گزارش تمرین شماره دو

نام و نام خانوادگی	امیرمحمد رنجبر پازکی
شماره دانشجویی	۸۱۰۱۹۹۳۴۰

سوال 1 – مساله 10-armed bandit

برای حل این مساله ابتدا یک environment پیاده سازی شد. این environment با نام MultiArmedBandit از کلاس EnvironmentBase مشتق شده‌است و ۱۰ arm دارد که هر کدام توزیع reward گاوسی دارند. این توزیع‌ها در کلاس GaussianReward پیاده‌سازی شده‌اند. توزیع پاداش هر کدام از این armها دارای واریانس ۱ و میانگینی هستند که از توزیعی با میانگین ۰ و واریانس ۱ می‌آید.

هسته اصلی این سوال پیاده‌سازی عامل و اجرا در محیط بود. عامل موردنظر در کلاس MultiArmedBandit Agent پیاده‌سازی شده‌است. این عامل از روش Thompson smapling استفاده می‌کند. به این صورت که به ازای امید ریاضی پاداش هر عمل q*(a) یک میانگین و معکوس واریانسπ در خود نگاه می‌دارد که باور عامل نسبت به توزیع امید ریاضی پاداش هر عمل است. حال در هر trial، عامل نمونه‌ای از این توزیع‌ها بر می‌دارد. هر کدام که بیشتر باشند، در لحظه به عنوان عمل بهینه انتخاب می‌شوند و انجام می‌شوند. پس از انجام عمل و گرفتن پاداش، باور عامل نسبت آماره‌های امید ریاضی پاداش عمل انجام شده به‌روزمی‌شود. برای به‌روزرسانی این آماره‌ها از قاعده‌ی بیز استفاده می‌شود.

در فایل Q1_multi_armed_bandit.py همه عوامل در کنار یکدیگر یک محیط را شبیه‌سازی کرده‌اند که در آن عامل طی ۱۰۰۰ آزمایش سعی می‌کند تا عمل بهینه را پیدا کند. این کار ۲۰ بار تکرار می‌شود تا نتایج به صورت آماری گزارش شوند و دقیق‌تر باشند. در انتهای هر بار اجرا میزان انتخاب عمل بهینه و regret یا همان حسرت محاسبه می‌شود.

عمل بهینه عملی است که در نهایت بزرگترین میانگین امید ریاضی را داشته‌باشد. حسرت به صورت مجموع اختلاف امید ریاضی پاداش اعمال انتخاب شده در هر یک از ۱۰۰۰ آزمایش با امید ریاضی عمل بهینه تعریف می‌شود.

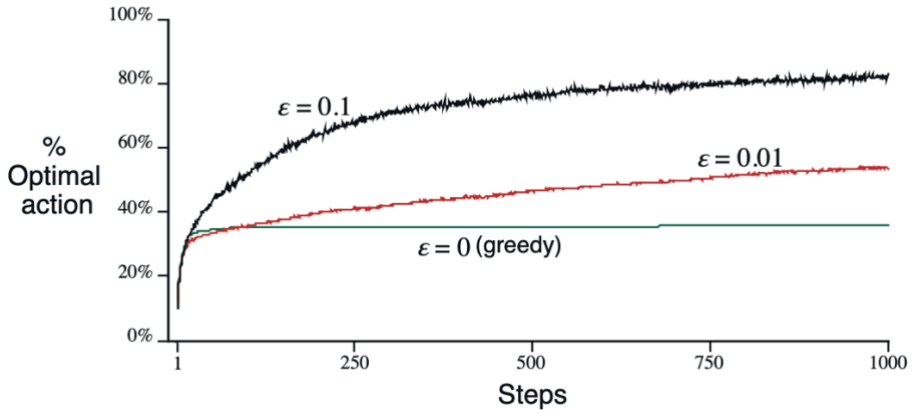
اعداد زیر میانگین و انحراف معیار به دست آمده به ازای ۲۰ بار تکرار آزمایش با روش Thompson sampling هستند.

```
Regrets:                        mean: 45.78910066381238 std: 8.88595570943907
Optimal action percentage:      mean: 0.97485   std: 0.005824731753480161
```

همانطور که مشاهده می‌کنید، به طور میانگین در ۹۷.۴۸ درصد مواقع عمل بهینه انتخاب شده‌است و انحراف معیار این معیار ۰.۵۸ درصد است. این درصد، درصد مناسب و بالایی است.

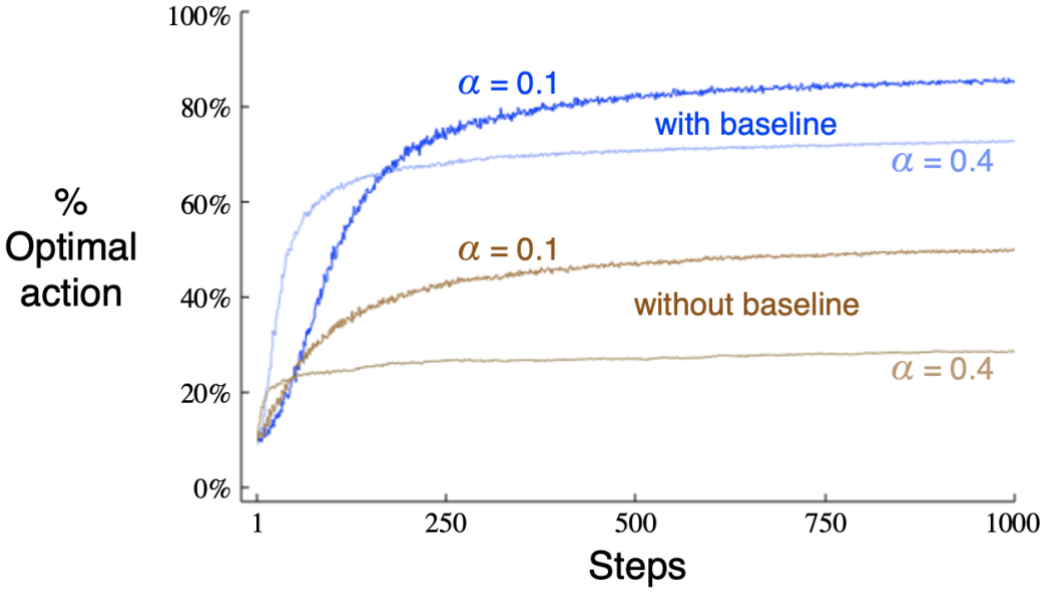
همچنین، میزان حسرت(اختلاف تا عمل بهینه) در ۱۰۰۰ آزمایش به طور میانگین ۴۵.۷۸ با واریانس ۸.۸۸ بوده‌است.

اولین روش معرفی شده در کتاب، روش e-greedy است که در بهترین حالت و حالت حدی در ۸۰ درصد موارد عمل بهینه انتخاب شده‌است که روش Thompson sampling عملکرد بهتری نسبت به آن داشته‌است.



روش بعدی که در کتاب به آن پرداخته شده‌است، روش gradient bandit است. همانطور که در شکل زیر می‌بینید، در بهترین حالت و حالت حدی، درصد انتخاب عمل بهینه حدودا زیر ۹۰ درصد است که این موضوع باز هم حکایت از عملکرد بهتر روش Thompson sampling دارد.

دلیل این موضوع استفاده از باور و به روز رسانی باور برای پیداکردن عمل بهینه است. در این روش کمتر کورکورانه عمل می‌کنیم و هدایتگر بهتری داریم.



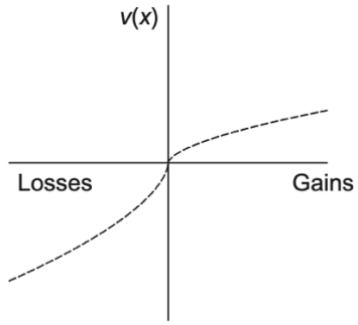
سوال ۲ – رفتن به دانشگاه

الف) در این مدل، عمل به صورت مقابل تعریف می‌شود: میزان صبر کردن فرد به دقیقه

با توجه به این‌که اتوبوس‌ها با توزیع گاوسی با میانگین ۸ و انحراف معیار ۳ می‌آیند، باز زمانی صبر کردن از ۰ تا ۱۷ دقیقه می‌تواند باشد.(در فاصله سه انحراف معیار از میانگین ۹۹ درصد داده‌ها قرار می‌گیرند. داده‌ی منفی برای زمان قابل قبول نیست.) پس یک محیط ۱۸ عمله داریم.

میزان پاداش باید بر اساس اختلاف زمان صبر کردن و زمان رسیدن اتوبوس تعیین شود. هر چه اتوبوس زودتر از انتظار به ایستگاه برسد، پاداش مثبت و بیشتری می‌گیریم. اگر اتوبوس دیرتر از حد انتظار برسد، پاداش منفی است و هر چه این مقدار بالاتر باشد، پاداش منفی‌تر است. برای پیاده سازی این مورد پاداش موردنظر صرفا زمان رسیدن اتوبوس را برمی‌گرداند و این به دلیل آن است که در این جا نیاز به utility داریم و زمان رسیدن اتوبوس برای محاسبه utility نیاز است و صرفا اختلاف کارساز نیست. به عنوان مثال، اگر زمان رسیدن اتوبوس از زمان مرزی رسیدن به کلاس (۱۵ دقیقه انتظار) بیشتر باشد، به صورت ذهنی اثر منفی به شدت زیادی دارد.

همانطور که گفته شد، میزان باور ذهنی نسبت به پاداش در افراد متفاوت است. به همین دلیل، از تابعی برای تبدیل پاداش به utility استفاده می‌شود. Utility function با استفاده از prospect theory بیان می‌کند که افراد پاداش منفی(جزا) را شدیدتر می‌بینند و همچنین، این تابع خطی نیست و در هر دو سوی رابطه نمایی دارد. تابع utility طبق تعریف زیر تعریف می‌شود.



$$v(x) = \begin{cases} x^{\alpha} & x \geq 0 \\ -\lambda(-x)^{\beta} & x < 0 \end{cases}$$

در این تابع میزان x همان میزان reward است که از اختلاف زمان رسیدن اتوبوس و مرز زمانی در حالت سوار شدن به اتوبوس و میزان زمان انتظار و مرز زمانی در حالت سوار شدن به تاکسی می‌آید.

میزان α و β و λ به ترتیب برابر با ۰.۸۸، ۰.۸۸ و ۲.۲۵ می‌باشد. این موارد طبق پژوهش Tversky و Kahneman در سال ۱۹۹۲ بر روی نمونه‌ای از دانشجویان به‌دست‌آمده‌است.

به تابع utility بالا باید نرسیدن به کلاس در شرایط انتخاب بد زمان انتظار را نیز اضافه کرد. همچنین، میزان ذخیره پول در صورت رفتن با اتوبوس را نیز باید به قسمت مثبت اضافه کرد. برای این کار ضریبی از پول ذخیرشده با میزان مثبت فعلی جمع می‌شود. دلیل این که این پول در قسمت مثبت قرار گرفته‌است این است که پول دادن برای رسیدن به کلاس ضروری می‌باشد و جنبه منفی برای ذهن ایجاد نمی‌کند اما رفتن با اتوبوس از نظر صرفه‌جویی حس مثبتی برای آدم ایجاد می‌کند.

ب) برای این کار از همان environment قبلی استفاده می‌شود.

کلاس reward برای ساده‌تر شدن پیاده‌سازی مشابه قبل پیاده‌سازی می‌شود؛ با این تفاوت که زمان رسیدن اتوبوس منفی نمی‌تواند باشد و باید آن‌قدر تلاش کند تا بتواند زمان رسیدن مثبت ایجاد کند.

عاملی که با استفاده از آن به یادگیری می‌پردازیم عامل e-greedy است که با احتمال 1-e عملی که در حال حاضر بیشترین میانگین پاداش را دارد، انتخاب می‌کند و با احتمال $\frac{\epsilon}{n}$ همه اعمال شانس پیدا می‌کنند. برای انتخاب عمل بهینه از میانگین اعمال تا این لحظه برای مقایسه استفاده می‌شود و بیشینه عمل greedy در نظر گرفته‌می‌شود.

پس از تعریف عامل در کلاس EGreedyAgent آن را به ازای ۱۰۰۰۰۰ آزمایش آموزش می‌دهیم تا بتواند زمان بهینه ایستادن در صف(عمل بهینه) را بیابد.

پیاده سازی این بخش در تابع e_greedy_run فایل Q2_uni_road.py آمده‌است. پس آموزش دادن زمان بهینه برابر ۱۵ به‌دست‌آمد. تصویر زیر بیانگر میزان متوسط پاداش هر عمل(زمان ایستادن) است. همانطور که می‌بیند ابتدا به دلیل صرفه جویی زمانی بالا پاداش زیاد است. هر چه جلوتر می‌رویم این پاداش کمتر می‌شود چراکه زمان ایستادن بیشتر می‌شود و فاصله تا مرز نرسیدن کمتر. از جایی به بعد، میزان صرفه جویی در پول مطرح می‌شود و پاداش را بالا می‌برد. از مرز به بعد که قطعا نمی‌رسیم، پاداش منفی دوباره متوسط پاداش اعمال را کاهش می‌دهد. بیشینه متوسط پاداش برای ۱۵ دقیقه صبر کردن یا همان مرز نرسیدن به‌دست می‌آید. البته شایان ذکر است اگر اهمیت پول را پایین‌تر بیاوریم، عمل صفر عمل بهینه خواهد بود چراکه با صرف منبع کم اهمیت(پول) می‌توانیم زمان رسیدن را بسیار زودتر کنیم.

```
Wait time: 0 => Expected Utility: 10.838278538293 61
Wait time: 1 => Expected Utility: 10.199823845187385
Wait time: 2 => Expected Utility: 9.555868176869588
Wait time: 3 => Expected Utility: 9.386092135920851
Wait time: 4 => Expected Utility: 8.914538825167636
Wait time: 5 => Expected Utility: 8.95259248555477
Wait time: 6 => Expected Utility: 9.028715319488283
Wait time: 7 => Expected Utility: 9.130979279175028
Wait time: 8 => Expected Utility: 9.507729537990503
Wait time: 9 => Expected Utility: 10.921787199616734
Wait time: 10 => Expected Utility: 10.691858055712773
Wait time: 11 => Expected Utility: 12.1274705657595 91
Wait time: 12 => Expected Utility: 12.733390773963352
Wait time: 13 => Expected Utility: 12.5582035339402 56
Wait time: 14 => Expected Utility: 12.514008018288273
Wait time: 15 => Expected Utility: 12.8841967732209 23
Wait time: 16 => Expected Utility: 11.963711162276512
Wait time: 17 => Expected Utility: 12.707345513218227
Best Wait Time: 15
```

پ) عامل epsilon-greedy در بخش قبل پیاده‌سازی شد. عامل UCB در کلاس UCBAgent پیاده سازی شد. در این سیاست ما ابتدا با فرض خوشبینانه حد بالای متوسط پاداش شروع می‌کنیم و رفته رفته آن را کنترل می‌کنیم. در این سیاست سعی بر آن است تا بر مبنای حدهای بالای متوسط پاداش عمل‌های مختلف بهترین عمل را انتخاب کنیم. پس از انتخاب عمل و گرفتن پاداش تخمین نقطه‌ای را به‌روز می‌کنیم و به این صورت حد بالا نیز به‌روز می‌شود. در انتهای باید سعی کنیم احتمال این که متوسط پاداش واقعی از این حد بالا بالاتر باشد را کم و کم تر کنیم که برای این مورد پارامتر δ در نظر گرفته‌می‌شود. حال اگر رابطه این پارامتر را معکوس زمان در نظر بگیریم، این پارامتر رفته‌رفته کم و کمتر می‌شود و تخمین خوبی از حد بالای متوسط پاداش به ما می‌دهد.

تفاوت این agent با agent قبلی در پیاده سازی چند مورد است. در ابتدا باید مقادیر تخمین متوسط پاداش را optimistic و زیاد تخمین بزنیم. تفاوت بعدی و اصلی این دو روش به نحوه انتخاب عمل بر می‌گردد. در این روش بر مبنای تخمین مقدار UCB هر عمل با فرمول cramer-chernoff می‌پردازیم و بیشینه این مقادیر به عنوان عمل بهینه در آن آزمایش انجام می‌شود.

```
UCB:
Wait time: 0 => Expected Utility: 10.838415044066393
Wait time: 1 => Expected Utility: 10.247536366965305
Wait time: 2 => Expected Utility: 9.657564072564995
Wait time: 3 => Expected Utility: 9.1391766765065
Wait time: 4 => Expected Utility: 8.663584829055027
Wait time: 5 => Expected Utility: 8.58901675654673
Wait time: 6 => Expected Utility: 8.388685133594612
Wait time: 7 => Expected Utility: 8.74296571940852
Wait time: 8 => Expected Utility: 8.645303316726316
Wait time: 9 => Expected Utility: 9.249773530405948
Wait time: 10 => Expected Utility: 9.5459435635515 34
Wait time: 11 => Expected Utility: 9.833462945089067
Wait time: 12 => Expected Utility: 10.143228919260913
Wait time: 13 => Expected Utility: 10.288326716546761
Wait time: 14 => Expected Utility: 10.318215028585492
Wait time: 15 => Expected Utility: 10.404444056584726
Wait time: 16 => Expected Utility: 10.468889928492272
Wait time: 17 => Expected Utility: 10.38888055680013
Best Wait Time: 0
```

در UCB نیز همان طرح دیده‌می‌شود. در اینجا اعداد بسیار به هم نزدیک بوده‌اند ولی همانطور که می‌بینید صفر به عنوان عمل بهینه انتخاب شده‌است.

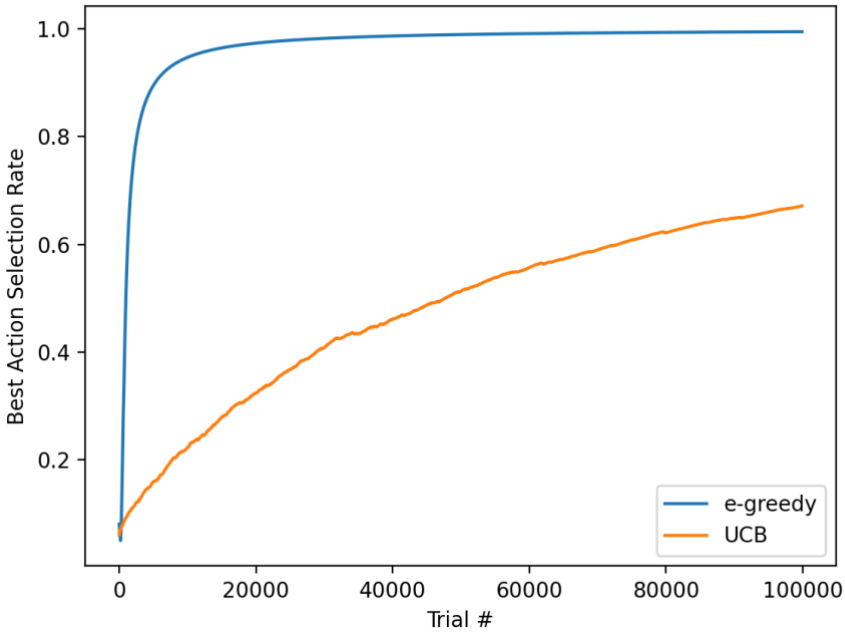
این دو عامل از نظر regret و درصد استفاده از عمل بهینه با یکدیگر مقایسه شده‌اند. عامل e-greedy میزان regret برابر با ۹۸۷.۱۷ برای خود ثبت کرد. در صورتی که عامل UCB میزان regret ش برابر ۲۱۳۷۰ شد. در این معیار عامل e-greedy بهتر عمل کرد.

در معیار استفاده از عمل بهینه در طول زمان میزان استفاده از عمل بهینه به‌دست‌آمده‌است و در نمودار زیر برای دو سیاست قابل مشاهده‌است.

همان طور که در نمودار دیده‌می‌شود، عامل e-greedy سرعت همگرایی بالاتری دارد و به همین دلیل، میزان استفاده از عملکرد بهینه آن بالاتر است. عامل UCB سرعت همگرایی پایین و عملکرد ضعیف‌تر در این معیار ثبت کرده‌است.

هر چند سیاست e-greedy از ابتدا خوش‌بینی کورکورانه دارد و نسبت به عمل بهینه عمل می‌کند و در UCB به دنبال خوشبینی در مقابل نایقینی هستیم اما در این مساله e-greedy عملکرد بهتری ثبت کرد.

البته برای بهبود این مقایسه می‌توان این مقایسه را چندین بار اجرا کرد و میانگین گرفت.



سوال 3 – تاخیر در شبکه

الف) در این مساله به دنبال کمینه کردن زمان ارسال هستیم. اعمال ما در این مساله مسیرهای مختلفی هستند که می‌توان رفت. در این مساله ۴۸ arm داریم که بیانگر مسیرهای مختلف ما هستند. یعنی یک مسئله 48-armed bandit داریم. برای این مساله از همان محیط MultiArmedBanditEnvironment استفاده می‌شود.

نکته کلیدی این مساله در تعریف rewardهاست. هدف مساله کمینه کردن تاخیر است و پاسخ محیط به ما همین تاخیر است. هر عمل متناظر با یک مسیر در این گراف است. پس، reward هر عمل میزان تاخیر آن مسیر است. ما در مسائل به دنبال بیشینه کردن reward هستیم. در نتیجه، برای خوش تعریف شدن این مساله reward را برابر با قرینه تاخیر تعریف می‌کنیم که هر چه بیشتر شود، بهتر شود. پیاده‌سازی reward جدید در کلاس NetReward قابل مشاهده‌است.

حال با این تفاسیر عامل مورد نظر با سیاست خود به دنبال بیشینه کردن متوسط پاداش(کمینه کردن تاخیر) می‌رود.

ب) برای پیاده‌سازی این عامل از همان عامل سوال قبل استفاده می‌کنیم با این تفاوت که میزان utility در این مساله همان reward است و تخمین خوش بینانه اولیه متفاوت است. با توجه به منفی بودن مقادیر reward، تخمین خوش‌بینانه اولیه صفر خواهد بود. پیاده‌سازی عامل جدید در کلاس NetEGreedyAgent قابل مشاهده‌است.

این عامل برای پیدا کردن بهترین مسیر در فایل Q3_network_delay پیاده شده‌است. بهترین مسیر، مسیر شماره ۱۲ است که از گره‌های ۰، ۲، ۵، ۱۰ و ۱۲ می‌گذرد. برای اینکه ببینیم چه تعداد آزمایش برای رسیدن به پاسخ نیاز است بررسی می‌کنیم که تغییرات عمل بهینه به چه صورت است. اگر برای ۱۰۰۰ آزمایش عمل بهینه تغییر نکرد، به نظرم کار عامل ما تمام شده‌است و عمل بهینه را پیدا کرده‌ایم. این اتفاق در ۱۴۶۴ امین آزمایش افتاده‌است. یعنی، در آزمایش ۴۶۴ ما بهترین مسیر را پیدا کرده‌بودیم.

تصویر زیر خروجی کد برای این قسمت است.

```
Final trial: 1464
Epsilon-Greedy:
Path: 0 => Expected Delay: -8.961800406401562
Path: 1 => Expected Delay: -25.75074118176814
Path: 2 => Expected Delay: -14.128639486064799
Path: 3 => Expected Delay: -22.324486941640323
Path: 4 => Expected Delay: -12.861054463574597
Path: 5 => Expected Delay: -10.543098070574962
Path: 6 => Expected Delay: -12.275839669446697
Path: 7 => Expected Delay: -8.755854628406533
Path: 8 => Expected Delay: -13.781874511685304
Path: 9 => Expected Delay: -26.896559955859914
Path: 10 => Expected Delay: -8.047021626280412
Path: 11 => Expected Delay: -23.043572795468844
Path: 12 => Expected Delay: -9.894972271186209
Path: 13 => Expected Delay: -10.867791391494858
Path: 14 => Expected Delay: -6.031381347852106
Path: 15 => Expected Delay: -11.85042413296611
Path: 16 => Expected Delay: -8.162188039938233
Path: 17 => Expected Delay: -15.758817696241858
Path: 18 => Expected Delay: -16.292288911860048
Path: 19 => Expected Delay: -6.792725023633679
Path: 20 => Expected Delay: -15.700442307814779
Path: 21 => Expected Delay: -17.12329386773143
Path: 22 => Expected Delay: -25.81010510613475
Path: 23 => Expected Delay: -16.951774748486404
Path: 24 => Expected Delay: -14.066624281886309
Path: 25 => Expected Delay: -13.545271615738361
Path: 26 => Expected Delay: -15.406339593032008
Path: 27 => Expected Delay: -17.692548142089443
Path: 28 => Expected Delay: -11.284614417915641
Path: 29 => Expected Delay: -20.612628907084407
Path: 30 => Expected Delay: -14.602976475699975
Path: 31 => Expected Delay: -15.656690553498075
Path: 32 => Expected Delay: -13.286755810516091
Path: 33 => Expected Delay: -11.515299602543788
Path: 34 => Expected Delay: -29.179683060280787
Path: 35 => Expected Delay: -22.926465548570437
Path: 36 => Expected Delay: -14.46916404269388
Path: 37 => Expected Delay: -23.096861555871634
Path: 38 => Expected Delay: -22.220460710482016
Path: 39 => Expected Delay: -12.215922457174115
Path: 40 => Expected Delay: -18.71220209562262
Path: 41 => Expected Delay: -25.704129743478408
Path: 42 => Expected Delay: -19.00900587367508
Path: 43 => Expected Delay: -24.446649328631658
Path: 44 => Expected Delay: -29.09956817618658
Path: 45 => Expected Delay: -34.05696770537626
Path: 46 => Expected Delay: -30.04192606354418
Path: 47 => Expected Delay: -20.44227688039913
Best Path Index: 14
Regret: 1675.851932375636
Best Path: 0 -> 2 -> 5 -> 10 -> 12
```

همانطور که می‌بینید، میانگین تاخیر بهترین مسیر ۶.۰۳ ثانیه است. این نتایج برای اپسیلون ۰.۵ است.

اگر اپسیلون را زیاد کنیم(مثلا ۰.۸)، تعداد trial بیشتری برای رسیدن به نتیجه زمان نیاز است چراکه کاهش epsilon و تغییر ندادن عمل بهینه بیشتر طول می‌کشد. در این حالت exploration بیشتری انجام می‌دهیم. در حالت حدی(اپسیلون ۱)، بیشترین میزان exploration را داریم و در نتیجه، میزان انتخاب عمل بهینه کمتر و میزان regret بیشتر می‌شود و تعداد آزمایش زیادی برای به جواب رسیدن نیاز است.(حدودا ۱۰۰۰ آزمایش: دو برابر حالت فعلی)

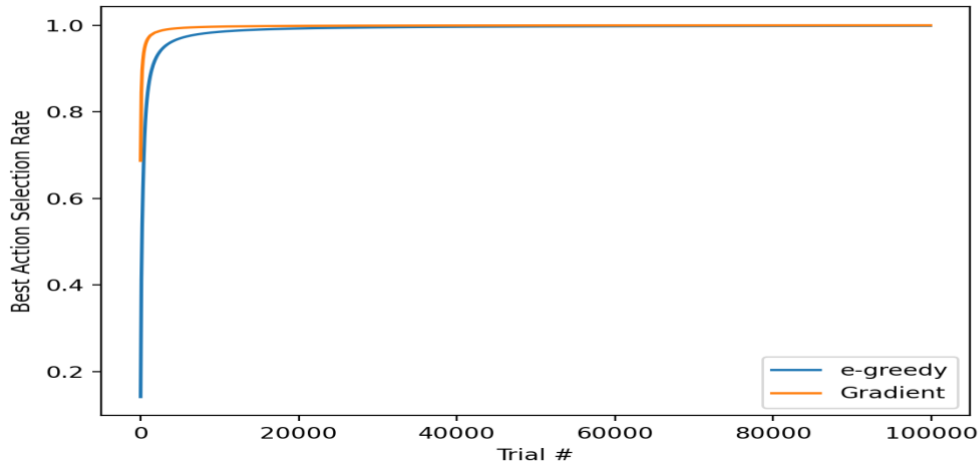
اگر اپسیلون را کم کنیم(مثلا ۰.۲)، خیلی سریعتر به جواب می‌رسیم چراکه بسیار زودتر به سراغ exploitation می‌رویم. در این جا خطر این وجود دارد که مسیر بهینه را پیدا نکنیم ولی به دلیل کوچک بودن فضا این اتفاق رخ نمی‌دهد. با حدودا ۱۵۰ آزمایش به پاسخ می‌رسیم. در حالت حدی(اپسیلون ۰)، تعداد آزمایش‌ها به کمترین حد خود می‌رسد. (حدود ۶۰ آزمایش) در این جا در بعضی حالت به پاسخ مطلوب نمی‌رسیم چراکه به طور کامل در حال exploitation هستیم.

پ) این عامل در کلاس NetGradientAgent پیاده‌سازی شده‌است. این سیاست مبتنی بر ترجیح متفاوت اعمال است. در ابتدا ترجیحات با یکدیگر برابر هستند و احتمالات انتخاب در سیاست برابر هستند. هدف از این سیاست بیشینه کردن مجموع پاداش کل یا همان متوسط پاداش کل است. پس، سعی می‌کنیم با روش متکی بر گرادیان افزایشی در همه ابعاد و به ازای همه اعمال متوسط پاداش کل را بیشینه کنیم. برای این منظور باید حرکت دادن میزان ترجیح اعمال مختلف بر مبنای پاداش گرفته‌شده و ضریب یادگیری در جهت بیشینه کردن پاداش کل گام برداریم. نتایج این عامل به صورت زیر به دست‌آمد. همان طور که مشاهده می‌کنید تعداد ۶۶۱ trial نیاز بود تا این عامل به مسیر بهینه برسد. مسیر بهینه این عامل نیز مشابه عامل قبل مسیر ۱۴ است که از گره‌های ۰، ۲، ۵، ۱۰، ۱۲ می‌گذرد.

```
Final trial: 1661
Gradient:
Best Path Index: 14
Best gradient Path: 0 -> 2 -> 5 -> 10 -> 12
```

این عامل تعداد آزمایش بیشتری نیاز دارد تا به نتیجه برسد. همچنین، زمان بیشتری نیاز دارد تا به نتیجه برسد. از طرفی نتیجه آن همواه ثابت نیست و تغییرات دارد که این می‌تواند به دلیل گیر افتادن در ماکسیمم‌های محلی باشد.

نمودار استفاده از عمل بهینه این دو عامل به صورت زیر است.



همانطور که مشاهده می‌شود، هر دو روش میزان استفاده از عمل بهینه مشابه و خوب دارند ولی روش گرادیان از ابتدا عمل بهینه را تقریبا شناسایی کرده‌است و به همین دلیل زودتر به حد بالای خود رسیده‌است البته با توجه به نقطه توقف آزمایش که در بالا بحث شد، احتمالا روش گرادیان جهش‌های نیز داشته‌است اما روش greedy از جایی به بعد تغییر نکرده‌است. البته حد توقف این دو سیاست چندان تفاوتی با یکدیگر ندارد. در مجموع به نظر در این بخش، gradient عملکرد بهتری داشته

است. البته نباید از نقش learning rate در روش گرادیان افزایشی گذشت چرا که می‌تواند سرعت رسیدن به جواب و گیر نکردن در sub optimumها را برای ما تعیین کند.

ت) به نظر من، روش گرادیان با تنظیم learning rate می‌تواند بهترین عملکرد را داشته باشد چرا که در این روش بر مبنای ترجیح اعمال پیش می‌رویم و این ترجیح وابسته به میزان پاداشی که دریافت می‌کنیم و میانگین پاداشی که تا اکنون دریافت کرده‌ایم و میزان فعلی ترجیح تغییر می‌کند. همچنین در این مساله هدف سریع رسیدن به جواب بهینه بود که همانطور که دیدیم روش گرادیان خیلی سریع به سمت هدف نهایی خود converge کرد. روش epsilon-greedy بسیار کورکورانه عمل می‌کند هرچند سریع به جواب می‌رسد. البته این عامل نیز در این مسئله خوب جواب داد اما اگر epsilon به خوبی تنظیم نشود می‌تواند مشکل ساز باشد و تعداد آزمایش زیادی برای رسیدن به جواب طول بکشد.

روش UCB زمان بر است و converge کردن آن بیشتر طول می‌کشد. همچنین reward در این مسئله با زمان تغییر نمی‌کند و به همین دلیل، UCB می‌تواند گزینه مناسبی نباشد. روش Thompson sampling نیز به دلیل استفاده از توزیع‌ها می‌تواند گزینه خوبی باشد اما ساخت دید آماری نسبت به این مسئله با این تعداد متغیر تصادفی کار ساده‌ای نیست و می‌تواند پیچیدگی اضافه کند.

در مجموع، به نظر روش gradient بهترین گزینه برای این مساله است. باید در نظر داشت که روش بهینه برای هر مساله متفاوت است و در بعضی مسائل ممکن است روش‌های مختلف عملکردی نزدیک به هم داشته باشند.