

## مقدمه

توی درس با الگوریتم‌های جستجو آشنا شدیم. یکی از کاربردهای متداول این الگوریتم‌ها استفاده از شون برای حل پازل‌ها و بازی‌های مختلفه. توی این پروژه قصد داریم با استفاده از الگوریتم‌های جستجو به حل مکعب روبیک دو در دو بپردازیم.

## راه‌اندازی محیط پروژه

اول از همه، پروژه رو از [این لینک](#) دریافت کنید. برای اجرای محیط پروژه نیاز به کتابخونه‌ی ursina رو با pip نصب کنید. بعدش با دستور زیر محیط بازی رو در حالت manual اجرا کنید:

```
python main.py --manual
```

می‌تونید با نگه داشتن کلیک راست و حرکت موس زاویه‌ی دوربین رو تغییر بدید. همچنین می‌تونید با کلیدهای ۱ تا ۶ و q تا y وجه‌های مکعب روبیک رو بچرخونید.

## ساختار پروژه

کد این پروژه از فایل‌های زیر تشکیل شده:

|             |   |
|-------------|---|
| main.py     | پیاده‌سازی روند کلی اجرای محیط و الگوریتم‌ها              |
| rubic.py    | پیاده‌سازی منطق و گرافیک روبیک در محیط ursina             |
| state.py    | تعریف حالت و تابع تبدیل حالات                             |
| location.py | تعریف موقعیت مکعب‌های کوچک (برای تابع heuristic نیاز است) |
| algo.py     | پیاده‌سازی الگوریتم‌های جستجو توسط شما                    |

توجه: تنها فایلی که نیازه تغییرش بدید algo.py هست و نیازی به تغییر سایر فایل‌ها نیست.

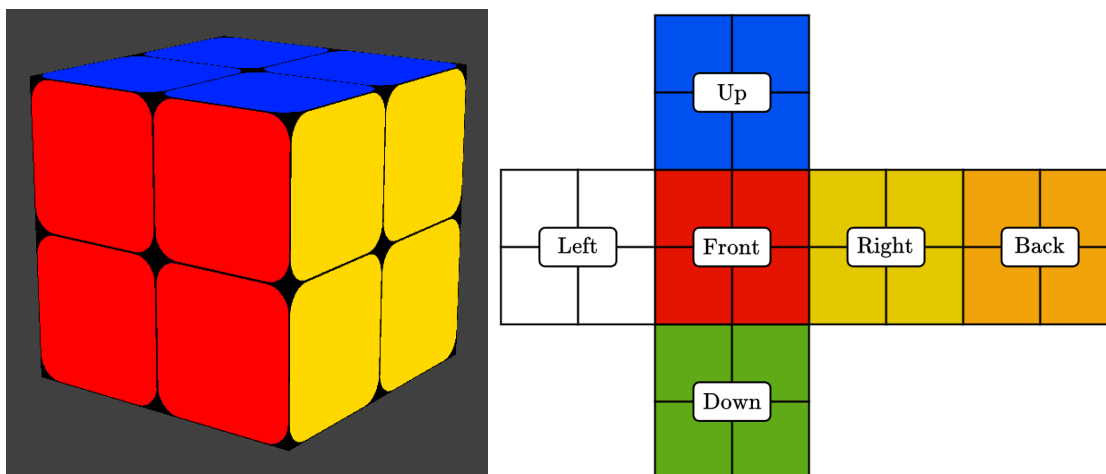
در طی این تعریف پروژه، بخش‌هایی که خودمون انجام دادیم رو توضیح میدیم و بخش‌هایی رو شما باید انجام بدید که به شکل **کد** یا **گزارش** هست.

### فرمول‌بندی مسئله

برای اینکه بتونیم مکعب روبیک رو با الگوریتم‌های جستجو حل کنیم، ابتدا نیازه که یک فرمول‌بندی مناسب از مسئله داشته باشیم.

#### ۱- حالت‌ها:

در اولین قدم نیازه که مکعب رو به صورت دو بعدی نمایش بدیم تا بتونیم راحت‌تر باهاش کار کنیم:



(شکل ۱)

حالا ۶ رنگ موجود در مکعب رو به صورت زیر شماره گذاری می کنیم:

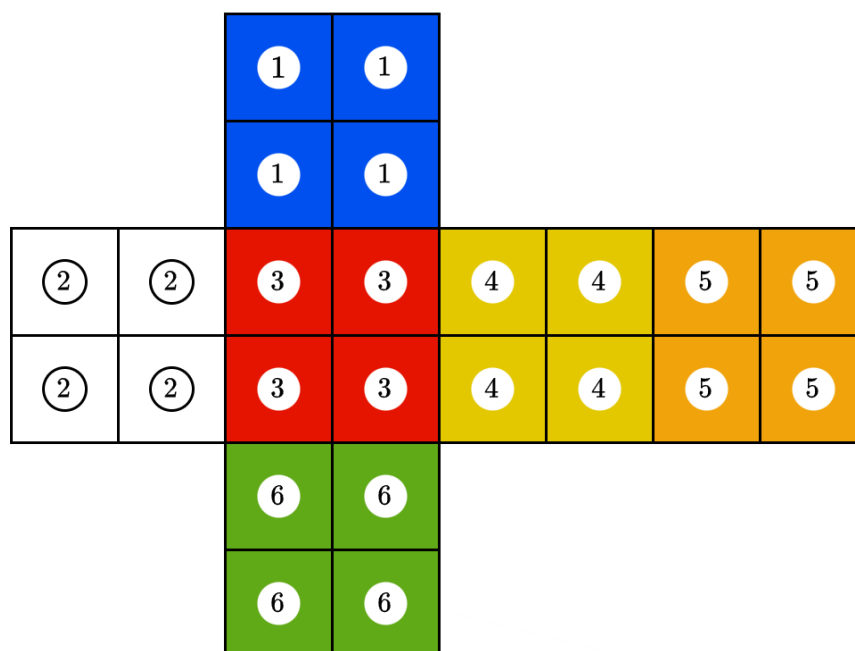
۱ = آبی، ۲ = سفید، ۳ = قرمز، ۴ = زرد، ۵ = نارنجی، ۶ = سبز

بعد حالت (state) رو به شکل یک آرایه دوبعدی مثل سمت چپ شکل پایین ذخیره می کنیم:

[

[1, 1],  
[1, 1],  
[2, 2],  
[2, 2],  
[3, 3],  
[3, 3],  
[4, 4],  
[4, 4],  
[5, 5],  
[5, 5],  
[6, 6],  
[6, 6],

]



(شکل ۲)

که به ترتیب سطرهای مربوط به سطوح بالا، چپ، روبه‌رو، راست، پشت، و پایین در اون ذخیره شدن.

## ۲- حالت اولیه و حالت هدف:

با وجود اینکه مکعب روبیک دو در دو ساده و جمع و جور به نظر میاد، در مجموع 3,674,160 حالت داره! هر یک از حالت‌های ممکن می‌تونه حالت اولیه ما در مسئله باشه.

برای سادگی، حالت هدف رو حالتی در نظر می‌گیریم که در اون در هر سطح تنها یک رنگ وجود داره، و ترتیب رنگ‌ها همون ترتیب شکل ۲ هستش. به این طریق، تنها یک حالت هدف واحد داریم.

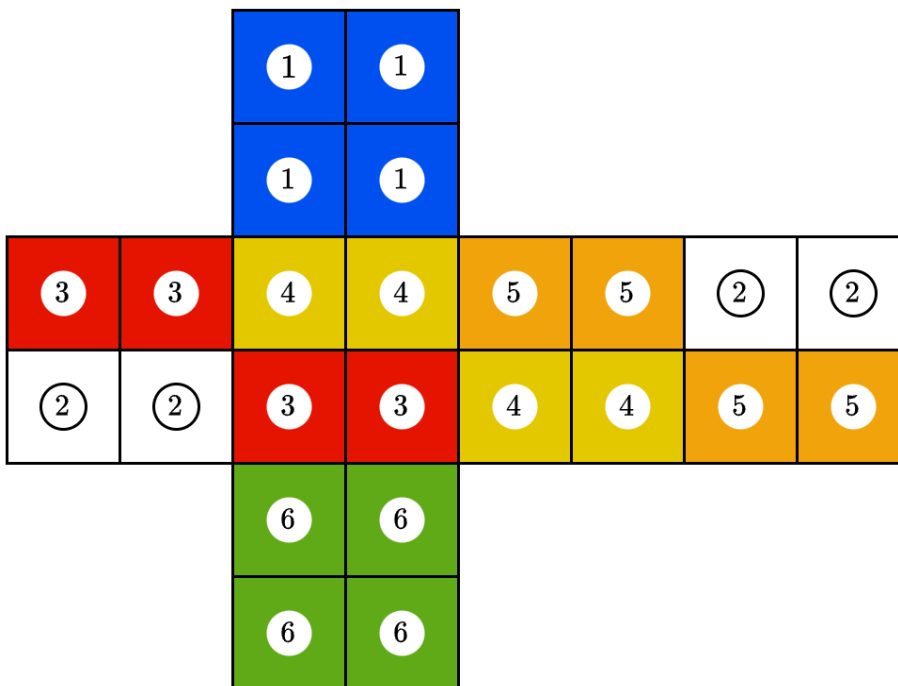
## ۳- اعمال و مدل انتقال:

هر یک از ۶ وجه مکعب رو می‌تونیم ۹۰ درجه به صورت ساعت‌گرد یا پادساعت‌گرد بچرخونیم. پس در مجموع ۱۲ عمل ممکن داریم. این ۱۲ عمل به ترتیب همون اعمالی هستن که در حالت manual با کلیدهای ۱ تا ۶ و q تا y قابل انجام بود.

هر یک از این اعمال منجر به تغییر حالت و آرایه مربوط به اون میشه. برای مثال، اگر وجه بالایی رو به صورت ساعت‌گرد بچرخونیم، به حالت زیر می‌رسیم:

[

[1, 1],  
 [1, 1],  
 [3, 3],  
 [2, 2],  
 [4, 4],  
 [3, 3],  
 [5, 5],  
 [4, 4],  
 [2, 2],  
 [5, 5],  
 [6, 6],  
 [6, 6],



(شکل ۳)

تعریف حالت‌ها و نحوه تغییر حالت به ازای هریک از ۱۲ عمل در فایل state.py تعریف شده. با اجرای مستقیم این فایل می‌تونید تعریف حالت هدف و نتیجه انجام اعمال رو تست کنید.

**گزارش ۱:** کد مربوط به `action == 4` (که همون عمل نشون داده‌شده در شکل ۳ هستش) رو از فایل state.py بخونید و توضیح بدید که قدم به قدم چجوری حالت جدید ساخته شده.

#### ۴- هزینه اعمال:

هزینه اعمال با یکدیگر برابر و مساوی ۱ هستش. بنابراین، ما به دنبال راه‌حلی هستیم که کمترین تعداد عمل در اون رخ بده.

## پیدا کردن راه حل از طریق جستجو

حالا که حالت ها و اعمال رو تعریف کردیم، می تونیم بریم سراغ اصل ماجرا. هدف ما اینه که با شروع از هر حالتی، بتونیم سلسله اعمالی رو انجام بدیم که مکعب روبیک حل بشه. البته تو این پروژه به دنبال هر مسیری هم نیستیم، بلکه به دنبال کوتاه ترین مسیر ممکن می گردیم.

شما باید در فایل `algo.py` و درون تابع `solve`، الگوریتم های جستجو رو پیاده سازی کنید. این تابع، حالت اولیه، مکان اولیه مکعب های کوچک (که فقط برای الگوریتم  $A^*$  نیاز و جلوتر توضیحش میدیم) و اسم روش رو به عنوان ورودی دریافت می کنه. در خروجی باید سلسله اعمالی که نیاز به ترتیب اجرا بشه تا مکعب حل شه رو برگردونید.

برای شروع، می تونید با دستور زیر الگوریتم `Random` رو بر روی `testcase` شماره ۱ اجرا کنید:

```
python main.py --testcase testcases/1.txt --method Random
```

این الگوریتم در واقع جستجویی انجام نمیده و صرفا یه لیست رندوم از اعمال رو برمی گردونه. بعد از اتمام جستجو می تونید با فشردن `enter`، انجام راه حل رو مشاهده کنید.

حالا شما باید در دستور بالا شماره تست کیس و اسم متدها رو تغییر بدید و روش هایی که در ادامه گفته میشه رو پیاده سازی کنید. در جهت راحت تر شدن پیاده سازی ها، می تونید به راهنمایی هایی که تو صفحه یکی مونده به آخر این فایل هست مراجعه کنید.

## الگوریتم ۱: جستجوی ID-DFS

ابتدا از الگوریتم Iterative Deepening Depth-first Search شروع می‌کنیم تا ببینیم یک الگوریتم Uninformed چقدر می‌تونه توی حل این پازل موفق باشه.

**کد:** الگوریتم ID-DFS رو به صورت گرافی پیاده‌سازی کنید. از  $\text{limit} = 1$  شروع کنید و بدون بهینه به این  $\text{limit}$  اضافه کنید تا به جواب برسید. الگوریتم شما باید به جواب بهینه دست پیدا کنه.

**گزارش ۲:** الگوریتم رو برای تست کیس‌های ۱ تا ۲ اجرا کنید و تعداد گره‌های  $\text{expand}$  و  $\text{explore}$  شده، عمق جواب و زمان جستجو (به ثانیه) رو توی یک جدول گزارش کنید.

**گزارش ۳:** در مورد God's Number و مقدار اون برای مکعب روبیک دو در دو بخونید و مختصر در گزارش‌تون بنویسید. دقت کنید که در پیاده‌سازی ما تنها چرخش‌های ۹۰ درجه به عنوان اعمال در نظر گرفته شده، در نتیجه God's Number رو برای این مجموعه اعمال گزارش کنید.

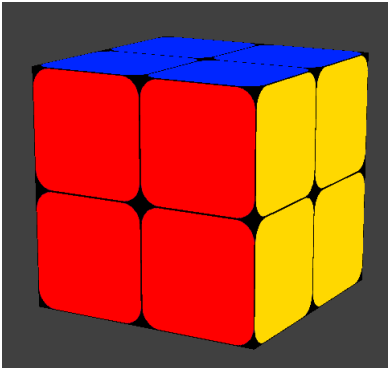
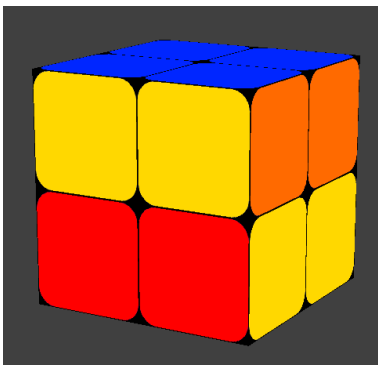
**گزارش ۴:** الگوریتمی که پیاده‌سازی کردید رو برای تست کیس‌های شماره ۳ و ۴ اجرا کنید و حداکثر ۴-۵ دقیقه صبر کنید. آیا تو این مدت روش ID-DFS موفق بود تا جواب رو پیدا کنه؟ تعداد حالات موجود در گراف جستجوی این مسئله به طور حدودی چه رابطه‌ای با عمق جستجو داره؟ با توجه به مقدار God's Number، به نظرتون آیا الگوریتم ID-DFS توانایی حل مکعب روبیک با هر حالت اولیه دلخواهی رو داره؟

## الگوریتم ۲: جستجوی A\*

همونطور که تو بخش قبل دیدیم، با افزایش عمق، زمان حل مسئله خیلی افزایش پیدا می‌کنه. یک راه حل برای چنین مشکلی اینه که روی بیاریم به الگوریتم‌های جستجوی Informed مثل A\*.

برای پیاده‌سازی A\* نیاز به یک هیوریستیک (heuristic) مناسب داریم. در ادامه تابع هیوریستیک پیشنهادی خودمون رو توضیح می‌دیم و شما باید اون رو پیاده‌سازی کنید.

برای تعریف هیوریستیک، ابتدا یک بازنمایی ساده‌تر از مسئله رو طرح می‌کنیم. یک مکعب روبیک دو در دو از ۸ مکعب کوچک‌تر تشکیل شده. این مکعب‌های کوچک رو به این شکل شماره‌گذاری کنیم: ۴ مکعب موجود در سطح روبه‌رو شماره ۱ تا ۴ می‌گیرند و ۴ سطر پشتی نیز شماره‌های ۵ تا ۸. حالا می‌خوایم که به ازای هر state، موقعیت مکانی هر یک از این ۸ تا مکعب کوچک رو ذخیره کنیم. برای این کار، موقعیت‌ها رو در یک آرایه سه‌بعدی ذخیره کرده‌ایم. برای مثال، در حالت هدف که هر مکعب کوچک در جای درست خود قرار داره، آرایه ما به صورت شکل ۴ (الف) هست.

| (الف)   |   | (ب)   |   |
|---|---|---|---|
| $\begin{bmatrix} [1, 2], \\ [3, 4], \\ [5, 6], \\ [7, 8] \end{bmatrix}$ |  | $\begin{bmatrix} [2, 6], \\ [3, 4], \\ [1, 5], \\ [7, 8] \end{bmatrix}$ |  |

(شکل ۴)



پس از انجام هر عمل، موقعیت مکانی هر کدوم از این مکعب‌ها دچار تغییر میشه. ما این تغییر موقعیت‌ها رو در فایل `location.py` نوشته‌ایم. با اجرای مستقیم این فایل می‌تونید تعریف موقعیت مکانی هدف و نتیجه انجام اعمال رو تست کنید.

**گزارش ۵:** کد مربوط به `action == 4` رو از فایل `location.py` بخونید و توضیح بدید که چجوری آرایه `location` جدید ساخته میشه. این عمل مطابق شکل ۳ و شکل ۴ (ب) است.

حالا با داشتن موقعیت، به تعریف هیوریستیک می‌رسیم. هیوریستیک ما مبتنی بر `Manhattan Distance` سه بعدی هست که بین دوتا نقطه به شکل زیر تعریف میشه:

$$\text{Manhattan Distance (A, B)} = |X_A - X_B| + |Y_A - Y_B| + |Z_A - Z_B|$$

در قدم اول، حساب می‌کنیم که موقعیت مکانی هر یک از مکعب‌های کوچکی که داریم، چه فاصله‌ای با موقعیت مکانی اش در حالت هدف داره. این فاصله می‌تونه برابر 0، 1، 2 یا 3 بشه. برای مثال، این فواصل برای مکعب‌های موجود در شکل ۴ (ب) به شکل زیر هست:

[1, 1, 0, 0, 1, 1, 0, 0]

چونکه هر کدوم از مکعب‌های موجود در وجه پایینی سر جای خود هستند، و مکعب‌های وجه‌های بالایی در فاصله ۱ از هدف خود هستند.

حالا، مقدار هیوریستیک رو به این شکل تعریف می‌کنیم: مجموع فواصل `Manhattan` مکعب‌های کوچک از موقعیت هدف خود، تقسیم بر ۴. در مثال بالا، مقدار هیوریستیک برابر با ۴ تقسیم بر ۴، یعنی ۱ هستش. این تقسیم بر ۴ به این علتیه که با هر عمل، ۴ مکعب جابه‌جا میشن و پتانسیل این رو دارن که به جای درست‌شون نزدیک‌تر بشن.

پس به طور خلاصه، ما یک بازنمایی ساده‌تر از مکعب روییک ارائه کردیم که در اون رنگ‌ها و جهت‌گیری مکعب‌های کوچک در فضا (orientation) مهم نیست و فقط موقعیت مکانی مکعب‌های کوچک ۱ الی ۸ برامون مهمه. در ادامه هم از طریق محاسبه اینکه به طور میانگین چقدر این مکعب‌ها از موقعیت هدف‌شون فاصله دارن، یک تخمین از cost بدست میاریم.

همچنین، از اونجایی که تنها موقعیت مکانی رو در نظر می‌گیریم و رنگ و orientation برامون مهم نیست، هیوریستیک ما لزوماً کوچکتر یا مساوی هزینه واقعیه. به این ترتیب، ما یک هیوریستیک قابل قبول (admissible) داریم.

**کد:** تابع هیوریستیکی که در بالا گفتیم رو پیاده‌سازی کنید. این تابع باید به ازای هر location مقدار هیوریستیک رو برگردونه.

**کد:** الگوریتم  $A^*$  رو پیاده‌سازی کنید. الگوریتم شما باید کوتاه‌ترین مسیر ممکن رو پیدا کنه. بهتره که به صورت گرافی پیش برید و وقتی با حالتی برخوردید که قبلاً دیده بودینش، چک کنید که آیا گره جدید cost کمتری داره یا نه، که اگر داشت، گره رو به frontier اضافه کنید. همچنین، حواستون باشه که شما جستجو رو همچنان روی state ها انجام میدید و از location صرفاً برای محاسبه هیوریستیک استفاده می‌کنید.

**گزارش ۶:** الگوریتم رو برای تست‌کیس‌های ۱ تا ۴ اجرا کنید و تعداد گره‌های explore و expand شده، عمق جواب و زمان جستجو رو توی یک جدول و کنار نتایج ID-DFS گزارش کنید.

**گزارش ۷:** به طور مختصر نتایجی که از ID-DFS گرفتید رو با  $A^*$  مقایسه کنید. چقدر هیوریستیکی که پیاده‌سازی کردید منجر به کاهش زمان جستجو و گره‌های explored شد؟

### الگوریتم ۳: جستجوی Bi-BFS

با وجود اینکه روش  $A^*$  کمک کرد که بتوانیم در زمان کمتری به جواب دست پیدا کنیم و تا اعماق بیشتری پیش ببریم، اما هنوز هم نمی‌تونیم تست‌کیس‌های سخت مثل ۶ و ۷ رو در زمان معقول حل کنیم. برای همین، تو این بخش قاره سراغ Bidirectional Breadth-first Search برید تا بتونید در زمان کمتری به جواب برسید.

**کد:** الگوریتم Bi-BFS رو به صورت گرافی پیاده‌سازی کنید. روش کار به این شکله که به اندازه یک عمق از مبدا با BFS رو به جلو حرکت می‌کنید و به اندازه یک عمق هم از مقصد با BFS به عقب حرکت می‌کنید. بین این دو مرحله و همچنین بعد از این دو مرحله چک می‌کنید که آیا در frontier های این دو جستجو، حالت مشترکی وجود داره یا نه. اگر وجود داشت، جواب پیدا شده، وگرنه این چرخه رو تکرار می‌کنید.

**گزارش ۸:** الگوریتم رو برای تست‌کیس‌های ۱ تا ۷ اجرا کنید و تعداد گره‌های explore و expand شده، عمق جواب و زمان جستجو رو توی یک جدول و کنار نتایج قبلی بیارید.

**گزارش ۹:** به طور مختصر نتایجی که از این ۳ تا روش گرفتید رو مقایسه کنید.

**گزارش ۱۰:** اگر تا اینجای مسیر رو به درستی پیش اومده باشید، می‌بینید که روش با Bi-BFS می‌تونید خیلی سریع سخت‌ترین چینش‌های روبیک رو حل کنید. حالا می‌تونید کامند زیر رو اجرا کنید تا چینش‌های رندوم تولید بشه و با الگوریتم شما حل بشه:

```
python main.py --method BiBFS
```

۳-۴ بار این دستور رو اجرا کنید. آیا الگوریتم‌تون می‌تونه این روبیک‌های رندوم رو حل کنه؟

## راهنمایی‌ها

- برای پیاده‌سازی frontier در ID-DFS و Bi-BFS می‌تونید از ساختمان داده OrderedDict استفاده کنید. به این شکل، می‌تونید همزمان یک پشته یا صف داشته باشید و وجود یا عدم وجود یک state رو توی زمان  $O(1)$  بررسی کنید.
- برای پیاده‌سازی Priority Queue در  $A^*$  می‌تونید از heapq استفاده کنید.
- برای نگهداری حالات explored، در صورتی که نیازی به محتوای اون حالات ندارید، می‌تونید اونا رو hash کنید تا به راحتی وجود یا عدم وجودشون رو بررسی کنید.
- اگر در جایی ترتیب مهم نیست، می‌تونید از ساختمان داده‌های Set و یا Dictionary استفاده کنید.