Dyna-Q and count-base exploration

Gabriele Prato, first semester M.Sc.[1]

[1] *McGill University*

Count-based exploration methods, such as UCB, don't usually deal well with large state spaces. In this paper I explore such methods mixed with planning algorithms in order to tackle large state spaces and outperform traditional planning algorithms. I evaluate these algorithms on various types of environments to discern their strengths and weaknesses. The results show that certain count-based exploration methods do provide a significant boost in performance to traditional planning methods.

I. INTRODUCTION

This paper presents the work that was done for my final COMP-767 project. I first started playing around with the Dyna-Q and Dyna-Q+ algorithms that were presented in the class text-book[1]. I implemented these algorithms to try and get the same results as in the book. After trying various configurations, I managed to get similar performance for Dyna-Q, but not for Dyna-Q+. For me, Dyna-Q+ always under-performed, while in the book, Dyna-Q+ performs better than Dyna-Q for the Blocking Maze and the Shortcut Maze environments. I then did the exercises concerning Dyna-Q and Dyna-Q+. The answer to these exercises can be found in section IV. The last exercise, which is the main part of this paper, was about implementing a variation to Dyna-Q+. For reference, we will call this version of the algorithm Greedy Dyna-Q+. After implementing this algorithm and testing it, I found that it performed better than Dyna-Q and Dyna-Q+. From there, I tried improving on the idea and managed to get even better results.

The following sections detail the various algorithms I've implemented, related work that I've come to known while reading on the subject of count-based exploration, exercises of the book and experiments that I have done while working on this final project and finally my closing thoughts. Acknowledgments and references can also be found at the end of this paper.

II. ALGORITHMS

A. Dyna-Q

Dyna-Q is Q-learning with a planning phase added to it. It does not do any count-based exploration.

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \epsilon$-greedy(S,Q)
(c) Execute action A; observe resultant reward, R, and state, S'
(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Q-learning

(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
(f) Repeat n times:
$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S,A)$
$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Planning

[1]

We show it here since it is the basis of all the following implementations and variations. Before testing any changes, we implemented and tested Dyna-Q in various environments. These results can be found in section V. We implement our own version of Dyna-Q, so as to make sure that the results we get reflect the changes made to our basis.

(My implementation of Dyna-Q can be found in [2].)

B. Dyna-Q+

Dyna-Q+ is a variant of Dyna-Q. It adds an exploration bonus to the reward in the planning phase.

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \epsilon$-greedy(S,Q)
(c) Execute action A; observe resultant reward, R, and state, S'
(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
(f) Repeat n times:
$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S,A)$
$R \leftarrow R + K\sqrt{T}$
$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Where T is the number of time steps since the action was last tried and K is some small constant.

(My implementation of Dyna-Q+ can be found in [3].)

For more details about Dyna-Q and Dyna-Q+, checkout page 172 to 177 of the text-book [1].

C. Greedy Dyna-Q+

In [1], a variant to Dyna-Q+ is described as follows: instead of having the reward bonus in the planning phase, let's try adding it to the action selection process. That is, instead of following an *e-greedy* policy, let's pick the action which maximizes $Q(S,a) + K\sqrt{T_{Sa}}$:

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow$ ~~ϵ-greedy(S,Q)~~ $\leftarrow \max_a Q(S,a) + K\sqrt{T_{Sa}}$
(c) Execute action A; observe resultant reward, R, and state, S'
(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
(f) Repeat n times:
$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S,A)$
~~$R \leftarrow R + K\sqrt{T}$~~
$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

This gives us an algorithm like Dyna-Q+, but that follows a greedy policy with a reward bonus.

One of the variations I've tried with this algorithm, is incrementing T_S only when in state S. That is, instead of always increasing the number of time steps since every action was last tried, I've tried increasing the number of time steps of only the actions that can be made in the current state. It turns out doing so provides a very small increase in performance. Results can be found in section V. This method seemed more logical, as an agent cannot perform

any action in the states other than the one he is in, simply because he is not in those states. The emphasis should be put on actions that were not selected, not actions that simply couldn't be made.

(My implementation of Greedy Dyna-Q+ can be found in [4].)

D. Greedy Dyna-Q+ log variant

Another variant I tried, was applying the natural logarithm to T:

$$R \leftarrow R + K\sqrt{\log T}$$

The idea is to have a bonus that keeps increasing such that all actions will eventually be selected, but not to impact performance as much as a linearly increasing T would. Such a change increased performance in the experiments I've made (see section V).

(My implementation of Greedy Dyna-Q+ can be found in [5].)

E. Greedy Dyna-Q+ UCB variant

Building on the log improvement, another variant I tried was using UCB as the reward bonus:

$$R \leftarrow R + K\sqrt{\frac{\log t}{N_t(a)}}$$

Where $\log t$ is the number of time steps since the beginning of the episode and $N_t(a)$ is the number of times that the action a was selected. Note that t is not the same thing as the T we've been using in previous algorithms.

UCB doesn't usually deal well with large state spaces, but the results I got with this variant are the best so far. UCB mixed with planning seems to work well in this case.

Since this variant is the one that performed the best on the Blocking Maze environment, I've also tried it on other types of environment to see how it would performed. All the results can be found in section V.

(My implementation of Greedy Dyna-Q+ can be found in [6].)

III. RELATED WORK

Like was mentioned previously, count-based exploration has a hard time dealing with high-dimensional state spaces. The work [7] made by Tang, Houthooft, Foote, Stooke, Chen, Duan, Schulman, De Turck and Abbeel tackles such a problem and shows how count-based exploration can be applied in Deep Reinforcement Learning to achieve near state-of-the-art on Atari 2600 games.

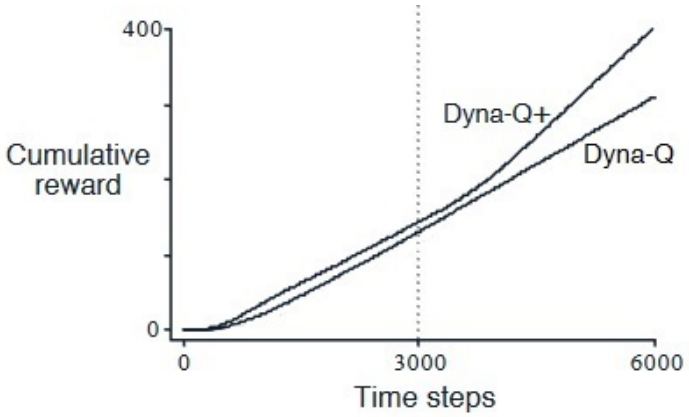
IV. EXERCISES

In the process of working on this project, I've worked on the following exercise to try an better understand these algorithms. I've included these to share my thoughts:

A. Exercice 8.2

Here's the question found in the book [1]:
Why did the Dyna agent with exploration bonus, Dyna-Q+, perform better in the first phase as well as in the second phase of the blocking and shortcut experiments?

In the book, the following graph is provided:



The results we get are different from this (see section V). In our tests, Dyna-Q performed better than Dyna-Q+. For the sake of the exercise, we will answer the question using the graph provided by the book [1]:

Dyna-Q+ performs better in both phases simply because it explores more. In the first phase, the exploration bonus allows it to find the goal more quickly. Dyna-Q finds it too, just with a bit more time and then both algorithms seem to have the same slope (Dyna-Q's slope being slightly better), so the difference in performance is small. For the second phase though, again with the exploration bonus allowing Dyna-Q+ to find the quicker route and then gaining a huge boost in performance.

B. Exercice 8.3

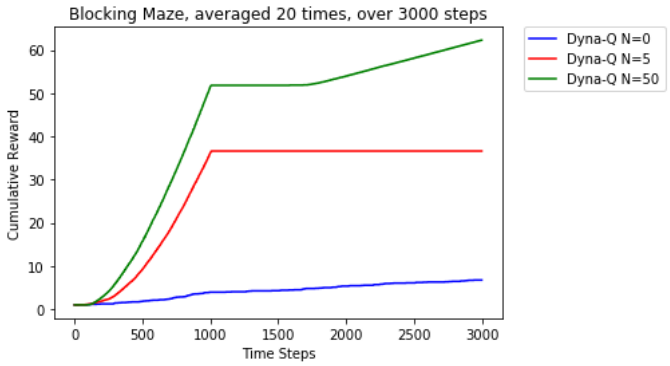
Question:
Careful inspection of Figure 8.6 reveals that the difference between Dyna-Q+ and Dyna-Q narrowed slightly over the first part of the experiment. What is the reason for this?

(Again here, we will answer the question using the provided graph.)

The way the exploration bonus works for Dyna-Q+ is that previous state-action pairs receive a reward bonus of $K\sqrt{T}$ where T is the number of time steps since that last state-action pair was last tried. So even if the optimal path was found, the Dyna-Q+ agent keeps on exploring previous state-action pairs in case things have changed and thus slows performance down compared to Dyna-Q only having the *e-greedy* policy affecting its performance. Once the optimal route has been found, Dyna-Q wastes less time steps exploring than Dyna-Q+ and thus Dyna-Q's slope is slightly more steep than Dyna-Q+'s.

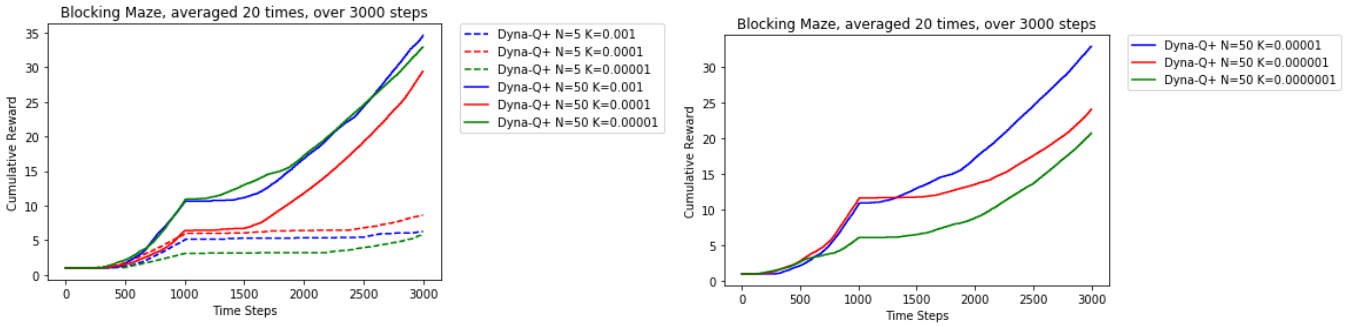
V. EXPERIMENTS

A. Dyna-Q

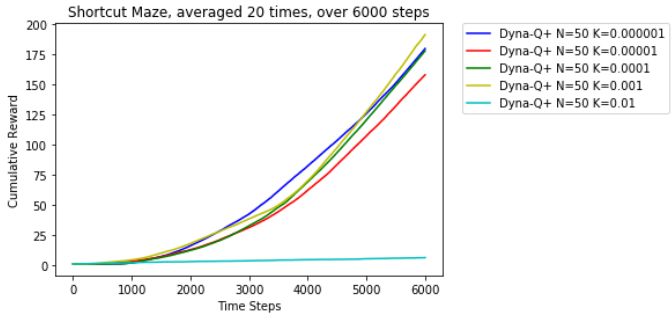


This test just shows that more planning is good, for the Blocking Maze with Dyna-Q.

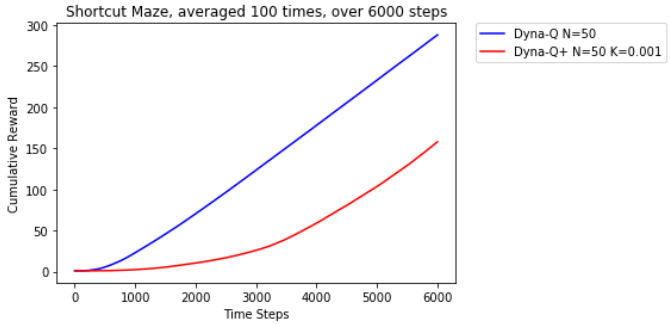
B. Dyna-Q+



With less planning $N = 5$, a middle value of $K = 0.0001$ performed best, but for $N = 50$, the middle K value of 0.0001 under-performed compared to $K = 0.001$ and $K = 0.00001$. Lowering the K value even more didn't help.

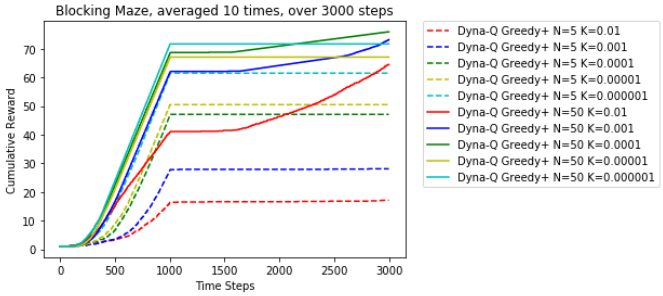


On the Shortcut Maze environment, all K values lower than 0.001 performed similarly.

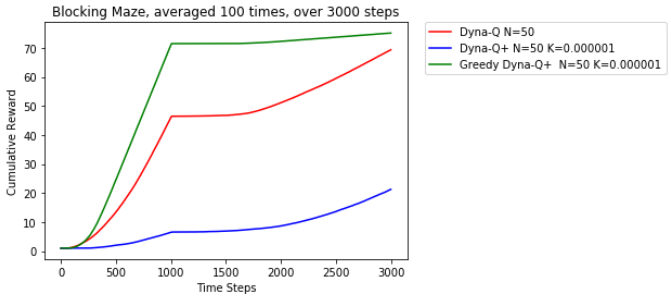


On the Shortcut Maze environment, Dyna-Q outperforms Dyna-Q+.

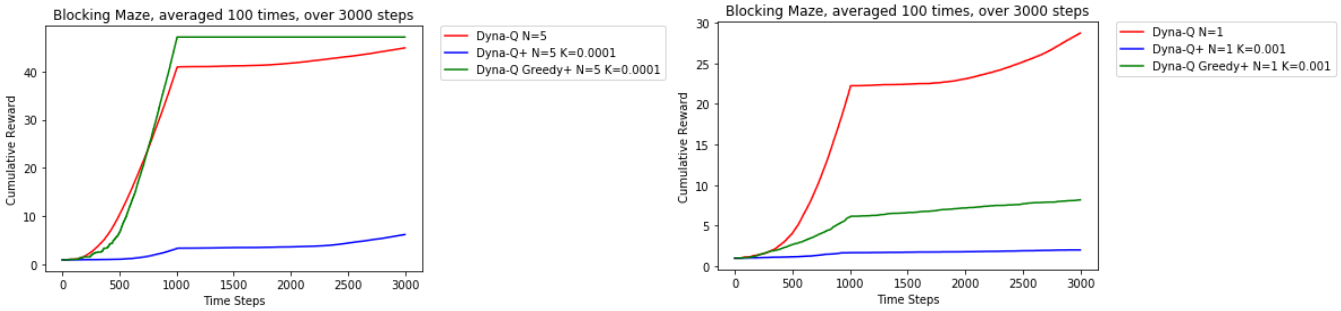
C. Greedy Dyna-Q+



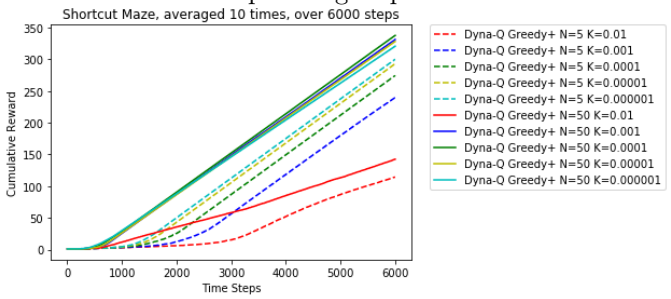
When there is less planning, $N = 5$, lower values of K seem to be better. When there is more planning, $N = 50$, K values lower than 0.001 seem to perform similarly.



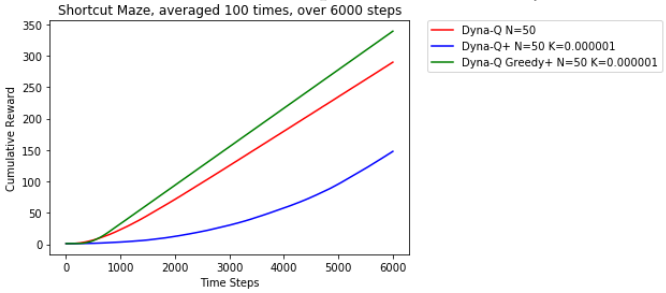
Greedy Dyna-Q+ performs better than Dyna-Q and Dyna-Q+ on the Blocking Maze environment, with planning $N = 50$.



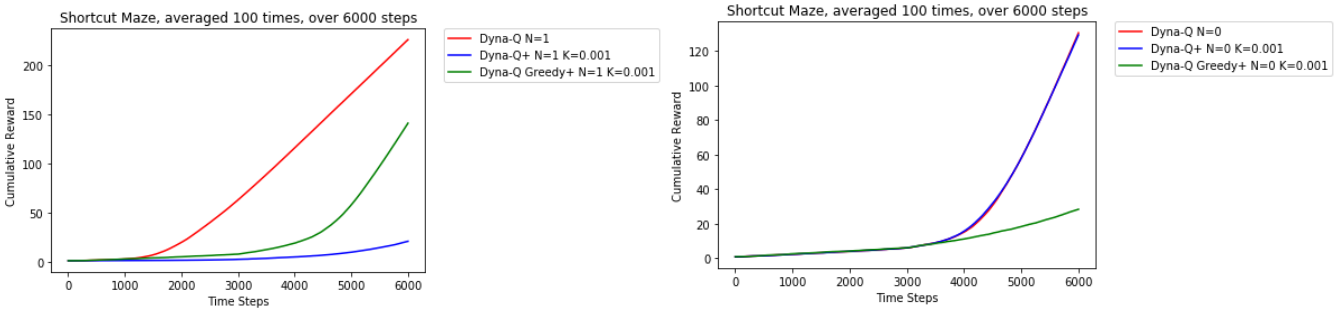
Comparing all three algorithms so far, with few planning, $N = 5$, Dyna-Q and Greedy Dyna-Q+ seem to perform similarly. When lowering even more the planning, $N = 1$, we see that Greedy Dyna-Q+ performs less well than when it had more planning steps. It seems like Greedy Dyna-Q+ performs better when it can perform more planning steps.



In the Shortcut Maze environment, with $N = 5$, lower K values performed better, while with $N = 50$, all K values lower than 0.001 performed similarly.

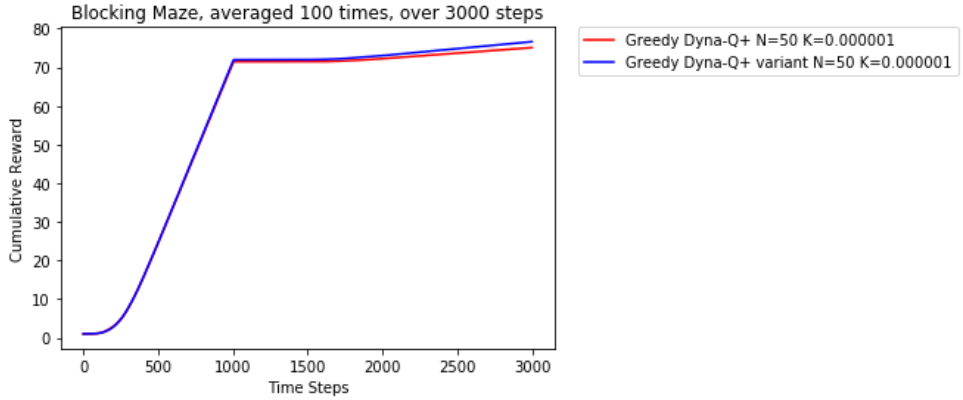


Comparing all algorithms on the Shortcut Maze environment, again Greedy Dyna-Q+ performs the best.



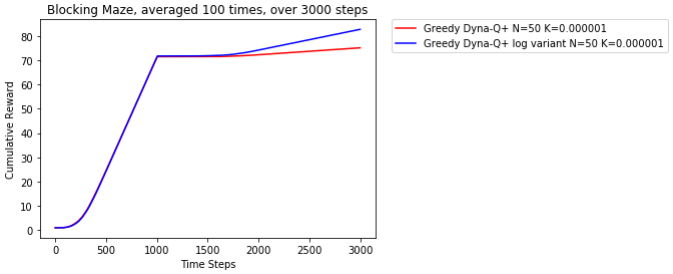
When lowering planning, $N = 5$, again Greedy Dyna-Q+ performs less well and gets beaten by Dyna-Q. And when we completely remove planning, Dyna-Q and Dyna-Q+ perform the same, which is normal since they are both just Q-learning in that case, while Greedy Dyna-Q+ doesn't do well.

Like with the Blocking Maze case, it seems like Greedy Dyna-Q+ outperforms Dyna-Q and Dyna-Q+ only when there is a lot of planning, $N = 50$. It would then seem like count-base exploration alone does not perform well, but when combined with planning, performs better than just planning alone.



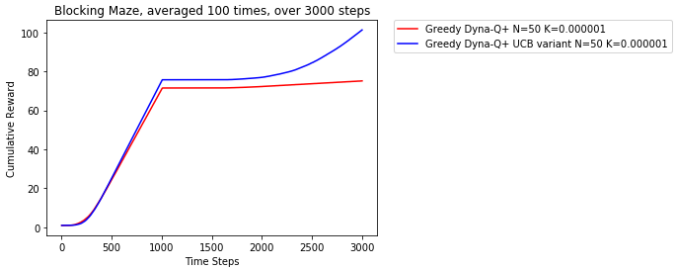
Increasing the number of time steps since an action was last tried, but only when in the state in which the action can be performed seemed to increase only slightly the performance. Though, results are very similar, so it is preferable to consider both methods as performing equally.

D. Greedy Dyna-Q+ log variant

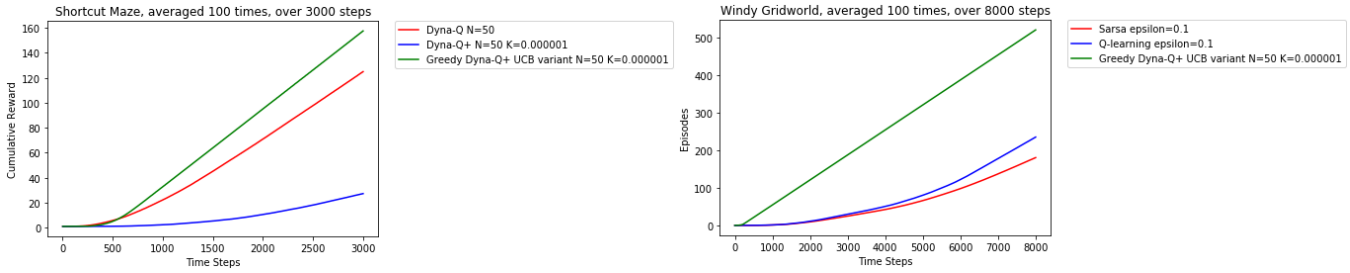


The log term $K\sqrt{\log T}$ seems to help performance.

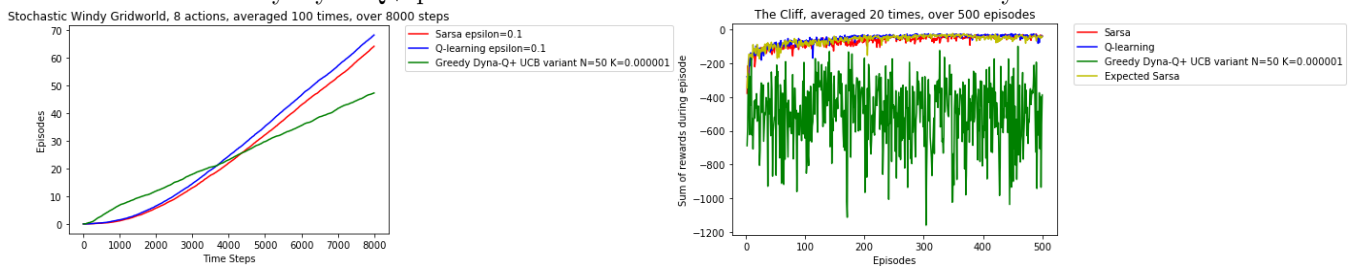
E. Greedy Dyna-Q+ UCB variant



Greedy Dyna-Q+ with UCB is the variant that performs the best in all my tests.



Greedy Dyna-Q+ performs the best in the Shortcut Maze and Windy Gridworld.



In the Stochastic Windy Gridworld environment, Greedy Dyna-Q+ doesn't perform very well. That seems to be caused by planning not taking into account stochasticity. We could modify the algorithm so that the planning phase does take into account stochasticity and results should then improve, but I haven't tried it.

In the Cliff environment, Greedy Dyna-Q+ under-performed. The reason for that would be that the algorithm is based on Q-learning, which follows the path alongside the cliff. Add to that an exploration bonus and you get an agent that will fall very frequently. One modification that could be done is to base our algorithm on Sarsa instead of Q-learning, which tends to take the safer path.

VI. CONCLUSION

We've tested multiple planning algorithms with count-based exploration variations and compared them on various environments. Count-based exploration alone didn't seem to perform well, but when combined with planning, it outperformed regular planning algorithms. Out of all the variants I've tried, the algorithm that performed best was the Greedy Dyna-Q+ with the UCB variant, which combines Dyna-Q+ and UCB.

It would be interesting to try more variants of Greedy Dyna-Q+ and try to improve it further more. It would also be interesting to try mixing other planning algorithms with count-based exploration.

A. Acknowledgments

I would like to thank Doina Precup and Pierre-Luc Bacon for their guidance throughout this project, for taking the time to answer my many questions and for their great work as teachers.

B. References

[1] Richard S. Sutton and Andrew G. Barto, "Reinforcement learning: An introduction", Second Edition, MIT Press, in preparation (All the content referenced in this paper can be found on pages 37 and 172 to 177.)
[2] https://github.com/rllabmcgill/rlcourse-final-project-pratogab/blob/master/dynaq.py
[3] https://github.com/rllabmcgill/rlcourse-final-project-pratogab/blob/master/dynaqplus.py
[4] https://github.com/rllabmcgill/rlcourse-final-project-pratogab/blob/master/greedydynaq.py
[5] https://github.com/rllabmcgill/rlcourse-final-project-pratogab/blob/master/greedydynaqlog.py
[6] https://github.com/rllabmcgill/rlcourse-final-project-pratogab/blob/master/greedydynaqucb.py
[7] https://arxiv.org/abs/1611.04717