به نام خدا

گزارش پروژه ۱ هوش مصنوعی

معین شیردل

۸۱۰۱۹۷۵۳۵

به طور کلی در این پروژه یک کلاس Node و یک کلاس State استفاده شده است. هر Node درون خودش یک ID و یک Parent ID دارد که نشان دهنده ی Node ایست که در درخت ها، پدر این Node است. هر Node شامل یک State است و اطلاعات مربوط به آن را درون خود نگه می دارد.

هر State، یک حالت مجزا و خاص در جدول داده شده است. این State ها با توجه به مکان مار در جدول و امتیاز باقی مانده از هر دانه، از یکدیگر مجزا می شوند. پس هر استیت دارای یک لیست از امتیاز دانه ها و یک لیست از مختصات بدن مار در جدول را دارد. در ابتدای برنامه مختصات اولیه مار و طول و عرض جدول را ورودی می گیریم و یک State به عنوان استیت اولیه درست ایجاد می شود. سپس آن را به یک Node می دهیم و آن را init_node می نامیم و درخت ها را از این گره شروع به پیمایش می کنیم. برای هر State، حرکت های رفتن مار به هر ۴ جهت راست، پایین، چپ و بالا بررسی می شوند و فرزندان این Node می شوند. ترتیب بررسی فرزندان هر State بسته به الگوریتم search تغییر می کند.

برای بررسی فرزندان (به عبارتی حالت های جدول پس از حرکت مار) ابتدا بررسی می شود که مار به خودش برخورد نکند. یعنی اگر مکان سر بعدی مار، روی بدن کنونی اش (به جز دم (مگر در یک سری حالت های خاص مثل حالتی که طول مار ۲ واحد است)) قرار داشته باشد، این استیت به عنوان استیت جدید قابل قبول نیست و رد می شود. سپس بررسی می شود که استیت کنونی مکان سر مار روی یک دانه (با امتیاز حداقل ۱) هست یا نه. در صورتی که باشد:

- برای فرزندان این استیت (حالت های بعدی) امتیاز آن دانه یک واحد کاهش می یابد.
- طول مار افزایش می یابد. به این صورت که دم مار از جایش تکان نمی خورد و خانه ی آخری که بدن مار در آن قرار دارد، در استیت بعدی نیز جزو بدن مار است. در حالتی که روی دانه نباشیم، آخرین خانه ی بدن مار در استیت بعدی دیگر جزو مار نیست.

با این دو کار، استیت بعدی را ساخته ایم (امتیاز جدید دانه ها و مکان بدن مار). حال بررسی می کنیم که قبلا در طول فرآیند به این استیت نرسیده باشیم. برای این کار از یک Set به نام set_of_states استفاده می کنیم که سرچ در این ساختمان داده با اردر زمانی O(1) انجام می گیرد. برای وارد کردن استیت ها به اینset، نیاز به hash کردن اطلاعات هر استیت داریم. برای همین لیست های موجود در این کلاس را به tuple تبدیل می کنیم که قابل hash کردن باشند و سپس حاصل hash را چک میکنیم که درset هست یا نه و به این روش، تکراری بودن استیت را چک میکنیم. نحوه برخورد با استیت های تکراری در ها الگوریتم متفاوت است. در نهایت هم برای هر استیت ساخته شده چک میکنیم که یک goal state هست یا نه. اگر بود که برنامه

را در همان لحظه متوقف می کنیم و راه حل رسیدن به آن استیت را به عنوان جواب اعلام می کنیم. در غیر این صورت این فرآیند را ادامه می دهیم. حال به طور خاص هر الگوریتم سرچ را بررسی میکنیم:

۱- روش BFS:

در این روش استیت های جدید به وجود آمده را به طور کامل وارد یک صف می کنیم. در این صف که با قانون FIFO کار می کند درخت لایه لایه پیش می رود و در این نوع search، استیت ها و گره های درخت، به ترتیب عمق و فاصله از حالت اولیه بررسی می شوند. در این روش استیت های تکراری بررسی نمی شوند چون به هر استیت در کمترین عمقی که ممکن باشد رسیده ایم و نیازی به بررسی دوباره آن نداریم. پس در صورت برخورد با استیت تکراری آن را نادیده می گیریم. در زیر، تصویر نمونه اجرای تست ۱ با این الگوریتم آورده شده است که در ۰,۰۴۸ ثانیه پایان یافته است و ۴۳۳۶ استیت مجزا را در مسیر خود دیده است.

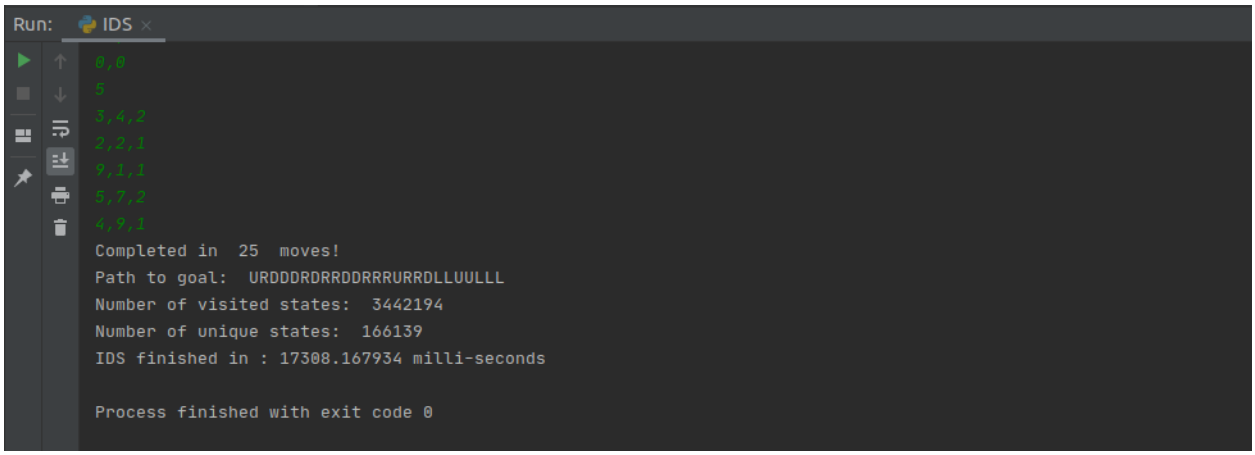
۲- روش IDS:

در این روش یک استک تعریف می شود که Node ها فرزندان خود را وارد آن می کنند ولی فقط اجازه داریم که در هر شاخه از این درخت، تا عمق depth_limit پیش برویم که این عمق از ۱ شروع می شود و زیاد می‌شود. ابتدا تا عمق ۱ همه شاخه ها را می رویم و اگر جواب پیدا نشد، عمق را یکی زیاد می کنیم تا به عمق کم عمق ترین جواب برسیم. در این مرحله کوتاه ترین مسیر یافت می شود و به محض پیدا شدن اولین جواب الگوریتم پایان می یابد. از آنجایی که Node های جدید در Stack ریخته می شوند و در هر مرحله بالاترین گره موجود در Stack را expand می کنیم، هر شاخه تا انتها (تا عمق depth_limit در آن لحظه) پیمایش می کنیم. در تابع run_ids این الگوریتم آغاز می شود و هر Node برایش یک عمق تعریف می شود و به واسطه

آن، Node ها به شرطی که عمقشان از depth_limit کمتر باشد اجازه expand شدن دارندو در غیر این صورت چون استیت نهایی نبوده اند از استک خارج می شوند. هر بار که استک خالی شد یعنی تا آن عمق هیچ جوابی موجود نبوده و depth_limit یک واحد افزایش می یابد. در کنار آن، تمام ساختمان های داده که برای گره ها و استیت ها استفاده شده نیز خالی می شوند تا دوباره پر شوند. (مثل set_of_states، ids_stack، nodes_list و ...)

* در این روش یک dictionary نیز استفاده شده است که نشان می دهد هر استیتی که قبلا مشاهده شده در چه عمقی دیده شده است. به این صورت که hash_key مربوط به هر استیت به عنوان dictionary key استفاده شده است و مقدار آن عمق آن استیت است. اگر دوباره به استیتی برسیم که قبلا دیده ایم، اگر با عمق کمتری به آن رسیده باشیم آن را جایگزین قبلی می کنیم و دوباره بررسی می کنیم، در غیر این صورت آن را نادیده می گیریم چون loop ایجاد می کند.

در این روش برای کلاس Stack تعدادی تابع مانند push، pop، size و top نیز تعریف شده است. در زیر حاصل اجرای تست سوم با این الگوریتم مشاهده می شود. مار با ۲۵ حرکت به هدف رسیده و این فرآیند ۱۷ ثانیه طول کشیده است و در مجموع حدود ۱۶۶ هزار استیت مختلف مشاهده شده است.



روش IDS نسبت به BFS روش کندتری است چون خیلی از Node ها را چند بار می بینیم. مزیت آن نسبت به BFS استفاده از حافظه ای به مراتب کوچک تر است ولی به علت انجام تعداد زیادی کار تکراری، از لحاظ زمانی به صرفه نیست.

۳- A*:

این قسمت از دو تابع تخمین استفاده می شود:

I) h(n) = مجموع امتیاز باقی مانده ی دانه ها

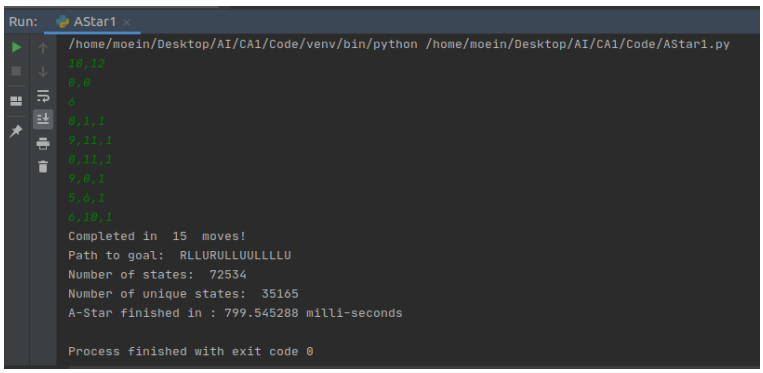
II) h(n) = تعداد دانه های باقیمانده

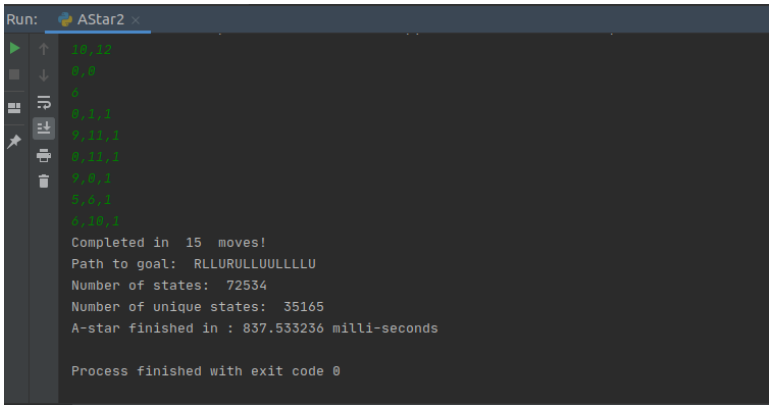
در هر دوی این حالت ها، تابع g(n) برابر هزینه ی طی کردن مسیر تا به اینجاست. یعنی تعداد حرکت هایی ک مار انجام داده تا به استیت n رسیده است. در این دو روش، هر استیت که به آن می رسیم یک تابع f(n) دارد که مجموع دو تابع g(n) و h(n) برای آن استیت است. Node متعلق به این استیت را وارد صفی به نام a_star_queue می کنیم ولی به ابتدای آن اضافه نمی کنیم. آن را در جایی از صف قرار می دهیم که Node های صف، به ترتیب نزولیِ مقدار h(n) در صف قرار بگیرند. در هر مرحله، Node ای از صف را بررسی می کنیم که مقدار f(n) کمتری داشته باشد. یعنی عنصر انتهایی صف.

مزیت این روش نسبت به دو روش قبلی این است که با بررسی استیت های کمتری به جواب می رسیم. چون تابع h(n) در اصل ما را به جواب نهایی و بهینه راهنمایی می کند. به همین دلیل از نسبت به دو روش قبلی در زمان کمتری نیز به جواب می رسد. به طور مثال این روش برای تست سوم حدود ۱۲۵ هزار استیت مجزا می بیند که در مقایسه با ۱۶۶ هزار استیت مجزای روش IDS برای این تست عدد کمتری است. این روش نوعی جستجوی آگاهانه است و به همین دلیل از لحاظ زمانی نسبت به دو روش قبلی برتری دارد.

* هر دو تابع heuristic گفته شده در بالا admissible هستند. چون ما در تابع اول فرض کرده ایم که می توانیم در چند حرکت متوالی تمام امتیاز های باقی مانده از دانه ها را بگیریم و در این بین هیچ مسیری وجود ندارد. واضح است که این تابع تخمین از هزینه ی واقعی ما کمتر است و در نتیجه دارای خاصیت admissibility است.

* هر دو تابع heuristic گفته شده consistent نیز هستند. چون در مسیر رفتن مار از یک خانه به خانه کناری اش، یک واحد هزینه می دهیم (g(n) یک واحد بیشتر می شود) ولی h(n) موجود در این دو استیت، یا تغییری نمی کند (در صورتی که مار در این حرکت دانه نخورده باشد) یا حداکثر یک واحد کمتر می شود (در صورتی که یک دانه خورده باشد) و چون بیش از این نمی‌تواند کم شود، نامساوی مربوط به consistency برقرار است و این دوتابع consistent نیز هستند.





همانطورکه در تصاویر مشخص است، این دو تابع تخمین زمان اجرا را کاهش می دهند و عملکرد این دوتابع تقریبا نزدیک به هم است.

۴- Weighted A*:

این روش نسبت به روش A* هیچ تفاوتی ندارد. فقط یک ضریب بزرگتر از ۱ در h(n) ضرب می شود و تاثیر آن را در محاسبه مقدار f(n) بیشتر می کند. این ضریب، سرعت الگوریتم را بسیار سریع تر می کند. در قسمت اول ضریب 1.95 در نظر گرفته شده است و در قسمت دوم، ضریب 4.8 استفاده شده است. در حالت دوم امکان رخ دادن اشتباه در رسیدن به هزینه بهینه نیز وجود دارد چون ضریب تخمین نسبت به ضریب هزینه واقعی خیلی بیشتر است. در مثال ها بیشتر به این احتمال پرداخته می شود.

در جدول های زیر نمایی از حاصل اجرای الگوریتم های مختلف روی تست های مختلف مشخص است.

جدول زمان اجرای تست ها:

Test1:

	فاصله جواب	مسیر جواب	استیت های دیده شده	استیت های مجزا دیده شده	زمان اجرا (ثانیه)
BFS	12	DLLUULULLLUU	8683	4336	0.046
IDS	12	LDUULULLUULL	32214	4360	0.189
A* (h1)	12	DLLUULULLLUU	4845	2948	0.038
A* (h2)	12	DLLUULULLLUU	7007	3550	0.043
Weighted A* (a = 1.95)	12	DLLUULULLLUU	989	799	0.02
Weighted A* (a = 4.8)	12	DLLUULULLLUU	259	198	0.006

Test2:

	فاصله جواب	مسیر جواب	استیت های دیده شده	استیت های مجزا دیده شده	زمان اجرا (ثانیه)
BFS	15	RLLURULLUULLLLU	102220	46226	0.994
IDS	15	URDLLUUUULULLL	537397	31458	2.787
A* (h1)	15	RLLURULLUULLLLU	72534	35165	0.852
A* (h2)	15	RLLURULLUULLLLU	72534	35165	0.841
Weighted A* (a = 1.95)	15	RLLURULLUULLLLU	30230	17446	0.712
Weighted A* (a = 4.8)	15	RLLURULLUULLLLU	970	875	0.026

	فاصله جواب	مسیر جواب	استیت های دیده شده	استیت های مجزا دیده شده	زمان اجرا (ثانیه)
BFS	25	RURDDDRRDRRRDDRRULLDLLLUU	449076	200850	10.54
IDS	25	URDDDRDRRDDRRRURRDLLUULLL	3442194	166139	17.3
A* (h1)	25	RURDDDRRDRRRDDRRULLDLLLUU	259236	125459	9.71
A* (h2)	25	RURDDDRRDRRRDDRRULLDLLLUU	313309	145936	9.03
Weighted A* (a = 1.95)	25	RURDDDRRDRRRDDRRULLDLLLUU	98457	53458	8.31
Weighted A* (a = 4.8)	**26**	RURDDDRRDRDLURRRDDRRULULDD	1558	1236	0.054

* همانطور که در جدول آخر برای تست سوم مشخص است، در حالتی که الگوریتم weighted A* را با a = 4.9 (نزدیک به ۵) اجرا می کنیم چون ضریب بسیار زیادی به هزینه تخمینی h(n) نسبت به هزینه واقعی g(n) می دهیم ممکن است این تخمین از حالت admissibility بیرون بیاید و دیگر جواب بهینه را ندهد. در عوض در زمان بسیار بسیار کوتاه تری جواب را به ما می دهد ولی در عوض طول مسیر را یک واحد افزایش داده و مسیر طولانی تر را به عنوان پاسخ نهایی معرفی می کند.