دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)
دانشکده مهندسی کامپیوتر

مبانی و کاربردهای هوش مصنوعی

پروژه اول: پیاده سازی الگوریتم های جستجو در بازی پکمن

استاد: دکتر مهدی جوانمردی

دانشجو: سید علیرضا خسروی

زمستان ۱۴۰۳

چکیده

این پروژه به پیاده‌سازی و بررسی روش های مختلف جستجو برای حل مسئله مسیریابی در بازی پکمن اختصاص دارد. الگوریتم های مورد بررسی شامل جستجوی عمق اول، جستجوی سطح اول، جستجوی هزینه یکنواخت و جستجوی A* هستند. همچنین در این پروژه به پیاده‌سازی تابع هیوریستیک مناسب در الگوریتم جستجوی A* برای حل مسائل پیچیده تر پرداخته می‌شود. در نهایت در این پروژه الگوریتم جستجوی عمیق سازی تکرار شونده پیاده‌سازی می‌شود. زبان برنامه‌نویسی مورد استفاده در این پروژه برای پیاده‌سازی الگوریتم های خواسته شده پایتون است.

واژه‌های کلیدی: الگوریتم های جستجو، هیوریستیک، بهینه، هوش مصنوعی

فهرست مطالب

عنوان صفحه

فصل ۱: حل سوالات پروژه

پیدا کردن یک نقطه‌ی ثابت غذا با استفاده از DFS

۱.۱ پاسخ سوال ۱:

خیر برابر نیستند. زیرا به دلیل تفاوت در ساختمان داده استفاده شده، زمان اجرا یکسان نیست. در استفاده پشته عملیات حذف و اضافه از لحاظ زمانی O(1) هستند اما در صف اولویت به O(log n) تبدیل می‌شود.

۱.۲ پاسخ سوال ۲:

خانه هایی که قرمزترند زودتر توسط پکمن بررسی شده‌اند. در فایل graphicsDisplay.py و در تابع زیر رنگ خانه ها تنظیم می‌شوند.

```
def drawExpandedCells(self, cells):
    """
    Draws an overlay of expanded grid positions for search agents
    """
    n = float(len(cells))
    baseColor = [1.0, 0.0, 0.0]
    self.clearExpandedCells()
    self.expandedCells = []
    for k, cell in enumerate(cells):
        screenPos = self.to_screen( cell)
        cellColor = formatColor(*[(n-k) * c * .5 / n + .25 for c in baseColor])
        block = square(screenPos,
                       0.5 * self.gridSize,
                       color = cellColor,
                       filled = 1, behind=2)
        self.expandedCells.append(block)
        if self.frameTime < 0:
            refresh()
```

جستجوی اول سطح (BFS)

۱.۳ پاسخ سوال ۳:

بله. با اجرای این دستور، کد ما بدون هیچ تغییری مسئله ۸-پازل را حل می‌کند. در واقع در هر مرحله چون جابه‌جایی هر قطعه یا جابه‌جایی مربع خالی هزینه یک واحد را دارد و BFS هم سطح به سطح بررسی می‌کند(در واقع همان UCS با هزینه یک در هر گام است)، با کمترین هزینه به وضعیت هدف می‌رسد.

۱.۴ پاسخ سوال ۴:

با توجه به اینکه از نظر حافظه الگوریتم BFS (اگر راه حل در درخت جستجو در عمق d باشد و ضریب انشعاب برابر b باشد) $O(b^d)$ است، مقدار حافظه زیادی را احتیاج دارد. اگر عمق جواب زیاد باشد، با توجه به اینکه order زمانی الگوریتم مشابه order حافظه الگوریتم است، زمان زیادی صرف می‌شود تا گره هدف پیدا شود.

۱.۵ پاسخ سوال ۵:

اگر حافظه محدود باشد می‌توان با ترکیب کردن DFS با BFS، از الگوریتم iterative deepening search استفاده کنیم. در واقع در این الگوریتم از حسن order کمتر حافظه DFS نسبت به BFS استفاده می‌شود.

جستجوی UCS

۱.۶ پاسخ سوال ۶:

در الگوریتم UCS، عامل هیچ اطلاعاتی درباره هدف یعنی اینکه چقدر با هدف فاصله دارد، ندارد و صرفا هزینه طی شده از گره اولیه تا گره فعلی را در نظر دارد. اما الگوریتم A* با اضافه کردن یک تابع هیوریستیک، هزینه را از گره فعلی تا گره هدف پیش‌بینی کرده و با بررسی گره های کمتر، زودتر به هدف می‌رسد. به همین دلیل است که از الگوریتم هایی مانند A* استفاده می‌شود.

۱.۷ پاسخ سوال ۷:

برای اینکار کافی است که هزینه مسیر ها را تغییر دهیم. یعنی اینکه پکمن وقتی یک غذا را بخورد، آن را جریمه کنیم و هزینه مسیر طی شده را بر اساس تعداد غذاهای خورده شده افزایش دهیم. که در این صورت تابع getCostOfActions را تغییر می‌دهیم.

برای این منظور یک تابع که تعداد غذاهای موجود در یک مسیر که توسط تعدای action دنبال می‌شود را می‌شمارد. شبه کد این تابع و بعد شبه کد تابع قبلی در زیر قرار گرفته است.

```
def countFood(actions):
    numberOfFoods = 0
    state = initialState()   ---> the first state of our problem
    for action in actions:
        state = getNextState(state, action)
        if isFood(state):  ------> this function is used to detect a food in the state.
            numberOfFoods += 1
    return numberOfFoods
```

```
def getCostFunction(actions):
    cost = len(actions)
    numberOfFoods = countFood(actions)
    cost += 10 * numberOfFoods    ---->   we define a cost for eating a food. for example 10.
    return cost
```

جستجوی A*

۱.۸ پاسخ سوال ۸:

نتایج برای الگوریتم های A*، BFS، UCS و DFS به ترتیب به شرح زیر هستند:

```
gent -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 0.0 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
```

```
gent -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
```

```
gent -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
```

```
gent -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: 576
Pacman emerges victorious! Score: 212
Average Score: 212.0
Scores:        212.0
Win Rate:      1/1 (1.00)
Record:        Win
```

همان‌طور که پیدا است، پکمن با الگوریتم *A تعداد گره های کمتری را برای رسیدن به هدف بسط داده است. الگوریتم های BFS و UCS به دلیل اینکه هزینه مسیر ها در الگوریتم UCS یکی هستند، مانند هم عمل کردند و مثل الگوریتم *A امتیاز یکسانی کسب کرده و مسیر بهینه را نیز پیدا کرده اند اما تعداد گره های بیشتری را بسط داده اند. در مقابل DFS با سرعت و امتیاز کمتری توانسته به هدف برسد و مسیر بهینه را پیدا نکرده است. اما یک حسنی در مقابل BFS و UCS داشته است که آن این است که تعداد گره های کمتری را بسط داده است. نکته ای که مهم است این است که به نظر می‌آمد امتیاز *A بیشتر از الگوریتم های BFS و DFS بشود اما احتمالا به دلیل زمان گرفته شده برای محاسبه تابع هیوریستیک، امتیاز این الگوریتم از دو الگوریتم گفته شده بیشتر نشده است.

۱.۹ پاسخ سوال ۹:

بله. می‌توان از این الگوریتم در محیط های تغییر پذیر استفاده کرد اما باید تغییراتی در آن ایجاد شود. یکی از این تغییرات این است که مسیری که توسط این الگوریتم پیدا می‌شود با توجه به شرایط محیط به‌روزرسانی شود. با توجه به تغییرات محیط بتوانیم تابع هیوریستیک را برای تخمین دقیق تر هدف به‌روز کنیم. در رابطه با این به‌روزرسانی ها می‌توانیم از الگوریتم های یادگیری در *A استفاده کنیم.

یکی دیگر از کارهایی که می‌توان انجام داد این است که مسیری را بدون تغییرات محیط تخمین بزنیم و سپس با توجه به تغییرات محیط، آن مسیر را به‌روزرسانی کنیم.

۱.۱۰ پاسخ سوال ۱۰:

یکی از این هیوریستیک ها، هیوریستیک Chebyshev distance است که به صورت زیر برای گره‌ای مانند n محاسبه می‌شود.

$$h(n) = max(|\ x_{goal} - x_n\ |, |\ y_{goal} - y_n\ |)$$

هیوریستیک دیگری که می‌توان استفاده کرد، weighted heuristics است که به صورت زیر برای گره‌ای مانند n محاسبه می‌شود.

$$h(n) = w \times h(n)$$

که در رابطه بالا، تابع h می‌تواند هر هیوریستیک دلخواهی مانند Chebyshev distance و Manhattan distance باشد. در اینجا اگر وزن ما برابر یک باشد، هیوریستیک ما مانند یکی از هیوریستیک هایی که تا الان گفته شد، عمل می‌کند.

هرچه قدر مقدار وزن از یک بیشتر شود، هیوریستیک ما الگوریتم را حریصانه تر می‌کند و باعث می‌شود مسیر رسیدن به هدف زودتر پیدا شود اما از احتمال بهینه بودن آن کم می‌کند. در مقابل هرچه قدر مقدار وزن از یک کمتر شود و به صفر نزدیک تر شود، بیشتر شبیه الگوریتم Dijkstra عمل می‌کند. این یعنی احتمال یافتن مسیر بهینه بیشتر می‌شود اما چون گره های بیشتری را بسط می‌دهد، زمان یافتن مسیر بهینه را افزایش می‌دهد.

در این الگوریتم وزن نمی‌تولند منفی باشد زیرا اگر مقدار منفی به وزن بدهیم، مجموع هزینه واقعی و هزینه تابع هیوریستیک می‌تواند از صفر کمتر بشود و می‌تواند مسیری نادرست را برگرداند. خلاصه این الگوریتم در واقع یک trade-off بین سرعت اجرای الگوریتم و بهینه بودن جواب نهایی الگوریتم است.

در این سوال هیوریستیک Chebyshev distance را انتخاب می‌کنیم. کد آن به صورت زیر است:

```python
def chebyshevHeuristic(position, problem, info={}):  ## ali_heuristic
    xy1 = position
    xy2 = problem.goal
    return max(abs(xy1[0] - xy2[0]), abs(xy1[1] - xy2[1]))
```

نتایج برای سه هیوریستیک Manhattan distance، Euclidean distance و Chebyshev distance روی openMaze به ترتیب به شرح زیر هستند:

```
gent -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 0.0 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
```

```
gent -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 0.0 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
```

```
gent -a fn=astar,heuristic=chebyshevHeuristic
[SearchAgent] using function astar and heuristic chebyshevHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.0 seconds
Search nodes expanded: 568
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```

با توجه به نتایج، امتیاز پکمن با استفاده از هر سه هیوریستیک ثابت است. اما هیوریستیک Manhattan، کمتر از بقیه گره بسط داده است. هیوریستیک Chebyshev بیشتر از بقیه گره بسط داده است و هیوریستیک Euclidean مقدار گره‌ای بین دو هیوریستیک قبلی بسط داده است. این یعنی هیوریستیک Manhattan زمان کمتری نسبت به بقیه هیوریستیک ها برای پیدا کردن مسیر بهینه گذاشته است و بهترین عملکرد را دارد و هیوریستیک Chebyshev زمان بیشتری نسبت به بقیه هیوریستیک ها برای پیدا کردن مسیر بهینه گذاشته است و بدترین عملکرد را دارد.

پیدا کردن همه گوشه‌ها

خروجی الگوریتم DFS برای این مسـئله در tinyCorners و mediumCorners به ترتیب به صـورت زیر است:

```
[SearchAgent] using function dfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 47 in 0.0 seconds
Search nodes expanded: 51
Pacman emerges victorious! Score: 493
Average Score: 493.0
Scores:        493.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
[SearchAgent] using function dfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 221 in 0.0 seconds
Search nodes expanded: 371
Pacman emerges victorious! Score: 319
Average Score: 319.0
Scores:        319.0
Win Rate:      1/1 (1.00)
Record:        Win
```

با توجه به نتایج بالا، از آن جهت که DFS الگوریتم بهینه ای نیسـت(یعنی کوتاه‌ترین مسـیر را پیدا نمی‌کند)، پکمن برای پیدا کردن همه گوشه‌ها نسبتا تعداد زیادی گره را بسط داده است. هرچه قدر هم maze پیچیده تر شود، به دلیل بهینه نبودن الگوریتم DFS، تعداد گره های بسط داده شده بیشتر شده و امتیاز پکمن در انتهای بازی کمتر می‌شـود. در اینجا نیاز به الگوریتم های بهتری مانند A*(البته با هیوریستیک مناسب) احساس می‌شود.

هیوریستیک برای مسئله گوشه‌ها

۱.۱۱ پاسخ سوال ۱۱:

هیوریستیکی که در این بخش پیاده‌سازی شده است، بیشترین فاصله Manhattan پکمن از گوشه ها را حساب می‌کند(اگر کمترین را لحاظ کنیم، تعداد گره ی بیشتری را بسط می‌دهد.). این هیوریستیک سازگار است، زیرا فاصله Manhattan در هر حرکت پکمن به دلیل نحوه حساب کردن آن، حداکثر یک واحد تغییر می‌کند. اگر تغییری نکند یا یک واحد بیشتر شود، شرط سازگاری برقرار است. اگر یک واحد کم شود، به دلیل اینکه هزینه هر حرکت پکمن در مارپیچ ثابت و برابر یک است، باز هم شرط سازگاری برقرار است(شرط سازگاری این است که تفریق هیوریستیک حالت جدید از هیوریستیک حالت قبل حداکثر یک واحد باشد). بنابراین هیوریستیک ما سازگار است.

۱.۱۲ پاسخ سوال ۱۲:

درهیوریستیک پیاده‌سازی شده، یکی از پارامتر هایی که لحاظ شده است، فاصله Manhattan پکمن تا هرکدام از گوشه های دیده نشده است. همان طور که گفته شد، این هیوریستیک فاصله Manhattan تا هر کدام از گوشه های دیده نشده را محاسبه می‌کند و بیشترین مقدار آن را برمی‌گرداند.

یکی از پارامتر هایی که می‌توان درنظر گرفت، موانع است. یعنی اینکه بتوانیم با در نظر گرفتن موانع موجود در مارپیچ، تخمین دقیق تری نسبت به فاصله Manhattan داشته باشیم. برای اینکار می‌توانیم از تابع mazeDistance بهره ببریم. قسمت تغییر یافته کد به شکل زیر است. همچنین attribute جدیدی با نام startingGameState به کلاس CornersProblem به شکل زیر اضافه شده است.

```
heuristic = max(mazeDistance(pacman_position, corner, problem.startingGameState) for corner in unvisited_corners)
```

```
self.startingGameState = startingGameState  ## I added
```

نتیجه استفاده از آن به صورت زیر است:

```
Path found with total cost of 106 in 6.0 seconds
Search nodes expanded: 801
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

با توجه به تصویر بالا، با استفاده از تابع mazeDistance، تعداد گره های کمتری بسط داده شده است.

خوردن همه غذاها

۱.۱۳ پاسخ سوال ۱۳:

در این اینجا، تابع هیوریستیک فاصله پکمن تا هرکدام از غذاها را با درنظر گرفتن موانع(یعنی با استفاده از تابع mazeDistance)، محاسبه می‌کند و بیشترین مقدار در میان این فاصله ها را برمی‌گرداند. این هیوریستیک سازگار است، زیرا فاصله maze در هر حرکت پکمن به دلیل نحوه حساب کردن آن، حداکثر یک واحد تغییر می‌کند. اگر تغییری نکند یا یک واحد بیشتر شود، شرط سازگاری برقرار است. اگر یک واحد کم شود، به دلیل اینکه هزینه هر حرکت پکمن در مارپیچ ثابت و برابر یک است، باز هم شرط سازگاری برقرار است(شرط سازگاری این است که تفریق هیوریستیک حالت جدید از هیوریستیک حالت قبل حداکثر یک واحد باشد). بنابراین هیوریستیک ما سازگار است.

۱.۱۴ پاسخ سوال ۱۴:

در اولین پیاده‌سازی تابع هیوریستیک قسمت قبل، از تابع manhattanDistance استفاده شد. در این هیوریستیک موانع(یعنی دیوار ها) در نظر گرفته نشدند اما در مسئله خوردن همه غذاها، موانع درنظر گرفته شده است که این یعنی هزینه پیش‌بینی شده برای رسیدن به هدف دقیق تر است.

در پیاده‌سازی دوم قسمت قبل، مانند مسئله خوردن همه غذا ها، از تابع mazeDistance استفاده شد که مانند این قسمت، تخمین دقیق تری برای رسیدن به هدف را برمی‌گرداند.

۱.۱۵ پاسخ سوال ۱۵:

برای این پیاده‌سازی کافی است که در تابع foodHeuristic، از تابع min به جای max جلوی heuristic استفاده شود. نتایج را به ترتیب استفاده از تابع max و استفاده از تابع min در زیر مشاهده می‌کنید:

```
Path found with total cost of 60 in 17.5 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
```

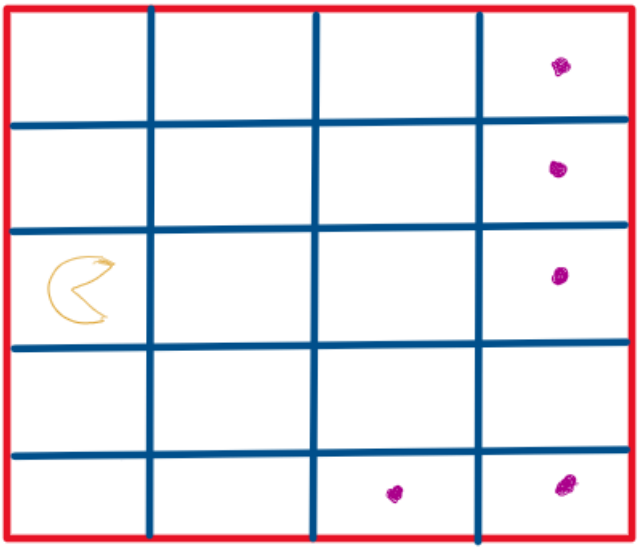
```
Path found with total cost of 60 in 57.6 seconds
Search nodes expanded: 12372
Pacman emerges victorious! Score: 570
Average Score: 570.0
Scores:        570.0
Win Rate:      1/1 (1.00)
Record:        Win
```

همان طور که از تصاویر پیدا است، اولویت قرار دادن جمع‌آوری نزدیک‌ترین غذاها به موقعیت شروع، باعث شده است که پکمن برای پیدا کردن مسیر بهینه، زمان بیشتری را صرف کند و حدود سه برابر بیشتر نسبت به حالت قبل، گره بسط بدهد. این اتفاق به این دلیل است که استفاده از تابع min تخمین نادقیق‌تری از هزینه رسیدن به هدف به پکمن می‌دهد. در واقع استفاده از تابع max، هزینه واقعی را دقیق‌تر تخمین می‌زند.

جستجوی نیمه بهینه

۱.۱۶ پاسخ سوال ۱۶:

همان‌طور که در متن پروژه ذکر شده است، این رویکرد در حل مسئله یعنی پیدا کردن نزدیک ترین غذا در هر مرحله همیشه نمی‌تواند کوتاه‌ترین مسیر را در مارپیچ پیدا کند. مثال نقض آن را در شکل زیر می‌بینید:



مطابق با این رویکرد، اگر پکمن همیشه به سمت نزدیک غذا برود، هزینه مسیر پیدا شده برابر ۱۰ خواهد بود. اما همان‌طور که از شکل پیدا است، پکمن باید برای پیداکردن مسیر بهینه اول باید به سمت غذا های پایین‌تر برود که در این صورت هزینه این مسیر برابر ۹ است.