

کلاس `SearchProblem` یک کلاس `abstract` است که باید پیاده‌سازی شود. این کلاس در واقع ساختار یک مساله جستجو را تعیین می‌کند. این مساله جستجو شامل فضای حالات، حالت شروع، حالت هدف، تابع پسین و تابع هزینه است.

متدها: `getStartState` حالت شروع مساله را برمی‌گرداند. `isGoalState` یک حالت می‌گیرد و مشخص می‌کند که آن حالت، حالت هدف است یا خیر. `getSuccessors` یک حالت می‌گیرد و لیستی از بردارهای سه بعدی بر می‌گردد که هر بردار شامل: یک حالت پسین از حالت فعلی، عمل (action) مورد نیاز برای رسیدن به حالت پسین و هزینه آن است. `getCostOfActions` لیستی از Action ها می‌گیرد و مجموع هزینه مورد نیاز برای این دنباله از Action ها را برمی‌گرداند.

کلاس Actions شامل تعدادی متد استاتیک است که وظیفه ایجاد تغییرات یا بررسی عمل‌های حرکتی را دارند به عنوان مثال متد reverseDirection یک action می‌گیرد و جهت برعکس آن را برمی‌گرداند. یا متد vectorToDirection یک بردار می‌گیرد و جهت متناظر با آن را برمی‌گرداند. کلاس Configuration در واقع نگه دارنده موقعیت مکانی به همراه جهت حرکت یک کاراکتر در بازی است. این کلاس علاوه بر متدهای getter یک متد generateSuccessor هم دارد که با گرفتن یک پر دار حالت پسین متناظر را تولید می‌کند.

کلاس Directions شامل جهت‌های حرکت بازی است به علاوه تعریف جهت‌های چپ و راست برای هر جهت حرکت به همراه معکوس جهت‌ها.

کلاس Agent یک کلاس Abstract که دارای یک متد `getAction` است. این متد یک state از بازی را می‌گیرد و یک action متناظر با رفتاری که Agent مورد نظر ما دارد تولید کرده و خروجی `action` را دریافت می‌کند.

کلاس Grid در واقع شامل یک آرایه دو بعدی به نام data است که در واقع نقشه بازی را شامل می شود و همچنین شامل متدهایی برای استفاده از این data و یا تغیر آن است.

کلاس **AgentState** شامل حالت کامل یک Agent می‌شود که علاوه بر Configuration مقداری مانند سرعت را هم دارد و متدهایی برای خروجی دادن آن‌ها دارد.

```
for index, (p, c, i) in enumerate(self.heap):
    if i == item:
        if p <= priority:
            break
        del self.heap[index]
        self.heap.append((priority, c, item))
        heapq.heapify(self.heap)
        break
    else:
        self.push(item, priority)
```

در متدهای update و push برای PriorityQueue باید برسی کنیم که item مورد نظر در صف وجود دارد یا خیر و اگر وجود نداشت push کنیم و اگر وجود داشته باشد با توجه به اولویت آن تصمیم می‌گیریم که روی آن تغییری ایجاد کنیم یا نه.

```

fringe = util.Stack()
visited = []
actions = []

fringe.push((problem.getStartState(), actions))
while not fringe.isEmpty():
    state, actions = fringe.pop() # pop the last element because fringe in DFS is LIFO
    if problem.isGoalState(state):
        return actions
    if state not in visited:
        visited.append(state)
        for successor, action, cost in problem.getSuccessors(state):
            fringe.push((successor, actions + [action]))
return []

```

چون DFS گرافی انجام می‌دهیم باید یک لیست `visited` داشته باشیم و همچنین `fringe` در این الگوریتم یک `Stack` است. در ابتدای کار فقط حالت شروع در `fringe` وجود دارد و سپس با توجه به ویژگی `LIFO` است شروع به برداشتن یک نود از `fringe` و بسط دادن آن می‌کنیم. البته ابتدا چک می‌کنیم که آیا نود مورد نظر هدف است یا آیا نود مورد نظر را قبلاً برسی کردیم یا نه.

```

farhad@farhad-Aspire ~ ~/Desktop/Pacman-AI-project/P1 ↵ main ± ➔ 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      return []
Average Score: 500.0
Scores: 500.0
Win Rate: 1/1 (1.00)
Record: Win
farhad@farhad-Aspire ~ ~/Desktop/Pacman-AI-project/P1 ↵ main ± ➔ 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      util.raiseNotDefined()
Average Score: 380.0
Scores: 380.0
Win Rate: 1/1 (1.00)
Record: Win
farhad@farhad-Aspire ~ ~/Desktop/Pacman-AI-project/P1 ↵ main ± ➔ python3 pacman.py -l bigMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300      util.raiseNotDefined()
Average Score: 300.0
Scores: 300.0
Win Rate: 1/1 (1.00)
Record: Win

```

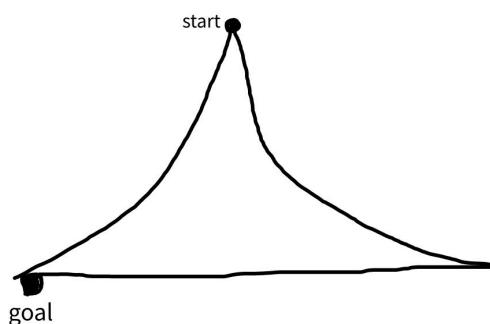
در اینجا هم نتیجه کد زده شده برای `layout` های `tinyMaze`, `mediumMaze`, `bigMaze` را مشاهده می‌کنیم.
همان‌طور که می‌دانیم در یک گراف با تعداد V راس و E یال پیچیدگی DFS برابر $O(E + V)$

است. چرا که DFS باید هر یال و هر راس را حداقل یکبار پیمایش کند تا مطمئن باشیم که به جواب رسیده‌ایم. البته پیچیدگی DFS در هنگام جستجوی درختی متفاوت است و با حالت گرافی تفاوت دارد. استفاده از DFS بهینه نیست و عامل رفتار منطقی ندارد. چرا که DFS به صورت عمق اول عمل می‌کند و هر مسیر را تا انتهای بیشترین عمق پیمایش می‌کند در نتیجه تضمینی برای اینکه مسیر منتهی به جواب مسیر بهینه باشد وجود ندارد.

در الگوریتم IDS در هر مرحله یک عمق را مشخص می‌کنیم و الگوریتم DFS را تا آن عمق اجرا می‌کنیم سپس در مراحل بعد این عمق را افزایش می‌دهیم.

```
def ids(node, depth):
    if depth == 0:
        if node == goal:
            return (node, true)
        else:
            return (null, true)
    else:
        is_remains = false
        for child successor(node):
            answer, remaining = ids(child, depth - 1)
            if answer != null:
                return (found, true)
            if remaining > 0:
                is_remains = true
        return (null, is_remains)
```

یک مثال برای عملکرد بهتر BFS مانند همان مثالی برای عملکرد بهتر DFS نسبت به BFS است. فرض کنید یک درخت بسیار عمیق داریم. نود هدف ما یکی از برگ‌های این درخت است و برگ سمت چپ هم هست. در IDS باید تمام عمق‌های درخت جستجوی کامل شوند تا به انتهای درخت برسیم و نود هدف پیدا شود. در حالیکه DFS همان ابتدا تا انتهای درخت را پیمایش می‌کند و نود هدف را می‌یابد در نتیجه بسیار سریع‌تر به جواب خواهد رسید.



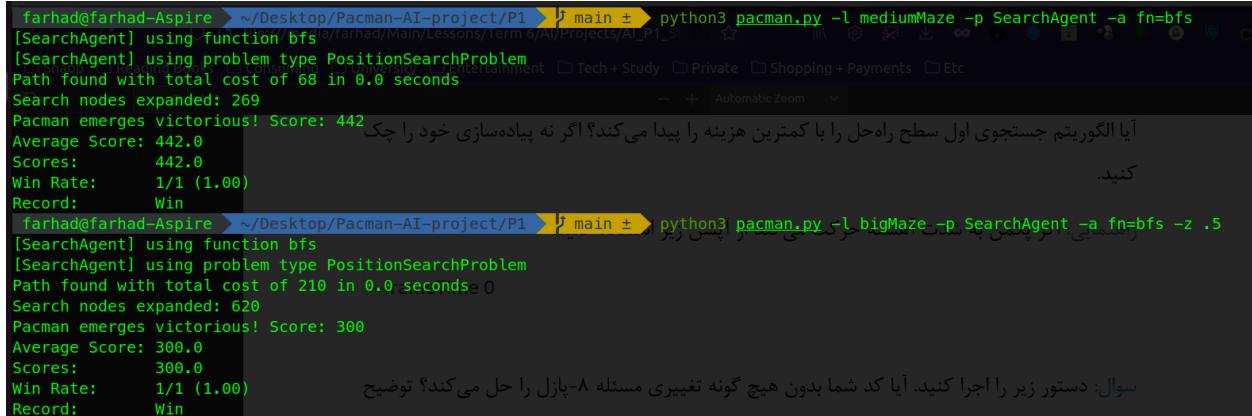
```

fringe = util.Queue()
visited = []
actions = []

fringe.push((problem.getStartState(), actions))
while not fringe.isEmpty():
    state, actions = fringe.pop() # pop the first element because fringe in BFS is FIFO
    if problem.isGoalState(state):
        return actions
    if state not in visited:
        visited.append(state)
        for successor, action, cost in problem.getSuccessors(state):
            fringe.push((successor, actions + [action]))
return []

```

تنها تفاوتی که کد DFS با BFS دارد این است که آن به جای یک Stack باید یک Queue باشد. که این مورد را در util پیاده‌سازی کرده ایم. در نتیجه الگوریتم به درستی عمل خواهد کرد.



```

farhad@farhad-Aspire:~/Desktop/Pacman-AI-project/P1$ 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 0.0 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
farhad@farhad-Aspire:~/Desktop/Pacman-AI-project/P1$ 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 0.0 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

```

آیا الگوریتم جستجوی اول سطح راه حل را با کمترین هزینه را پیدا می کند؟ اگر نه پیاده‌سازی خود را چک کنید.

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

همانطور که می‌بینیم الگوریتم برای دو حالت تست به درستی عمل کرده. از طرفی ما الگوریتم BFS را به صورت کلی به عنوان یک الگوریتم جستجو نوشته‌ایم در نتیجه این الگوریتم برای مسائل جستجوی دیگری مانند مساله eightpuzzle هم به درستی عمل خواهد کرد که با اجرای آن مشاهده می‌شود.

الگوریتم BBFS به جای اینکه جستجو را از راس مبدا شروع کند و آنقدر ادامه دهد تا به مقصد برسد جستجو را از راس مبدا و مقصد به صورت موازی و همزمان شروع می‌کند و آنقدر کار را ادامه می‌دهد تا دو مسیر در میانه راه به یکدیگر بررسند.

```
def bi_directional_search(graph, start, goal):
    if start == goal:
        return [start]
    active_vertices_path_dict = {start: [start], goal: [goal]}
    inactive_vertices = set()
    while len(active_vertices_path_dict) > 0:
        active_vertices = list(active_vertices_path_dict.keys())
        for vertex in active_vertices:
            current_path = active_vertices_path_dict[vertex]
            origin = current_path[0]
            current_neighbours = set(graph[vertex]) - inactive_vertices
            if len(current_neighbours.intersection(active_vertices)) > 0:
                for meeting_vertex in current_neighbours.intersection(active_vertices):
                    if origin != active_vertices_path_dict[meeting_vertex][0]:
```

```

        active_vertices_path_dict[meeting_vertex].reverse()
        return active_vertices_path_dict[vertex] +
active_vertices_path_dict[meeting_vertex]
    if len(set(current_neighbours) - inactive_vertices - set(active_vertices)) == 0:
        active_vertices_path_dict.pop(vertex, None)
        inactive_vertices.add(vertex)
    else:
        for neighbour_vertex in current_neighbours - inactive_vertices -
set(active_vertices):
            active_vertices_path_dict[neighbour_vertex] = current_path +
[neighbour_vertex]
            active_vertices.append(neighbour_vertex)
            active_vertices_path_dict.pop(vertex, None)
            inactive_vertices.add(vertex)
return None

```

پیچیدگی زمانی و مکانی الگوریتم BBFS برابر

$O(b^d/2)$

است در حالیکه این مقادیر برای BFS عادی برابر

$O(b^d)$

است. همچنین این الگوریتم کامل و بهینه است.

هر دو الگوریتم DFS و BFS از نظر پیچیدگی زمانی

$O(b^d)$

هستند که b ضریب انشعاب و d حداقل عمق هدف است.

اما در پیچیدگی مکانی برای BFS همان

$O(b^d)$

است در حالیکه در DFS این مقدار برابر

$O(bd)$

هر دو الگوریتم کامل هستند و اگر عمق محدود باشد جواب را در صورت وجود می‌یابند. اما اگر همه هزینه‌ها یکسان باشد بهینه هم هست در حالیکه DFS این چنین نیست. در نتیجه اگر بهینه بودن مد نظر باشد باید از BFS استفاده کنیم. اما از طرفی DFS می‌تواند از نظر فضای حافظه و همچنین تعداد node هایی که بسط می‌دهد با توجه به نتایج کسب شده بهتر باشد.

```

fringe = util.PriorityQueue()
visited = []
actions = []

fringe.push((problem.getStartState(), actions), 0)
while not fringe.isEmpty():
    state, actions = fringe.pop() # pop the element with the lowest priority
    if problem.isGoalState(state):
        return actions
    if state not in visited:
        visited.append(state)
        for successor, action, cost in problem.getSuccessors(state):
            fringe.push((successor, actions + [action]), problem.getCostOfActions(actions + [action]))
return []

```

همانطور که می‌دانیم در الگوریتم UCS تابع هزینه یا `cost` در نتیجه `PriorityQueue` است. این داده ساختار در `util` وجود دارد و در فاز ۰ آن کامل کردیم. تابع `push` برای این داده ساختار علاوه بر خود `item` یک مقدار هزینه هم می‌گیرد و هنگام کردن مقدار با کمترین هزینه را برمی‌گرداند. مقدار هزینه تعدادی `Action` را هم با استفاده از متدهای `getCostOfActions` به دست می‌آوریم.

```

farhad@farhad-Aspire ~/Desktop/Pacman-AI-project/P1 main ± python3 pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs [University] [Entertainment] [Tech + Study] [Private] [Shopping + Payments] [Etc]
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442.0
Average Score: 442.0
Scores: 442.0
Win Rate: 1/1 (1.00)
Record: Win

farhad@farhad-Aspire ~/Desktop/Pacman-AI-project/P1 main ± python3 pacman.py -l mediumDottedMaze -p StayEastSearchAgent
Path found with total cost of 1 in 0.0 seconds
Search nodes expanded: 186
Pacman emerges victorious! Score: 646.0
Average Score: 646.0
Scores: 646.0
Win Rate: 1/1 (1.00)
Record: Win

farhad@farhad-Aspire ~/Desktop/Pacman-AI-project/P1 main ± python3 pacman.py -l mediumScaryMaze -p StayWestSearchAgent
Path found with total cost of 68719479864 in 0.0 seconds
Search nodes expanded: 108
Pacman emerges victorious! Score: 418.0
Average Score: 418.0
Scores: 418.0
Win Rate: 1/1 (1.00)
Record: Win

```

و سپس الگوریتم را برای تست‌های مورد نظر اجرا می‌کنیم و از صحت آن اطمینان حاصل می‌کنیم.

برای الگوریتم UCS را به DFS یا BFS تغییر دهیم باید تابع هزینه خود را تغییر دهیم برای این کار باید در `Search Agent` های مورد نظر در زمانی که توابع هزینه تعریف می‌شوند تغییرات را ایجاد کنیم. برای BFS باید تابع هزینه‌ای به صورت لایه‌ای افزایش یابد و برای DFS تابع هزینه باید به صورت عمقی افزایش یابد.

الگوریتم UCS بهینه است و کامل است اما ممکن است که پیچیدگی زمانی بیشتری نسبت به DFS و BFS داشته باشد. باید توجه کرد که ما با استفاده از UCS مسائلی را حل می‌کنیم که DFS یا BFS قادر به حل آن به صورت بهینه نیستند. در واقع اگر گراف ما یا هایی با اندازه یکسان داشته باشد همان BFS بهینه و کامل بودن را تضمین می‌کند و نیازی به استفاده از UCS نیست اما هنگامی که اینگونه نباشد باید از UCS استفاده کنیم.

```

Farhad Aman *
def manhattanHeuristic(position, problem, info={}):
    """The Manhattan distance heuristic for a PositionSearchProblem"""

    return abs(position[0] - problem.goal[0]) + abs(position[1] - problem.goal[1])


Farhad Aman *
def euclideanHeuristic(position, problem, info={}):
    """The Euclidean distance heuristic for a PositionSearchProblem"""

    return ((position[0] - problem.goal[0]) ** 2 + (position[1] - problem.goal[1]) ** 2) ** .5

```

در ابتدا هیوریستیک‌های مورد نظر را کامل می‌کنیم. در اینجا **position** حالت فعلی ما است و همچنین با توجه به داشتن **goal** و **problem** مساله کار پیاده‌سازی این توابع راحت است.

```

fringe = util.PriorityQueue()
visited = []
actions = []

fringe.push((problem.getStartState(), actions), heuristic(problem.getStartState(), problem))
while not fringe.isEmpty():
    state, actions = fringe.pop() # pop the element with the lowest priority (cost + heuristic)
    if problem.isGoalState(state):
        return actions
    if state not in visited:
        visited.append(state)
        for successor, action, cost in problem.getSuccessors(state):
            fringe.push((successor, actions + [action]), problem.getCostOfActions(actions + [action])
                        + heuristic(successor, problem))
return []

```

الگوریتم **A*** بسیار شبیه به **UCS** است با این تفاوت که هزینه ما مجموع هزینه داخل الگوریتم **UCS** و هیوریستیک مورد نظر ما است. این هیوریستیک به عنوان ورودی به متده **aStarSearch** داده می‌شود و مقدار پیشفرض آن **nullHeuristic** است که همیشه مقدار 0 بر می‌گرداند و مشابه **UCS** عمل می‌شود. در نتیجه ما هنگام ایجاد هزینه اولیه یا هنگام محاسبه هزینه جدید باید مقدار هیوریستیک را مقدار هزینه جمع کنیم.

```

farhad@farhad-Aspire ~ ~/Desktop/Pacman-AI-project/P1 main ± python3 pacman.py -l openMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch Entertainment □ tech + study □ Private □ Shopping + Paymetns □ Etc
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 298 in 0.0 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
● manhattanHeuristic
farhad@farhad-Aspire ~ ~/Desktop/Pacman-AI-project/P1 main ± python3 pacman.py -l openMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs searchAgents.py
[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
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
farhad@farhad-Aspire ~ ~/Desktop/Pacman-AI-project/P1 main ± 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 0.1 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
پس از اجرای دل دل با این الگوریتم شواهد دیده ام که این الگوریتم جواب بهتری را در حدی سریع تر ارائه می کند.
 UCS پیدا می کند.
سوال: الگوریتم های جستجویی که تا به این مرحله پیاده سازی کردہ اید را روی openMaze اجرا کنید و توضیح دهید چه اتفاقی می افتد (تفاوت ها را شرح دهید).

```

ابتدا با DFS می رویم این الگوریتم کامل است و جواب را پیدا می کند اما بهینه نیست و نمی تواند کوتاه ترین مسیر را بیابد. سپس BFS را اجرا می کنیم. این الگوریتم بهینه است و می تواند کوتاه ترین مسیر با هزینه 54 را پیدا کند اما مشکل آن تعداد بسیار زیاد نودهای بسط داده شده است به این علت که ناآگاهانه است و هیچ ایده ندارد که آیا به هدف نزدیک می شود یا خیر در مرحله بعد UCS را اجرا می کنیم. در این حالت UCS مانند BFS عمل می کند چرا که تمام هزینه ها 1 در نظر گرفته شدند. پس این الگوریتم هم بهینه و کامل است اما چون ناآگاهانه است تعداد نود بسیار زیادی را بسط می دهد.

```

farhad@farhad-Aspire ~ ~/Desktop/Pacman-AI-project/P1 main ± python3 pacman.py -l openMaze -p SearchAgent -a fn=astar,heuristic=euclideanHeuristic
[SearchAgent] using function astar and heuristic euclideanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.1 seconds
Search nodes expanded: 550
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores: 456.0
Win Rate: 1/1 (1.00)
Record: Win
در مرحله بعد UCS را اجرا می کنیم. در این حالت UCS عمل می کند چرا که تمام هزینه ها 1 در نظر گرفته شدند. پس این الگوریتم هم بهینه و کامل است اما چون ناآگاهانه است تعداد نود بسیار زیادی را بسط می دهد.
farhad@farhad-Aspire ~ ~/Desktop/Pacman-AI-project/P1 main ± python3 pacman.py -l openMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.1 seconds
Search nodes expanded: 535
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores: 456.0
Win Rate: 1/1 (1.00)
Record: Win

```

در مرحله بعد A* را اجرا می کنیم. همانطور که مشاهده می شود این الگوریتم نیز کامل و بهینه است و کوتاه ترین مسیر را پیشنهاد می دهد. اما به دلیل آگاهانه بودن تعداد نودهایی را که بسط می دهد کمتر از UCS و BFS است. همانطور مشاهده می شود که تعداد نودهای بسط داده شده در A* کمتر از euclideanHeuristic می باشد.

ایده اصلی الگوریتم Dijkstra بسیار شبیه تر به UCS است تا A* و به این شکل است که ابتدا به تمام روزوس گراف یک فاصله یا همان هزینه می دهیم. که برای راس شروع 0 و برای بقیه روزوس بی نهایت

است. سپس نزدیک ترین راس را از fringe خارج کرده و آن را بسط می‌دهیم و همینطور فاصله راس های پیمایش شده را آپدیت می‌کنیم. در A* هم ما تقریباً همین کار را انجام می‌دهیم با این تفاوت که راسی که از fringe خارج می‌شود راسی نیست که لزوماً هزینه فعلی کمتری داشته باشد بلکه مقدار heuristic هم مهم است. و در واقع مجموع هزینه فعلی و هزینه هیوریستیک یا تخمینی که برای آینده داریم همان معیار ما برای خارج کردن یک راس از fringe است.

```

# Farhad Aman

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
    self.costFn = lambda x: 1

```

در ابتدا در متد `init` تنها تعریفی که نیاز است اضافه شود تعریف تابع `هزینه` است که همیشه برابر 1 می باشد.

```

def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    return self.startingPosition, set()

```

حالت شروع شامل موقعیت شروع به اضافه یک مجموعه خالی است. این مجموعه برای نگهداری گوشه های دیده شده است. به طور کلی هر حالت ما شامل موقعیت پکمن + مجموعه گوشه های دیده شده است.

```

def isGoalState(self, state):
    """
    Returns whether this search state is a goal state of the problem.
    """
    _, visited_corners = state
    return len(visited_corners) == len(self.corners)

```

برای رسیدن به هدف کافی است بررسی کنیم که آیا تمام گوشه هارا دیده ایم یا خیر.

```

successors = []
for action in [Directions.NORTH, Directions.WEST, Directions.SOUTH, Directions.EAST]:
    # 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]

    current_position, visited_corners = state
    x, y = current_position
    dx, dy = Actions.directionToVector(action)
    nextx, nexty = int(x + dx), int(y + dy)
    hitsWall = self.walls[nextx][nexty]

    if not hitsWall:
        next_visited_corners = set(visited_corners)
        if (nextx, nexty) in self.corners:
            next_visited_corners.add((nextx, nexty))
        cost = self.costFn((nextx, nexty))
        successors.append(((nextx, nexty), next_visited_corners), action, cost))

```

برای پیدا کردن حالت‌های پس از مشخص کردن همه جهت‌ها بررسی می‌کنیم که در جهت موردنظر به دیوار برخورد نداشته باشیم. مورد مهم بعدی این است که اگر حالت پسین مورد نظر یک خانه گوشه است این خانه به عنوان گوشه‌های دیده شده در حالت پسین در نظر گرفته شود.

(۵) پیدا کردن همه گوشه‌ها (۳ امتیاز)

قررت واقعی الگوریتم A* تنها توسط مسافت جستجوی چالش برانگیزتر نمایان می‌شود. اکنون می‌خواهیم می‌باشد (از ماریپیچ‌های متقاضوی برای این بخش استفاده می‌کنیم)، به ازای هر گوشه یک گرفته شده است. مسئله جستجوی جدید ما این است که کوتاهترین مسیری که از هر چهار گوشه پکنده را در ماریپیچ پیدا کنیم (بدون توجه به آنکه در گوشه‌ای غذا وجود دارد یا نه)، توجه کنید که در برخی از ماریپیچ‌ها مثل tinyCorners، کوتاهترین مسیر، همیشه اول سمت نزدیکترین غذا نمی‌رود.

ابتدا این مساله جستجو را با DFS اجرا می‌کنیم. مساله حل می‌شود. اما همانطور که مشاهده می‌شود هزینه بسیار بالا است. باز هم به این نتیجه رسیدیم که DFS کامل است اما بهینه نیست.

```

farhad@farhad-Aspire ~ ~/Desktop/Pacman-AI-project/P1 main ± 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: 249
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores: 512.0
Win Rate: 1/1 (1.00)
Record: Win
راهنمایی: کوتاهترین مسیر در tinyCorners به اندازه ۲۸ قدم است.

توجه: حتما پیش از حل بخش ۵، بخش ۱ را به طور کامل حل کنید.

farhad@farhad-Aspire ~ ~/Desktop/Pacman-AI-project/P1 main ± python3 pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
پیاده‌سازی کرد (این کلاس از قاعده searchAgents.py را در فایل CornersProblem قرار دارد).
کلاس شده است؛ نیاز است که شما قسمت‌های مورد نیاز را کامل کنید. شما نیاز دارید که یک حالت طراحی کنید که بتواند تمام اطلاعات مورد نیاز برای تشخیص این که آیا مسیر به هر چهار گوشه رفته است یا نه، را مشخص کند. حال عامل هوشمند شما می‌تواند دو مساله زیر را حل کند:

```

سپس در مرحله بعدی همین مساله را با BFS اجرا می‌کنیم. این بار هزینه بسیار کاهش یافت و برای tinyMaze برابر 28 شد که نشان می‌دهد BFS در صورت برابر بودن هزینه‌ها بهینه است.

هیوریستیک: حداقل فاصله منهتنی از موقعیت فعلی تا گوشه‌های بررسی نشده.

اثبات سازگار بودن: برای سازگار بودن باید ثابت کنیم که اگر دو نود A و B داشته باشیم $h(A) \leq h(B) + d(A, B)$

یعنی اگر از A به B برمی‌مقدار هیوریستیک حداقل به اندازه فاصله واقعی A و B کاهش یابد.

دو حالت درنظر می‌گیریم یا دورترین گوشه از نظر فاصله منهتنی به A و B یک گوشه یکسان است یا دو گوشه متفاوت است.

برای حالت یکسان بودن این گوشه. فرض کنید شرط بالا نقض شود و با رفتن از A به B مقدار هزینه کمتر از میزان کم شدن هیوریستیک باشد. ما می‌دانیم که در یک مارپیچ کمترین هزینه ممکن برای رسیدن به یک نقطه فاصله‌ی منهتنی آن است. یعنی فاصله A تا B حداقل فاصله منهتنی آن دو است.

یعنی ما از یک نقطه با حداقل فاصله‌ی منهتنی به نقطه‌ی دیگری رفته‌ایم و مقداری که فاصله منهتنی‌مان با مقصد اصلی کاهش یافته بیشتر بوده که این اتفاق غیر ممکن است (مانند اثبات سازگار بودن هیوریستیک فاصله منهتنی برای مساله جستجوی عادی)

حالت دوم این است که گوشه یکسان نباشد. در این حالت فرض کنید

$$h(A) = x; \quad h(B) = y \quad d(A, B) = z$$

و x فاصله منهتنی نقطه A تا گوشه C است و y فاصله منهتنی B تا گوشه متفاوت D است.

فرض کنیم شرط سازگار بودن نقض شده یعنی

$$x > y + z$$

در این حالت می‌دانیم که فاصله منهتنی B تا نقطه C قطعاً مقداری کمتر یا مساوی از y است.

در این حالت اگر برای محاسبه فاصله منهتنی A تا C ابتدا از A به B با z هزینه برویم و سپس با فاصله منهتنی B تا C که مقداری کمتر از y است سعی کنیم به C برمی‌مقدار هزینه ما در کل $y + z$ خواهد شد. که کمتر از x است. در حالیکه که ما می‌دانیم که کمترین فاصله ممکن برای رسیدن A به C همان x یا فاصله منهتنی بوده پس ما در محاسبه x اشتباه کردیم. در نتیجه فرض خلف ما باطل می‌شود و حکم اثبات می‌شود.

```

corners = problem.corners # These are the corner coordinates
walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

# We can use max of manhattan distances to the unvisited corners as a heuristic
heuristic_candidates = [0]
current_position, visited_corners = state
unvisited_corners = set(corners) - visited_corners

for corner in unvisited_corners:
    heuristic_candidates.append(util.manhattanDistance(current_position, corner))

return max(heuristic_candidates)

```

در کد بالا ابتدا مجموعه گوشه‌های دیده نشده را ایجاد کردیم. سپس فاصله منهشتی حالت فعلی خود با هرکدام را محاسبه کرده و در انتها مقدار `max` آن را برگرداندیم.

```

farhad@farhad-Aspire: ~/Desktop/Pacman-AI-project/P1 $ python3 pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.1 seconds
Search nodes expanded: 1136
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores: 434.0
Win Rate: 1/1 (1.00)
Record: Win

farhad@farhad-Aspire: ~/Desktop/Pacman-AI-project/P1 $ python3 pacman.py -l mediumCorners -p SearchAgent -a fn=ucs,prob=CornersProblem
[SearchAgent] using function ucs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.3 seconds
Search nodes expanded: 1937
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores: 434.0
Win Rate: 1/1 (1.00)
Record: Win

```

در اینجا هم نتیجه اجرای کد فوق را مشاهده می‌کنیم. میزان هزینه برای اجرای الگوریتم A^* و UCS هر دو 106 شده که نشان دهنده بهینه بودن A^* می‌دارد. در حالیکه تعداد گره‌های بسط داده شده در A^* مقدار 1136 است که مقداری کمتر است.

هیوریستیک: حداقل فاصله واقعی (`mazeDistance`) موقعیت فعلی تا غذاهای خورده نشده.

اثبات سازگار بودن: اثبات سازگار بودن این هیوریستیک هم دقیقا مشابه هیوریستیک مساله قیل است. دو نقطه دلخواه A و B را در نظر گرفته و دو حالت داریم. یا دورترین غذا به آنها یکسان است. در این حالت هیوریستیک ما فاصله واقعی تا نقطه غذا است که می‌دانیم و بدیهی است که سازگار است.

حالت دوم این است که دورترین غذا به A نقطه C با فاصله x و دورترین غذا به B نقطه مقاول D با فاصله y است. و فاصله واقعی A تا B هم d است.

در این حالت فرض کنیم شرط سازگار بودن برقرار نباشد یعنی

$$x > d + y$$

در این حالت فاصله واقعی C تا B را در نظر بگیرید و فرض کنید z است. چون نقطه D دورترین نقطه به B بوده می‌دانیم که

$$y \geq z$$

در نتیجه می‌توان نتیجه گرفت که

$$x > d + z$$

یعنی برای رفتن از A به C می‌توانیم ابتدا با d هزینه به B برویم و سپس با z هزینه به C برویم و هزینه ما کمتر می‌شود در حالیکه حداقل فاصله ما x بود و این تناقض است و فرض خلف باطل می‌شود.

```
position, foodGrid = state
unvisited_food = foodGrid.asList()

# We can use max of maze distances to the unvisited food as a heuristic
# And we use memoization to speed up the process
# We should use a tuple of position and unvisited food as a key for memoization
heuristic_candidates = [0]
for food in unvisited_food:
    if (position, food) not in problem.heuristicInfo:
        problem.heuristicInfo[(position, food)] = mazeDistance(position, food, problem.startingGameState)
    heuristic_candidates.append(problem.heuristicInfo[(position, food)])

return max(heuristic_candidates)
```

کد این بخش هم بسیار شبیه به بخش قبل است و تنها دو تفاوت کلی دارد.

ابتدا اینکه به جای متده `manhattanDistance` از متده `mazeDistance` استفاده شده است.

با توجه به اینکه در این سوال بر خلاف سوال قبل از هزینه واقعی رسیدن به دورترین غذا برای هیوریستیک خود استفاده کردیم هزینه محاسبات هیوریستیک بسیار بیشتر از قبل می‌شود (توجه کنید که تعداد غذاها هم به مراتب بیشتر از تعداد گوشها است). اما با توجه به اینکه هیوریستیک ما بسیار دقیق‌تر خواهد بود و تخمین بهتری از هزینه واقعی است تعداد نودهای بسط داده شده بسیار کم می‌شود. ما برای اینکه بار محاسبات را کمتر کنیم از تکنیک `memoization` استفاده می‌کنیم. در این تکنیک با فاصله `mazeDistance` بین یک نقطه و غذا را محاسبه می‌کنیم در صورتیکه قبل این کار را انجام

نداده باشیم. پس از هر بار محاسب نتایج این محاسبه را در heuristicInfo ذخیره می‌کنیم و احتمال این وجود دارد که باز هم به این نتایج نیاز داشته باشیم. مشاهده می‌شود که بار محاسبات بسیار کاهش می‌یابد.

```

farhad@farhad-Aspire: ~/Desktop/Pacman-AI-project/P1 [main ±] python3 pacman.py -l trickySearch -p AStarFoodSearchAgent
Path found with total cost of 60 in 1.4 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

farhad@farhad-Aspire: ~/Desktop/Pacman-AI-project/P1 [main ±] python3 pacman.py -l trickySearch -p SearchAgent -a fn=ucs,prob=FoodSearchProblem
[SearchAgent] using function ucs
[SearchAgent] using problem type FoodSearchProblem
Path found with total cost of 60 in 14.5 seconds
Search nodes expanded: 16688
Pacman emerges victorious! Score: 570
Average Score: 570.0
Scores: 570.0
Win Rate: 1/1 (1.00)
Record: Win

```

همانطور که می‌بینید مساله را در دو حالت A* و UCS حل کردیم. می‌توان نتیجه گرفت که هیوریستیک ما بهینه است و هزینه 60 داشت. در حالیکه تعداد نودهای بسط داده شده آن بسیار کمتر بوده 4137 در برابر 16688 در UCS

از طرفی هنگامی که هیوریستیک خود را برای مساله mediumMaze اجرا می‌کنیم باز هم انجام شدن محاسبات بسیار زیاد طول می‌کشد که یکی از نشانه‌های سازگار بودن هیوریستیک ما است.

علت اینکه به الگوریتم A* یک الگوریتم جستجوی آگاهانه می‌گوییم در واقع به خاطر وجودتابع هیوریستیک است. هیوریستیک در واقع یک تخمين از هزینه واقعی رسیدن به هدف است. با این تفاوت که محاسبه این تابع بسیار آسان‌تر و سریع‌تر از محاسبه هزینه واقعی است. وجود این تابع باعث می‌شود که الگوریتم A* به جای اینکه مانند الگوریتم UCS به صورت کورکورانه تمام مسیرهای ممکن را به صورت یکنواخت برای رسیدن به هدف جستجو کند با داشتن تخمينی از هزینه واقعی بیشتر مسیرهای را بررسی کند که امید رسیدن به هدف در آن‌ها بیشتر است. به این طریق تعداد نودهایی که بسط داده می‌شوند یا در واقع تعداد مسیرهایی که بررسی می‌شوند بسیار کمتر از تعدادی است که در الگوریتم‌های غیر آگاهانه مانند UCS بررسی می‌شود.

-8

ابتدا تابع هدف را پیاده‌سازی می‌کنیم.

```
def isGoalState(self, state):
    """
    The state is Pacman's position. Fill this in with a goal test that will
    complete the problem definition.
    """
    x, y = state

    return self.food[x][y]
```

تابع هدف این است که آیا حالت فعلی ما در خانه غذا قرار دارد یا خیر

```
def findPathToClosestDot(self, gameState):
    """
    Returns a path (a list of actions) to the closest dot, starting from
    gameState.
    """
    # Here are some useful elements of the startState
    startPosition = gameState.getPacmanPosition()
    food = gameState.getFood()
    walls = gameState.getWalls()
    problem = AnyFoodSearchProblem(gameState)

    # we use iterative deepening search to find the path with limit of 100
    return search.iterativeDeepeningSearch(problem, 100)

    # we use breadth first search to find the path
    # return search.breadthFirstSearch(problem)
```

برای پیدا کردن نزدیک نقطه غذا از دو روش جستجو استفاده کردیم. یک روش BFS و دیگری IDS است.

```

def iterativeDeepeningSearch(problem, param):
    for i in range(1, param + 1):
        result = depthLimitedSearch(problem, i)
        if result:
            return result
    return []

```

درIDS در هر مرحله یک DFS با محدودیت عمق اجرا می‌کنیم تا هنگامی که به جواب برسیم.

```

def depthLimitedSearch(problem, param):
    fringe = util.Stack()
    visited = []
    actions = []

    fringe.push((problem.getStartState(), actions))
    while not fringe.isEmpty():
        state, actions = fringe.pop() # pop the last element because fringe in DFS is LIFO
        if problem.isGoalState(state):
            return actions
        if state not in visited:
            visited.append(state)
            if len(actions) < param:
                for successor, action, cost in problem.getSuccessors(state):
                    fringe.push((successor, actions + [action]))
    return []

```

درDFS با محدودیت هم صرفاً یک شرط داریم که تعداد Action ها از حدی بیشتری نشود.

```

farhad@farhad-Aspire ~ ~/Desktop/Pacman-AI-project/P1 main ± python3 pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
[SearchAgent] using function depthFirstSearch Entertainment Tech + Study Private Shopping + Payments Etc
[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

farhad@farhad-Aspire ~ ~/Desktop/Pacman-AI-project/P1 main ± python3 pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with cost 380.
Pacman emerges victorious! Score: 2330
Average Score: 2330.0
Scores: 2330.0
Win Rate: 1/1 (1.00)
Record: Win

```

پیدا کنیم. در این بخش، عاملی می‌نویسید که همیشه به طور حریصانه نزدیک‌ترین نقطه را می‌خورد. به این منظور کلاس ClosestDotSearchAgent در فایل searchAgents.py موجود در تابع findPathToClosestDot معرفی شده است. این کلاس برای شما پیاده‌سازی می‌کند.

سپس برنامه را اجرا می‌کنیم.

$$F_C \xrightarrow{1} F_A - P \xrightarrow{2} F_B$$

برای اینکه نشان دهیم عملکرد حریصانه همیشه باعث پیدا کردن جواب بهینه نمی‌شود مثل بالا را نگاه کنید. در حالت حریصانه ابتدا A را می‌خوریم (1) سپس B را می‌خوریم (4) و در انتها C را می‌خوریم. (17)
اما حالت بهینه این است که ابتدا B را بخوریم (2) سپس A را بخوریم (5) و در انتها C را بخوریم. (15) که بهینه تر است.