محمدرضا کیانی‌فر
۸۱۰۱۹۴۳۸۸

گزارش کار

برای انجام این پروژه، من قدم‌های زیر را انجام دادم تا نهایت توانستم خواسته‌ی مسئله را انجام دهم:

۱. فهم کد داده شده:

ابتدا زمانی را به بررسی کد داده شده اختصاص دادم و در طی بررسی آن با کارکرد کلاس های مختلف مانند کلاس Node و Tree و RandomMinimax و ... آشنا شدم و فهمیدم که Utility هر نود به دو صورت مقدار دهی می‌شود: اگر برگ باشد، با Evaluation Function و در غیر این صورت با استفاده از Minimax.

۲. نوشتن یک cliBoardGame:

به علت مشکل‌دار بودن کتاب‌خانه‌ی pygame روی سیستم‌عامل مک و به خصوص macOS Mojave، من نتوانستم رابط گرافیکی را اجرا کنم و پس از مدت زیادی جستجو به این نتیجه رسیدم که یک بردگیم بنویسم که در ترمینال نتیجه‌ی هر دور بازی را نمایش دهد.

۳. پیدا کردن Evaluation Function مناسب:

سپس به کلاس Node تابعی اضافه کردم که utility آن نود را محاسبه می‌کرد که آن را در قسمت‌های بعدی به طور مفصل شرح خواهم داد.

۴. نوشتن کد Minimax به صورت DFSای:

کد داده شده به ما به ابتدا utility تمام برگ‌های ارتفاع نهایی را حساب می‌کرد و سپس از پایین به بالا الگوریتم Minimax را روی آن اجرا می‌کرد. این کد قابلیت افزودن الگوریتم Alpha Beta را نداشت چرا که مسئله از بالا به پایین حل نمی‌شود و جایی نمی‌توانیم حرصی انجام دهیم. بنابراین من یک الگوریتم بازگشی و با دید DFS نوشتم که همان کار Minimax را انجام می‌داد ولی پتانسیل بهبود با Alpha Beta را نیز داشت.

۵. پیاده‌سازی الگوریتم Alpha Beta Pruning:

برای پیاده‌سازی این قسمت مطابق سودوکد گفته شد در سر کلاس، دو پارامتر alpha و beta را به تابع compute_minimax_value در کلاس AlphaBetaPruning اضافه کردم که آلفا نشان‌دهنده‌ی بیشنه مقدار utility پیدا شده برای سطرهای زوج (مربوط به بازیکن خودی) و بتا مقدار کمینه‌ی utility پیدا شده برای سطرهای فرد (مربوط به بازیکن حریف) است. حال اگر در پیمایش یک نود، (برای مثال برای آلفا) مقدار یوتیلیتی یکی از برگ‌های آن نود از آلفا کمتر باشد، می‌توانیم از پیمایش سایر برگ های آن نود صرف نظر کنیم چرا که در این حالت نود مینیموم مقدار برگ‌های خود را به عنوان یوتیلیتی انتخاب می‌کند و این مقدار عددی کمتر از بهترین یوتیلیتی پیدا شده برای آن نود (آلفا) می‌باشد؛ پس بنابراین پیمایش سایر برگ‌های آن نود تاثیر مثبتی نخواهد داشت.

قبل از پیاده‌سازی این روش، در صورتی که من با عمق ۴ می‌خواستم جستجو کنم، زمانی بیش از ۱۲ ثانیه طول می‌کشید اما زمانی که این بهبود را انجام دادم، زمان برای هر قدم در عمق ۴ به ۴-۵ ثانیه کاهش یافت.

پیدا کردن Evaluation Function مناسب

من سه نوع تابع یوتیلیتی مختلف را بررسی کردم که به معرفی هر کدام از آن‌ها می‌پردازم و در نهایت تابع نهایی خودم را مشخص می‌کنم.

۱. Custom Evaluation

اولین راهی که من به نظرم رسید این بود که هم تعداد مهره‌ها را بررسی کنم و هم جایگاه مهره‌ها، یعنی هر چه جلوتر باشند امتیازی بالاتری می‌گیرند. این روش پس از پیاده‌سازی ایجنت رندوم را همواره شکست می‌داد.

۲. Defensive Evaluation

در این راه استراتژی بازیکن حفظ مهره‌های خود است. یعنی تابع utility برابر است با تعداد مهره‌های خودی * ۲ + یک عدد رندوم کوچک برای ایجاد و کنترل خطا حریف. این روش ایجنت با تابع قبلی را شکست می‌داد.

۳. Aggressive Evaluation

در این راه استراتژی بازیکن زدن مهره های طرف مقابل است و utility آن برابر است با(مهره‌های حریف ـ ۳۰) * ۲ + یک عدد رندوم کوچک برای ایجاد و کنترل خطا حریف. این روش ایجنت با دو تابع قبلی را شکست می‌داد بنابر این تابه نهایی من این شد.

مزایای Alpha Beta Search نسبت به Minimax Search

در الگوریتم آلفا بتا، ما تمام حالت ها را مانند Minimax جستجو نمی‌کنیم بلکه طبق روشی که بالاتر توضیح داده شد، حرص‌هایی در پیمایش درخت انجام می‌دهیم و این باعث می‌شود برای جستجوی یک ارتفاع برابر در هر دو الگوریتم، الگوریتم آلفا بتا جستجو را در زمان کمتری انجام می‌دهد زیرا با حرص‌هایی که انجام می‌دهد، فضای حالت کوچک‌تری نسبت به Minimax دارد. پس بنابراین ممکن است در آلفا بتا بتوانیم در یک زمان برابر ارتفاع بیشتری را با همان Optimum ای نسبت به Minimax پیمایش کنیم.

بررسی اکسپنشن درخت

با توجه به این که هر بازیکن ۱۲ مهره در اختیار دارد و هر کدام قادر به ۳ جابه‌جایی هستند، در بدترین حالت هر نود، 3^{12} بچه یعنی 531,441 بچه می‌تواند داشته باشد اما با توجه به حرکت‌های مجاز ممکن، این عدد بالاتر از ۵۰۰-۳۰۰ نخواهد بود و این یعنی در ارتفاع ۳، حدود ۳۰ میلیون حالت داریم. با استفاده از الگوریتم آلفا بتا، ما می‌توانیم ارتفاع ۴ را در زمان ۴-۵ ثانیه پیمایش کنیم و این یعنی بسیاری از این حالت‌ها در آلفا و بتا حذف می‌شوند.

بهبودهای مسئله

من زمان پیاده‌سازی این بهبودها را نداشتم ولی این ایده‌ها برای بهبود زمان اجرایی ممکن است کارآمد باشند:

۱. در هر مرحله، Evaluation Function را برای تمام فرزندان یک نود حساب کنیم و در نهایت اکسپنشن مینیمکس را تنها برای مثلا ۲۰٪ جمعیت برتر آن‌ها انجام دهیم. این باعث می‌شود حرکاتی که خوب نیستند، زودتر حذف شوند و ما بی‌جهت چند لایه برای بررسی آن‌ها پیمایش نکنیم. (مثلا در بازی شطرنج وقتی وزیر بازیکن در حرکت بعدی از دست می‌رود، به احتمال زیاد آن حرکت حرکت مناسبی نبوده و به جای بررسی چندلایه پایین‌تر و بعد تصمیم نگرفتن آن، می‌توان از همان ابتدا آن را حرص کرد.)

۲. ایده‌ی دیگر استفاده از دیپنینگ سرچ می‌باشد که در آن برای استفاده‌ی بهینه‌تر از زمان، لول‌های جستجو را به جای عدد ثابت، به شکل داینامیک بیشتر یا کمتر می‌کنیم. البته این ایده تاثیر زیادی در بهینه‌سازی نخواهد داشت.