به نام خدا

دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر

یادگیری ماشین

پروژه Minesweeper

سید محمد جلالی فریزهندی	۸۱۰۱۹۵۰۶۳
محمد حسین سخاوت	۸۱۰۱۹۵۵۴۳

سال تحصیلی ۹۵

فهرست

فهرست شکل ها

فهرست جداول

مقدمه

در این پروژه بازی minesweeper را پیاده سازی کردیم و از الگوریتم های یادگیری برای حل بازی استفاده کردیم. و به بررسی عوامل مختلف از جمله تعداد مین، ابعاد صفحه، پارامترهای مختلف در الگوریتم یادگیری پرداختیم. و در ادامه سعی کردیم که بتوانیم این بازی را وقتی ابعاد صفحه بزرگ می شود حل کنیم. چرا که وقتی ابعاد صفحه بزرگ می شود تعداد state ها نیز بالا می رود و برای یادگیری لازم است که تعداد episode ها بسیار افزایش پیدا کند. ما سعی کردیم تا راهی را ارئه دهیم که بتوانیم این مسئله با ابعاد بزرگ را حل کنیم.

الگوریتم Q-Learning

این الگوریتم یکی از روش های یادگیری در حوزه reinforcement learning می باشد. که از این روش برای یافتن بهترین action در هر state در محیط MDP (Markov Decision Process) استفاده می شود.

در بازی minesweeper هم از این روش به منظر حل بازی استفاده نمودیم. همانطور که در شبه کدی که در شکل ۱ آمده است الگوریتم را پیاده سازی نمودیم. که توضیحات بیشتر در باب پیاده سازی در بخش مختص به آن آورده شده است.

```
Initialize Q(s, a) arbitrarily
Repeat (for each episode):
   Initialize s
   Repeat (for each step of episode):
      Choose a from s using policy derived from Q
         (e.g., ε-greedy)
      Take action a, observe r, s'
      Q(s, a) <-- Q(s, a) + α [r + γ max_α' Q(s', a') - Q(s, a)]
      s <-- s';
   until s is terminal
```

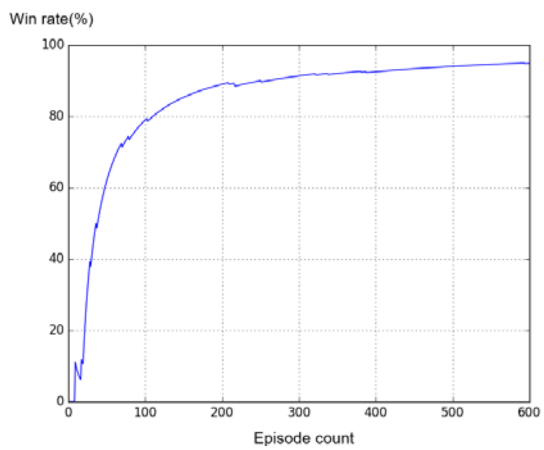
شکل ۱ شبه کد الگوریتم Q-Learning

همانطور که در شبه کد هم آمده است یک ماتریس داریم که ابعاد آن برابر تعئتئ state در تعداد action ها است. در هر باری که بازی انجام میشود، یک action به وسیله سیاست epsilon-greedy انتخاب می شود. و آن action به محیط اعمال می شود. و با توجه به state که پس از اعمال action در آن قرار می گیریم و reward ای که از آن action میگیریم مقدار q برای آن اقدام در آن وضعیت را طبق رابطه به روز می نماییم.

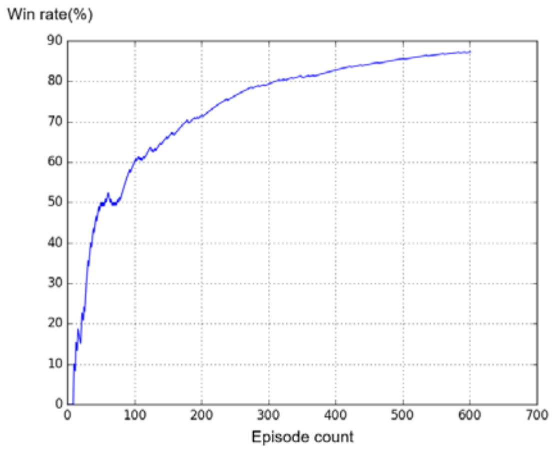
آنچه که در این الگوریتم می تواند مورد بررسی قرار گیرد ، سیاست epsilon greedy به ازای مقادیر مختلف epsilon است. و همینطور بررسی کردن اثر پارامتر نرخ یادگیری α (learning rate).

بررسی اثر تعداد مین

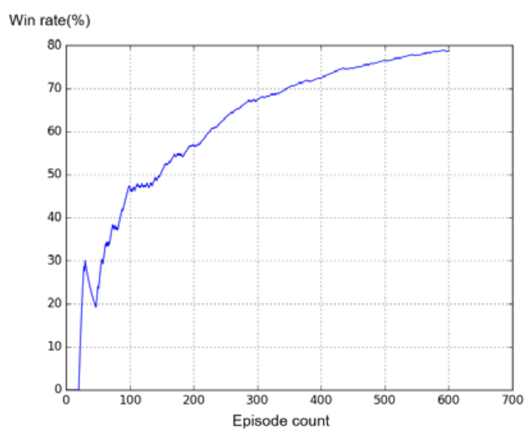
به منظور بررسی تعداد مین در صفحه بازی و نرخ یادگیری ، اندازه صفحه بازی را ۴ در ۴ در نظر گرفتیم و برنامه را به ازای تعداد مین مختلف اجرا کردیم. این کار را برای ۱ تا ۶ مین در این صفحه انجام دادیم و در هر بار سعی کردیم حدود ۶۰۰ episode بازی انجام شود که نتایج آن شکل های ۲ تا ۷ شده است. که این شکل ها در فولد output/figure موجود می باشند. در این قسمت مقدار epsilon در سیاست epsilon greedy را به صورت کاهش در نظر گفتیم و مقدار آن در هر مرحله برابر بود با $\frac{1}{n}$ که n برابر است با تعداد اقدام هایی که از اول بازی تا به حال عامل (Agent) انجام داده است. و مقدار α یعنی همان نرخ یادگیری را همین صورت همانند epsilon به کار گرفتیم.



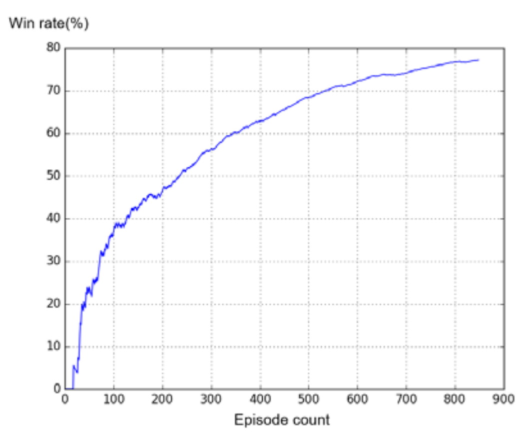
شکل ۲ نرخ برد به ازای ۱ مین در صفحه ۴ در ۴



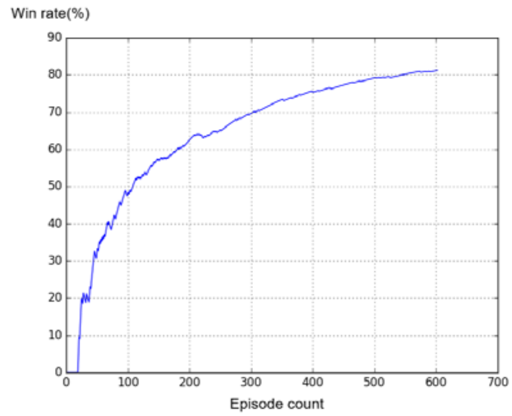
شکل ۳ نرخ برد به ازای ۲ مین در صفحه ۴ در ۴



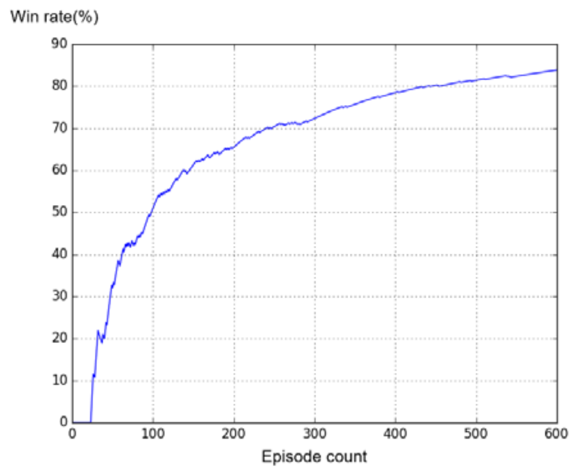
شکل ۴ نرخ برد به ازای ۳ مین در صفحه ۴ در ۴



شکل ۵ نرخ برد به ازای ۴ مین در صفحه ۴ در ۴



شکل ۶ نرخ برد به ازای ۵ مین در صفحه ۴ در ۴



شکل ۷ نرخ برد به ازای ۶ مین در صفحه ۴ در ۴

همانطور که انتظار می رفت در روند پیروزی تفاوت چندانی وجود ندارد اما از آنجا که وقتی تعداد مین ها زیاد می شود تعداد state ها هم زیاد میشود. پیروزی های پی در پی دیرتر شروع می شود چرا که برای انتخاب درست یک action باید episode های بیشتری گذشته باشد.

برای مقایسه بهتر نیز شماره episode که به طور متوسط روند برد آغاز میشد را ذخیره کردیم و جدول شماره ۱ حاصل شد. این اعداد به ازای ۱۰۰۰ بار اجرا بدست آمده است.

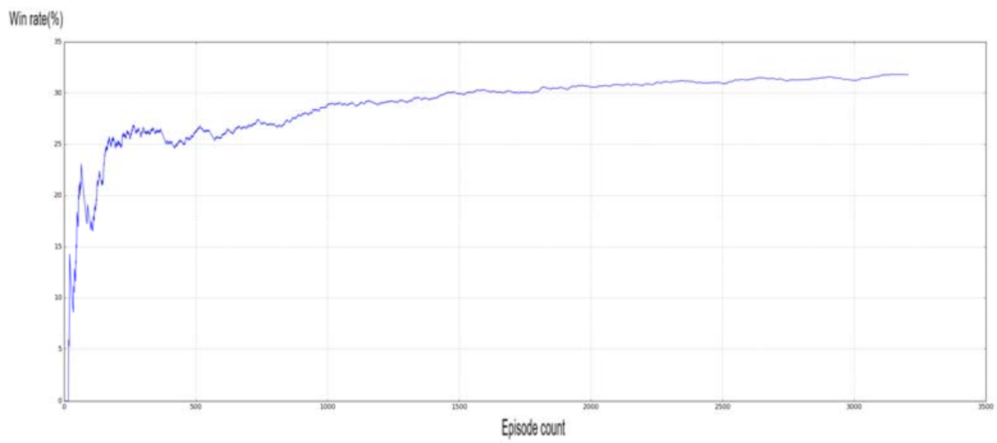
صفحه ۴ در ۴	اولین episode برد
۱ مین در بازی	۹,۲
۲ مین در صفحه	۱۱,۵
۳ مین در صفحه	۲۱,۲

۴ مین در صفحه	۲۵,۴
۵ مین در صفحه	۳۱,۱
۶ مین در صفحه	۳۴,۵

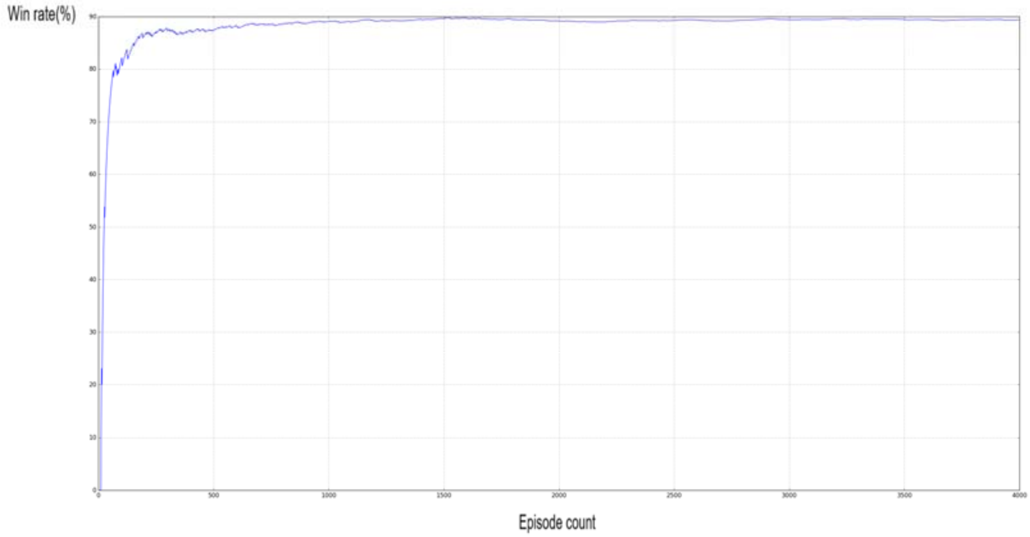
جدول ۱متوسط اولین episode برد به ازای ۱۰۰۰ بار اجرا

بررسی اثر epsilon در سیاست epsilon greedy به ازای نقشه های ثابت

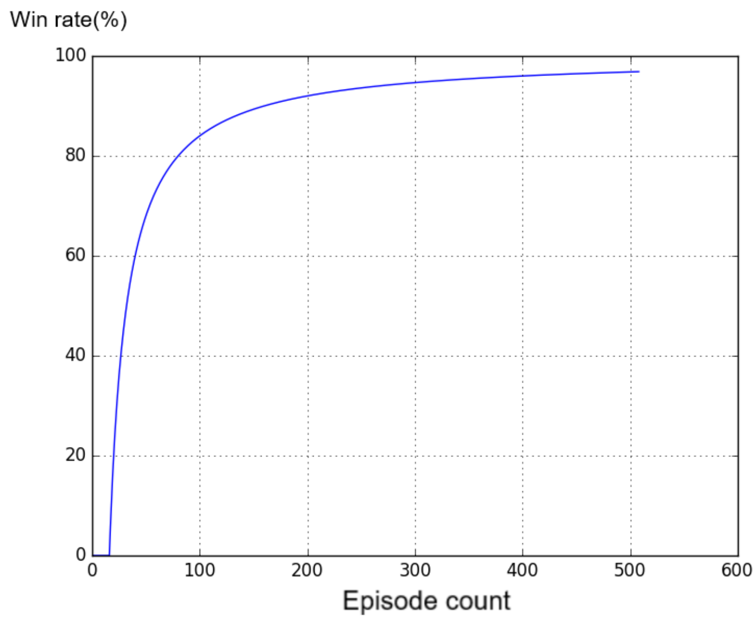
در این بخش به منظور بررسی اثر مقدار epsilon مقادیر α یعنی learning rate را مقدار ثابت ۰,۵ قرار دادیم و مقدار γ یعنی discount factor را نیز ۰,۵ قرار دادیم و سپس در یک بازی با ابعاد ۴ در ۴ و تعداد ۴ تا مین در صفحه ، بازی را به ازای مقادیر ۰,۱ و ۰,۰۲ و مقدار کاهشی برای epsilon اجرا کردیم. و این بازی را به ازای چند نقشه ثابت اجرا کردیم تا تفاوت قرارگیری مین ها اثر نگذارد. در شکل های ۸ تا ۱۱ میتوانیم نرخ پیروزی را ببینیم.



شکل ۸ نرخ برد به ازای epsilon برابر 0.1(نقشه ثابت)



شکل ۹ نرخ برد به ازای epsilon برابر ۰,۰۱(نقشه ثابت)



شکل ۱۰نرخ برد وقتی epsilon کاهشی است (نقشه ثابت)

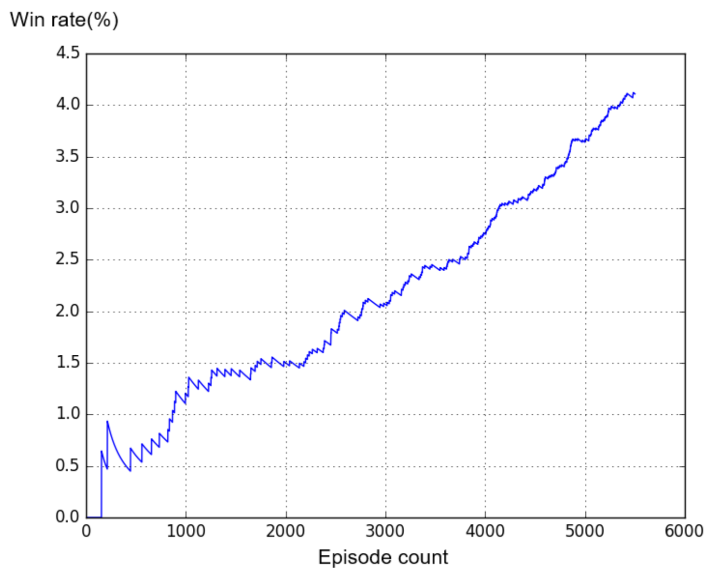
همانطور که در شکل۸ میبینیم وقتی epsilon برابر با ۰,۱ است احتمال انتخاب action های غیر بهینه بالاتر است برای همین عامل حتی با انجام ۵۰۰۰ دور از بازی تازه به نرخ ۳۰ درصد پیروزی رسیده است.

اما وقتی epsilon را کم کردیم و برابر ۰,۰۱ قرار دادیم همانطور که در شکل ۹ میبینیم خیلی سریعتر به جواب رسیدیم و با episode۳۰۰۰ به نرخ پیروزی ۹۰ درصد رسیدیم.

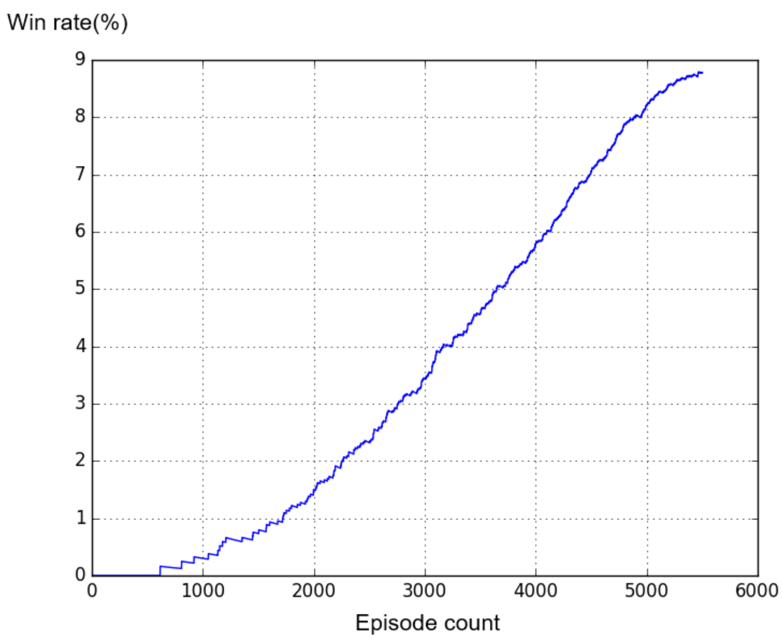
و در نهایت epsilon را یک مقدار کاهشی در نظر گرفتیم به این صورت که را تقسیم بر تعداد دفعاتی که عامل action انتخاب کرده بود میکنیم و با نرخ $\frac{1}{n}$ در حال کم شدن بود که باعث بهترین نتیجه ممکن شد. همانطور که در شکل ۱۰ دیده می شود در ۶۰۰ بار اجرا تقریبا به مقدار ۱۰۰ درصد پیروزی نزدیک می شود.

بررسی اثر epsilon در سیاست epsilon greedy به ازای نقشه های تصادفی

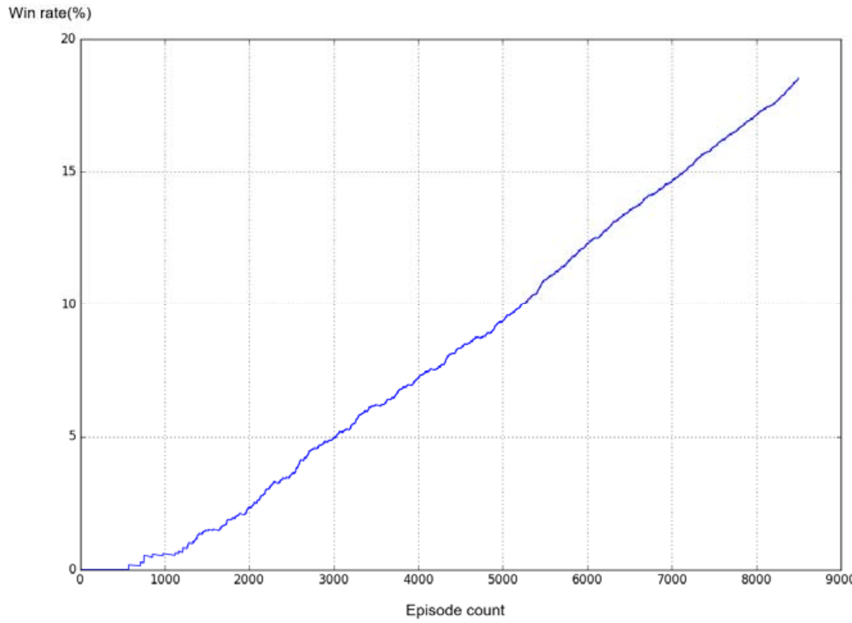
همان کاری که در بخش قبل کردیم را تکرار میکنیم با این تفاوت که در هر بار اجرا یعنی در هر episode یک نقشه تصادفی جدید تولید شود که در این صورت نتایجی مشابه با همان بخش قبل گرفتیم که در شکل های ۱۱ ، ۱۲ و ۱۳ نمایش داده شده است. همانطور که میبینیم وقتی epsilon مقدار زیادی دارد نوسان نرخ پیروزی بسیار زیاد است و بعد از حدود ۵۰۰۰ episode تازه به نرخ ۴ درصد رسیده است درحالی که با مقدار epsilon برابر ۰,۰۱ در episode شماره ۵۰۰۰ به نرخی بالاتر از ۸ درصد دست یافته ایم. اما نکته ای جالب تر اینجاست که وقتی ۰,۱ بود شروع پیروزی خیلی زودتر از حالتی است که epsilon = 0.01 است.



شکل ۱۱ نرخ برد به ازای **epsilon** برابر ۰,۱ (نقشه تصادفی)



شکل ۱۲ نرخ برد به ازای **epsilon** برابر ۰,۰۱(نقشه های تصادفی)

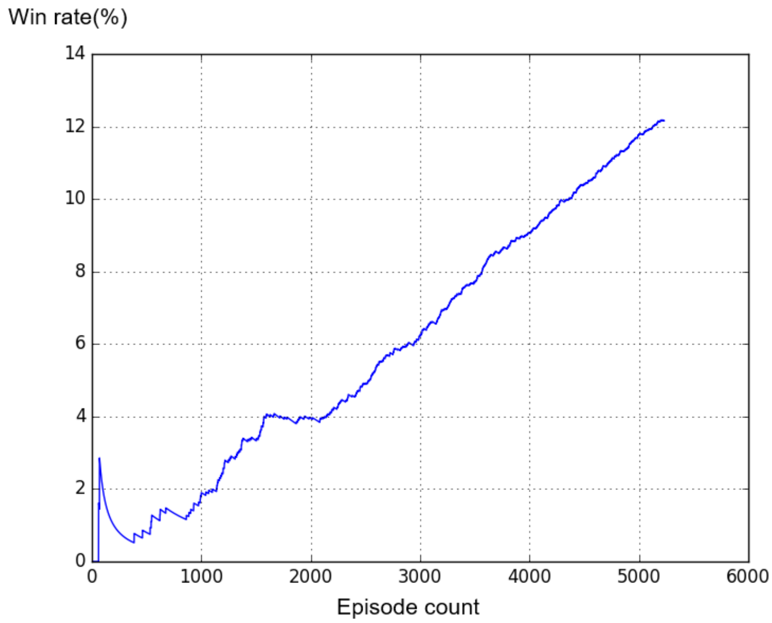


شکل ۱۳ نرخ برد به ازای **epsilon** کاهشی (نقشه های تصادفی)

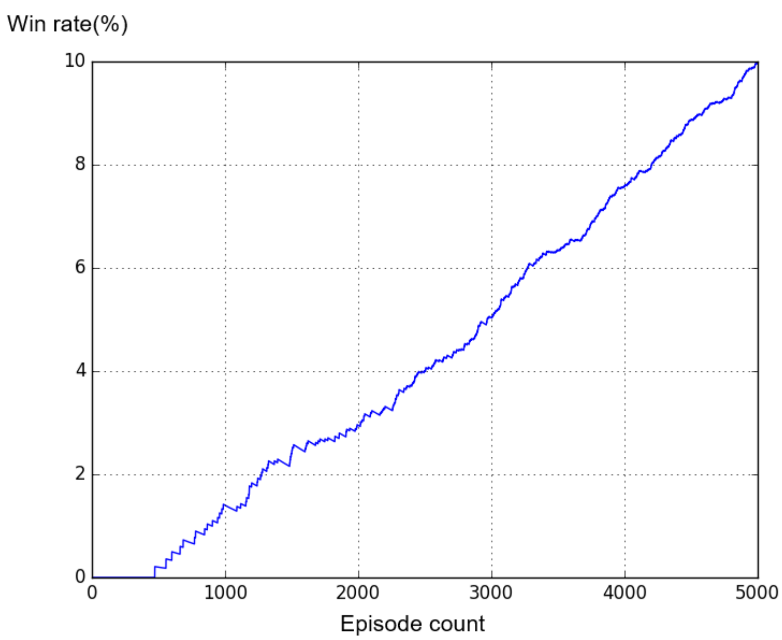
بررسی اثر α (learning rate)

در این قسمت به منظور برررسی اثر α ، بازی را حدود ۵۰۰۰ episode به ازای مقادی مختلف آلفا ، ۰,۱ ، ۰,۰۱, مقدار کاهشی اجرا کردیم و نمودار نرخ پیروزی بر اساس تعداد مراحل اجرا شده را بدست آوردیم که در شکل های شماره ۱۴ ،۱۵ و ۱۶ آورده شده است.

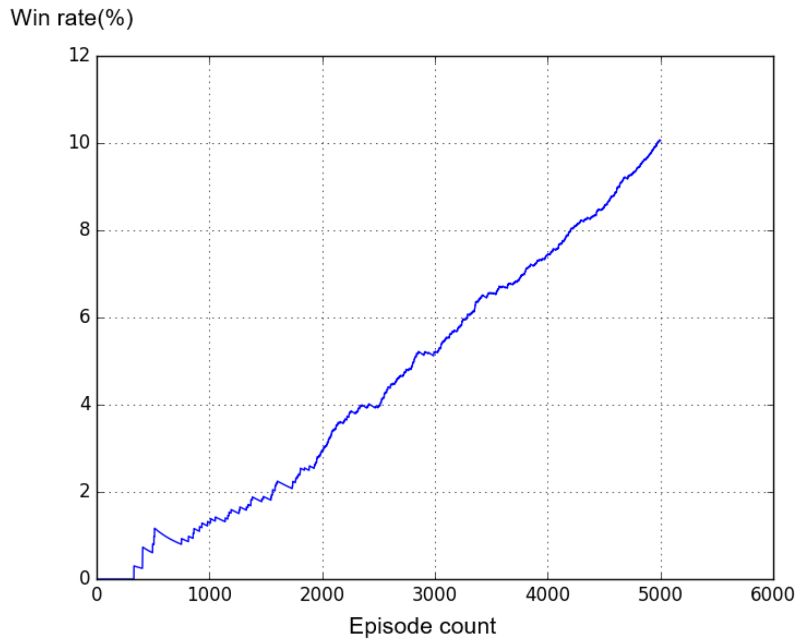
همانطور که در شکل ۱۴ مشخص است وقتی که آلفا بزرگ و ثابت انتخاب میشود روند پیروزی زودتر آغاز میشود چرا که به داده های reward و آینده توجه بیشتری دارد.اما وقتی خیلی کوچک انتخاب می شود پیروزی ها خیلی دیر اتفاق می افتد همانند شکل ۱۵. و در شکل ۱۶ میبینیم که نمودار گویا میانگینی بین دو نمودار دیگر است چرا که از شکل ۱۵ خیلی سریعتر روند پیروزی را آغاز کرده و هم نوسان کمتری را در انتخا متحمل شده است.



شکل ۱۴ نرخ برد وقتی آلفا ۰,۸ است



شکل ۱۵ شکل ۱۴ نرخ برد وقتی آلفا ۰,۱ است



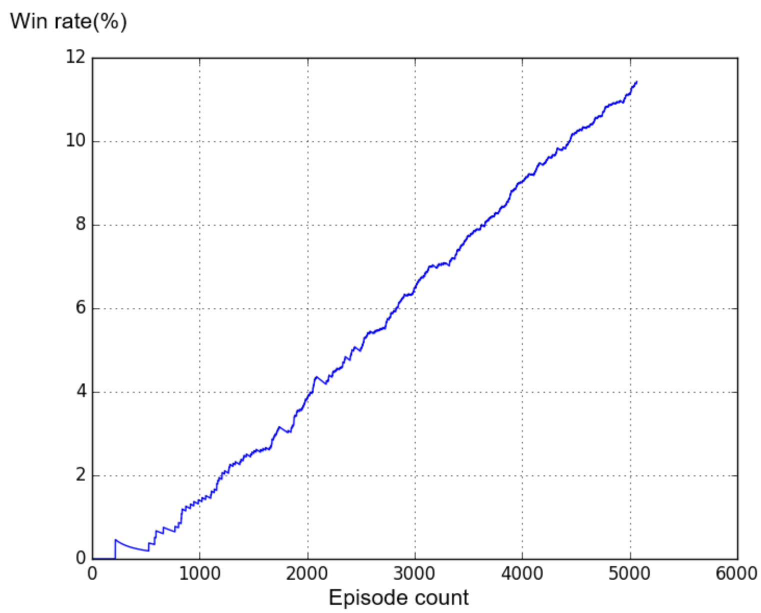
شکل ۱۶ نرخ برد وقتی آلفا کاهشی است.

اثر کاهش حالات متقارن

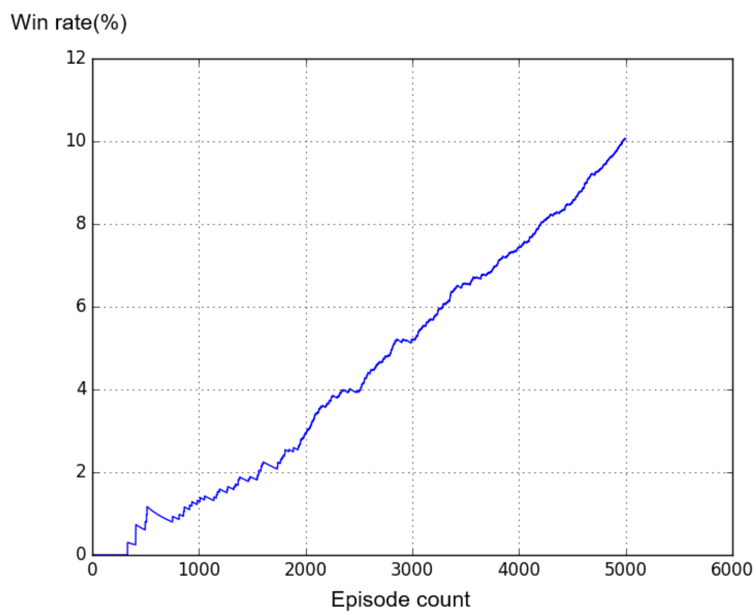
وقتی یک نقشه ۴ در ۴ داریم و یا حتی n در n ، خیلی اوقات اگر صفحه بازی را ۹۰،۱۸۰ و یا ۲۷۰ درجه بچرخانیم حالاتی بدست می آید که به نظر جدید میرسند آما در واقع همان حالتی است که چرخیده است. به همین منظور وقتی میخواهیم مقدار Q را برای یک action در یک state به روزرسانی کنیم. نقشه را با سه زاویه گفته شده دوران می دهیم و action معادل را نیز در آن نقشه ها نیز بدست می آوریم. سپس Q نقشه های دوران داده شده را هم به روز میکنیم. این کار باعث می شود که تعداد episode هایی که باید تجربه کنیم کمتر شود. این مسئه در عاملی که در فایل qLearnAgent2.py است پیاده سازی شده است.

سپس برای اینکه ببنیم چقدر این عامل بهتر عمل میکند. آلفا و epsilon را کاهشی قرار داریم و هر دو عامل را به ازای نقشه های تصادفی اجرا نمودیم که نمودار های شکل های ۱۷ و ۱۸ بدست آمد.

در شکل ۱۷ وقتی تعداد episode به ۵۰۰۰ میرسد نرخ پیروزی نزدیک به ۱۲ درصد می شود در حالی که در عامل پیشین که در شکل ۱۶ آمده است نرخ پیروزی تازه به ۱۰ درصد رسیده است.



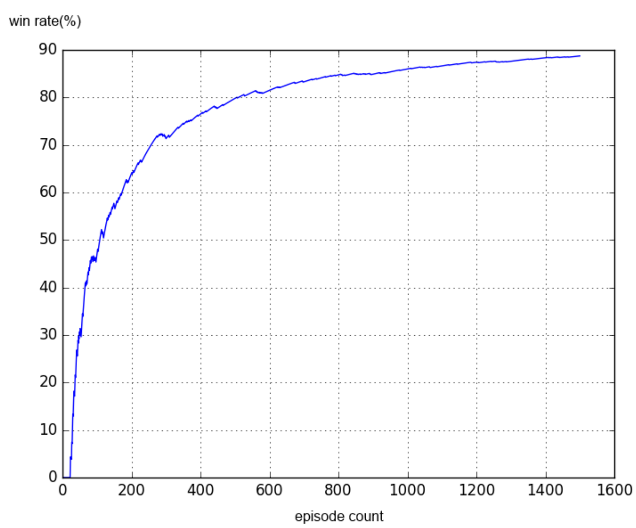
شکل ۱۷ نرخ پیروزی برای عاملی که حواسش به چرخش نقشه هست.



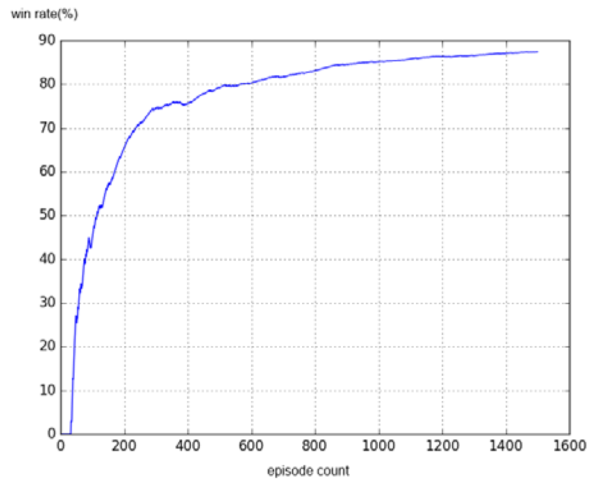
شکل ۱۸ نرخ پیروزی عاملی که چرخش نقشه را در نظر نمیگیرد

اثر discount factor

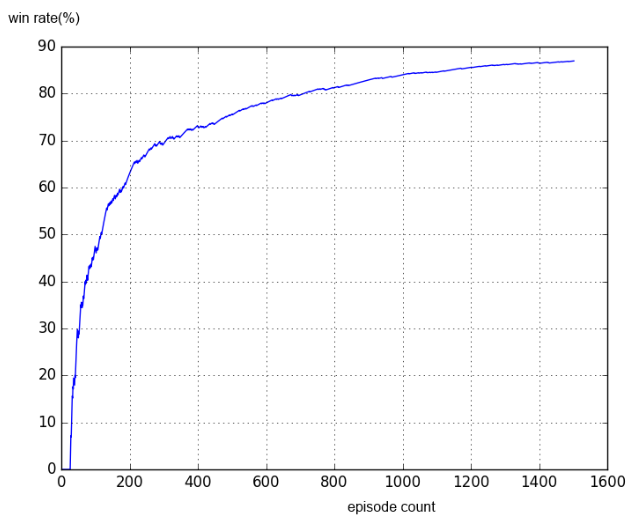
به منظور بررسی اثر γ یعنی discount factor این متغیر را مقادیر ۱ ، ۰,۵ و ۰,۱ گذاشتیم. و نمودار های حاصل شکل های ۱۹،۲۰ و ۲۱ شد که در نگاه سطحی تفاوت چندانی ندارند. وقتی گاما بیشتر شد نرخ افزایش برد با یک افت خیلی کمی روبه رو شد.



شکل ۱۹ نرخ برد به ازای گاما برابر ۰,۱



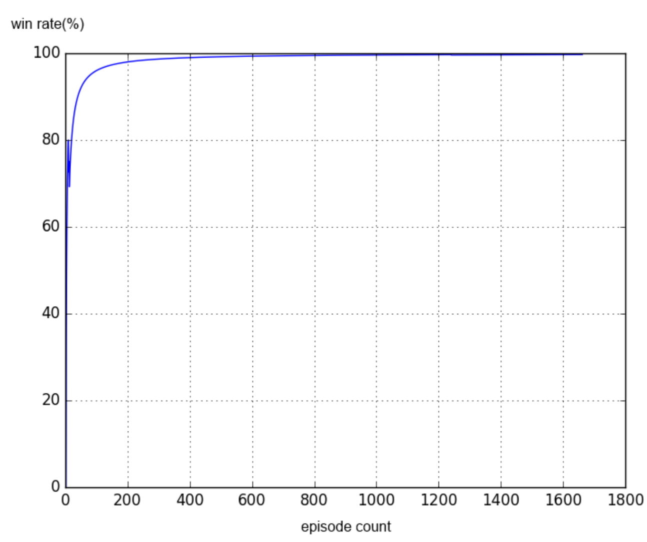
شکل ۲۰ نرخ برد به ازای گاما برابر ۰٫۵



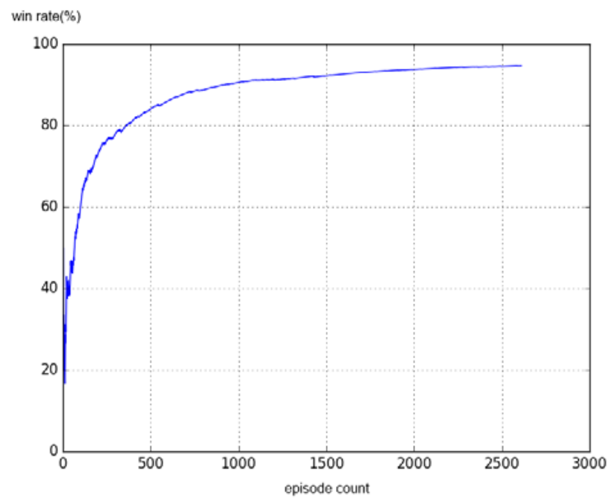
شکل ۲۱ نرخ پیروزی به ازای گاما برابر ۱

اثر افزایش بعد با ثابت بودن چگالی مین

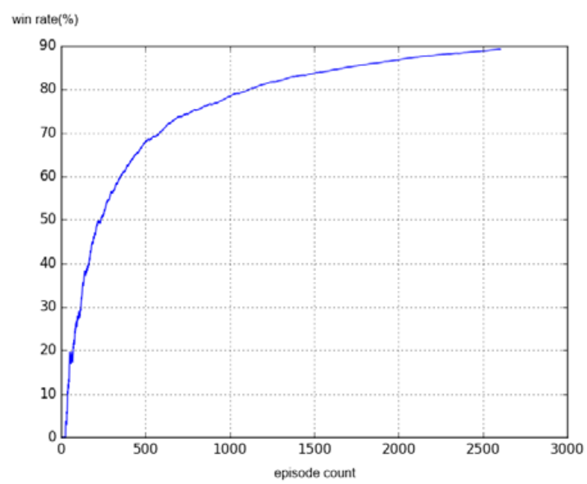
به منظور بررسی این مسئله، چگالی تعداد مین را ثابت برابر با ۰,۲۵ در نظر گرفتیم و به ازای صفحه 2x2 و 3x3 ، 4x4، 5x5 و 6x6 بازی را اجرا نمودیم که نمودار های شکل های ۲۲ تا ۲۶ را حاصل شد. همتنطور که از نمودار ها می توان نتیجه گرفت هرچه که ابعاد زمین را بزرگتر میکنیم شروع پیروزی نیاز به episode های بیشتری دارد و همینطور برای رسیدن به یک میزان برد کافی مثلا ۹۰ درصد نیاز به تعداد خیلی زیاد اجرای بازی می باشد. همانطور که در شکل۲۶ هم میبینیم عامل برای رسیدن به نرخ برد ۲۵ درصدی ۳۰۰۰۰ episode را طی کرده است.



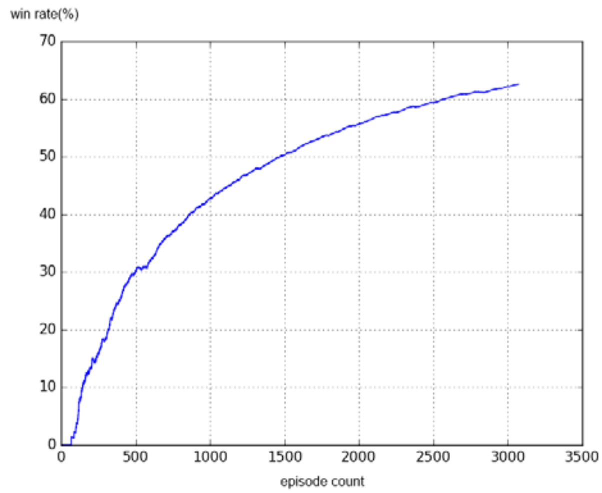
شکل ۲۲ نرخ پیروزی وقتی صفحه 2x2 است



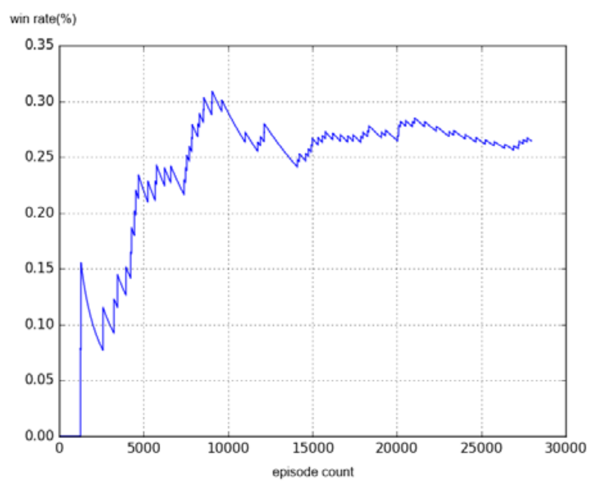
شکل ۲۳نرخ پیروزی وقتی صفحه 3x3 است



شکل ۲۴ نرخ پیروزی وقتی صفحه 4x4 است



شکل ۲۵ نرخ پیروزی وقتی صفحه 5x5است



شکل ۲۶ نرخ پیروزی وقتی صفحه 6x6است

راه حل پنجره متحرک برای حل مسئله با ابعاد بزرگتر

در این قسمت اگر بخواهیم همانند قبل ماتریسی داشته باشیم از حالات و اقدامات و به همان منوال مقادیر Q را به روز رسانی کنیم شاید برای یک خانه ۶در ۶ حتی پس از ساعت ها نیز به جواب نرسیم.

ایده ای که ما برای حل این مسئله زدیم این بود که بدون در نظر گرفتن ابعاد بازی، یک پنجره ۴ در ۴ در نظر بگیریم. به عنوان مثال ابتدا ۱۶ خانه سمت چپ بالای نقشه اصلی را حل کنیم و سپس تحت شرایطی که در ادامه ذکر میکنیم آن پنجره را به راست و یا پایین انتقال بدهیم.

ابتدا پاداش انتخاب هر خانه را اینطور در نظر گرفتیم که اگر آن خانه مقدار صفر داشت ارزش آن اقدام برابر ۹ و اگر یک بود برابر ۸ و .... . و اگر مین بود پاداش 9- را بگیرد.

برای انتقال پنجره نیز این شرط را در نظر گرفتیم که اگر مجموع مقادیر Q برای خانه های دیده نشده یعنی Q برای action های وضعیت فعلی منفی بود باید پنجره راshift دهیم وگرنه به بازی در آن پنجره ادامه می دهیم.

با پیاده سازی این الگوریتم که در فایل highDimensionAgent.py می باشد به این نتیجه رسیدیم که برای هر نقشه عامل به طور متوسط برای یک نقشه ۶ در ۶ با ۷ مین باید ۵۵ بار بازی کند.

راه حل های ابتدایی برای حل مشکل نفرین ابعاد

با توجه به اینکه تعداد مین ها محدود است، می توان هر حالت از بازی را با شماره ی خانه هایی که حاوی مین هستند وcover بود یا نبودن سایر خانه ها نشان داد. وضعیت اگر هر خانه را با مقادیر {uncovered, 0, 1, ..., 8} بخواهیم در جدول حالت نشان دهیم، برای یک جدول N*N که دارای M مین است، فضای حالت دارای حداکثر $\binom{N^2}{M} * 2^{N^2-M}$ حالت است (انتخاب M از N^2 خانه برای مین و 2^{N^2-M} حالت برای cover بودن یا uncover بودن خانه هایی که مین نیستند).

اولین مشاهده ای که داشتیم، این بود که می توان هر صفحه از بازی را با زوایای ۹۰ و ۱۸۰ و ۲۷۰ درجه دوران داد و همچنین می توان صفحه را وارون (ترانهاد) نمود و ترانهاده را نیز می توان با همان زوایا دوران داد بدون اینکه در استراتژی بهینه تغییری ایجاد شود (طبیعتا استراتژی بهینه را نیز دوران می دهیم).

در برای اکثر حالت های بازی بجز آنها که متقارن هستند، هر ۸ حالت در یک کلاس هم ارزی قرار می گیرند و اندازه ی فضای حالت یک هشتم می شود.

همچنین که حتی پس از حذف حالت های هم ارزی دورانی، تعداد زیادی از حالت ها invalid هستند و در هیچ وضعیتی از قرارگیری مین ها و بازی کاربر، به آن حالت ها نمیرسیم زیرا دارای تناقض در تعداد مین های خانه ها هستند.

در قسمت #PART1 از فایل state_representation.py، تعداد حالات (با حذف حالات معادل دورانی و وارونی) قرار گیری مین ها و اعتبار سنجی ممکن بودن حالت در بازی واقعی، تعداد حالات مختلف uncover شدن خانه ها به ازای قرار گیری های مختلف برای مقادیر مختلف M و N محاسبه شده است که خلاصه نتایج آن در جدول زیر آمده است:

جدول ۲ مقایسه نتایج اعتبارسنجی و یکسان سازی دوران ها در تعداد حالات

تعداد معتبر حالات بازی (مستقل از چخش)	$\binom{N^2}{M} * 2^{N^2-M}$	M	N
292	2304	1	3
451	4608	2	

	3	5376	421
	4	4032	268
	8	18	4
4	1	524288	58493
	2	1966080	186840
	3	4587520	361664
	4	7454720	482308
	5	8945664	472453
	6	8200192	356076

همانگونه که در جدول ۲ مشاهده می شود، این دو تکنیک ساده (یکسان سازی دوران/ترانهاده و اعتبار سنجی حالت) حدود ۲۰ برابر تعداد حالات بازی را کاهش می دهد بدون آنکه هیچ اطلاعات مهمی را از حالت حذف کند.

استفاده از توابع numpy برای سرعت بخشیدن به محاسبات

با توجه به اینکه در زبان پایتون loop های طولانی کندتر از پیاده سازی الگوریتم های مشابه در زبانهای C و C++ اجرا می شوند، تا جای ممکن در فایل state_representation.py از توابع کتابخانه ی numpy استفاده نمودیم و عملیات را به صورت ماتریسی انجام دادیم. زیرا عملیات ماتریسی numpy با واسطه ی کتابخانه هایی که به زبان C نوشته شده اند و performance بسیار بالاتری از loop های پایتون دارند اجرا می شود و این تکنیک کمک کرد بتوانیم تعداد حالات ابعاد بالای صفحه را در مدت زمان بسیار کمتری بررسی کنیم.

مثلا برای شمارش تعداد مین های مجاور هر کدام از خانه های یک حالت جدول، با استفاده از دستور board == MINE یک ماتریس N*N که هر خانه ی آن صفر یا یک است را تولید نموده و با convolve کردن فیلتر $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ بر روی آن با استفاده از numpy، تعداد مین های مجاور تمام خانه ها را با یک convolution و بدون loop محاسبه نمودیم. اکثر محاسبات انجام شده به صورت ماتریسی انجام شده اند.

راه حل توصیف محلی مولفه ها حل مشکل نفرین ابعاد

برای توصیف بهتر راه حل، برای هر حالت از بازی که در آن تعدادی از خانه ها uncover شده اند (منظور از uncover شدن خانه ها این است که روی آن ها در مراحل قبلی کلیک شده و تعداد مین اطراف آن ها مشخص است). اگر خانه های uncover شده ی جدول را به صورت رئوس یک گراف مدل کنیم و بین خانه های مجاوری که uncover شده اند یال قرار دهیم، گرافی حاصل می شود که دارای تعدادی مولفه ی همبدنی است.

با بررسی حالت های مختلف وضعیت بازی و گراف معادل با آن و راه حل بهینه برای هر کدام از حالت ها، مشاهده نمودیم که جابجا نمودن یا چرخاندن یا وارونه کردن یک مولفه ی هم بندی در جدول، به شرطی که آن مولفه با سایر مولفه های هم بندی حداقل ۲ خانه فاصله داشته باشد (منظور از فاصله ی دو مولفه ی هم بندی در اینجا، مینیمم فاصله‌ی ممکن بین یک خانه از مولفه ی اول با یک خانه از مولفه ی دوم است)، تاثیری در استراتژی بهینه برای حل آن حالت از بازی ندارند زیرا آنقدر تعداد حالات قرار گیری مین ها درحالتی که حداقل فاصله ی ۲ خانه رعایت شود زیاد است که عملا موقعیت هر کدام مولفه ها و چرخش آن ها تا زمانی که ساختار محیط مولفه و مقادیر روی محیط تغییر نکند، در انتخاب استراتژی بهینه بی تاثیر است.

اما اگر مولفه های همبندی نزدیک به هم باشند، استراتژی خانه های بین این دو مولفه قابلیت بهبود دارد و بهتر است در نمایش حالت، این را در نظر بگیریم.

در نتیجه اولین ایده ای که برای بهبود مشکل نفرین ابعاد مطرح شد، این بود که به جای آنکه وضعیت تمام خانه های جدول را در state نگه داریم، حالت هر یک از مولفه ها (مقادیر روی محیط آن و شکل لبه های محیط آن) را در state نگه داریم و حالت خانه های داخلی مولفه ها و علاوه بر آن، خانه هایی که هنوز uncover نشده‌اند، حاوی اطلاعات بسیار کمی هستند و می توان با حذف آن ها از state، تعداد حالت ها را کاهش و generalization را نیز افزایش دهیم. (از دانش سیاست بهینه برای یک شکل خاص از مولفه ی همبدی می توان در تمام موقعیت ها و چرخش های آن استفاده نمود).

در نتیجه حالت یک بازی، شامل حالت مولفه های همبندی آن است و این موقعیت قرارگیری و چرخش آن ها در حالت قرار نمی گیرد. همچنین فقط خانه های محیط هر مولفه حاوی اطلاعات هستند.

این نحوه ی نمایش حالت، به جای آنکه وابسته به ابعاد صفحه ی بازی باشد، به ابعاد مولفه های همبندی خانه های uncover شده در طول بازی وابسته است و در نتیجه در ابتدای بازی که مولفه ها کوچک هستند،

بسیار کارآمد است و می توان از دانشی که برای ارزش مولفه ها در یک ابعاد از جدول داریم ، در ابعاد دیگر هم استفاده کنیم.

برای نمایش هر مولفه ی همبندی، می توان از نقطه ای روی محیط آن شروع به حرکت کرده و در هر مرحله مقدار آن خانه و جهت حرکت برای رسیدن به خانه ی بعدی را نمایش دهیم تا کل مولفه بدست آید. البته بسته به محل شروع و جهت حرکت، نمایش های تکراری برای یک مولفه وجود خواهند داشت که با انتخاب جهتی مشخص (مثلا ساعتگرد) و قاعده ای برای انتخاب خانه ی شروه (مثل بزرگترین عدد) می توان از تعداد تکرار ها کاست.

راه حل توصیف محلی خانه‌ها برای حل مشکل نفرین ابعاد

راه حل توصیف محلی مولفه ها که در بخش قبل ارائه شد، تعداد حالات را به طور قابل توجهی کاهش می دهد و مستقل از ابعاد جدول است. اما به ابعاد خود مولفه های همبندی وابسته است و با افزایش ابعاد جدول و پیشرفت بازی، تعداد حالت های مولفه های همبندی هم بسیار زیاد گشته و عملا برای در صفحه هایی با ابعاد 10x10 به بالا، یادگیری Q-Value برای حالت هایی که بیشتر از ۵۰ درصد خانه های صفحه پر شده باشند یرعملی است تعداد حالت های مولفه هایی با محیط بزرگتر از ۱۵ عملا غیر ممکن است.

با بررسی راه حل های مبتنی بر CSP (Condition Satisfying Problem)، و شرط هایی که برای خانه ها اعمال می شوند و راه حل هایی که برای آن ها پیدا می شوند، مشاهده کردیم که تعداد زیادی از جواب های CSP، برای پیدا شدن مقدار، فقط به متغیرهای خانه های مجاور وابسته اند و جواب های local به خانه های مجاور یک سلول هستند و حتی از مولفه ی همبندی که آن سلول در آن قرار دارد هم مستقل اند. یعنی نهایتا با مشخص بودن مقادیر تا شعاع ۱ الی ۲ خانه‌ای مجاور یک سلول که در محیط یک مولفه قرار دارد، در تعداد زیادی از حالات می توان مساله ی CSP را برای آنکه آن خانه مین هست یا خیر حل نمود.

این مشاهده ما را به سمت توصیف محلی خانه های uncover نشده هدایت کرد. در این روش توصیف، مهم نیست که هر خانه در کجای صفحه قرار دارد، بلکه فقط مهم است مقادیر خانه های تا شعاع ۱ الی ۲ در اطراف هر خانه چه می باشند.

پس حالت یک صفحه از بازی را می توان با مجموعه ی حالت های خانه های uncover نشده ی آن (شامل حالت شعاع نمایش داد و مهم نیست که این خانه ها چطور در کنار هم قرار گرفته اند.

ایده ی دیگری که ما استفاده کردیم و در قسمت Part 2# از فایل state_representation.py آن را پیاده‌سازی کردیم، این بود که با توجه به اینکه Q(s,a) را یاد می گیریم و فضای ضرب دکارتی state و action ها ارزش دهی می شود، حالت خانه ها را به جای state، در action ها نمایش دهیم. اگر شرط محلی بودن راه حل برای هر خانه که در پاراگراف های قبل مطرح شد برقرار نبود، این عمل توجیه نداشت. اما با توجه به برقراری شرط محلی بودن راه حل در خیلی از موارد، مساله ی minesweeper بیشتر شبیه به مساله ی n-armed bandit می باشد و می توان آن را با یک state هم مدل کرد و هر arm می شود یکی از خانه هایی که هنوز uncover نشده است و حالت خانه های تا شعاع ۲ برای آن خانه نیز در توصیف آن arm لحاظ می شود. در عوض، در state می توانیم ماکسیمم عددی که با uncover کردن سلول ها دیده ایم را قرار دهیم و این به

agent کمک می کند در ابتدای بازی، exploration بیشتر انجام دهد و خانه های random را بیشتر امتحان کند و هرچه پیش می رود، خانه هایی که روی محیط هستند را انتخاب کند.

1	2	1		
	3			
	4	1		

برای توضیح بهتر، یک مثال از حالت یک خانه (با زمینه ی آبی) و خانه های مجاور آن با فاصله ی ۲ را در شکل بالا آورده ایم. همانطور که مشاهده می شود، با توجه به اینکه خانه ی سمت چپ خانه ی آبی که مقدار ۳ دارد، تنها ۳ همسایه ی uncover نشده دارد، حتما هر ۳ همسایه ی آن از جمله خانه ی آبی رنگ مین هستند و انتظار داریم agent در طول زمان، چنین قواعدی را یاد بگیرد. این قواعد که از مقادیر با شعاع ۲ حاصل می شوند، در اکثر حالت های بازی یافت می شوند و منجر به پیشرفت agent در بازی می شود.

در نتیجه در هر مرحله از بازی، به تعداد خانه های uncover نشده، action دارد که هر action شامل توصیف خانه های با شعاع ۲ می باشد.

با اینکه توصیف بالا بسیار راهگشاست، اما هنوز با مشکل تعداد حالات زیاد مواجه هستیم. حتی با حذف حالت های غیر معتبر و حالت های تکراری ناشی از چرخش یا ترانهاده نمودن حالت، برای شمارش حالت ها به چند ساعت محاسبه نیاز داریم. آخرین ایده ای که توانستیم نهایتا در state_represenation.py به کمک آن مشکل نفرین ابعاد را حل کنیم، این بود که فقط ۸ خانه ی با شعاع ۱ را در حالت نمایش دهیم اما برای هر همسایه، علاوه بر مقدار آن، تعداد خانه هایی که هنوز uncover نشده اند را نیز نمایش می دهیم. مثلا شکل بالا را به صورت زیر نمایش می دهیم که عدد داخل پرانتز نمایش گر تعداد خانه های uncover نشده ی مجاور هر همسایه (اطلاعات فشرده ای از وضعیت خانه های با شعاع ۲ از خانه ی آبی رنگ) می باشد.

2(5)	1	?(7)
3(3)		?(6)
4(6)	1(6)	?(7)

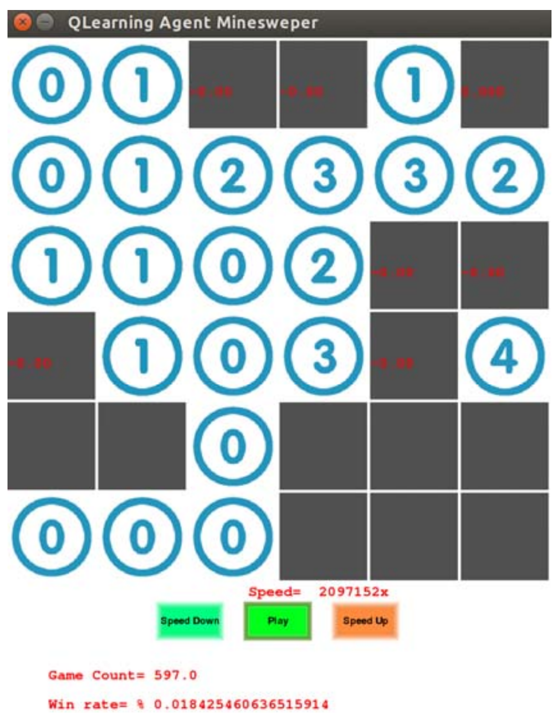
مزیت های راه حل توصیف محلی خانه‌ها برای حل مشکل نفرین ابعاد

- تعداد حالت ها در عین حفظ بیشترین اطلاعات برای حل مساله، کاهش قابل ملاحظه ای یافت به طوریکه برای تمام حالت های مختلف یک صفحه ی ۴ در ۴ با ۴ مین، تنها ۱۴۵۲۲۴ حالت برای هر خانه داریم که این تعداد با یک کامپیوتر شخصی در عرض چند دقیقه به یادگیری قابل قبولی می رسد.
- تعداد حالت های یک خانه (تعداد action ها)، حداکثر تا ابعاد ۵ در ۵ با ۲۴ مین افزایش می یابد و از آن به بعد، هر چقدر هم تعداد ابعاد صفحه بزرگ شود (مثلا 50x50) ، این تعداد ثابت می ماند و مشکل نفرین ابعاد ریشه کن شده است.
- از Q-Value های بدست آمده برای یک ابعاد مشخص صفحه ی بازی با این نمایش، می توان در ابعاد دیگر نیز استفاده نمود.
- در صورت پروژه ذکر شده بود بهتر است اولین حرکت بازی، در نقطه ی گوشه ی صفحه انجام گیرد زیرا احتمال آنکه مقدار آن صفر باشد بیشتر است. در این نمایش، اگر reward برای مقدار صفر را بیشتر قرار دهیم، agent به صورت خودکار پس از چند بار شروع بازی این حقیقت را یاد می گیرد و نیازی نیست مستقیما به آن یاد دهیم. زیرا وضعیت border های اطراف خانه ها هم در state representation پیاده سازی شده موجود می باشد.

پیاده سازی

در این بخش سعی می کنیم آنچه را که برای پیاده سازی این بازی انجام دادیم را به طور خلاصه بیان نماییم.

ابتدا سعی کردیم برای آنکه دید بهتری روی انجام بازی داشته باشیم یک GUI برای این بازی طراحی کنیم تا بتوانیم روند اجرای الگوریتم یادگیری ماشین خود را بهتر مورد بررسی قرار دهیم. ظاهر آنچه برای بازی طراحی نمودیم به صورت شکل ۲۷ است که خانه هایی طوسی رنگ به معنای خانه هایی است که هنوز انتخاب نشده اند. و خانه هایی که عدد هستند نشان می دهند که اطراف آنها چه تعداد مین وجود دارد. و سعی کردیم مقادیرQ به ازای هر action را روی ظاهر برنامه به نمایش درآوریم.



شکل ۲۷ GUI بازی minesweeper

مکانیزم اجرای بازی به عهده کلاسی است که در فایل gui.py نوشته شده است. و ظاهر بازی نیز در این قسمت پیاده سازی شده است. و اگر بخواهیم تغییری در تعداد مین ها ، ابعاد بازی بدهیم باید به این کلاس مراجعه شود. برای پیاده سازی GUI از کتابخانه pygame استفاده کردیم. در این کلاس تابعی دیده می شود به

نام generate_mines . که این تابع با توجه به تعداد مین ها و همینطور احتمال clear بودن هر خانه که به صورت زیر است ، موقعیت آنها را تعیین میکند.

$$p(x_{ij} = 0 | ij = \text{corner}) = (1 - d)^4$$
$$p(x_{ij} = 0 | ij = \text{edge}) = (1 - d)^6$$
$$p(x_{ij} = 0 | ij = \text{inside}) = (1 - d)^9$$

تابع دیگری در این کلاس به نام generate_map وجود دارد که با توجه به موقعیت مین هایی که تعیین شده است مقادیر مورد نظر را به هر خانه می دهد.

تابع دیگر نیز به نام replay_game وجود دارد که با فراخوانی آن می توان همان map قبلی را از اول بازی کرد.

و برای ایجاد یک نقشه جدید می توان از تابع reset_game استفاده کرد.

تابع مهم دیگری که در این کلاس وجود دارد تابع select_tile است که اقدامی که عامل انتخاب کرده است را به محیط ارائه می دهد و نتیجه برد باخت و پاداش را برمیگرداند.

پیاده سازی عامل هم در فایلی به نام qLearnAgent.py انجام شده است. که دوتابع مهم دارد که یکی از آنها action را بر اساس policy انتخاب میکند. و دیگری Q-value را به روز رسانی میکند.

منابع

۱- http://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html

۲- مقاله luis gardia, Griffin koontz , Ryan silva