گزارش پروژه دوم هوش مصنوعی

امین زینالی
دانشکده ریاضی،آمار و علوم کامپیوتر
دانشگاه تهران

۱ تعریف اجزای مسئله

۱.۱ تعریف States

می‌دانیم State های این مسئله باید جایگاه تمامی اعداد در جدول پازل را مشخص کند بنابراین می‌توانیم به صورت یک آرایه دوبعدی به اندازه $3 * 3$ در نظر بگیریم ولی از آنجایی که اندازه پازل و اعداد داخل آن ثابت هستند برای سادگی State ها را به صورت یک آرایه یک بعدی با 9 عضو تعریف می‌کنیم به عبارتی جدول پازل را Flat می‌کنیم. به مثال زیر توجه کنید فرض کنید در وضعیت زیر قرار داریم:

4	1	6
5	3	2
8	7	0

به صورت Flat به شکل زیر خواهد بود:

4	1	6	5	3	2	8	7	0

لازم به ذکر است که عدد صفر نشان دهنده کاشی خالی است.

۲.۱ تعریف Operators و Transition model

در این مسئله 4 حرکت زیر را انجام دهیم:

- بالا
- پایین
- چپ
- راست

اعمال فوق برای کاشی خالی تعریف شده‌اند بنابراین هدف ما این است که با جابه‌حایی کاشی خالی بتوانیم به هدف برسیم.برای مثال فرض کنید در وضعیت زیر هستیم:

4	1	3
8	0	6
5	7	2

با انجام عمل بالا به وضعیت زیر می‌رویم:

4	0	3
8	1	6
5	7	2

برای سایر اعمال نیز وضعیت بعدی به طور مشابه به دست می‌آید.

۳.۱ تعریف **Goal test**

ابتدا وضعیت هدف را تعیین می‌کنیم.هدف ما این است که با جابه‌جایی خانه خالی بتوانیم اعداد را مرتب شده سرجایشان قرار دهیم در نتیجه وضعیت هدف به شکل زیر است:

1	2	3
4	5	6
7	8	0

با این توضیحات کافیست هر وضعیت را با وضعیت هدف مقایسه کنیم در صورت برابری True و در غیر این صورت False برگردانیم.

۴.۱ تعریف **Path cost**

در حالت کلی هزینه یک مسیر را تعداد کل Operator های انجام شده در نظر می‌گیریم بنابراین وقتی با انجام یکی از چهار عمل ذکر شده از یک وضعیت به وضعیت دیگر می‌رویم یک واحد به هزینه کل اضافه می‌شود.

۵.۱ تابع **Heuristic**

در طول درس با دو تابع Heuristic زیر برای این مسئله آشنا شدیم:

- تعداد کاشی هایی که سر جای خود نیستند
- محاسبه فاصله manhattan

در تابع اول برای هر کاشی یک حالت باینری در نظر می‌گیریم اگر کاشی سرجای خود باشد هیچ اگر نباشد یک واحد به مقدار تابع اضافه می‌شود به عبارتی فرض می‌کنیم می‌توانیم هر کاشی را با یک حرکت سر جای خود قرار دهیم در نتیجه مقدار این تابع همواره از هزینه واقعی کمتر است.

در تابع دوم نیز ما فرض می‌کنیم هر دو کاشی مجاور را می‌توانیم باهم جابه‌جا کنیم در حالی که در مسئله چنین چیزی نداریم و مقدار این تابع نیز از هزینه واقعی کمتر است .

با این توضیحات می‌توان گفت دو تابع admissible هستند ولی از آنجایی که تابع دوم یعنی Manhattan distance به تابع هزینه واقعی نزدیکتر است ما آن را انتخاب می‌کنیم.

۲ پیاده سازی

۱.۲ اطلاعات لازم در هر گره

یک کلاس به اسم State تعریف می‌کنیم تا اطلاعات یک گره را در خود ذخیره کند این اطلاعات عبارتند از:

- لیست نشان دهنده محل کاشی ها
- پدر گره
- یک لیست شامل همه بچه های گره
- مقادیر $g(n), h(n), f(n)$

پدر هر گره طی الگوریتم مشخص می‌شود و مقدار اولیه آن None است همچنین مقدار g هر گره یک واحد بیشتر از g پدرش است و h نیز مقدار تابع Heuristic است و برای f نیز رابطه زیر را داریم:

$$f(n) = g(n) + h(n)$$

۲.۲ شبه کد الگوریتم ها

در حالت کلی می‌توان گفت الگوریتم هایی که استفاده می‌کنیم از قالب زیر پیروی می‌کنند:

```
initialize queue with init_state
initialize array seen_states
while queue is not empty repeat
    pop a state from queue
    if state is equal to goal_state then
        return state
    for child in state childs
        set child attributes
        if child is not in seen_states then
            add child to queue and seen_states
```

۳.۲ اندازه گیری زمان و حافظه مصرفی

برای اندازه گیری زمان اجرای الگوریتم ها از کتابخانه time استفاده می‌کنیم به این صورت که قبل و بعد از فراخوانی الگوریتم تابع

```
time.time()
```

را صدا می‌زنیم و با به دست آوردن اختلاف زمان این دو مقدار زمان اجرای الگوریتم به دست می‌آید.حاصل به دست آمده به این شیوه در واحد نانوثانیه است که ما آن را به پیکوثانیه تبدیل می‌کنیم.

برای اندازه گیری مقدار حافظه مصرفی سراغ کتابخانه tracemalloc می‌رویم قبل از فراخوانی الگوریتم تابع tracemalloc.start() را صدا می‌زنیم تا کتابخانه شروع به کار کند پس از اتمام الگوریتم کد زیر را اجرا می‌کنیم:

```
tracemalloc.get_traced_memory()
```

خروجی کد بالا یک tuple است که دو عضو دارد و عضو دوم آن بیشترین مقدار حافظه شده در طی اجرای الگوریتم است ما این مقدار را ملاک مقایسه حافظه مصرفی الگوریتم ها قرار می‌دهیم.

۳ نتایج

ورودی زیر را به الگوریتم ها می‌دهیم عملکرد آن ها در جدول زیر آمده است:

1	3	5	7	2	6	8	0	4

algorithm	time	memory	visited nodes
A*	8.965	160417	259
BFS	287.40	1068955	1665
DFS	-	-	-
UCS	199.011	1071114	1665
IDS	195.53	731359	1157

همانطور که مشاهده می‌کنید الگوریتم A* با اختلاف از بقیه الگوریتم ها بهتر است حالا در ادامه مسیر پیدا شده توسط این الگوریتم را مشاهده می‌کنیم:

step 1	1	3	5	7	2	6	8	0	4
step 2	1	3	5	7	2	6	8	4	0
step 3	1	3	5	7	2	0	8	4	6
step 4	1	3	0	7	2	5	8	4	6
step 5	1	0	3	7	2	5	8	4	6
step 6	1	2	3	7	0	5	8	4	6
step 7	1	2	3	7	4	5	8	0	6
step 8	1	2	3	7	4	5	0	8	6
step 9	1	2	3	0	4	5	7	8	6
step 10	1	2	3	4	0	5	7	8	6
step 11	1	2	3	4	5	0	7	8	6
step 12	1	2	3	4	5	6	7	8	0