دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

مبانی و کاربردهای هوش مصنوعی

پروژه یک

رادین شایانفر

پاییز ۱۳۹۹

پروژه یک

مدل‌سازی مسئله:

شرح هر یک از کلاس‌ها برای مدل‌سازی مسئله در ادامه آمده است.

- کلاس Card: این کلاس برای ذخیره‌سازی رنگ (color) و شماره (number) هر کارت بازی استفاده می‌شود.
- کلاس Batch: این کلاس برای نگهداری هر ستون از کارت‌های بازی به کار می‌رود. جزییات قسمت‌های مختلف Batch در زیر آمده است.
  - پشته‌ی cards: برای نگهداری کارت‌های موجود در هر ستون از یک صف LIFO یا یک پشته (stack) از کارت‌ها (اشیای کلاس Card) استفاده می‌شود.
  - خصوصیت sortCount: تعداد کارت‌هایی که با شمارش از ابتدای (پایین) پشته‌ی cards هم‌رنگ هستند و ترتیب نزولی دارند در این متغیر ذخیره می‌شود. به عنوان مثال برای ترتیب زیر از کارت‌ها در یک ستون مقدار sortCount برابر ۳ خواهد داشت. به این دلیل که تنها ۳ کارت اول از ابتدای پشته هم‌رنگ و دارای ترتیب نزولی هستند.

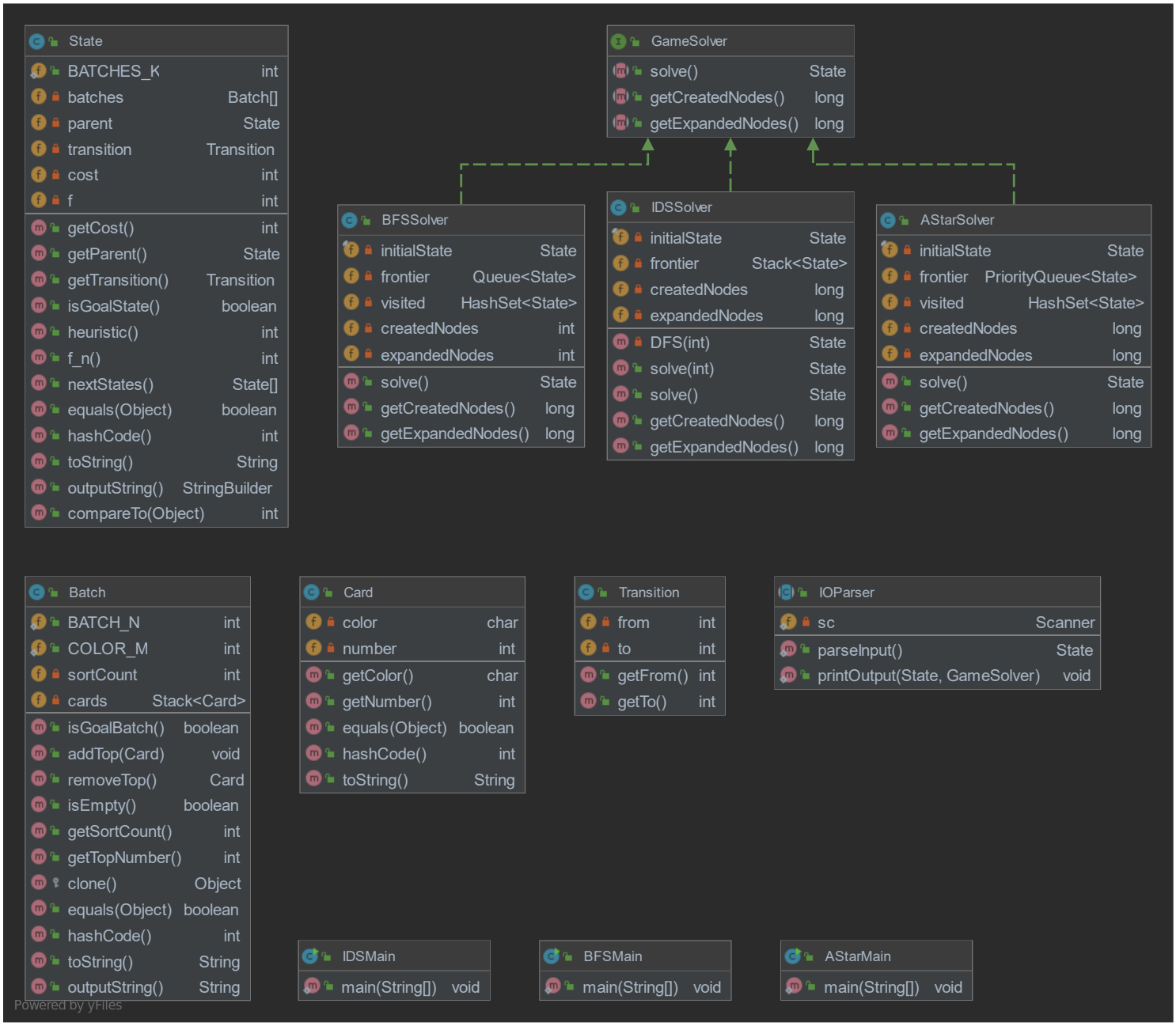
    6r 5r 3r 2g 1g 1b

    در ادامه نحوه‌ی به‌روزرسانی مقدار این متغیر در متدهای addTop و removeTop آمده است. از این متغیر برای تعیین هدف بودن هر حالت (state) از بازی و تعیین مقدار هیوریستیک آن استفاده می‌شود.

  - متد addTop: این متد یک کارت جدید بر روی پشته‌ی cards اضافه می‌کند. در این متد در صورتی که پیش از این تمامی کارت‌های این ستون ترتیب درست (هم‌رنگ و نزولی بودن) داشتند مقدار sortCount یک واحد افزایش می‌یابد.
  - متد removeTop: این متد یک کارت را از روی پشته‌ی cards حذف می‌کند و آن را برمی‌گرداند. در این متد در صورتی که پیش از این تمامی کارت‌های این ستون ترتیب درست (هم‌رنگ و نزولی بودن) داشتند مقدار sortCount یک واحد کاهش می‌یابد.
  - متد isGoalBatch: در این متد اگر ترتیب چیده شدن تمامی کارت‌ها درست باشد (یعنی هم‌رنگ و نزولی باشند) مقدار true و در غیر این صورت false برگردانده می‌شود. این کار در زمان O(1) و تنها با چک کردن برابری sortCount با تعداد کارت‌های پشته‌ی cards انجام می‌پذیرد.
  - متد isEmpty: از این متد برای چک کردن خالی بودن ستون کارت‌ها استفاده می‌شود.
  - متد getTopNumber: شماره‌ی کارت بالای پشته‌ی cards را برمی‌گرداند.
- کلاس State: برای ذخیره‌سازی هر حالت از بازی استفاده می‌شود. شرح بخش‌های مختلف این کلاس در ادامه آمده است.

- آرایه batches: یک آرایه از اشیای کلاس Batch که ستون‌های موجود در هر حالت (state) را نگهداری می‌کند.
- خصوصیت cost: هزینه‌ی حالت (state) در این متغیر ذخیره می‌شود.
- خصوصیت f: مقدار f(s) = cost(s) + heuristic(s) در این متغیر ذخیره می‌شود.
- خصوصیت parent: خصوصیتی از جنس State که حالت والد هر حالت را نگهداری می‌کند. از این ویژگی برای تولید مسیر رسیدن به هدف پس از پیدا شدن آن استفاده می‌شود.
- خصوصیت transition: شی از جنس کلاس Transition که آخرین عمل برداشته شدن و گذاشته شدن کارت‌ها برای رسیدن به آن حالت را نگهداری می‌کند. از این ویژگی برای چاپ حرکت‌های انجام شده برای رسیدن به هدف در انتهای برنامه استفاده می‌شود.
- متد nextStates: تمام حالت‌های پسین حالت فعلی را در آرایه‌ای از Stateها برمی‌گرداند. برای این کار تمام ترتیب‌های دو تایی ممکن از ستون‌ها در نظر گرفته می‌شوند و با برداشتن کارت بالایی از یک ستون ناتهی و انتقال آن به ستون دیگر، یک حالت جدید ایجاد می‌شود. در این متد خصوصیت‌های cost و parent برای حالت‌های ساخته شده توسط والد آن‌ها مقدارگذاری می‌شود.
- متد heuristic: این متد برای تعیین مقدار هیوریستیک حالت استفاده می‌شود. نحوه‌ی محاسبه‌ی این هیوریستیک در بخش‌های بعدی آمده است.
- متد f_n: این متد در اولین فراخوانی مقدار cost + heuristic() را در متغیر f ذخیره می‌کند و در فراخوانی‌های بعدی تنها مقدار f را برمی‌گرداند.
- متد isGoalState: تعیین می‌کند که حالت مورد نظر هدف است یا نه. در صورتی که isGoalBatch برای همه‌ی ستون‌های یک حالت مقدار true داشته باشد آن حالت هدف است و مقدار true توسط این متد بازگردانی می‌شود. در غیر این صورت مقدار false برگردانده می‌شود.

- کلاس Transition: این کلاس برای نگهداری نحوه‌ی تولید هر حالت (state) استفاده می‌شود. در این کلاس شماره ستونی که در هنگام تولید حالت از آن کارت برداشته شده است (from) و شماره ستونی که کارت روی آن گذاشته شده است (to) ذخیره می‌شوند.
- واسط GameSolver: این واسط متد solve را اعلان می‌کند. واسط GameSolver توسط هر یک از کلاس‌های BFSSolver، IDSSolver و AStarSolver پیاده‌سازی می‌شود. توضیحات مربوط نحوه‌ی عملکرد این سه الگوریتم در بخش بعدی آمده است.
- کلاس IOParser: این کلاس متدهایی ایستا (static) برای اعمال ورودی گرفتن از کاربر و چاپ خروجی فراهم می‌کند.



شکل (۱) – نمودار UML کلاس‌های برنامه

پیاده‌سازی الگوریتم‌های جست‌وجو:

در این برنامه سه الگوریتم BFS، IDS و *A برای حل مسئله پیاده‌سازی شده‌اند. این الگوریتم‌ها به ترتیب توسط BFSMain، IDSMain و AStarMain اجرا می‌شوند. این سه الگوریتم با گرفتن حالت ابتدایی و با فراخوانی متد solve به جست‌وجوی حالت هدف در مسئله می‌پردازند. در تمامی این الگوریتم‌ها آزمون هدف در هنگام بسط هر حالت انجام می‌شود. نحوه‌ی پیاده‌سازی این الگوریتم‌ها در زیر آمده است.

- الگوریتم BFS: در این الگوریتم حالت‌های تولید شده داخل یک صف FIFO به نام frontier قرار می‌گیرند و به همان ترتیبی که وارد شده‌اند برای بسط داده شدن انتخاب می‌شوند. هم‌چنین حالت‌هایی که پیش از این بازدید شده‌اند در مجموعه visited نگهداری می‌شوند تا حالت‌های تکراری بسط داده نشوند.
- الگوریتم IDS: در این الگوریتم حالت‌های تولید شده در یک صف LIFO یا یک پشته (stack) قرار می‌گیرند و برای بسط داده شدن از پشته خارج می‌شوند. برخلاف الگوریتم BFS در اینجا حالت‌های بازدید شده ذخیره نمی‌شوند و حافظه کمتری مصرف می‌شود. برای جلوگیری از به وجود آمدن مشکلات جست‌وجوهای درختی عمق جست‌وجو در هر مرحله محدود می‌شود (آرگومان limit). در صورت پیدا نشدن هدف یک واحد به حداکثر عمق اضافه می‌شود و جست‌وجو مجدد از ابتدا انجام می‌شود.
- الگوریتم *A: این الگوریتم درست مانند الگوریتم BFS عمل می‌کند با این تفاوت که حالت‌های ایجاد شده در یک صف اولویت‌دار (PriorityQueue) قرار می‌گیرند. در این صف اولویت هر حالت برابر مقدار f(s) آن حالت است و در هر مرحله حالتی که کمترین f(s) را دارد برای بسط داده شدن انتخاب می‌شود.

تابع هیوریستیک:

برای محاسبه‌ی هیوریستیک هر حالت، مقدار sortCount برای هر ستون را جمع می‌زنیم و از تعداد کل کارت‌ها کم می‌کنیم. به صورت دقیق‌تر مقدار هیوریستیک یک حالت برابر است با:

$$h(s) = mn - \sum_{i=1}^{k} sortCount_i$$

که در آن $sortCount_i$ مقدار sortCount برای ستون iام است.

در این هیوریستیک در واقع تعداد کارت‌هایی که سر جایشان نیستند شمرده می‌شود. این هیوریستیک در واقع نسخه‌ی ساده شده (relaxed) از مسئله را در نظر می‌گیرد. به این صورت که فرض می‌کند روی هم چیده‌شدن کارت‌ها در یک ستون محدودیتی برای تغییر دادن ستون آن‌ها ایجاد نمی‌کند و هم‌چنین محدودیت اینکه هر کارت روی کارت بزرگ‌تر از خودش قرار می‌گیرد نیز وجود ندارد. در نتیجه همیشه مقدار h(s) از مقدار واقعی کمتر است و این هیوریستیک قابل قبول (admissible) است.

از طرف دیگر این هیوریستیک سازگار (consistent) نیز می‌باشد. طبق تعریف هیوریستیکی سازگار است که برای هر حالت s و پسین آن (s′) رابطه زیر برقرار باشد:

$$h(s) \leq c(s, a, s') + h(s')$$

در این مسئله از آن‌جا که در هر حرکت تنها یک کارت جابجا می‌شود پس اختلاف h(s) و h(s′) یک واحد است. یا به عبارتی:

$$|h(s) - h(s')| \leq 1$$

از طرفی هزینه‌ی هر حرکت برابر یک است یا $c(s, a, s') = 1$.

در نتیجه شرط سازگاری برای این هیوریستیک برقرار است.

مقایسه‌ی الگوریتم‌های BFS، IDS و *A:

برای مقایسه‌ی الگوریتم‌های جست‌وجو از نمونه ورودی زیر استفاده می‌کنیم:

```
4 3 3
#
1r 3y 3g 2y
1y 2g
1g 2r 3r
```

خروجی هر الگوریتم را به ازای ورودی بالا می‌بینیم.

- الگوریتم BFS:

```
3g 2g
3r 2r 1r
3y 2y 1y
1g
========
Depth: 11
Created nodes: 3875
Expanded nodes: 1523
========
Actions:
2 => 4
2 => 1
3 => 1
3 => 1
2 => 3
4 => 3
1 => 3
2 => 1
4 => 2
4 => 2
1 => 2
```

- الگوریتم IDS (با عمق اولیه صفر):

```
3g 2g
3r 2r 1r
3y 2y 1y
1g
========
Depth: 11
Created nodes: 68192605
Expanded nodes: 68192596
========
Actions:
2 => 4
2 => 1
3 => 1
3 => 1
2 => 3
4 => 3
2 => 3
4 => 2
4 => 2
3 => 2
1 => 3
```

- الگوریتم *A:

```
3g 2g
3r 2r 1r
3y 2y 1y
1g
========
Depth: 11
Created nodes: 1047
Expanded nodes: 359
========
Actions:
2 => 4
2 => 1
3 => 1
3 => 1
2 => 3
4 => 3
2 => 3
4 => 2
4 => 2
3 => 2
1 => 3
```

همان‌طور که می‌بینیم الگوریتم IDS به علت استفاده بسیار کم از حافظه تعداد گره (node) های بسیار بیشتری نسبت به دو الگوریتم دیگر تولید می‌کند و بسط می‌دهد. با مقایسه‌ی BFS و A* نیز از آنجا که الگوریتم A* از هیوریستیک مناسبی استفاده می‌کند و به نوعی آگاهانه جست‌وجو می‌کند تعداد گره‌هایی که تولید می‌کند و بسط می‌دهد از BFS کمتر است و به نظر از دو الگوریتم دیگر از نظر زمانی عموما سریع‌تر است.

عمق جواب اما در هر ۳ الگوریتم برابر و برابر ۱۱ است. می‌دانیم که BFS و IDS هر دو جواب بهینه را پیدا می‌کنند. A* نیز از آن‌جا که هیوریستیک مورد استفاده سازگار است جواب بهینه را پیدا می‌کند. در نتیجه هر ۳ الگوریتم بهینه هستند و کم‌هزینه‌ترین جواب را پیدا می‌کنند. به همین دلیل هر سه در عمق ۱۱ به هدف رسیده‌اند و از این جهت تفاوتی با یکدیگر ندارند.

در نهایت خروجی الگوریتم A* را برای ورودی زیر (ورودی نمونه در صورت پروژه) می‌بینیم. لازم به ذکر است به علت عمق زیاد جواب هیچ یک از الگوریتم‌های BFS و IDS نتوانستند در زمان مناسبی به حالت نهایی برسند.

ورودی:

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

(خروجی در صفحه بعد)

خروجی:

```
5g 4g 3g 1g
2g
4r
5r 3r 2r 1r
5y 4y 3y 2y 1y
========
Depth: 22
Created nodes: 2066067
Expanded nodes: 427019
========
Actions:
3 => 2
4 => 3
4 => 5
4 => 3
1 => 5
4 => 5
1 => 4
3 => 1
3 => 4
1 => 4
2 => 4
3 => 1
3 => 1
2 => 3
1 => 3
2 => 1
5 => 1
2 => 5
3 => 2
3 => 5
2 => 5
2 => 3
```