تمرین اول هوش مصنوعی

امیرحسین رجبی (۹۸۱۳۰۱۳)

۱۴ اسفند ۱۴۰۰

سوال اول

آ)

ب)

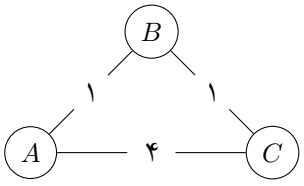
سوال دوم

آ) درست. بنابر توضیحات کتاب [۳، صفحه ۶۴ فصل دوم] یک بازی ویدیویی جدید ممکن است صفحه بازی برای ما به طور کامل قابل مشاهده باشد ولی ندانیم که با فشردن هر دکمه چه اتفاقی خواهد افتد.

ب) درست. تابع transition در این مدل‌ها روی فضای متناهی تعریف می‌شود و در فضاهای پیوسته پاسخگو نیست.

ج) درست. اگر در الگوریتم A^* تابع هیوریستیک برابر تابع ثابت صفر باشد تابع ارزیابی (f) همان تابع هزینه (g) خواهد بود.

د) نادرست. گراف زیر را در نظر بگیرید. کوتاهترین مسیر از گره A به C از طریق B خواهد بود. اگر $C = ۳$ را به تمام یال‌ها اضافه کنیم، کوتاهترین مسیر یال AC خواهد بود.



ه) درست. می‌دانیم الگوریتم A^* همه گره‌ها مانند x را بسط می‌دهد که هزینه ارزیابی آنها حداکثر هزینه ارزیابی گره هدف باشد؛ یعنی اگر n گره هدف باشد، $f(x) \leq f(n) = g(n)$. اما چون $g(x) \leq g(x) + h(x) = f(x)$ پس $g(x) \leq g(n)$. بنابر شرایط گفته شده $g(n)$ بهینه است و در نتیجه گره x توسط الگوریتم Uniform Cost Search نیز بسط داده می‌شود چرا که این الگوریتم همه گره‌ها با هزینه کمتر از $g(n)$ را بسط می‌دهد.

و) نادرست. مشکل اصلی A^* پیچیدگی زمانی آن نیست بلکه پیچیدگی فضای مصرفی آن زیاد است. (همان طور که الگوریتم IDS برای کاهش فضای مصرفی الگوریتم BFS و completeness الگوریتم DFS ارائه شد.)

سوال سوم

با توجه به شماره دانشجویی مسئله چهارم را حل می‌کنم. درستی گزاره را ثابت می‌کنیم. فرض کنیم گره v مجاور گره u باشد و هزینه انتقال از u به v برابر c باشد. چون x و y تابع‌هایی consistent هستند، خواهیم داشت:

$$x(u) \leq c + x(v)$$
$$y(u) \leq c + y(v)$$

نامساوی اول را در α و نامساوی دوم را در $1 - \alpha$ ضرب می‌کنیم: (دقت کنید به دلیل برقراری شرط $0 < \alpha < 1$ می‌توانیم بدون تغییر جهت نامساوی این کار را انجام دهیم.)

$$\alpha x(u) \leq \alpha c + \alpha x(v)$$
$$(1 - \alpha)y(u) \leq (1 - \alpha)c + (1 - \alpha)y(v)$$

با جمع روابط بالا داریم:

$$\alpha x(u) + (1 - \alpha)y(u) \leq c + \alpha x(v) + (1 - \alpha)y(v)$$

که حکم نتیجه می‌شود.

سوال چهارم

مسئله اول را حل می‌کنیم.

آ) هر حالت قرارگیری n پکمن در m خانه نقشه یک حالت یا state خواهد بود. همچنین بین هر دو حالت reachable یک transition وجود دارد.

ب) با توجه به بخش قبل و فرض مسئله که امکان قرارگیری چندین پکمن در یک خانه از نقشه وجود دارد، می‌توان گفت اندازه فضای حالت برابر m^n خواهد بود چرا که هر پکمن می‌تواند در یکی از m خانه نقشه قرار گیرد.

ج) در صورتی که موقعیت همه پکمن‌ها به گونه‌ای باشد که بتوانند در هر چهار جهت بالا، پایین، چپ و راست حرکت کنند و همچنین بتوانند در جای خود باقی بمانند، تعداد یال‌های خروجی هر حالت برابر $b \leq 5^n - 1$ خواهد بود. (دقت کنید حداقل یک پکمن باید حرکت کند. در غیر این صورت درجا می‌زنیم. یک واحد به این دلیل کسر شده است.) به وضوح در صورت قرارگیری پکمن‌ها در حاشیه و مرز نقشه تعداد یال‌های خروجی آن حالت یا branching factor از b کمتر خواهد بود. همچنین کران پایین $2^n - 1 \leq b$ را نیز داریم چرا که هر پکمن یا یک خانه مجاور دارد (اگر نقشه همبند باشد.) یا می‌تواند در جای خود باقی بماند.

د) در این مسئله تابع هزینه گام‌های مسئله [1] عددی صحیح بین ۱ و n است چرا که حداقل یک پکمن یک گام حرکت کرده و همچنین حداکثر همه پکمن‌ها هر کدام یک گام حرکت می‌کنند. می‌دانیم در الگوریتم Uniform Cost Search حداکثر همه گره‌هایی بسط داده می‌شوند که مسیر بهینه آنها هزینه‌ای کمتر از پاسخ بهینه مسئله (C^*) را دارد. چون تابع هزینه حداقل $\epsilon = 1$ است پس حداکثر همه گره‌ها تا عمق $d = \lfloor \frac{C^*}{\epsilon} \rfloor + 1 = C^* + 1$ [2] بسط داده می‌شوند؛ یعنی حداکثر $b + b^2 + \cdots + b^d = \frac{b(b^d - 1)}{b - 1}$ گره. با توجه به بخش قبل b کران پایین بزرگی دارد و می‌توانیم فرض کنیم عبارت بالا از b^d کمتر است. از طرفی در این مسئله هر گره هدف حداکثر در عمق $m - 1$ خواهد بود زیرا محدودیتی برای تعداد پکمن‌هایی که می‌توانند همزمان در یک خانه از نقشه قرار داشته باشند نداریم و هر پکمن با پیمایش این تعداد گام همه‌ی خانه‌های نقشه را یک بار مشاهده می‌کند. همچنین تابع هزینه هر گام حداکثر n خواهد بود. پس $C^* \leq n(m - 1)$. به کمک بخش قبل داریم $b^d \leq (5^n - 1)^{n(m-1)+1}$.

[1] Step cost function

[2] دلیل وجود یک در این عبارت این است که الگوریتم UCS زمانی خاتمه می‌یابد که گره هدف را از صف اولویت خارج کند و قبل از بسط آن، هدف بودن آن را بررسی می‌کند و نه زمانی که آن را ایجاد می‌کند.

ه) این تابع هر دو شرایط را دارد. ابتدا ثابت می‌کنیم تابع admissible است؛ یعنی تابع هیوریستیک، از هزینه واقعی بیشتر نیست. اگر مستطیل $R = [X_1, X_2] \times [Y_1, Y_2]$ مستطیلی با کمترین مساحت باشد که همه پکمن‌ها درون یا روی آن قرار بگیرند، (چپ‌ترین پکمن روی خط $x = X_1$، راست‌ترین پکمن روی خط $x = X_2$، پایین‌ترین پکمن روی خط $y = Y_1$ و بالاترین پکمن روی خط $y = Y_2$ قرار می‌گیرند.) تابع هیوریستیک برابر $h = \frac{1}{2}\max(X_2 - X_1, Y_2 - Y_1)$ خواهد بود. بدون کاسته شدن از کلیت مسئله اگر $h = \frac{1}{2}(X_2 - X_1)$ و همچنین جواب بهینه برای این مسئله، ملاقات پکمن‌ها در نقطه (α, β) باشد، در این صورت اگر $\alpha \leq \frac{X_1+X_2}{2}$، فاصله منهتن [3] پکمن روی خط $x = X_2$ حداقل h خواهد بود و اگر $\alpha \geq \frac{X_1+X_2}{2}$، فاصله منهتن پکمن روی خط $x = X_1$ حداقل h خواهد بود. به طوری مشابه برای $h = \frac{1}{2}(Y_2 - Y_1)$ نیز ثابت می‌شود ک h کران پایینی برای هزینه واقعی حل مسئله خواهد بود.

اکنون ثابت می‌کنیم تابع هیوریستیک consistent است. اگر v گره‌ی مجاور گره‌ی u باشد و هزینه انتقال از u به v برابر c باشد، نشان می‌دهیم $h(u) \leq h(v) + c$. مستطیل‌های R_u و R_v را مانند اثبات قبل به ترتیب برای وضعیت‌های u و v تعریف می‌کنیم. (مستطیل‌هایی با کمترین مساحت و اضلاع موازی محورهای مختصات که پکمن‌ها درون یا روی آنها قرار دارند.) همچنین اندازه طول مستطیل R را با $L(R)$ و اندازه عرض آن را با $W(R)$ نشان می‌دهیم. بدون کاسته شدن از کلیت مسئله فرض کنیم $L(R_u) \geq W(R_u)$ و در نتیجه $h(u) = \frac{L(R_u)}{2}$. چون طول گام پکمن‌ها یک واحد است، $|L(R_u) - L(R_v)| \leq 2$ و $|W(R_u) - W(R_v)| \leq 2$ چرا که پکمن‌های روی طول یا عرض R_u می‌توانند به سمت یکدیگر یا خلاف همدیگر حرکت کنند. در نتیجه $|h(u) - h(v)| \leq 1$. اگر $h(u) - h(v) = 1$، چون $c \geq 1$ آنگاه $h(u) \leq h(v) + c$. در غیر این صورت (مثلا $h(v) \geq h(u)$) رابطه $h(u) < h(v) + c$ برقرار خواهد بود. پس در هر صورت داریم $h(u) \leq h(v) + c$.

گزارش پروژه

بخش اصلی پروژه در ماژول puzzle.py قرار دارد و ماژول priority_queue.py پیاده سازی صف اولویت است که مستقیما از کدهای کتاب ساختمان داده گودریخ و تاماسیا [۱، فصل ۹] که در اینجا قرار داده شده‌اند استفاده شده است. (دلیل عدم استفاده از کتابخانه heapq در ادامه گفته می‌شود.) برای پیاده سازی از سه کلاس State، Node و Puzzle استفاده شده است. نمونه‌های کلاس State همگی hashable هستند زیرا لازم است بتوان آنها را به عنوان کلید در ساختمان داده‌های dict و set به کار برد. بدین منظور توابع __eq__() و __hash__() آنها override شده‌اند. تابع __hash__() ماتریس حالت را به tuple تبدیل می‌کند و سپش هش آن را برمی‌گرداند. (شایان ذکر است که برای کاسته شدن از سربار تبدیل ماتریس حالت به چندتایی، آن را یک بار و هنگام مقدار دهی اولیه ایجاد کرده و ذخیره می‌کنیم.) تابع find_position() که public است، موقعیت هر سلول (در این پیاده سازی سلول‌ها می‌تواند عدد، رشته یا هر شی دیگری باشند.) که به آن ورودی داده شود را محاسبه می‌کند. سلول خالی را با None نشان می‌دهیم. نمونه‌های کلاس State را می‌توان به کمک ماتریس حالت ایجاد کرد. تابع actions() که protected است، حرکات مجاز سلول خالی را به صورت لیستی از Actionها برمی‌گرداند. سپس تابع adjacent_states() که public است، حالت‌هایی که به کمک این حرکات مجاز قابل دسترسی هستند را به صورت لیستی از Stateها برمی‌گرداند.

```
1  class State:
2      def __init__(self, state_matrix: List[List]):
3          union = []
4          for row in state_matrix:
5              union.extend(row)
6          self.__state_tuple: tuple = tuple(union)
7          self.__state_matrix = state_matrix
8
9      def to_tuple(self) -> tuple:
10         return self.__state_tuple
11
12     def __hash__(self):
13         return hash(self.__state_tuple)
```

[3] Manhattan distance

```
14
15      def __eq__(self, other):
16          return other.to_tuple() == self.__state_tuple
17
18      def find_position(self, x) -> (int, int):
19          for i in range(len(self.__state_matrix)):
20              if x in self.__state_matrix[i]:
21                  return i, self.__state_matrix[i].index(x)
22
23      def __actions(self) -> List[Action]:
24          i, j = self.find_position(None)
25          n = len(self.__state_matrix) - 1
26          result = []
27          if j != 0:
28              result.append(Action.Left)
29          if j != n:
30              result.append(Action.Right)
31          if i != 0:
32              result.append(Action.Up)
33          if i != n:
34              result.append(Action.Down)
35          return result
36
37      def adjacent_states(self):
38          i, j = self.find_position(None)
39          result = []
40          for action in self.__actions():
41              x = deepcopy(self.__state_matrix)
42              if action == Action.Up:
43                  x[i - 1][j], x[i][j] = x[i][j], x[i - 1][j]
44              elif action == Action.Down:
45                  x[i + 1][j], x[i][j] = x[i][j], x[i + 1][j]
46              elif action == Action.Right:
47                  x[i][j + 1], x[i][j] = x[i][j], x[i][j + 1]
48              elif action == Action.Left:
49                  x[i][j - 1], x[i][j] = x[i][j], x[i][j - 1]
50              result.append((State(x), action))
51          return result
```

کلاس Node نماینگر یک گره است که برای پیمایش الگوریتم A^* به کار می‌رود و هزینه هیوریستیک، کل هزینه تخمین زده شده و همچنین هزینه مسیر از گره شروع تا گره فعلی را ذخیره سازی می‌کند. آدرس گره والد، حالت مسئله در گره فعلی و نوع حرکت انجام شده برای انتقال از حالت متناظر گره والد به حالت نظیر گره فعلی نیز در این کلاس ذخیره می‌شود.

```
1   class Node:
```

```
    def __init__(self, state: State, g_cost: int, h_cost: int,
                 parent, action: Action | None):
        self.parent = parent
        self.state: State = state
        self.f_cost: int = g_cost + h_cost
        self.g_cost: int = g_cost
        self.h_cost: int = h_cost
        self.action: Action = action
```

نمونه‌های کلاس Puzzle با دو ماتریس متناظر حالت شروع و حالت هدف مقدار دهی اولیه می‌شوند. این ماتریس‌ها بلافاصله به صورت State ذخیره می‌شوند. همچنین یک dict از سلول‌ها و موقعیت آنها در ماتریس حالت هدف ایجاد می‌شود. کاربرد این dictionary در تابع heuristic(state: State) است که private است. در واقع تابع هیوریستیک با ورودی گرفتن یک حالت، به کمک تابع manhattan_distance(t1: (int, int), t2: (int, int)) مجموع فاصله منهتن بین هر سلول در حالت هدف و حالت داده شده state را محاسبه می‌کند.

```
class Puzzle:
    def __init__(self, initial_state_matrix: List[List],
                 goal_state_matrix: List[List]):
        self.__initial_state = State(initial_state_matrix)
        self.__goal_state = State(goal_state_matrix)
        self.__goal_state_cells_positions = {
            cell: self.__goal_state.find_position(cell)
            for cell in self.__goal_state.to_tuple() if cell is not None
        }

    def __heuristic(self, state: State) -> int:
        s = 0
        for cell, goal_position in self.__goal_state_cells_positions.items():
            s += self.__manhattan_distance(
                state.find_position(cell),
                goal_position
            )
        return s

    @staticmethod
    def __manhattan_distance(t1: (int, int), t2: (int, int)) -> int:
        return abs(t1[0] - t2[0]) + abs(t1[1] - t2[1])
```

الگوریتم A^* در تابع solve() کلاس Puzzle پیاده سازی شده است که کد آن را در زیر می‌بینید. این الگوریتم مانند الگوریتم Uniform Cost Search یا Dijkstra پیاده سازی شده است.

```
def solve(self) -> List[Node] | None:
    frontier = AdaptableHeapPriorityQueue()
    explored = set()
    frontier_nodes_by_states = {}
```

```
    frontier_locators_by_nodes = {}
    initial_node = Node(
        self.__initial_state,
        0,  # initial g_cost is 0
        self.__heuristic(self.__initial_state),
        None,
        None
    )
    add_to_frontier(initial_node.f_cost, initial_node)
    while not frontier.is_empty():
        key, node = frontier.remove_min()
        remove_from_frontier_finders(node)
        if self.__is_goal_state(node.state):
            return self.__solution(node)
        explored.add(node.state)
        for child in self.__children(node):
            if child.state not in explored and \
                    child.state not in frontier_nodes_by_states:
                add_to_frontier(child.f_cost, child)
            elif child.state in frontier_nodes_by_states:
                suspect = frontier_nodes_by_states[child.state]
                if suspect.f_cost > child.f_cost:
                    suspect_locator = frontier_locators_by_nodes[suspect]
                    remove_from_frontier_finders(suspect)
                    update_frontier(suspect_locator, child.f_cost, child)
    return None
```

همان طور که مشخص است دو dictionary به نام‌های frontier_nodes_by_states و frontier_locators_by_nodes تعریف شده است که اولی اجازه می‌دهد گره‌ای درون صف اولیت که نظیر یک حالت از مسئله است را پیدا کنیم. (خط ۲۲ را ببینید.) دومی اجازه می‌دهد به locator یک گره درون صف اولویت دسترسی پیدا کنیم. (خط ۲۷ را ببینید.) در واقع به کمک locator در پیاده سازی AdaptableHeapPriorityQueue() می‌توان کلید یک گره در صف اولویت را بروزرسانی کرد یا یک گره دلخواه را از صف اولویت خارج کرد. این همان دلیل استفاده از این صف اولویت به جای کتابخانه heapq است. در واقع در صورت برقراری شرط خط ۲۶، مسیری با هزینه کمتری برای گره suspect پیدا شده است. در نتیجه به کمک locator آن، این گره از صف اولویت خارج شده و گره child که همان وضعیت مسئله را دارد ولی هزینه کمتری دارد جایگزین آن می‌شود. (این عملیات به صورت بروزرسانی همزمان کلید و گره در خط ۲۹ پیاده سازی شده است.) همچنین چهار inner function به نام‌های add_to_frontier، add_to_frontier_finders، remove_from_frontier_finders و update_frontier برای هماهنگی frontier، frontier_nodes_by_states و frontier_locators_by_nodes نوشته شده‌اند.

```
def add_to_frontier_finders(locator, node: Node):
    frontier_nodes_by_states[node.state] = node
    frontier_locators_by_nodes[node] = locator

def add_to_frontier(f_cost: int, node: Node):
    locator = frontier.add(f_cost, node)
```

```
    add_to_frontier_finders(locator, node)

def remove_from_frontier_finders(node: Node):
    del frontier_nodes_by_states[node.state]
    del frontier_locators_by_nodes[node]

def update_frontier(locator, f_cost: int, node: Node):
    frontier.update(locator, f_cost, node)
    add_to_frontier_finders(locator, node)
```

هنگامی که الگوریتم گره هدف را از صف اولویت خارج کند یعنی همه گره‌ها با هزینه کوچکتر از $f(goal)$ گسترش داده شده‌اند و مسیر بهینه گره هدف پیدا شده است.

در این صورت به کمک تابع `solution(goal_node: Node)` و تابع بازگشتی `recursively_find_solution(node: Node)` مسیر حرکت از گره شروع به گره هدف را به کمک آدرس گره والد پیدا می‌کنیم.

```
def __recursively_find_solution(self, node: Node) -> List[Node]:
    if node.parent is None:
        return [node]
    path = self.__recursively_find_solution(node.parent)
    path.append(node)
    return path

def __solution(self, goal_node: Node) -> List[Node]:
    return self.__recursively_find_solution(goal_node)
```

همچنین تابع `children(parent_node: Node)` گره‌های مجاور گره والد را به کمک تابع `adjacent_states()` ایجاد می‌کند.

تابع `child_node(parent_node: Node, child_state: State, action: Action)` با ورودی گرفتن گره والد و حالت فرزند، یک گره با هزینه‌های $g(child) = g(parent) + ۱$ و $f(child) = g(child) + h(child)$ برمی‌گرداند.

```
def __children(self, parent_node: Node) -> List[Node]:
    return [
        self.__child_node(parent_node, child_state, action)
        for child_state, action in parent_node.state.adjacent_states()
    ]

def __child_node(self, parent_node: Node, child_state: State,
                 action: Action) -> Node:
    return Node(
        child_state,
        parent_node.g_cost + 1,  # step cost is 1
        self.__heuristic(child_state),
        parent_node,
        action
    )
```

چون اینجا الگوریتم A^* را روی گراف حالت‌ها پیاده سازی کردیم و بنابر قضیه کتاب [۲، صفحه ۹۵ فصل سوم] برای داشتن جواب بهینه باید تابع هیوریستیک consistent باشد. اثبات این موضوع ساده است چرا که اگر v گره مجاور u باشد، هزینه انتقال از u به v برابر یک خواهد بود و همچنین $h(v)$ حداکثر یک واحد کمتر از $h(u)$ خواهد بود زیرا فاصله منهتن عدد مجاور مربع خالی می‌تواند حداکثر یک واحد کاهش یابد یا به عبارتی نزدیک هدف شود و این حرکت در فاصله منهتن مابقی اعداد تاثیری ندارد.

مراجع

[1] M.T. Goodrich, R. Tamassia, and M.H. Goldwasser. *Data Structures and Algorithms in Python*. John Wiley & Sons, Incorporated, 2013.

[2] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall, 3rd edition, 2009.

[3] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, Global Edition*. Pearson series in artificial intelligence. Pearson, 4th edition, 2020.