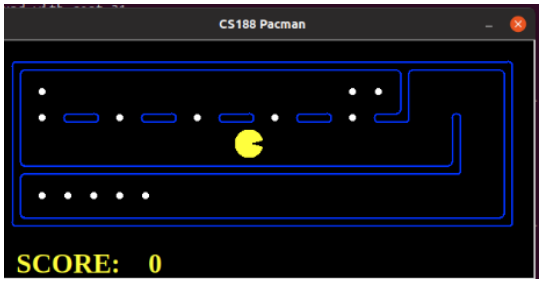
```python
def foodHeuristic(state, problem):
    """
    Your heuristic for the FoodSearchProblem goes here.

    This heuristic must be consistent to ensure correctness.  First, try to come
    up with an admissible heuristic; almost all admissible heuristics will be
    consistent as well.

    If using A* ever finds a solution that is worse uniform cost search finds,
    your heuristic is *not* consistent, and probably not admissible!  On the
    other hand, inadmissible or inconsistent heuristics may find optimal
    solutions, so be careful.

    The state is a tuple ( pacmanPosition, foodGrid ) where foodGrid is a Grid
    (see game.py) of either True or False. You can call foodGrid.asList() to get
    a list of food coordinates instead.

    If you want access to info like walls, capsules, etc., you can query the
    problem.  For example, problem.walls gives you a Grid of where the walls
    are.

    If you want to *store* information to be reused in other calls to the
    heuristic, there is a dictionary called problem.heuristicInfo that you can
    use. For example, if you only want to count the walls once and store that
    value, try: problem.heuristicInfo['wallCount'] = problem.walls.count()
    Subsequent calls to this heuristic can access
    problem.heuristicInfo['wallCount']
    """
    position, foodGrid = state
    "*** YOUR CODE HERE ***"

    # I use mazeDistance() function for finding the distance between two points. it usess search method we use in the problem
    # the goal is eating all food. by this function we can estimate the distance between this position and each food.
    # so we can say our heuristic must show the longest distance between this position and all food

    heuristic_value = 0
    for food_pos in foodGrid.asList():
        pos_food_distance = mazeDistance(position,food_pos,problem.startingGameState)
        # if the distance between dood and position is greater than heuristic value
        # then we should update heuristic value (heuristic_value contains the longest distance to food)
        if pos_food_distance > heuristic_value:
            heuristic_value = pos_food_distance
    return heuristic_value

    # in code below there is another implementation
    """
    heuristic_value = 0
    # find the farthest distance by Astar search using mazeDistance() function.
    for y in range(foodGrid.height):
        for x in range(foodGrid.width):
            # first we should check in that coordiate we have food
            if(foodGrid[x][y] == True):
                pos_food_distance = mazeDistance(position,(x,y),problem.startingGameState)
                # if the distance between dood and position is greater than heuristic value
                # then we should update heuristic value (heuristic_value contains the longest distance to food)
                if pos_food_distance > heuristic_value:
                    heuristic_value = pos_food_distance
    return heuristic_value
    """
```

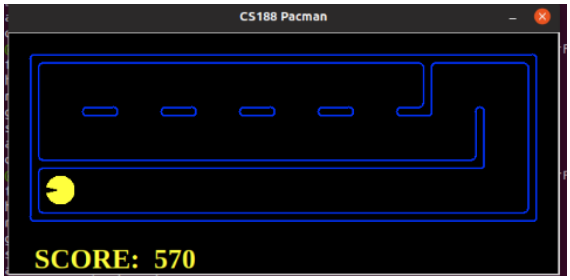
در این روش برای اینکه تعداد نودهای کمتری اکسپند شود لازم است تا مثلا به جای فاصله‌ی منهتن از هیوریستیک قوی تری استفاده کنیم. در اینجا از maze distance استفاده کردیم. همچنین درواقع مقدار هیوریستیک ما بیشترین فاصله maze distance از غذاها برای هر گره میباشد.

نمونه‌ای از اجرای کد:

python3 pacman.py -l trickySearch -p AStarFoodSearchAgent

```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 pacman.py -l trickySearch -p AStarFoodSearchAgent
Path found with total cost of 60 in 72.5 seconds
Search nodes expanded: 4137
Pacman emerges victorious! Score: 570
Average Score: 570.0
Scores:        570.0
Win Rate:      1/1 (1.00)
Record:        Win
mahdi@OSlab:~/Desktop/AI_P1/search$
```





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

در این جا هدف آن است که تمامی غذاها توسط پکمن خورده شود. همچنین برای حل مسئله از روش A* استفاده میشود که یک تابع هیوریستیک نیاز دارد تا هزینه هر گره تا مقصد را تخمین بزند.

در این روش برای اینکه تعداد نودهای کمتری اکسپند شود لازم است تا مثلا به جای فاصله‌ی منهتن از تخمین قوی تری برای یافتن فاصله استفاده کنیم. در اینجا از maze distance استفاده کردیم. مقدار هیوریستیک ما بیشترین فاصله maze distance از غذاها برای هر گره میباشد.

برای اینکه نشان دهیم هیوریستیک ما admissible هست لازم است بگوییم که cost داده شده توسط هیوریستیک از مقدار cost واقعی کمتر است. میدانیم که تابع mazeDistance مقدار فاصله دو نقطه را به ما برمیگرداند و اگر به پیاده سازی آن نگاه کنیم درواقع طول مسیری که با الگوریتم bfs بین دونطه میابد را میدهد. از طرفی چون هزینه ۱ است و درواقع فاصله هرخانه تا خانه مجاورش ۱ است پس همین bfs به صورت بهینه عمل میکند و تخمین خوبی به ما میدهد و میتوان گفت به کمک آن حداقل فاصله بین مکان فعلی و هر کدام از غذا ها را میدهد. حال میتوان گفت برای اینکه همه غذاها را بخوریم میتوان فاصله از دورترین غذا را به عنوان هیوریستیک گرفت. این از هزینه واقعی کمتر است چراکه ما مسئله را ریلکس کردیم و درواقع اگر همه غذاها در مسیر ما تا دورترین غذا باشند دراین صورت هزینه واقعی برابر هزینه ما تا دورترین غذاست ولی اگر هر یک از غذاها در این مسیر نباشند و ما بخواهیم از مسیر تعیین شده تادورترین غذا برای خوردن آن‌ها خارج شویم پس هزینه واقعی بیشتر از هزینه تخمینی میشود پس:

$$0 \leq h(A) \leq h^*(A) \rightarrow admissible$$

حال برای اثبات سازگاری، دو نود A وB را درصفحه بازی نظر بگیریم. طبق بالا میتوان برای هریک گفت:

$$\begin{cases} h(A) \leq h^*(A) \\ h(B) \leq h^*(B) \end{cases} \xrightarrow{\text{از ریاضیات داریم}} h(A) - h(B) \leq |\, h^*(A) - h^*(B)| \quad (1)$$

همچنین اگر فرض کنیم از A به B راهی وجود داشته باشد و نقطه شروع A باشد میتوان گفت:

$$h^*(A) \leq h^*(B) + cost(A\ to\ B) \rightarrow h^*(A) - h^*(B) \ \leq cost(A\ to\ B) \ \ (2)$$

همچنین اگر از A به B راهی باشد قطعا از Bبه A هم مسیر وجود دارد چراکه در این مسئله مسیرهای ما یک طرفه نیستند. حال اگر فرض کنیم ابتدا در B باشیم مثل بالا میتوان گفت:

$$h^*(B) \leq h^*(A) + cost(B\ to\ A) \rightarrow h^*(B) - h^*(A) \ \leq cost(B\ to\ A) \ \ (3)$$

در توضیح روابط ۲ و ۳ میتوان گفت که هزینه‌ی واقعی رفتن از A به هدف قطعا کمتر مساوی مجموع هزینه‌های واقعی رفتن از A به B و سپس از B به هدف میباشد. چرا که در بهترین حالت B در مسیر A تا هدف است و مجموع هزینه گفته شده برابر هزینه A تا هدف است. اگر B در این مسیر نباشد پس اگر از هر مسیر دیگری برویم هزینه بیشتر خواهد شد.

از آنجایی که در این مسئله هزینه‌ها برحسب فاصله میباشد و هزینه‌ی هر خانه تا خانه‌ی مجاورش ۱ میباشد پس اگر راهی از Aبه B باشد هزینه اش با راهی که از B به A هست برابر میباشد:

$$cost(A\ to\ B) = cost(B\ to\ A)\ \ (4)$$

حال باتوجه به رابطه۳ پیش میرویم:

$$\overset{3}{\Rightarrow} h^*(B) - h^*(A)\ \leq cost(B\ to\ A)\ \rightarrow -\big(h^*(A) - h^*(B)\big) \leq cost(B\ to\ A)$$

$$\overset{4}{\Rightarrow} -\big(h^*(A) - h^*(B)\big) \leq cost(A\ to\ B) \xrightarrow{\text{ضرب طرفین در } -1} h^*(A) - h^*(B) \geq -cost(A\ to\ B) \quad (5)$$

حال باتوجه به روابط ۵ و ۲ داریم:

$$\overset{2\ و\ 5}{\longrightarrow} -cost(A\ to\ B) \leq h^*(A) - h^*(B)\ \leq cost(A\ to\ B)$$

$$\rightarrow |h^*(A) - h^*(B)|\ \leq cost(A\ to\ B)\ \ (6)$$

حال با این اوصاف میتوان گفت:

$$\begin{cases} \overset{6}{\rightarrow} |h^*(A) - h^*(B)|\ \leq cost(A\ to\ B) \\ \overset{1}{\rightarrow} h(A) - h(B) \leq |\ h^*(A) - h^*(B)| \end{cases} \rightarrow h(A) - h(B)\ \leq cost(A\ to\ B)$$

بنابراین ثابت کردیم که سازگار نیز میباشد.

روش دوم اثبات راه حل اول:

روش دوم (هیوریستیک متفاوت)

در این حالت میتوان برای هیوریستیک از فاصله منهتنی استفاده کرد. به این صورت میباشد که فاصله‌ی غذاهای نخورده را تا مکان فعلی به کمک منهتن حساب میکنیم و کمترین فاصله را میابیم و در heuristic_value میریزیم. غذایی که در این فاصله است را انتخاب میکنیم. سپس فاصله منهتن غذاهای دیگر تا غذای یافت شده در مرحله قبل را میابیم و غذایی که در بیشترین فاصله قرار دارد را در نظر میگیریم سپس فاصله منهتن بین غذای قبلی و این غذا را به مقدار heuristic_value جمع میکنیم اگر این مقدار کمتر از تعداد غذاهای مانده باشد این مقدار را به عنوان هیوریستیک برمیگردانیم در غیر این صورت تعداد غذا ها را برمیگردانیم. در اینجا چون هزینه ها ۱ میباشد میتوان این کار را کرد. درواقع اگر مسیر کمتر از تعداد غذا ها باشد یعنی کل غذاها را نمیخوریم. همچنین اگر غذایی در لیست غذاها نباشد ۰ را برمیگردانیم و این شرط را ابتدای الگوریتم چک میکنیم.

این روش admissible است چرا که در بهترین حالت همه غذاها در مسیر منهتنی یافت شده است ولی درغیر اینصورت قطعا هزینه واقعی بیشتر است .

سازگار بودن آن هم مثل روابط نوشته شده برای روش قبل قابل اثبات است.

بخش هشتم

جست و جوی نیمه بهینه

کد نوشته شده:

ابتدا لازم است تا AnyFoodSearchProblem را کامل کنیم. فقط تابع isGoalState را کامل کنیم:

```
628
629     def isGoalState(self, state):
630         """
631         The state is Pacman's position. Fill this in with a goal test that will
632         complete the problem definition.
633         """
634         x,y = state
635
636         "*** YOUR CODE HERE ***"
637         goal_pos = self.food.asList()[0]
638         for food_pos in self.food.asList():
639             if util.manhattanDistance(state, food_pos) < util.manhattanDistance(state, goal_pos):
640                 goal_pos = food_pos
641         if state == goal_pos:
642             return True
643         return False
644
```

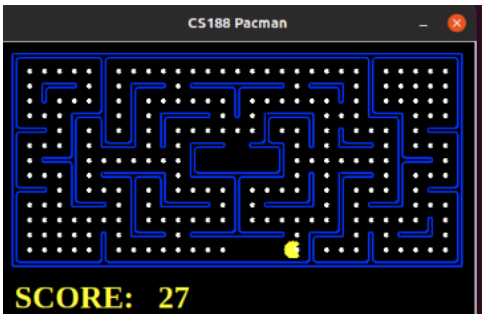
درواقع کمترین فاصله تا غذاهای باقی مانده را میابیم و آن غذا را به عنوان هدف فرضی میگیریم. اگر مکان فعلی برابر آن بود یعنی به goal رسیدیم و مقدار True را برمیگردانیم.

سپس لازم است تا تابع findPathToClosestDot در فایل serachAgents.py تکمیل کنیم. در اینجا فقط لازم است تا یکی از توابع searchای که در search.py نوشته بودیم را صدا بزنیم. البته سرچ‌ها BFSوUCS و A* همگی هزینه ۳۵۰ برای یافتن مسیر دارند ولی DFS هزینه ۵۳۲۴ را دارد. حال ما برای مثال با A* نوشته ایم:

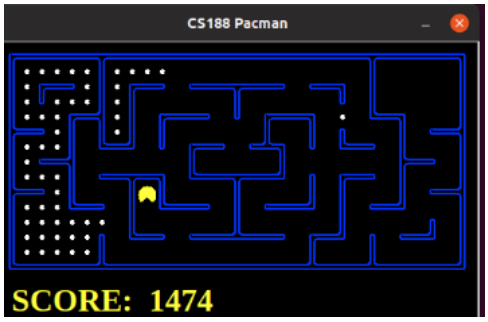
```
582
583     def findPathToClosestDot(self, gameState):
584         """
585         Returns a path (a list of actions) to the closest dot, starting from
586         gameState.
587         """
588         # Here are some useful elements of the startState
589         startPosition = gameState.getPacmanPosition()
590         food = gameState.getFood()
591         walls = gameState.getWalls()
592         problem = AnyFoodSearchProblem(gameState)
593
594         "*** YOUR CODE HERE ***"
595         return search.astar(problem) # 350
596         #other methods:
597         #return search.dfs(problem)   # 5324
598         #return search.bfs(problem)    # 350
599         #return search.ucs(problem) # 350
600         #util.raiseNotDefined()
```

نمونه‌ای از اجرا:

python3 pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

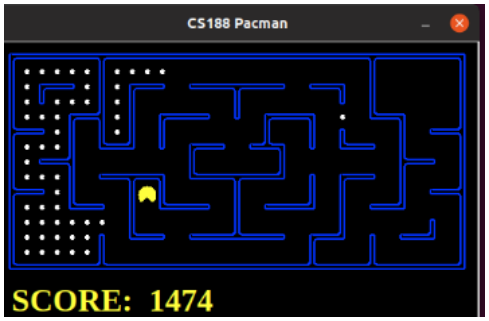



```
mahdi@OSlab:~/Desktop/AI_P1/search$ python3 pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with cost 350.
Pacman emerges victorious! Score: 2360
Average Score: 2360.0
Scores:        2360.0
Win Rate:      1/1 (1.00)
Record:        Win
mahdi@OSlab:~/Desktop/AI_P1/search$
```

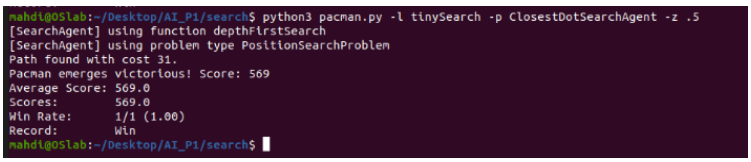


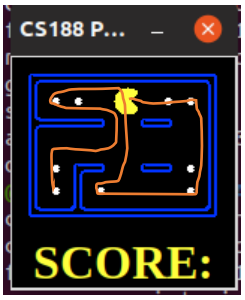
سوال ClosestDotSearchAgentشما، همیشه کوتاه ترین مسیر ممکن در ماز را پیدا نخواهد کرد. مطمئن شوید که دلیل آن را درک کرده اید و سعی کنید یک مثال کوچک بیاورید که در آن رفتن مکرر به نزدیکترین نقطه منجر به یافتن کوتاهترین مسیر برای خوردن تمام نقاط نمی شود.

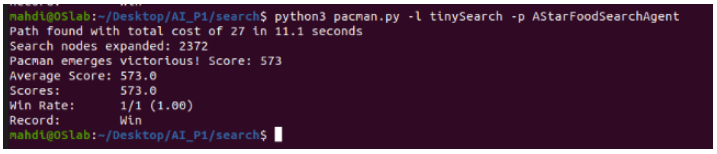
در این روش چون یک روش حریصانه را در پیش گرفتیم و تاز مانی که نقاط نزدیک تری به حالت فعلی باشد صرفا آن ها را میخوریم موجب میشود که لزوما در کل بهینه عمل نکنیم. یک مثال آن را در همین اجرای بالا دیدیم. یکی از نقاط درجایی بود که اگر پکمن میخواست آن را بخورد حتما باید مسیرش را عوض میکرد و اون موقع لزوما نزدیک ترین نقطه را نمیخورد. به همین خاطر همین طور هر بار نزدیک ترین نقاط رو خورد ولی در انتها به خاطر اون یک نقطه‌ای که قبلا میتونست با طی کردن مسیر کمتری بخورد، مسیر طولانی‌ای را برگشت که بهینه نیست:



مثال دیگری نیز میتوان گفت.







همانطور که مشاهده میشود در روش اول که نزدیک ترین نقاط رو میخورد بعضی مسیرها بیهوده مکرر طی شدند و هزینه ۳۱ شده است منتها در دومی که A* است مسیر بهینه انتخاب شده است و هزینه هم ۲۷ شده است.

نتیجه‌ی autograder به صورت زیر میباشد: