دانشگاه تبریز

پیاده سازی بازی دوز تک‌نفره با هوش مصنوعی با درخت مین–مکس و هرس آلفا–بتا

امتیاز ویژه این پروژه: پیاده سازی اختیاری هرس آلفا–بتا

استاد: جناب آقای دکتر فیضی درخشی

نویسنده: پارسا یوسفی نژاد محمدی

شماره دانشجویی: ۱۴۰۰۵۳۶۱۱۰۴۸

مقدمه

دوز یک بازی بسیار محبوب و جذاب بوده که طرفداران زیادی دارد، این بازی دو نفره و دارای دو نماد X و O است و به همین دلیل به بازی XO نیز شهرت دارد، در این پروژه می‌خواهیم محل تک‌نفره این بازی را پیاده سازی کنیم، به گونه‌ای که در سمت دیگر بازی، هوش مصنوعی قرار داشته باشد و بتواند با الگوریتم‌های که در این پروژه به آن می‌پردازیم، سعی در انتخاب بهترین حرکت در جهت مقابله با بازیکن واقعی که انسان است بکند و از برد حریف جلوگیری کند.

هدف پروژه

در این پروژه می‌خواهیم **بازی دوز ۳×۳** را به کمک الگوریتم‌های درخت min-max و همچنین بکارگیری **هرس آلفا-بتا** برای کاهش فضای جستجوی حالات پیاده‌سازی کرده و بازی را در یک قالب گرافیکی مجزا به کمک کتابخانه **pygame** نمایش بدهیم، در ابتدا به بررسی نحوه پیاده‌سازی UI بازی می‌پردازیم و سپس بازی را به کمک ساختمان داده‌های گوناگون، به درون دنیای الگوریتم‌ها **مدل‌سازی** می‌کنیم. در گام بعدی به پیاده‌سازی بخش اصلی پروژه که همان الگوریتم‌های از پیش ذکر شده است، خواهیم پرداخت. در این بازی همواره **برنده** در صورت وجود **هوش مصنوعی** خواهد بود، چرا که همواره بهترین بازی خود را ارائه می‌دهد و امکان باخت ندارد ولی **ممکن** است که بازی به **تساوی** کشیده شود.

تشریح پروژه

کد این پروژه از قسمت‌های گوناگونی تشکیل شده است، در ابتدا ثابت‌های مورد نیاز بازی را تعریف کردیم و سپس یک صفحه گرافیکی را توسط کتابخانه pygame ایجاد کردیم و در مرحله بعدی کلاسی به نام TicTacToeGame ایجاد کردیم و تمام توابع مورد نیاز بازی را در آن درست کردیم و بعد آن یک نمونه از این کلاس ایجاد و متد StartGame() آن را جهت اجرای بازی فراخوانی کردیم تا بازی شروع به کار کند؛ در ادامه قدم به قدم به بررسی اجزای مختلف این پروژه خواهیم پرداخت و جزئیات و نحوه عملکرد هر یک را به تفصیل بیان خواهیم کرد:

پیاده سازی گرافیک بازی:

بازی دوز از یک صفحه مربعی با **۹ خانه** ۳×۳ تشکیل شده است، در ابتدا باید کتابخانه pygame را جهت ساخت چنین گرافیکی import کنیم و سپس شروع به initialization کردن گرافیک تخته کنیم، برای این کار تخته‌ای با ابعاد مربعی و با رنگ background و نام تخته دلخواه ایجاد می‌کنیم در مراحل بعدی توابعی را می‌سازیم تا چنین شکلی را از بازی برای کاربر ایجاد کنند. در ادامه به توابع پیاده سازی شده جهت محقق کردن این خواسته‌ها می‌پردازیم:

- **DrawBoardLines():**

به کمک این تابع، برای صفحه بازی که gameBoard نام دارد، **۲** خط **عمودی** و **۲** خط **افقی** و هچنین **۴** خط برای **حاشیه**های بازی ایجاد می‌کنیم تا شکل اولیه تخته بازی دوز تشکیل شود.

- **DrawSymbol():**

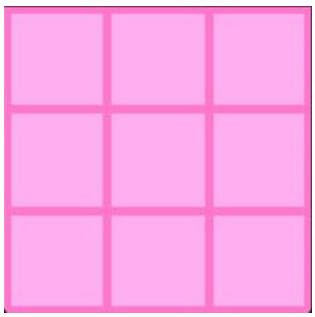
با این تابع میتوانیم نمادهای **X** و **O** را بر روی صفحه بازی ۹ خانه‌ای رسم کنیم، این تابع مختصات خانه‌ای که قرار است سمبل در آن قرار گرفته شود و همچنین بازیکنی که می‌خواهد این سمبل را رسم کند می‌گیرد و در ادامه بررسی می‌کند که آیا بازیکن فعلی که درخواست رسم را داده است، کامپیوتر یا انسان است. در صورتی که بازیکن کامپیوتر باشد در خانه فعلی یک **دایره**(O) رسم می‌شود و در غیر این صورت که حریف انسان است، تابع یک X (**ضربدر**) در آن خانه رسم می‌کند.

- **DrawWinLine():**

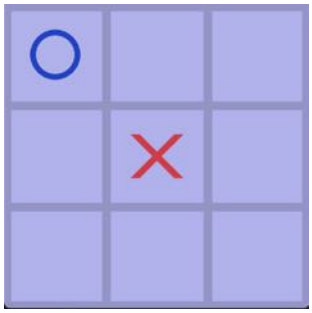
این متد زمانی فراخوانی می‌شود که بازی با برد، آن هم برد کامپیوتر همراه بوده است، چرا که این بازی غیر قابل بردن توسط انسان و یا هر موجودیت دیگری است، سپس اقدام به رسم یک خط بر روی تمام سمبل‌هایی که منجر به تشکیل دوز شده‌اند می‌کند. این تابع با گرفتن چند آرگومان مانند جهت دوز و یک خانه هم‌جهت دوز می‌تواند خطی را به صورت **عمودی**- **افقی**- **مورب** (کاهشی و افزایشی) بکشد.

- **DrawEqual():**

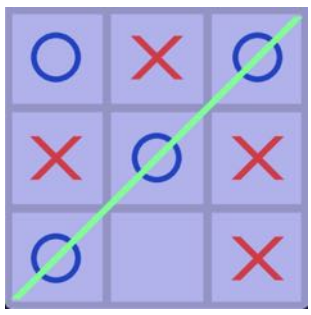
در صورتی که تمام مهره‌های بازی بر روی تخته قرار گرفته باشند اما هیچ بازیکنی موفق به دوز کردن نشده باشد، این تابع برای نشان دادن وضعیت تساوی بازی فراخوانی میشود و متن Draw را با بک‌گراند شفاف بر روی صفحه تخته بازی به نمایش می‌گذارد.



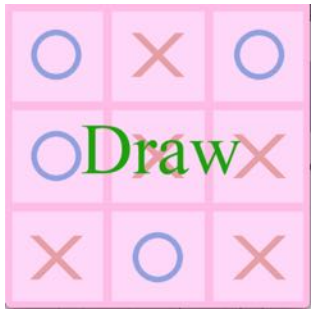
DrawBoardLines()



DrawSymbol()



DrawWinBoard()



DrawEqual()

پیاده سازی متدهای اصلی کلاس بازی دوز:

در این بخش به بررسی هر یک از توابعی که برای انجام بازی دوز احتیاج است خواهیم پرداخت و هر کدام را مفصل بررسی می‌کنیم:

- ### `__init__():`

این تابع تنها یکبار در زمان نمونه‌سازی از کلاس به صورت اتوماتیک فراخوانی می‌شود و برخی از تنظیمات و متغیرسازی های اولیه بازی را انجام می‌دهد، کارهایی که این تابع انجام می‌دهد عبارت‌اند از:

1. **تعریف یک تخته ۹ تایی ۳×۳ با کمک یک لیست ۳ بعدی با مقدار Empty**
2. **مقداردهی اولیه سلول‌های خالی با خود تخته اصلی بازی**
3. **انتخاب بازیکن شروع کننده (پیش‌فرض = HUMAN)**
4. **ست کردن وضعیت بازی (running) بر روی true**
5. **کشیدن خطوط تخته بازی با Draw BoardLines()**

- ### `MarkSymbolToCell():`

با کمک این تابع می‌توانیم یک خانه از تخته بازی را با مقدار بازیکن درخواست دهنده پر کنیم و آن را از حالت Empty خارج کنیم، در این بازی، علامت بازیکن انسان عدد ۱ که همان X است، بوده و علامت هوش مصنوعی عدد ۲ (O) می‌باشد با اینکار ماتریس ۳ در ۳ تخته بازی مقدارش تغییر پیدا میکند.

- ### `GetEmptyCells():`

همانطور از نام این تابع مشخص است، وظیفه این تابع بررسی هر ۹ سلول خانه تخته بازی بوده و در صورتی که خانه خالی با مقدار 0 پیدا کند، مختصات سطر و ستون آن سلول را داخل لیستی به نام emptyCells به صورت تاپل وارد کرده و در آخر لیست مختصات خانه‌های خالی را برمی‌گرداند.

- ### `CheckWin():`

این تابع با گرفتن یک حالت از تخته بازی، بررسی می کند که آیا حالت فوق، حالتی است که منجر به برنده شده و دوز کردن بازیکنی شود یا خیر، و در صورتی که این حالت یافت شد، آن بازیکن را به عنوان خروجی برمی‌گرداند و در صورتی که هیچ کسی برنده نشده باشد، عدد 0 را برمی‌گرداند. همچنین در صورتی که پرچم gameFinished فعال شده باشد و در تخته **حالت دوز** وجود داشته باشد با کمک فراخوانی تابع drawWinLine، اقدام به رسم خط برد برای این دوز می‌کنیم. این تابع به ترتیب حالات فوق را که هر کدام **الگوی حالت نهایی** هستند را بررسی کرده و در صورت تطابق با یکی از الگوهای گفته شده بازیکن فوق را برمی‌گرداند:

1. **بررسی بردهای ستونی: بررسی می‌کند که آیا دوزی در ۳ ستون تخته اتفاق افتاده یا نه.**
2. **بررسی بردهای سطری: بررسی می‌کند که آیا دوزی در ۳ سطر تخته وجود دارد یا نه.**
3. **بررسی بردهای مورب قطری: آیا دوزی در یکی از دو قطر اصلی و فرعی تخته بازی هست یا خیر.**

- ## TakeNextAction():

ورودی این تابع مختصات یک خانه به صورت سطر و ستون است، این سطر و ستون نتیجه انتخاب بازیکن انسان و یا کامپیوتر در نوبت خودشان است، وظیفه‌ای که این تابع دارد این است که مجموعه دستوراتی را در خانه فوق با توجه به اینکه بازیکن درخواست دهنده حرکت بعدی که است، انجام دهد، کارهایی که این دستور جهت تولید حالت جدید بعدی انجام می‌دهد به شرح ذیل است:

1. **علامت زدن آن خانه با توجه به بازیکن درخواست دهنده با تابع MarkSymbolToCell()**
2. **کشیدن سمبل بازیکن بر روی خانه انتخاب شده بر روی تخته بازی گرافیکی با تابع DrawSymbol()**
3. **عوض کردن نوبت بازیکن فعلی با بازیکن دیگر بخاطر قانون نوبتی بودن بازی دوز**

- ## IsCellEmpty():

ورودی این متد سطر و ستون یک خانه از حالتی از بازی بوده و خروجی آن، یک بولین مبنی بر خالی یا پر بودن آن سلول از تخته بازی می‌باشد.

- ## IsBoardFull():

بررسی میکند که آیا تخته بازی دوز تمام خانه‌های پر است یا نه و بولینی بر همین مبنا برمی‌گرداند.

- ## CheckGameOver():

وظیفه این تابع این است که بررسی کند آیا بازی فوق به اتمام رسیده است یا نه، این تابع خروجی نداشته و فقط در صورتی که در حالت اتمام بازی قرار داشته باشیم، وضعیت running بازی را به false تغییر می‌دهد تا دیگر هوش مصنوعی در خانه‌ای که تمام خانه‌های آن پر هستند و یا برنده شده است دست به انتخاب خانه دیگری که نباید انتخاب شود نزند و به ارور برخورد نکند. دستورالعمل‌هایی که این تابع بررسی میکند تا بفهمد که آیا بازی تمام شده است به صورت زیر است:

1. **بررسی کردن اینکه آیا بازی برنده‌ای داشته است و یا اینکه تخته بازی پرشده باشد و در صورت true شدن این بررسی، اقدام به تغییر وضعیت بازی و false کردن runnning.**
2. **در صورتی که بازی برد نداشه باشد اما تخته بازی پر شده باشد، تابع drawEqual را برای اعلام نتیجه مساوی فراخوانی کردن.**

- ## Reset():

این تابع در صورتی فراخوانی می‌شود که انسان تمایل به ادامه بازی **از ابتدا** داشته باشد و این آمادگی را با فشردن دکمه **space** اعلام میکند و موجب فراخوانی این تابع و رست کردن برنامه می‌شود. کاری که این برنامه انجام می‌دهد فراخوانی و مقدار دهی اولیه به تمام متغیرهای بازی توسط کال کردن **تابع __init__** می‌باشد.

StartGame():

startGame() از مهمترین توابع کلاس بازی دوز می‌باشد چرا که وظیفه هماهنگ کردن قسمت‌های گوناگون توابع کلاس و گرافیک و اجرا کردن بازی را برای کاربر بر عهده دارد. و در بیرون کلاس پس از نمونه سازی از کلاس فوق، با فراخوانی این تابع، بازی دوز را شروع می‌کنیم. حال به بررسی نحوه عملکرد این تابع مهم می‌پردازیم:

1. **یک حلقه با بینهایت بار تکرار داریم که همواره به صورت Polling اقدام به بررسی ورودی‌های کاربر انسان که به صورت ورودی ماوس جهت وارد کردن مختصات سمبل انتخاب شده در تخته و فشردن کلید space جهت رست کردن بازی، میکند.**
2. **در داخل حلقه نامحدودمان در ابتدا اتفاقاتی را که مربوط به اعمالی است که انسان می‌تواند به واسطه رابط گرافیکی انجام دهد را بررسی می‌کنیم و پس از آن در صورتی که نوبت بازیکن کامپیوتر شود، کارهایی را که کامپیوتر پس از انسان بر روی تخته بازی انجام می‌دهد را بررسی می‌کنیم.**
3. **میگویم که pygame که همان رابط گرافیکی‌ای است که انسان با آن در تعامل است به ازای تمام رویدادهایی که انسان در اختیار دارد بررسی کند که آیا در ابتدا بازی به رویداد خروج (فشردن دکمه بستن برنامه) رفته است یا خیر، در صورت اینکه دکمه خروج فشرده شده باشد، به پای‌گیم می‌گوییم که آن هم از برنامه بیرون بیاید و برنامه خاتمه پیدا کند.**
4. **در صورتی که رویداد مربوط به فشردن کلید space باشد، به پایتون میگوییم که تابع Reset را فراخوانی کند.**
5. **اگر رویداد مربوط به فشردن کلید ماوس باشد، از پای‌گیم می‌خواهیم که مختصات محل کلیک را بدست بیاورد و سپس آن را به مختصات خانه‌های بازی دوز تبدیل می‌کنیم و آن را به تابع TakeNextAction جهت تغییر وضعیت بازی به حالت جدیدتر (افزودن سمبل به بازی توسط انسان) پاس می‌دهیم و سپس توسط تابع CheckGameOver چک می‌کنیم که آیا بازی تمام شده است یا خیر.**
6. **پس از اتمام رویدادهای مربوط به انسان و اینکه او خانه مورد نظرش را تعیین کرد و pygame آن تغییرات را در پنجره گرافکی نمایش داد و پس از اینکه نوبت به کامپیوتر رسید، به اتفاقات مربوط به کامپیوتر می‌پردازیم:**
7. **در صورتی که بازیکن فعلی کامپیوتر باشد و بازی در حال اجرا باشد، بهترین حرکتی را که توسط هوش مصنوعی با الگوریتم درخت MINIMAX با بهره‌گیری از هرس آلفا-بتا (FindBestMove) که در قسمت‌های بعدی به آن می‌پردازیم انتخاب شده باشد را TakeNextAction می‌کنیم و مجددن بررسی می‌کنیم که آیا بازی تمام شده است یا نه، در آخر سر هم نتایج تغییرات حالت را در پنجره گرافیکی نمایش میدهیم.**

جمع بندی

در این بخش به بررسی توابع مهم بازی و نحوه کارکرد و تاثیرگذاری آن ها بر روی یکدیگر پرداختیم و اینکه نحوه مدل سازی بازی به دنیای کامپیوتر را نیز به صورت کامل تشریح کردیم. در بخش دیگر به بررسی الگوریتم پیاده سازی درخت Min-Max با بهره‌گیری از هرس آلفا-بتا خواهیم پرداخت.

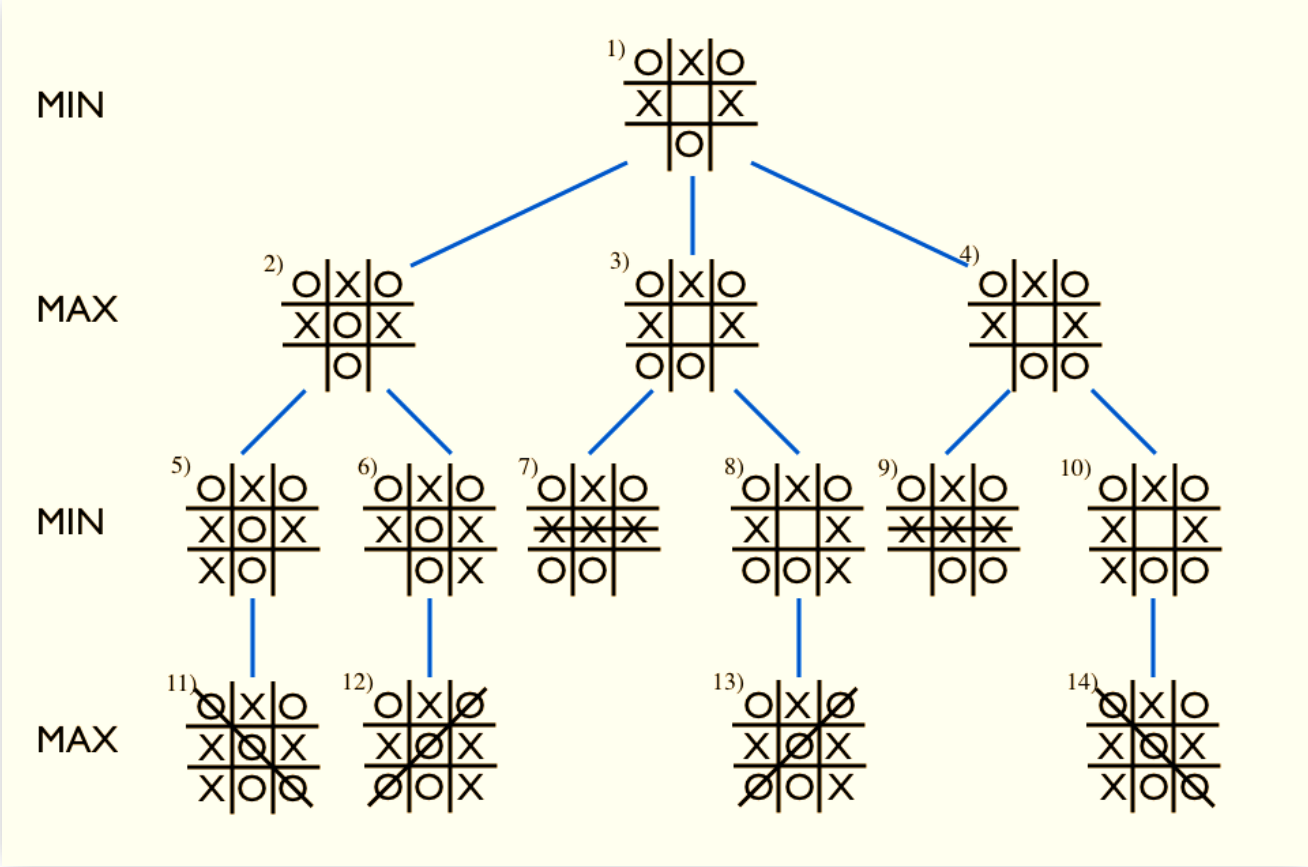
پیاده سازی و بررسی الگوریتم مین-مکس و هرس آلفا-بتا :

در این قسمت که مهمترین قسمت پروژه است پیاده سازی الگوریتم درخت MIN-MAX با **هرس** Alpha Beta - را مورد بررسی قرار خواهیم داد و نحوه انتخاب بهترین حرکت بعدی دوز توسط هوش مصنوعی کامپیوتر را شرح خواهیم داد:

توجه: الگوریتم MinMAx بدون استفاده از Alpha-Beta Pruning نیز موجود می‌باشد.

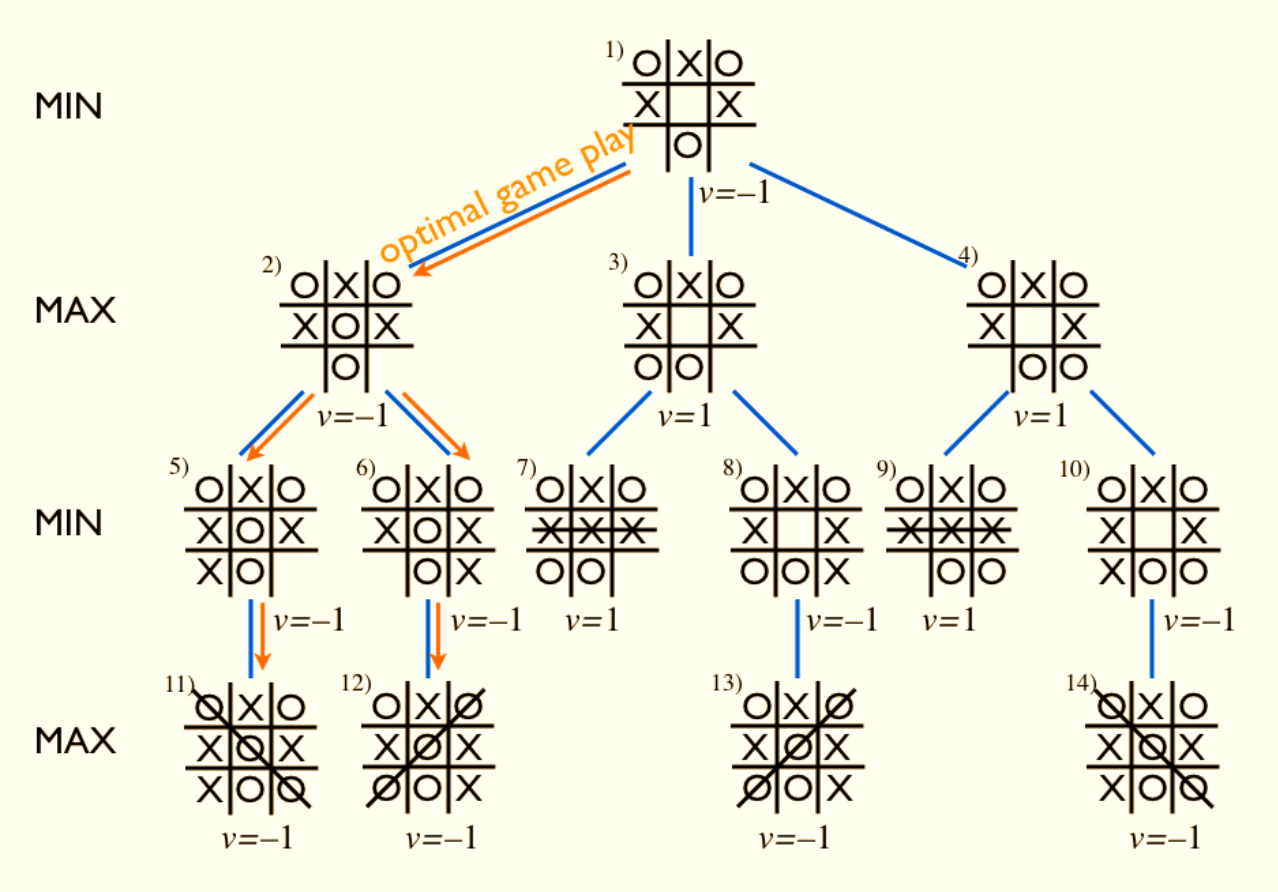
توضیح اجمالی الگوریتم مین-مکس

بخشی از درخت بازی مین-مکس این بازی به صورت زیر است:



در این بازی ، در یک سمت **هوش مصنوعی** که همان **O** است قرار دارد که بازیکن **MIN** کننده است و در سمت دیگر بازی انسان که دارای سمبل X است قرار دارد و نقش MAX را ایفا می‌کند. هوش مصنوعی همواره سعی دارد تا کمترین امتیاز را که منجر به برد خودش و باخت انسان می‌شود را انتخاب کند.
در تصویر بالا، یک حالت از درخت بازی را می‌بینید، در حالت ۱، نوبت بازیکن مینیمم کننده(کامپیوتر) است.
او حالات مختلفی از تخته دوز را به صورت **بازگشتی** و **DFS** تولید می‌کند و سعی میکند که کمترین امتیازی را که از دید نظر انسان است را **انتخاب** و آن روند را بازی کند. این موضوع به خوبی در تصویر بعدی نشان داده شده است:

پارسا یوسفی نژاد

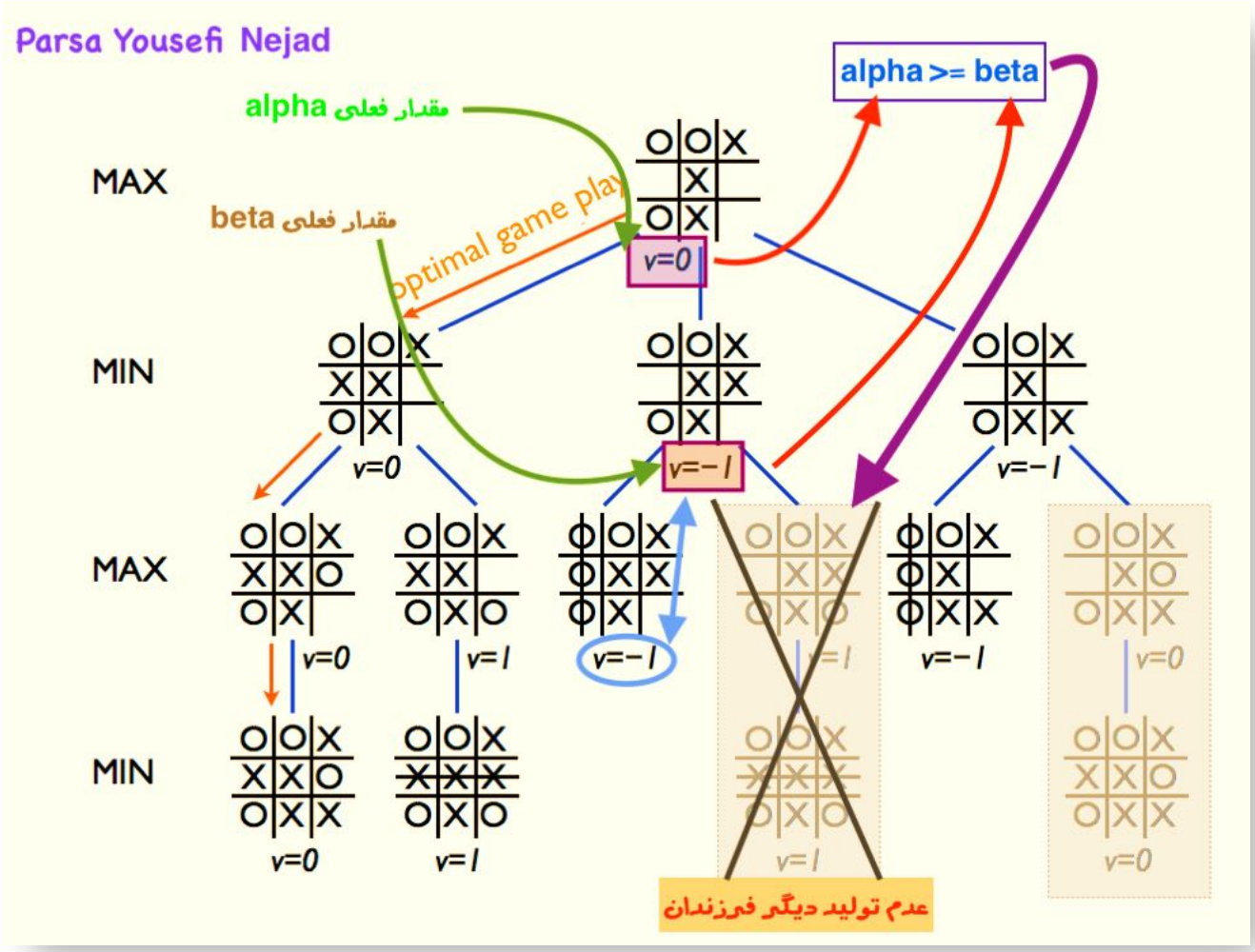


در این بازی ارزش حالات پایانی می‌توانند مقادیر **0 (تساوی)**- 1 **(برد انسان)** - 1- **(برد هوش مصنوعی)** باشند.
همانطور که در این تصویر می‌بینید نود مینیمم ریشه همواره سعی می‌کند که بازی‌ای را بازی کند که کمترین امتیاز را به انسان بدهد از این رو او پس از پیمایش کل حالات این درخت به صورت جستجوی اول عمق، نود فرزندی را بازی می‌کند که منجر به **کمترین ارزش** می‌شود.

توضیح اجمالی الگوریتم هرس آلفا-بتا

همانطور که از پیش می‌دانیم، در نود های ماکسیمم کننده(انسان) بیشترین ارزش فعلی alpha نامیده می‌شود و در نودهای مینیمم کننده(هوش مصنوعی) کمترین ارزش فعلی beta نامیده می‌شود.

هنگام اجرای الگوریتم مین-مکس در حالات MaximizerTurn در صورتی **که امتیاز تولید شده فعلی**(به صورت بازگشتی) **بزرگتر مساوی امتیاز beta** از نود مینیمایز کننده بالاتر باشد، در آن صورت دیگر **نیازی** به تولید و پیمایش فرزندان دیگر نود مکسیمایزر نمی‌باشد چرا که دیگر بازیکن انسان **نمیتواند امتیاز بیشتری** نسبت به beta **به دست بیاورد**، به طور مشابه در نود not MaximizerTurn که برای بازیکن کامپیوتر است، alpha نود قبل بزرگتر مساوی بتا نود فعلی باشد، دیگر نیازی به پیمایش و تولید نودهای دیگر این حالت نخواهد بود، چرا که هیچ‌گاه قرار نیست آلفا مقدارش کمتر از بتا نود پایین تر باشد، به عبارت دیگر بتا مقدارش می‌تواند کمتر شود اما آلفا دیگر هیچگاه مقداری را که مقدار خودش را کمتر کند انتخاب نمی‌کند و از این رو دیگر نیازی به تولید سایر نودها نبوده چرا که به ازای هیچ بتا دیگری، مقدار آلفا تغییر پیدا نخواهد کرد. در ابتدا **مقادیر آلفا و بتا به ترتیب 1 و 1-** مقدار دهی می‌شوند. همچنین باید توجه شود که مقادیر آلفا تنها در نودهای مکس کننده و مقادیر بتا تنها در نودهای مینیمم کننده باید آپدیت بشوند، این موضوع به وضوح در تصویر فوق قابل بررسی است:



در عمل حتی در فضاهای حالت کوچکی چون بازی دوز، استفاده از الگوریتم هرس آلفا-بتا موجب کاهش **چشمگیر** تعداد فضای حالت (کاهش مرتبه پیچیدگی فضایی) و افزایش سرعت اجرای برنامه(کاهش پیچیدگی مرتبه زمانی) در **عمل** می‌شود.

حال به نحوه پیاده سازی الگوریتم‌ها در عمل می‌پردازیم:

- **FindBestMove_MINIMAX_ABPruning()**

این تابع همانطور که از نامش مشخص است، بهترین حرکت را برای بازیکن COMPUTER با استفاده از الگوریتم درخت MINIMAX و استفاده از تکنیک هرس کردن Alpha-Beta پیدا میکند و مختصات بهترین حرکت ممکن را برمی‌گرداند.
این تابع با استفاده از تکنیک برنامه نویسی بازگشتی **اول عمق یا همان DFS** پیاده سازی شده است. لذا نیازمند شرط توقف بوده تا از عمقی معین که همان حالت‌های **برگ** است (همان تعداد خانه‌های بازی دوز) فراتر نرود و در چرخه تکرار بی‌نهایت گیر نکند. این شرط توقف را بررسی وجود دوز در بازی تعریف میکنیم، به این نحو این مورد را بررسی می‌کنیم:

1. **حالت فعلی تخته را CheckWin می‌کنیم و آن را درون متغیر winner ذخیره می‌کنیم که نشان‌دهنده بازیکن برنده (در صورت وجود است).**
2. **اگر winner انسان بود، در آن صورت دیگر نیازی به انجام حرکت دیگری نبوده (bestMove = None) و امتیاز این حالت نهایی را عدد 1+ برگرداند.**
3. **اگر winner کامپیوتر بود، در این صورت هم نیازی به انجام حرکت دیگری نیست ولی امتیاز این برگ برابر 1- خواهد بود.**
4. **در صورتی که برنده‌ای نداشته باشیم winner = 0، آنگاه امتیاز این حالت را 0 مبنی بر تساوی برگرداند.**

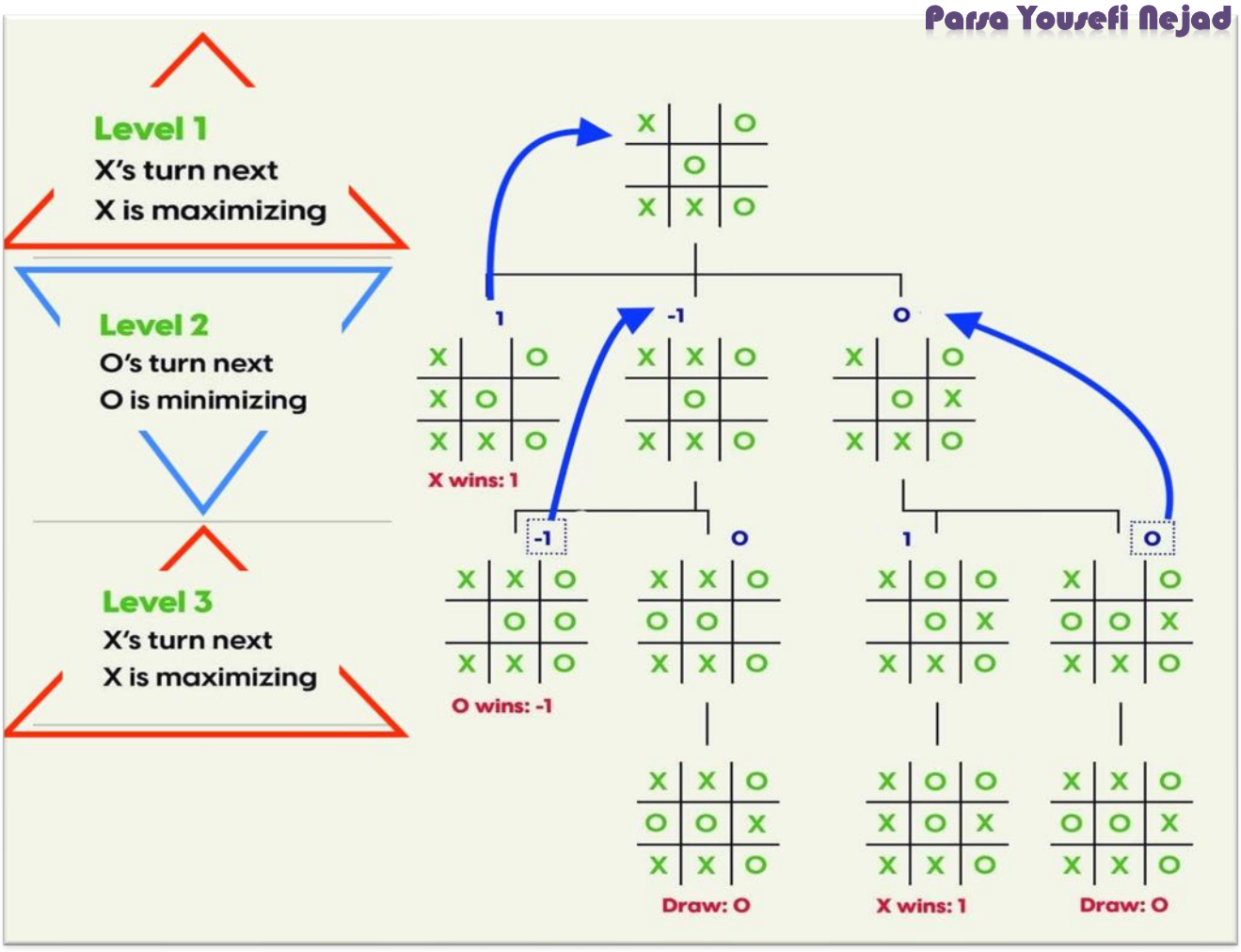
حال پس از بررسی حالات بازگشت نهایی، باید بررسی کنیم که بازیکنی که این تابع را فراخوانی کرده است، چگونه نودی بوده است، آیا بازیکن مکس‌کننده یا همان انسان است(maximizerTurn)؟ و اینکه نود مینیمم کننده کامپیوتر است. پس از تعیین این مورد که بهترین حرکت را می‌خواهیم برای کدام نوبت از بازیکنان انجام بدهیم باید عملیات خاصی را که منجر به انتخاب بهترین امتیاز برای هر نود می‌شود را انجام دهیم، در ابتدا فرض میکنیم که **بازیکنی که این تابع را فراخوانی کرده است، کامپیوتر** بوده است، باید این دستورات را برای یافتن بهترین حرکت برای آن بازیکن انجام بدهیم:

1. **در ابتدا متغیر bestMove را برابر None مقدار دهی می‌کنیم و لیست خانه‌های خالی که امکان گذاشتن سمبل جدید برای کامپیوتر در آن ممکن است را تهیه می‌کنیم.**
2. **به ازای هر یک از خانه‌های خالی تخته دوز، یک تخته ازمایشی موقت (tiralBoard) را می‌سازیم و بررسی می‌کنیم که آیا گذاشتن سمبل O در آن خانه‌ها ما را به امتیاز حداقلی از نظر بازیکن انسان می‌رساند و یا نه، این کار را به صورت بازگشتی برای آن تخته آزمایشی و فرزندانش به صورت جستجوی عمق اول انجام می‌دهیم و هنگامی که در آخر این فراخوانی‌های بازگشتی به یک حالت نهایی رسیدیم، با جایگذاری‌های متوالی امتیاز نهایی که برای آن حالت trial به دست آمده را درون متغیر thisScore جایگذاری می‌کنیم و سپس آن را با متغیر beta همین متغیر که در ابتدای فراخوانی مقدار پیش‌فرض 1+ (یا هر مقدار بزرگتر) را داشته است مقایسه می‌کنیم و در صورتی که مقدار thisScore به دست آمده کمتر از beta باشد، beta را آپدیت میکنیم.**
3. **سپس بررسی می‌کنیم که آیا آلفا حالت قبلی (پیشفرض = 1-) بزرگتر مساوی این beta می‌باشد یا خیر،**

اگر این شرط برقرار بود، بدین معنا است که آلفا دیگر قرار نیست که مقدارش بخاطر وجود چنین betaای کمتر از مقدار سابقش بشود و همچنین چون که beta قرار نیست که مقدارش بیشتر شود تا بتواند مقدار alpha را تغییر بدهد، هرس (pruning) یا همان Cut Off رخ می‌دهد و تا حد زیادی از بررسی حالات دیگری که بررسی آن‌ها هیچ ضرورتی ندارد جلوگیری میشود و در زمان اجرای کد تا حد زیادی صرفه‌جویی می‌شود.

4. **در اینجا ذکر این مورد هم خالی از لطف نیست: تنها در چنین بازی خاصی بدلیل وجود ۳ نوع امتیاز میتوانیم یک ترفند سریعتر از هرس آلفا-بتا انجام بدهیم و آن هم Fast Cut Off می‌باشد، در این مورد در نود مینیمم کننده اگر دیدیم که امتیاز فعلی برابر 1- شد، در آن صورت می‌توانیم به سادگی بگوییم که بهترین پیمایش را از نظر بازیکن کامپیوتر که همان 1- است را پیدا کردیم و میتوانیم سایر حالات دیگر را بررسی نکنیم و بهترین بازی را در زمان بسیار کمتری انجام بدهیم.**
5. **در آخر نیز پس از بدست آوردن تمام امتیازات فرزندان و تعیین beta بهترین حرکت را که از همان خانه‌های خالی بدست آمده و همچنین امتیاز beta را به عنوان خروجی به بخش کامپیوتر بازی می‌دهیم تا آن را در مقابل انسان بازی کند.**

بخش زیادی از فرایندهای قسمت بازیکن ماکسیمم کننده (X) با بازیکن مینیمم کننده(کامپیوتر) **مشترک** بوده و تنها **تفاوت اصلی** این قسمت با قسمت قبلی در **نحوه برخورد با امتیاز thisScore** بدست آمده است. در حالت بازیکن ماکسیمم، ما با alpha و beta قبلی سروکار داریم و مقدار alpha را در صورتی که thisScore مقداری بیشتر از alpha داشته باشد، آپدیت می‌کنیم و به راحتی وجود امکان هرس alpha-beta را بررسی می‌کنیم، و سپس بهترین حرکت ممکن را از نظر بازیکن انسان که ماکسیمم کننده است برمی‌گردانیم. تصویر فوق کمک شایانی در فهم مسائل گفته شده می‌کند:



جمع‌بندی پروژه Tic Tac Toe

پروژه پیاده‌سازی بازی دوز از جلمه نمونه‌ها رایج و معروف در دنیای نظریه بازی ها و هوش مصنوعی می‌باشد، ما در این پروژه به بررسی صفر تا صد نحوه پیاده‌سازی بازی دوز تک نفره انسان و هوش‌مصنوعی با کمک الگوریتم‌های هوش مصنوعی درخت MiniMax و استفاده از الگوریتم Alpha-Beta Pruning پرداختیم و موفق به پیاده سازی یک بازی واقعی گرافیکی و استفاده کردن از کاربردهای واقعی درس هوش مصنوعی در یکی از بازی‌های محبوب دنیای واقعی شویم.

در ابتدای این پروژه به بررسی نحوه پیاده‌سازی گرافیک بازی با کمک کتابخانه pygame شدیم و بعد از آن تمام توابع مورد نیاز برای اجرای بازی را به طور مفصل مورد بررسی قرار دادیم و سپس از آن شروع به بررسی اجمالی الگوریتم‌های فوق کردیم و هر یک را با مثال‌ها و تصاویر مختلف مورد تحقیق قرار دادیم و در نهایت الگوریتم MiniMax + Alpha-Beta Pruning را به طور کامل پیاده‌سازی و بررسی کردیم؛ در نهایت با انجام تمام مراحل فوق موفق به ساخت یک بازی گرافیکی جذاب تک‌نفره به زبان Python و استفاده از الگوریتم‌های هوش‌مصنوعی شدیم.

by Parsa Yousefi Nejad