تمرین سری اول

درس هوش مصنوعی

سارا شاه‌محمدی

۳- مشکل روش جستجوی BFS مصرف زیاد حافظه است به نحوی که جستجو به علت مصرف زیاد حافظه می‌شود. در پیاده‌سازی ما (در صورتی که حالت‌های تکراری حذف نشود) جستجو تا عمق ۱۶ پیش می‌رود.

۴- مشکل DFS (در صورتی که حالت‌های تکراری حذف نشوند و دور داشته باشیم) این است که اولا ممکن است در دور گیر کند و terminate نکند؛ دوما جواب بهینه را پیدا نکند؛ سوما ممکن رسیدن به جواب مدتی بسیار طولانی به درازا بکشد.

نقطه قوت آن در صرفه‌جویی در مصرف حافظه است. این الگوریتم مانند BFS حافظه زیادی مصرف نمی‌کند، اما همان‌طور که گفتیم به لحاظ زمانی نسبت به BFS مزیتی ندارد و در صورتی که در دور گیر کند، ممکن است جستجو تا ابد طول بکشد.

۵- مزیت IDS این است که جواب بهینه را قطعا پیدا می‌کند، مصرف حافظه‌ی بسیار کمتری نسبت به BFS دارد و بر خلاف DFS در دور گیر نمی‌کند. نقطه ضعف آن این است که با وجود آن که در دور گیر نمی‌کند، باز هم زمان بسیار زیادی طول می‌کشد تا به پاسخ برسد؛ چرا که در هر iteration باید تمام نودهای درخت را تا عمق مشخصی بررسی کند و این به معنای پیچیدگی زمانی ($O(b^d)$ است.

۶-

الگوریتم	***پیچیدگی حافظه***	***پیچیدگی زمانی***
BFS	$O(b^d)$	$O(b^d)$
DFS	O(bm)	$O(b^m)$/unbounded*
Iterative Deepening	O(bd)	$O(b^d)$

*در صورتی که نودهای تکراری را حذف نکنیم، ممکن است الگوریتم DFS در دور گیر کند و terminate نکند؛ در این حالت پیچیدگی زمانی unbounded است. در غیر این صورت، پیچیدگی زمانی ($O(b^m)$ است.

۷- یک تابع شهودی این است که هزینه رسیدن به حالت نهایی را در هر وضعیت، برابر با مجموع تعداد اعدادی که در جای غلط قرار گرفته‌اند در نظر بگیریم؛ یعنی هزینه رساندن هر عدد غلط جایگذاری‌شده را به جایگاه درستش یک در نظر بگیریم.

۴	۳	۱
۲	۵	۶
	۸	۷

برای مثال، در شکل روبه‌رو، هزینه رسیدن از این وضعیت به وضعیت نهایی برابر است با:

$h_1(n) = 4$

زیرا اعداد ۳، ۴، ۶ و ۲ در جای غلط قرار گرفته‌اند.

این روش admissible است زیرا این مساله حالت relaxed شده‌ی مساله‌ی اصلی است زیرا برای رساندن هر عدد غلط به جایگاه اصلیش حداقل یک حرکت لازم است.

تابع شهودی دیگر این است که هزینه رسیدن به وضعیت نهایی را مجموع فاصله منهتنی هر عدد تا جایگاه واقعیش در نظر بگیریم. برای مثال در شکل زیر هزینه رسیدن این وضعیت به وضعیت ایده‌ال با تابع شهودی منهتنی برابر با ۸ است.

۱	۳	۴
۶	۵	۲
۷	۸	

$h_2(n) = 1 + 3 + 2 + 2 = 8$

زیرا فاصله منهتنی عدد ۳ از جایگاه اصلیش برابر با ۱، فاصله عدد ۴ برابر با ۳، فاصله عدد ۶ برابر با ۲ و فاصله عدد ۲ برابر با ۲ است.

این تابع نیز admissible است زیرا حالت relaxed شده‌ای از مساله اصلی است: برای رساندن هر عدد غلط به جایگاه اصلیش باید آن را حداقل به اندازه فاصله منهتنی‌اش از جایگاه اصلیش حرکت داد.

۸- در تابع شهودی اول، که relaxation بیشتری دارد، انتظار داریم رسیدن به پاسخ بیشتر طول بکشد؛ به این دلیل که $h_2(n)$، $h_1(n)$ را dominate می‌کند، زیرا تعداد حالت های بررسی شده در A* با تابع h1، حداقل به اندازه A* با تابع h2 است.

به علاوه، می‌توان این دو تابع را از منظر effective branching factor (EBF) با هم مقایسه کرد. EBF برابر است با مقدار b*ای که معادله زیر را حل می‌کند (N تعداد حالت های بررسی شده و d عمق پاسخ است.) [1]

$$x^d + x^{d-1} + \ldots + x + 1 = N$$

هرچه مقدار این عدد به یک نزدیک‌تر باشد، تابع شهودی ما تابع بهتری است. مقدار EBF محاسبه‌شده برای هر دو این توابع (که در کتاب راسل و نورویگ نیز آمده است)، نشان می دهد که h_2 تابع بهتری است؛ زیرا مقدار EBF در آن کمتر است. مقادیر محاسبه شده EBF برای هر دو این توابع در جدول ۱ در زیر گزارش شده است. همان‌طور که می‌بینید مقدار آن در h2 کمتر از h1 و همواره به یک نزدیکتر است.

هم‌چنین، می‌توان گفت اگر h_2، h_1 را dominate کند، مقدار EBF آن هرگز از h_1 بیشتر نمی‌شود.

عمق	A*(h1)	A*(h2)
۲	۱/۷۹	۱/۷۹
۴	۱/۴۸	۱/۴۵
۶	۱/۳۴	۱/۳۰
۸	۱/۳۳	۱/۲۴
۱۰	۱/۳۸	۱/۲۲
۱۲	۱/۴۲	۱/۲۴
۱۴	۱/۴۴	۱/۲۳
۱۶	۱/۴۵	۱/۲۵
۱۸	۱/۴۶	۱/۲۶
۲۰	۱/۴۷	۱/۲۷
۲۲	۱/۴۸	۱/۲۸
۲۴	۱/۴۸	۱/۲۶

جدول ۱- مقدار EBF در دو تابع شهودی h1 و h2 [2]

[1] http://www.divms.uiowa.edu/~tinelli/classes/145/Fall05/notes/4.2-informed-search.pdf

[2] http://www.cs.nott.ac.uk/~pszgxk/courses/g5aiai/004heuristicsearches/heuristic-searches.htm

مشاهده ما نیز از نحوه عملکرد این دو تابع با آن‌چه در بالا گفته شد، همخوانی داشته است و الگوریتم با تابع شهودی اول معمولا زمان بیشتری برای رسیدن به پاسخ صرف می‌کند.

۹- به صورت کلی، زمان و حافظه مصرف‌شده در A* به تابع شهودی آن بستگی دارد. اما پیچیدگی زمانی الگوریتم A* را می‌توان $O(b^d)$ و پیچیدگی حافظه آن برابر با $O(b^d)$ است؛ زیرا در بدترین حالت، در صورتی که تابع شهودی خوبی انتخاب نکرده باشیم، ممکن است تمام نودهای درخت تا عمق d را ببینیم. ممکن است نودی در عمق k ببینیم و به عمق k+1 برویم و دوباره، به عمق k بازگردیم.
در مقایسه با الگوریتم‌های جستجوی غیرآگاهانه، A* به شرط انتخاب تابع شهودی مناسب، ممکن است به طور متوسط branching factor کوچکتری نسبت به الگوریتم‌های ناآگاهانه داشته باشد (همان‌طور که در پاسخ به پرسش ۸ دیدیم)؛ یعنی نیازی به مشاهده تمام نودها تا جواب وجود نداشته باشد. این امر می‌تواند پیچیدگی زمانی و حافظه را به طور متوسط کاهش دهد، زیرا b (پایه تابع نمایی) را کاهش داده است. با این حال، همچنان یک تابع نمایی است و این امر به دلیل پیچیدگی ذاتی مساله اجتناب‌ناپذیر است.

۱۰- یک روش non-admissible این است که هزینه رسیدن به مقصد به صورت تصادفی تولید شود. در این صورت، تضمینی برای رسیدن به جواب وجود ندارد. در صورت یافتن پاسخ نیز ممکن است مسیر رسیدن به پاسخ طولانی‌تر شود و زمان رسیدن به آن نیز به طرز قابل‌توجهی افزایش یابد.

گزارش کار

یک نکته

یک نکته بسیار مهم در مساله پازل هشت تکه این است که بعضی حالت‌های این مساله قابل‌حل نیستند. قابل‌حل بودن یا نبودن حالت ابتدایی بر اساس عرض جدول (که ما ۳ در نظر گرفتیم) و تعداد inversion هاست. اگر تعداد خانه‌های جدول فرد باشد، تعداد inversionها باید زوج باشد تا مساله قابل‌حل باشد. برای مثال به شکل زیر نگاه کنید:

۱	۳	۴
۶	۵	۲
۷	۸	

برای پیدا کردن تعداد inversionها باید اعداد پازل را در یک ردیف بنویسید؛ یعنی:
1, 3, 4, 6, 5, 2, 7, 8
در این صورت، inversion جفتی به صورت (a, b) است که a پیش از b ظاهر شود، اما a>b.[۳]یعنی در این حالت:

عدد ۱: inversion ندارد

عدد ۳: ۱ inversion دارد

عدد ۴: ۱ inversion دارد

عدد ۶: ۲ inversion دارد

عدد ۵: ۱ inversion دارد

عدد ۲: inversion ندارد

عدد ۷: inversion ندارد

عدد ۸: inversion ندارد

پس مجموع تعداد inversionها در این حالت برابر با ۵ است و از آن‌جا که عرض جدول برابر عددی فرد است، این حالت ابتدایی قابل حل نیست.

اما برای مثال، حالت زیر قابل‌حل است:

۱	۲	۳
۴	۶	۵
۸	۷	

زیرا تعداد inversionها برابر با ۲ است.

ما نیز تابعی به نام isSolvableGrid3 نوشتیم که در ابتدای برنامه قابل‌حل بودن مساله را بررسی می‌کند و در صورتی که مساله قابل حل نباشد، پیغام مناسب را به نمایش می‌گذارد.

[3] https://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html

شرح مختصری از برنامه ما در زیر آمده است:

ما Stack و Queue را خودمان پیاده‌سازی کردیم؛ ولی از PriorityQueue کتابخانه‌ای جاوا استفاده کردیم.

علاوه بر آن، دو کلاس دیگر تعریف کردیم:

۱. اول کلاس EightPuzzle که حاوی همه روش‌های جستجو است و دریافت و پردازش ورودی و الخ در این کلاس انجام می‌شود. این کلاس حاوی دو instance variable است؛ (۱) filterRepeatedStates که برای حذف حالات تکراری است و به صورت پیش‌فرض true در نظر گرفته می‌شود. (۲) یک هش‌تیبل از حالاتی که مشاهده شده است و در روش‌های جستجو، در صورتی که بخواهیم حالات تکراری را حذف کنیم، از این هش‌تیبل استفاده می‌کنیم. کلیدهای این دیکشنری حالت‌های مشاهده‌شده (که توسط تابع convertToNumeric به یک عدد نه‌رقمی تولید می‌کند) و مقادیر آن عمق حالت‌های مشاهده‌شده است.

۲. دوم کلاس Node: ویژگی‌های این کلاس شامل لیستی از نودهای فرزند، نود والد، عمق، حالت نود و اندیس فرزند بعدی که باید مشاهده کنیم، است.

   i. کلاس ComparableNode فرزند کلاسNode است و خود نیز سه فرزند دارد که در روش A* از فرزندان این کلاس استفاده می‌کنیم؛ زیرا متد compare موجود در اینترفیس comparator و متد compareTo موجود در اینترفیس Comparable در فرزندان این کلاس پیاده‌سازی تا بتوان از PriorityQueue برای پیاده‌سازی الگوریتم A* استفاده کرد.

   ii. سه فرزند این کلاس: ManhattanDistanceNode, DistanceHeuristicNode, UnadmissibleHeuristicNode هستند که در هر کدام از این زیرکلاس‌ها متدهای compareTo و compare بر اساس تابع شهودی در نظر گرفته‌شده محاسبه می‌شوند. در DistanceHeuristicNode تابع شهودی ما تعداد عددهایی است که در جایگاه اشتباه قرار دارند. در ManhattanDistanceNode تابع شهودی ما فاصله منهتی هر عددی که به اشتباه جایگذاری شده است تا جایگاه مطلوبش است و در UnadmissibleHeuristicNode، تابع شهودی ما فاصله منهتی هر عددی که به اشتباه جایگذاری شده است به علاوه یک عدد رندوم است. برای محاسبه فاصله، از تابع getCoordinates که در کلاس ComparableNode پیاده‌سازی‌شده است، استفاده می‌کنیم. این تابع عددی اندیس مطلوب اعداد ۱ تا ۸ را در پازل بازمی‌گرداند؛ برای مثال اندیس مطلوب عدد ۷ (۰، ۲) است یا اندیس مطلوب عدد ۵، (۱، ۱) است.

- تابع getInput: هشت عدد ورودی آرایه را از کاربر می‌گیرد. در صورتی که عدد تکراری وارد شود، به کاربر پیام خطا می‌دهیم که عدد صحیحی وارد کند و دوباره ورودی می‌گیریم. در صورتی که عددی بزرگتر از ۸ یا کوچکتر از ۱ وارد شود نیز آرایه ورودی را دوباره می‌گیریم.
- تابع create2dInitialState: ورودی را به همان صورتی که در صورت تمرین ذکر شده به صورت یک آرایه یک بعدی دریافت می‌کند و یک آرایه دو بعدی سه در سه از آن می‌سازد که جای خانه خالی، صفر قرار می‌گیرد. علت تبدیل آرایه ورودی به آرایه دو بعدی این است که ما ه بخشی از کد را پیش از دریافت

صورت تمرین با آرایه دو بعدی نوشته بودیم و برای جلوگیری از به وجود آمدن تعارض تابع create2dInitialState را نوشتیم که مجبور به دست بردن در کد قبلی نشویم.

- تابع printState: یک آرایه دوبعدی که نشانگر حالت پازل است، می‌گیرد و آن را چاپ می‌کند.
- تابع printSolution: یک نود و یک رشته می‌گیرد و مسیر رسیدن به پاسخ را چاپ می‌کند.
- تابع getAnswerLength: نود حالت نهایی (حالت مطلوب) را می‌گیرد و طول مسیر جواب (رسیدن از حالت آغازین به حالت نهایی) را برمی‌گرداند.
- تابعgenerateNextStates : یک کانفیگوریشن از پازل را به صورت یک آرایه دو بعدی می‌گیرد و حالت‌های ممکن بعدی را در arraylistای برمی‌گرداند. هر عضو این arraylist خود یک آرایه دو بعدی است. برای این بخش مجبور به پیاده‌سازی تابع deepCopy شدیم؛ چرا که در غیر این صورت پس از یک حرکت حالت اولیه صفحه پازل به‌هم می‌ریخت. پس پیش از هر حرکت، یک کپی از حالت اولیه می‌سازیم و حرکت را روی آن کپی انجام می‌دهیم و سپس به arraylist اضافه می‌کنیم.
- تابع deepCopy: یک آرایه دوبعدی می‌گیرد و یک آبجکت جدید (با پوینتر جدید) می‌سازد.
- تابع testGoal: که حالت صفحه را به صورت آرایه دو بعدی می‌گیرد و تعیین می‌کند آیا این حالت، حالت مطلوب نهایی است یا خیر.
- تابع convertToNumeric: این تابع حالت صفحه را (به صورت یک آرایه دو بعدی) دریافت می‌کند و آن را به یک عدد نه‌رقمی تبدیل می‌کند.
- تابع isRepteated: در هر روش جستجو می‌توانیم اگر بخواهیم حالات تکراری را حذف کنیم یا خیر. در صورتی که نخواهیم حالات تکراری را حذف کنیم، تمام حالت‌ها را اعم از تکراری و غیرتکراری بررسی می‌کنیم. در صورتی که بخواهیم حالات تکراری را فیلتر کنیم، با فراخوانی convertToNumeric حالت را به یک عدد نه‌رقمی تبدیل می‌کنیم. اگر حالت در دیکشنری بود، عمق آن را چک می‌کنیم: اگر عمق این حالت بار قبلی که دیده شده از عمق فعلی کمتر بود، true برمی‌گردانیم (یعنی حالت تکراری است) و در غیر این صورت، آن را به دیکشنری اضافه کرده و false برمی‌گردانیم.
- توابع breadthFirstSearch ، depthFirstSearch، depthLimitedSearch و الخ همگی دو آرگومان می‌گیرند: یکی حالت اولیه مساله (که توسط تابع create2dInitialState تبدیل به آرایه دو بعدی شده است) و دیگری یک مقدار بولی که بیانگر این است که آیا می‌خواهیم حالات تکراری را حذف کنیم یا خیر.
- تابع runAllAlgorithms نیز تمام توابع جستجو را با ورودی کاربر فراخوانی می‌کند.