

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

دانشکده مهندسی کامپیووتر

پروژه اول مبانی و کاربردهای هوش مصنوعی

نگارش

مهدیه سادات بنیس

استاد درس

دکتر مهدی جوانمردی

نیم سال دوم ۱۴۰۱

۰ بخش ۰: شناخت فایل‌ها و برخی از **function**‌های پروژه سوال

کاربرد کلاس SearchProblems در فایل search.py را به همراه متودهای آن توضیح دهید.

همچنین به اختصار کاربرد هر یک از کلاس‌های Agent, Directions, Configuration, Actions, Grid، همچنین game.py قرار دارند، بیان کنید.

- کلاسی AbstractSearchProblem برای ساختار دهی و سازماندهی مسله سرج می‌باشد در واقع شامل المان‌های یک مسئله سرج می‌باشد که در درس معرفی شد:

- getStartState: حالت شروع را مشخص می‌کند.

- isGoalState: آزمون هدف می‌باشد برای تشخیص اینکه آیا حالت داده شده حالت هدف می‌باشد یا نه.

- getSuccessors: تابع پسین می‌باشد و برای یک حالت داده شده لیستی از شامل (successor, action, stepCost) است. برمیگرداند که در آن 'successor' جانشین وضعیت فعلی است، 'Action' اقدامی است که برای رسیدن به آنجا لازم است، و 'stepCost' هزینه افزایشی رسیدن به آن جانشین است.

- getCostOfActions: تابع هزینه است که هزینه کل یک دنباله خاص از اقدامات را برمی‌گرداند.

- کلاسی برای ایجاد و نشان دادن یک عامل می‌باشد (برای مثال خود پکمن یا روح‌ها عامل‌ها هستند). مهم ترین متدهای کلاس getAction است که حرکت بعدی عامل را با توجه به حالت فعلی برمیگرداند.

- Directions: جهت‌های حرکتی که عامل‌ها میتوانند از آن استفاده کنند را نشان میدهد علاوه بر چهار جهت اصلی؛ جهت چپ و راست و همچنین جهت عکس نیز به صورت نسبی تعریف شده‌اند.

- Configuration: مختصات (x,y) یک کاراکتر را به همراه جهت حرکت آن نگه میدارد و در صورت نیاز به کمک generateSuccessors میتواند موقعیت و جهت را با توجه به اکشن آپدیت کند.

- AgentState: حالت عامل را که شامل configuration و سرعت و ... می‌باشد را نگه میدارد.

- Grid: در واقع این کلاس نمایانگر نقشه بازی می‌باشد. نقشه را در قالب یک آرایه دو بعدی از اشیاء که توسط لیستی از لیست‌ها پشتیبانی می‌شود ذخیره می‌کند. داده‌ها از طریق شبکه [y][x] قابل دسترسی هستند که در آن (x,y) موقعیت‌هایی در نقشه بازی هستند.

پیاده‌سازی: متود push و pop از کلاس‌های Stack و Queue و همچنین متود update از کلاس PriorityQueue را بررسی کنید و در صورت لزوم آنها را در فایل util.py اصلاح کنید.

```
class Stack:  
    """A container with a last-in-first-out (LIFO) queuing policy."  
    def __init__(self):  
        self.list = []  
    def push(self, item):  
        """Push 'item' onto the stack"  
        "*** YOUR MAY CODE HERE ***"  
        # self.list.insert(0, item)      insert(0, item) inserts at the front of the list.  
        self.list.append(item)         # appends an element to the end of the list.  
    def pop(self):  
        """Pop the most recently pushed item from the stack"  
        
```

```

    "*** YOUR MAY CODE HERE ***"
    return self.list.pop()          # removes and returns the last value from the List
def isEmpty(self):
    "Returns true if the stack is empty"
    return len(self.list) == 0
class Queue:
    "A container with a first-in-first-out (FIFO) queuing policy."
    def __init__(self):
        self.list = []
    def push(self,item):
        "Enqueue the 'item' into the queue"
        "*** YOUR MAY CODE HERE ***"
        self.list.insert(0,item)      # insert(0,item) inserts at the front of the list.
        # self.list.append(item)      appends an element to the end of the list.
    def pop(self):
        """
            Dequeue the earliest enqueued item still in the queue. This
            operation removes the item from the queue.
        """
        "*** YOUR MAY CODE HERE ***"
        return self.list.pop()          # removes and returns the last value from the List
    def isEmpty(self):
        "Returns true if the queue is empty"
        return len(self.list) == 0

```

```

def update(self, item, priority):
    # If item already in priority queue with higher priority, update its priority
    # and rebuild the heap.
    # If item already in priority queue with equal or lower priority, do nothing.
    # If item not in priority queue, do the same thing as self.push.
    "*** YOUR MAY CODE HERE ***"
    for index, (p, c, i) in enumerate(self.heap):
        if i == item:
            # If item already in priority queue with equal or lower priority, do
            # nothing.

            if p <= priority:
                break
            # If item already in priority queue with higher priority, update its
            # priority and rebuild the heap.
            del self.heap[index]
            self.heap.append((priority, c, item))
            heapq.heapify(self.heap)
            break
        else:
            # If item not in priority queue, do the same thing as self.push.
            self.push(item, priority)

```

• بخش 1: پیدا کردن یک نقطه ثابت غذا با استفاده از جستجوی اول عمق (DFS)

الگوریتم پیاده سازی شده:

در الگوریتم DFS ما یک Stack fring می باشد. ابتدا در آن حالت اولیه و مسیری که تا الان پیدا کردیم (که فعلاً تهی است) را اضافه میکنیم و یک مجموعه هم برای ذخیره نودهایی که بررسی میکنیم نگه میداریم سپس تا وقتی Stack خالی نشده هر بار عنصر بالای Stack خارج میکنیم آن را به عناصر بررسی شده اضافه میکنیم اگر استیت عنصر خارج شده از پشته استیتنهایی باشد که جواب را پیدا کرده ایم و در نتیجه action های این عنصر نتیجهنهایی ما است. اما در غیر این صورت وضعیت بعدی پس از انتخاب عنصر فعلی را به کمکتابع problem.getSuccessors می دیده و اگر نود بعدی تا الان دیده نشده بود آن را داخل پشته قرار میدهیم و مسیر جدید را نیز به همراه آن اضافه میکنیم.

سوال: راه حل DFS را از نظر پیچیدگی زمانی و فضایی تحلیل کنید. طبق چیزی که از حرکت عامل مشاهده کردید به سوال زیر پاسخ دهید.

DFS دارای پیچیدگی زمانی $O(V+E)$ است که V تعداد رئوس و E تعداد یال های گراف است.

در الگوریتم DFS، هر رأس حداکثر یک بار دیده می شود و هر یال حداکثر دو بار پیمایش می شود (یک بار برای بازدید از راس شروع و یک بار برای بازگشت به راس قبلی). بنابراین، تعداد کل عملیات انجام شده توسط الگوریتم با مجموع تعداد رئوس و یال های گراف متناسب است.

در بدترین حالت، الگوریتم DFS از هر رأس بازدید می کند و از هر یال در نمودار عبور می کند، به این معنی که پیچیدگی زمانی $O(V+E)$ است. با این حال، در برخی موارد، الگوریتم DFS ممکن است در صورت بازدید از تمام رئوس قابل دسترسی، زودتر خاتمه یابد، که تعداد عملیات انجام شده را کاهش می دهد و منجر به پیچیدگی زمانی کمتر می شود.

پیچیدگی فضایی الگوریتم DFS به پیاده سازی بستگی دارد. اگر با استفاده از بازگشت پیاده سازی شود، پیچیدگی فضا متناسب با عمق درخت بازگشتی است، که برابر است با طول طولانی ترین مسیر در نمودار یا درخت در حال پیمایش.

اگر نمودار یک درخت باشد و عمق آن d باشد، پیچیدگی فضا $O(d)$ خواهد بود. با این حال، اگر نمودار یک درخت نباشد، ممکن است چندین مسیر برای پیمایش وجود داشته باشد و حداکثر عمق درخت بازگشتی ممکن است بسیار بزرگتر باشد. در بدترین حالت، زمانی که نمودار یک نمودار کامل است، پیچیدگی فضایی می تواند $O(V)$ باشد، که در آن V تعداد رئوس نمودار است.

اگر با استفاده از Stack پیاده سازی شود، پیچیدگی فضایی متناسب با حداکثر تعداد راس هایی است که می توان در Stack ذخیره کرد. حداکثر اندازه Stack برابر است با حداکثر طول هر مسیری که در نمودار طی می شود. بنابراین، اگر نمودار درختی باشد و عمق آن d باشد، پیچیدگی فضا نیز $O(d)$ است. در بدترین حالت، اگر نمودار یک نمودار کامل باشد، پیچیدگی فضا می تواند $O(V)$ باشد. به طور کلی، پیچیدگی فضایی الگوریتم DFS متناسب با حداکثر عمق درخت بازگشتی یا حداکثر طول هر مسیری در نمودار است که پیمایش می شود.

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

خیر، الگوریتم DFS بهینه نیست و عامل رفتار منطقی ندارد زیرا که بدون در نظر گرفتن عمق یا هزینه، "سمت چپ ترین" راه حل را پیدا میکند. الگوریتم DFS همیشه در یافتن راه حل بهینه یا منطقی نیست. الگوریتم تضمین نمی کند که کوتاه ترین مسیر یا راه حل بهینه در همه موارد پیدا شود.

سوال: در غالب یک شبکه کد مختصر الگوریتم IDS را توضیح دهید و حالتی را شرح دهید که این الگوریتم عملکرد بدتری نسبت به DFS دارد.

در این الگوریتم میخواهیم مزیت فضای DFS را با مزیت زمانی و بهینگی BFS ترکیب کنیم. برای این کار در هر مرحله الگوریتم DFS را تا عمق معین و محدودی انجام میدهیم یعنی در هر بار فراخوانی DFS را اجرا میکنیم اما آن را از فراتر رفتن از عمق معین محدود می کنیم. بنابراین اساساً ما DFS را به روش BFS انجام می دهیم. همان طور که در شبکه کد زیر نشان داده میشود DFS را برای عمق های مختلف اجرا میکنیم و محدودیت عمق را یکی یکی زیاد میکنیم تا جایی که به جواب برسیم.

```
# Returns true if target is reachable from # src within max_depth
bool IDDFS(src, target, max_depth)
    for limit from 0 to max_depth

        if DLS(src, target, limit) == true
            return true
        return false

bool DLS(src, target, limit)
    if (src == target)
        return true

# If reached the maximum depth,
# stop recursing.
    if (limit <= 0)
        return false

    foreach adjacent i of src
        if DLS(i, target, limit+1)
            return true
    return false
```

برای تبدیل DFS به IDS کافیست هنگام فراخوانی بازگشتی عمق را نیز لحاظ کنیم (شرط محدودیت عمق و افزایش عمق در هر مرحله)

• بخش 2: جستجوی اول سطح (BFS)

الگوریتم پیاده سازی شده:

این الگوریتم همانند الگوریتم DFS پیاده سازی می شود با این تفاوت که fringe ما یک صف است و همچنین محل اضافه کردن نود به آرایه دیده شده ها متفاوت است.

سوال: دستور زیر را اجرا کنید. آیا کد شما بدون هیچ گونه تغییری مسئله ۸-پازل را حل میکند؟ توضیح دهید.
همان طور که مشاهده میکنیم چون الگوریتم به صورت کلی نوشته شده است برای حل مسئله ۸-پازل نیز کار میکند.

```
(base) mahdieh@mahdiehs-MBP P1 % python eightpuzzle.py
A random puzzle:
-----
| 3 | 1 | 4 |
|-----|
| 5 | 2 | |
|---|---|---|
| 6 | 7 | 8 |
-----
Press return for the next state...
After 0 moves: up
-----
| 1 | 1 | 4 |
|-----|
| 3 | 5 | 2 |
|-----|
| 6 | 7 | 8 |
-----
Press return for the next state...
After 1 move: up
-----
| 1 | 1 | 4 |
|-----|
| 3 | 5 | 2 |
|-----|
| 6 | 7 | 8 |
-----
BFS found a path of 7 moves: ['up', 'right', 'right', 'down', 'left', 'up', 'left']
After 1 move: up
-----
| 1 | 1 | 4 |
|-----|
| 3 | 5 | 2 |
|-----|
| 6 | 7 | 8 |
-----
Press return for the next state...
After 2 moves: right
-----
| 1 | 1 | 4 |
|-----|
| 3 | 5 | 2 |
|-----|
| 6 | 7 | 8 |
-----
Press return for the next state...
After 3 moves: right
-----
| 1 | 4 | 2 |
|-----|
| 3 | 5 | 1 |
|-----|
| 6 | 7 | 8 |
-----
Press return for the next state...
After 4 moves: down
-----
| 1 | 4 | 2 |
|-----|
| 3 | 5 | 1 |
|-----|
| 6 | 7 | 8 |
-----
Press return for the next state...
After 5 moves: left
-----
| 1 | 4 | 2 |
|-----|
| 3 | 5 | 1 |
|-----|
| 6 | 7 | 8 |
-----
Press return for the next state...
After 6 moves: up
-----
| 1 | 1 | 2 |
|-----|
| 3 | 4 | 5 |
|-----|
| 6 | 7 | 8 |
-----
Press return for the next state...
After 7 moves: left
-----
| 1 | 1 | 2 |
|-----|
| 3 | 4 | 5 |
|-----|
| 6 | 7 | 8 |
-----
Press return for the next state...
```

```
Press return for the next state...
After 3 moves: right
-----
| 1 | 4 | 2 |
|-----|
| 3 | 5 | 1 |
|-----|
| 6 | 7 | 8 |
-----
Press return for the next state...
After 4 moves: down
-----
| 1 | 4 | 2 |
|-----|
| 3 | 5 | 1 |
|-----|
| 6 | 7 | 8 |
-----
Press return for the next state...
After 5 moves: left
-----
| 1 | 4 | 2 |
|-----|
| 3 | 5 | 1 |
|-----|
| 6 | 7 | 8 |
-----
Press return for the next state...
After 6 moves: up
-----
| 1 | 1 | 2 |
|-----|
| 3 | 4 | 5 |
|-----|
| 6 | 7 | 8 |
-----
Press return for the next state...
After 7 moves: left
-----
| 1 | 1 | 2 |
|-----|
| 3 | 4 | 5 |
|-----|
| 6 | 7 | 8 |
-----
Press return for the next state...
```

سوال: الگوریتم BBFS را به صورت مختصر با نوشتمن یک شبکه کد ساده توضیح دهید و آن را با الگوریتم BFS مقایسه کنید.

در این الگوریتم بر خلاف DFS، سرج از دو سمت انجام میشود یعنی همزمان هم از سمت شروع به هدف و هم از سمت هدف به شروع به دنبال کوتاه ترین مسیر میگردیم.

جستجوی دوطرفه، گراف جستجوی منفرد (که احتمالاً به صورت تصاعدی رشد میکند) را با دو گراف فرعی کوچکتر جایگزین میکند - که در یکی راس شروع در ریشه است و در دیگری راس هدف. جستجو با برخورد/اشتراک دو نمودار پایان می یابد. همان طور که در شبکه کد زیر مشاهده میشود کافیست دو گراف در نظر بگیریم که در یکی راس شروع در ریشه است و در دیگری راس هدف و تا وقتی که حداقل یکی از دو گراف به طور کامل پیمایش نشده BFS را بر روی آن در صدا بزنیم تا جایی که هر دو گراف به نود یکسانی برسند.

```
def bidirectional_search(self, src, dest):
    # Add source to queue and mark visited as True and add its parent as -1

    self.src_queue.append(src)
    self.src_visited[src] = True
    self.src_parent[src] = -1
    # Add destination to queue and mark visited as True and add its parent as -1
    self.dest_queue.append(dest)
    self.dest_visited[dest] = True
    self.dest_parent[dest] = -1
    while self.src_queue and self.dest_queue:
        # BFS in forward direction from Source Vertex
        self.bfs(direction = 'forward')
        # BFS in reverse direction from Destination Vertex
        self.bfs(direction = 'backward')
```

```

# Check for intersecting vertex
intersecting_node = self.is_intersecting()
# If intersecting vertex exists
# then path from source to
# destination exists
if intersecting_node != -1:
    print(f"Path exists between {src} and {dest}")
    print(f"Intersection at : {intersecting_node}")
    self.print_path(intersecting_node, src, dest)
    exit(0)

```

سوال: الگوریتم BFS را از نظر پیچیدگی زمانی و فضایی با الگوریتم DFS مقایسه کنید و مطابق آنچه در این دو بخش از پژوهش مشاهده کردید، بیان کنید هریک از الگوریتم‌های مذکور در چه مواردی عملکرد بهتری در حرکت عامل دارد.

پیچیدگی زمانی:

در بدترین حالت، BFS از تمام گره‌ها در گراف یا درخت بازدید می‌کند و بنابراین پیچیدگی زمانی $O(V+E)$ است، که در آن V تعداد رئوس (گره‌ها) و E تعداد یال‌ها است. این به این دلیل است که در بدترین حالت، BFS یک بار از هر رأس و یک بار از هر یال بازدید می‌کند.

پیچیدگی فضایی:

پیچیدگی فضایی BFS نیز $O(V+E)$ است زیرا باید صفحی از رئوس برای بازدید و یک آرایه (explored) برای علامت گذاری رئوس بازدید شده حفظ کند. در بدترین حالت، تمام رئوس و یال‌ها بازدید خواهند شد، بنابراین فضای مورد نیاز برای ذخیره رئوس بازدید شده و صفحه متناسب با $V+E$ خواهد بود.

در برخی موارد، پیچیدگی فضایی BFS را می‌توان با استفاده از یک ساختار داده متفاوت برای ذخیره رئوس بازدید شده و صفحه مانند یک صف اولویت کاهش داد. با این حال، پیچیدگی زمانی یکسان باقی خواهد ماند، $O(V+E)$.

پیچیدگی زمانی BFS با برابر DFS است اما پیچیدگی فضایی DFS کمتر از BFS می‌باشد.

الگوریتم BFS زمانی بهتر عمل می‌کند که هدف به ریشه نزدیک است و الگوریتم DFS زمانی بهتر عمل می‌کند که هدف از ریشه دور است.

• بخش ۳: تغییرتابع هزینه

پیاده سازی الگوریتم:

این الگوریتم مانند الگوریتم های قبلی است با این تفاوت که `fringe` یک صف اولویت سنت و همچنین عناصری که در قرار میدهیم به صورت tuple (حالت مسیری که تا الان تعیین شده هزینه تا به الان) می باشد و اولویت آن برابر با هزینه است. همچنین با توجه به ینکه در این الگوریتم بررسی شرط پایانی در هنگام تولید نود است ابتدا این شرط را چک میکنیم.

سوال: آیا ممکن است که با مشخص کردن یک تابع هزینه مشخص برای الگوریتم UCS، به الگوریتم BFS و یا DFS بررسیم؟ در صورت امکان برای هر کدام از الگوریتم های DFS و یا BFS، تابع هزینه مشخص شده را با تغییر کد خود توضیح دهید (نیاز به پیاده سازی کد جدیدی نیست؛ صرفاً تغییراتی را که باید به کد خود اعمال کنید را ذکر نمایید).

برای BFS ممکن است اگر تعداد یال هایی که تا رسیدن به آن گره از گره شروع طی شده است را به عنوان تابع هزینه در نظر میگیریم در واقع یعنی وزن هر یال را اگر یک در نظر بگیریم و UCS را مانند زمانی که گراف وزن دار داریم اجرا کنیم به میرسیم. اما برای DFS ممکن نیست زیرا میدانیم DFS لزوماً جواب بهینه را به ما نمیدهد اما میدانیم یکی از ویژگی های UCS بهینه بودن جواب آن است پس نمیتوان تابع هزینه ای پیدا کرد که باعث شود UCS جواب DFS را بدهد. در کد UCS اگر هزینه اضافه شده تا نود بعدی را یک در نظر بگیریم به BFS میرسیم:

```
fringe = util.PriorityQueue()
explored = set()
fringe.push((problem.getStartState(), [], 0), 0)
while not fringe.isEmpty():
    # state: location, actions: path
    state, actions, cost = fringe.pop()
    if problem.isGoalState(state):
        return actions
    if state not in explored:
        explored.add(state)
        for s, a, c in problem.getSuccessors(state):
            if s not in explored:
                fringe.push((s, actions + [a], cost + c), cost + c)
```

تغییری که باید در کد خود اعمال کنیم این است که به جای `c`، یک قرار بدهیم.

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

نمیتوان گفت که آیا الگوریتم UCS به طور کلی بهتر از DFS و BFS است یا خیر، زیرا عملکرد هر الگوریتم به مسئله خاصی که را حل می کند.

یک الگوریتم جستجوی آگاهانه تر از DFS یا BFS است، زیرا هنگام انتخاب گره بعدی برای گسترش، هزینه هر مرحله پیماش شده در یک مسیر را در نظر می گیرد. این می تواند UCS را در یافتن راه حل کم هزینه در مسائلی که هزینه هر مرحله متفاوت است کارآمدتر کند.

با این حال، در مسائلی که هزینه هر مرحله یکنواخت است یا راه حل نزدیک به ریشه درخت جستجو است، BFS و DFS می توانند کارآمدتر از UCS باشند. وقتی هدف در نزدیکی ریشه درخت جستجو قرار دارد، BFS می تواند سریع تر راه حلی پیدا کند، در حالی که DFS می تواند با ذخیره نکردن تمام گرهها در مرز، در حافظه و زمان صرفه جویی کند.

یک الگوریتم جستجو است که فضای جستجو را با در نظر گرفتن هزینه هر مرحله برای رسیدن به یک گره بررسی می کند و UCS چندین مزایا و معایب دارد:

مزایا:

- راه حل بهینه: UCS همیشه راه حل بهینه را از نظر هزینه پیدا می کند، با این فرض که تابع هزینه سازگار است (یعنی هزینه برآورد شده برای رسیدن به هدف از هر گره هرگز کمتر از هزینه واقعی نیست).

کامل: UCS راه حلی را در صورت وجود پیدا می کند، مشروط بر اینکه فضای جستجو محدود باشد.

کارآمد: UCS اغلب از الگوریتم‌های جستجوی ناآگاهانه مانند DFS یا BFS کارآمدتر است، زیرا از هزینه هر مرحله برای هدایت جستجو و اجتناب از کاوش مسیرهای نامطلوب استفاده می کند.

- اکتشافی قابل قبول: UCS می تواند با هدایت جستجو به سمت گره هدف، از یک مسیر قابل قبول برای بهبود کارایی خود استفاده کند.

معایب:

- پیچیدگی زمانی: UCS می تواند از نظر محاسباتی گران باشد زیرا تمام مسیرهای ممکن را از گره شروع تا گره هدف بررسی می کند. در بدترین حالت، UCS می تواند به زمان تصاعدی برای تکمیل نیاز داشته باشد.

- استفاده از حافظه: UCS به مقدار زیادی حافظه برای ذخیره مرز جستجو نیاز دارد که می تواند در فضاهای جستجوی بزرگ به مشکل تبدیل شود.

- محدود به هزینه تک مسیر: UCS فقط هزینه هر مرحله جداگانه در طول یک مسیر را در نظر می گیرد، به این معنی که ممکن است در شرایطی که هزینه رسیدن به یک گره به مسیر طی شده برای رسیدن به آن بستگی دارد، به خوبی کار نکند.

- حساسیت به هزینه یال ها: عملکرد UCS می تواند به هزینه یال ها در نمودار جستجو حساس باشد. اگر هزینه های لبه به خوبی رفتار نشود، الگوریتم ممکن است زمان را برای کاوش در مسیرهای بی امید تلف کند.

به طور خلاصه، UCS یک الگوریتم جستجوی مؤثر در بسیاری از موقعیت‌ها است، به ویژه زمانی که فضای جستجو به خوبی رفتار می کند و هزینه هر مرحله نشانگر خوبی از کیفیت مسیر است. با این حال، عملکرد آن می تواند تحت تأثیر ثبات تابع هزینه، محدودیت های حافظه و پیچیدگی فضای جستجو قرار گیرد.

• بخش 4: جستجوی A استار (A^*)

پیاده سازی الگوریتم:

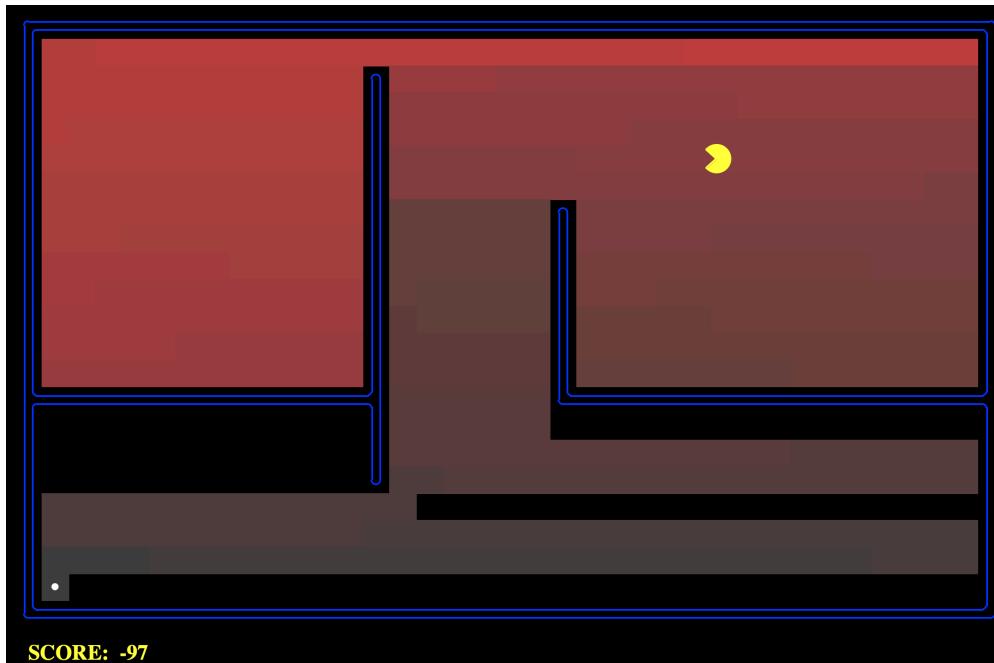
این الگوریتم مانند UCS است با این تفاوت که اولویت هر عنصر در صف اولویت برابر با هزینه فعلی به علاوه هزینه تخمین زده شده از نod فعلی تا مقصد می باشد.

سوال: الگوریتم های جستجویی که تا به این مرحله پیاده سازی کردہ اید را روی openMaze اجرا کنید و توضیح دهید چه اتفاقی میافتد (تفاوت ها را شرح دهید).

:DFS

مسیر بهینه را پیدا نمیکند و بیشترین تعداد گره را باز میکند زیرا مسیر به گونه ای است که امکان عمیق تر شدن وجود دارد و در نتیجه عامل با اینکه مسیر بهینه نیست اما در عمق پیش میروند و به هدف میرسد.

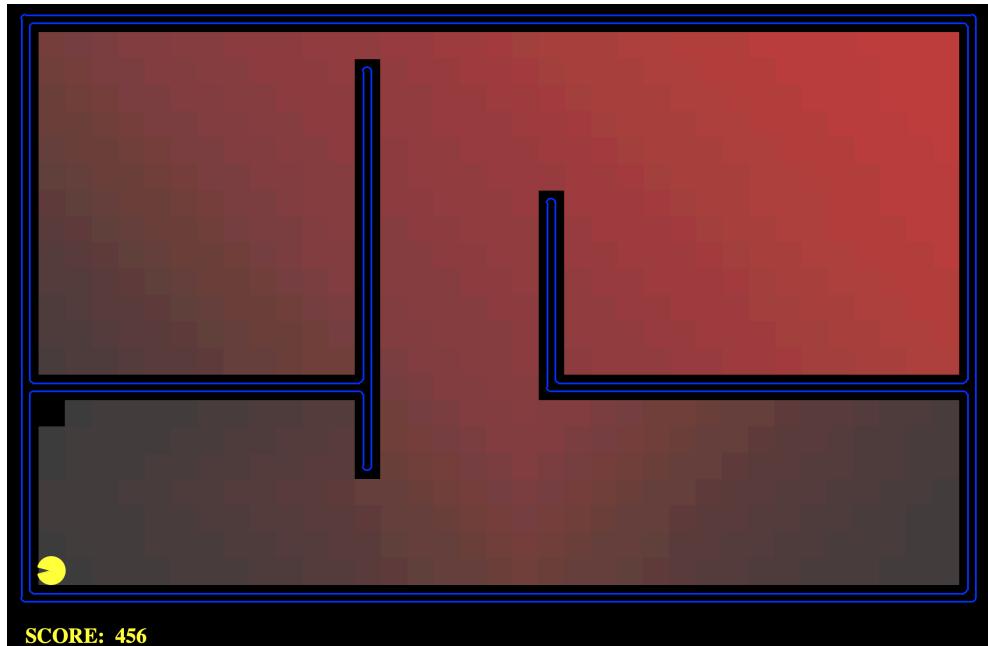
```
(base) mahdieh@mahdiehs-MBP P1 % python pacman.py -l openMaze -p SearchAgent -a fn=dfs
[SearchAgent] using function dfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 298 in 0.0 seconds
Search nodes expanded: 806
Pacman emerges victorious! Score: 212
Average Score: 212.0
Scores:      212.0
Win Rate:    1/1 (1.00)
Record:     Win
```



:BFS

مسیر بهینه پیدا میشود اما تقریبا تمام صفحه بررسی میشود.

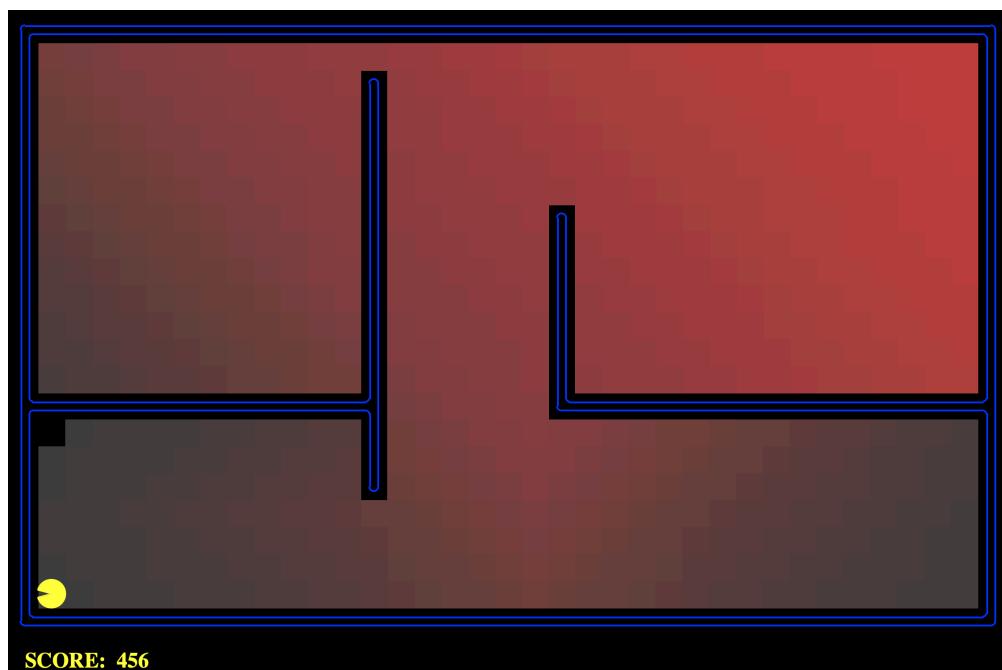
```
(base) mahdieh@mahdiehs-MBP P1 % python pacman.py -l openMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.0 seconds
Search nodes expanded: 682
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:      456.0
Win Rate:    1/1 (1.00)
Record:     Win
```



:UCS

عملکردی مشابه BFS دارد زیرا هزینه‌ای تعریف نشده.

```
(base) mahdieh@mahdiehs-MBP:~/P1% python 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 0.0 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
```



مسیر بهینه را پیدا میکند و نسبت به الگوریتم های دیگر تعداد گره کمتری را باز میکند زیرا از یکتابع تخمین هزینه برای سوگیری جست و جویش استفاده میکند تا سعی کند تقریبا در جهت هدف و به صورت بهینه ترجیح جست و جو کند.



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

الگوریتم Dijkestra و الگوریتم A^* هر دو برای یافتن کوتاه ترین مسیر در یک گراف وزن دار استفاده می شوند، اما در رویکردشان و اطلاعات مورد استفاده برای هدایت جستجو متفاوت هستند.

- ایده اصلی الگوریتم Dijkestra این است که از یک گره منبع شروع شود و به طور مکرر گره های همسایه را با کمترین هزینه از گره منبع پیمایش کند. یک صف اولویت از گره هایی که باید پیمایش شوند را حفظ می کند و هزینه هر گره را هنگام بازدید به روز می کند. الگوریتم تا رسیدن به گره مقصد یا تا زمانی که تمام گره های قابل دسترسی کشف شوند ادامه می یابد.

- ایده اصلی الگوریتم A^* استفاده از یکتابع هیوریستیک برای هدایت جستجو به سمت گره هدف کارآمدتر است. هزینه هر گره از گره مبدأ را با تخمینی از هزینه باقیمانده برای رسیدن به گره مقصد، همانطور که توسط تابع هیوریستیک ارائه می شود، ترکیب می کند. این الگوریتم یک صف اولویت از گره ها را برای بررسی بر اساس هزینه کل حفظ می کند و هزینه هر گره را در حین بازدید به روزرسانی می کند. الگوریتم تا رسیدن به گره مقصد یا تا زمانی که تمام گره های قابل دسترسی کشف شوند ادامه می یابد.

به طور خلاصه، هدف هر دو الگوریتم یافتن کوتاه ترین مسیر در یک گراف وزن دار است، اما الگوریتم Dijkestra گره های همسایه را با کمترین هزینه از گره منبع بررسی می کند، در حالی که الگوریتم A^* هزینه را با تخمین هزینه باقیمانده ترکیب می کند تا به گره مقصد، همانطور که توسط تابع هیوریستیک ارائه شده است. این به الگوریتم A^* اجازه می دهد تا جستجو را به طور مؤثر تر به سمت گره هدف هدایت کند، و در نتیجه زمانی که تابع هیوریستیک به خوبی طراحی شده است و فضای جستجو بزرگ است زمان های جستجوی سریع تر می شود.

• بخش ۵: پیدا کردن همه گوشه‌ها

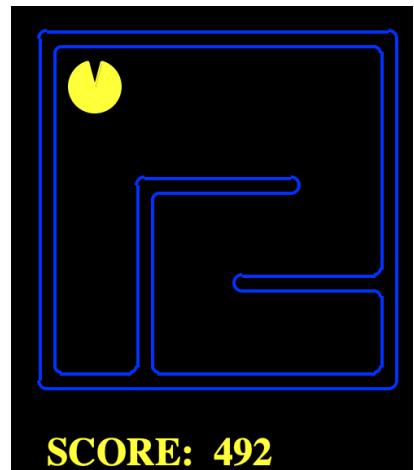
پیاده سازی الگوریتم:

حالی که برای تعیین گذر از ۴ گوشته تعیین میکنیم به صورت یک tuple می باشد که عنصر اول آن وضعیت پکمن و دومی tuple از گوشه‌هایی که هنوز دیده نشده اند. پس در حالت ابتدایی tuple به صورت زیر خواهیم داشت: `(self.startingPosition, (0, 1, 2, 3))`

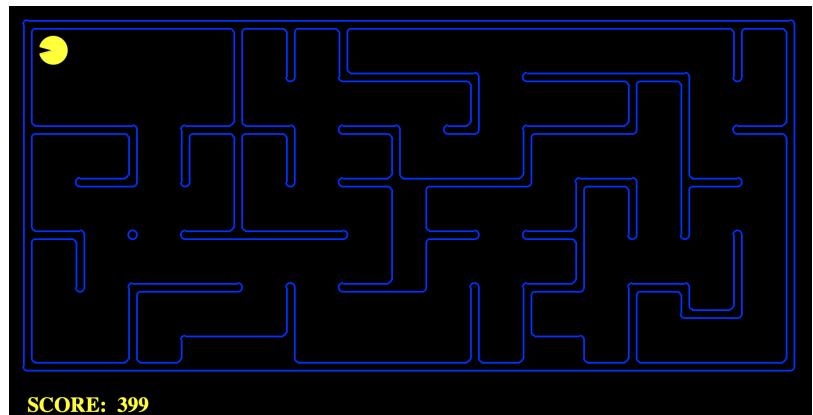
چون پکمن در وضعیت ابتدایی هست و هنوز هیچ گوشه‌ای را ندیده. حالت نهایی زمانی است که عنصر دوم tuple حالت که گوشه‌های ندیده شده را نگه میدارد خالی شود یعنی همه گوشه‌ها را دیده باشیم. تابع successor نیز این حالت توصیف شده را به ازای حرکات مختلف عوض میکند. برای این کار مختصات بعدی با توجه به حرکت انجام شده را پیدا میکند در صورت قانونی بودن این حرکت اگر گوشه بود آن گوشه را از لیست گوشه‌های باقی مانده حذف کرده و حالت جدید را می‌سازد.

خروجی‌های مربوط به دستورات ذکر شده را تحلیل کنید.

```
(base) mahdieh@mahdiehs-MBP P1 % python pacman.py -l tinyCorners -p SearchAgent -a fn=dfs,prob=CornersProblem
[SearchAgent] using function dfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 48 in 0.0 seconds
Search nodes expanded: 72
Pacman emerges victorious! Score: 492
Average Score: 492.0
Scores: 492.0
Win Rate: 1/1 (1.00)
Record: Win
```



```
(base) mahdieh@mahdiehs-MBP P1 % python pacman.py -l mediumCorners -p SearchAgent -a fn=dfs,prob=CornersProblem
[SearchAgent] using function dfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 141 in 0.0 seconds
Search nodes expanded: 417
Pacman emerges victorious! Score: 399
Average Score: 399.0
Scores: 399.0
Win Rate: 1/1 (1.00)
Record: Win
```



همان طور که مشاهده میکنیم الگوریتم DFS کامل میباشد یعنی هر چهار گوشه را پیمایش میکند اما لزوماً بهینه نیست و به سمت نزدیک ترین هدف لزوماً نمیرود. اگر با الگوریتم اجرا کنیم مشاهده میکنیم که هزینه کاهش می‌یابد.

• بخش 6: پیدا کردن همه گوشه‌ها

پیاده سازی کد:

اگر گوشه‌ای نمانده بود که صفر بر میگردانیم در غیر این صورت حداقل فاصله منهتن در بین تمام گوشه‌های باقی نمانده را خروجی میدهیم و برای این کار از تابع منهتن تعریف شده در `utils` استفاده میکنیم.

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

هیوریستیک انتخابی حداقل فاصله منهتن در بین تمام گوشه‌های باقی نمانده است. با توجه به وجود دیوار و موانع در مسیر فاصله منهتن همواره کوچک‌تر یا مساوی فاصله واقعی بین دو نقطه است همچنین چون باید تمام گوشه‌ها را ببینیم پس حتماً دورترین گوشه را را هم میبینیم اگر مستقیم به سراغ آن برویم که دقیقاً همان حالت ماکسیمم است اما اگر بخواهیم اول به گوشه دیگری برویم و بعد از آنجا به دورترین گوشه برویم باز هم مسافت با توجه به قضیه حمار هندسه از اینکه مستقیم به دورترین گوشه برویم (هیوریستیک ما) بیشتر خواهد بود پس این هیوریستیک قابل قبول است.

از طرفی درباره سازگار بودن هم اینگونه اثبات میکنیم که اگر به سمت دورترین راس حرکت نکنیم، و به سراغ راس دیگری برویم، در اینصورت این دور ترین راس تاثیر زیادی بر روی نا دقیق بودن تابع هیوریستیک ما خواهد داشت. اگر به سمت راس دورتر، که از اول تا آخر باید سراغش برویم حرکت کنیم، تابع هیوریستیک ما قطعاً در مرحله بعدی عددی بزرگ‌تر از مقدار هیوریستیک فعلی نمیدهد چرا که ما در هر حرکت تنها یک واحد از رؤوس دور یا نزدیک میشویم.

پس با این حساب اگر دوباره مقدار هیوریستیک راس قبلی که دورترین فاصله را داشت انتخاب شود، مقدار هیوریستیک جدید قطعاً کمتر خواهد شد.

اما اگر راس دیگری این بار به عنوان دورترین راس انتخاب شود، باز هم نگرانی نداریم چرا که صرفاً یک واحد نسبت به قبلی حرکت کرده‌ایم و از دورترین راس فعلی دور یا نزدیک شدیم. اگر نزدیک شده باشیم که قطعاً مقدار هیوریستیک ما کمتر میشود. اما اگر دور شده باشیم نیز مقدار هیوریستیک فعلی حداقل با هیوریستیک قبلی برابر میشود و باز هم افزایش مقدار هیوریستیک را نداریم. بدین صورت در هیچ یک از حالت‌ها، مقدار تابع هیوریستیک کاهش پیدا نمیکند. پس سازگار است.

```
(base) mahdieh@mahdiehs-MBP P1 % python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.0 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
```

• بخش 7: خوردن همه غذاها

پیاده سازی کد:

در صورت وجود غذا فاصله mazeDistance را برای تمام غذاها با کمک تابع mazeDistance بدست آورده و ماکسیمم آن ها را برمیگردانیم.

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

فاصله mazeDistance برای غذاها را پیدا میکنیم و ماکسیمم آن را به عنوان هیوریستیک در نظر میگیریم زیرا در هر صورت باید تمام غذاها را بخوریم (از جمله دورترین غذا و همچنین بقیه غذاها) و mazeDistance هم فاصله واقعی با توجه به شرایط صفحه را به ما میدهد پس این هیوریستیک قابل قبول است زیرا در واقع کمترین فاصله تا دورترین غذا را میدهد و چون این غذا حتما باید خورده شود یا از همین مسیر استفاده میکنیم یا از مسیر دورتر. از طرفی این هیوریستیک سازگار هم می باشد زیرا مقدار f در مسیر افزایش میابد

سوال: پیاده سازی خودتان در این بخش و بخش قبل را مقایسه کنید.

در قسمت قبل از فاصله منهتن استفاده کردیم ولی در این بخش از فاصله ماربیج استفاده میکنیم. از طرفی در این بخش برای اینکه در زمان اجرای محاسبه فاصله ها صرفه جویی شود، میتوان صرفاً فاصله مازی دو یا تعدادی از غذا هارا از موقعیت فعلی پکمن حساب شده است. همچنین مشاهده میشود که هیوریتسک اول تعداد گره های کمتری را باز میکند.

```
(base) mahdieh@mahdiehs-MBP P1 % python 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 0.9 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
(base) mahdieh@mahdiehs-MBP P1 % python pacman.py -l trickySearch -p AStarFoodSearchAgent
Path found with total cost of 60 in 5.0 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
```

سوال: با توجه به کدتان و مطالب درس بیان کنید که چرا به A^* الگوریتم جستجوی آگاهانه میگویند؟

الگوریتم A^* یک الگوریتم جستجوی آگاهانه است زیرا از تابع هیوریستیک برای هدایت جستجوی خود استفاده می کند.

در الگوریتم A^* ، به هر گره هزینه ای اختصاص داده می شود که جمع هزینه رسیدن به گره از گره شروع (معروف به "g-value") و تخمینی از هزینه برای رسیدن به گره هدف از گره فعلی (معروف به "h-value") است. سپس الگوریتم گرهها را با کمترین هزینه گسترش می دهد (هزینه مجموع g-value و h-value است)

h-value مقدار هیوریستیکی است که تخمینی از نزدیک بودن یک گره معین به گره هدف ارائه می دهد. تابع هیوریستیک باید قابل قبول باشد، به این معنی که هرگز هزینه واقعی برای رسیدن به هدف را بیش از حد برآورد نکند.

از آنجایی که A^* هم از هزینه واقعی و هم از تابع هیوریستیک برای هدایت جستجوی خود استفاده می کند، می تواند کوتاه ترین مسیر را برای رسیدن به هدف موثرتر از الگوریتم های جستجوی ناآگاه که فقط هزینه واقعی را در نظر می گیرند، بیابد. بنابراین، A^* یک الگوریتم جستجوی آگاهانه در نظر گرفته می شود.

• بخش 8: جستجوی نیمه بهینه

پیاده سازی کد:

پیاده سازی IDS تا حدی زیادی شبیه DFS است با این تفاوت که هر بار تا عمق محدودی DFS را انجام میدهیم و اگر به نتیجه نرسیدیم عمق را بیشتر میکنیم و ادامه میدهیم به همین علت لازم است عمق را نیز در حالتی که در Stack ذخیره میکنیم نگه داریم و به ازای بررسی هر نود علاوه بر هدف بودن آن عمق آن هم با عمق مجاز فعلی مقایسه کنیم.

سوال: ClosestDotSearchAgent شما، همیشه کوتاهترین مسیر ممکن در مارپیچ را پیدا نخواهد کرد. مطمئن شوید که دلیل آن را درک کرده‌اید و سعی کنید یک مثال کوچک بیاورید که در آن رفتن مکرر به نزدیکترین نقطه منجر به یافتن کوتاهترین مسیر برای خوردن تمام نقاط نمیشود.

این الگوریتم همیشه کوتاه ترین مسیر را پیدا نمیکند و ممکن است اگر الان نقطه ای را بخورد ضرر کمی کند اما بعداً از برگشت ها و طی کردن های مسیر های طولانی دیگر جلوگیری کند.

برای مثال در شکل های زیر الگوریتم به ما میگویند که ابتدا غذاهای سمت راست صفحه را بخوریم که در این صورت مجبور است مسیر طولانی را برای خوردن دو نقطه باقی مانده باز گردد، در صورتی که اگر ابتدا غذای بالایی را بخورد و بعد به سمت راست میرفت بهینه تر بود و مجبور به بازگشت نبود.

