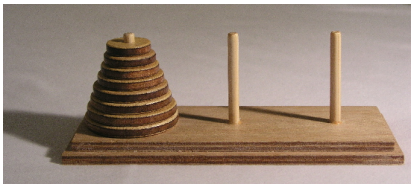
Tower of Hanoi: A Reinforcement Learning Approach

Yann Adjanor, Richard Mann, Msc Data Science, City University of London.

Abstract— We apply a Q-learning agent to the Tower of Hanoi problem. **Index Terms**—Reinforcement Learning, Agent, Recursion.



1 Introduction

The Tower of Hanoi, also known as Tower of Brahma, is a mathematical puzzle invented by the French mathematician Édouard Lucas in 1883 [1]. In its basic version it comprises of 3 rods or poles and a number N of holed disks of different sizes that can fit on the poles. All the disks are initially placed on the first pole in decreasing order of size, forming a tower. The aim of the game is to move all the disks onto the third pole one at a time with the condition that, at any time, no disk can be stacked above a disk that is smaller and only the top disk on one of the pole can be moved. The game has been extensively studied and has found practical applications in psychological testing, technical codes, and modelling of certain physical phenomena. It has also attracted the interest of computer scientists for decades as a poster child for recursive algorithms. The Tower of Hanoi appeals as a reinforcement learning challenge because of the way it can be scaled. Increasing the number of disks increases the complexity of the puzzle as a trial-and-error challenge indefinitely, with 3^N possible states. Yet an optimum solution is easy to define due to the recursive nature of the problem for any number of disks, therefore reinforcement learning performance can be measured objectively against a known optimum.

2 The Classic Problem

2.1 Domain and Task

The objective is to train an agent to solve the Tower of Hanoi (TH). Solving the game is defined as follows:

> "Given a starting position, usually defined by all disks stacked on the first pole (start state), determine the sequence of valid moves that lead to the end position (goal), defined as all disks stacked on the third pole (goal state)."

We also want to see if the agent can find the optimal solution, ie the solution incorporating the least amount of moves.
Let's first define the parameters of the TH problem:

1. There are {N} disks numbered from 1 (smallest) to N (largest)
2. A state {s} is any valid position of all the disks across the 3 poles
3. An action {a} is a move from one state to another
4. Given that there is only one way to sort a list of distinct integers in increasing order, a valid position is any set of 3 lists of distinct integers (p_1, p_2, p_3), or 3-tuples of lists , that are a partition of {1,…,N} In other words $\bigcup_{i=1}^{3} p_i = \{1, ..., N\}$
5. In the following we look at the general N case for theory, N=3 for illustration and N=4 or higher for experimental results.

2.2 State Transition Function

2.2.1 Number of possible states: S(N)

We denote S(N) as the number of possible states: it can be calculated by simple recurrence as follows.

1. For N=1 we have 3 possible states, corresponding to 3 different positions for the sole disk, so S(1)=3
2. To go from any state with (N-1) disks to a state with N disks, means adding disk N which is therefore the largest. It can only be situated at the bottom of any pole (see Fig.1). Therefore, for any possible state with N-1 disks, there are 3 corresponding state with N disks, therefores: $S(N) = 3 \cdot S(N-1)$
3. By recurrence we have: $S(N) = 3^N$

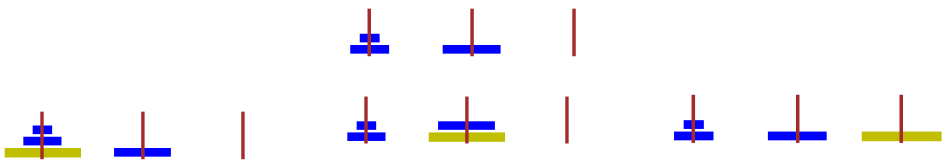


Figure 1: A state with (N-1) disks (top) and the 3 equivalent states with N disks (bottom)

2.2.2 Minimum number of moves from start state to goal state: T(N)

A similar proof by recurrence can be applied to determine the minimum path length from the start state to the goal state

1. For N=1, the problem is solved in one move: moving one disk from pole 1 to pole 3. So, $T(1) = 1$

2. Assuming that we can solve TH(N-1) then TH(N) is the problem where we have the largest disc N on pole 1 and the other (N-1) disks stacked on top of it (see Figure 1). The solution therefore takes 3 operations: move the (N-1) disks to pole 2 in T(N-1) moves, then move disk N to pole 3 in 1 move and finally move the (N-1) disks from pole 2 to pole 3 in T(N-1) moves.
3. So we have $T(N) = 2 \cdot T(N-1) + 1$ hence given that $T(1) = 1$, we have $T(N) = 2^{N-1} + 1$

2.2.3 Compact State Transition Function

A transition in the game is defined as a move from a valid state to another valid state. We saw that the N-case TH problem has $S(N) = 3^N$ valid states. So theoretically there could be 9^N possible transitions. But in reality, the rules of the TH game preclude moving any disk other than one that is at the top of one of the 3 stacks.

As noted before, we numbered the disks from 1 to N (smallest to largest) and any valid state is uniquely defined by a 3-tuple of integer lists (p_1, p_2, p_3) where p_i is a list of integers representing the disks numbers on pole i, given that hey can only be arranged in decreasing order.

We therefore introduce the notion of a macro-state which is a 3-tuple of integers (t_1, t_2, t_3) where t_i is the number of the top disk on pole i or zero if the pole is empty. In effect we have $t_i = min(p_i)$ *or* $(t_i = 0$ *if* $p_i = [\,])$.
A move from pole i to pole j is possible if and only if $(t_i \neq 0)$ *and* $(t_i < t_j$ *or* $t_j = 0)$.
We can compact the representation further by transforming (t_1, t_2, t_3) into a three character string $c_1c_2c_3$ representing the ordering of the top discs with the special condition that empty poles and the pole with the largest disc are represented by 0 (cannot be moved)

1. $c_i = 0$ *if* $(t_i = 0)$ *or* $(i = argmax_j(t_j)$ *and* $t_j \cdot t_k \neq 0$ *for* $i \neq j$ *and* $i \neq k)$
2. $c_i = 1$ *if* $i = argmin_j(t_j)$
3. $c_i = 2$ *otherwise*

Note that (t_1, t_2, t_3) and $c_1c_2c_3$ are effectively feature extractions and do not map bijectively to actual states.

A generic transition table can then be represented as in Table 1 where action i->j means a move from pole i to pole j.

The state transition is exactly defined above and can be generated in a programmatic way for any N. Its formulaic representation is $s' = s_t = T(s_t, a_t)$ where s_t is the the current state, a_t is the action chosen and s' is the state resulting from that action.

From any state s_t, the state transition function T, defines a set $A(s_t)$ of possible actions available. This will be used in the learning algorithm. In the particular case of the TH problem we can represent this function as a matrix where rows and columns are states since the transition function effectively maps states into other states.

	State	Action1	Action2	Action3
1	120	1->2	1->3	2->3
2	102	1->2	1->3	3->2
3	210	1->3	2->1	2->3
4	201	1->2	3->1	3->2
5	012	2->1	2->3	3->1
6	021	2->3	3->1	3->2
7	100	1->2	1->3	
8	010	2->1	2->3	
9	001	goal		

Table 1: Compact state transition function (T)

2.3 Reward Function

In order for our agent to learn to solve the task, in addition to the transition function that determines which actions are available to choose from, the agent will collect rewards as it moves through the state-action space. The reward function R determines at each state what reward is associated with which action: $R(s_t, a_t) = r_{t+1}$. Note that the reward is allocated at time t+1.
We choose a sparse reward function where the agent receives a reward (set at 100) only if it chooses an action that brings it to the goal state. The reward function is represented as matrix, similarly to the state transition function. Figure 2 displays the R matrix for N=3 showing that rewards are only allocated to actions that directly lead to the goal state (state 26). Cells marked '-' are invalid transitions.

Note on state representation:
The 3-tuple integer list representation for all valid states has the benefit of clarity but can be cumbersome to manipulate programmatically. We have therefore chosen to associate a unique number to each state in the N case TH problem using prime number factorisation. The process is as follows:

1. Each pole is represented by its number: 1, 2 or 3, and (p_1, p_2, p_3) is the list *of the list* of the disks on each pole
2. Each disk i is associated to the number i and we also associate it to $Z(i)$ which is the i^{th} prime number greater than 1. We also define $p(i)$ as the number of the pole where disk i is placed ie: $p(i) = p_j$ *if and only if* $i \in p_j$
3. Each state $s = (p_1, p_2, p_3)$ can then be uniquely represented by the code $C(s) = \prod_{i=1}^{N} Z(i)^{p(i)}$

The advantage of this representation is that each configuration for any set of N disks is uniquely represented. Additionally, moving a disk is a simple arithmetic operation on the state code. Move disk i one pole to the right: multiply by $Z(i)$ -> $C(s_{t+1}) = C(s_t) \cdot Z(i)$ and move disk i one pole to the left: divide by $Z(i)$ -> $C(s_{t+1}) = C(s_t) \div Z(i)$. The state list for the TH(N) problem is a 3^N size array of numbers of this type. In the rest of the paper we will refer either to state numbers (the code C(s)) or a state index (the index position in the state list array).

2.4 Graph representation

We can represent the agent search through the state/action space for N=3 as a navigation through the graph in figure 3. This graph exhibits self-similarity as it is composed of subgraphs that have all the same shape with states arranged as triangles within larger triangles. We have seen before that the TH problem has strong recursive properties and the state/action graph reflects this recursivity through its self-similar structure.

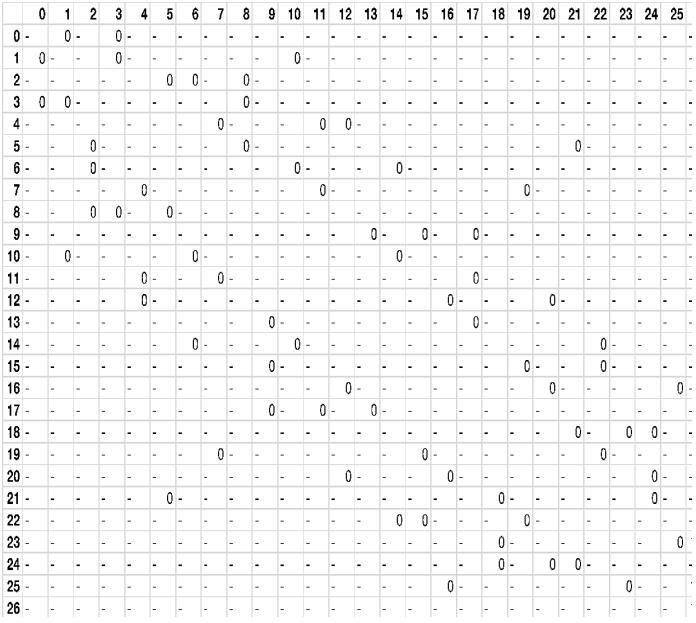


Figure 2: Reward Matrix for the 3-disks Tower of Hanoi

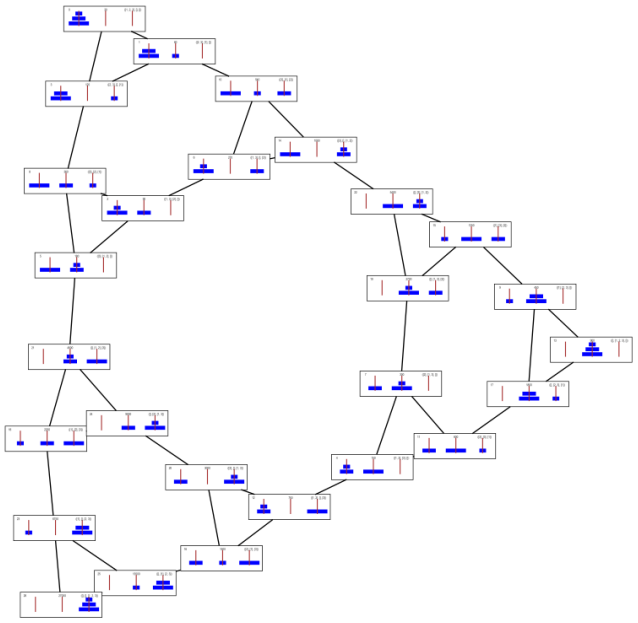


Figure 3: Graphical representation of the state-action space for 3-Disks.

2.5 Policy

We choose the Q-Learning algorithm for agent learning, with the learning process summarised as follows:

Q learning algorithm	
1. Algorithm parameters:	Alpha (learning rate) = α . Epsilon (ε/greedy exploration) = ε Gamma (discount rate) = γ . Epsilon decay factor (if appropriate) = δ
2. Initialisation:	Initialise null **Q(s,a)** matrix Initialise current state (**s**) to 0 (start of the puzzle)
3. Loop for each episode:	Choose action a from s using epsilon-greedy policy: -randomly among any possible actions, or -among the best actions only : a = argmax_a**Q(s,a')** Take action a, observe reward (**r**) and state (**s'**) Update Q: $\mathbf{Q(s,a)} \leftarrow \mathbf{Q(s,a)} + \alpha[\mathbf{r} + \gamma . \max_a \mathbf{Q(s',a')} - \mathbf{Q(s,a)}]$ Reduce ε by factor δ , if applicable Until **s** = terminal state (end of puzzle)
4. Repeat	Repeat step 3 until stopping criteria met (convergence or maximum number of timesteps/episodes)

2.6 Parameters for Q-Learning

In this report, we consider the impact of each of the Q-learning parameters in turn. As explained in the analysis, we focus mostly on varying the learning rate (α) and epsilon greedy policy (ε). We find that varying the discount rate (γ) and decay rate (δ) less insightful although these are discussed briefly towards the end of section 2.

2.7 Updating the Q-matrix step by step

We now illustrate the first few steps of Q-Learning: how the agent learns by updating the Q matrix in the course of a serie of episodes.
To start with, the Q matrix is initialized as zero.

Each episode begins with the agent placed in a random state and ends when the agent reaches the end goal. Moves from one state to the other during an episode are made following an "epsilon-greedy" policy: random (governed by the epsilon parameter) or greedy (following its behavioural policy directed by the Q-values). In the following we assume that alpha and gamma have been set both to 0.8:

Episode 1:

The agent has been (randomly) placed in state s = 25. Given that it is not the goal state, the agent finds the available actions A={ai} from this state by checking the reward matrix (which is also a representation of the state transition function). The possible actions are a move to state 16, state 23 or state 26 (which is the goal state but *the agent does not know this yet*): A={16,23,26} (actions are states in our case)

To select an action, the agent either chooses randomly (by drawing a sample from a uniformly distributed random variable and comparing it to the parameter epsilon) or chooses based on which action maximizes Q(25,a). In this episode the Q matrix is still initialised at zero so a greedy move corresponds to a random one.

For illustration we assume that the agent chooses action (state) 26. By doing so the agent receives the reward R(25,26)=100.

Since state 26 is the goal state there are no other available actions from this state and the Q matrix update is simplified:

Q(25,26)= $Q_{old}(25,26) + \alpha(r + \gamma.0 - Q_{old}(25,26))$ =0+0.8*100 = **80**

Episode 2:

Step1:

The agent is randomly placed in state s=16. From there states 12, 20 and 25 are accessible: A={12,20,25}.

Assuming the agent choses greedily then it chooses a= $argmax_{a\in A}(Q(16,a))$. Given that all those Q values are zero, this is again a random choice. Let's assume the agent chooses state s'=25.

The agent then receives the reward r=R(16,25)=0 and then considers all the possible moves/actions from state s'.

We have A'={16,23,26} and the agent computes $max_{a\in A'}Q(s',a)$

This is equal to max(Q(25,16),Q(25,23),Q(25,26))=Q(25,26)=80 as per episode 1.

The Q matrix can then be updated with **Q(16,25)** = $Q_{old}(16,25) + \alpha(r + \gamma.max_{a'\in A'}Q(s',a') - Q_{old}16,25))$ =0+0.8*(0+0.8*80-0)=**51.2**

Step2:

The agent is now back in state s=25

Assuming it chooses randomly and ends again in the goal state (26), then the update of the Q matrix becomes:

Q(25,26)= $Q_{old}(25,26) + \alpha(r + \gamma.0 - Q_{old}(25,26))$ =80+0.8*(100+0-80) = **96**

As we can see the Q values are updated each time the agent goes through the same state-action combination.

This terminates the episode.

The process is iterated until the Q matrix is entirely populated and remains close to stationary.

These steps are illustrated in figure 4.

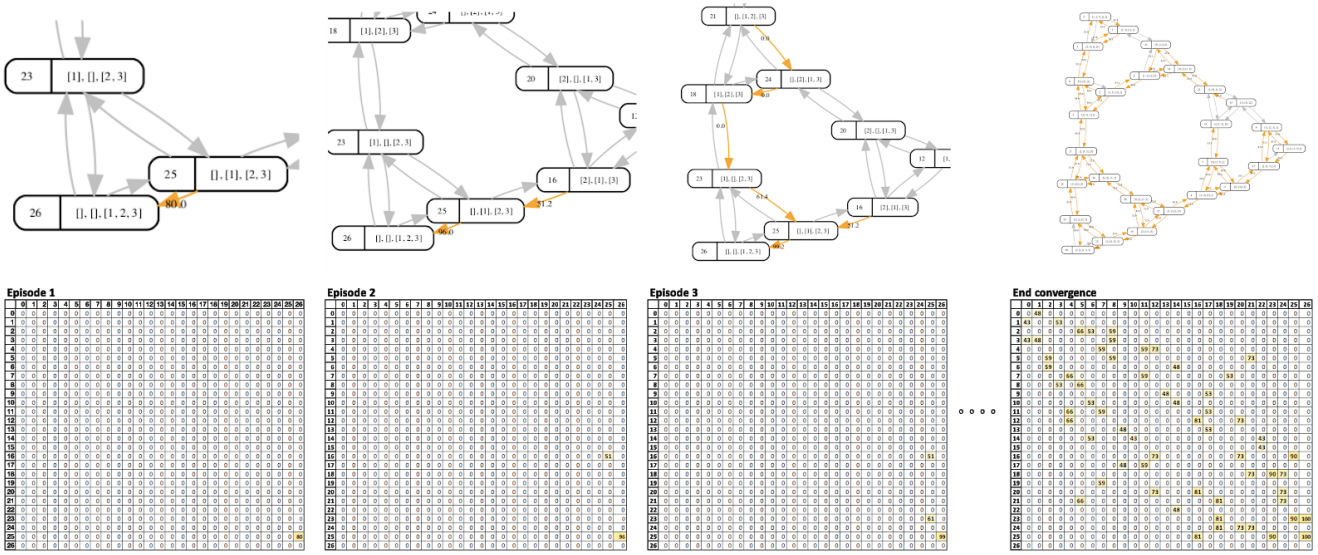


Figure 4: Q matrix update example

2.8 Results of Q-Learning

2.8.1 Solving the 3-disk and 4-disk puzzle

The optimum solution of the 3-disk problem is solved quickly provided ε is not too small. If we apply Q learning over a fixed number of timesteps and repeat for 100 iterations, we see (figure 5a(i)), that the puzzle is solved in a majority of attempts, but the optimum route is found more frequently with higher α and ε. After learning for 3,000 timesteps (figure 5a(ii)), a solution is always found but an agent with higher ε and α finds the optimum solution most frequently.

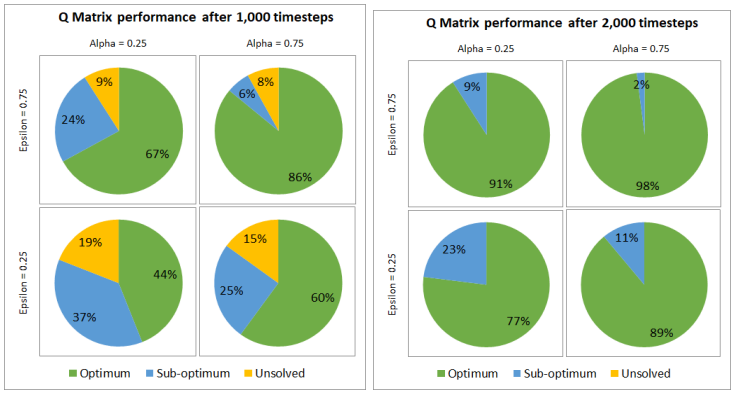


Figure 5a: Greedy routes taken after (i) 1,000 and (ii) 3,000 timesteps of 3 Disk puzzle.

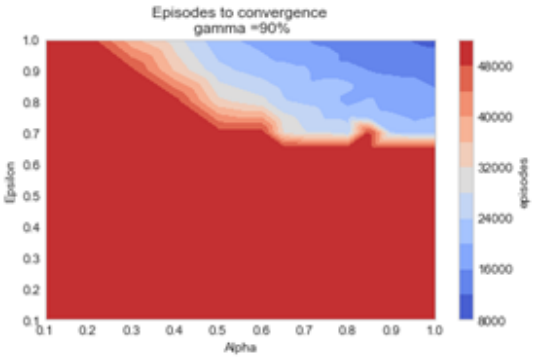


Figure 5b: Number of episodes to reach convergence with 4 disks.

The 4 disk problem requires significantly more learning to solve. After 2,000 timesteps, the Q learning matrix fails to solve the puzzle at all. However, similar relative performance between for four α/ε combinations above is seen at 10,000 and 20,000 steps respectively:
In summary, higher ε and higher α are both beneficial when learning the Q matrix from scratch.

Unless otherwise stated, we now focus on the 4 disk puzzle as we consider the level of difficulty results in more interesting analysis without being too difficult to solve by Q learning.

2.8.2 Convergence measures

Q matrix has been shown to converge to optimum action-values ($\mathbf{Q} \rightarrow \mathbf{Q}^c$) with probability = 1 so long as all actions are repeatedly sampled in all states and the action-values are represented discretely [2][Hinz, 2013][3][Watkins, 1992].

In the Tower of Hanoi puzzle, as visualised in section 2.5, there is a 'long way round' the puzzle as well as the optimum solution. Whilst convergence in the direction of the optimum start-to-end puzzle solution is observed in the timeframes discussed above, most learning occurs in the frequently visited parts of the Q matrix. The more the Q matrix improves, the less this long route is visited and therefore convergence to optimality for the entire Q matrix is slow unless we allow the agent to start randomly from different states. In this special case, we can measure convergence to Q^c with the root-mean-square-error ('RMSE') between Q and Q^c. Figure 5b shows the number of episodes taken to reach RMSE < 0.001, illustrating that convergence is fastest with high α and ε.

Returning to the more natural problem of learning the puzzle always starting at state 0, we need to look at a convergence metric that focuses on performance compared to the optimum route. One option already presented (as shown in figure 5) is to assess how good the Q matrix is at finding a solution, and indeed the optimum solution, after learning. But this is an 'off-line' measure of convergence in that learning is stopped and a greedy policy adopted to measure performance.

To examine performance throughout learning, we can measure the cumulative reward obtained compared to the number of timesteps taken and also the average rate of reward as timesteps progress.

Cumulative reward R_c: $R_c = \sum_{1}^{T} R(S(t), a(t))$ where T = total number of timesteps

and *Rc* should be averaged over n learning iterations to represent typical performance

Rate (velocity) of reward R_v: $R_v = dR_c/dt$ where the rate of change is averaged over a timestep window (say 500) and n learning iterations.

In this approach, visualising convergence becomes easier as R_c tends towards a constant gradient and R_v tends towards a maximum.
For example, figure 6 shows Rc and Rv for the first 3,000 of Q learning.

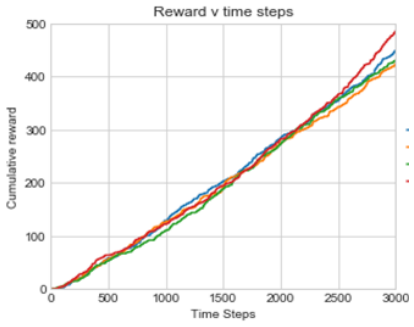
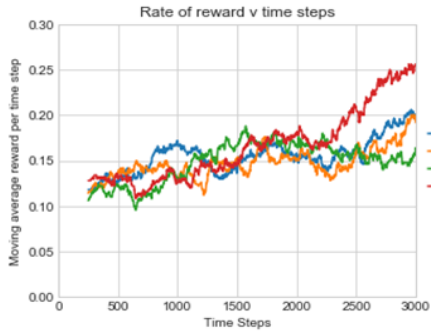
R_c	R_v	
		Initially, moves are random irrespective of the e-greedy policy as the Q matrix starts empty. As the challenge is entirely predictable (no randomness) and static (unchanging), changing learning rate has no noticeable impact. Note that a low α is not detrimental as all new learning is in the right direction. Towards 3000 steps, the Q matrix is still being learned and there is a benefit in continuing to explore rather than exploit, so high ε is beneficial and ***red/orange*** are moving ahead.

Figure 6: Rc (left) and Rv (right) for Q learning with 4 disks, first 3,000 timesteps. All examples are averaged over 100 iterations with $\gamma = 0.75$, $\delta = 1$

Generally, we consider that the Rv (rate of reward) graph is more informative and therefore focus on this graph for further analysis.

2.8.3 Q learning for 4-disk puzzle

We now extend the analysis to the 4-disk puzzle and analyse learning over different time periods as in figure 7.

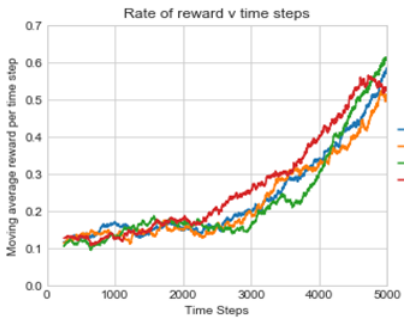
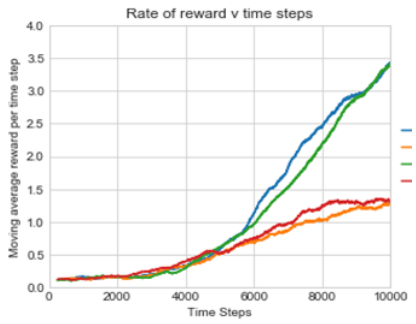
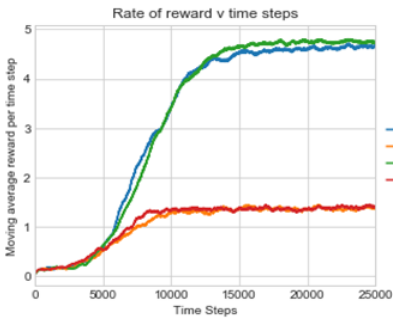
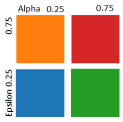
		
Between <u>3000 and 5000 steps</u>, the Q matrix has converged enough that exploitation (lower ε) becomes a better strategy than exploration ,with ***blue*** and ***green*** performance catching up 	<u>Beyond 5000 steps</u>, exploitation is clearly advantageous but improvement is slowing as the Q matrix is beginning to converge towards optimality. At this point, high learning rate has the benefit of retaining small improvements more quickly once they are discovered: ***blue*** and ***red*** reach convergence marginally more quickly.	<u>Beyond 10000 steps</u>, convergence is now clear for all agents. A high ε policy (***red/orange***) encourages exploration which results in convergence from about 10,000 timesteps but continued exploration means Rv converges at a much lower level. The earlier convergence for high learning rate (***red*** v ***orange*** and ***green*** v ***blue***) is visible but the advantage is small in both cases.

Figure 7: Q learning with 4 disks, 3,000 to 25,000 timesteps. All examples are averaged over 100 iterations with $\gamma = 0.75$, $\delta = 1$.

2.8.4 Q learning for 4-disk with epsilon decay

The learning policy used for all scenarios above has been with epsilon decay factor $\delta = 1$ i.e. constant ε.. Revising this policy to decay ε over time with $\delta < 1$, we can achieve a balance between rapid learning in the early stages followed by later exploitation. Ideally, the agent would first learn the optimum solution by exploring and then exploit without any further exploration, resulting in the optimum puzzle solution every time. As the optimum solution for the 4 disk puzzle is a reward of 100 every 15 steps, the optimum rate of reward (Rv^+) can be calculated as $Rv^+ = 100 * 1/15 = 6.67$.

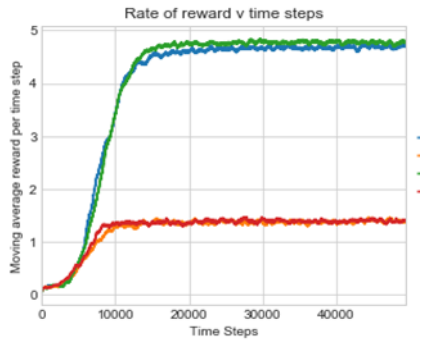
A decay policy was adopted as follows: $\delta = 0.99999$ *per timestep if* $\varepsilon > 0.5$
$\delta = 0.9999$ *per timestep if* $\varepsilon < 0.5$

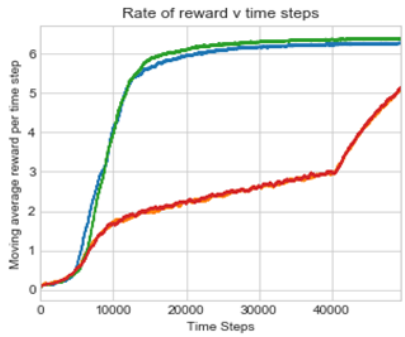
The impact of this policy on the initial ε parameters of 0.25 and 0.75 are shown in table 2.

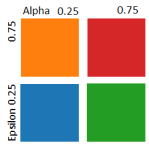
Steps	10000	20000	30000	40000	50000
Initial ε: 75%	68%	61%	56%	50%	18%
Initial ε: 25%	9%	3%	1%	0.5%	0.2%

Table 2:
Impact of ε decay on selected initial ε values

The learning behaviour resulting from adopting this decay policy is shown in figure 8.







Without decay, as per the scenario in figure 6 but over extended number of timesteps..

Initial learning behaviour is not materially different with and without ε decay (as shown below).

With decay, the impact of δ is clear beyond 10000 steps - all agents achieve a higher rate of reward than before.
For ***red/orange***, ε falls to 0.5 at about 40,000 steps. Whilst the rate of reward is higher than before up to this point, the acceleration after 40,000 steps is clear.
For ***blue/green***, ε is below 0.5 from the start and therefore decays quickly. Increased reward is noticeable after about 10,000 steps. Convergence towards $R_v^* = 6.67$ is clear, although somewhat later than convergence with $\delta = 1$.

Figure 9: Q learning with 4 disks and ε decay, 3,000 to 25,000 timesteps. All examples are averaged over 100 iterations with $\gamma = 0.75$, $\delta = 1$.

2.9 Sensitivity Analysis

2.9.1 Reflection

As shown above with the standard TH puzzle, the ε/greedy policy has a clear impact on learning and rate of reward. The best results are achieved with exploration (high ε) in early learning and exploitation (low ε) at a later stage. However, the learning rate (α) has minimal impact on learning in this scenario, as the challenge is a static (i.e. non-changing) puzzle from a 'no knowledge' starting point.

In order to investigate the Q learning process further, we need to extend the puzzle by introducing a non-static element. In the sensitivity analysis below, we introduce the concept of a temporary 'blockage' on the optimal path, i.e. a barrier that can change the routes to solving the puzzle part-way through learning.

2.9.2 Q learning for 4-disk 'blocked' puzzle

In the next stage of or analysis, we introduce a 'blockage' after 10,000 timesteps. By removing or re-introducing an allowed move, we introduce a dynamic change the Q learning state function. This blocking concept is illustrated on the 3-disk graph below, although our analysis continues with the 4-disk puzzle.

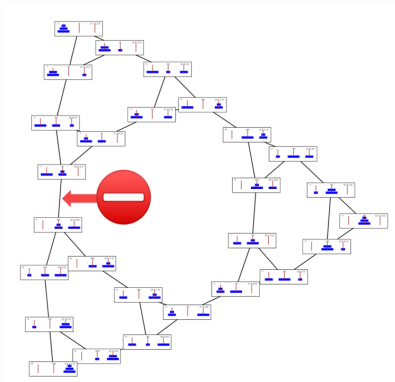


Figure 9: Illustration of blockage added to the Q learning puzzle after 10,000 moves. After blocking, the only available solutions are the 'long way round' and the impact on the optimum solution is as shown below..

Optimum solution	**Without block**	**With block**
3 disks	7 steps	11 steps
4 disks	15 steps	23 steps

The impact on learning of blocking after 10,000 moves is shown in figure 10.

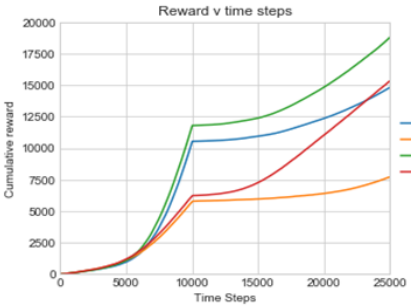
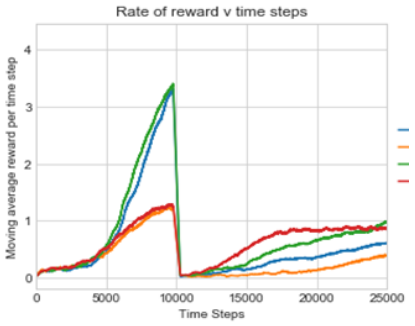
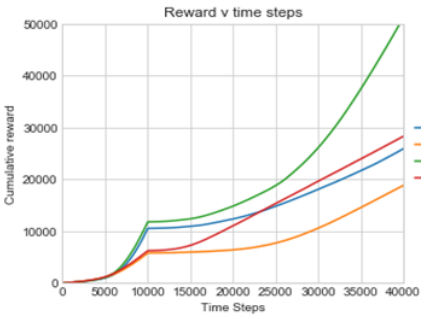
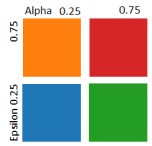
R_c	R_v	25,000-40,000 timesteps
		After blocking at 10,000 steps, re-learning is improved with both high ε (more exploration of previously non-optimum moves) and high α (more willingness to overwrite previous experience with new results). ***Red*** and ***green*** are seen to benefit from high α but ***red*** (with both high learning and exploration) performs best. Note that, with the longer optimum route, convergence in a 'blocked' puzzle is at a lower R_v level than before.
		By 40,000, ***orange*** and ***red*** are both converging on their new optimum, although ***orange*** has taken significantly longer to re-learn. After a slow start caused by low exploration (low ε), ***green*** is also converging on its new optimum but ***blue*** has performed poorly, with its 'greedy and stubborn' policy failing to learn the new puzzle. 

Figure 10: Q learning of 'blocked' puzzle with 4 disks. All examples are averaged over 100 iterations with $\gamma = 0.75$, $\delta = 1$.

2.9.3 Unblocking the 'blocked' puzzle

The impact of a non-static puzzle is further explored (figure 11) in a scenario where the blockage that was introduced at 10,000 steps is now removed at 40,000 steps.

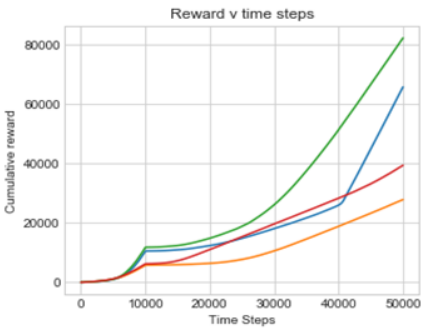
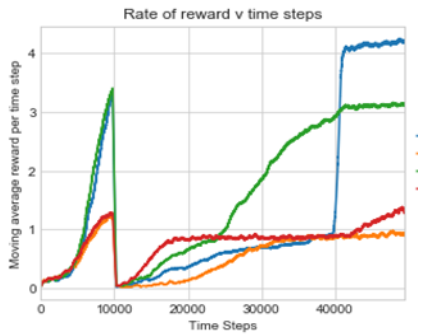
R_c	R_v	50,000 timesteps 
		The most significant change after removing the blockage at 40,000 steps is that the 'greedy stubborn' ***blue*** immediately improves. In this case, ***blue*** had failed to learn the new puzzle and still had a strong memory of the previous puzzle, which it immediately exploits. All other agents must re-learn the initial puzzle. Performance does not deteriorate as the 'long way round' the puzzle is still viable but this means that re-learning the short route takes longer than it did initially. Only ***red*** (with its high exploration and learning) has made noticeable relearning progress by 50,000 steps.

Figure 11: Q learning of 'blocked' puzzle with 4 disks. All examples are averaged over 100 iterations with $\gamma = 0.75$, $\delta = 1e$.

In summary, the dynamic puzzle created by blocking and unblocking the shortest path results in considerably different behaviour for each of the four agents.

- After blocking, only ***red*** (high α and ε) adapts quickly. 5,000 steps after blocking, ***red*** can solve the puzzle using a greedy policy 40% of the time with the optimum route. Other agents at this point only solve the puzzle less than 10% of the time.
- After unblocking, ***blue*** (with its 'greedy and stubborn') policy considerably outperforms as it is the only agent that has retained its knowledge of the original puzzle. Other agents have to re-learn and unsurprisingly ***red*** re-learns the fastest of the three. 5,000 steps after unblocking, agent performance with a greedy behavioural policy is very different for the four scenarios as shown in figure 12.

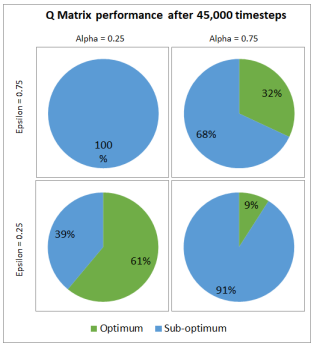
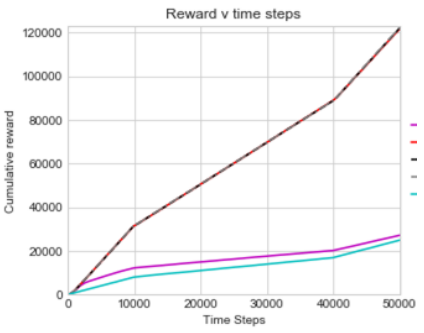
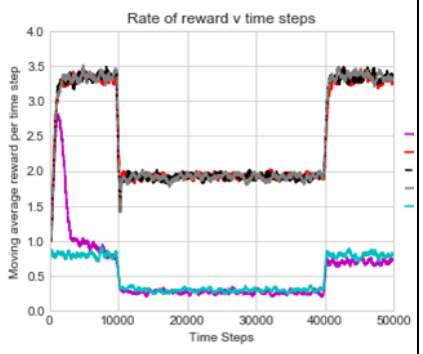


Figure 12: Greedy routes after 45,000 timesteps

2.9.4 Gamma sensitivity

The discount factor, γ, has been kept at 0.9 for all scenarios discussed so far in this paper. In general, we find learning is not sensitive to γ provided that it is neither 0 or 1 as shown in figure 13.

R_c	R_v	50,000 timesteps
		With a discount factor of $\gamma = 1.00$, there is no penalty for long routes to goal. Whilst ***magenta*** demonstrates some element of initial learning, the Q matrix gradually fills up with 100s and performance deteriorates. With a discount factor or zero, there is no learning at all and cyan does no better than a random walk. All values of γ away from these extremes perform consistently.

Graph legend:

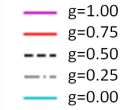


Figure 13: Q learning of 'blocked' puzzle with 4 disks and varying γ (g) between 0 and 1. All examples are averaged over 100 iterations with $\alpha = 0.75$ and $\varepsilon = 0.75$.

3 Advanced Techniques

3.1 Online planning

3.1.1 Non-deterministic Dyna-Q learning

Q-learning, as analysed so far in this report, has been shown to be an effective learning method however the Q-learning agent only updates its knowledge of the environment based on its current 'real-world' experience. An alternative approach to learning involves the agent maintaining a model to record past experiences and then using this model to make a prediction of the outcome of a particular state/action and updating its knowledge based on these replayed experiences as if they were in the real-world. This approach, referred to as 'online planning' [4][Sutton, 2018] is integrated into a dynamic planning, acting, learning process called Dyna-Q. In this scenario, the agent selects a number ('***n***') of past experiences at random after each timestep of real-world learning and the Q matrix is updated based on the replayed experience.

Dyna-Q learning can significantly accelerate the learning process. With ***n***=1, there are already twice as many learning opportunities but these may also be more fruitful in early learning as they can start to backfill learning for past routes to goal whilst the agent is actually exploring new parts of the state matrix without any real-world learning.

Sutton's Dyna-Q model [4] assumes a deterministic environment. In order to account for our dynamic 'blocking' move, we introduce a non-deterministic variant so that the agent can learn about changes to allowed actions, but only at the point the agent is in the appropriate state so that it can see those changes in real-time. This is achieved by updating the model **M** if the agent observes from its current state that a previously recorded state/action that is no longer allowed in the real-world.

Dyna-Q learning algorithm adapted for non-deterministic state changes (changes from Q-learning algorithm highlighted)	
1. Algorithm parameters:	Q learning parameters plus: number of experience replays per timestep $= n$
2. Initialisation:	Initialise null **Q(s,a)** matrix and current state (**S**) to 0 (start of the puzzle) as per Q learning Initialise null M(s,a) matrix for the past experience learning model
3. Loop for each episode:	Choose action a from s using epsilon-greedy policy. Take action a, observe reward (**r**) and state (**s'**) as per Q learning. Update M: **M(s,a)** ←**R(s')** (i.e. store reward/action in M, even if r is zero) Remove any previous actions from M if no longer valid Update Q: $\mathbf{Q(s,a)} \leftarrow \mathbf{Q(s,a)} + \alpha[\mathbf{r} + \gamma . \max_a \mathbf{Q(s',a')} - \mathbf{Q(s,a)}]$ Reduce ε by factor δ , if applicable, until **s** = terminal state (end of puzzle)
4. Loop n times	Select random state s and action a from previously observed state Observe **predicted** reward (**r**) and state (**s'**) from taking action a from s Update Q based on experience replay: $\mathbf{Q(s,a)} \leftarrow \mathbf{Q(s,a)} + \alpha[\mathbf{r} + \gamma . \max_a \mathbf{Q(s',a)} - \mathbf{Q(s,a)}]$
4. Repeat	Repeat step 3 until stopping criteria met as per Q learning

3.1.2 Dyna-Q for the 4-disk blocked/unblocked puzzle

A comparison of Dyna-Q learning with n=1 and n=5 is compared to normal Q-learning (i.e. n=0) in figure 14.

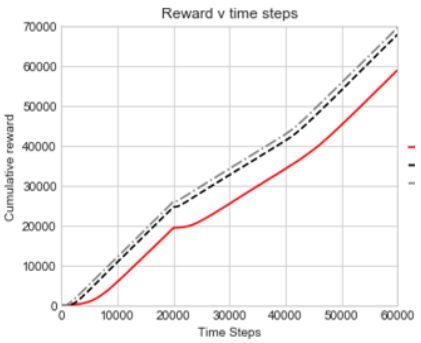
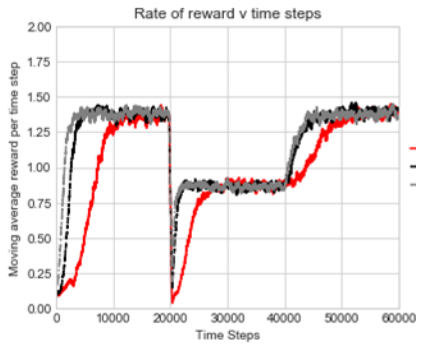
R_c	R_v	50,000 timesteps
		The impact of learning from experience replay is clearly illustrated. Whilst red reaches 50% of the maximum rate of reward after 6,000 timesteps, black (n=1) only takes 2,000 timesteps and grey (n=5) 1,000 timesteps to reach the same milestone. Blocking and unblocking the puzzle (at 20,000 and 40,000 timesteps respectively to allow ***red*** to converge first) we see that Dyna-Q is clearly beneficial at each learning stage n=0 n=1 n=5

Figure 14: Dyna-Q learning of 'blocked' puzzle with 4 disks showing learning for n=0,1 and 5. All examples are averaged over 100 iterations.

As expected, Dyna-Q clearly outperforms Q-learning for a fixed number of timesteps. However it is worth noting that this improvement is only of practical benefit if off-line processing can be done whilst waiting for the next timestep. If the timesteps are only limited by processing power (for example when solving a problem programmatically), it is likely to be as beneficial to simply process timesteps faster.

3.2 Function approximation

3.2.1 Linear function approximation

As described in this paper, the Q-learning algorithm is a random/greedy walk through the state-action space. While this algorithm has nice convergence properties, it can be computationally expensive with larger Ns, as each state-action pair has to be sampled a sufficient number of times to fill the Q matrix to adequate precision.

Additionally, since the Q-values are stored in a lookup table, memory requirements might become too large.
An alternative approach is to summarize the state-action space via function approximators fi(s,a). In this representation the Q function can be written as a function of those approximators.

We look at the case where the representation is a linear combination of the approximators ie $Q(s,a) = \sum_{i=1}^{K} w_i f_i(s,a)$

Learning the Q function then becomes simply a matter of evaluating the weights' vector $W = \{w_i\}_{i=1,..K}$

It can be shown that determining the optimal weights can be done by a simple gradient descent over the weight space and the step-update is very similar to the traditional Q-learning algorithm, thanks to the linear modelling.

Q learning algorithm with function approximation	
1. Algorithm parameters:	Q learning parameters.
2. Initialisation:	Initialise weight vector **W** Initialise current state (**s**) to 0 (start of the puzzle)
3. Loop for each episode:	Choose action a from s using epsilon-greedy policy. Take action a, observe reward (**r**) and state (**s'**) as per Q learning. Update $W = \{w_i\}_{i=1,..K}$: $\mathbf{w_i} \leftarrow \mathbf{w_i} + \alpha\,[\mathbf{r} + \gamma \cdot \max_{a'}\mathbf{Q(s',a')} - \mathbf{Q(s,a)}]\,\mathbf{f_i(s,a)}$ always using the relation: $Q(s,a) = \sum_{i=1}^{K} w_i f_i(s,a)$ Reduce ε by factor δ , if applicable until **s** = terminal state (end of puzzle)
4. Repeat	Repeat step 3 until stopping criteria met (convergence or maximum number of timesteps/episodes)

3.2.2 Results

We first validate the algorithm by considering an approximator that exactly represents the entire state/action space.

It is defined as $f(s,a) = \sum_{i=1}^{N_s}\sum_{j=1}^{N_a} w_{i,j} f_{i,j}(s,a)$ where $f_{i,j}(s,a) = \{1 \text{ if } s = states[i] \text{ and } a = actions[j] \text{ and } f_{i,j}(s,a) = 0 \text{ otherwise}\}$

Here, *states* and *actions* are the lists of all N_s states and all N_a actions respectively. Note that there are a total of 6 possible actions, defined as 2-tuples (p_n, p_m) representing a move from pole p_n to p_m, ie. $actions = [(p_1,p_2),(p_1,p_3),(p_2,p_1),(p_2,p_3),(p_3,p_1),(p_3,p_2)]$
So in this case we have effectively $w_{i,j} = Q(states[i], actions[j])$ which is the exact representation of a full Q matrix.

Figure 15 shows convergence behaviors and highlights a power law type of weight distribution in the no-approximation case.
We observe convergence for N=3, N=4 and N=5 with the following parameters:
[alpha=0.05, gamma=0.8, epsilon=0.5, episodes=5000, decay=True, maxsteps=1000].

We then try a series of different approximators. Each time we summarise a state by a set of N_f features $\{f_i(s)\}_{i=1..N_f}$ from which we create a function by combining all possible actions (allowed or not) to give a total of 6xF free parameters (plus a bias).

So we have: $f(s,a) = w_0 + \sum_{i=1}^{N_f}\sum_{j=1}^{6} w_{i,j} \cdot f_i(s) \cdot h_j(a)$ where $h_j(a) = \{1 \text{ if } a = action[j],\ 0 \text{ otherwise}\}$ and with w_0 the bias term.

To be clear, the approximated features $f(s,a)$ correspond to the representation of state s' which is reached from s after taking action a.
We consider the following approximators:

- F1: 3 features per state corresponding to the compact state c1c2c3 described in 2.2.3
- F2: 6 features: compact state plus normalised number of disks on each pole
- F3: 3 features: macro state (value of the top disk on each pole)
- F4: 6 features: macro state plus pole heights (normalised number of disks on each pole)
- F5: 6 features: macro state plus pole weights (normalised weight of each pole (sum of the codes Z(i) of each pole))
- F6: 9 features: macro state plus pole heights plus pole weights
- F8: 9 features: macro state plus inverse of pole heights plus inverse of pole weights
- We also try various other combinations and inverse functions of the above features

We only observe convergence for approximator F4 for N=3. In any other case we see no convergence of the weight vector, and the result of every training shows the agent making a few moves out of state 0 then alternate between 2 or 3 attractor states indefinitely. In those cases, only a few weights (if any) show monotonic to stable behaviour.

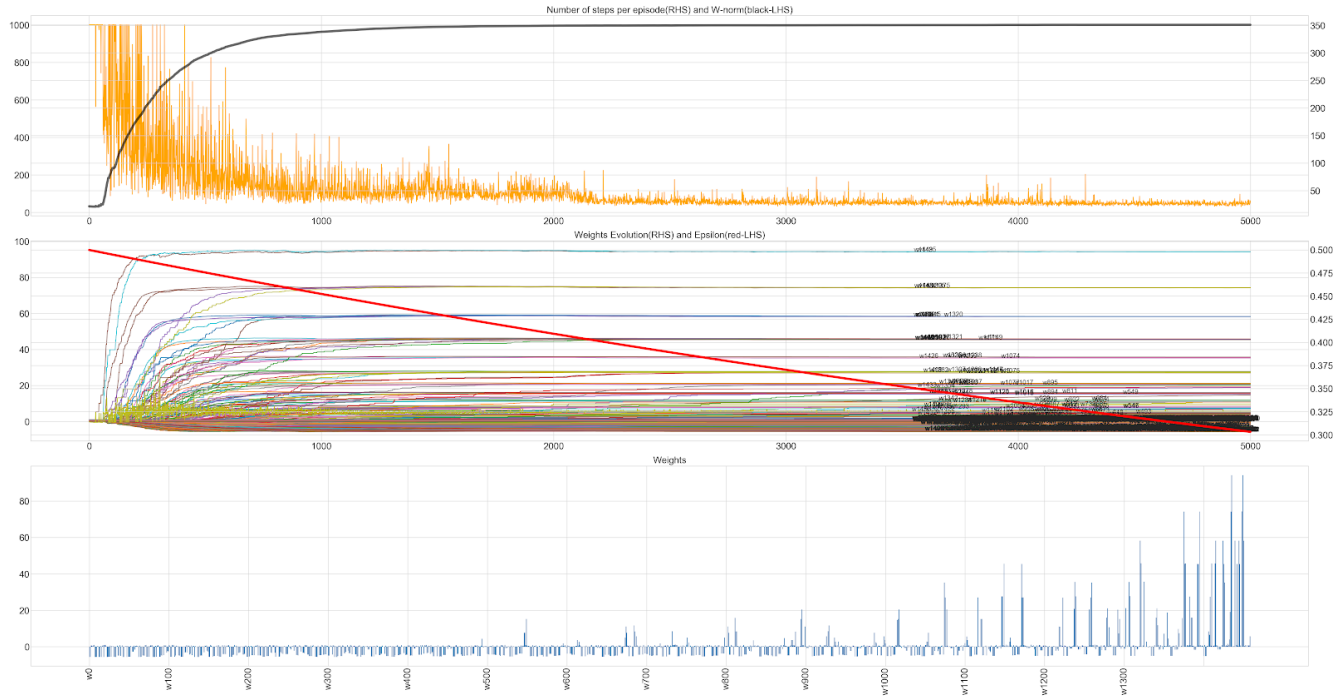


Figure 15: Benchmark case: Function approximation with no approximation for 5 disks. Total number of steps,Weight vector norm, epsilon and individual weights per episode.

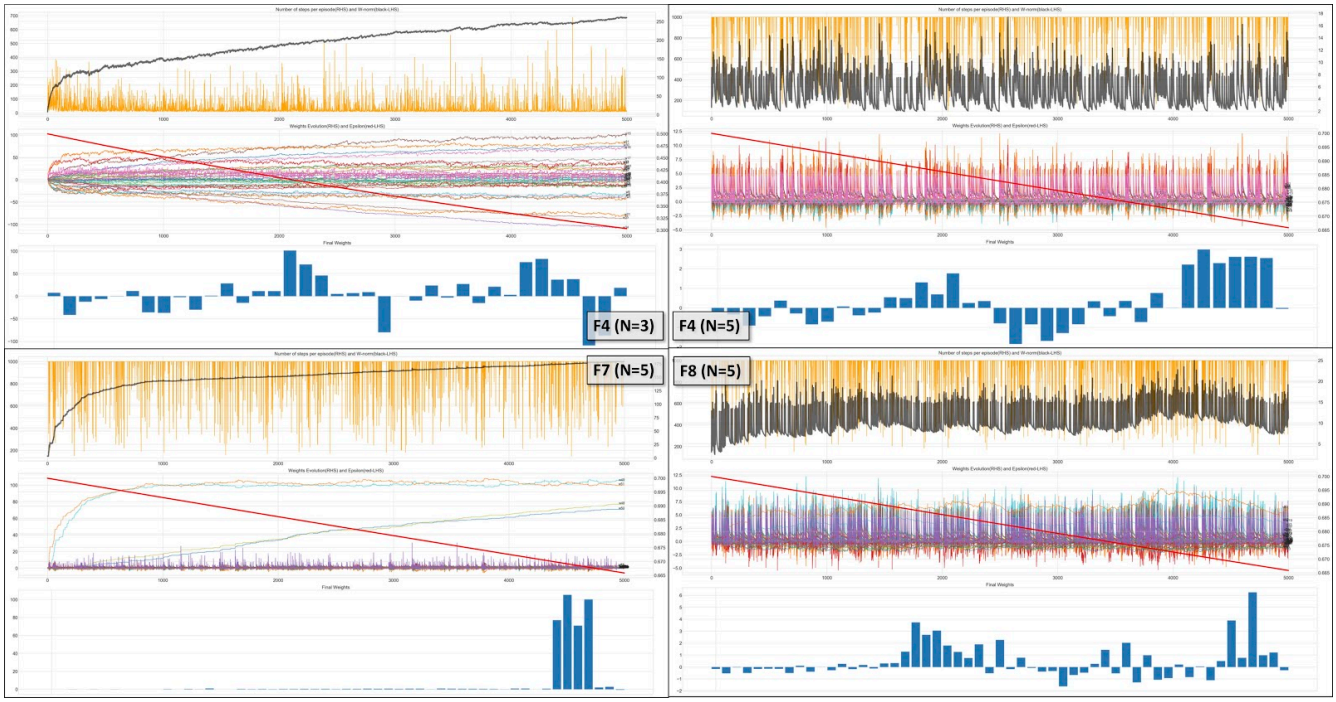


Figure 16: Comparison between different approximators: note the weak convergence on the only approximator that converges (F4 for 3 disks)

There are at least two possible reasons for this behaviour:

1. The approximated state-action space is too sparse for the function approximation algorithm to attain the optimal policy: the number of free parameters could be too small to approximate this recursive problem. Adding more feature is the obvious solution since we know that, at the limit, the algorithm converges for full state representation, but this is more art than science and the absence of a theoretical convergence proof might mean wasted efforts.
2. The objective function contains many local extremas and the gradient descent algorithm appears to be easily stuck on one of those. Varying the step size dynamically (adding momentum) could be the solution but given that we are running an epsilon-greedy algorithm we would need to set different alphas for each point of the search space.

4 Conclusions

4.1 Summary of lessons from 4-disk model learning

The main lessons from the work summarised in this paper were as follows:

1. Learning appears to be most sensitive to the epsilon parameter. Generally, high epsilon in early learning (exploration) and low epsilon later (exploitation) is the best strategy, justifying the popular use of epsilon decay..
2. Alpha has minimal impact on a deterministic matrix but can have a significant impact when a non-deterministic element is introduced, such as the path blocking/unblocking concept discussed
3. Adding a planning element through experience replay can significantly improve Q-Learning performance, although practically this is only useful if there is spare time to perform additional calculations before the next time step of normal Q-learning can occur.
4. Reinforcement learning with linear function approximation does not seem suited for the TH problem, due to the recursive nature of the problem and what looks like a self-similar form of the Q function.

4.2 Beyond 4-disks

This paper has focused primarily on the 4-disk Hanoi puzzle with 81 states. The learning challenge becomes greater as the number of disks increases. With 6 disks (729 states), an agent applying Q-learning would be expected to solve the puzzle only 7 times in 250,000 timesteps and has therefore only populated 7 (0.3%) of the allowed moves in the matrix. With this added complexity in the state space, the enhanced performance achievable through a policy such as dyna-Q that includes a planning element becomes more critical. The 6-disk puzzle is solved nearly 400 times in 250,000 timesteps with n=1 (50x improvement on n=0) and nearly 600 times with n=5 (80x improvement) as shown in figure 16.

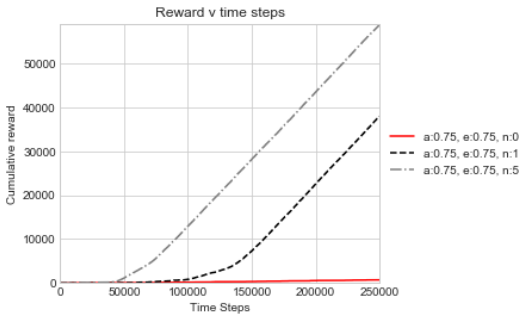


Figure 16: Dyna-Q learning of 6 disk puzzle (no blocking) showing learning for n=0,1 and 5 over 250,000 timesteps..
Averaged over 10 iterations with $\alpha = 0.75$ and $\varepsilon = 0.75$.

4.3 Further work

As a Q-learning challenge, the puzzle discussed in this paper was enhanced by the simple blocking strategy discussed in section 2.9.2. This concept could be extended in further work by analysing the impact of a blocking strategy that is more dynamic in either time or location.

Whilst our attempts at function approximation had limited success, successful neural network solutions have been published [5][Kaplan, 2001] and similar approaches, particularly using non-linear function approximations, could be considered in further work.

5 References

[1] Wikipedia, https://en.wikipedia.org/wiki/Tower_of_Hanoi.
[2] Hinz, A. M. et al. (2013) The Tower of Hanoi - Myths and Maths. doi: 10.1007/978-3-0348-0237-6.
[3] Watkins, J.C.H et al (1992) Q-learning. doi: 10.1007/BF00992698
[4] Sutton, R and Barto, A (2018) Reinforcement Learning: An Introduction Second edition, in progress pp 129-133
[5] Kaplan, G. B., & Güzeliş, C. (2001). Hopfield networks for solving Tower of Hanoi problems. ARI - An International Journal for Physical and Engineering Sciences, 52(1), 23. https://doi.org/10.1007/s007779900077