به نام خدا

پروژه پایانی درس فرآیندهای کاربردی دکتر علیشاهی

دانشکده علوم ریاضی دانشگاه شریف

عنوان پروژه: حل و شبیه‌سازی مسائل مربوط به بازی دوز(تیک تک تو) در نسخه‌های متفاوت

دانشجو: محمد سوری

تابستان 1402

مقدمه

هدف پروژه، شبیه‌سازی و تحلیل نسخه‌ای تصادفی از بازی دوز یا tic tac toe است. می‌توان نسخه‌های مختلفی از بازی را تحلیل کرد و به فرم یک MDP مدل کرد. مثلا بازی در مقابل یک رقیب تصادفی که بین خانه‌های خالی موجود، یک خانه را با اندازه احتمال مشخصی انتخاب می‌کند. یا بازی در حضور ریسک خانه‌های خالی. یا بازی در حضور ریسک تمام خانه‌ها(خانه‌های پر هم ممکن است بعد از تعدادی حرکت تصادفی خالی شوند). همچنین در ادامه، توسعه‌های دیگری از پروژه نظیر بزرگ‌کردن صفحه بازی، بازی در فضای سه بعدی و بازی با بیش از یک رقیب هم ممکن خواهد بود. در این پروژه، تمرکز ما روی حل دو نسخه از بازی است. ابتدا بازی تصادفی دو بعدی سه در سه، و سپس بازی تصادفی سه بعدی سه در سه در سه.

در این گزارش، در ابتدا ساده‌ترین نسخه از بازی را تحلیل کرده و معادله بلمن را برای آن‌ها خواهیم نوشت و تلاش می‌کنیم با روشی عددی، تابع ارزش تمام حالت‌های فضای حالت را محاسبه کنیم. سپس نسخه سه بعدی بازی را با معرفی چند الگوریتم یادگیری تقویتی برای همگرایی سریع‌تر حل می‌کنیم و نتایج را تحلیل می‌کنیم. در پایان، به سراغ نسخه‌ای از بازی در فضای دو بعدی ولی بزرگ‌تر از فضای 3 در 3 خواهیم رفت.

نسخه دو بعدی سه در سه

فرض کنیم تخته‌ی بازی ما، یک تخته‌ی 3 در 3 به فرم زیر باشد و برای هریک از خانه‌ها هم یک شماره انتخاب کرده‌ایم:

1	2	3
4	5	6
7	8	9

فضای حالت

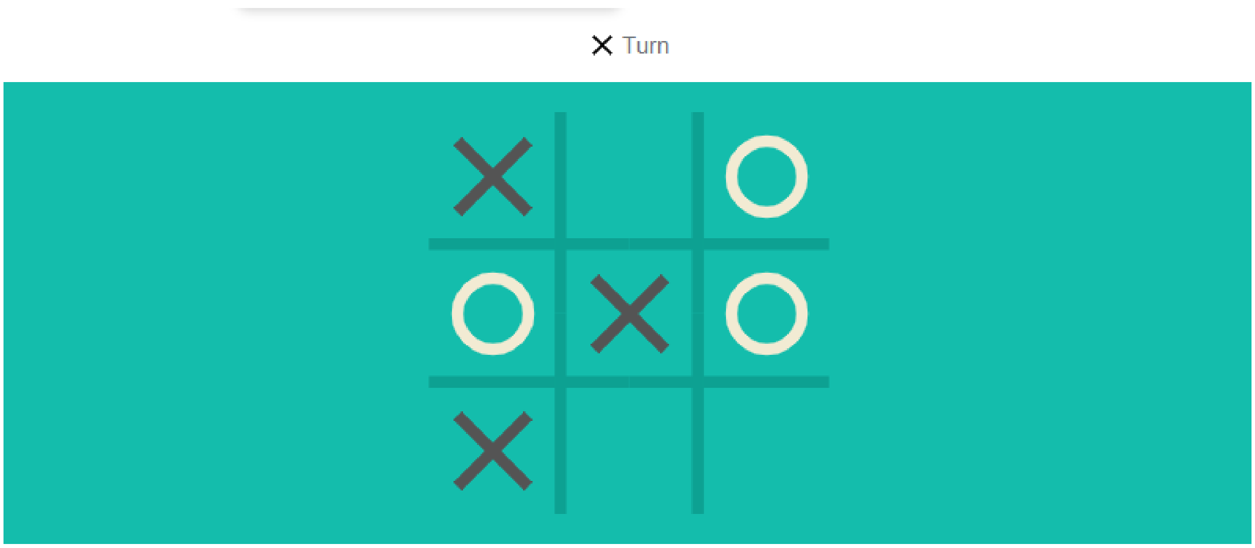
در تمامی مدل‌ها و نسخه‌های مختلف بازی، فضای حالت ما زیرمجموعه‌ای از فضایی به فرم زیر است:

$$S = \{\ (b, t) |\ b \in \{E, X, O\}^{size}\ and\ t \in \{X, O\}\ \}$$

در نمادگذاری بالا، منظور از b یک وضعیت تخته است که در قالب یک بردار *size*تایی نمایش داده می‌شود. اعضای این بردار، می‌توانند یکی از 3 عضو *E,X,O* باشند که *E* به معنی خالی بودن آن خانه، *X* به معنی این است که نماد *X* در آن خانه قرار گرفته است و *O* هم به معنی این است که نماد *O* در آن خانه قرار گرفته است. *size* در واقع عددی ثابت است که اندازه صفحه را نمایش می‌دهد و در بازی فعلی *size=9* است. در بازی سه بعدی *3*3*3* مقدار آن برابر با 27 خواهد

بود. همچنین منظور از t نوبت بازیکنی است که باید حرکت بعدی را انجام بدهد. اگر X باشد یعنی نوبت بازیکن X است و اگر O باشد هم یعنی نوبت بازیکن O است.

به عنوان مثال تختهی زیر را در نظر بگیرید و توجه کنید که نوبت X است.



وضعیت این تخته و نوبت عبارت است از:

$$s = ([X, E, O, O, X, O, X, E, E], X)$$

پس سایز فضای حالت در این نسخهی خاص که در حال بررسی آن هستیم در مجموع عددی کوچکتر یا مساوی از $3^9 * 2 = 39366$ خواهد بود.

فضای اکشن‌ها و اکشن‌های مجاز

در بازی در صفحه‌ی 3 در 3، هر بازیکن در هر حالتی، نهایتا 9 اکشن مجاز دارد که معادل با انتخاب یکی از 9 خانه‌ی صفحه معادل با شماره‌گذاری‌ای است که قبل‌تر آمد. پس:

$$A = \{i \mid i \in \{1,2,3,4,5,6,7,8,9\}\}$$

ولی واضح است که در هر حالت، هر اکشنی مجاز نیست. به همین خاطر، فضای اکشن‌های مجاز هر حالت مثل s را به شکل زیر تعریف می‌کنیم:

$$A_{s|s=(b,t)} = \{i \in A \mid b(i) = E\}$$

که البته می‌توان این تعریف را به فرم زیر هم نوشت:

$$A_s = \{i \in A \mid s(1)(i) = E\}$$

که *s(1)* همان مولفه اول s یا b است و *s(2)* همان مولفه دوم s یا t است.

تابع پاداش

برای معرفی تابع پاداش، ابتدا حالات برنده و بازنده را شناسایی می‌کنیم. در تمام طول بازی، فرض را بر این می‌گذاریم که X بازیکن بیشینه‌ساز یا ماکزیمم‌کننده‌ی امتیازات است و O بازیکن کمینه‌ساز یا مینیمم‌کننده امتیازات.

اگر صفحه را با یک 9-بردار مثل b نمایش دهیم، واضح است که حالت‌های برنده برای هریک از دو بازیکن، حالت‌های زیر هستند:

$$\begin{aligned} W_X = \{s = (b,t) \in S \,|b(1) = b(2) = b(3) = X \; or \; b(4) = b(5) \\ = b(6) = X \; or \; b(7) = b(8) = b(9) = X \; or \; b(1) = b(4) \\ = b(7) = X \; or \; b(2) = b(5) = b(8) = X \; or \; b(3) = b(6) \\ = b(9) = X \; or \; b(1) = b(5) = b(9) = X \; or \; b(3) = b(5) \\ = b(7) = X \,\} \end{aligned}$$

$$\begin{aligned} W_O = \{s = (b,t) \in S \,|b(1) = b(2) = b(3) = O \; or \; b(4) = b(5) \\ = b(6) = O \; or \; b(7) = b(8) = b(9) = O \; or \; b(1) = b(4) \\ = b(7) = O \; or \; b(2) = b(5) = b(8) = O \; or \; b(3) = b(6) \\ = b(9) = O \; or \; b(1) = b(5) = b(9) = O \; or \; b(3) = b(5) \\ = b(7) = O \,\} \end{aligned}$$

همچنین به این خاطر که حالت‌های تساوی نیز ارزشی در ادامه کار نخواهند داشت و حالت ترمینال هستند، خوب است آن‌ها را هم در یک مجموعه تعریف کنیم:

$$\begin{aligned} D = \{s = (b,t) \in S \,|(\forall i\colon i \in \{1,2,3,4,5,6,7,8,9\}\colon b(i) = X \; or \; b(i) \\ = O) \; and \; s \notin \; W_X \cup W_O\} \end{aligned}$$

حال تابع پاداش را به شکل زیر تعریف می‌کنیم:

$$R : S \to \{-1,0,1\}$$

$$R(s) = \begin{cases} 1 \ \ if\ s \in W_X \\ -1\ if\ s\ \ \in W_O \\ 0\ otherwise \end{cases}$$

یعنی ایجنت در تمام حالاتی که X برنده می‌شود، پاداش 1، در تمام حالاتی که O برنده می‌شود، پاداش 1- و در تمام حالات دیگر، پاداش 0 می‌گیرد.

چند عملگر کمکی

پیش از رسیدن به مسائل اصلی و به دست آوردن معادلات بلمن، بد نیست چند عملگر کمکی که جلوتر به دردمان خواهد خورد هم معرفی شوند.

عملگر اول، عملگر تغییر نوبت است که به صورت زیر کار می‌کند:

$$t' = \begin{cases} X \; if \; t = O \\ O \; if \; t = X \end{cases}$$

پس در واقع، عملگری است از فضای *{X,O}* به خود این فضا که ورودی را معکوس می‌کند.

عملگر بعدی، عملگری است که در یک نوبت مشخص، یک اکشن مجاز مشخص و یک وضعیت صفحه‌ی مشخص را می‌گیرد و یک وضعیت صفحه‌ی جدید برمی‌گرداند که در آن، مارک بازیکنی که نوبتش بوده است را در خانه‌ی مربوط به آن اکشن قرار داده است. تعریف دقیق آن به شکل زیر است:

$$b_{a,t}^{same}(a) = t \; and \; b_{a,t}^{same}(i)_{i \neq a} = b(i)$$

عملگر مشابهی هم داریم که دقیقا مثل عملگر بالاست، فقط در خانه‌ی انتخاب شده با اکشن، مارک رقیب کسی که نوبتش است را قرار می‌دهد:

$$b_{a,t}^{reverse}(a) = t' \; and \; b_{a,t}^{reverse}(i)_{i \neq a} = b(i)$$

به کمک این عملگرها، به دو عملگر زیر روی تمام فضای حالات می‌رسیم.

$$\begin{gathered} if \; s = (b,t) \rightarrow s_{a,t}^{same} = \left(b_{a,t}^{same}, t'\right) \\ if \; s = (b,t) \rightarrow s_{a,t}^{reverse} = \left(b_{a,t}^{reverse}, t'\right) \end{gathered}$$

در واقع در هردو عملگر، نوبت عوض می‌شود و در اولی، صفحه یا بورد جدیدی داریم که در خانه‌ی مربوط به اکشن، مارک کسی که نوبتش بوده قرار گرفته و در دومی، صفحه جدیدی داریم که مارک رقیب در آنجا قرار گرفته است.

اصلاحی بر اکشن‌های مجاز و فضای حالت

حالا که حالت‌های نهایی را بررسی کردیم، باید به این نکته توجه کنیم که اگر حالتی نهایی باشد، هیچ اکشنی برای هیچ ایجنتی در آن دیگر مجاز نیست. پس به اکشن‌های مجاز جدیدی بر اساس قوانین پایان بازی می‌رسیم که باید آن را تعریف کنیم.

برای این کار، ابتدا حالت‌های نهایی یا ترمینال را تعریف می‌کنیم.

$$T = \{s \in S | \; s \in W_X \; or \; s \in W_O \; or \; s \in D\}$$

حال می‌توان اکشن‌هایی که از قوانین حالات نهایی هم پیروی می‌کنند و کاملا مجاز هستند را به فرم زیر تعریف کرد:

$$\mathrm{AT}_{\mathrm{s=(b,t)}} = \{\mathrm{a} \in \mathrm{A_s} | \; \mathrm{s} \notin \mathrm{T}\}$$

همچنین در تعریف فضای حالت هم باید دقت کنیم حالت‌هایی که هر دو بازیکن در آن برنده هستند، حالت‌های مجاز نخواهند بود و باید از فضای حالت حذف شوند. پس فضای حالت جدید را به شکل زیر در نظر می‌گیریم:

$$S' = \{s \in S \mid s \notin W_X \cap W_O\}$$

مدل‌سازی بازی و معادله بلمن

در ادامه، بازی در حضور ریسک برای خانه‌ها را مدل‌سازی خواهیم کرد و معادله بلمن را برای تابع ارزش حالت‌های آن خواهیم نوشت. سپس، روشی عددی برای محاسبه تابع ارزش ارائه خواهیم کرد.

بازی در حضور ریسک

فرض کنیم یک 9-بردار مثل ***P*** داریم که در واقع احتمال خطر و ریسک هر خانه را نمایش می‌دهد. بنابراین، برای هر خانه‌ی ***i***، اگر بازیکن ***t*** اکشن مجاز ***i*** را انتخاب کند، با احتمال ***1-P(i)*** علامتی که در خانه‌ی تا پیش از این خالی ***i*** قرار می‌گیرد، علامت ***t*** خواهد بود و با احتمال ***P(i)*** علامت ***t'*** خواهد بود.

معادله بلمن

فرض کنیم تابع ***v*** با تعریف زیر، تابع ارزش هریک از حالت‌های بازی باشد:

$$v: S' \to R$$

با توجه به تعاریفی که از تابع‌های پاداش و اکشن‌های مجاز و عملگرهای کمکی آمد، می‌توان معادله بلمن را برای هر حالت ***s*** به فرم زیر نوشت:

$$for\ s = (b,t) \in S' |\ s \notin W_X \cup W_O \cup D:$$

$$v(s) = \begin{cases} \max\limits_{a \in AT_s} \begin{matrix} P(a) * \left(R\left(s_{a,t}^{reverse}\right) + v\left(s_{a,t}^{reverse}\right)\right) + \\ \left(1 - P(a)\right) * \left(R\left(s_{a,t}^{same}\right) + v\left(s_{a,t}^{same}\right)\right) \end{matrix} & if\ t = X \\ \min\limits_{a \in AT_s} \begin{matrix} P(a) * \left(R\left(s_{a,t}^{reverse}\right) + v\left(s_{a,t}^{reverse}\right)\right) + \\ \left(1 - P(a)\right) * \left(R\left(s_{a,t}^{same}\right) + v\left(s_{a,t}^{same}\right)\right) \end{matrix} & if\ t = O \end{cases}$$

و البته با توجه به اینکه در هر استیت پایانی یا ترمینال، دیگر ارزشی به دست نخواهد آمد، تابع ارزش در آن حالت‌ها صفر است، پس:

$$for\ s = (b,t) \in S' |\ s \in W_X \cup W_O \cup D:$$
$$v(s) = 0$$

پس در واقع می‌توان گفت یک فضای توابع مثل V به فرم زیر داریم:

$$V = \{v: S' \to R : v\ is\ a\ function\}$$

و بعد یک عملگر مثل T به فرم زیر داریم:

$$T: V \to V$$

که

$$T(v(s)) = \begin{cases} \max\limits_{a \in AT_s} \begin{matrix} P(a) * \left(R\left(s_{a,t}^{reverse}\right) + v\left(s_{a,t}^{reverse}\right)\right) + \\ \left(1 - P(a)\right) * \left(R\left(s_{a,t}^{same}\right) + v\left(s_{a,t}^{same}\right)\right) \end{matrix} & if\ t = X \\ \min\limits_{a \in AT_s} \begin{matrix} P(a) * \left(R\left(s_{a,t}^{reverse}\right) + v\left(s_{a,t}^{reverse}\right)\right) + \\ \left(1 - P(a)\right) * \left(R\left(s_{a,t}^{same}\right) + v\left(s_{a,t}^{same}\right)\right) \end{matrix} & if\ t = O \end{cases}$$

و تابع ارزش بهین، تابعی از فضای توابع V است که نقطه ثابت این عملگر باشد یا

$$T(v^*) = v^*$$

بررسی انقباضی بودن عملگر و ارائه روش عددی برای value iteration

در ادامه در انقباضی بودن T بحث می‌کنیم.

اولا که باید در نظر بگیریم که فقط توابعی را انتخاب خواهیم کرد و در نظر خواهیم گرفت که مقدار آن‌ها در استیت‌های ترمینال یا عضو T برابر با 0 باشد.

برای اثبات انقباضی بودن T باید ثابت کنیم تحت نرم بی‌نهایت یا نرم سوپریمم، ثابت k بین 0و1 چنان وجود دارد که:

$$\forall\, v_1, v_2 \in V : \|T(v_1) - T(v_2)\|_\infty \leq k.\, \|v_1 - v_2\|_\infty$$

که تعریف نرم بی‌نهایت هم عبارت است از:

$$\|v_1 - v_2\|_\infty = \max_{s \in S'} |v_1(s) - v_2(s)|$$

چون فضایی که توابع عضو V روی آن تعریف شده‌اند، متناهی هستند، می‌توان به راحتی شرط را در هر بار اعمال T به شکل عددی در کد بررسی کرد. ولی در ادامه از نظر ریاضیاتی آن را بررسی خواهیم کرد.

دو حالت را در نظر می‌گیریم. ابتدا حالاتی را در نظر می‌گیریم که توابع روی یک عضو از S' مثل s اعمال می‌شوند که $s=(b,t)$ و $t=X$

بنابراین، عملگر T همیشه ماکزیمم‌گیری می‌کند. پس در این حالات داریم برای هر s اینچنینی:

- اگر $s \in W_X \cup W_O \cup D$

  در این حالت به وضوح:

$$\left|T\big(v_1(s)\big) - T\big(v_2(s)\big)\right| = |0 - 0| = 0 \leq \|v_1 - v_2\|_\infty$$

- اگر $s \notin W_X \cup W_O \cup D$

$$
\begin{aligned}
&\left|T\big(v_1(s)\big)-T\big(v_2(s)\big)\right| \\
&\qquad=\left|\max_{a\in AT_s} p(a)*\Big(R\big(s_{a,t}^{reverse}\big)+v_1\big(s_{a,t}^{reverse}\big)\Big)+\big(1-p(a)\big)\right. \\
&\qquad*\Big(R\big(s_{a,t}^{same}\big)+v_1\big(s_{a,t}^{same}\big)\Big) \\
&\qquad-\max_{a\in AT_s} p(a)*\Big(R\big(s_{a,t}^{reverse}\big)+v_2\big(s_{a,t}^{reverse}\big)\Big)+\big(1-p(a)\big) \\
&\qquad\left.*\Big(R\big(s_{a,t}^{same}\big)+v_2\big(s_{a,t}^{same}\big)\Big)\ \right| \\
&\qquad=|\max_{a\in AT_s}\Big(P(a)*R\big(s_{a,t}^{reverse}\big)+P(a)*v_1\big(s_{a,t}^{reverse}\big) \\
&\qquad+\big(1-P(a)\big)*R\big(s_{a,t}^{same}\big)+\big(1-P(a)\big)*v_1\big(s_{a,t}^{same}\big) \\
&\qquad-P(a)*R\big(s_{a,t}^{reverse}\big)-P(a)*v_2\big(s_{a,t}^{reverse}\big)-\big(1-P(a)\big) \\
&\qquad*R\big(s_{a,t}^{same}\big)-\big(1-P(a)\big)*v_2\big(s_{a,t}^{same}\big)\Big| \\
&\qquad=|\max_{a\in AT_s}(P(a)*(v_1\big(s_{a,t}^{reverse}\big)-v_2\big(s_{a,t}^{reverse}\big)) \\
&\qquad+\big(1-P(a)\big)*\Big(v_1\big(s_{a,t}^{same}\big)-v_2\big(s_{a,t}^{same}\big)\Big)\,|
\end{aligned}
$$

پس داریم:

$$
\begin{aligned}
&\left|T\big(v_1(s)\big)-T\big(v_2(s)\big)\right| \\
&\qquad=\left|\max_{a\in AT_s}(P(a)*(v_1\big(s_{a,t}^{reverse}\big)-v_2\big(s_{a,t}^{reverse}\big))\right. \\
&\qquad\left.+\big(1-P(a)\big)*\Big(v_1\big(s_{a,t}^{same}\big)-v_2\big(s_{a,t}^{same}\big)\Big)\right| \\
&\qquad\le\max_{a\in AT_s}|\Big(P(a)*\Big(v_1\big(s_{a,t}^{reverse}\big)-v_2\big(s_{a,t}^{reverse}\big)\Big) \\
&\qquad+\ \big(1-P(a)\big)*\Big(v_1\big(s_{a,t}^{same}\big)-v_2\big(s_{a,t}^{same}\big)\Big)\ \Big| \\
&\qquad\le\max_{a\in AT_s}(P(a)*\left|v_1\big(s_{a,t}^{reverse}\big)-v_2\big(s_{a,t}^{reverse}\big)\right|+\big(1-P(a)\big) \\
&\qquad*|v_1\big(s_{a,t}^{same}\big)-v_2\big(s_{a,t}^{same}\big)|) \\
&\qquad\le\max_{a\in AT_s}(P(a)*\|v_1-v_2\|_\infty+\big(1-P(a)\big)*\|v_1-v_2\|_\infty) \\
&\qquad=\|v_1-v_2\|_\infty
\end{aligned}
$$

در حالتی که *t=O* باشد استدلال به شکل کلی مشابه است، فقط در یک قدم نامساوی یک تبدیل از *min* به *max* با قرار دادن علامت منفی داریم. در واقع از این موضوع استفاده می‌کنیم که اگر *U* لیستی از اعداد باشد، آنگاه داریم:

$$|\min U| = |\max -U| \leq \max|-U| = \max|U|$$

پس استدلال کامل در این حالت به قرار زیر است که:

- اگر $s \in W_X \cup W_O \cup D$

  در این حالت به وضوح:

$$\left|T\left(v_1(s)\right) - T\left(v_2(s)\right)\right| = |0 - 0| = 0 \leq \|v_1 - v_2\|_\infty$$

- اگر $s \notin W_X \cup W_O \cup D$

در این حالت، عینا مشابه حالت قبل رفتار خواهیم کرد. فقط در یکی از مراحل از نتیجه‌ی زیر استفاده می‌کنیم که:

$$|\min U| \leq \max |U|$$

استدلال دقیق به قرار زیر است:

$$
\begin{aligned}
&\left|T\big(v_1(s)\big)-T\big(v_2(s)\big)\right| \\
&\qquad = \left|\min_{a\in AT_s} p(a)*\left(R\big(s_{a,t}^{reverse}\big)+v_1\big(s_{a,t}^{reverse}\big)\right)+\big(1-p(a)\big)\right. \\
&\qquad * \left(R\big(s_{a,t}^{same}\big)+v_1\big(s_{a,t}^{same}\big)\right) \\
&\qquad - \min_{a\in AT_s} p(a)*\left(R\big(s_{a,t}^{reverse}\big)+v_2\big(s_{a,t}^{reverse}\big)\right)+\big(1-p(a)\big) \\
&\qquad \left. * \left(R\big(s_{a,t}^{same}\big)+v_2\big(s_{a,t}^{same}\big)\right)\ \right| \\
&\qquad = |\min_{a\in AT_s}\big(P(a)*R\big(s_{a,t}^{reverse}\big)+P(a)*v_1\big(s_{a,t}^{reverse}\big) \\
&\qquad + \big(1-P(a)\big)*R\big(s_{a,t}^{same}\big)+\big(1-P(a)\big)*v_1\big(s_{a,t}^{same}\big) \\
&\qquad - P(a)*R\big(s_{a,t}^{reverse}\big)-P(a)*v_2\big(s_{a,t}^{reverse}\big)-\big(1-P(a)\big) \\
&\qquad * R\big(s_{a,t}^{same}\big)-\big(1-P(a)\big)*v_2\big(s_{a,t}^{same}\big)\big| \\
&\qquad = |\min_{a\in AT_s}(P(a)*(v_1\big(s_{a,t}^{reverse}\big)-v_2\big(s_{a,t}^{reverse}\big)) \\
&\qquad + \big(1-P(a)\big)*\left(v_1\big(s_{a,t}^{same}\big)-v_2\big(s_{a,t}^{same}\big)\right)|
\end{aligned}
$$

پس داریم:

$$
\begin{aligned}
&\left|T\big(v_1(s)\big)-T\big(v_2(s)\big)\right| \\
&\qquad = \left|\min_{a\in AT_s}(P(a)*(v_1\big(s_{a,t}^{reverse}\big)-v_2\big(s_{a,t}^{reverse}\big))\right. \\
&\qquad \left. + \big(1-P(a)\big)*\left(v_1\big(s_{a,t}^{same}\big)-v_2\big(s_{a,t}^{same}\big)\right)\right| \\
&\qquad \le \max_{a\in AT_s} |\left(P(a)*\left(v_1\big(s_{a,t}^{reverse}\big)-v_2\big(s_{a,t}^{reverse}\big)\right)\right. \\
&\qquad + \ \big(1-P(a)\big)*\left(v_1\big(s_{a,t}^{same}\big)-v_2\big(s_{a,t}^{same}\big)\right)\ \Big| \\
&\qquad \le \max_{a\in AT_s}(P(a)*\left|v_1\big(s_{a,t}^{reverse}\big)-v_2\big(s_{a,t}^{reverse}\big)\right|+\big(1-P(a)\big) \\
&\qquad * |v_1\big(s_{a,t}^{same}\big)-v_2\big(s_{a,t}^{same}\big)|) \\
&\qquad \le \max_{a\in AT_s}(P(a)*\|v_1-v_2\|_\infty+\big(1-P(a)\big)*\|v_1-v_2\|_\infty) \\
&\qquad = \|v_1-v_2\|_\infty
\end{aligned}
$$

پس با سوپریمم‌گیری از طرفین نسبت به *s* در تمام *S’* داریم:

$$\|T(v_1) - T(v_2)\|_\infty \leq 1. \|v_1 - v_2\|_\infty$$

اما یک مشکل این است که انقباضی بودن اکید نیست که البته چون فرآیند تصمیم مارکف، اپیزودیک بوده است و ضریب تنزیل نداشته‌ایم، طبیعی است. یعنی ضریب انقباض یا *k=1* است و بنابراین با قضیه‌ی نقطه ثابت، لزوما با اعمال متوالی *T* روی هر *v* دلخواه آغازین، به تابع ارزش واقعی نخواهیم رسید و ممکن است در یک حلقه بی‌نهایت گیر کنیم. ولی دست کم خیالمان راحت است که عملگر، ما را از تابع ارزش نهایی دور هم نمی‌کند.

در عمل و با قرار دادن *v* اولیه برای تمام حالت‌ها برابر با عدد 0(یعنی تابع ارزش اولیه را تابع ثابت صفر می‌گیریم)، با فقط ده بار تکرار، همگرایی رخ می‌دهد و به تابع ارزش واقعی می‌رسیم.

کد پایتون شبیه‌سازی تمام این پروسه در پیوست خواهد آمد.

پیاده‌سازی و نتایج بازی سه در سه

کد پیاده‌سازی شده برای این بازی به شرح زیر است:

```python
from ast import Continue
from numpy.core.multiarray import empty
import numpy as np
import random


def board_printer(b):
    # Verify if the vector has the correct size
    if len(b) != 9:
        print("Invalid board size!")
        return

    c=[]
    for i in range(9):
      if b[i]!='E':
        if b[i]=='X':
          c.append('X')
        else:
          c.append('O')
      else:
        c.append(' ')

    # Print the Tic Tac Toe board
    print(" " + c[0] + " | " + c[1] + " | " + c[2] + " ")
    print("---+---+---")
    print(" " + c[3] + " | " + c[4] + " | " + c[5] + " ")
    print("---+---+---")
    print(" " + c[6] + " | " + c[7] + " | " + c[8] + " ")

def check_win(board):
    # Define all possible winning combinations
    winning_combinations = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # Rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # Columns
        [0, 4, 8], [2, 4, 6]               # Diagonals
    ]
```

```python
    # Check if any player has won
    for combination in winning_combinations:
        if board[combination[0]] == board[combination[1]] ==
board[combination[2]] != 'E':
            return board[combination[0]]  # Return the winning player ('X'
or 'O')

    # Check if the board is full (draw)
    if 'E' not in board:
        return 'Draw'

    # If no player has won and the board is not full, continue the game
    return 'Continue'


def next_turn(turn):
  #Gets a turn and returns its erverse
  if turn=='X':
    return 'O'
  else:
    return 'X'

def A(board,action):
  #Checks if an action is possible and legal on a board
  if board[action]!='E':
    return False

  if check_win(board)!= 'Continue':
    return False

  return True

def reward(board):
  #rewards +1 if X wins, -1 if O wins and 0 otherwise
  if  check_win(board)=='X':
    return 1
  elif check_win(board)=='O':
    return -1
  else:
    return 0


def next_state_same(board,turn,action):
```

```
  #Gets a board, an action indice and a turn and returns a new board with
that player's mark in that action
  new_board=[]

  for i in range(9):
    if i==action:
      new_board.append(turn)
    else:
      new_board.append(board[i])

  return new_board

def next_state_reverse(board,turn,action):
  #Gets a board, an action indice and a turn and returns a new board with
that player's opposite mark in that action

  new_turn=next_turn(turn)
  new_board=[]

  for i in range(9):
    if i==action:
      new_board.append(new_turn)
    else:
      new_board.append(board[i])

  return new_board







import itertools


def generate_vectors():
  #Generates all possible board states
    elements = ['X', 'O', 'E']
    vectors = []

    for combination in itertools.product(elements, repeat=9):
        vectors.append(list(combination))

    return vectors
```

```python
def initial_v(states):
  #Assigns initial value of 0 to all states
    v_dict = {}
    turns = {'X', 'O'}
    for s in states:
        for t in turns:
          v_dict[tuple(s),t]=0

    return v_dict


def initial_a(states):
  #Implements a greedy policy for winning if winning in one step is
possible and otherwise just chooses the first empty square it can find
  a_dict={}
  turns=['X','O']
  for s in states:
    for t in turns:
     for i in range(9):
        if s[i]=='E':
         a_dict[tuple(s),t]=i
         break

  for s in states:
    for t in turns:
      for i in range(9):
        if check_win(next_state_same(s,t,i))==t:
          a_dict[tuple(s),t]=i

  return a_dict


def v(board,turn,danger_p,calculated_v,best_actions):
  #Implements the Bellman equation
  best_action_max=-1
  best_action_min=-1
  max = float('-inf')
  min= float('inf')
  actionable=False
  candidate=0
```

```
  new_v= calculated_v.copy()
  new_ba= best_actions.copy()

  for action in range(9):
    if A(board,action) and check_win(board)=='Continue':

      candidate=  (   danger_p[action]*(reward(next_state_reverse(board,tu
rn,action))+calculated_v[tuple(next_state_reverse(board,turn,action)),next
_turn(turn)]) +
                       (1-
danger_p[action])*(reward(next_state_same(board,turn,action))+calculated_v
[tuple(next_state_same(board,turn,action)),next_turn(turn)])     )
      actionable=True

    if candidate>max and A(board,action) and check_win(board)=='Continue':
      max=candidate
      best_action_max= action

    if candidate < min and A(board,action) and
check_win(board)=='Continue':
      min=candidate
      best_action_min=action



  if turn=='X'and actionable and check_win(board)=='Continue':
    new_v[tuple(board),turn]=max
    new_ba[tuple(board),turn]=best_action_max
  if turn=='O' and actionable and check_win(board)=='Continue':
    new_v[tuple(board),turn]=min
    new_ba[tuple(board),turn]=best_action_min

  return new_v,new_ba


states= generate_vectors()
```

بعد از این پیاده‌سازی، می‌توان آزمایش‌هایی در نسخه‌های مختلف بازی میان ایجنت MDP و یک ایجنت تصادفی ترتیب داد و نتایج را دید.

ما هفت بازی مختلف را پیاده‌سازی کرده و شبیه‌سازی می‌کنیم. بردار احتمال خطر هریک از بازی‌ها به قرار زیر است:

```
Stochastic_P=[0.9, 0.3, 0.5, 0.2, 0.95, 0.7, 0.6, 0.45, 0.1]

Classic_P= [0,0,0,0,0,0,0,0,0]

Reverse_P=[1,1,1,1,1,1,1,1,1]

Noisy_reverse_P=[1,1,1,1,1,1,1,1,1]
for i in range(9):
  temp=random.random()*0.1
  Noisy_reverse_P[i]-=temp

Noisy_classic_P= [0,0,0,0,0,0,0,0,0]
for i in range(9):
  temp=random.random()*0.1
  Noisy_classic_P[i]+=temp

Half_P=[0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]

totally_random=[]
for i in range(9):
  totally_random.append(random.random())
```

بازی اول، اعداد شبه تصادفی ولی ساخته دست انسان را به عنوان احتمالات خطر در نظر می‌گیرد.

در بازی دوم، احتمالات خطر صفر هستند و بنابراین بازی، همان بازی کلاسیک است.

بازی سوم، اصطلاحا بازی معکوس یا ***reverse*** است که در آن، هر بازیکن در واقع به جای حریفش بازی می‌کند.

بازی چهارم، یک نسخه نویزدار از بازی معکوس است و بازی پنجم، یک نسخه نویزدار از بازی کلاسیک است.

بازی ششم، یک بازی کاملا کور است که تمام خانه‌ها، پنجاه درصد احتمال خطر دارند.

در نهایت بازی هفتم، یک بازی کاملا تصادفی است که هر خانه، یک احتمال خطر ناشناخته و تصادفی دارد و بنابراین هیچ توزیع و بایاس انسانی هم روی تصادفی بودن خانه‌ها وجود ندارد.

از تابع زیر برای تقریب عددی تابع ارزش با استفاده از *value iteration* استفاده می‌شود:

```
def find_value_function(P, name,iterations,states):


  cv=initial_v(states)
  ba=initial_a(states)

  for i in range(iterations):
    old_v=cv.copy()
    old_ba=ba.copy()
    diff={}
    for s in states:
      for t in ['X','O']:
        if check_win(s)=='Continue':
          temp_cv,temp_ba= v(s,t,P,old_v,old_ba)
          cv[tuple(s),t]= temp_cv[tuple(s),t]
          ba[tuple(s),t]=temp_ba[tuple(s),t]
          diff[tuple(s),t]= abs(cv[tuple(s),t]-old_v[tuple(s),t])
    print("\n\nmax diff:")
    print(max(diff.values()))
    print("max diff state:")
    print(max(diff, key=lambda k: diff[k]))
    print("average diff: ")
    print(sum(diff.values()) / len(diff.values()))

  import sys


  # Open the file in write mode
  with open(name, "w") as file:
      for s in states:
          for t in ['X', 'O']:
              if check_win(s) == 'Continue':
                  # Print the output of board_printer to the file
                  old_stdout = sys.stdout
```

```python
                    sys.stdout = file  # Redirect stdout to the file
                    board_printer(s)
                    sys.stdout = old_stdout  # Restore stdout

                    # Redirect print statements to the file
                    print("turn=", file=file)
                    print(t, file=file)
                    print("value= ", file=file)
                    print(cv[tuple(s), t], file=file)
                    print("action= ", file=file)
                    print(ba[tuple(s), t] + 1, file=file)



                    # Print a separator in the file
                    print("-" * 20, file=file)

    return ba
```

مقادیر زیر، لاگ به دست آمده برای آموزش نسخه شبه تصادفی در ده گام است:

```
max diff:
0.95
max diff state:
(('X', 'X', 'O', 'O', 'E', 'X', 'X', 'X', 'O'), 'O')
average diff:
0.29935094203553664
```

```
max diff:
0.8999999999999999
max diff state:
(('X', 'X', 'O', 'X', 'E', 'X', 'O', 'X', 'E'), 'X')
average diff:
0.18552375371856858
```

```
max diff:
0.8999999999999999
max diff state:
(('X', 'X', 'O', 'X', 'E', 'E', 'O', 'E', 'X'), 'X')
average diff:
0.12448458487334439
```

```
max diff:
0.8999999999999999
max diff state:
```

```
(('X', 'X', 'O', 'E', 'E', 'E', 'O', 'E', 'X'), 'X')
average diff:
0.08418015865861354


max diff:
0.8999999999999999
max diff state:
(('X', 'E', 'O', 'E', 'E', 'E', 'O', 'E', 'X'), 'X')
average diff:
0.03923012507887873


max diff:
0.8202119999999999
max diff state:
(('X', 'E', 'O', 'E', 'E', 'E', 'O', 'E', 'E'), 'X')
average diff:
0.013901889434778688


max diff:
0.7528356
max diff state:
(('E', 'E', 'X', 'E', 'E', 'E', 'X', 'E', 'E'), 'O')
average diff:
0.0031635938564860736


max diff:
0.37447880000000006
max diff state:
(('X', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'), 'X')
average diff:
0.00033904301316145325


max diff:
0.27007661000000005
max diff state:
(('E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'), 'X')
average diff:
2.434657982511494e-05


max diff:
0.0
max diff state:
(('X', 'X', 'O', 'X', 'X', 'O', 'O', 'O', 'E'), 'X')
average diff:
0.0
```

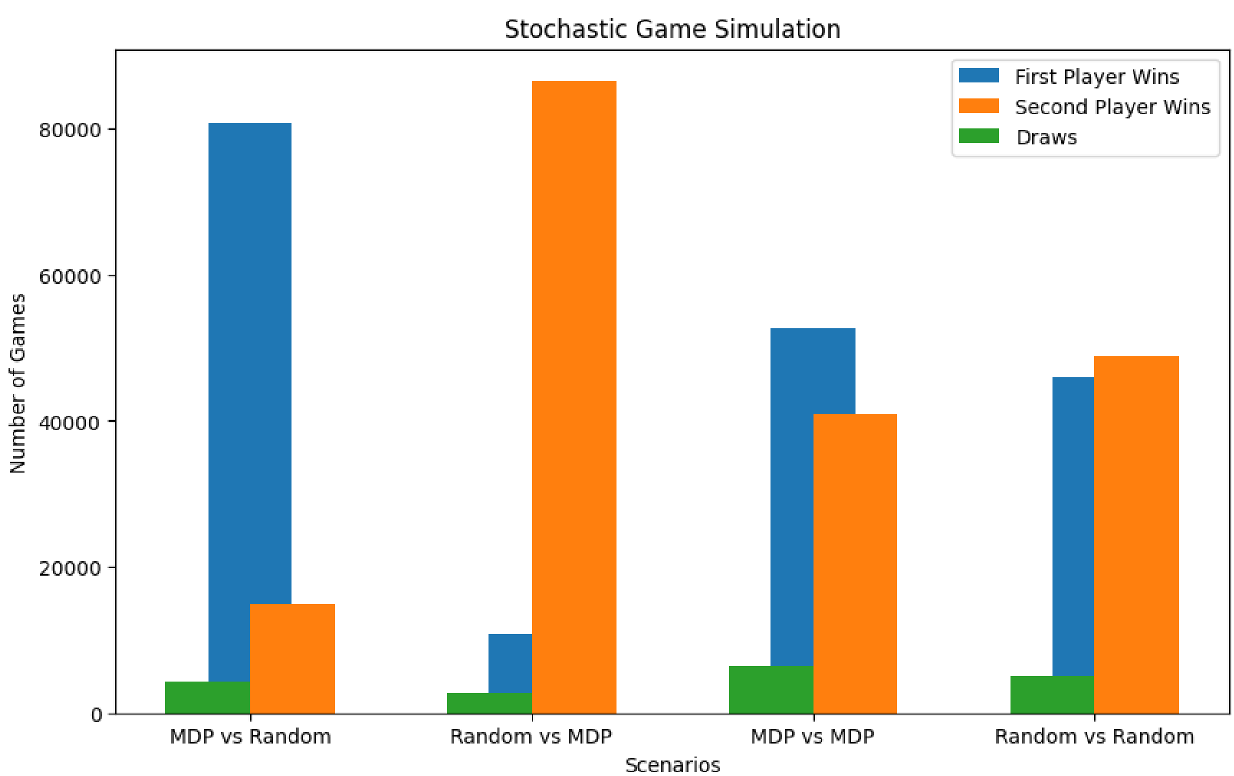
در این لاگ و تمامی لاگ‌های بعدی، منظور از عبارت ***max_diff*** مقدار زیر است:

$$\|T(v) - v\|_\infty$$

همچنین منظور از *average_diff* هم میانگین تفاضل ارزش تمام حالت‌ها با تابع ارزش *V* و تابع ارزش *T(V)* است.

همانطور که دیده می‌شود، بعد از تنها ده گام، همگرایی رخ داده است و به تابع ارزش واقعی و بهین رسیده‌ایم.

در ادامه در قطعه کدی که برای اجتناب از شلوغ شدن دیگر در اینجا آن را نمی‌آوریم، بین ایجنت آموزش دیده برای این بازی و یک ایجنت رندوم و تصادفی، 100 هزار بازی برگزار می‌شود و نتایج به شرح زیر هستند:



نمودار 1: نتایج بازی سه در سه در محیط شبه تصادفی

همانطور که در نتایج بالا دیده می‌شود، وقتی *MDP* اول یا دوم شروع می‌کند، تعداد زیادی از بازی‌ها را می‌برد و تنها تعداد کمی از بازی‌ها را به دلیل شرایط تصادفی و رندوم بازی می‌بازد. در حالت *MDP* مقابل *MDP* می‌بینیم که در این بازی خاص، یک بایاس به نفع کسی که اول بازی

را شروع می‌کند وجود دارد. ولی با حالت ایجنت تصادفی در مقابل خودش می‌بینیم که این بایاس تنها در صورتی دیده می‌شود که انتخاب‌ها هوشمندانه انجام شوند و در غیر این صورت نفر دوم حتی برتری اندکی هم دارد.

در ادامه، لاگ آموزش برای بازی کلاسیک را مشاهده می‌کنید:

```
max diff:
1
max diff state:
(('X', 'X', 'O', 'X', 'X', 'O', 'O', 'O', 'E'), 'X')
average diff:
0.5936175966825926

max diff:
1
max diff state:
(('X', 'X', 'O', 'X', 'X', 'E', 'O', 'E', 'E'), 'O')
average diff:
0.1732624177409177

max diff:
1
max diff state:
(('X', 'X', 'O', 'X', 'E', 'E', 'O', 'E', 'E'), 'X')
average diff:
0.06373388623456233

max diff:
1
max diff state:
(('X', 'X', 'O', 'X', 'E', 'E', 'E', 'E', 'E'), 'O')
average diff:
0.034977012530424595

max diff:
1
max diff state:
(('X', 'X', 'O', 'E', 'E', 'E', 'E', 'E', 'E'), 'O')
average diff:
0.011989542955016677
```

```
max diff:
1
max diff state:
(('E', 'X', 'E', 'E', 'E', 'E', 'E', 'X', 'E'), 'O')
average diff:
0.00018029387902280717
```

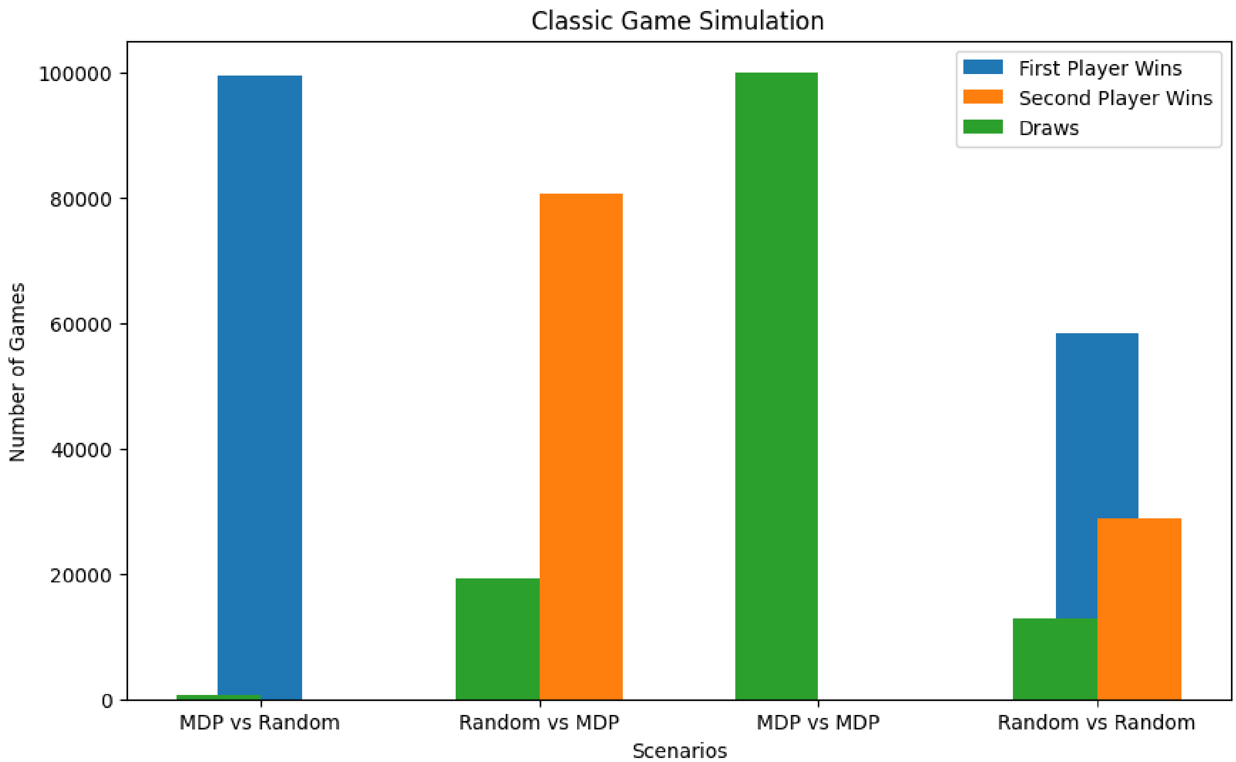
```
max diff:
0
max diff state:
(('X', 'X', 'O', 'X', 'X', 'O', 'O', 'O', 'E'), 'X')
average diff:
0.0
```

```
max diff:
0
max diff state:
(('X', 'X', 'O', 'X', 'X', 'O', 'O', 'O', 'E'), 'X')
average diff:
0.0
```

```
max diff:
0
max diff state:
(('X', 'X', 'O', 'X', 'X', 'O', 'O', 'O', 'E'), 'X')
average diff:
0.0
```

```
max diff:
0
max diff state:
(('X', 'X', 'O', 'X', 'X', 'O', 'O', 'O', 'E'), 'X')
average diff:
0.0
```

دیده می‌شود که این بازی حتی سریع‌تر هم همگرا می‌شود. نتایج شبیه‌سازی صد هزار بازی ایجنت آموزش‌دیده مقابل حریف تصادفی در این بازی به شرح زیر است:

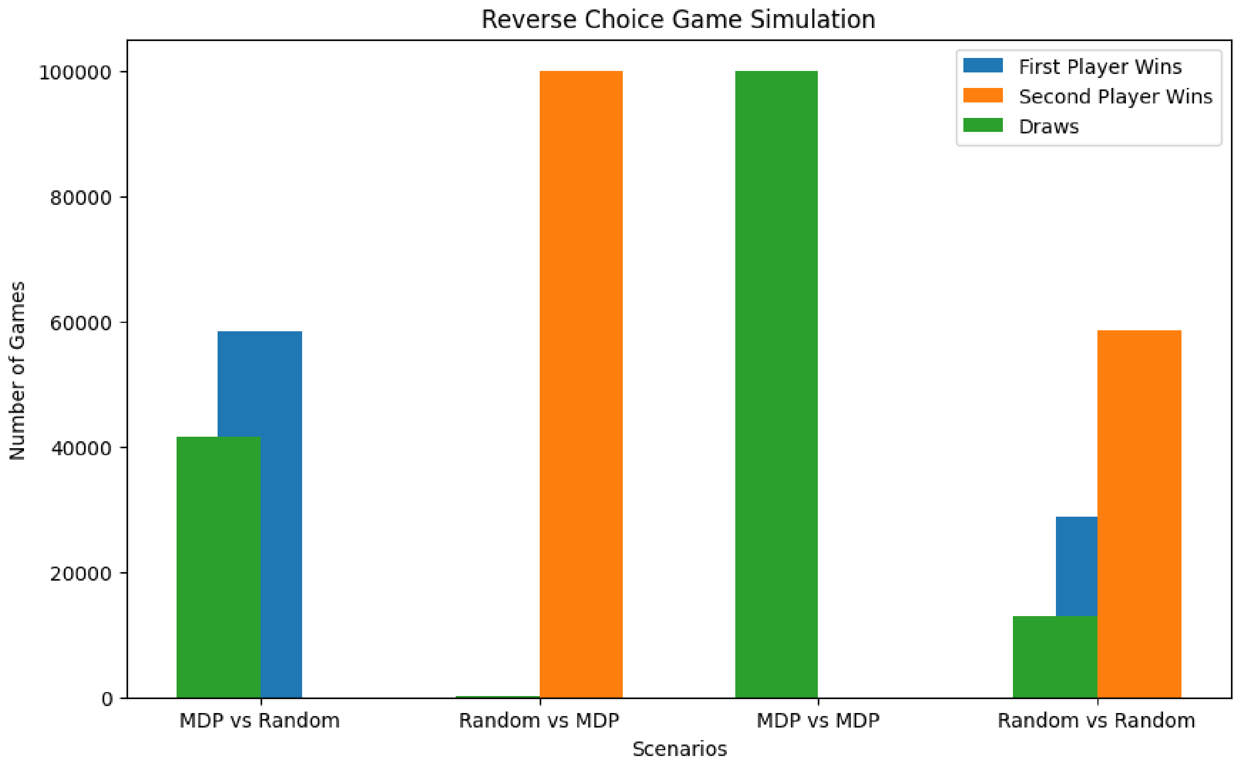


نمودار 2: شبیه‌سازی بازی کلاسیک

می‌بینیم که مطابق انتظارمان از نظریه بازی، ایجنت وقتی بازی را ابتدا شروع می‌کند تقریبا همیشه راه حل بهین برنده شدن را که اگر رقیب هوشمندانه بازی نکند، وجود دارد، پیدا می‌کند. تنها در تعداد اندکی از بازی‌ها، رقیب رندوم توانسته به شکل تصادفی استراتژی تساوی گرفتن را پیدا کند. در حالتی که *MDP* دوم شروع می‌کند، تعداد بیشتری از بازی‌ها تساوی می‌شوند ولی همچنان اکثریت غالب بازی‌ها با پیروزی *MDP* به پایان می‌رسند. نکته مهم این است که *MDP* هرگز هیچ بازی‌ای را واگذار نمی‌کند. مهم‌تر آنکه در بازی دو ایجنت آموزش دیده مقابل هم، تمام بازی‌ها بدون استثنا با تساوی به پایان می‌رسند.

برای اجتناب از طولانی شدن گزارش، دیگر در ادامه لاگ آموزش مابقی بازی‌ها را نمی‌آوریم. لاگ‌های کامل در نوت‌بوک ژوپیتری که در گیت‌هاب مربوط به پروژه بارگذاری می‌شود قابل ملاحظه خواهد بود.

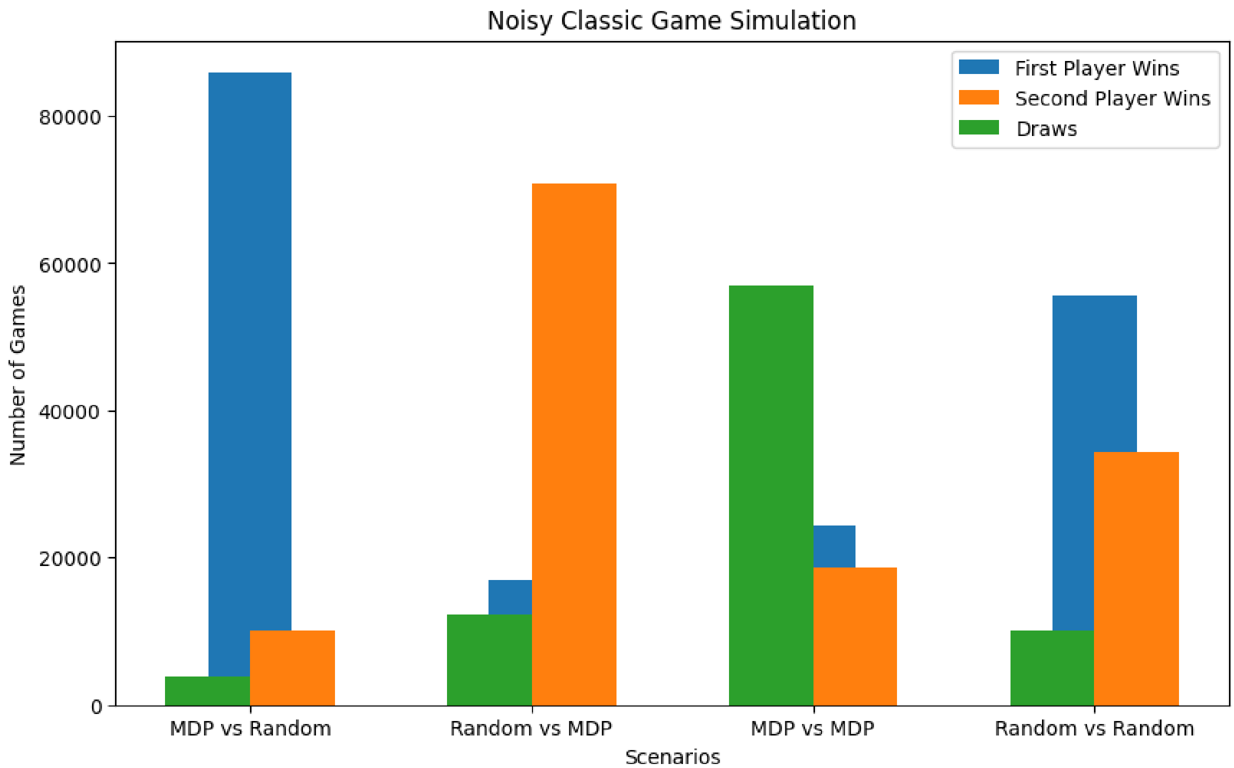
در بازی معکوس، نتایج زیر به دست آمدند:



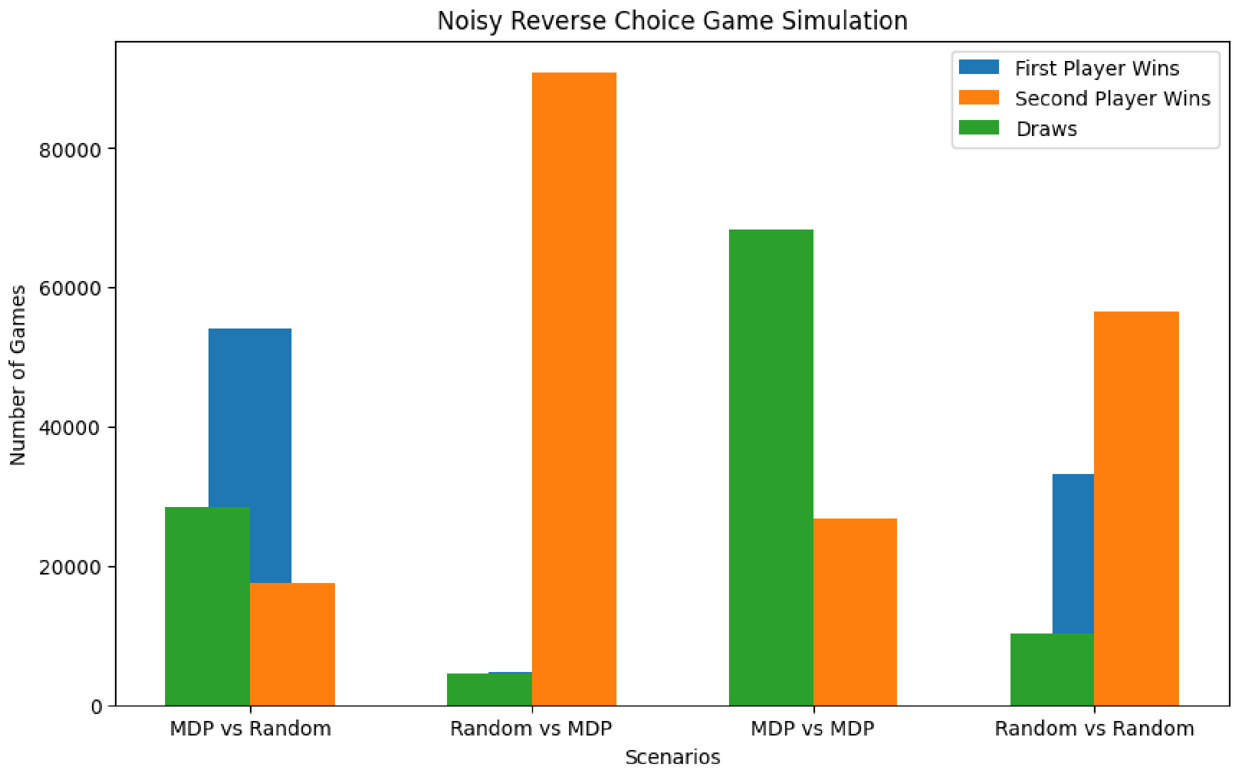
نمودار 3: شبیه‌سازی‌های بازی معکوس

نتایج در بازی معکوس، کمی متفاوت بودند. اولا واضح است که کسی که دوم بازی را شروع کند، دست برتری دارد. وقتی ایجنت آموزش دیده دوم بازی را شروع می‌کند، تقریبا تمام بازی‌ها را می‌برد و تعداد خیلی ناچیزی بازی با تساوی تمام می‌شود. ولی در مقابل، وقتی ایجنت ابتدا بازی را شروع می‌کند، گرچه همچنان تعداد زیادی از بازی‌ها را می‌برد، ولی تعداد قابل توجهی بازی نیز با تساوی به پایان می‌رسد. همچنین بازی دو ایجنت آموزش‌دیده مقابل هم نشان می‌دهد این بازی هم مطابق انتظار، استراتژی فورسینگ برای تساوی دارد و کسی نمی‌تواند به طور قطعی برنده شود.

دو شکل بعدی، مربوط به نسخه‌های نویزی بازی کلاسیک و معکوس می‌شوند:



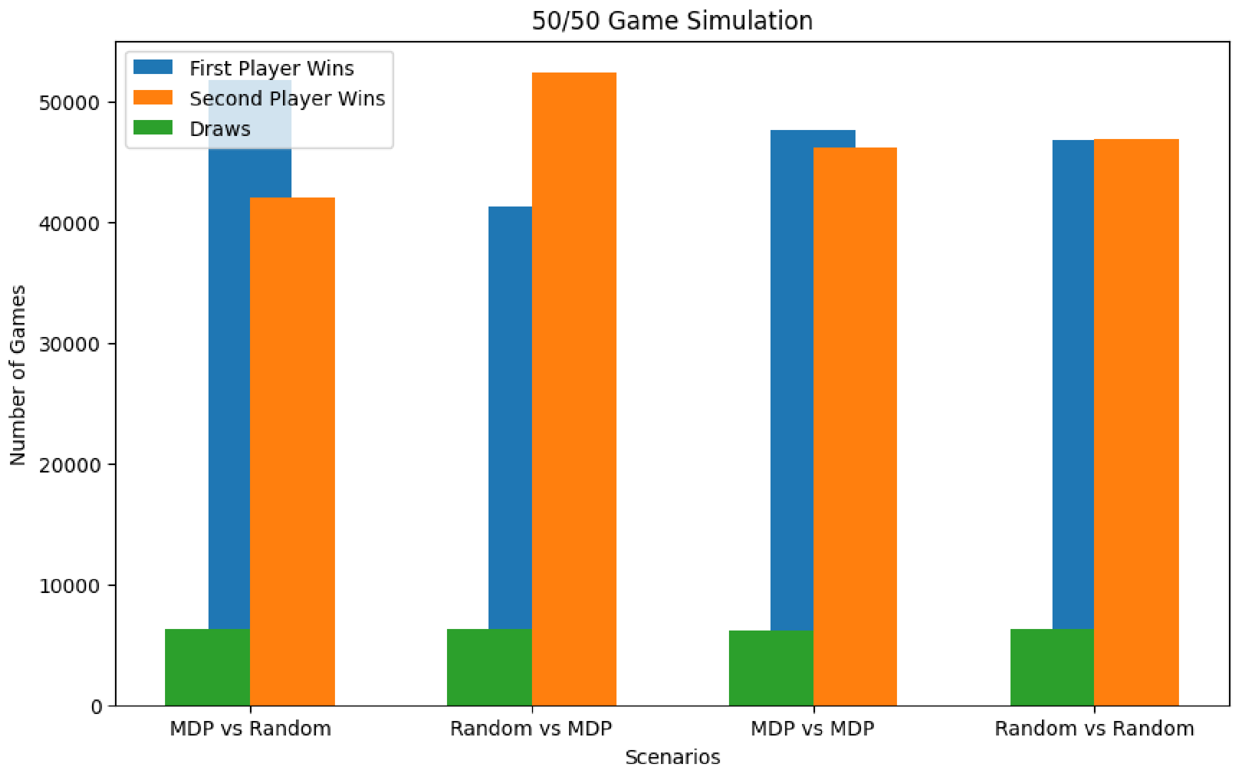
نمودار 4: نتایج شبیه‌سازی بازی کلاسیک نویزی



نمودار 5: نتایج شبیه‌سازی بازی معکوس نویزی

می‌بینیم که در بازی‌های معکوس هم همچنان دست بالا با ایجنت آموزش دیده است، فقط به اندازه میزان نویز ایجاد شده، گاهی نوفه‌ای در الگو می‌افتد که منجر به افزایش برد ایجنت رندوم می‌شود. همچنین در حالتی که دو *MDP* با هم بازی می‌کنند، نفر دوم هیچوقت بازی را واگذار نمی‌کند ولی تعداد زیادی بازی را برنده می‌شود، هرچند همچنان بیشتر بازی‌ها با تساوی تمام می‌شوند.

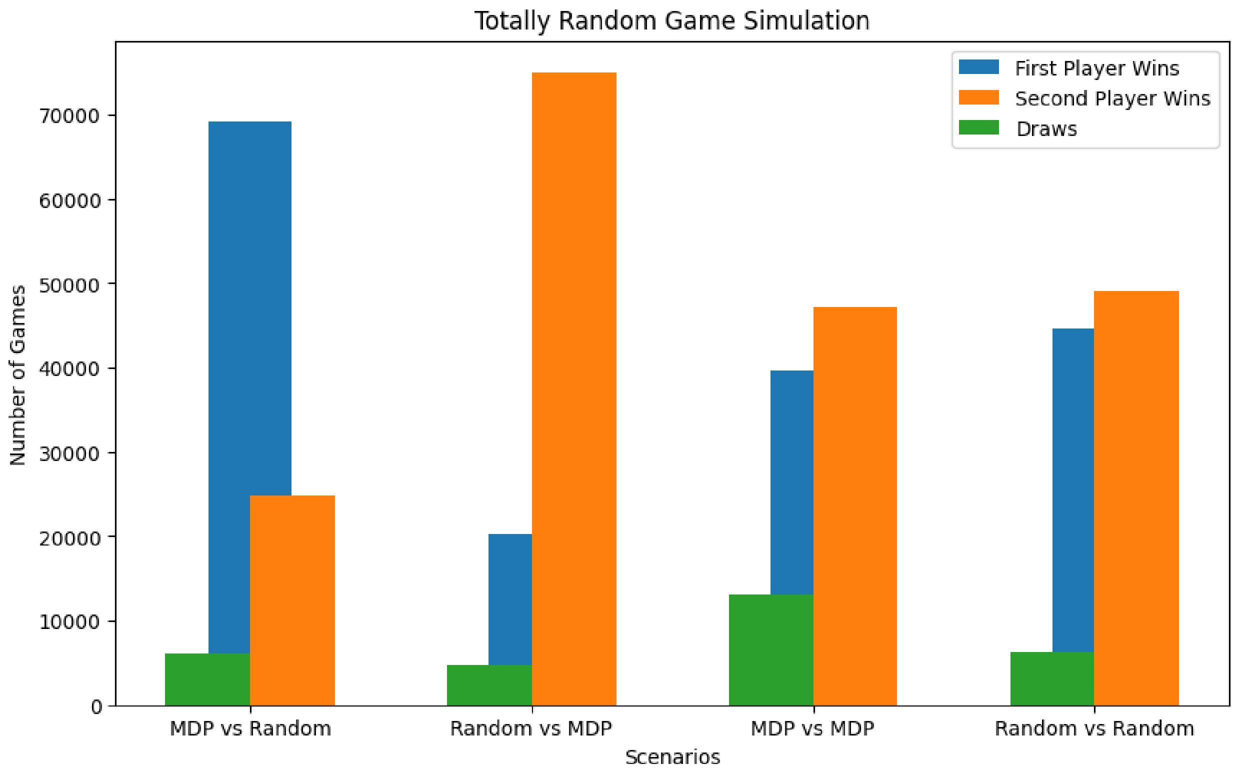
نتیجه زیر هم برای بازی کورکورانه است:



نمودار 6: نتایج شبیه‌سازی بازی کورکورانه

در این بازی کورکورانه و 50-50 انتظار داشتیم که استراتژی معنای چندانی نداشته باشد. ولی برخلاف تصور میبینیم که استراتژی اینجا هم تأثیر دارد و گرچه تأثیرش کمتر است، ولی همچنان ایجنتی که آموزش دیده، ده هزار بازی بیشتر برنده می‌شود. حدس خود من این هست که چنین مواردی فقط مربوط به جاهای نزدیک به پایان بازی میشن که ایجنت آموزش دیده، هوشمندانه خانه‌ای را انتخاب می‌کند که حتی اگه نتیجه‌ش برعکس بشود، حریف ببازد و در واقع خانه بی‌اثر باشد. ولی ایجنت رندوم گاهی خانه‌ای را انتخاب می‌کند که منجر به بردن ایجنت آموزش‌دیده می‌شود.

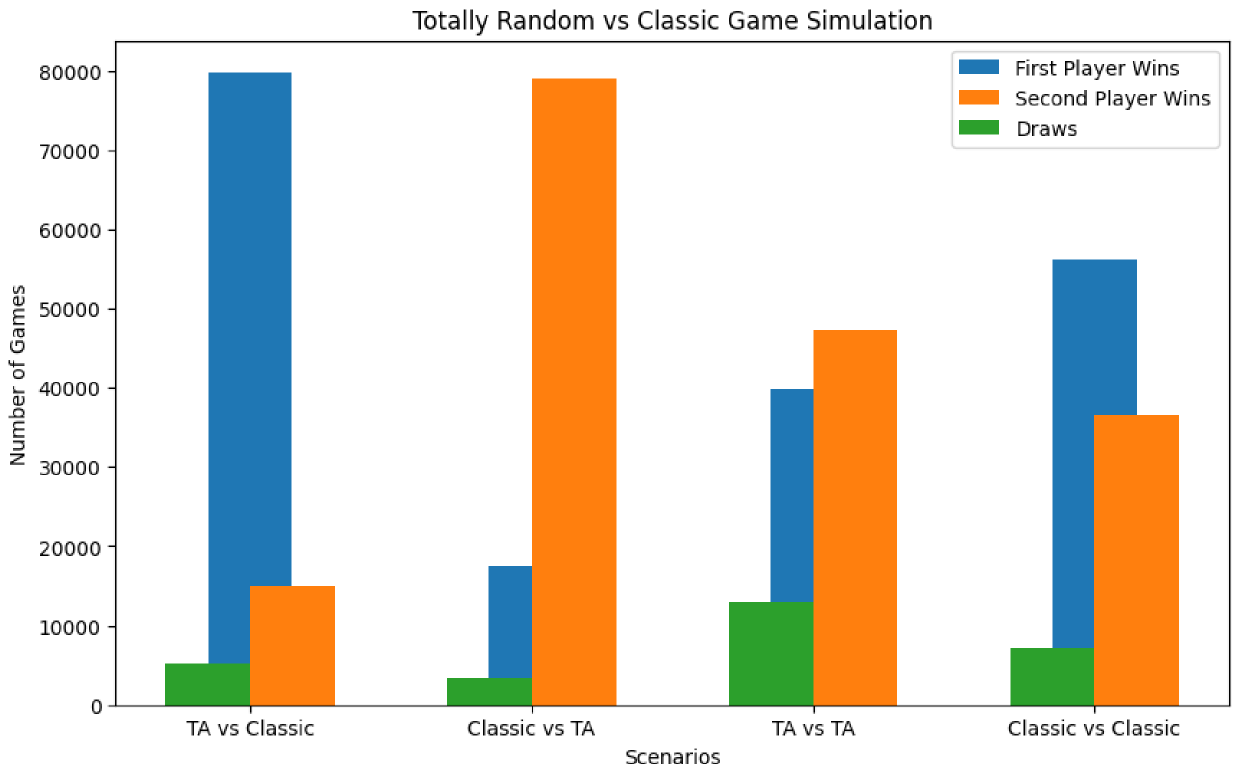
در نهایت برای بازی کاملا تصادفی هم به نتایج زیر می‌رسیم:



نمودار 7 : نتایج شبیه‌سازی بازی کاملا تصادفی

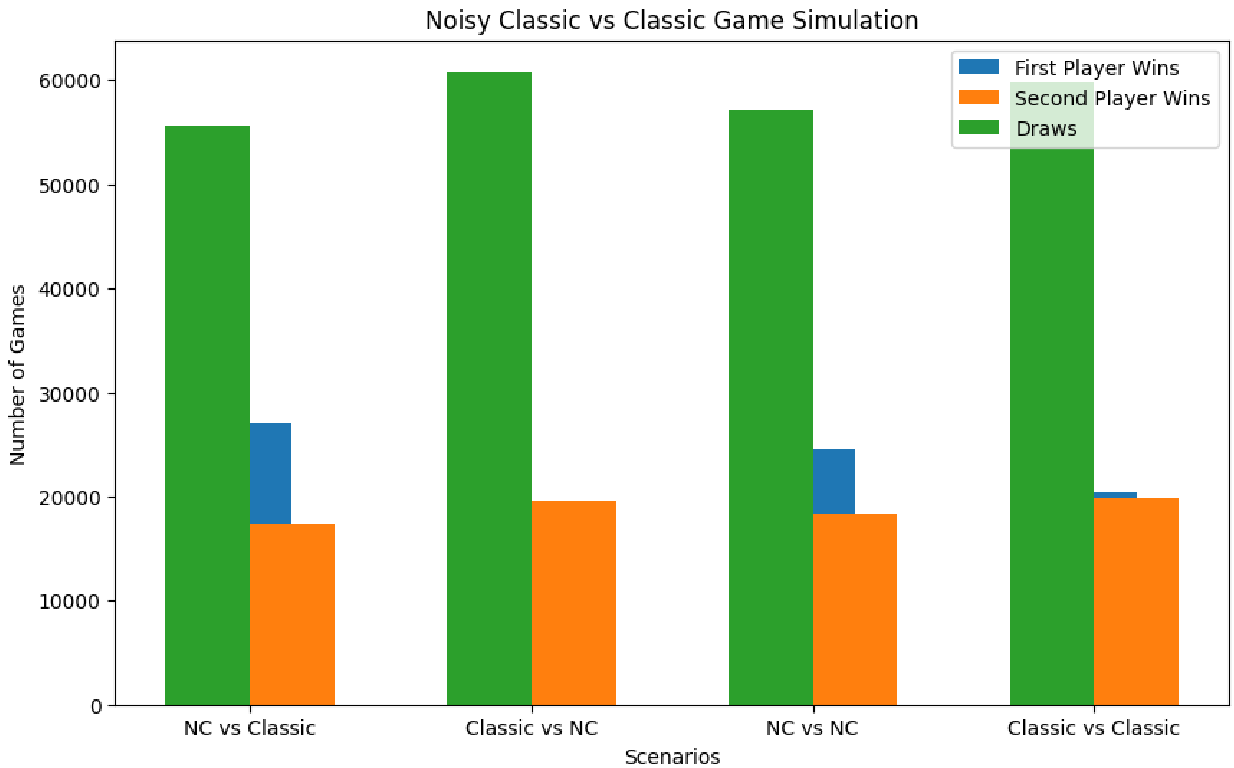
که کمابیش همان الگوی بازی‌های شبه‌تصادفی را دنبال می‌کند.

در پایان یک آزمایش نهایی هم انجام دادیم که از این قرار بود که هر ایجنت را در محیط مربوط به خودش، مقابل ایجنت کلاسیک قرار دادیم و نتایج را شبیه‌سازی کردیم. نتایج به شرح زیر هستند:



نمودار 8 : نتایج شبیه‌سازی در محیط کاملا تصادفی در بازی ایجنت آن محیط مقابل ایجنت کلاسیک

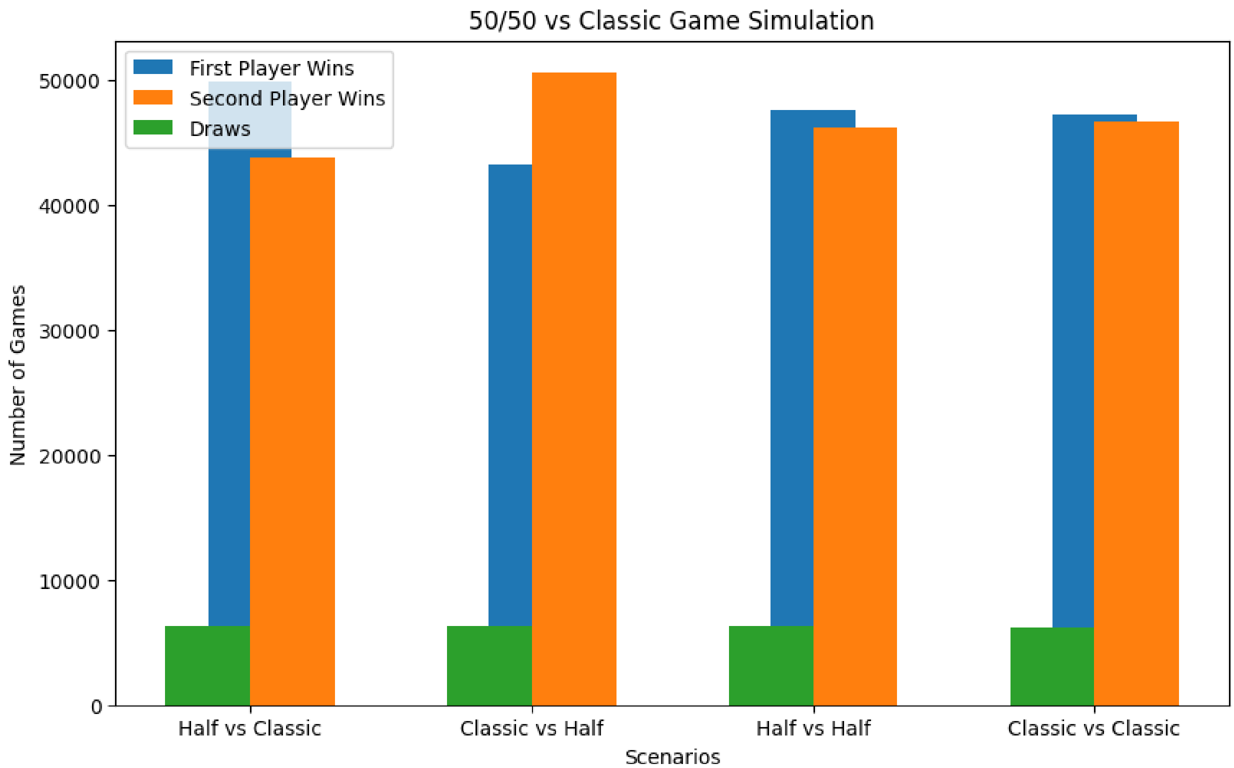
مشاهده می‌شود که استراتژی بهین بازی کلاسیک، مقابل یک ایجنت آموزش دیده در محیط کاملا تصادفی، حتی از یک ایجنت کاملا تصادفی هم نتیجه به مراتب بدتری می‌گیرد(در حدود 5 هزار بازی بیشتر می‌بازد).



نمودار 9 : نتایج شبیه‌سازی در محیط کلاسیک نویزی مقابل ایجنت کلاسیک

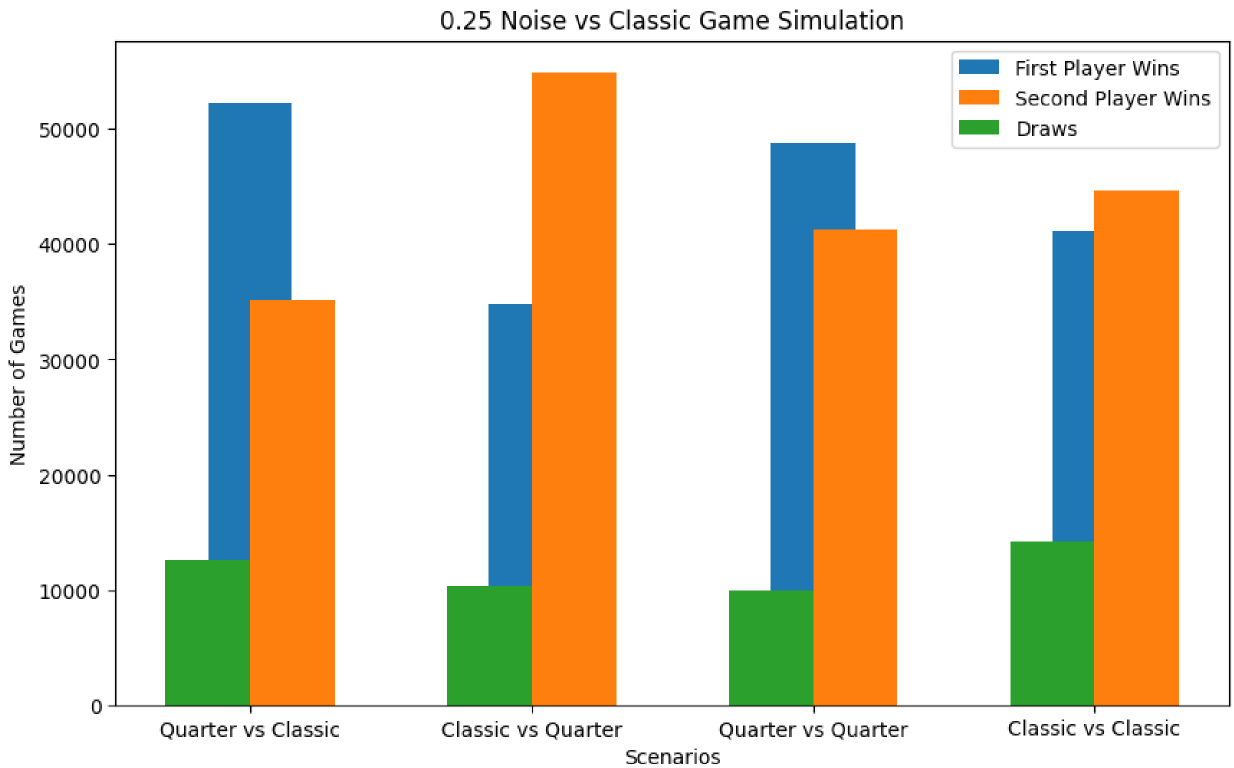
در حالتی که محیط بازی، کلاسیک نویزی باشد اما مطابق انتظار، ایجنت کلاسیک از ایجنت کاملا رندوم نتایج به مراتب بهتری می‌گیرد و در واقع، نتایجش بیشتر از هر نتیجه دیگری تساوی می‌گیرد. هرچند همچنان ایجنت کاملا آموزش دیده بهترین نتایج را برای محیط مختص به خود می‌گیرد.

نتیجه‌گیری شهودی: اگر یک MDP در اختیار نداشته باشیم و بخواهیم در محیط تصادفی بازی کنیم، بسته به میزان تصادفی بودن محیط می‌توانیم یک استراتژی مناسب پیدا کنیم. اگر محیط خیلی تصادفی باشد، بهترین استراتژی در عدم قطعیت کامل این است که کاملا رندوم انتخاب کنیم. ولی اگر میزان تصادفی بودن محیط کم باشد، باید بیشتر به استراتژی بهین بازی کلاسیک(که الگوریتمی ساده دارد) نزدیک شویم.



نمودار 10: نتایج شبیه‌سازی در محیط کورکورانه مقابل ایجنت کلاسیک

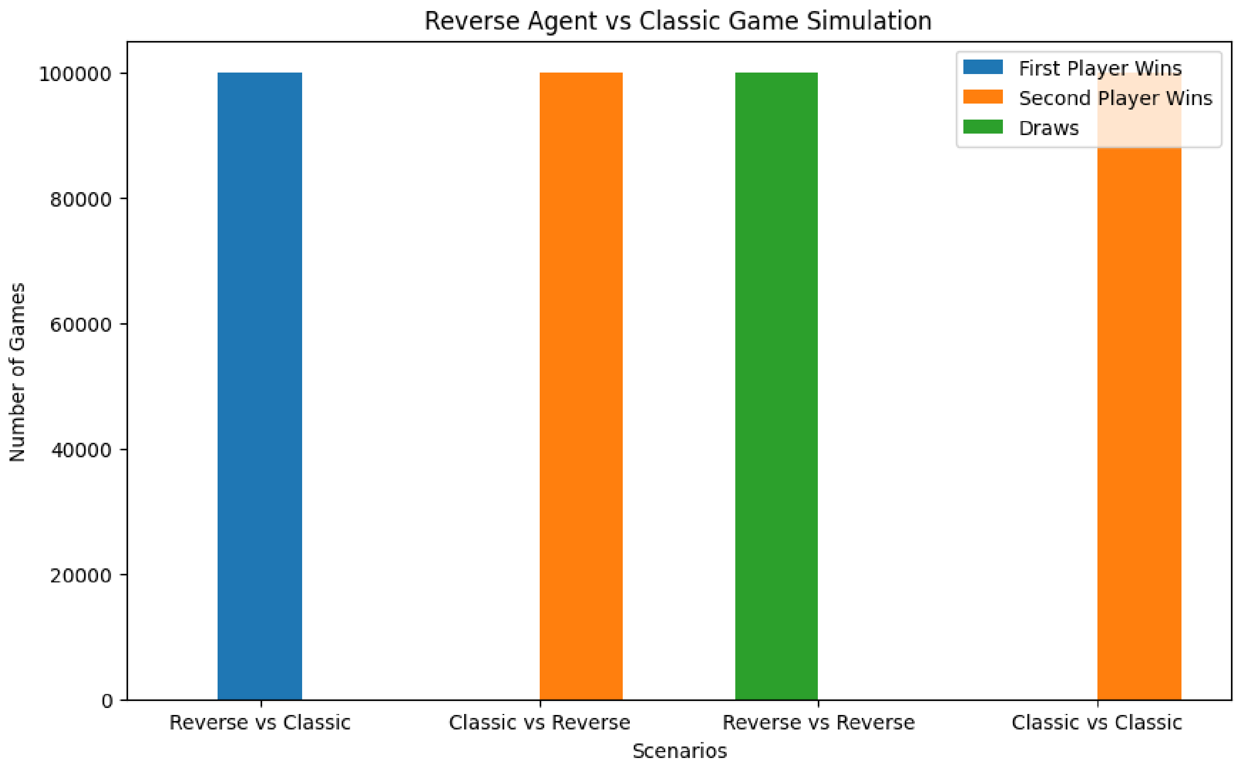
در بازی کورکورانه هم می‌بینیم که استراتژی بهین کلاسیک کمابیش به خوبی همان استراتژی رندوم عمل می‌کند. حدس من این است که این نقطه‌ی قله‌ی ***trade-off*** میان استراتژی رندوم و استراتژی کلاسیک برای نویز یکنواخت است. واضح است که از این نقطه به بعد، به سمت بازی معکوس می‌رویم و در این حالات، اتفاقا استفاده از استراتژی بهین کلاسیک به ما ضربه خواهد زد. این فرضیه را هم بعد از بررسی فرضیه 25 درصدی، بررسی خواهیم کرد. نتیجه بررسی فرضیه 25 درصدی به قرار زیر است:



نمودار 11: نتایج شبیه‌سازی محیط ربع احتمالاتی مقابل ایجنت کلاسیک

گرچه نمودار بالا لزوما این فرضیه را تایید نمی‌کند و برای تایید آن نیاز است نمودارهای این فاصله هم رسم شوند و همچنین اثباتی ریاضیاتی هم ارائه شود، ولی به نظر می‌رسد که واقعا با افزایش نویز یکنواخت تا 50 درصد، شانس برنده شدن ایجنت کلاسیک بیشتر می‌شود.

در پایان، در محیط بازی معکوس، عملکرد ایجنت معکوس و ایجنت کلاسیک را می‌سنجیم:

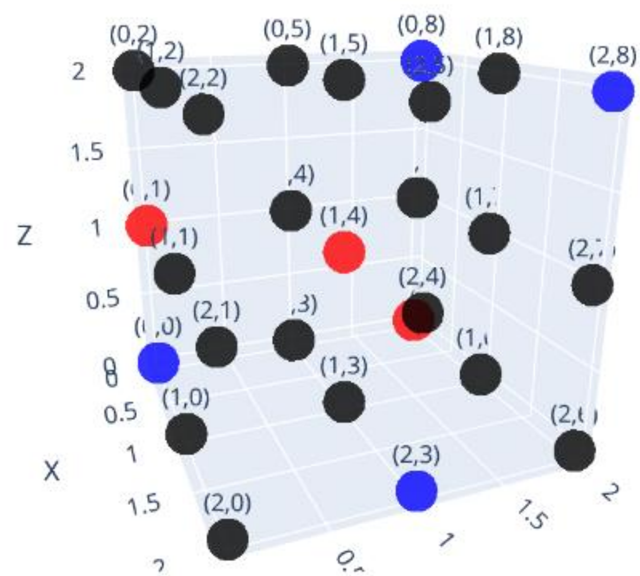


نمودار 12: نتایج شبیه‌سازی در محیط معکوس در بازی ایجنت آن محیط مقابل ایجنت کلاسیک

ملاحظه می‌شود که به نظر می‌آید فرضیه درست بوده است. با افزایش نویز یکنواخت تا 50 درصد، همچنان ایجنت کلاسیک شانسی دارد، ولی در بازی کاملا معکوس، ایجنت کلاسیک عملا نابود می‌شود و هیچ شانسی برای برنده شدن، مگر مقابل خودش و آن هم وقتی نفر دوم باشد ندارد.

بازی سه بعدی 3 در 3 در 3

بازی بعدی که بررسی خواهیم کرد، بازی تیک تک تو در یک مکعب 3 در 3 در 3 است. در این حالت، سایز فضای حالت، برابر با $2 . 3^{27}$ خواهد بود که از اوردر تریلیون است. بنابراین حل این مساله از روش عادی *value iteration* و برنامه‌ریزی و پلنینگ ممکن نیست.



حالت کلاسیک این بازی در مقاله دوم مراجع، مطالعه و بررسی شده است. نتیجه به این صورت است که چنین بازی‌ای هرگز نمی‌تواند به تساوی ختم شود. اگر نفر اول بازی را شروع کند، نفر اول می‌تواند با انتخاب خانه وسطی، قطعا برنده شود. اگر نفر اول خانه وسطی را انتخاب نکند، نفر دوم با انتخاب خانه وسطی می‌تواند برنده شود.

فرمول‌بندی مساله به عنوان یک مساله‌ی *MDP* عینا مشابه همان حالت قبل است، ولی قوانین برنده شدن فرق دارند. در ادامه، این قوانین را در قالب کد پایتون می‌بینیم:

```
def check_win_3d(board):
    # Define all possible winning combinations on each layer
    winning_combinations = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # Rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # Columns
        [0, 4, 8], [2, 4, 6]               # Diagonals
    ]

    # Check if any player has won on any layer
    for i in range(3):
        for combination in winning_combinations:
            if board[i][combination[0]] == board[i][combination[1]] ==
board[i][combination[2]] != 'E':
                return board[i][combination[0]]  # Return the winning player
('X' or 'O')

    # Check for winning combinations that span layers
    for i in range(9):
        # Columns that span layers
        if board[0][i] == board[1][i] == board[2][i] != 'E':
            return board[0][i]
    for i in range(3):
      if board[i][0]==board[i][1]==board[i][2]!='E':
        return board[i][0]
      if board[i][3]==board[i][4]==board[i][5]!='E':
        return board[i][3]
      if board[i][6]==board[i][7]==board[i][8]!='E':
        return board[i][6]

    # Check for diagonals that span layers
    if board[0][0] == board[1][4] == board[2][8] != 'E':
        return board[0][0]
    if board[0][2] == board[1][4] == board[2][6] != 'E':
        return board[0][2]

   #Check for rows that span layers

    if board[0][0] == board[1][1] == board[2][2] != 'E':
        return board[0][0]

    if board[0][3] == board[1][4] == board[2][5] != 'E':
        return board[0][3]

    if board[0][6] == board[1][7] == board[2][8] != 'E':
        return board[0][6]


    # Check if the board is full (draw)
    if 'E' not in np.array(board).flatten():
        return 'Draw'
```

```
    # If no player has won and the board is not full, continue the game
    return 'Continue'
```

عملگرهای مشابهی مثل حالت قبل در این کد هم تعریف می‌شوند. ولی در اینجا، چون فضای حالت خیلی بزرگ است، لازم است از یک تقریب تابعی استفاده کنیم. همچنین، چون حل عددی هم ممکن نیست(زیرا تعداد اکشن‌های ممکن هم در درازمدت خیلی زیاد می‌شوند)، از منطق یادگیری تفاوت زمانی استفاده خواهیم کرد.

کدهای زیر، برای انکود کردن صفحه و نوبت به یک بردار عددی و سپس به دست آوردن یک تقریب تابعی درجه 2 برای حل مساله به روش تمپورال دیفرنس لرنینگ است:

```
def v_encode_state(board, turn):
    # Define a mapping from cell state to code
    cell_to_code = {'X': 3, 'O': 1, 'E': 2}

    # Flatten the board and convert each cell to code
    flat_board = [cell_to_code[cell] for layer in board for cell in layer]

    # Convert turn to code and append it to the encoded state
    turn_code = 1 if turn == 'X' else -1
    flat_board.append(turn_code)

    return flat_board
def board_transformer(board):
    # Define a mapping from code to cell state
    code_to_cell = {1: 'O', 2: 'E', 3: 'X'}

    # Transform the board
    transformed_board = [[code_to_cell[code] for code in layer] for layer in
board]

    return transformed_board


def value_function(x, coef):
    # Initialize the value to the constant term
    value = coef['constant']

    # Add the linear terms
    for i in range(len(x)):
        value += coef['linear'][i] * x[i]

    # Add the quadratic terms
    for i in range(len(x)):
        for j in range(len(x)):
            value += coef['quadratic'][(i, j)] * x[i] * x[j]
```

```
    return value

def value_function_gradient(x, coef):
    # Initialize the gradient to an empty dictionary
    gradient = {'constant': 0, 'linear': [], 'quadratic': {}}

    # Compute the gradient for the linear terms
    for i in range(len(x)):
        gradient['linear'].append(x[i])

    # Compute the gradient for the quadratic terms
    for i in range(len(x)):
        for j in range(len(x)):
            gradient['quadratic'][(i, j)] = x[i] * x[j]

    # The gradient for the constant term is just 1
    gradient['constant'] = 1
    return gradient

def td_error_value_function(reward, old_state, new_state, coef):
    # Calculate the TD error
    V_old = value_function(old_state, coef)
    V_new = value_function(new_state, coef)

    # The discount factor is set to 1 as the game horizon is 27 moves
    gamma = 1

    TD_error = reward + gamma * V_new - V_old
    return TD_error

def value_function_update_coefficients(coef, td_error, gradient,
learning_rate):
    # Update the coefficients of the value function using the TD error and
the learning rate
    coef['constant'] += learning_rate * td_error * gradient['constant']

    for i in range(len(coef['linear'])):
        coef['linear'][i] += learning_rate * td_error * gradient['linear'][i]

    for key in coef['quadratic'].keys():
        coef['quadratic'][key] += learning_rate * td_error *
gradient['quadratic'][key]

    return coef

def value_greedy_action(board, turn, coef, danger_prob):
    # Initialize the best value and action
    if turn == 'X':
        best_value = -float('inf')
    else:  # turn == 'O'
        best_value = float('inf')
    best_action = None

    # Iterate over all possible actions
    for layer in range(3):
        for square in range(9):
            action = (layer, square)
```

```
            # Check if the action is legal
            if A(board, action):
                # Take the action and encode the next state
                prob = danger_prob[layer][square]
                next_board = np.random.choice([next_state_same,
next_state_reverse], p=[1-prob, prob])(board, turn, action)
                next_state = v_encode_state(next_board, next_turn(turn))

                # Calculate the value of the next state
                value = value_function(next_state, coef)

                # If this value is the best so far, store the value and
action
                if turn == 'X' and value > best_value:
                    best_value = value
                    best_action = action
                elif turn == 'O' and value < best_value:
                    best_value = value
                    best_action = action

    # Return the action that leads to the highest (or lowest) value state
    return best_action

import random

def random_action(board, *args):
    # Initialize a list to store all legal actions
    legal_actions = []
    action=None
    # Iterate over all possible actions
    for layer in range(3):
        for square in range(9):
            action = (layer, square)

            # Check if the action is legal
            if Q_A(board, action):
                # If the action is legal, add it to the list of legal actions
                legal_actions.append(action)

    # Choose a legal action at random

    action = random.choice(legal_actions)

    return action

import numpy as np
def td_learning(board, turn, coef, danger_prob, learning_rate, epsilon):
    # Initialize the state
    state = v_encode_state(board, turn)

    # Play the game until it's over
    while True:
        if np.random.rand() < epsilon:  # epsilon is a small positive number
            action = random_action(board)
        else:
            action = value_greedy_action(board, turn, coef, danger_prob)
```

```
        # Decay epsilon

        # If the action is None, the game has ended, so we break the loop
        if action is None:
            break

        # Take the action and observe the new state and reward
        # Use the danger probability to decide whether to place the same or
the opposite mark
        layer, square = action
        prob = danger_prob[layer][square]
        next_board = np.random.choice([next_state_same, next_state_reverse],
p=[1-prob, prob])(board, turn, action)

        next_state = v_encode_state(next_board, next_turn(turn))
        r = reward(next_board)

        # Calculate the TD error
        td_err = td_error_value_function(r, state, next_state, coef)

        # Calculate the gradient of the value function
        gradient = value_function_gradient(state, coef)

        # Update the coefficients of the value function
        coef = value_function_update_coefficients(coef, td_err, gradient,
learning_rate)

        # Update the state and turn
        board = next_board
        turn = next_turn(turn)
        state = next_state

        # Check if the game has ended
        if check_win_3d(board) != 'Continue':
            break

    print(td_err)
    return coef
```

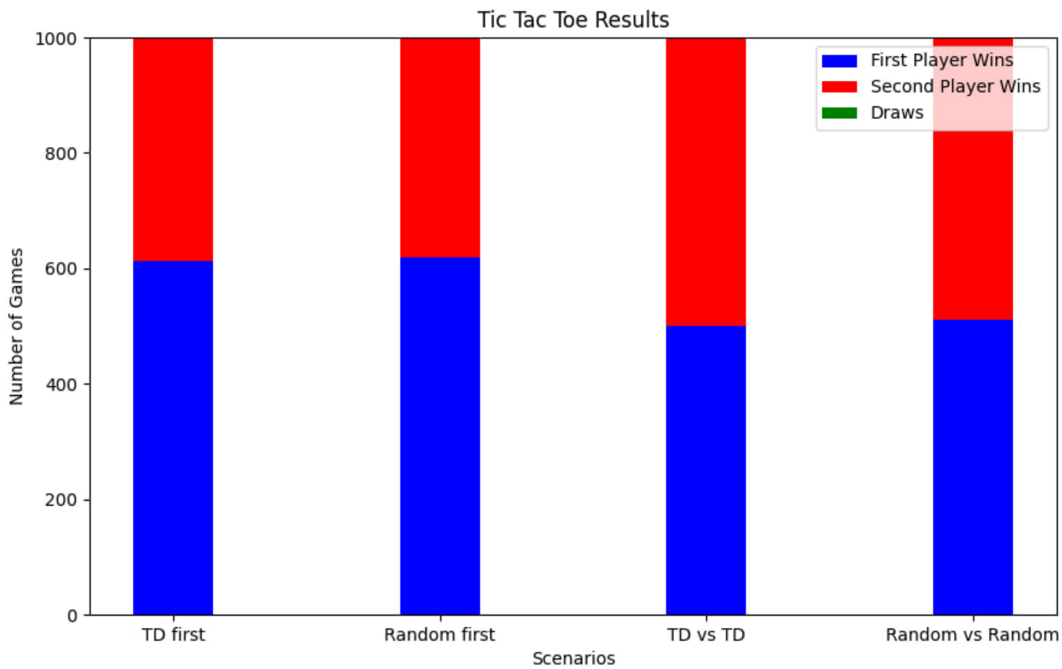
تمپورال دیفرنس لرنینگ یا یادگیری تفاوت زمانی، از ایده اضافه کردن خطا یا مانده‌ی بلمن از تقریب تابعی تا آن لحظه، با یک نرخ یادگیری مشخص، به پارامترهای تقریب تابعی استفاده می‌کند. فرمول اصلی آن به قرار زیر است:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta \left[r_t + \gamma V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t) \right] \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t)$$

ضمنا برای تنظیم استراتژی *exploration* مقابل *exploitation* از استراتژی اپسیلون-گریدی یا اپسیلون-حریصانه استفاده می‌کنیم که با یک احتمال اپسیلون، اکشن جدید و رندومی انجام می‌دهد و با احتمال یک منهای اپسیلون، اکشنی که به شکل حریصانه نسبت به تابع ارزش فعلی بهین باشد. همچنین برای اینکه در ابتدا ایجنت تشویق به کاوش بیشتر شود ولی جلوتر کمتر کاوش کند، از یک نرخ تباهیدگی برای اپسیلون استفاده می‌کنیم تا به تدریج آن را کاهش دهد و نهایتا به مقدار ثابتی برساند.

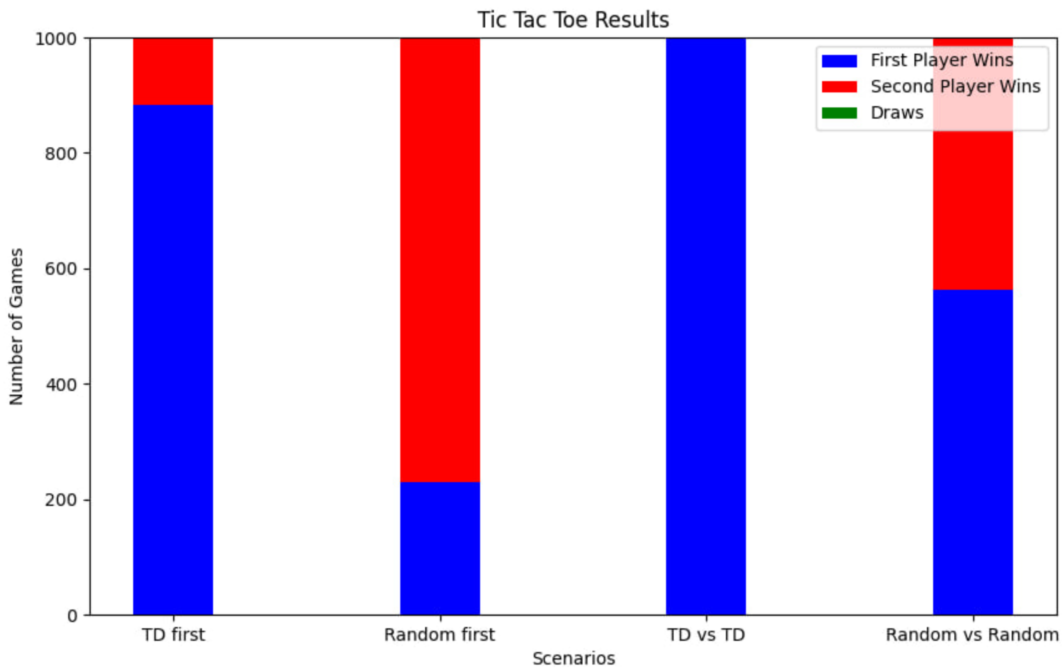
```
coef = {
    'constant': 0,
    'linear': [0 for _ in range(30)],  # 27 cells in the board plus the turn
    'quadratic': {(i, j): 0 for i in range(30) for j in range(30)}
}

# Set the learning rate
learning_rate = 0.00001
epsilon=1
epsilon_decay=0.9998

min_epsilon=0.1

# Set the number of games to train on
num_games = 10000
simulations=1000
# Initialize the danger probability matrix
danger_prob = [[random.random() for _ in range(9)] for _ in
               range(3)]  # for example, all cells have a 0.1 danger
probability


# Train the agent using Q-learning
for i in range(num_games):
    board = [['E' for _ in range(9)] for _ in range(3)]
    turn = 'X'
    coef = q_learning(board, turn, coef, danger_prob, learning_rate, epsilon)
    epsilon = min(epsilon_decay*epsilon, min_epsilon)
```

اولین تلاش با هایپرپارامترهای تنظیم نشده در بازی غیرتصادفی(کلاسیک) اصلا نتایج جالبی نداشتند:



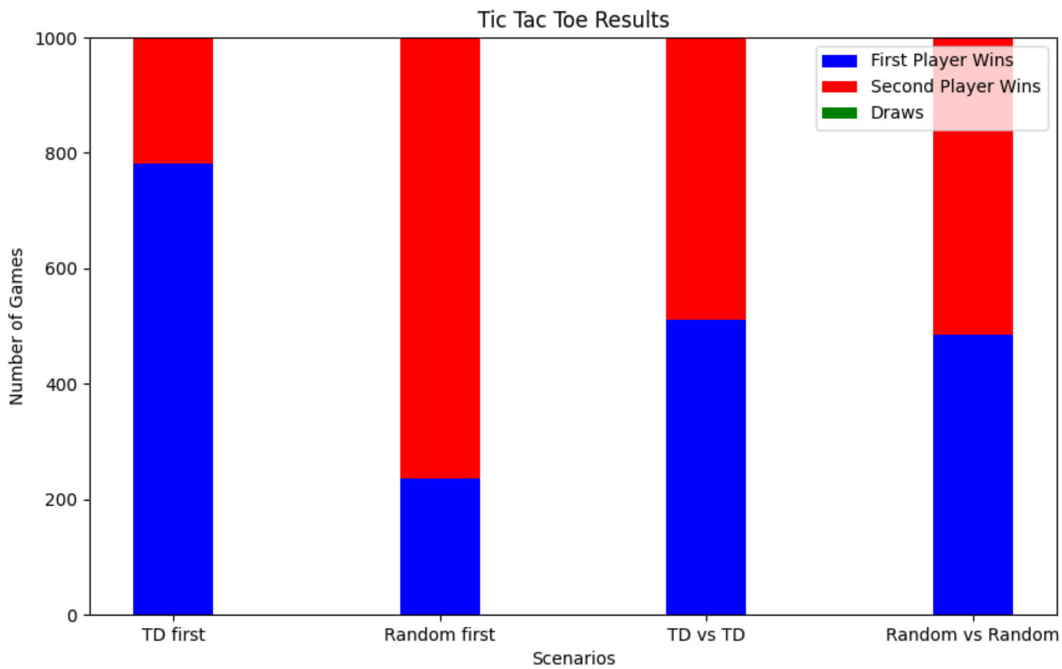
نمودار 13: نتایج با تقریب تابعی درجه 2 قبل از Finetune شدن هایپرپارامترها

ولی به مرور با تنظیم مکرر هایپرپارامترها، به نتایج بهتری رسیدیم:



نمودار 14: نتایج یادگیری تقویتی تفاوت زمانی بعد از تنظیم هایپرپارامترها با تقریب تابعی درجه 2

حالا که در بازی بی‌خطر به نتایج نسبتا معقولی رسیدیم، ایجنت را در محیط کاملا رندوم آموزش دادیم و نتایج به قرار زیر بودند:



نمودار 15: نتایج شبیه‌سازی در محیط کاملا تصادفی با تقریب تابعی درجه 2 از تابع ارزش

به نظر می‌رسد که تمپوران دیفرنس لرنینگ، با تعداد کم بازی در حد ده هزار بازی، قادر به پیدا کردن استراتژی بهین بازی بی‌خطر نیست. در عین حال، من توان پردازشی تعداد بیشتری تکرار را هم نداشتم. با این وجود، هم در محیط غیرتصادفی هم در محیط کاملا رندوم، ایجنت آموزش دیده از روش *TD* به نتایج نسبتا قابل قبولی رسید.

در ادامه، با همان تقریب تابعی درجه 2، روش *Q-learning* را که همگرایی سریع‌تری دارد، آزمایش کردیم:

$$Q(s,a) \leftarrow Q(s,a) + \eta \left[r + \gamma \max_{b} Q(s',b) - Q(s,a) \right]$$

Algorithm 35.1: Q-learning with ϵ-greedy exploration

Initialize value function parameters $\boldsymbol{w}$
repeat
  Sample starting state s of new episode
  repeat
    Sample action $a = \begin{cases} \text{argmax}_b\, Q_{\boldsymbol{w}}(s,b), & \text{with probability } 1-\epsilon \\ \text{random action}, & \text{with probability } \epsilon \end{cases}$
    Observe state s', reward r
    Compute the TD error: $\delta = r + \gamma \max_{a'} Q_{\boldsymbol{w}}(s',a') - Q_{\boldsymbol{w}}(s,a)$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta\delta\nabla_{\boldsymbol{w}} Q_{\boldsymbol{w}}(s,a)$
    $s \leftarrow s'$
  until *state s is terminal*

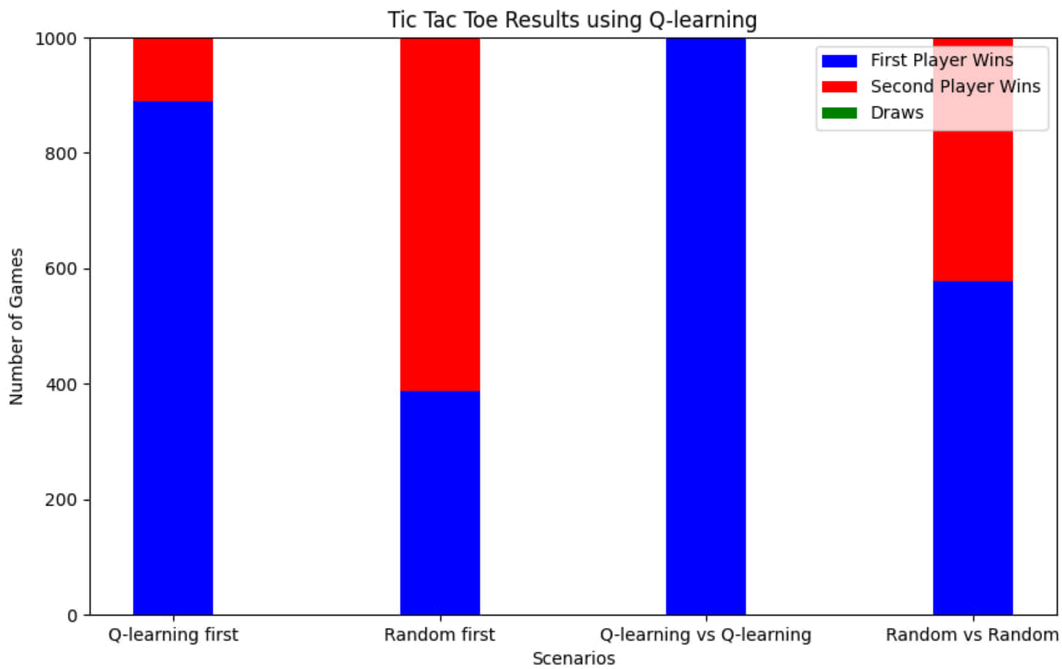
که در آن تابع Q به فرم زیر تعریف می‌شود:

$$Q_{\pi}(s,a) \triangleq \mathbb{E}_{\pi}\left[G_0 | s_0 = s, a_0 = a\right] = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a\right]$$

و ثابت می‌شود که معادله بهینگی بلمن آن به شکل زیر است:

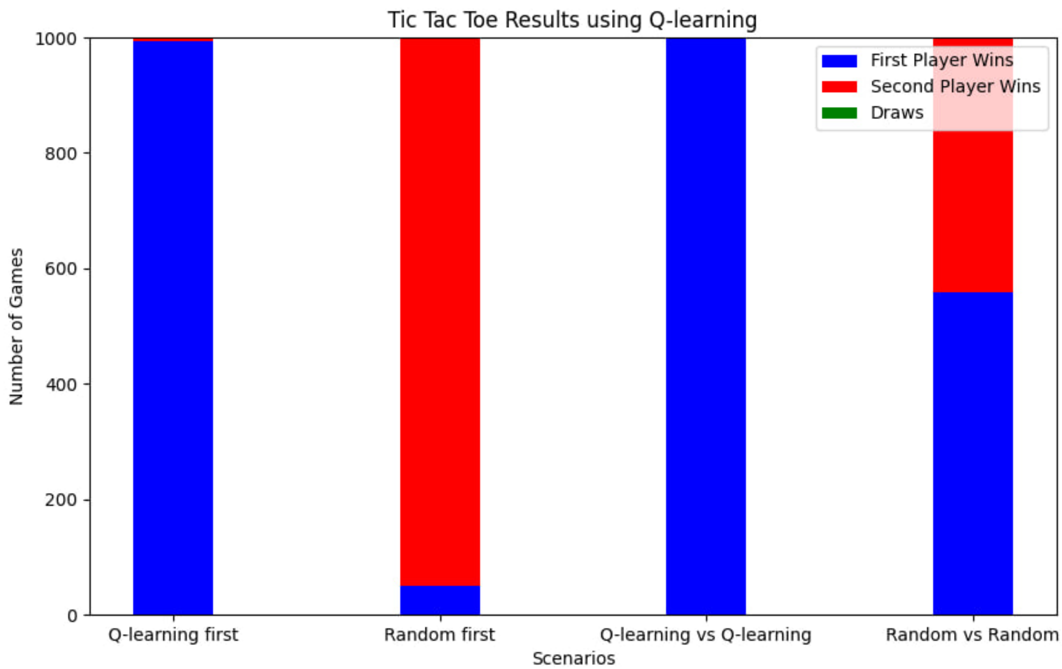
$$Q_*(s,a) = R(s,a) + \gamma \mathbb{E}_{p_T(s'|s,a)}\left[\max_{a'} Q_*(s',a')\right]$$

این الگوریتم، یک الگوریتم ***off-policy*** است که برای مسائل یادگیری آفلاین مثل این مساله، مناسب است. بعد از تنظیم مناسب هایپرپارامترها، نتیجه زیر برای هزار بازی مقابل ایجنت رندوم به دست آمد:



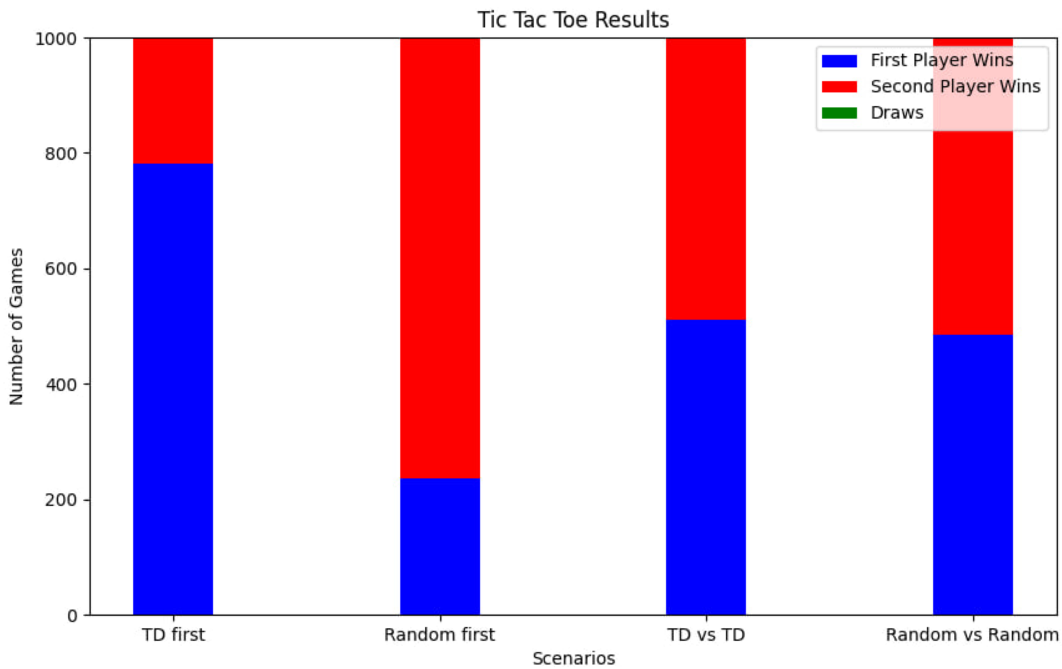
نمودار 16: نتیجه ایجنت با تقریب تابعی درجه 2 از تابع Q

در تقریب تابعی درجه دوم، من هرکاری کردم نتوانستم به نتیجه بهتری برسیم. ولی با جایگزین کردن تابع درجه 2 با یک شبکه عصبی پیشخور(*feed forward*) با یک لایه پنهان با تعداد نورون‌های 64 و تابع فعال‌سازی *relu* و تابع فقدان(لاس فانکشن) ای که همان اختلاف مانده بلمن را نمایندگی کند، به نتیجه زیر رسیدیم:



نمودار 17: نتیجه در محیط بی‌خطر و با تقریب تابعی شبکه عصبی کم‌عمق

این ایجنت مبتنی بر شبکه عصبی، در محیط کاملا رندوم هم چنین عملکردی داشت:



نمودار 18: نتیجه عملکرد کیولرنینگ با شبکه عصبی کم‌عمق به عنوان تابع Q و در محیط کاملا تصادفی

ایده برای توسعه پروژه

همانطور که در ابتدا بحث شد، برای توسعه پروژه می‌توان به نسخه‌های جالب‌تر زیر از بازی فکر کرد:

- بازی مقابل رقیب تصادفی: در این حالت، رقیب با یک بردار احتمال مثل Q به شکل تصادفی یک خانه را انتخاب می‌کند و البته بردار احتمال P به عنوان بردار ریسک خانه‌ها هم در جای خود خواهد بود. پس ایجنت، به جای در نظر گرفتن بهترین بازی ممکن رقیب، باید با توجه به بردار احتمال Q بازی رقیب را در نظر بگیرد و به یک معادله بلمن دیگر برای این حالت خواهیم رسید.
- بازی در یک صفحه یا بورد بزرگ‌تر: یکی از مزایای معادله بلمنی که به آن رسیدیم، این بود که با تغییر جزئی برخی تعاریف مثل تعریف فضای اکشن‌ها و فضای حالات، همین معادله قابل استفاده برای صفحه با هر سایز دلخواه نیز خواهد بود. البته در عمل با آزمایش روش عددی ارائه شده برای صفحه 4*4 در گوگل کولب دیده شد که تمام فضای رم تخصیص داده شده توسط گوگل برای ایجاد و انجام محاسبات روی این فضای حالت به سایزی کمتر از $2* 3^16$ مصرف می‌شود. پس گرچه در این حالت تغییر معادله بلمن ضرورتی ندارد، ولی برای روش عددی باید به دنبال راه دیگری گشت.
- بازی تیک تک توی سه بعدی یا کیوبیک: به جای اینکه بازی روی یک صفحه‌ی 3 در 3 باشد، می‌تواند در یک آرایش سه بعدی از تعدادی صفحه باشد. نسخه‌ی کلاسیک کیوبیک معمولا به این شکل است که 4 صفحه‌ی 4 در 4 داریم که روی یکدیگر چیده شده‌اند. در هریک از این صفحه‌ها می‌توان با قرار دادن 4 مارک مشابه سطری، ستونی یا قطری، برنده شد. همچنین بازیکنان می‌توانند با قرار دادن 4 مارک مشابه از خود در یک ستون در بعد سوم از 4 صفحه‌ی مختلف، یا یک قطر سه بعدی 4 تایی از 4 صفحه‌ی مختلف هم برنده شوند. می‌توان برای شروع نسخه‌ی ساده‌تر 3 صفحه‌ی 3 در 3 را بررسی کرد و بعد به سراغ حل نسخه‌ی کلاسیک کیوبیک رفت.

- بازی با ریسک خالی شدن خانه‌ها: می‌توان یک نسخه از بازی را در نظر گرفت که در آن، در ابتدای هر نوبت، با احتمال کمی، هر خانه‌ی پر شده ممکن است مستقل از تمامی خانه‌های دیگر خالی شود.
- بازی با ریسک از دست رفتن نوبت: می‌توان به جای یک بردار احتمال P، دو بردار احتمال $P1$ و $P2$ داشت و وقتی که هر بازیکن، یک خانه مثل i را انتخاب می‌کند، با احتمال $P1(i)$ برعکس مارک خودش در آنجا قرار می‌گیرد، با احتمال $P2(i)$ مارک خودش آنجا قرار می‌گیرد و با احتمالی مثل $1-P1(i)-P2(i)$ چیزی در خانه‌ی i قرار نمی‌گیرد و به نوعی نوبت ایجنت از بین می‌رود.
- بازی در حالت زمان پیوسته: در هریک از حالت‌های بالا و حتی حالتی که در این گزارش بررسی شد، به جای یک بردار احتمال ثابت، هر خانه یک نرخ خطر دارد (پس یک بردار نرخ خطر داریم) و احتمال خطرها توزیعی نمایی با گذشت زمان و با آن نرخ‌ها خواهد داشت. به این ترتیب، هرچقدر که ایجنت صبر کند، خطر انتخاب خودش بالاتر می‌رود، ولی خطر رقیب هم بالا می‌رود. ضمنا می‌توان برای هر نوبت یا برای کل بازی یک سقف زمانی قرار داد. پس اکشن ایجنت نه صرفا خانه‌ی انتخابی، بلکه یک دوتایی خانه و زمان خواهد بود.

منابع و مراجع

https://www.ai.rug.nl/~mwiering/GROUP/ARTICLES/TTT3D_FINAL.pdf

http://library.msri.org/books/Book42/files/golomb.pdf

Probabilistic Machine Learning: Advanced Topics Book by Kevin P. Murphy

Reinforcement Learning: An Introduction Book by Andrew Barto and Richard S. Sutton