گزارش پروژه ۱ درس هوش مصنوعی

آتیه آرمین – ۸۱۰۱۹۷۶۴۸

- **مقدمه**

این پروژه مشابه بازی snake است. ما در این پروژه قصد داریم در یک صفحه دو بعدی که مختصات دانه ها و امتیاز دانه ها در آن مشخص است، ماری به طول ۱ و با مختصات اولیه‌ی مشخص را به گونه‌ای حرکت دهیم که تمام دانه ها را بخورد.
مار با هر بار خوردن دانه ها به اندازه ۱ امتیاز از امتیاز دانه ها کم میکند و طولش ۱ عدد افزایش پیدا میکند.
حال ما قصد دارین با پیاده سازی این مساله به صورت یک گراف (درخت) و با کمک الگوریتم های جست و جوی bfs ، ids و A^* نشان دهیم که مار از چه مسیری باید برود که تمام دانه ها را بخورد.

- **پیاده سازی مساله**

برای پیاده سازی این مساله ما نیاز داریم تا state های مساله را مشخص کنیم. در اینجا ما state ها را به صورت object هایی در نظر گرفته‌ایم که مختصات بدن مار، مختصات دانه ها و امتیاز هر دانه را در خود نگه میدارد. پس state اولیه ما همان ورودی ای خواهد بود که از کاربر می گیریم. و به همین شکل میتوان goal state را state ای تعریف کرد که تمام دانه های آن خرده شده و دانه‌ای در آن نمانده است.action ما در این مساله حرکت مار است. مار می تواند در هر مرحله به یکی از جهات راست، چپ، بالا و پایین حرکت کند و هر حرکت او یک action به حساب می آید. حال میتوانیم هر node گراف یا درخت را object ای تعریف کنیم که state آن مرحله را در خود نگه می دارد و علاوه بر آن id پدرش را نیز نگه میدارد (وقتی بخواهیم جواب نهایی را پیدا کنیم از id پدر استفاده میکنیم.جزییات پیاده سازی را در هر روش توضیح میدهیم.

- **الگوریتم های جست و جو**

← **الگوریتم bfs**

برای انجام این الگوریتم ابتدا داخل تابع bfs می رویم. در این تابع یک صف تعریف کرده‌ایم که ابتدا تنها مقدار آن nodeاولیه که شامل initial state است، در آن ریخته شده است. حال تا زمانی که دیگر چیزی در این صف باقی نماند، اولین node درون این صف را pop میکنیم و برای این node عملیات پیدا کردن بچه هایش را انجام میدهیم. اگر بچه‌ای پیدا شد که goal state بود، همان موقع آن را بر میگردانیم و توسط تابعfind_sol مسیر را پیدا میکنیم به طوری که از هر node به node پدر میرویم و nodeها را در یک لیست به منظور لیست جواب میریزیم.
برای پیدا کردن بچه های یک node تابع find_children را تعریف کرده‌ایم که برای ۴ جهت راست، چپ، بالا و پایین بچه های این node را پیدا کند. پس از پیدا کردن هر بچه آن را در آخر لیست تمامی node ها و در آخر صف اضافه میکنیم و چک میکنیم که اگرstate این بچه=goal state بود، همان جا اعلام کند که الگوریتم تمام شده و به هدف رسیده‌ایم(مقدار true را برگرداند).
در پیدا کردن بچه ها ممکن است شرایطی پیش بیاید که nodeای ساخته نشود(در ادامه توضیح داده شده‌است) در این صورت node بدون مقدار(none) وارد هیچ لیست و صفی نمیشود.

حال برای ساختن node بچه ها به تابعcreate_node میرویم. در این تابع ابتدا تابع snake_move صدا زده میشود و stateجدید گرفته می‌شود. سپس باید چک کنیم که آیا state ساخته شده تکراری است یا خیر. برای این کار، تابع hash و eq را برای کلاس state باز نویسی میکنیم به این صورت که اگر ()hash بر روی یک state اعمال شد باید یک tupleاز مختصات کل بدن مار و همینطور امتیاز دانه ها را hash کند.برای تابع eqهم در نظر میگیریم که اگر چیزی که قرار است با stateمقایسه شود مقدارnone داشته باشد باید false برگردانده شود و در غیر این صورت باید چک کنیم که آیا مختصات کل بدن مار، امتیاز دانه ها و دانه ها همگی با مختصات کل بدن مار، امتیاز دانه ها و دانه های objectدیگر برابر است یا خیر. اگر بودtrueبرگردانده شود.
حال در تابع check_duplication از یک set کمک میگریم و چک میکنیم که آیا hash شده‌ی stateمورد نظر در این setهست یا نه. اگر باشد، این stateتکراری است. حال در تابع create_node پس از چک کردن تکراری نبودن state و چک کردن none نبودن state ساخته شده،node جدیدی برای این state می سازیم.برای ساختnode باید state ساخته شده،id که در واقع اندازه‌ی لیست تمامیnode ها است، id پدر و یک direction برای اینکه بدانیم این node با چه حرکتی ساخته شده است(برای چاپ کردن جواب) را به node بدهیم.
در تابعsnake_move حرکتی که قرار است مار برود را بررسی میکنیم. ابتدا باید چک کنیم که اگر سر مار در یک سمت جدول و در انتهای آن ستون یا آن ردیف است و میخواهد باز هم به همان سمت برود، باید سر مار از طرف دیگر جدول وارد شود. پس از انجام این شرط باید ببینیم که آیا مار در مرحله قبلی دانه‌ای خورده است یا نه. اگر خورده باشد متغیر eaten در آن stateمقدار ۱ را دارد. حال اگر مار دانه خورده باشد باید تمام بدن مار ثابت بماند و فقط یک خانه به سر مار اضافه شود که این کار را با insert انجام میدهیم. اگر نخورده باشد تنها دم مار را pop میکنیم و باز یک خانه به سر مار اضافه میکنیم. اگر با انجام این حرکات سر مار روی خانه‌ای برود که در آن خانه دانه‌ای وجود دارد، مار آن دانه را با تابع eat میخورد. این تابع اگر امتیاز دانه ۲ باشد، امتیاز آن را ۱ می کند و اگر امتیازش ۱ باشد، آن دانه را از لیست دانه ها حذف میکند. مساله ای که در اینجا پیش می آید این است که مار نباید با بدن خودش برخورد کند. برای مدیریت این شرایط به طور کلی میتوان گفت که سر جدید مار داخل لیست مختصات جدید مار قبل از اضافه کردن سر مار به آن، نباشد. ولی در حالتی که طول مار ۲ باشد و هر دو خانه بدن مار در یک راستا باشند و ابعاد جدول هم در همان راستا ۲ باشد، اگر سر جدید مار مختصاتش با دم کنونی مار یکی باشد، مار با خودش برخورد میکند، در غیر این صورت به این معنا است که مار از یک طرف جدول خارج شده و میخواهد از طرف دیگر وارد شود که این در صورتی که دانه نخورده باشد، مشکلی ایجاد نمیکند.
حال این تابع state جدیدی که با مختصات جدید مار و با امتیاز و مقدار دانه های جدید است را برمیگرداند.
در آخر توسط تابع print_sol مقادیر direction جواب را چاپ میکنیم.

← **الگوریتم** ids

در این قسمت تمامی مراحل ساخت node و state ها مانند bfs است تنها با چند تفاوت.
برای انجام الگوریتم ids باید یک عمق مشخص را به الگوریتم dls بدهیم که درواقع همان الگوریتم dfs است با این تفاوت که باید تا عمق مشخص داده شده پیش برود. پس از انجام dls اگر goal_state را پیدا نکردیم، عمق را یکی افزایش می دهیم و dictionary ای که برای state های دیده شده داشتیم را خالی میکنیم و دوباره الگوریتم dls را با عمق جدید اجرا میکنیم.
برای الگوریتم dls از یک stack استفاده میکنیم به این منظور که سیستمش FIFO است. سپس تا زمانی که این stack خالی شود یک node از top آن یا درواقع ته لیست برمیداریم، اگر عمق آن node با عمقی که از تابع ids گرفته ایم برابر بود، ادامه میدهیم و سراغ node های دیگر درون stack می رویم. اگر نه باید بچه های آن node را پیدا کنیم.
برای پیدا کردن بچه های node از همان تابع find_children در الگوریتم bfs استفاده میکنیم. فقط به عنوان ورودی به آن عمق node پدر را به آن میدهیم تا آن هم این عمق را به تابع create_node بدهد تا در آن هنگام ساخت node جدید، عمق آن با عمق node پدر محاسبه شود.
تنها تابع دیگری که در اینجا تغییر کرده است تابع check_duplication است. در این الگوریتم ما مجموعه state های دیده شده را به عنوان یک dictionay در نظر گرفته ایم زیرا در این الگوریتم ممکن است با یک state تکراری مواجه شویم اما عمق آن بهتر از عمق state ای باشد که قبلا دیده‌ایم، پس در این dictionary، hash شده‌ی state به عنوان key و عمق آن به عنوان value نگه داری میشود. در تابع check_duplication عمق این state را با value، hash شده‌ی این state مقایسه میکنیم. اگر بیشتر بود پس none بر میگردانیم ولی اگر کمتر بود، مقدار value را برابر با این عمق جدید که عمق بهتری است قرار می‌دهیم.

← **الگوریتم** A^*

این الگوریتم بسیار شبیه به الگوریتم bfs است و تمام مراحل مانند آن انجام میشود تنها تفاوت در اولویت صف است. صف در این الگوریتم بر مبنای $g(n) + h(n)$ است. در اینجا ۲ heuristic در نظر گرفته ایم. یکی تعداد دانه ها است که این حدس consistant است و دیگری تعداد دانه هایی که امتیاز آن ها ۱ است. $g(n)$ هم مسیر طی شده توسط مار است.
براس اثبات اینکه heuristic اول consistant است میتوان گفت که همانطور که در اسلاید های درس گفته شده میتوانیم رابطه‌ی هزینه‌ی حرکت بین ۲ node ، $h(n)$ و $h(n')$ را به صورت یک مثلث در نظر گرفت. هزینه حرکت مار بین ۲ node ۱ است. حال یا مار دانه ای خرده است در طی این حرکت و یک عدد از دانه ها کم شده است یا دانه‌ای نخرده و مقدار $h(n)$ و $h(n')$ با هم برابر است. در هر دوی این حالات رابطه‌ی مثلثی بین این ۳ برقرار است.
از طرفی هر دوی این heuristic ها admissible هستند زیرا قطعا از هزینه واقعی مار بیشتر هستند.
در اینجا ما عمق هر node را در آن نگه میداریم که نشان دهنده طول مسیری است که مار تا آن state طی کرده است. پس هنگام اضافه کردن بچه ها به صف، صف را طی میکنیم اولین جایی

که g(n) + h(n) عنصر داخل صف کمتر از node g(n) + h(n)بچه باشد، node را اضافه میکنیم.
در فایل این الگوریتم میتوانید به جای تابعheuristic ، تابع consistant_heuristic را در تابع add_child قرار دهید.

← **الگوریتم**weighted A*

این الگوریتم کاملا مشابه الگوریتم A* است تنها یک ضریب در heuristic ضرب می شود. در صورت مساله از ما خواسته شده یک ضریب نزدیک ۲ و یک ضریب کمتر از ۵ در نظر بگیریم.
برای این کار دو مقدار a و bدر نظر گرفته شده است که در توابع heuristicl ها در مقدارشان ضرب شده اند.

← **نتیجه گیری**

به طور کلی الگوریتم ids چون تا آخر یک شاخه را میبیند بسیار بیشتر از بقیه الگوریتم ها طول میکشد.
الگوریتم A* الگوریتم سریعی است زیرا یک حدس تقریبی را وارد محاسباتمان میکنیم.
الگوریتم bfs به طور کلی خوب عمل میکند و سریع است زیرا تا ته یک شاخه را نمی رود و سطح به سطح بررسی میکند.
الگوریتم weighted A* به طور کلی سریع است ولی حتما optimal نیست زیرا ممکن است حالتی پیش بیاید که هزینه حدس زده شده بیشتر از هزینه واقعی باشد. مثلا در تست ۱ این اتفاق افتاده است که ۱۴ حرکت جواب است.

- **خروجی**

← **تست ۱**

	فاصله جواب	**مسیر جواب**	**تعداد state دیده شده**	**تعداد state جزای دیده شده**	**زمان اجرا به ثانیه**
BFS	12	LDRRDDRRDRDD	6468	2847	0.04678
IDS	12	DLDDRRRDRDDR	29620	16597	0.20721
A*(consistant_heuristic)	12	LDLUULULLLUU	5729	2602	0.05837
A*(just admissible heuristic)	12	LDLUULULLLUU	6239	2603	0.09525
Weighted A* (a = 4 and consistant heuristic)	14	RUURRDRDDRRRRR	3454	1553	0.03972
Weighted A* (b = 1.8 and consistant heuristic)	12	LDLUULULLLUU	4382	2147	0.08433

← تست ۲

	فاصله جواب	مسیر جواب	تعداد state دیده شده	تعداد state جزای دیده شده	زمان اجرا به ثانیه
BFS	15	RLLURULLUULLLLU	37161	15804	0.23603
IDS	15	URDLLUUUULULLLL	263006	127184	2.27527
A*(consistant_heuristic)	15	RLLURULLUULLLLU	32485	14354	0.31394
A*(just admissible heuristic)	15	RLLURULLUULLLLU	32485	14354	0.53017
Weighted A* (a = 4 and consistant heuristic)	15	RLLURULLUULLLLU	19412	10187	0.46750
Weighted A* (b = 1.8 and consistant heuristic)	15	RLLURULLUULLLLU	19412	10187	0.47510

← تست ۳

	فاصله جواب	مسیر جواب	تعداد state دیده شده	تعداد state جزای دیده شده	زمان اجرا به ثانیه
BFS	25	RURDDDRRDRRRDDRRULLDLLLUU	165361	66440	1.29523
IDS	25	URDDDRDRRDDRRRURRDLLLUULL	2018402	880099	16.9172
A*(consistant_heuristic)	25	RURDDDRRDRRRDDRRULLDLLLUU	145663	60229	2.38575
A*(just admissible heuristic)	25	RURDDDRRDRRRDDRRULLDLLLUU	159622	64877	5.30953
Weighted A* (a = 4 and consistant heuristic)	25	RURDDDRRDRRRDDRRULLDLLLUU	40601	20287	3.40256
Weighted A* (b = 1.8 and consistant heuristic)	25	RURDDDRRDRRRDDRRULLDLLLUU	105632	47415	6.48882