دانشکده‌ی مهندسی کامپیوتر

پروژه‌ی اول مبانی و کاربردهای هوش مصنوعی

نام استاد : بهنام روشنفکر

نام دانشجو : آریان بوکانی

شماره دانشجویی : ۹۷۳۱۰۱۲

نیم‌سال اول ۰۰-۹۹

نحوه‌ی پیاده‌سازی:

برای حل مساله از کلاس‌های زیر استفاده شده است.

- **Card**
- **Column**
- **State**
- **Node**

کلاس `Card` شامل اطلاعات مربوط به یک کارت یعنی رنگ و شماره است.
کلاس `Column` شامل لیستی از `Card` های یک ستون است.
کلاس `State` شامل لیستی از `Column` های مساله است.
و در نهایت کلاس `Node` شامل اطلاعات مربوط به هر گره در هنگام توسعه در جست‌وجو هاست.

در کلاس `Node` تابع `compare` برای مقایسه‌ی دو کارت استفاده شده که بسته به ورودی و رنگ و شماره‌ها چهار حالت خروجی دارد که یکی از حالات `Enum` تعریف‌شده در مسئله یعنی `cardComparisons` است.

تابع `checkAvailability` در کلاس `Column` برای چک کردن اینکه آیا کارت داده شده در ورودی توانایی وارد شدن به این ستون را دارد یا خیر گذاشته شده است. تابع `putCardOnTop` و `removeCardFromTop` برای گذاشتن و برداشتن کارتهای ستون نوشته شده‌اند. و در نهایت تابع `checkValidation` درست بودن قرارداده شدن کارت‌های ستون را چک می‌کند.(حالتی که کارت‌ها همرنگ باشند و به صورت نزولی چیده‌شده باشند)

تابع `checkTermination` در کلاس `State` نهایی بودن حالت را بررسی می‌کند.(همه‌ی ستون‌ها درست چیده‌شده باشند) تابع `validActions` حرکت‌های مجاز برای حالت کنونی را در قالب لیستی از `tuple` ها و بر اساس محدودیت‌های مسئله برمی‌گرداند.

در کلاس `Node` تابع `__hash__` برای قابل ذخیره شدن در `set` و `dict` گذاشته شده است.

پی‌نوشت: کلاس‌های بالا همگی در فایل `Essentials.py` پیاده‌سازی شده‌اند.

تابع `child` در `Essentilas.py` برای تولید فرزند یک گره بر اساس ورودی‌های تابع استفاده می‌شود.

با استفاده از تابع `readInputs` ورودی‌های مسئله را می‌خوانیم. اگر این تابع ورودی داشته باشد(اسم فایل `txt` مربوط به ورودی‌ها) می‌توان ورودی‌ها را به همان فرمت دستورکار در قالب فایل `txt` خواند و اگر ورودی نداشته باشد به همان صورت در کنسول ورودی‌ها را می‌گیرد.

تابع `showResults` برای نمایش نتیجه‌ی جست‌وجو بکار برده می‌شود.

BreadthFirstSearch

برای پیاده‌سازی این روش جست‌وجو گرافی از دو موجودیت frontier و explored استفاده شده است که به ترتیب deque و set هستند. در ابتدای هر مرحله‌ی توسعه‌ی گره، نهایی بودن گره گرفته شده از frontier تست می‌شود و اگر شرط نهایی بودن ارضا شود، گره کنونی برگردانده می‌شود وگرنه گره کنونی بر اساس حرکت‌های مجاز توسعه داده می‌شود و فرزندهایش تولید می‌شوند. در هر بار تولید گره فرزند، چک می‌شود که آیا گره تولید شده قبلا دیده شده است یا نه (explored). اگر دیده‌نشده بود به frontier که یک صف FIFO است اضافه می‌شود و totalCreatedNodes یکی زیاد می‌شود. وگرنه ignore می‌شود و به گره بعدی پرداخته می‌شود.

IterativeDeepeningSearch

در ابتدا باید روش جست‌وجوی درختی DepthLimitedSearch توسعه داده شود. برای اینکار از frontier که یک موجودیت deque و با خصوصیت LIFO است استفاده شده است. این تابع به عنوان ورودی یک عدد که حداکثر عمق جست‌وجو است را نیز باید بگیرد. (limit) تا زمانی که frontier خالی نشده است، عملیات تولید فرزند و تست حالت نهایی انجام می‌شود.(تا زمانی که عمق فرزندها از مقدار limit بیشتر نشود این عملیات انجام می‌شود)

درنهایت نیز با استفاده از تابع IterativeDeepeningSearch مقدار limit یکی یکی زیاد می‌شود و تابع DepthLimitedSearch را صدا می‌زند تا در نهایت به جواب غیربهینه‌ای برسد.

A* Search

برای پیاده‌سازی A* از دو موجودیت frontier و cost بهره برده شده است. Frontier یک صف اولویت است که گره‌ها بر اساس مقدار تابع ارزیاب خود در آن چیده می‌شوند. هر کدام که دارای مقدار کمتری باشند، اولویت انتخاب بیشتری دارند. هم‌چنین cost یک دیکشنری است که کلید آن گره‌های تولید شده و مقدار آنها، هزینه‌ی پرداخت شده برای رسیدن به آن گره از ریشه است. (هزینه‌ی هر گره تولیدی یکی بیشتر از گره والد آنهاست) در هنگام تولید گره‌های فرزند، در صورتی که گره تولیدشده در لغت‌نامه‌ی cost وجود نداشته باشد(گره قبلا تولید نشده است)، هزینه‌ی تولید این گره در cost ذخیره می‌شود و priotity آن بر اساس تابع ارزیاب (f(n)=g(n)+h(n)) به frontier پاس داده می‌شود و totalCreatedNodes یکی زیاد می‌شود.

توضیح هیوریستیک استفاده شده برای مسئله

برای این مسئله‌ی بخصوص هیوریستیک زیر در نظر گرفته شده است:

در میان همه‌ی ستون‌های یک حالت شروع به حرکت می‌کنیم. شماره‌ی کارت زیری را در `temp` ذخیره می‌کنیم. یکی یکی کارت‌های روی زیری‌ترین کارت را بررسی می‌کنیم. در صورتی‌که رنگ کارت مقایسه‌شونده با رنگ کارت زیری فرق داشته باشد، یا اینکه شماره‌ی کارت از `temp` بیشتر باشد، به اندازه‌ی تعداد کارت‌های بعدی این کارت و هم‌چنین خود این کارت به یک متغیر که در ابتدا صفر بوده است. اضافه می‌کنیم و دستور `break` را انجام داده و به ستون بعدی خواهیم رفت. در غیر این صورت شماره‌ی کارت را در `temp` ذخیره می‌کنیم.

این هیوریستیک قابل‌قبول است. زیرا برای اینکه کارت ناسازگار پیدا شده جابه‌جا شود باید همه‌ی کارت‌های روی آن جابه‌جا شوند و این حداقل حرکت ممکن برای رسیدن به یک حالت نهایی است.
این هیوریستیک سازگار نیز هست. زیرا با انجام یک حرکت، در بهترین حالت یک کارت ناسازگار در یک ستون به یک ستون دیگر رفته و در آنجا سازگار است. در نتیجه مقدار هیوریستیک `Node` جدید یکی کمتر از `Node` والد خود است. در بدترین حالت نیز یک کارت سازگار در یک ستون به یک ستون دیگر منتقل شده و در آنجا ناسازگار می‌شود. در نتیجه هیوریستیک یکی اضافه می‌شود. از آنجا که هزینه‌ی هر حرکت 1 در نظر گرفته می‌شود، خواهیم داشت:

```
h(child) + 1 >= h(parent)
```

بنابراین هیوریستیک در نظر گرفته شده سازگار و قابل قبول است.

مقایسه‌ی عملکرد میان روش‌های جست‌وجو

تست‌های داده شده به سوالات:

```
1.
5 2 5
#
1r 5y 3r
5r 2r
4y 1y 2y 4r
3y
```

	BFS	**IDS**	**A***
گره‌های تولیدی	280723	764651	1161
گره‌های بازشده	59632	764628	185
عمق جواب	6	6	6

```
2.
3 2 3
1y 1r
3r 2y
3y 2r
```

	BFS	**IDS**	**A***
گره‌های تولیدی	320	13195	71
گره‌های بازشده	222	13195	48
عمق جواب	7	7	7

```
3.
4 3 3
1r 2b 3y
2r 1y 3r
#
3b 1b 2y
```

	BFS	**IDS**	**A***
گره‌های تولیدی	838973	34482622	2807
گره‌های بازشده	318755	34482601	838
عمق جواب	9	9	9

4.
3 2 3
#
#
3r 3b 1r 1b 2b 2r

	BFS	**IDS**	**A***
گره‌های تولیدی	1055	31100	169
گره‌های بازشده	785	31093	108
عمق جواب	9	9	9

پی‌نوشت: حد ابتدایی عمق برای IDS، صفر در نظر گرفته شده است.(به همین دلیل تعداد گره‌های تولیدی و بازشده‌ی آن زیاد است)