محسن امجدی - 810896043

در این پروژه قصد داریم بازی snake را برای اجرا توسط الگوریتم های جستجوی ناآگاهانه و جستجوی آگاهانه مدل سازی کنیم.

بازی در یک صفحه ی مختصات در جریان است که اندازه ی صفحه ی مختصات و همچنین مختصات اولیه مار و مختصات دانه ها همراه با امتیاز آن ها در قالب یک فایل به برنامه داده می شود.

نحوه ی مدل سازی مسیله به شرح زیر می باشد :

بازی در state های مختلفی قرار می گیرد و الگوریتم روی node های گراف صفحه ی مختصات گسترش می یابد. بنابراین برنامه از کلاس های Game و State و Node تشکیل شده است.

در ابتدا برنامه فایل مربوط به بازی را خوانده و اطلاعات اولیه بازی در اختیار game که نمونه ای از کلاس Game است قرار می گیرد تا state اولیه ی بازی مشخص شود.

Initial State در این برنامه عبارت است از مختصات اولیه مار و حالت اولیه وجود دانه های درون صفحه.

Goal State در این برنامه عبارت است از حالتی که همه ی دانه های موجود در صفحه توسط مار خورده شود. که این به این معنی است که امتیاز مربوط به هر دانه در هر مختصات صفر شود.

بنابراین state اولیه بازی به محظ ساخته شدن نمونه ای از کلاس Game ساخته می شود و تابعی به نام is_goal در کلاس Game در هر مرحله از بازی چک می کند که آیا هدف بازی محقق شده یا خیر. که در این تابع با توجه به این که برای دانه های درون صفحه یک dict در نظر گرفتم که امتیاز هر دانه را به مختصات آن دانه مربوط می کند در صورتی که امتیاز همه ی مختصات های درون dict صفر شده باشد به goal state رسیده ایم.

از این پس روند بازی توسط الگوریتم های جستجو ادامه می یابد. به این ترتیب که نمونه ی ساخته شده از کلاس Game به الگوریتم داده می شود تا بازی را پیش ببرد و به state نهایی که هدف ماست برسد.

هر الگوریتم با گسترش node ها روی گراف کار میکند که هر node اطلاعاتی نظیر state بازی و parent مربوط به خود و همچنین هزینه ی مسیر رفته شده و action مربوط به خود را نگه می دارد.

ابتدا از الگوریتم BFS که یک الگوریتم جستجوی ناآگاهانه است استفاده می کنیم. در این الگوریتم ابتدا یک node اولیه ساخته شده که شامل state اولیه بازی می شود و چک می شود که اگر نود ریشه نود هدف است به عنوان جواب برگردانده شود.

الگوریتم BFS شروع به جستجو از گره ریشه درخت می کند و قبل از انتقال به گره های سطح بعدی ، همه گره های جانشین را در سطح فعلی گسترش می دهد.

برای ذخیره کردن node هایی که باید بسط داده شوند از یک صف استفاده می کنیم که یک صف fifo است و برای node هایی که بررسی شده اند از یک set استفاده میکنیم که با نام reached در الگوریتم وجود دارد.

الگوریتم در یک حلقه چک می کند که تا زمانی که node برای expand شدن در صف وجود دارد اجرا شود. برای هر node که از صف بر میدارد تابع expand را صدا می کند که تابع expand مفهوم گسترش دادن و حرکت کردن مار را اجرا می کند. به این صورت که برای state نودی که گرفته action های بازی را اجرا و نتیجه ی هر action که توسط تابع result برگردانده می شود یک state جدید است که ذخیره می شود.

در الگوریتم برای هر node که expand داده می شود حالت نهایی بازی چک می شود و اگر حالت نهایی محقق شده بود متوقف می شود اگر نه node دیگری که state تکراری نیست برای expand شدن انتخاب می شود.

Action مار برای هر مرحله از بازی در کلاس Game تعریف شده که حرکت مار را مشخص می کند. مار به مختصاتی حرکت می کند که عضوی از مختصات بدن خودش نباشد. مختصات بدن مار در هر state ذخیره می شود بنابراین تابع action یک state به عنوان ورودی می گیرد تا مشخص کند در state فعلی مختصات بدن مار و سر مار به چه صورت است و در نهایت حرکت های ممکن مار را بر می گرداند.

Result تابعی است که پس از انجام هر action مشخص می کند که state بازی به چه صورت update می شود. State بازی در هر حرکت منجر به خوردن یا نخوردن دانه ای می شود. در هر دو صورت مختصات سر مار update می شود اما در صورت خوردن دانه مختصات ته مار در snake_body_coordinates باقی مانده و در صورت نخوردن دانه مختصات ته مار از snake_body_coordinates حذف می شود.

بنابراین result هر حرکت می تواند منجر به تغییر مختصات بدن و سر مار شود همچنین می تواند منجر به کم شدن امتیاز دانه خورده شده شود. که این نتایج به عنوان یک state جدید و برای نود expand داده شده در نظر گرفته می شود.

الگوریتم مورد استفاده ی بعدی الگوریتم iterative deepening search است که نوعی الگوریتم جستجوی ناآگاهانه است. استراتژی جستجوی این الگوریتم بر اساس یک نسخه ی depth-limited از الگوریتم depth-first-search است که تا وقتی به هدف برسد مکررا اجرا می شود و هر بار depth limit را افزایش می دهد.
نحوه ی گسترش node ها در این الگوریتم نیز توسط همان تابع expand انجام می شود.

هر دو الگوریتم BFS و IDS کامل هستند زیرا حتما به جواب می رسند اما از نظر بهینگی حافظه الگوریتم IDS حافظه بسیار کمتری اشغال می کند. بنابراین الگوریتم IDS در مواقعی که فضای جستجوی خیلی بزرگ است برای استفاده ارجحیت دارد .

تفاوت دیگر این دو نوع الگوریتم در این است که برای پیاده سازی لیست frontier در الگوریتم BFS از یک صف (FIFOQueue) استفاده می کنیم و برای پیاده سازی frontier در الگوریتم IDS از stack یا LIFOQueue استفاده می کنیم.

توجه شود که با توجه به این که IDS نسخه ی depth-limited الگوریتم DFS است در کد نوشته شده تابع مربوط به IDS تابع depth limited search را برای این منظور صدا می کند.

الگوریتم جستجوی عمق محدود مشابه جستجوی عمق اول با حد از پیش تعیین شده است.
بنابراین ، جستجوی depth limited را می توان یک نسخه گسترده و تصحیح شده از الگوریتم DFS نامید. به طور خلاصه ، می توان گفت که برای جلوگیری از وضعیت حلقه بی نهایت هنگام اجرای کدها ، الگوریتم جستجوی depth limited در یک مجموعه محدود از عمق به نام depth limit اجرا می شود.

جستجوی depth limited با دو شرط fail می تواند خاتمه یابد:
standard failure : این نشان می دهد که مشکل هیچ راه حلی ندارد.
Cutoff failure : هیچ راه حلی برای مسئله در یک depth limit مشخص تعریف نمی شود.

حال الگوریتم جستجو IDS بهترین depth limit را پیدا می کند و این کار را با افزایش تدریجی حد تا یافتن هدف انجام می دهد. این الگوریتم جستجوی dfs را تا یک "depth limit" مشخص انجام می

دهد و پس از هر تکرار تا یافتن گره هدف ، محدودیت عمق را افزایش می دهد.این الگوریتم وقتی فضای جستجو بزرگ است و عمق گره هدف مشخص نیست ، جستجو ناآگاهانه مفیدی است.

الگوریتم مورد استفاده ی بعدی الگوریتم A* است که نوعی الگوریتم جستجوی آگاهانه است.
این الگوریتم از یک تابع heuristic و یک تابع g(n) که هزینه رسیدن به گره n از حالت شروع است استفاده می کند. و جمع این دو تابع را تحت عنوان تابع f(n) در نظر می گیرد.

برای پیاده سازی الگوریتم a star از آنجایی که عملکرد آن مشابه الگوریتم best first search است اما با یک f متفاوت با محاسبه ی f مورد نظر تابع best first search را صدا می کنیم.

در BFS و DFS ، وقتی در یک گره هستیم ، می توانیم هر یک از مجاورها را به عنوان گره بعدی در نظر بگیریم. بنابراین هر دو BFS و DFS بدون در نظر گرفتن هیچ تابع هزینه ای کورکورانه مسیرها را کاوش می کنند. ایده best first search استفاده از یک تابع ارزیابی برای تصمیم گیری در مورد همسایگی و سپس کاوش است.

در این الگوریتم best first search دنبال گره با کمترین f(n) می گردیم. برای این منظور از نوعی صف اولویت priority queue استفاده می کنیم. این صف که پیاده سازی آن در کد موجود است همیشه f(item) کمینه را اول از همه pop می کند.
الگوریتم جستجوی best-first همیشه مسیری را انتخاب می کند که در آن لحظه به بهترین شکل ظاهر شود.
الگوریتم a star کامل و بهینه است.
الگوریتم جستجو A* بهترین الگوریتم نسبت به الگوریتم های جستجوی دیگر است.
همیشه کوتاهترین مسیر را ایجاد نمی کند زیرا بیشتر مبتنی بر heuristic و تقریب است.
اشکال اصلی A* نیاز به حافظه است زیرا تمام گره های تولید شده را در حافظه نگه می دارد ، بنابراین برای مشکلات مختلف در مقیاس بزرگ عملی نیست.