به نام خدا

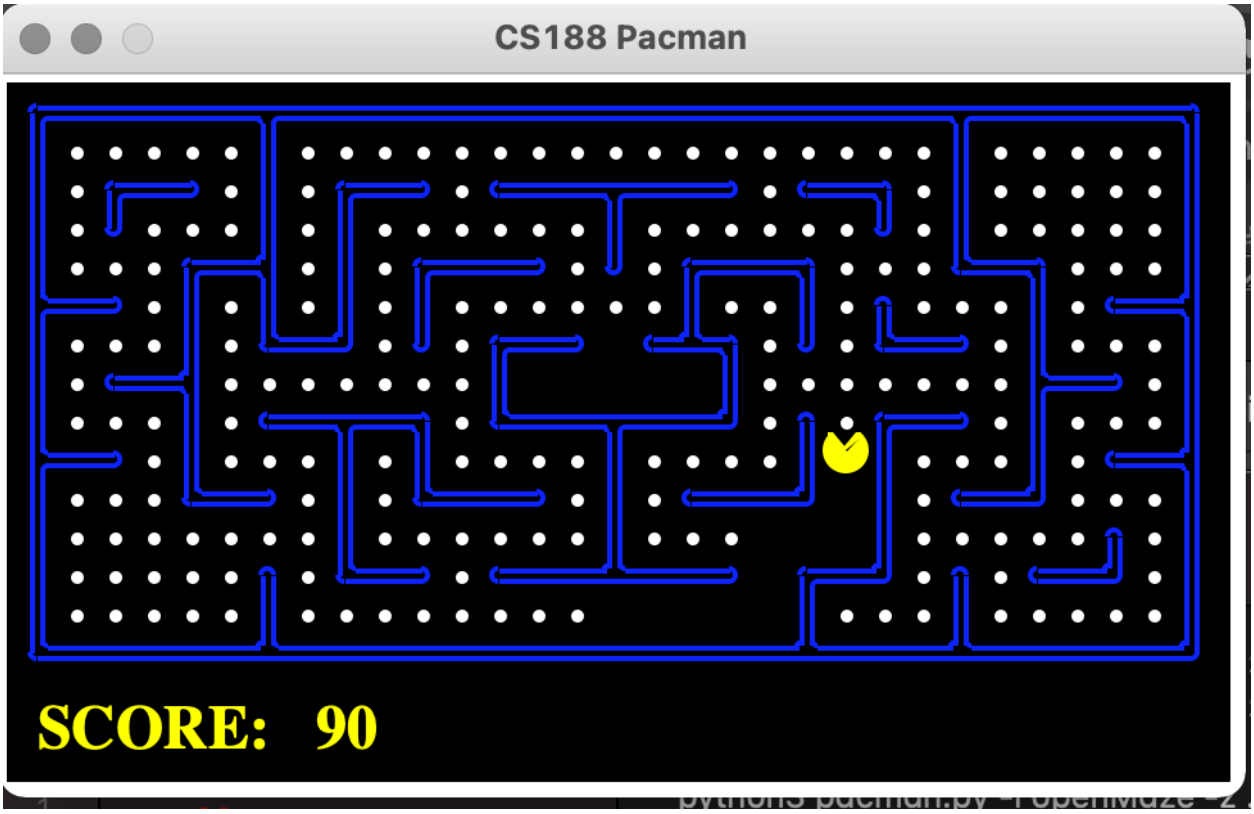
گزارش پروژه اول هوش مصنوعی

هلیاسادات هاشمی پور-۹۸۳۱۱۰۶

مقدمه

بازی پکمن یکی از معروف ترین بازی ها در سراسر جهان می باشد. جمع آوری امتیاز با خوردن نقطه های موجود در ماز، هدف از این بازی این است. در بازی تعدادی روح در ماز حرکت می کنند که در صورت برخورد با پکمن، از جان پکمن کم می شود.

حال در این پروژه ما با استفاده از الگوریتم های جستجوی متفاوت ، به برنامه ریزی مسیرها توسط پکمن کمک می کنیم. (الگوریتم های موجود DFS و BFSوUCSوASTAR هستند) واضح است که الگوریتم های جستجوی پیاده سازی شده هر کدام خصوصیات مخصوص به خود را دارند و با توجه به موقعیت های مختلف موجود در بازی ، می توانند به صورت کارآمد عمل بکنند.



- **قسمت ۱: Depth First Search**

الگوریتم DFS به این صورت کار می کند که همیشه عمیق ترین گره را expand می کند. در کد زده شده از پشته استفاده شده است. در اصل برای اینکه آن گره ای که اخیرا درست شده است برای expand کردن انتخاب میشود. (خاصیت LIFO)

جدول نتیجه ی الگوریتمDFS

Maze Form	total cost	Search nodes expanded	Score
tinyMaze	10	15	500
mediumMaze	130	146	380
bigMaze	210	390	300

سوال:

آیا ترتیب کاوش همان ترتیبی بود که انتظار میرفت؟ آیا پکمن در راه رسیدن به هدف، به همه مربع های کاوش شده میرود؟ آیا این راحل کم ترین هزینه را دارد؟ اگر نه فکر کنید که جستجوی اول عمق چه کاری انجام میدهد.

در اصل در این الگوریتم ، پکمن نیازی به رفتن به همه ی مربع در راه رسیدن به آزمون هدف ندارد. در این الگوریتم گره ای که expand شده است، آن ترتیبی که انتظار داشتم هست. این راه حل متاسفانه کم ترین هزینه را ندارد و همانطور که گفتیم بدی این الگوریتم این است که عمیق ترین گره را expand می کند و راه حل مطلوب (optimal)را به ما نمی دهد؛ در واقع کار آن به صورت کاوش گره ها به صورت عمقی به روش LIFO است. گاهی اوقات با expand کردن به شکل عمقی ممکن است استیت نهایی (همان جواب نهایی)به ما داده نشود و مسیر، هزینه ی مطلوبی نداشته باشد.(زیرا سمت چپ ترین(leftmost) راه حل را به ما میدهد) در اصل بهینه بودن هزینه ی مسیر در این الگوریتم مرتبط به نقشه ی بازی می باشد. به عبارتی گسترش حالات مشاهده شده را می توان نقص این الگوریتم دانست. به دنبال آن در الگوریتم های قسمت های بعدی، با در نظر گرفتن پارامترهای دیگر(مثل هزینه و ...) می توان راه حل مربوطه را بهتر کرد.

از لینک زیر برای درک بهتر این الگوریتم در جهت زدن کد مربوطه کمک گرفته ام.

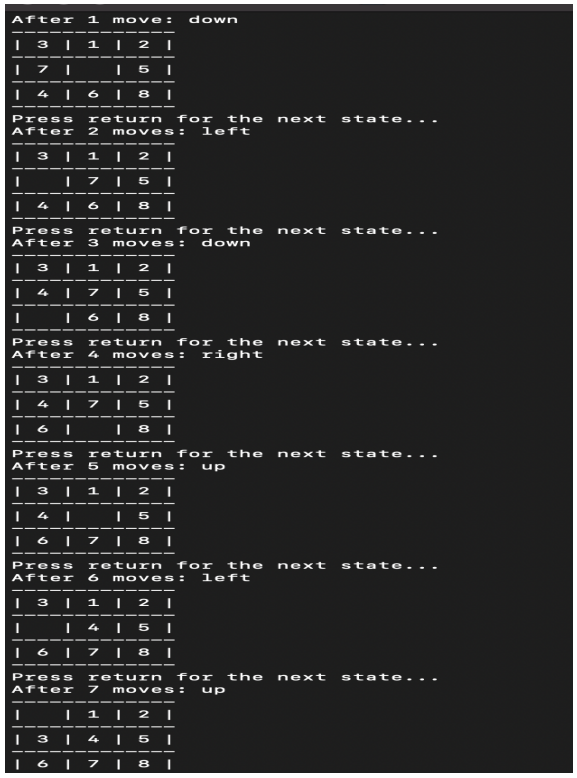
https://www.programiz.com/dsa/graph-dfs

- **قسمت ۲: Breadth First Search**

الگوریتم جستجویBFS به صورت سطح به سطح expand می شود. به این شکل که ابتدا root گره ها expand شده سپس به صورت مرحله به مرحله جستجو صورت می گیرد. در این روش، برای fringe من از Queue استفاده کرده ام. (به دلیل خاصیت FIFO) این الگوریتم جستجو، optimal است؛ زیرا به این صورت عمل می کند که آن گره ای که اول از همه دیده شده اس باید زودتر گسترش یابد.

جدول نتیجه ی الگوریتمBFS

Maze Form	total cost	Search nodes expanded	Score
tinyMaze	8	15	502
mediumMaze	68	269	442
bigMaze	210	620	300



درالگوریتم جستجوی BFS ای که نوشته ام با توجه به دستوری که در دستور کار آمده است ، مسئله ۸–پازلی که به شکل رندوم تولید می شوند ، را حل می کند.(در اینجا ۷ حرکت پازل حل شده است)

از لینک زیر برای درک بهتر این الگوریتم در جهت زدن کد مربوطه کمک گرفته ام.

https://www.programiz.com/dsa/graph-bfs

- **قسمت ۳: Uniform Cost Search**

در الگوریتم جستجو گره ای که کم ترین هزینه ی مسیر را دارد را expand می کنیم. در کد نوشته شده برای انجام این الگوریتم ،fringe را به صورت priority queue تعریف کرده ام(که شامل هزینه ی مسیر رسیدن به استیت هدف، است) تا هر بار آن گره ای که کم ترین هزینه را دارد، pop شود. این الگوریتم complete و optimal است.

جدول نتیجه ی الگوریتمUCS

Maze Form	total cost	Search nodes expanded	Score
mediumMaze	68	269	442

جدول نتیجه ی الگوریتمUCS با استفاده از StayEastSearchAgent

Maze Form	total cost	Search nodes expanded	Score
mediumDottedMaze	1	186	646
mediumScaryMaze	1	230	-503

جدول نتیجه ی الگوریتمUCS با استفاده از StayWestSearchAgent

Maze Form	total cost	Search nodes expanded	Score
mediumDottedMaze	17183894840	169	-
mediumScaryMaze	68719479864	108	418

با توجه به دستورکار ما هزینه ی بسیار پایین و بسیار بالابرای StayEastSearchAgent و StayWestSearchAgent به دلیل تابع هزینه ی نمایی داریم. هزینه ی ارائه شده برای StayWestSearchAgent ،تابع هزینه ی آن به صورت $g=2x$ برای مختصات (x,y) است. از طرفی تابع هزینه StayEastSearchAgent به صورت $g=0.5x$ برای مختصات (x,y) است (باعث می شود هزینه مسیر نسبتا کم باشد.)

از لینک زیر برای درک بهتر این الگوریتم در جهت زدن کد مربوطه کمک گرفته ام.

https://www.educative.io/edpresso/what-is-uniform-cost-search

- **قسمت ۴: A* search**

A^* گره ها را توسط هزینه ی رسیدن به گره ($g(n)$) (در اصل هزینه ی رسیدن از گره ی شروع به n است) و هزینه ی به رسیدن به گره ی هدف ($h(n)$) (در واقع تخمین کم ترین هزینه راه، از n به هدف می باشد توسط تابع هیوریستیک) به صورت زیر است:

$$F(n)=g(n)+h(n)$$

$F(n)$ تخمین کم هزینه ترین راه حل تا به n می باشد.

بنابراین ما باید کم هزینه ترین را با توجه به فرمول بالا در هر مرحله انتخاب کنیم. الگوریتم A^* تقریبا مانند UCS است با این تفاوت که A^* یک $h(n)$ اضافه تر دارد. در اصل این الگوریتم معقول تر است؛ زیرا شامل تابع هیستوریک ($h(n)$) است و این الگوریتم هم complete و هم optimal می باشد.

برای اینکه ثابت کنیم الگوریتم A^* جواب بهینه را تاحدی سریع تر از UCS پیدا می کند می توان پیاده سازی های خود را بر روی مسئله ی پیدا کردن ماز به نقطه ای مشخص به کمک هیستوریک Manhattan distance تست کنیم. بنابراین جدول زیر را داریم:

مقایسه ی الگوریتم A^* با UCS در bigMaze

Algorithm	Total Cost	Search nodes expanded
A^*	210	549

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.

جدول زیر تمام الگوریتم های جستجوی پیاده سازی شده را با هم مقایسه میکند.

نتیجه ی openMaze

Algorithm	total cost	Search nodes expanded	Score
DFS	298	576	212
BFS	54	682	456
UCS	54	682	456
A*	54	**535**	456

با توجه به جدول بالا، روشن است که A* عملکرد بهتری نسبت به دیگر الگوریتم های جستجو دارد. در اصل total cost آن با الگوریتم های UCS وBFS برابر است. چون این الگوریتم برای بسط دادن کم تر، جهت رسیدن به راه حل مطلوب، از هیوریستیک Manhattan distance استفاده می کند (که در این روش دیوار ها در نظر گرفته نشده و فاصله ی منهتنی را به صورت h(n)در نظر می گیریم بنابراین دیگر نیاز به پیمایش تمامی خانه ها نداریم.)؛ بنابراین همان طور که می بینیم تعداد گره هایی که expand شده است؛ از دیگر الگوریتم ها کم تر است. به طور کل الگوریتم UCS در اینجا هزینه را برابر یک در نظر گرفته ایم بنابراین با الگوریتم BFS تفاوتی ندارد و این الگوریتم آگاهانه نمی باشد. الگوریتم A* با nullheuristi چون h(n) برابر صفر می شود تفاوتی با UCS نداشته و ناآگاهانه عمل می کند.

از لینک زیر برای درک بهتر این الگوریتم در جهت زدن کد مربوطه کمک گرفته ام.

https://towardsdatascience.com/a-star-a-search-algorithm-eb495fb156bb

- **قسمت ۵: Finding All the Corners**

در این قسمت از پروژه قدرت واقعی جستجوی الگوریتم A* در برابر چالش های مسئله های جستجو را به ما نشان داده می شود. برای این منظور کلاس CornersProblem را که هدف آن، پیدا کردن کوتاه ترین مسیر در ماز، به شکلی که از هر چهار کناره عبور بکند، را کامل می کنیم.

توابع پیاده سازی شد:

- **getStartState()** : به ما استیت شروع پکمن و کناره هایی که باید ببینیم را می دهد.
- **isGoalState()** : اگر چهار کناره ای که در بازی داریم پیدا شوند به ما true بر می گرداند.
- **getSuccessor()**: successor های مربوطه که شامل استیت آن ها و action و هزینه ی یک را بر میگرداند.

نتیجه ی CornersProblem با استفاده از bfs

Maze Form	Total Cost	Search nodes expanded	Score
tinyCorners	28	252	512
mediumCorners	106	1966	434
bigCorners	162	7949	378

- **قسمت ۶ : Corners Problem-Heuristic**

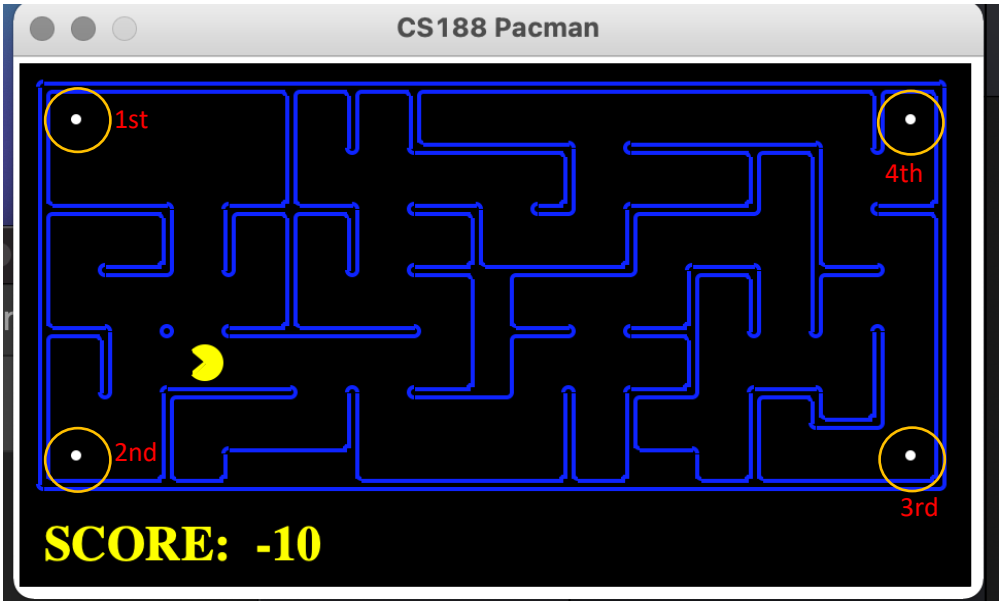
سوال:

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

در این قسمت یک هیوریستیک غیر بدیهی سازگار برای CornersProblem پیاده سازی کرده ام .چون مسئله ی ما به صورت جستجوی گرافی می باشد ما باید یک هیوریستیک به صورت consistent طراحی کنیم. در کد نوشته شده ما با استفاده از Manhattan distanceدو موقعیت را بدون در نظر گرفتن دیواره های موجود در نقشه ی بازی استفاده می کنیم. که این تابع به ما فاصله ی دو گره را می دهد. با توجه به کد شرط ما این است که اگر ما به کناره ای رسیده باشیم، ما نیاز داریم که نقشه را پیمایش کرده تا همه ی کناره هایی که ندیده ایم را پیمایش بکنیم. واضح است طبق شکل زیر ترتیب حرکت پکمن از کناره ها نشان داده شده است. در این قسمت به جای اینکه موقعیت کناره ها را در یابیم، تعداد کناره هایی که دیده نشده اند برایمان اهمیت دارد. حال با استفاده از Manhattan Distance فاصله ی بین موقعیت فعلی و کناره ی دیده نشده را به دست آورده و کم ترین مقدار آن را به عنوان heuristic_val نگه می داریم. سپس موقعیت فعلی خود را به کناره دیده نشده تغییر می دهیم و این وضعیت در یک لوپ ادامه دارد تا زمانی که تمامی چهار گوشه دیده شوند. بنابراین با توجه به توضیحات داده شده در می یابیم که تابع هیوریستیک پیاده سازی شده ی ما admissible است و چون هزینه ی(cost) آن توسطManhattan distance محاسبه می شود بنابراین آن consistent می باشد.

واضح است که دو شرط سازگار بودن عبارت است از:

$h(n)<=c(n,p)+h(p)$ و $h(g)=0$



گره های والد و فرزند به دلیل اینکه در مجاورت هم قرار دارند cost آن ها برابر یک می شود حال می دانیم دو گره ی ما نزدیک ترین کناره به آن ها با توجه فاصله منهتن ، نزدیک ترین ناحیه را انتخاب می کنیم(به صورتی عمل می کند که از گره شروع ابتدا نزدیک ترین کناره را پیدا می کند؛ سپس با توجه به فاصله منهتنی نزدیک ترین کناره را انتخاب می کند در واقع از گوشه به گوشه ی دیگر حرکت می کنیم) . مسیری که هیوریستیک دو گره ی والد و فرزند توسط فاصله ی منهتنی طی می کند تنها در اولین کناره با هم متفاوت اند بنابراین هزینه ی آن ها هم در اولین کناره متفاوت است. اما باقی آن ها چون بر آن کناره می افتد، با رفتن از یک کناره به کناره ی دیگر، دیگر مسیرهای یکسانی دارند. بنابراین برای حساب کردن تفاوت هیوریستیک دو تا والد و فرزند کافی است. تفاوت آن ها را از استیت اولیه به اولین کناره ببینیم چگونه است. چون از بعد از کناره اول هیوریستیک های فاصله ی منهتن یکسان می شود، بنابراین باید تفاوت هیوریستیک ها را در بخش اول آن ها حساب کنیم یعنی از دو گره والد و فرزند به سمت کناره اول برود. با توجه به اینکه از فاصله ی منهتن استفاده کرده ایم بنابراین دیوارها در نظر گرفته نمی شود و چون در دو گره ی مجاور هستیم بنابراین فاصله ی منهتنی ما یک خانه فرق دارد بنابراین هیوریستیکشان یک واحد اختلاف دارند و هزینه ی واقعی آن ها به اندازه ی یک واحد تفاوت دارد. پس هیوریستیک ما کوچکتر مساوی هزینه واقعی می شود. بنابراین شرط نخست ثابت می شود.

برای شرط بعدی با توجه به کد ما که براساس کناره هایی که دیده نشده بودند در یک لوپ بودیم تا زمانی که تمام کناره ها دیده شوند(در اصل نزدیک ترین گوشه پیدا می شد سپس به ترتیب دیگر گوشه ها را می دیدیم) در اصل هنگامی به استیت هدف می رسیم که ما هر چهار کناره را دیده باشیم یعنی لیست کناره ها ی دیده شده صفر می شود در نتیجه هنگامی که به استیت هدف میرسیم به این معنی است که چهار کناره را طی کرده پس لیستی که داریم طول آن صفر شده و وارد لوپ نمی شویم و هزینه ی ما همان هزینه ی اولیه که صفر بود می ماند پس شرط دوم هم ثابت می شود. بنابراین سازگاری آن ثابت می شود.

```python
# initialize
currNode = state[0]
unvisCorners = list(state[1])

if len(unvisCorners) == 0:
    return 0

heuristic_val = 0
# find the nearest corner
while len(unvisCorners):
    lowestHeuristicDistance = 9999
    for corner in range(0, len(unvisCorners)):
        # calculate the manhattan distance from the current node to the one of the corners
        # then find the lowest distance and corner
        if util.manhattanDistance(currNode, unvisCorners[corner]) < lowestHeuristicDistance:
            currentCorner = unvisCorners[corner]
            # set distance to the nearest corner
            lowestHeuristicDistance = util.manhattanDistance(currNode, unvisCorners[corner])
    # update the current node to the nearest corner
    currNode = currentCorner
    # adding the lowest distance to the value
    heuristic_val += lowestHeuristicDistance
    # removing corner from the list(un visited corners)
    unvisCorners.remove(currentCorner)

return heuristic_val
```

نتیجهCornersProblem برای mediumCorners

Maze Form	Total Cost	nodes expanded	Score
mediumCorners	106	**692**	434

از طرفی اگر UCSو A* مسیرهایی با طول ای برابر برگردانند هیورستیک سازگار است.

```
[heliaa@MacBook-Pro search % python3 pacman.py -l mediumCorners -p  SearchAgent -]
a fn=ucs,prob=CornersProblem,heuristic=cornersHeuristic -z 0.5
[SearchAgent] using function ucs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.0 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
[heliaa@MacBook-Pro search % python3 pacman.py -l mediumCorners -p  SearchAgent -]
a fn=astar,prob=CornersProblem,heuristic=cornersHeuristic -z 0.5
[SearchAgent] using function astar and heuristic cornersHeuristic
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.0 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
```

- **قسمت ۷: Eating All The Dots**

سوال:

هیوریستیک خود را توضیح دهید و سازگازی آن را استدلال کنید.

در این قسمت پکمن باید همه ی نقطه های موجود را بخورد به جای اینکه(در قسمت پیشین) فقط از کناره ها عبور کند. به جای استفاده از Manhattan distance در قسمت قبل این بار از تابعی استفاده می کنیم(یکی از نقص های Manhattan distance این بود که در این قسمت اگر تعداد زیادی نقطه داشته باشیم در دنبال کردن کناره ها به مشکل خواهیم خورد چون در این تابع دیوارها را در نظر نمی گیریم). بنابراین از یک تابعی به اسم MazeDistance() استفاده می کنیم(این تابع با استفاده از bfs کار می کند)؛ که فاصله ی دو نقطه را به ما برمی گرداند؛(با توجه به صف اولویت، آخر از همه خارج شود که باعث می شود به مرور(f(n زیاد شود) این بار بیشترین فاصله را در نظر می گیریم.(سپس بعد از خورده شدن نقطه ها به استیت هدف می رسیم. (در اصل می توان به این شکل توضیح داد که چون ما باید به دورترین نقطه برای خوردن همه ی نقاط میرویم پس ما کم ترین actual cost را زمانی که دیگر نقطه ها در راه دورترین نقطه هستند ،را داریم. در یک شرایط دیگر هم actual cost از این مقدار بیشتر می شود، زیرا ما باید از این مسیر برای خوردن دورترین نقطه استفاده کنیم بنابراین سازگار است.) البته می توانستیم به جای تابع MazeDistance() از استفاده بکنیم ManhattanDistance() اما تعداد گره هایی که در این الگوریتم هیوریستیک باز می شد9551 تا بود (نمره ی کامل از این بخش دریافت می شد) اما با توجه به این که می توان الگوریتم را بهتر کرد و تعداد گره های باز شده را به 4137تا رساند از MazeDistance() استفاده کرده ایم.

دو شرط سازگار بودن عبارت است از:

$h(n)<=c(n,p)+h(p)$ و $h(g)=0$

هنگامی که ما در استیت هدف هستیم در اصل در لیست ما نقطه ای برای خوردن توسط پکمن وجود ندارد؛ بنابراین شرط دوم برقرار می شود. هیوریستیک ما در این بخش خانه هایی که نقطه ای برای خوردن وجود دارند را می گیرد و بررسی می کند، دورترین نقطه ای که می شود رفت را طول مسیرش را به عنوان هیوریستیک به ما می دهد .این مسیر پیمایش شده که توسط bfs پیمایش می کند، در واقع ما ساده سازی در این سوال نداریم(دیوار ها را در نظر می گیریم) حال می دانیم در bfs اگر هزینه ی همه ی مسیرها برابر یک باشد بهینه می باشد.(سطح به سطح بررسی می کند)پس کاری که الگوریتم ما می کند این است که طول کوتاه ترین مسیر به دورترین خانه ی غذا را به ما به عنوان هیوریستیک می دهد. دورترین خانه غذا برای گره ی والدمان همان دورترین گره ای است که برای گره ی فرزند بوده یا اگر آن نباشد؛ خانه ی دیگری است که هیوریستیک آن برابر می شود با هیوریستیک گره ی فرزند. در حالت اول هیوریستیک تفریق والد و فرزند برابر یک می باشد چون MazeDistance بر اساس bfs که به صورت سطح به سطح هست کار می کند پس برای رسیدن به خانه ی مورد نظر ما دوباره از خانه ی فرزند میگذرد که هیوریستیک والد منهای فرزند برابر با یک می شود و حالت دیگر هم هیوریستیک ها باید برابر باشند که تفاضل أن ها صفر می شود که صفر کوچک تر از هزینه ی واقعی است پس دو حالت سازگاری اثبات شد.

```python
position, foodGrid = state
"*** YOUR CODE HERE ***"
# make the list of the x,y and of all of the food
uneatenList = foodGrid.asList()

if len(uneatenList) == 0:
    return 0

# the farthest food from the current node
result = []
for food in uneatenList:
    # calculate the maze distance
    # distance = (mazeDistance(position, i, problem.startingGameState))
    # find the farthest and then update the result
    result.append(mazeDistance(position, food, problem.startingGameState))

return max(result)
```

نتیجه Food Problem با استفاده از تابع MazeDistance()

Maze Form	Total Cost	nodes expanded	Score
trickySearch	60	**4137**	570

نتیجه Food Problem با استفاده از تابع ManhattanDistance()

Maze Form	Total Cost	nodes expanded	Score
trickySearch	60	9551	570

- **قسمت ۸: Suboptimal Search**

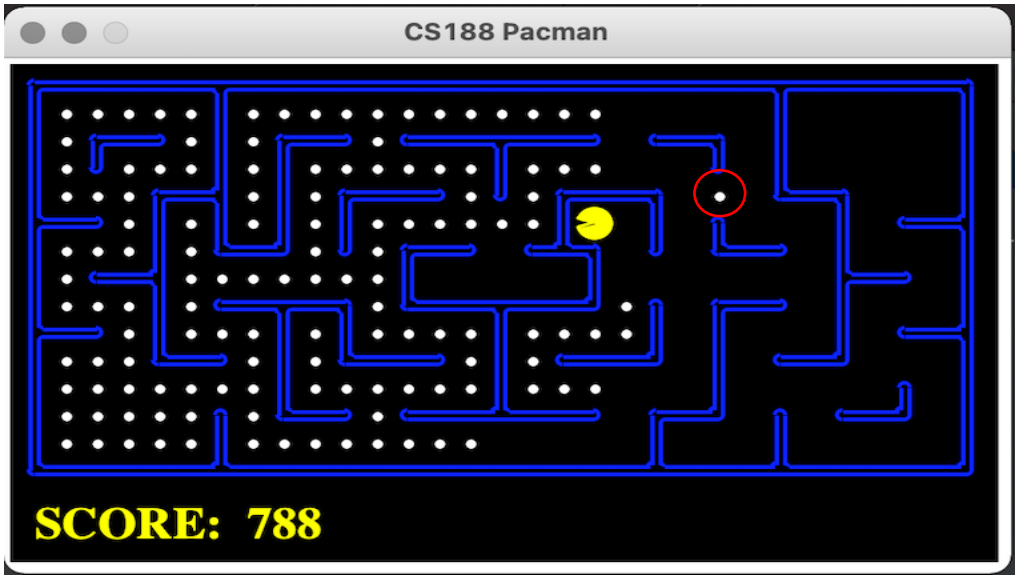
سوال:

ClosestDotSearchAgent شما همیشه کوتاه ترین مسیر ممکن در ماز را انتخاب نخواهد کرد . مطمئن شوید که دلیل آن را درک کرده اید و سعی کنید یک مثال کوچک بیاورید که در آن رفتن مکرر به نزدیک ترین نقطه منجر به یافتن کوتاهترین مسیر برای خوردن تمام نقطه ها نمی شود.

این قسمت این را به ما نشان می دهد که حتی یک هیوریستیک مناسب هم پیدا کردن یک مسیر مناسب از میان تمام نقطه ها سخت می شود. بنابراین، دنبال یک راه خوب هستیم. یک عاملی که نوشتیم این هست که همیشه نقطه هایی که نزدیک به پکمن هستند خورده شوند.(به شکل حریصانه) جدول زیر نتیجه ی راه بالا برای الگوریتم های جستجوی مختلف است:

Algorithm	total cost	Score
DFS	5324	-2614
BFS	350	2360
UCS	350	2360
A^*	350	2360

همانطور که می بینیم الگوریتم ما چون به شکل جستجوی حریصانه هست optimal نیست وsuboptimal می باشد. در اصل اینکه همیشه کوتاه ترین مسیر. ممکن ماز توسط ClosestDotSearchAgent انتخاب نمی شود را می توان در شکل زیر دید. در شکل زیر مثالی را می بینیم که رفتن مکرر به نزدیک ترین نقطه منجر به یافتن کوتاهترین مسیر برای خوردن تمام نقطه ها نمی شود.



با توجه به شکل نقطه ی مشخص شده می ماند و پس از طی کردن مسافتی باز برمیگردد تا نقطه ی مشخص شده را بخورد و چون باید برای خوردن این نقطه برگردد نشان از این است منجر به یافتن کوتاه ترین مسیر نمی شود.

با توجه به جدولی که نوشته ام به شکل دیگر هم می توان نشان داد برای الگوریتم DFS حتی با وجود خوردن همه نقطه ها توسط عامل، باعث ایجاد امتیاز منفی شده است. واضح است این ClosestDotSearchAgent همیشه راه حل optimal را برنمی گرداند.

مثال دیگر الگوریتم حریصانه نزدیک ترین جوابی را که می بیند را اجرا می کند بدون اینکه به آینده توجهی داشته باشد. مثل عبور از بین شهر ها توسط این الگوریتم که اگر راه دیگری را پیش می گرفتیم مسیر کوتاه تر و بهینه تر می شد.