دانشگاه صنعتی امیرکبیر
(پلی‌تکنیک تهران)
دانشکدهٔ ریاضی و علوم‌کامپیوتر

گزارش ۲: پیاده‌سازی روشی هیوریستیک برای یک مسئله و بررسی آن

نگارش
نیما حسینی دشت بیاض
استاد
دکتر مهدی قطعی

فروردین ۱۴۰۰

مقدمه

روش‌های جست‌وجو مانند DFS و BFS روش‌هایی ارزشمند هستند که رسیدن به جواب بهینه را تضمین می‌کنند. اما این روش‌ها به دلیل بررسی تمام حالات ممکن، زمان و حافظه‌ی بسیار زیادی اشغال می‌کنند که در بعضی مسائل رسیدن به جواب در یک زمان معقول را عملاً غیر ممکن می‌کند. از این‌جاست که اهمیت روش‌های هیوریستیک برای رسیدن به جواب بهینه پیدا می‌شود.

در این گزارش ابتدا یک مسئله را طرح می‌کنیم و سپس آن را به صورت هیوریستیک حل می‌کنیم و پیاده‌سازی آن در زبان پایتون را بررسی می‌کنیم.

طرح مسئله[1]

در این مسئله یک پارکینگ با اندازه‌ی $n \times n$ را در نظر می‌گیریم که n ماشین در خانه‌های $(1,1)$ تا $(n,1)$ پارک شده‌اند. می‌خواهیم این ماشین‌ها را به خانه‌های $(1,n)$ تا (n,n) منتقل کنیم. اما ماشینی که در جایگاه $(i,1)$ بوده است باید به خانه‌ی $(n-i+1,n)$ منتقل شود. به طور مثال اگر مقدار n برابر ۵ باشد. ماشینی که در جایگاه $(2,1)$ قرار داشته است، به جایگاه $(4,5)$ منتقل می‌شود. ماشین‌ها در هر واحد زمان می‌توانند یکی از ۵ عمل زیر را انجام دهند.

۱. حرکت به راست: از خانه‌ی (i,j) به $(i,j+1)$ حرکت کند.

۲. حرکت به چپ: از خانه‌ی (i,j) به $(i,j-1)$ حرکت کند.

۳. حرکت به بالا: از خانه‌ی (i,j) به $(i+1,j)$ حرکت کند.

۴. حرکت به پایین: از خانه‌ی (i,j) به $(i-1,j)$ حرکت کند.

۵. توقف: هیچ حرکتی انجام ندهد.

در هیچ مرحله‌ای از حرکت ماشین‌ها، دو ماشین نمی‌توانند در یک خانه قرار بگیرند. همچنین اگر یک ماشین در حالت توقف باشد، فقط یک ماشین دیگر می‌تواند با حرکت از کنار آن، دو خانه (به‌جای یکی) جابه‌جا شود.

هدف نهایی مسئله، رسیدن به خانه‌های نهایی کم‌ترین تعداد جابه‌جایی است.

---

۱ این مسئله از تمرین ۳.۲۷ کتاب Artificial Intelligence A Modern Approach, Third Edition اقتباس شده است.

حل مسئله

در ادامه سعی می‌کنیم این مسئله را حل کنیم.

راه‌حل با جست‌وجوی ناآگاهانه

یک راه حل ساده استفاده از نسخه‌های گرافی الگوریتم‌های BFS یا DFS است. باید توجه داشته باشیم که چون ماشین‌ها می‌توانند در خلاف جهت هدفشان حرکت کنند، گراف مسأله دارای دور خواهد بود و لازم است که از ایجاد گره‌های تکراری در درخت جلوگیری کنیم. با انجام این کار، تضمین می‌شود که الگوریتم‌های BFS و DFS حتماً به جواب می‌رسند و این جواب بهینه است.

با توجه به اینکه هر ماشین در هر واحد زمانی می‌تواند ۵ حرکت مختلف داشته باشد و در کل n ماشین داریم، می‌توان حدس زد که تعداد حالات فضای مسئله بسیار زیاد باشد و استفاده از الگوریتم‌های BFS و DFS که کل حالات را بررسی می‌کنند به‌صرفه نباشد. درواقع در هر مرحله، 5^n حرکت مختلف می‌توان داشته باشیم؛ هرچند که تعداد زیادی از این حرکات به دلیل محدودیت‌های مسأله قایل انجام نیستند که پیش از ایجاد گره در درخت، امکان‌پذیر بودن آن را بررسی کنیم. اگر مسئله relax شده را نظر بگیریم، در هر مرحله حداکثر می‌توان $b = 5^n$ حرکت انجام داد. کم‌ترین عمق جواب مسأله را می‌توان برابر $d = 2n$ در نظر گرفت چرا که ماشینی که در بالا سمت چپ قرار دارد باید تمام طول و عرض پارکینگ را طی کند تا به هدفش برسد. همچنین بیشترین عمق ممکن را می‌توان برابر $m = n^3$ قرار داد؛ چرا که می‌توان حالتی را در نظر گرفت که هر بار فقط یک ماشین حرکت کند و تمام خانه‌ها را برای رسیدن به هدفش پیمایش کند و بعد از آن که به هدف رسید، ماشین بعدی شروع به حرکت کند. پس در چنین شرایطی روش BFS دارای پیچیدگی زمان و حافظه‌ی $\mathcal{O}(b^d) = \mathcal{O}(5^{n^{2n}})$ خواهد بود که به‌شدت غیر بهینه است.

در ادامه سعی می‌کنیم روشی هیوریستیک برای حل مسئله ارائه دهیم.

راه‌حل هیوریستیک[۳۲]

در این بخش سعی می‌کنیم که با استفاده از روش A^* این مسئله را حل کنیم. در روش A^* لازم است تابع هیوریستیک h را تعریف کنیم که تخمینی از هزینه‌ی رسیدن به جواب نهایی را در حالت میانی نشان می‌دهد. همچنین تابع g باید تعریف شود که نشان‌دهنده‌ی هزینه‌ی پرداخت شده برای رسیدن به یک حالت میانی است. در نهایت تابع f که نشان‌دهنده‌ی هزینه‌ی نهایی هر node است را به صورت زیر تعریف می‌کنیم.

---

۲ کتاب Artificial Intelligence A Modern Approach, Third Edition

۳ مباحث تدریس شده در کلاس هوش مصنوعی دکتر قطعی

$$f = g + h$$

با توجه به اینکه در این مسئله هدف رسیدن به مقصد با کم‌ترین تعداد جابه‌جایی است، تابع g را برابر تعداد جابه‌جایی‌هایی ماشین‌ها در نظر می‌گیریم. باید توجه داشته باشیم که در حالت «توقف»، ماشین جابه‌جایی ندارد.

مجموع فاصلهٔ منهتن[۴] ماشین‌ها

یکی از توابع هیوریستیکی که در ابتدا به ذهن می‌رسد، استفاده از فاصله‌ی منهتن ماشین‌ها تا هدفشان و محاسبه‌ی مجموع آن‌ها است. در این صورت اگر فاصله‌ی منهتن ماشین i تا هدفش $h_i(s)$ باشد، تابع h برای حالت[۵] s به صورت زیر تعریف می‌شود.

$$h(s) = \sum_{i=1}^{n} h_i(s)$$

در این صورت، هر چه ماشین‌ها به هدفشان نزدیک‌تر شوند، مقدار تابع هیوریستیک کاهش می‌یابد.

اجرا

این روش را به ازای $n = 3$ اجرا می‌کنیم. به ازای این ورودی، الگوریتم بعد از پیدا کردن ۴۰۴ نود مختلف در گراف، جواب را در عمق ۵ پیدا می‌کند. همچنین هزینه‌ی رسیدن به این جواب (تعداد جابه‌جایی ماشین‌ها) برابر ۱۰ است. مسیری که برای رسیدن به جواب طی شده است در پایین آمده است.

[1, 0, 0]	[0, 1, 0]	[0, 0, 1]
[2, 0, 0]	[0, 2, 0]	[0, 2, 0]
[3, 0, 0]	[0, 3, 0]	[0, 3, 0]
[0, 3, 0]	[0, 0, 3]	[0, 0, 3]
[0, 2, 1]	[0, 2, 0]	[0, 0, 2]
[0, 0, 0]	[0, 0, 1]	[0, 0, 1]

این تابع دارای خاصیت Admissibility می‌باشد چراکه تعداد حداقل‌های جابه‌حایی‌های لازم برای هر ماشین برای رسیدن به هدف را محاسبه می‌کند و هزینه‌ی نهایی قطعاً از این مقدار کم‌تر نیست. پس با استفاده از این تابع در روش A^* قطعاً جوابی که یافت می‌شود بهینه است.

این الگوریتم برای برای ورودی‌های ۲، ۴ و ۵ نیز اجرا می‌کنیم.

- $n = 2$: با یافتن ۱۲ نود، جواب در عمق ۳ و هزینه‌ی ۴ پیدا شد.

۴ Manhattan Distance

۵ State

- $n = 3$: با یافتن ۴۰۴ نود، جواب در عمق ۵ و هزینه‌ی ۱۰ پیدا شد.
- $n = 4$: برای خروجی ۴ به بالا با گذشت ۱ دقیقه، جوابی پیدا نشد.

پیاده‌سازی در پایتون[۶]

در این بخش پیاده‌سازی روش بالا در پایتون را توضیح می‌دهیم. از آنجاییکه آپلود فایل کد امکان‌پذیر نبود، کد را گیت‌هاب قرار دادم تا بررسی بفرمایید.

https://github.com/nimahsn/AI-Course-Projects/tree/main/report01-Search

برنامه شامل سه فایل اصلی زیر است.

- Solver: این فایل شامل کلاس Solver است که الگوریتم A^* در آن اجرا می‌شود.
- Node: این فایل شامل کلاس node است که همان گره‌های درخت جست‌وجو هستند.
- State: این فایل شامل دو کلاس Car و State است که مسئله را مدل‌سازی می‌کند.

Car

این کلاس کوچکترین عنصر مدل مسأله است. هر شیء از این کلاس، نشنان دهنده‌ی یک ماشین است که دارای مختصات ابتدایی، مختصات هدف و مختصات فعلی ماشین است. همچنین یک تابع برای آن تعریف شده است که فاصله منهتن ماشین از هدفش را پیدا می‌کند.

```python
class Car:
    def __init__(self, loc: List[int], target: tuple) -> None:
        self.loc = loc
        self.target = target
        self.init_loc = loc
        self.waiting_for: int = False
        self.dont_remove = False

    def check_target(self) -> bool:
        if self.loc == self.target:
            return True
        else:
            return False

    def manhattan_distance(self) -> int:
        return abs(self.target[0] - self.loc[0]) + abs(self.target[1] - self.loc[1])
```

۶ برای پیاده‌سازی برنامه، از کد زیر که برای مسأله‌ی n-puzzle نوشته شده است، کمک گرفته شد؛ اگرچه با توجه به تفاوت مسأله، بخش زیادی از کد تغییر کرده است.

https://github.com/andavies/n-puzzle

State

این کلاس نشان دهنده‌ی حالت فعلی مسئله است. هر شیء آن دارای لیستی از ماشین‌ها (Car) است که نشان‌دهنده وضعیت مسأله هستند.

مهم‌ترین بخش این کلاس، تابع move است. این تابع با دریافت لیستی از اعمال برای ماشین‌ها، در صورت امکان مختصات ماشین‌ها را به‌روز می‌کند و در صورتی که این جابه‌جایی‌ها، شرایط مسأله را نقض کند (مثلاً دو ماشین در یک خانه)، مقدار False را برمیگرداند. این تابع در داخل خود از یک تابع کمکی به نام move_recursive نیز استفاده می‌کند. این تابع وظیفه دارد تا جابه‌جایی‌ها را به طور بازگشتی انجام دهد. مثلاً اگر ماشینی از خانه‌ی ۴ به ۵ می‌رود و ماشین داخل خانه‌ی ۵ باید به خانه‌ی ۶ برود، با فراخوانی این تابع به ازای ماشین اول، ابتدا ماشین دوم به خانه‌ی ۶ رفته و سپس ماشین اول به خانه‌ی ۵ می‌رود. همچنین حالا خاصی مانند توقف ماشین‌ها و گذشتن ماشین دیگر از کنار آن (hop) نیز در این تابع بررسی می‌شود. در نهایت اگر بتواند ماشین‌ها را جابه‌جا کند، مقدار True و در غیر این‌صورت False را برای تابع move برمیگرداند. تابع move هم درصورتی که همه‌ی ماشین‌ها با موفقیت جابه‌جا شوند، تعداد ماشین‌هایی را که دو خانه (به‌جای یکی) جابه‌جا شده‌اند را برمیگرداند. این مقدار برای محاسبه هزینه لازم است.

این کلاس همچنین دارای ۳ تابع هیوریستیک max_manhattan_distance ، sum_manhattan_distance و min_manhattan_distance است. تابع sum در این گزارش توضیح داده شد. دو تابع دیگر عملکرد ضعیف‌تری از sum داشتند.

```python
class State:
    def __init__(self, cars: List["Car"], n: int) -> None:
        self.cars: List["Car"] = cars
        self.n = n
```

Node

این کلاس برای نشان دادن گره‌های درخت ایجاد شده است. هر node درواقع یک state را نشان می‌دهد و در کنار آن شامل برخی دیگر از ویژگی‌های گره مانند عمق، گره‌ی پدر و هزینه‌ی رسیدن به آن گره (g) است.

```python
class Node:
    def __init__(self, state: "State", n: int, parent: "Node", depth: int, g_n: int) -> None:
        self.state = state
        self.n = n
        self.parent = parent
        self.depth = depth
        self.heuristic_cost = state.sum_manhattan_distance()
        self.g_n = g_n
        self.total_cost = g_n + self.heuristic_cost
```

Solver

این کلاس الگوریتم A^* را پیاده می‌کند. مهم‌ترین عضو این کلاس، یک صف اولویت یا Heap است که node های درخت در آن قرار می‌گیرند. این heap بر اساس مجموع هزینه‌ی هیوریستیک و هزینه‌ی پرداخت شده (total cost) هر node کار می‌کند. همچنین یک لیست explored نیز وجود دارد که state هایی که پیش‌تر مشاهده شده‌اند را نگه می‌دارد تا از ایجاد دور جلوگیری شود.

تابع ast_search قسمت اصلی این کلاس است که الگوریتم را اجرا می‌کند. اگر به جواب برسد، گره‌ی جواب را برمیگرداند و در غیر این‌صورت پیام dead end را چاپ می‌کند.

تابع explore_nodes هم با دریافت یک گره، تمام حرکاتی را که می‌توان برای ماشین‌ها در نظر گرفت را ایجاد می‌کند و با استفاده از تابع move که پیش‌تر توضیح داده شد، state های جدید را ایجاد می‌کند. در صورتی که تابع move بتواند حرکت را انجام دهد و state جدید قبلاً مشاهده نشده باشد، این state به صف اولویت اضافه می‌شود.

```
class ParkingSolver:
    def __init__(self, n: int) -> None:
        self.n = n
        self.frontier_heap: List[Tuple[int, int, "Node"]] = []
        heapify(self.frontier_heap)
        self.explored_set = []
        self.__entry_i = 0
```

```
def ast_search(self, initial_state: "State"):
    init_node = Node(initial_state, self.n, None, 0, 0)
    heappush(self.frontier_heap, (init_node.total_cost, self.__entry_i, init_node))
    self.explored_set.append(init_node.state)
    self.__entry_i +=1
    while self.frontier_heap:
        popped_node: "Node" = heappop(self.frontier_heap)[2]
        if popped_node.state.check_goal_state():
            print("\n\n")
            print(self.__entry_i)
            return popped_node
        self.explore_nodes(popped_node)

    else:
        print("Dead End!")
```

در نهایت در فایل solver.py، یک شیء از کلاس solver ایجاد شده و با ساختن state اولیه و دادن آن به تابع ast_search، کار الگوریتم شروع می‌شود.

```python
def print_route(node: "Node"):
    if node == None:
        return
    print_route(node.parent)
    print(node.state)

n = 4
solver = ParkingSolver(n)
cars = [Car(loc=[i, 0], target=[n-i-1, n-1]) for i in range(n)]
state: "State" = State(cars=cars, n=n)
final_state = solver.ast_search(state)
print(final_state.depth)
print_route(final_state)
print(final_state.g_n)
```

در صورتی که الگوریتم جواب را پیدا کند، تابع print_route مسیر رسیدن به جواب را چاپ می‌کند.

جمع‌بندی

در این تمرین، یک مسأله را با روش A^* و با استفاده از سه تابع هیوریستیک متفاوت حل کردیم که یک مورد آن در گزارش ذکر شد. همچنین دیدیم که به با توجه به ابعاد مسأله پیاده‌سازی آن با روش‌های Uninformed به صرفه نیست و حتی با روش هیوریستیک هم به ازای ورودی بزگتر از ۴ نتوانستیم در زمان معقولی خروجی را دریافت کنیم. دو تابع هیوریستیک دیگری که در کد قرار دارد، با توجه به اینکه تخمین کم‌تری برای هزینه می‌زدند، در این گزارش بررسی نشدند.