گزارش کار اول

برای مشاهده کد و خروجی‌اش به code_report.html **مراجعه کنید.**

اول به سه جدول مقایسه عملکرد الگوریتم‌ها نگاهی می‌کنیم.

زمان برای رسیدن به جواب			
	IDFS	BFS	A*
in1.csv	-	-	0.865
In2.csv	-	-	2.079
In3.csv	-	-	1.047
test_a.csv	0.250	6.860	0.009
test_b.csv	15.948	76.053	0.036
test_c.csv	161.348	-	0.667
Average	59.182	41.456	0.784

تعداد کل حرکات انجام شده			
	IDFS	BFS	A*
in1.csv	-	-	599
In2.csv	-	-	1420
In3.csv	-	-	566
test_a.csv	69045	19118	4
test_b.csv	4608473	240512	16
test_c.csv	49972179	-	460

تعداد حرکات مسیر جواب			
	IDFS	BFS	A*
in1.csv	-	-	6
In2.csv	-	-	6
In3.csv	-	-	7
test_a.csv	3	3	3
test_b.csv	4	4	4
test_c.csv	5	-	5

بررسی و توضیح کلی مساله:

در این مساله هر استیتمان یکی از حالات هشت وزیر در شطرنج است و هدفمان پیدا کردن حالتی است که تعداد تهدیدها صفر باشد در توابع بخش [2] IN توابعی وجود دارند که با گرفتن یک جدول دو در هشت که موقعیت کنونی وزیرهاست. تعداد تهدیدها را می‌توانند خروجی دهند.

همچنین در بخش [3] IN تابع move قرار دارد که با گرفت جدول و شماره وزیر در جدول می تواند وزیر را در یکی از هشت جهت جابه‌جا کند به این شکل که جهت صفر جابه‌جایی به بالات است و بقیه جهت‌ها به شکل ساعت گرد ادامه پیدا می‌کنند. تابع pure_move برای وقتی است که از مجاز بودن جابه‌جاییمان اطلاع داریم (مثلا وقتی می‌خواهیم جابه‌جاییمان به حالت قبل برگردد) که دیگر در این تابع چک نمی‌شود جابه‌جایی مجاز است یا خیر که باعث افزایش سرعت می‌شود.

بررسی و توضیح هر یک از الگوریتم‌ها:

IDFS

این الگوریتم در بخش [6] IN پیاده سازی شده است. پیاده سازی آن به شکل بازگشتی است. از مزایا و خوبیهای آن مصرف کم رم برای انجام الگوریتم نسبت به دو الگوریتم دیگر است. و چون حافظه‌ای برای خانه‌های طی شده ندارد. حالات تکراری زیادی طی می‌کند ولی با این حال به علت استفاده نکردن از حافظه برای ذخیره سازی استیت‌ها می‌تواند هر استیت را در زمان کمتری بررسی کند و همانطور که در جداول مشخص است با این که تعداد استیت بیشتری بررسی کرده است ولی در زمان بسیار زودتری نسبت به BFS به جواب رسیده است. هرچند که هر چه عمق فاصله بیشتر شود این اختلاف کمتر می‌شود. چون درخت حالات عمیقتر شده و حالات تکراری به شکل نمایی زیاد می‌شوند.

BFS

این الگوریتم در بخش [7] IN پیاده سازی شده است. پیاده‌سازی آن یک حلقه بینهایت برای پیدا کردن جواب است. از یک صف برای ذخیره‌سازی استیتهایی که باید بررسی

شوند استفاده شده است و از یک دیکشنری برای استیت‌های ویزیت شده. علت استفاده از دیکشنری بازدهی بالاتر آن برای چک کردن وجود داشتن یک استیت بود.
چون دیکشنری از لیست به عنوان key پشتیبانی نمی کند از یک تابع my_hash استفاده شده است که که در ازای هر حالت جدول یک استرینگ ۸ کاراکتری بر می‌گرداند.
چیزی که در این الگوریتم قطعی است این است که هر استیت فقط یک بار بررسی می‌شود ولی مشکل بارز آن استفاده از حافظه زیاد و به همین علت پایین آمدن سرعت برنامه به علت save و load است. همچنین در این الگوریتم نمی‌توان فاصله جواب را به سادگی فهمید و برای همین یک صف move_count برای این منظور در نظر گرفته شده است.

A*

این الگوریتم در بخش [8] IN پیاده سازی شده است. پیاده‌سازی آن دقیقا همانند BFS است. با این تفاوت که به جای آن که استیت بعدیمان اولین عضو صف باشد. عضوی است که کمترین امتیاز را داراست. یعنی تابع هیوریستیک آن تعداد تهدیدهاست. امتیاز هر وضعیت تعداد تهدیدات آن به علاوه تعداد حرکات انجام شده ضرب در ۱.۲ است. مقدار ۱.۲ با تست کردن اعداد مختلف به دست آمده است. اگر ضریب تعداد حرکات را صفر بگذاریم یک Greedy bfs داریم که سریعترین زمان رسیدن به جواب را دارد ولی بهترین جواب را به ما ارایه نمیکند. جوابهای با ضریب ۱.۲ در اینجا برای هر ۶ فایل بهینه‌ترین جوابها هستند. (جوابها از طریق الگوریتمهای دیگر تست شده اند) پس این تابع هیرویستیک ما یک تابع خوب حساب میشود چون سرعت آسیب زیادی نیز نمیبیند. برای مثال اگر ضریب ۱,۵ باشد زمان به جواب رسیدن بالا ۱۰ ثانیه میرود.
سرعت این الگوریتم همانطور که در نتایج پیداست به مقدار فاصله ما تا جواب نهایی بستگی چندانی ندارد و این نشان میدهد که در تعداد بالا تنها الگوریتمی که به خوبی کار می‌کند همین است.
این الگوریتم حافظه بیشتری به نسبت dfs می گیرد ولی چون سریع به جواب میرسد مقدار حافظه گرفته شده آن در مجموع زیاد نمی‌شود.

از تفاوت فاحش آن در مجموع حرکات انجام شده نسبت به دو الگوریتم دیگر نیز نمی‌توان چشم پوشی کرد.