به نام خدا

گزارش پروژه اول کاربرد های هوش مصنوعی

سید مهدی میرفندرسکی ۹۷۲۳۰۹۳

ستایش ثانوی ۹۶۲۸۰۲۴

۱.نحوه‌ی مدل سازی:

برای تبدیل جدول داده شده (میز) به گراف ، تست داده شده را از فایل خط به خط میخوانیم و بعد از خواندن اولین خط که تعداد سطر و ستون ها را میدهد به همان تعداد سطر و ستون خانه ها را به آرایه grid اضافه میکنیم و در همین قسمت هم باید برای هر خانه اگر ربات یا کره یا هدف بود مکان آن ها را در متغییر هایی نگه داریم که البته چون ممکن است چندین کره یا هدف داشته باشیم آن ها را در آرایه ذخیره میکنیم.

مقادیر " مختصات هر خانه، پدر هر گره، مقدار هر خانه " کلاس گره (node) به صورت ساختمان داده گره نگهداری میشوند. پس گره ها نشان دهنده هر خانه از میز هستند و یال ها برای حرکت کردن نشان دهنده‌ی مسیر و ارتباط همسایگی و پدر فرزندی خانه های میز هستند.

در این پیاده سازی ما مسئله را به دو قسمت کلی برای مسیر یابی تقسیم کردیم : یکی پیدا کردن مسیر مناسب از کره ها به هدف ها و دومی پیدا کردن مسیر از ربات تا کره

برای مدل سازی این مسئله ابتدا الگوریتم ها را برای پیدا کردن مسیر از کره به هدف در نظر گرفتیم . برای پیدا کردن مسیر از کره به هدف ، چون ربات در خانه‌ی مجاور کره قرار دارد و فقط میتواند آن را هل بدهد نباید جوری مسیر یابی کرد که کره به چهار گوشه‌ی میز برود به عبارتی با تعمیم این موضوع میتوان گفت که کره نباید به خانه هایی برود که دو سمت مجاور بسته دارد ("بالا و راست " یا " بالا و چپ" یا "پایین و راست" یا "پایین و چپ" مانع باشد یا از محدوده میز خارج شود) چون در این صورت ربات از هیچ جهتی نمیتواند کره را هل بدهد. پس در پیدا کردن مسیر باید خانه هایی که دو طرف بسته دارند غیر مجاز شوند.

همچنین اگر چند کره و چند هدف داشته باشیم برای یک هدف آن کره ای را انتخاب میکنیم که طبق فاصله‌ی منهتن ($|X_1-X_2|+|Y_1-Y_2|$) به هدف نزدیک تر باشد و باید توجه کنیم که با انتخاب یک کره برای رساندن به یک هدف تمام کره های دیگر مانع در نظر گرفته میشوند و ربات نمیتواند آن را هم هل بدهد.

بعد از اینکه مسیر مناسب برای کره تا هدف را بدست آوردیم باید مسیری از ربات به کره بیابیم. برای این کار ابتدا مسیری که از کره تا هدف داشتیم را در نظر میگیریم و بررسی می‌کنیم که اگر کره در اولین حرکت به سمت هدف باید به بالا برود ، ربات باید به خانه‌ی پایین کره برود و انگار که هدف ربات رفتن به آن خانه‌ی مجاور کره باشد (اگر کره در اولین حرکت بخواهد به راست برود باید ربات را به سمت چپ آن ببریم و به همین منوال هر جهتی که کره بخواهد حرکت کند ربات باید در خانه‌ی مخالف آن جهت قرار گیرد) پس همچنین باید توجه داشته باشیم که آن خانه‌ی مجاور کره که قرار است ربات به آنجا برود خالی باشد و مانعی نداشته باشد. در صورتی که آن خانه دارای مانع بود باید همسایه دیگری برای حرکت انتخاب شود که امکان حرکت داشته باشد.

بعد از بررسی گره اول حرکت کره، همسایه‌ی سمت مخالف آن را به عنوان هدف برای ربات در نظر گرفته و مسیر یابی را از مکان ربات تا آن گره مجاور کره انجام میدهیم. سپس با هربار تغییر جهت حرکت کره (حرکت کره را قبلا بدست آوردیم و همه‌ی جهت های آن را داریم) ربات باید به خانه‌ی سمت مخالف آن جهت حرکت برود. پس به عبارت دیگر میتوان گفت بعد از اینکه ربات به خانه‌ی مجاور کره رفت حرکت های آن با کره یکی میشود و هر جا کره تغییر جهت داد ربات باید در یک مسیر یابی جدید از جایی که الان قرار گرفته تا همسایه سمت مخالف تغییر جهت کره مسیر یابی کند و این مسیر در کل مسیر نهایی را میسازد.

۲. تابع شهودی انتخاب شده و بررسی قابل قبول بودن آن:

تابع شهودی انتخاب شده در این پروژه تابع فاصله منهتن است که هزینه‌ی کوتاه ترین فاصله از مبدا تا مقصد را بدون در نظر گرفتن موانع در نظر میگیرد و صرف نظر از تمام منابع $|X_1-X_2|+|Y_1-Y_2|$ را حساب میکند که ۱ همان مبدا و ۲ مقصد است.

چون در این تابع ما موانع را در نظر نمیگیریم و مجاز به حرکت اریب هم نیستیم پس هزینه‌ی کل کمتر و یا برابر با حالت واقعی خواهد بود پس میتوان گفت که تابع قابل قبول یا admissible است.

۳. توضیح کلی توابع و کلاسهای تعریف شده از کد:

ابتدا یک کلاس Node تعریف کردیم تا به عنوان ساختمان داده ای برای نگهداری مکان خود گره، پدر گره و مکان هدف آن گره استفاده کنیم و البته برای الگوریتم A^* تابع هیوریستیک هم همینجا تعریف و برای هر گره محاسبه و نگهداری میشود.

در کلاس Graph در تابع __init__ ابتدا یک فایل را برای دریافت ورودی خط به خط می‌خوانیم و بعد از خواندن اولین سطر که مربوط به تعداد سطر و ستون است آرایه grid را به صورت یک ماتریس میسازیم و بعد مکان های مربوط به b را در start list و مکان های مربوط به p را نیز در goal list میریزیم و چون یک رباط داریم مکان آن را نیز در rx,ry نگه داشتیم.

در تابع neighbors برای تولید همسایه های مجاز هر گره در صورتی که num == 1 باشد یعنی مسیر از کره تا هدف مورد بررسی است پس همسابه هایی که تولید میکند دارای این محدودیت هستند که از محدوده‌ی میز خارج نشوند یا به عبارتی مقدار x,y هر گره در مسیر از x,y کل ماتریس بیشتر نشود که این محدودیت برای حالت num != 1 هم برقرار است و محدودیت اضافه تری که num ==1 در مسیر کره دارد این است که باید در تابع validation هم بررسی شده و مجاز شمرده شود تا بتواند جزو همسایه ها تلقی شود.

در تابع validation هر گره بررسی میشود که در دو خانه‌ی مجاور بسته نباشد ("بالا و راست " یا " بالا و چپ" یا "پایین و راست" یا "پایین و چپ" مانع نباشد یا از محدوده میز خارج نشود) چون در این صورت دیگر قادر به حرکت نیست.

در تابع first neighbor برای اولین همسایه‌ی کره بررسی میشود چون اگر اولین بار بخواهد به سمتی حرکت کند سمت مخالف باید برای ربات خالی باشد ، در این تابع با صدا زدن تابع neighbor آن محدودیت های گفته شده را خواهیم داشت و علاوه بر آن چون این تابع فقط برای کره است پس اگر num != 1 باشد همان تابع neighbor را برمیگرداند ولی اگر

برای کره باشد در این صورت برای همسایه های تولید شده باید بررسی شود که دو همسایه موجود باشد که yها(همسایه های بالا و پایین) یا xهای(همسایه های چپ و راست) یکسان دارند که با این کار بررسی میشود که اگر همسایه راست تولید شده فقط در صورتی جزو مسیر حساب میشود که همسایه چپ نیز جزو همسایه های تولید شده باشد تا ربات بتواند به آن برود و کره را هل بدهد و یکی از این همسایه های مجاز به عنوان اولین همسایه کره استفاده خواهد شد.

در تابع goal test اگر برای کره باشد بررسی میشود که آیا به p که رسیده یا نه و برای ربات هم بررسی میشود که به هدف نهایی رسیده یا خیر.

در تابع finding SG فقط برای کره چون ممکن است چندتا کره و هدف داشته باشیم باید یکی از کره ها را برای رسیدن به یکی از هدف ها انتخاب کنیم ، برای این کار ابتدا نقاط x,y شروع (کره ها) را تعیین میکنیم سپس مقدار min manhatan را بینهایت قرار میدهیم و در یک حلقه‌ی for تو در تو ابتدا فواصل تمام هدف ها از یک کره را بررسی کرده و در min manhatan میریزیم و برای تمام کره ها این کار را ادامه میدهیم تا نهایتا کمترین فاصله منهتن را بدست آورده و همان را به عنوان شروع کره و هدف نهایی در نظر بگیریم ، بعد از این کار تمام کره های دیگر را به عنوان x در ماتریس در میگذاریم چون نمیتوانیم دو کره را هل بدهیم یا اینکه از روی آن رد شویم.

در تابع routing کل مسیر طی شده را دریافت میکنیم (به صورت مختصاتی) و با توجه به آن، جهت حرکت را تعیین میکنیم(از اختلاف یک گره با گره قبلی متوجه میشویم که چطور حرکت کرده)

در تابع full که کلیت اجرای کد است ابتدا با صدا زدن تابع الگوریتم مورد نظر(توابع الگوریتم ها در ادامه توضیح داده میشوند.) با ورودی ۱ مسیر یابی برای کره انجام میشود و مسیر پیدا شده را در آرایه butter میریزد (فقط در مورد الگوریتم AStar این خروجی هزینه‌ی مسیر پیدا شده را نیز در متغیر cost میریزد) بعد برای بررسی تغییرات در جهت مسیر کره تا زمانی که طول آرایه butter صفر نشده بررسی میکنیم اگر طول آرایه butter یک نشده بود برای مختصات های داده شده در butter جهت ها را از طریق صدا زدن تابع routing انجام میدهیم و آن را در آرایه finalResult میریزیم و مکان اولیه ربات را نیز در متغیر های current نگه میداریم برای بررسی تغییر جهت دادن یک متغیر ex در نظر گرفتیم که جهت حرکت قبلی را نگه داشته و اگر با جهت فعلی یکی بود مختصات ربات را در آن جهت حرکت میدهیم تا محل فعلی ربات به روز شود و در صورتی که ex با جهت حرکت قبلی یکسان نبود یعنی دچار تغییر جهت در مسیر کره شدیم و در این صورت محل فعلی ربات را به عنوان مبدا(شروع) و خانه‌ی مجاور خانه‌ای که الان کره در آن است (با توجه به جهت حرکت بعدی) را به عنوان هدف جدید میگریم و بار دیگر الگوریتم مورد نظر را با عدد ۲ برای مسیر یابی ربات انجام میدهیم و مسیر داده شده را پس از جهت یابی با تابع routing خانه به خانه در یک حلقه به آرایه finalResult اضافه میکنیم و نهایتا آخرین جهت را در متغیر edge نگه میداریم تا مکان کره و ربات را بر اساس آن به روز کنیم و نهایتا مکان های جدی کره و ربات را در آرایه grid به روز میکنیم و تابع finalResult را که مسیر نهایی است چاپ میکنیم ، اگر در همان ابتدای کار طول آرایه مسیر یابی شده برای کره صفر بود یعنی ربات اصلا نمیتواند کره را هل دهد پس جوابی ندارد.

در انتها نیز با صدا زدن plotting شکل خروجی در عکسی در نشان داده میشود.

تابع های الگوریتم ها :

تابع AStar :

ابتدا با صدا زدن تابع findingSG با توجه به کمترین فاصله یک کره و یک هدف انتخاب میشود . شروع را گره ای که در startx,starty ذخیره شده در نظر میگیریم و آن را در آرایه frontier ذخیره میکنیم و تا زمانی که طول frontier صفر

نشده در حلقه‌ی while اعمالی را انجام میدهیم : در ابتدا گره ای که در خانه اول آرایه‌ی frontier است را در expandNode میریزیم و برای تمام عناصر موجود در frontier بررسی میکنیم اگر هزینه‌ی آن عنصر طبق هیوریستیک و f ای که تعریف کردیم از expandNode کمتر است و با این کار کم هزینه ترین عنصر برای expandNode انتخاب میشود و باید تمام عناصر دیگر که هزینه‌ی بیشتری داشتند از آرایه frontier ، pop شوند .

بعد از این باید برای هر گره که گسترش میدهیم هدف بودن آن را با تابع goalTest انجام دهیم که اگر به هدف رسیده بودیم تمام گره های گسترش داده شده را از طریق رابطه‌ی پدر فرزندی که گره ها داشتند برمیگردانیم و این گره ها در واقع مسیر ما را مشخص میکنند . اگر هم گره ها هنوز به هدف نرسیده بودند آن ها را به آرایه‌ی explored اضافه میکنیم تا در loop تکرار گیر نکنند و بعد اگر اولین حرکت از کره بود باید با صدا زدن firstNeighbors همسایه های تولید شده را به Frontier اضافه کنیم و در غیر این صورت همسایه هایی که در آرایه های frontier و explored نبودند را به frontier اضافه میکنیم ولی اگر همسایه ها از قبل در frontier بودند کم هزینه ترین آن ها را با توجه به f نگه میداریم و بقیه همسایه ها را از frontier، pop میکنیم.

تابع IDDFS :

ابتدا یک سقف maxDepth را برای حداکثر تعداد افزایش عمق cutoff در نظر میگیریم و تا زمانی که جواب را پیدا نکردیم و یا به maxDepth نرسیدیم به مقدار cutoff یکی یکی اضافه میکنیم و در هر بار الگوریتم DLS را در آن cutoff محدود اجرا میکنیم به این صورت که ابتدا با صدا زدن تابع findingSG با توجه به کمترین فاصله یک کره و یک هدف انتخاب میشود . نقطه‌ی شروع را در frontier میریزیم و اینجا باید بررسی کنیم که اگر هدف انتخاب شده در سطر های صفر یا row-1 و یا ستون های صفر یا col-1 نبود در انتخاب همسایه ها به این خانه های موجود در این سطر ها و ستون ها نرود در بقیه موارد درست مانند A^* عمل میکند با این تفاوت که هزینه ندارد و از بین همسایه ها اولین همسایه مجاز را انتخاب میکند.

تابع Bidirectional BFS :

چندین تابع داریم : ابتدا BBFS را بررسی میکنیم که در هر بار صدا شدن آرایه visited را به تعداد کل خانه های میز false میکند و بعد یک نقطه‌ی شروع و یک نقطه‌ی پایان میگیرد و شروع را به src_queue اضافه کرده و visited آن را true میکنیم و برای نقطه‌ی پایان هم dest_queue اضافه کرده و visited آن را true میکنیم و تا زمانی که این دو آرایه مقدار دارند در حلقه‌ی while یکبار bfs با جهت رو به جلو و یکبار هم با جهت رو به عقب میزنیم(bfs جلوتر توضیح داده میشود.) بعد با استفاده از تابع is_intersecting مقدار i,j خانه ای که توسط هر دو حالت شروع و پایان true شده بود را برمیگرداند که این خانه همان خانه‌ای است که دو مسیر از ابتدا و انتها در آن یکدیگر را ملاقات میکنند که اگر این خانه موجود نبود یعنی جواب نداریم ولی اگر جواب موجود بود به تابع print_path میرویم و در این تابع با توجه به والدین هر گره مسیر را از هر دو سمت میسازیم.

تابع bfs ابتدا جهت را میگیرد اگر رو به جلو بود از آرایه src_queue آخرین عنصر را pop کرده و گسترش میدهیم به این صورت که که اگر اولین حرکت کره است باید تابع firstNeighbors را اجرا کند و در غیر این صورت تابع neighbors

و بعد از گرفتن همسایه های مجاز برای تک تک آن ها اگر قبلا ویزیت نشده بودند ویزیت میکنیم و به src_queue اضافه میکنیم و پدر آن را هم گره فعلی ذخیره میکنیم و اگر جهت رو به عقب بود یعنی direction = 'backward' همه موارد گفته شده برای حرکت رو به جلو را برای این قسمت با تغییر آرایه ها به آرایه رو به عقب انجام میدهیم.

۴. مقایسه روشهای پیاده سازی شده در موارد (زمان صرف شده،پیچیدگی زمانی،تعداد گره های تولید شده،تعداد گره های گسترش داده شده،عمق راه حل):

در IDS :

پیچیدگی زمانی : $O(b^d)$

تعداد گره های تولید شده : $N_{IDS} = (d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$

تعداد گره های گسترش داده شده : تعداد گره های گسترش داده شده و تعداد گره های تولید شده از هر دو الگوریتم دیگر بیشتر است چون هر بار با افزایش سطح از ابتدا شروع به گسترش دادن میکند.

در bidirectional BFS :

پیچیدگی زمانی : $O(b^{d/2})$

تعداد گره های تولید شده : $N(BBFS) = 2*b + 2*b^2 + \ldots + 2*b^{d/2} = O(b^{d/2})$

که البته ممکن است b ها در هر طرف متفاوت باشند ما در اینجا فرض میکنیم با تقریب یکسان اند.

تعداد گره های تولید شده توسط BBFS کمتر از IDS است.

تعداد گره های گسترش داده شده از IDS کمتر ولی از A^* بیشتر است.

در A^* :

پیچیدگی زمانی: به تابع هیوریستیک بستگی دارد هرچه خطا کمتر باشد (یعنی تابع هیوریستیک به هزینه واقعی مسیر نزدیک تر باشد) بهتر است : Polynomial when $|h(x)-h^*(x)| \le O(\log h^*(x))$

تعداد گره های تولید شده : از هر دو الگوریتم کمتر است به خصوص رمانی که تابع هیوریستیک دقیق باشد.

تعداد گره های گسترش داده شده : $N + 1 = 1 + b^* + (b^*)^2 + (b^*)^3 + \ldots + (b^*)^d$

که هرچه تابع هیوریستیک دقیق تر باشد مقدار b^* کمتر شده و گره های گسترش داده شده کمتر میشوند ولی در بدترین حالت هیوریستیک باز هم تعداد گره های گسترش داده شده از دو الگوریتم بالا کمتر است.

زمان مصرف شده	IDS	BBFS	A*
Test1	۰,۰۱۹۹۴۷۲۹۰۴۲۰۵۳۲۲۲۷	۰,۰۰۸۳۹۴۰۰۲۹۱۴۴۲۸۷۱۱	۰,۰۱۰۱۱۴۶۶۹۷۹۹۸۰۴۶۸۸
Test2	۰,۰۰۹۰۱۷۲۲۹۰۸۰۲۰۰۱۹۵	۰,۰۰۷۹۴۶۲۵۲۸۲۲۸۷۵۹۷۷	۰,۰۴۸۴۱۴۲۳۰۳۴۶۶۷۹۶۹
Test3	۰,۰۰۴۹۸۶۲۸۶۱۶۳۳۳۰۰۷۸	۰,۰۰۱۹۹۴۸۴۸۲۵۱۳۴۲۷۷۳۴	۰,۰۰۲۰۰۱۵۲۳۹۷۱۵۵۷۶۱۷
Test4	۰,۳۸۲۰۱۱۸۹۰۴۱۱۳۷۶۹۵	۰,۰۰۱۹۹۴۳۷۱۴۱۴۱۸۴۵۷۰۳	۰,۰۰۰۹۹۸۲۵۸۵۹۰۶۹۸۲۴۲۲
Test5	۰,۰۰۳۹۸۱۵۹۰۲۷۰۹۹۶۰۹۴	۰,۰۰۰۹۹۳۲۵۱۸۰۰۵۳۷۱۰۹۴	۰,۰۰۰۹۹۶۸۲۸۰۷۹۲۲۳۶۳۲۸

عمق جواب	Test1	Test2	Test3	Test4	Test5
IDS	۱۴	۱۲	۱۹	جواب ندارد	۱۷
BBFS	۵	۶	۴	جواب ندارد	۶
A*	۱۰	۱۲	۵	جواب ندارد	۱۱