دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)
دانشکده ریاضی و علوم کامپیوتر

تحقیق درس هوش مصنوعی و کارگاه

حل هیوریستیک مساله n-وزیر

نگارش
ایلیا خلفی

استاد راهنما
دکتر مهدی قطعی

استادمشاور
روح الله احمدیان

آبان ۱۴۰۱

چکیده

مساله n-وزیر از مسائل کلاسیک رشته علوم کامپیوتر است که راهکارهای بسیاری برای آن پیشنهاد شده است. همچنین حل هیوریستیک این مساله یکی از این راهکارها است که سعی می‌کند با ارائه‌ی تابع هزینه‌ای، حداقل میزان حرکات لازم برای رسیدن به وضعیت نهایی از طریق وضعیت موجود را تقریب بزند. در این تحقیق ۳ تابع هیوریستیک برای حل این مسئله پیشنهاد خواهیم داد و از طریق الگوریتم‌های جستجوی حریصانه و A^* سعی در یافتن مینیمم این تابع خواهیم کرد.

واژه‌های کلیدی:

هیوریستیک، مساله n وزیر، جستجوی حریصانه، جستجوی A^*

فهرست مطالب

صفحه

فصل اول

مقدمه

مسـاله موسـوم به ۸-وزیر اولین بار توسـط شـطرنج باز آلمانی، مکس بزل[۱]، در سـال ۱۸۴۸ مطرح شـد. صورت این مساله بدین صورت است:

تعداد ۸ وزیر را بر روی صفحه شطرنجی ۸x۸ به گونه ای قرار دهید که با حرکات مجاز وزیر در بازی شطرنج، هیچ دو وزیری یکدیگر را تهدید نکنند.

نمونه جامع تر آن، مساله n-وزیر است که در آن هدف قرار دادن n وزیر در صفحه شطرنجی با سایز nxn است. همچنین مساله ای موسوم به مساله توروس[۲] وجود دارد که در آن مساله n-وزیر را اینگونه مطرح می‌کند که حرکات مجاز وزیرها در خارج صفحه نیز در نظر گرفته می‌شود، بدین صورت که تصور می‌کنیم صفحه شطرنج بر روی یک کره باشد و از هر طرف صفحه که خارج شویم از طرف دیگر آن وارد می‌شویم.

در این تحقیق، ابتدا کلاسی برای الگوریتم های جستجوی حریصانه و A^* تعریف می‌کنیم و سپس از این الگوریتم‌ها برای جستجوی مینیمم ۳ تابع هیوریستیکی که بعدتر ارائه می‌کنیم، استفاده خواهیم کرد. سه تابع هیوریستیکی که معرفی می‌کنیم به طور خلاصه عبارتند از:

۱) تعداد جفت وزیرهایی که یکدیگر را تهدید می‌کنند.

۲) تعداد وزیرهای که حداقل توسط یک وزیر دیگر تهدید می‌شوند

۳) تعداد جفت وزیرهایی که روی یک قطر قرار دارند (با چینش اولیه جایگشت بی تکرار)

همچنین لازم به ذکر است که در ۲ تابع اول، چینش وزیرها را به صورت یک جایگشت با تکرار نشان می‌دهیم و **هر عدد j در خانه i از جایگشت، به معنی حضور وزیری در سطر i و ستون j** است.

---

[۱] Max Bezzel

[۲] Torus Problem

فصل دوم

تعریف کلاس‌های جستجو

برای آنکه بتوانیم توابع هیوریستیک مختلف را مقایسه کنیم، نیازمند آن هستیم که توابع جستجویی تعریف کنیم که با گرفتن توابع هیوریستیک مختلف کار خود را انجام دهند، یعنی به طور خلاصه باید الگوریتم‌های جستجو و پیاده سازی آنها برای هیوریستیک های مختلف یکسان باشد تا مفهوم مقایسه معنا داشته باشد. به همین جهت و برای شروع کار، ۴ کلاس در فایل NQueens.py تعریف شده است:

۱. کلاس SearchMethod

این یک کلاس انتزاعی[۳] برای پیاده سازی الگوریتم های جستجو است که توابع __init__ و status را پیاده سازی می‌کند، تابع اول تابع سازنده‌ی نماینده های کلاس است و تابع دوم که به صورت property تعریف شده است، با توجه به مقدار تابع هزینه و تعداد حالات بعدی ممکن، گزارش می‌دهد که آیا الگوریتم جستجو جواب را یافته یا در حال جستجو است و یا جستجو به پایان رسیده و جوابی یافت نشده است.

همچنین در این کلاس یک تابع انتزاعی step تعریف شده است که کلاس های جستجویی که از این کلاس ارث می‌برند باید حتما آن را پیاده سازی کنند و پیاده سازی تابع باید به گونه‌ای باشد که با فراخواندن آن یک مرحله به جلو حرکت کند و در صورتی که جستجو پایان یافت مقدار True و در بقیه موارد مقدار False را خروجی دهد.

۲. کلاس های GreedySearch و AStarSearch

این ۲ کلاس پیاده سازی های الگوریتم های جستجوی حریصانه و A* هستند. با ارث بندی از کلاس SearchMethod تابع سازنده آنها یک شی State در اختیار آنها می‌گذارد و که توابع h

---

[۳] Abstract

و g در آن پیاده سازی شده اند. وظیفه این ۲ کلاس استفاده از توابع شی State و مینیمم کردن توابع داخلی آن است. همچنین این کلاس یک تابع خصوصی به نام push_state__ دارند که قبل از کاندید کردن وضعیت جدید، ابتدا وقوع آن را در مجموعه passed_states چک می‌کند که اینکار متناظر با بررسی این است که آیا این وضعیت قبلا پیمایش شده است یا نه. این عمل جهت جلوگیری از گیر کردن در حلقه در زمان جستجو الزامی است.

۳. کلاس State

این کلاس اصلی ای است که برای جستجو با هیوریستیک های مختلف باید پیاده سازی شود. این کلاس یک کلاس انتزاعی است و نمی‌توان به صورت مستقیم از آن نماینده ای ساخت، بلکه باید از آن ارث بری کنیم و در کلاس فرزند ساخته شده توابع انتزاعی آن را پیاده سازی کنیم.

لازم به ذکر است که طبق تعریفی که در مقدمه ارائه کردیم، هر چینش مختلف وزیرها را به صورت یک جایگشت با تکرار در نظر می‌گیریم، به همین دلیل هم یک متغیر curr_indices تشکیل شده است که لیستی شامل جایگشت با تکرار متناظر با وزیرها در وضعیت فعلی است.

در این کلاس ۴ تابع انتزاعی تعریف شده اند که عبارتند از:

- random : این تابع ایستا است و متعلق به کلاس می‌باشد و با فراخواندن آن یک شی از کلاس State با چینش تصادفی از وزیرها را برگرداند.
- h : این همان تابع هیوریستیک ما است که باید مقدار تابع هیوریستیک را به ازای چینش فعلی وزیرها خروجی دهد.
- g : این همان تابع g است که در الگوریتم جستجوی *A از آن بهره می‌بردیم و حداقل تعداد حرکات لازم برای رسیدن از وضعیت اولیه به وضعیت فعلی را خروجی می‌دهد.
- next_state : این تابع تمام وضعیت های ممکن به ازای حرکت های مجاز در وضعیت فعلی را بر می‌گردلند. در حقیقت این همان تابع successor function است که در کتاب هوش مصنوعی راسل از آن نام برده شده است.

در نهایت برای پیاده سازی و بررسی بهینگی عملکرد هر تابع هیوریستیک کافیست که که یک کلاس فرزند بسازیم که از کلاس State ارث بری کند و در آن توابع بالا رو پیاده سازی کنیم.

فصل سوم

تابع تعداد برخورد میان وزیرها (H1)

از آنجا که در مســـاله n-وزیر به دنبال آن هســـتیم که هیچ وزیری، وزیر دیگری را تهدید نکند، منطقی است که تابع هیوریستیک را تعداد برخورد میان وزیرها در نظر بگیریم، زیرا در نهایت با صفر شدن تعداد برخوردها، هیچ وزیری وجود نخواهد داشت که وزیر دیگری را تهدید کند.

۳-۱- تعریف تابع هیوریستیک

تابع هیوریســـتیک این بخش تعداد هر جفت وزیری اســـت که یکدیگر را تهدید کنند. با توجه به تعریف اولمان که چینش وزیرها را متناظر با جایگشـت های با تکرار از اعدادی ۱ تا n در نظر گرفتیم، هر ۲ وزیر یکدیگر را تهدید خواهند کرد در صورتی که یکی از شرایط زیر برقرار باشد:

- عنصر i از جایگشت با عنصر j از جایگشت برابر باشد
- قدر مطلق اختلاف عناصر i و j برابر با قدر مطلق اختلاف اندیس آنها در جایگشت باشد

در حقیقت شرط اول متناظر با هم ستون بودن ۲ وزیر و شرط دوم متناظر با هم قطر بودن ۲ وزیر است. از آنجا که چینش وزیرهایمان در سـطرها با جایگشـت نشـان می‌دهیم، پس هیچ دو وزیری در هیچ زمان در یک سطر نخواهند بود و شرطی برای هم سطر بودن وزیرها نیاز نیست.

همچنین واضـح اسـت که دو شـرط بالا با یکدیگر اشـتراک ندارند، یعنی هیچ جفت وزیری پیدا نمی‌شـود که هر دو شـرط را برقرار سـازد، پس برای محاسـبه این تابع هیوریسـتیک کافیسـت که تعداد وقوع شـرط های بالا بر روی جایگشت چینش وزیرهایمان را به صورت جدا محاسبه کرده و جمع کنیم.

۳-۲- چینش اولیه و حرکات ممکن در جستجو

همانطور که گفتیم چینش وزیرها در سطرهایشان متناظر با یک جایگشت با تکرار از اعداد ۱ تا n است و در نتیجه هر جایگشت با تکرار از اعداد ۱ تا n می‌تواند یک حالت اولیه برای جستجو باشد.

همچنین از آنجا که وزیرها را در سطرهای مجزا چیده ایم، پس منطقی آن است که تنها وزیرها را در سطرها حرکت دهیم و نه ستون ها، زیرا حرکت در ستون ها تنها باعث تهدید شدن وزیرها توسط وزیر آن سطر و افزایش تابع هیوریستیک می‌شود، گرچه زیاد شدن تابع هیوریستیک به معنی از کار افتادن توابع جستجو نیست، اما سرعت آنها را کاهش می‌دهد، در حالیکه بدون حرکت وزیرها در سطرها هم می‌توانیم جوابها را بسازیم.

همچنین در نظر داشته باشید که به ازای هر جوابی از مساله n-وزیر، می‌توانیم از هر چینش اولیه تنها با حرکت دادن وزیرها در سطرها به آن جواب برسیم، زیرا در هر جواب ممکن قطعا و دقیقا یک وزیر در هر سطر قرار دارد.

۳-۳- پیاده سازی

همانطور که گفتیم برای تعریف هر تابع هیوریستیک و انجام جستجو برای آن کافیست که ۴ تابع انتزاعی کلاس State را پیاده سازی کنیم. در فایل H1 کلاس StateCollisions برای همین منظور نوشته شده است و عملکرد توابع آن عبارتند از:

- همانطور که قبل تر اشاره کردیم، هر جایگشت باتکرار n تایی از اعداد ۱ تا n می‌تواند به عنوان چینش اولیه وزیرها در نظر گرفته شود، پس کافیست که با کمک تابع randint از کتابخانه استاندارد random همچنین جایگشتی را بسازیم.

```python
def random(n):
    # Making a random state with board size n
    from random import randint
    begin_state = [randint(0, n-1) for i in range(n)]
    return StateCollisions(n, begin_state)
```

- برای پیاده سازی تابع h کافیست که ۲ شرطی که در قسمت ۳-۱ به آنها اشاره کردیم را بر روی هر جفت از وزیرها بررسی کنیم و تعداد جفت هایی که شرط را برآورده می‌کنند بشماریم. تابع خصوصی __check_collision صادق بودن شرط را بر روی یک جفت ورودی بررسی می‌کند.

```python
def __check_collisions(self, a, b):
    if a == b:
        return False
    if (a[1]-b[1] == 0) or (abs(a[0]-b[0]) == abs(a[1]-b[1])):
        return True
    return False

def h(self):
    # Heuristic function
    h_cnt = 0
    for i in range(n):
        for j in range(i+1, n):
            h_cnt += int(self.__check_collisions((i, self[i]), (j, self[j])))
    return h_cnt
```

- در تابع g باید تعداد حرکات لازم وزیرها در سطرهایشان خودشان برای رسیدن به وضعیت فعلی از وضعیت اولیه را بشماریم. در حقیقت تعداد حرکات لازم یک وزیر برای رسیدن به خانه موردنظر در جایگشت نهایی برابر با قدر مطلق اختلاف مقدار فعلی و مقدار اولیه آن است، پس کافیست به ازای هر عنصر وضعیت فعلی این مقدار را حساب کرده و مجموع مقادیر را حساب کنیم. لازم به ذکر است که متغیر __begin_state جایگشت اولیه است که در تابع سازنده اشیا

کلاس State تعریف شده است و تابع begin_state که به صورت property تعریف شده است این مقدار را از کلاس پدر دریافت می‌کند.

```python
def g(self):
    # g Function for A* Algorithm
    g_cnt = 0
    for i in range(n):
        g_cnt += abs(self[i] - self.__begin_state[i])
    return g_cnt
```

- برای پیاده سازی تابع مابعد یا successor function که در اینجا آن را با نام next_states پیاده سازی می‌کنیم، باید به ازای هر عنصر جایگشت فعلی، اگر آن عنصر امکان بزرگتر شدن داشت، یک مقدار به آن اضافه کرده و اینگونه یک فرزند برای وضعیت فعلی بسازیم. این کار در حقیقت متناظر با یک خانه حرکت دادن وزیر متناظر با آن عنصر به سمت راست است. به طور متناظر، در صورت بزرگتر بودن از صفر، باید عمل کاهش را نیز بر روی آن عنصر اعمال کنیم.

```python
def next_states(self):
    # Return next states from all possible actions
    states = []
    for i in range(self.n):
        if self[i] > 0:
            state = self.deepcopy()
            state[i] = state[i] - 1
            states.append(state)
        if self[i] < self.n-1:
            state = self.deepcopy()
            state[i] = state[i] + 1
            states.append(state)
    return states
```

۳-۴- پیچیدگی محاسباتی توابع g و h

در محاسبه مقدار g، تنها قدر مطلق اختلاف آن عنصر با مقدار اولیه اش محاسبه می‌شود که از پیچیدگی محاســباتی O(1) اســت و از آنجا که n عنصــر در این جایگشــت داریم، پس در کل O(n) عمل انجام می‌شود.

همچنین در محاسبه مقدار h، تابع check_collisions__ به تعداد جفت های ممکن فراخوانده می‌شود. از آنجا که پیچیدگی محاســباتی این تابع O(1) اســت و به تعداد $\frac{n.(n-1)}{2}$ جفت وزیر وجود دارد، پس پیچیدگی محاسباتی این تابع $O(n^2)$ است.

۳-۵- بهینه سازی ممکن توابع g و h

در محاســـبه تابع g، در صـــورتی که مقدار g وضـــعیت پدر و اندیس وزیری که با تغییر موقعیت آن به وضعیت فعلی رسیدیم را بدانیم، می‌توانیم تنها در **O(1)** این مقدار را محاسبه کنیم.

برای بهینه سازی محاسبات تابع h نیز کافیست در محاسبه این مقدار در وضعیت اولیه و تشکیل گراف برخورد میان وزیرها، به طوری که هر وزیر یک راس و هر برخورد یک یال باشد، می‌توانیم این مقدار را به **O(n)** کاهش دهیم، زیرا در زمان حرکت دادن هر وزیر، تنها برخوردهای به وجود آمده با آن وزیر تغییر می‌کنند و کافیست برخوردهای ممکن آن وزیر را مجددا محاسبه کنیم

فصل چهارم

تابع تعداد وزیرهای تهدید شده (H2)

این تابع مستقیما هدف مساله را به عنوان تابع هیوریستیک قرار می‌دهد. زمانی که تعداد وزیرهای تهدید شده به صفر برسد دقیقا به همان خواسته مساله می‌رسیم.

۴-۱- تعریف تابع هیوریستیک

منظور از تعداد وزیرهای تهدید شده، تعداد وزیرهای است که توسط حداقل یک وزیر دیگر مورد تهدید قرار گرفته اند و تفاوت این تابع با تابع قسمت قبل یعنی H1 در این است که اگر وزیری توسط چند وزیر دیگر تهدید شود، تنها یک بار شمرده می‌شود در حالیکه در قسمت قبل به ازای هر تهدید یک بار شمرده می‌شد.

۴-۲- چینش اولیه و حرکات ممکن جستجو

چینش اولیه و حرکات ممکن جستجو کاملا مانند قسمت قبل یعنی تابع H1 است.

۴-۳- پیاده سازی

پیاده سازی کلاس State برای این تابع هیوریستیک در فایل H2.py و با نام کلاس StateThreatenQueens انجام شده است. به دلیل آنکه چینش اولیه وزیرها و حرکات ممکن جستجو مشابه تابع H1 هستند، در نتیجه توابع انتزاعی random و next_states و g نیز مانند فصل قبل پیاده‌سازی شده اند.

اما در پیاده سازی تابع h، مجددا از تابع check_collision__ استفاده می‌کنیم و بدین صورت عمل می‌کنیم که به ازای هر وزیر، در میان وزیرهای دیگر دنبال وزیری میگردیم که با وزیر مدنظر ما برخورد داشته باشد و در صورتی که همچنین وزیری داشتیم یکی به شمارنده مان اضافه می‌کنیم.

```python
def h(self):
    # Heuristic function
    h_cnt = 0
    for i in range(n):
        for j in range(n):
            if i != j and self.__check_collisions((i, self[i]), (j, self[j])):
                h_cnt += 1
                break
    return h_cnt
```

۴-۴- پیچیدگی محاسباتی توابع g و h

از آنجا که پیاده سازی g در تابع H2 مشابه تابع H1 است، در نتیجه پیچیدگی محاسباتی این تابع همانطور که در فصل قبل توضیح دادیم، $O(n)$ است.

همچنین برای محاسبه مقدار h باید در بدترین حالت تمامی جفت وزیرها را بررسی کنیم (زمانی که هیچ وزیری مورد تحت واقع نشود و به جواب رسیده باشیم) پس در بدترین حالت پیچیدگی محاسباتی این تابع $O(n^2)$ است. اما در عمل انتظار می‌رود محاسبه تعداد وزیرهای مورد تهدید سریعتر از تعداد جفت برخوردها باشد، زیرا برای محاسبه تعداد وزیرهای مورد تهدید در بهترین حالت از زمان $O(n)$ محاسبه نیاز است و احتمال سریعتر بودن آن وجود دارد.

۴-۵- بهینه سازی ممکن توابع g و h

تمامی راهکار هایی که برای بهینه سازی این ۲ تابع در فصل قبل ارائه دادیم در اینجا نیز می‌توان به کار برد. در نتیجه بهینه ترین حالت ممکن g از پیچیدگی محاسباتی $O(1)$ و بهینه ترین حالت ممکن h از پیچیدگی محاسباتی $O(n)$ خواهد بود.

فصل پنجم

تابع تعداد جفت وزیرهای هم قطر (H3)

از ابتدای کار تا به اینجا دیدیم که با قرار دادن وزیرها در سطرهای مجزا، توانستیم مشکل تهدید سطری وزیرها را حل کنیم، اما همچنان در ۲ تابع هیوریستیک قبل امکان تهدید ستونی بین وزیرها وجود داشت. حال این ایده به ذهن خطور می‌کند که وزیرها را در سطرها و ستون‌های مجزا قرار دهیم و سعی کنیم تنها تهدیدهای قطری را کاهش دهیم. اما با این چینش دیگر نمی‌توانیم مانند ۲ فصل قبل وزیرها را در سطر خودشان حرکت دهیم، پس چگونه می‌توانیم حرکات ممکن برای جستجو را تعریف کنیم؟

۵-۱- تعریف تابع هیوریستیک

همانطور که گفتیم تابع هیوریستیک H3 را به صورت تعداد جفت وزیرهایی که در یک قطر قرار دارند تعریف می‌کنیم. در ادامه چینش اولیه و حرکات ممکن در جستجو را طوری تعریف می‌کنیم که با انجام این حرکات بر روی چینش اولیه هیچ تهدید سطری و ستونی اتفاق نیافتد، در نتیجه با صفر شدن تعداد برخوردهای قطری و نبود برخوردهای سطری و ستونی به جواب خواهیم رسید. برای محاسبه این مقدار، کافیست شرط دوم که در ۳-۱ معرفی کردیم را بررسی کنیم و شرط اول که مربوط به برخوردهای سطری بود را کنار می‌گذاریم.

۵-۲- چینش اولیه و حرکات ممکن در جستجو

برای از بین بردن امکان برخوردهای سطری و ستونی، این بار چینش اولیه وزیرها را متناظر با یک جایگشت بدون تکرار در نظر می‌گیریم، در نتیجه از آنجا که مقدار i در خانه j جایگشت نشان دهنده

وجود وزیری در سطر i و ستون j است، پس به دلیل متفاوت بودن هر جفت i و j در جایگشت هیچ دو وزیری نیز در یک سطر و ستون نخواهند بود.

سپس حرکات ممکن در جستجو را جابجا کردن دو عنصر متوالی جایگشت تعریف می‌کنیم که متناظر با جابجا کردن دو سطر از جدول به همراه وزیرهای درون آن سطرها است. با این حرکت ممکن نیست تهدید سطری جدیدی به وجود آید، زیرا سطرها جا به جا می‌شوند و همچنین ستون وزیرها که برابر با مقدار عنصر متناظر با آنها در جایگشت است نیز تغییر نمی‌کند.

از دو بند قبل نتیجه می‌شود که در چینش اولیه برخورد سطری و ستونی نخواهیم داشت و هیچ حرکت ممکنی در جستجو یک برخورد سطری یا ستونی جدید ایجاد نمی‌کند و تعداد برخوردهای سطری و ستونی همچنان برابر صفر باقی می‌ماند. در نتیجه هدف ما تنها صفر کردن تعداد برخوردهای قطری خواهد بود که همان تابع هیوریستیک تعریف شده است.

۵-۳- پیاده سازی

برای پیاده سازی عملکرد تابع هیوریستیک H3 کلاس StateDigonalCollisions در فایل H3.py پیاده‌سازی شده است. هر ۴ تابع انتزاعی پیاده سازی شده در این کلاس با توابع انتزاعی فصل های قبل تفاوت دارند و شرح آنها عبارت است از:

- برای پیاده سازی تابع random باید یک جایگشت بدون تکرار بسازیم، پس ابتدا یک لیست از اعداد ۱ تا n می‌سازیم و سپس با متد shuffle از کتابخانه استاندارد random، ترتیب این اعداد را به صورت تصادفی مشخص می‌کنیم.

```python
def h(self):
    # Heuristic function
    h_cnt = 0
    for i in range(n):
        for j in range(n):
            if i != j and self.__check_collisions((i, self[i]), (j, self[j])):
                h_cnt += 1
                break
    return h_cnt
```

- برای پیاده سازی تابع h از تابع __check_diag_collisions استفاده می‌کنیم که در اصل همان تابع قبلی __check_collisions است که ولی تنها شرط اول از بخش ۳-۱ را بر روی جفت وزیر ورودی بررسی می‌کند. سپس برای بررسی هر جفت وزیرها مانند پیاده سازی H1 عمل می‌کنیم.

```python
def __check_diag_collisions(self, a, b):
    if a == b:
        return False
    if abs(a[0]-b[0]) == abs(a[1]-b[1]):
        return True
    return False


def h(self):
    # Heuristic function
    h_cnt = 0
    for i in range(n):
        for j in range(i+1, n):
            h_cnt += int(self.__check_diag_collisions((i,self[i]), (j, self[j])))
    return h_cnt
```

- برای پیاده سازی تابع g باید از یک مفهوم الگوریتمی به نام ناهمگونی[۴] استفاده کنیم. اگر اعداد i و j اعدادی غیر یکسان از یک دنباله باشند که تابع index مقدار اندیس آنها در این دنباله را نشان دهد، آنگاه جفت i و j تشکیل یک ناهمگونی می‌دهند اگر:

$$index(i) < index(j) \;\; and \;\; i > j$$

توجه کنید که با جابجایی هر ۲ عنصر متوالی از یک دنباله یا جایگشت، دقیقا مقدار ناهمگونی یک واحد تغییر (افزایش یا کاهش) می‌کند. در نتیجه اگر شماره اندیس عناصر جایگشت اولیه را در یک آرایه بریزیم و این آرایه را همانطور که جایگشت اولیه به جایگشت فعلی تبدیل شده است، تبدیل کنیم، تعداد ناهمگونی ها دقیقا برابر تعداد حرکات لازم برای رسیدن به موقعیت فعلی خواهد بود. در اینجا تعداد ناهمگونی ها را با روشی بسیار غیر بهینه محاسبه کرده ایم و بین هر دو جفت از اعداد جایگشت وجود ناهمگونی را بررسی کرده ایم. در بخش های بعد راهکاری برای بهینگی این متد پیشنهاد خواهیم داد.

---

[۴] Inversion

```python
def g(self):
    # g Function for A* Algorithm
    perm_indices = [self.curr_indices.index(i) for i in self.begin_state()] # permutation array

    inv_cnt = 0
    for i in range(self.n):
        for j in range(i+1, self.n):
            if (perm_indices[i] > perm_indices[j]):
                inv_cnt += 1
    return inv_cnt
```

- در نهایت برای پیاده سازی متد next_states باید هر ۲ عنصر متوالی از جایگشت فعلی را جابه‌جا کنیم تا حالت های ممکن بعدی به دست آیند.

```python
def next_states(self):
    # Return next states from all possible actions
    states = []
    for i in range(self.n-1):
        state = self.deepcopy()
        state[i], state[i+1] = state[i+1], state[i]
        states.append(state)
    return states
```

۵-۴- پیچیدگی محاسباتی توابع g و h

در محاسبه مقدار g، تمامی جفت اعداد موجود در جایگشت را برای شمارش تعداد ناهمگونی ها بررسی کردیم، از آنجا که بررسی ناهمگونی از پیچیدگی محاسباتی $O(1)$ و تعداد جفت ها از مرتبه $O(n^2)$ است، در نتیجه محاسبه مقدار g نیز از مرتبه پیچیدگی محاسباتی $O(n^2)$ می‌باشد.

همچنین مجددا برای محاسبه h، باید هر دو جفت از وزیرها را برای برخورد قطری بررسی کنیم که در نتیجه مانند متد h در پیاده سازی تابع H1 پیچیدگی محاسباتی h در اینجا نیز از مرتبه $O(n^2)$ می‌باشد.

۵-۵- بهینه سازی ممکن توابع g و h

در محاسبه تعداد ناهمگونی ها دیدیم که هر جفت از اعداد جایگشت را بررسی کردیم، اما اینکار روشی بهینه نیست. بهینه ترین روش برای شمارش تعداد ناهمگونی ها استفاده از الگوریتم تقسیم و غلبه[۵] و به خصوص الگوریتم مرتب سازی ادغامی[۶] است، بدین صورت که در هر مرحله که این الگوریتم قصد ادغام ۲ زیر آرایه را داشت، تعداد ناهمگونی ها را بشماریم. بدین شکل پیچیدگی زمانی شمارش تعداد ناهمگونی مانند پیچیدگی زمانی الگوریتم مرتب سازی ادغامی شده و به O(n.logn) کاهش می‌یابد.

همچنین روشی که برای بهینه کردن محاسبه h در پیاده سازی H1 ذکر کردیم در اینجا نیز می‌تواند مورد استفاده قرار گیرد تا پیچیدگی محاسباتی متد h را به O(n) کاهش دهیم.

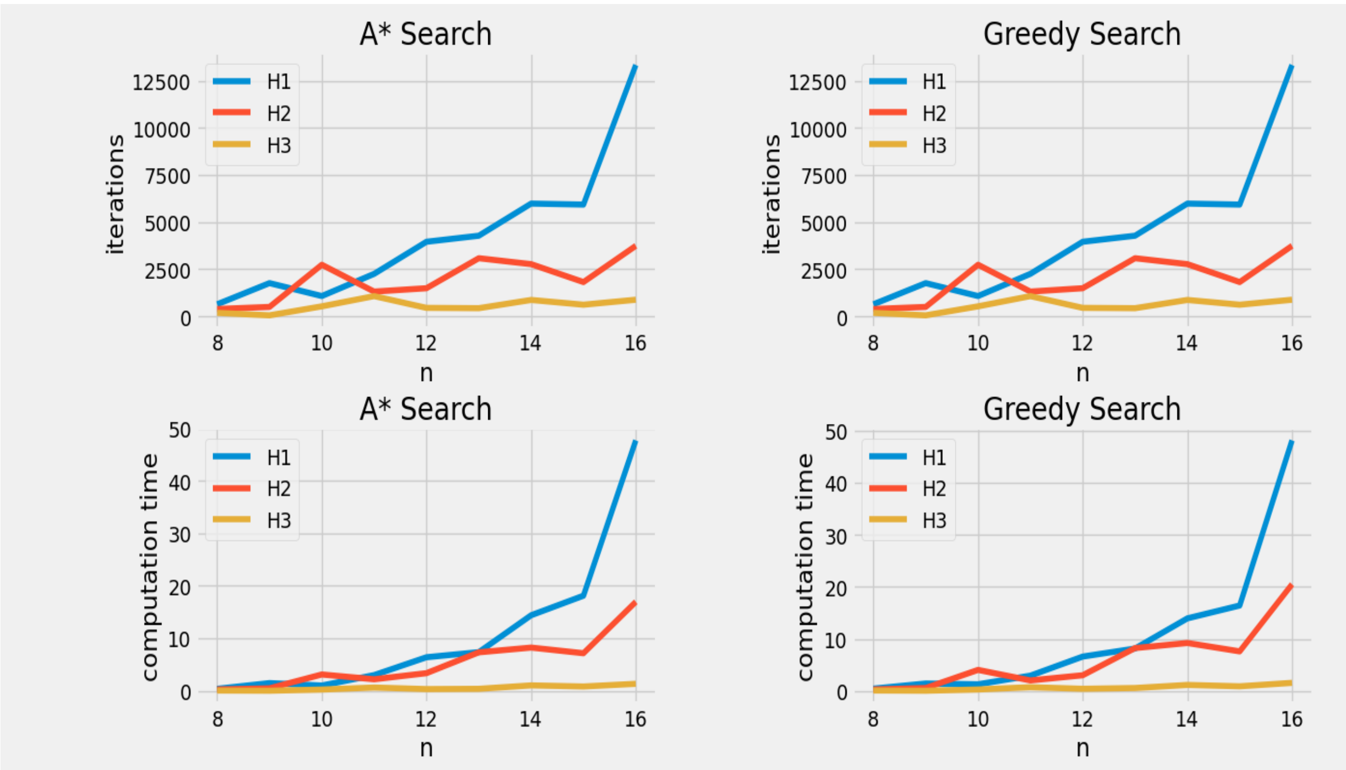
[۵] Divide and Conquer Algorithm

[۶] Merge Sort

فصل ششم

مقایسه عملکرد توابع پیشنهاد شده

در فایل Comparator.py برنامه‌ای جهت مقایسه زمان و تعداد مراحل جستجو بر روی توابع هیوریستیک پیشنهاد شده، پیاده سازی شده است. این برنامه کلاس های فرزند State که در فصل‌های قبل پیاده سازی شده بودند را فراخوانی کرده و به تعداد test_times که یک متغیر در برنامه است، چینش اولیه تصادفی تولید می‌کند. سپس اشیای از کلاس های فرزند State با این چینش های اولیه ساخته و با الگوریتم های جستجوی حریصانه و A* مینیمم توابع هیوریستیک پیشنهادی را می‌یابد. سپس زمان و تعداد مراحل مورد نیاز برای جستجو را به ازای توابع هیوریستیک مختلف و با کمک کتابخانه‌ی matplotlib نمایش می‌دهد. این برنامه به ازای n های ۸ تا ۱۶ و به تعداد ۱۰ بار توابع هیوریستیک را جستجو می‌کند که در نتیجه اجرای آن کمی زمانبر است، اما در نهایت با اجرای یک بار آن همچین نموداری را نمایش می‌دهد:



طبق نمودار نمایش داده شـده می‌توان اسـتدلال کرد که تابع هیوریسـتیک تعداد جفت وزیرهای هم‌قطر بهترین گزینه بین ۳ تابع پیشنهادی برای جستجو و یافتن جواب در مساله n-وزیر است.

همچنین نکته جالب آن است که استفاده از الگوریتم جستجوی A^* در این مساله، بهبود زیادی در زمان محاسبه و تعداد مراحل ایجاد نکرده است، به طور مثال تنها تفاوت قابل مشاهده به ازای $n = 16$ در بین ۲ نمودار پایین تصویر است که مربوط به زمان محاسبه این ۲ الگوریتم جستجو است. البته ممکن است به ازای n های بزرگتر این تفاوت بیشتر قابل مشاهده باشد.

فصل هفتم

جمع بندی و نتیجه گیری

در این تحقیق ۳ تابع هیوریستیک را پیشنهاد دادیم که ۲ تای اولی سعی در از بین بردن تهدیدهای سطری میان وزیرها و در نتیجه جستجوی برای کاهش تهدیدهای ستونی و قطری داشتند. اما در تابع سوم تهدیدهای ستونی را نیز غیرممکن کردیم با تعریف خلاقانه‌ی اعمال ممکن در جستجو، تابع هیوریستیک تعداد برخوردهای قطری را تعریف کردیم که در نهایت بسیار سریعتر از ۲ تابع دیگر بود. از این تحقیق ۲ نکته جالب را می‌توان مطرح کرد.

در بخش های قبل دیدیم که پیچیدگی محاسباتی متدهای h و g در تعریف تابع H3 بهتر از پیچیدگی محاسباتی این متدها در تعریف توابع H1 و H2 نبودند، حتی پیچیدگی محاسباتی g از مرتبه $O(n^2)$ بود، اما با این حال تابع هیوریستیک H3 عملکرد بهتری از ۲ تابع دیگر داشت. علت این عملکرد بهتر، چینش اولیه بهبود یافته‌ی این تابع است. برای درک بهتر می‌توان گفت زمانی که تابع جستجو در تابع هیوریستیک اول، سعی در کاهش برخوردهای ستونی و قطری می‌کند، تابع H3 از همان ابتدای تعریفش با هیچ برخورد ستونی ای مواجه نیست و به همین دلیل از وضعیت بهتری شروع به جستجو می‌کند.

در نتیجه نکته جالبی که از مقایسه میان توابع هیوریستیک پیشنهادی در این تحقیق می‌توان استدلال کرد، این است که **تعریف خلاقانه توابع هیوریستیک حتی اهمیتی بالاتر از پیچیدگی محاسباتی و دقت پیاده سازی دارد.**

پیوست

لینک فایل کدها

متاسفانه به دلیل بسته بودن دسترسی ارسال فایل فشرده، امکان ارسال کدها در سامانه کورسز وجود نداشت. با این وجود فایل کدهای این تحقیق در لینک زیر در دسترس هستند:

https://drive.google.com/drive/folders/17UyGVZ_wLWXWGKN09BhUC4LqbFjRQdov?usp=share_link