معرفی بازی

بازی کریدور با داشتن فضای حالتی به اندازه‌ی 10^{42} و ضریب انشعاب متوسط 60 ، جزو بازی های چالش برانگیز در حوزه‌ی هوش مصنوعی محسوب می‌شود ، چراکه اندازه ی فضای حالت آن تقریبا به بزرگی فضای شطرنج می‌باشد و زمان حل آن به ازای افزایش عمق درخت به صورت تقریبا نمایی رشد میکند (چرا که هنوز مشخص نشده است که این بازی به کدام کلاس پیچیدگی تعلق دارد). موضوعی که سبب شده است بازی کریدور به اندازه ی شطرنج محبوب واقع نشود، پیچیدگی حافظه ی مساله می‌باشد که ثابت شده است که متعلق به کلاس چندجمله ای می‌باشد ، ما در این پروژه با استفاده از برنامه ریزی پویا، راه‌حلی ارائه می‌دهیم که بتوان به کمک آن، مساله را برای عمق دلخواه، در مرتبه ی ثابت حل نمود مشروط به اینکه تمام داده های زیرمساله های بازی، از قبل در پایگاه داده ی بازی وجود داشته باشد. روش پیاده‌سازی بازی با استفاده از درخت min-max می‌باشد و از الگوریتم ژنتیک برای آموزش عامل های بازی بهره برده‌ایم.

شرح روش ها

- **پیاده‌سازی عامل های بازی**

عامل های بازی، تصمیم های خود را مبتنی بر شاخص فاصله‌ی منهتن خود و رقیب از مقصد نهایی در بازی می‌گیرند: بنابراین تابع ارزیابی موقعیت های حاصل شده در بازی بر اساس همین دو پارامتر خواهد بود و قصد داریم ضریب های مناسب را در پایان برای این دو متغیر بیابیم.

- **مشخص کردن فضای حالت بازی**

ما قصد داریم مساله را به صورت پویا حل کنیم، بنابراین نیازمند آنیم که بتوانیم زیر مساله های بازی را به شیوه‌ی مناسب نمایش بدهیم، برای دسته بندی نمایش بازی، ساختار های مختلفی قابل استفاده هستند که مزایا و معایب خود را دارند ، ما از دو ساختار SubProblem برای کوتاه سازی زیردرخت های مساله و LeafSubProblem برای کوتاه سازی برگ های درخت مساله استفاده کرده ایم که دلایل استفاده را در ادامه شرح خواهیم داد :

SubProblem[p1][p2][d][vs][hs] : $p1, p2, pc \in \{1,2\}$ $vs, hs \in \{0, 10^{32} - 1\}$

SubProblem[i][j][k][l][m]: $\mathbb{N} \times \mathbb{N} \to \mathbb{R}$

LeafSubProblem[p1][p2][cp]: $\mathbb{N} \times \mathbb{N} \to \langle d1, d2, f \rangle$

که در آن، SubProblem آرایه ای از هش لیست‌هایی است که موقعیت دقیق چینش بازی را مشخص می‌کنند ، p1 مشخص کننده ی موقعیت بازیکن اول در صفحه‌ی بازی، p2 مشخص کننده‌ی موقعیت بازیکن دوم در صفحه‌ی بازی، d عمق درخت زیرمساله و همچنین دو متغیر vs و hs هش‌هایی برای مشخص کردن طریقه ی چینش مانع‌ها در بازی می باشند که درباره‌ی آن توضیح میدهیم. همچنین تابع هش مربوط به SubProblem به این گونه است که با دریافت دو عدد long که اطلاعات دقیق چینش مانع‌ها می‌باشند، مقدار عددی که کاملا وابسته به ضرایب تعیین شده برای تابع ارزیابی است را برمی‌گرداند. LeafSubProblemنیز آرایه ای از لیست های مبتنی بر هش است با این تفاوت که برای صرفه جویی در حافظه‌ی مصرفی ، ساختار نقشه را به یک آرایه‌ی سه بیتی، تصویر می کند که مستقل از ضریب های تابع ارزیابی می‌باشد : p1 موقعیت بازیکن اول، p2 موقعیت بازیکن دوم و cp بازیکنی که قرار است تصمیم بگیرد را مشخص می‌کند همچنین آرایه ای که به عنوان جواب برگردانده می‌شود،

می دانیم که بنابر شکل صفحه‌ی بازی، دقیقا 64 مربع وجود دارد که هر بازیکن می تواند در آن ها مانع هایی را قرار بدهد، که مانع ها یا افقی هستند یا عمودی، با توجه با این که نوع داده‌ی long دقیقا 64 بیت دارد، می‌تواند یک کاندید مناسب برای نمایش نقشه باشد.

اکنون بپردازیم به موضوع چرایی استفاده از دو نوع مختلف زخیره سازی درخچه ها و برگ ها : این کار به این دلیل صورت گرفته که پیاده سازی برگ ها از ضرایب مستقل باشد تا بتوان هنگام آموزش از آن ها بهره برد، چون قرار است درختچه های جدید با استفاده از برگ ها ساخته شوند، که البته هزینه ای مانند پیچیده شدن الگوریتم درخت دارد که ارزش پرداخت دارد.

به طور کلی استفاده از کاهش برگ، هزینه دوبار جستوجوی در گراف را به یک مقدار ثابت کاهش می دهد و کاهش درخت نیز، می تواند هزینه را به صورت نمایی کاهش دهد که این موضوع سبب می شود که بتوانیم جستوجوهای عمیق را در زمان کم انجام دهیم.(با فرض داشتن حافظه ی کافی)

- **طریقه ی ساخت جدول *memorize* برای بازی**

برای ساخت جدول، می توان راه های مختلف رو در نظر گرفت که ما در پروژه، دو راه را پیاده سازی نموده ایم، که راه اول به خاطر نیاز داشتن به حافظه ی بیشتر و زمان به مراتب بیشتر، برای انجام دادن مقدور نبود.

راه اول ساختن جدول :

در این روش، عامل در هنگام بازی کردن باید درخت ها را کشف کرده و در جدول قرار دهد، این راه علاوه براینکه بسیار زمان بر است، نمی تواند زمان ثابت برای عامل را در طول بازی پیش بینی کند و اگر عمق درخت زیاد باشد، با توجه به زمان نمایی، عملا عامل متوقف می شود. ولی با فرض تعداد زیاد بازی های عامل و اینکه طرف مقابل به صورت هوشیارانه بازی میکند، می تواند برای عمق های کم مورد استفاده قرار بگیرد.

راه دوم ساخت جدول :

در این راه، درخت را به صورت منظم و از برگ ها تا عمق دلخواه می سازیم، این روش یک مشکل اساسی دارد : حافظه ،اگر بتوانیم به اندازه ی کافی حافظه فراهم کنیم، با ساختن درخت از پایین به بالا می توانیم تا عمق دلخواه پیشروی کنیم و چون زیر درخت ها را زخیره می کنیم، عملا می توان گفت که تمام مراحل بعدی در زمان ثابت قابل برسی هستند.متاسفانه به دلیل کمبود سخت افزاری از پیاده سازی این قسمت خودداری شد هرچند که کد آن موجود است.

(توجه : به خاطر حجم بسیار بالای داده های بازی ، از ارسال آن ها معذوریم. البته بازی با گذشت زمان داده ها را برای خود جمع آوری کرده و در پوشه ی داده ها دوباره فایل های داده را می‌سازد.)

- **طریقه ی آموزش عامل های بازی**

در آموزش عامل های بازی از یک گروه به اندازه ی 2 ژن برای والد ها استفاده شده است،10 فرزند از این دو گروه مستقل از برنده به دست می آوریم، سپس از بین فرزندان ژن منتخب را بر می گزینیم. با توجه به دانش اولیه از مساله، فاصله ی خریف از مقصدش را بر فاصله ی عامل تا مقصد، ترجیح دادیم. و 2 ژن والد را تعیین نمودیم، سپس با ادامه ی فرآیند و رقابت بین فرزندان، میزان شایستگی آن ها را بر اساس شاخصه های زیر سنجیدیم :

شاخصه ی اول : فاصله ی کلی که عامل برنده تا رسیدن به نقطه ی پایان طی کرده

شاخصه ی دوم : فاصله‌ای که حریف بازنده طی کرده است

شاخصه ی سوم : تعداد مراحل بازی که صورت گرفته در کل

شاخصه ی چهارم : فاصله ای که برای حریف بازنده تا مقصدش باقی مانده است

تعدادی از ژن های 14 نسل آخر پروسه را می توانید در پوشه‌ی پروژه پیدا کنید. به بازنده مقدار 1- نسبت داده ایم و از لیست حذف شده اند، ازبین 5 برنده، با روش تورنمنت، ژن های شایسته را برمی‌گزینیم تا مقدار در نهایت به عددی ثابت همگرا

شود.در ضمن آموزش عامل ها به خاطر وابستگی آن ها به ضرایب، تنها با کاهش برگ و در عمق 2 صورت گرفت.