Q-Learning Tower of Hanoi

Yann Adjanor, Richard Mann, Feb 2018

5 Python files are provided to demonstrate Q-learning for the Tower of Hanoi project.

The first 4 files are identical apart from minor changes to parameters as described below.

The last file is dedicated to function approximators and the code has different features

Each of the first 4 files is structured as follows:

- Function imports
- Define global variables
- Define functions for Q learning (see below)
- Run Q learning once to show optimum solution graphically
- Create parameter grid for analysing the impact of varying alpha, epsilon etc
- Iterate through parameter grid (and average each setting over n iterations to obtain an average result), storing results in arrays
- Summarise results and display graphs

The script allows two optional variations to Q learning of the standard puzzle.

1. Blocking and unblocking the optimal path so that the agent has to re-learn the puzzle at a certain point.
2. Addition of experience to Q learning, i.e. Dyna-Q Learning

Functions defined in the script are structured as follows:

- Functions to define the Hanoi puzzle for n disks
    - prime_list
    - is_valid_state
    - encode_state
    - decode_state
    - macro_state
    - next_allowed_state
- Functions to create the R/Q matrix for the puzzle
    - set_R_matrix
    - init_Q_matrix
    - init_M_matrix (for dyna-Q)
- Q learning functions
    - Q_learning_only_one_step_ahead (random start, one step episodes - not used in report)
    - Q_learning_until_goal_state (Q learning always to goal state without convergence test)
    - Q_learning_until_goal_state_conv (Q learning always to goal state with convergence test, blocking/unblocking and dyna-Q)
    - run_agent
- Q learning non-core functions
    - create_parameter_mesh (to create grid of alpha, epsilon, gamma parameters)
    - convergence_graph
    - step_progression_graph
    - manage_data (save and reload data)
- Hanoi tower visualisations
    - display_state
    - save_solution_images
    - save_all_state_images
    - dispay_graphviz
    - display_graphnx
- Q learning greedy behaviour test
    - length_to_solve

The four versions of the script provided have different settings as described below.

Hanoi_Final1_Qlearning_Convergence_3disks.py

Settings:

3 disk problem. Normal Q learning, no blocking/unblocking.

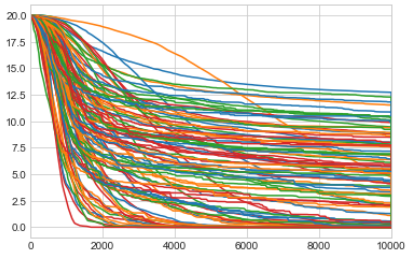
100 different parameters (all alpha, epsilon combinations 0.1,0.2…1) averaged over 5 iterations.

Graphical output:

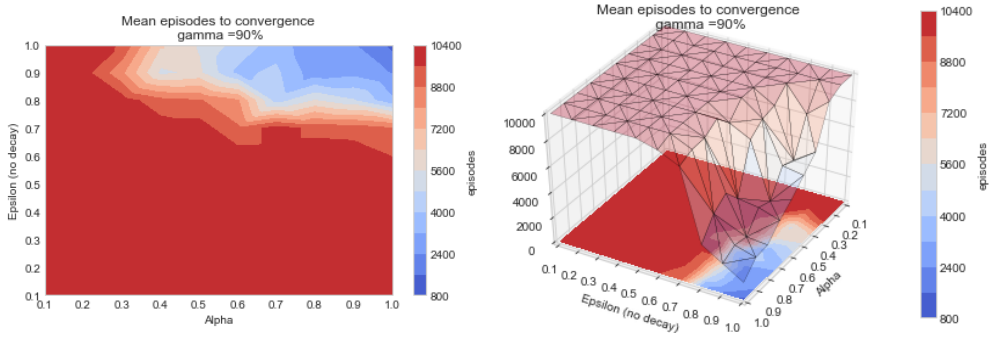
Solution:

.....

RMSE between Q and Q converged over timesteps



Mean number of episodes to achieve convergence (capped at 10,000 = no convergence)



Hanoi_Final2_Qlearning_Rgraphs.py

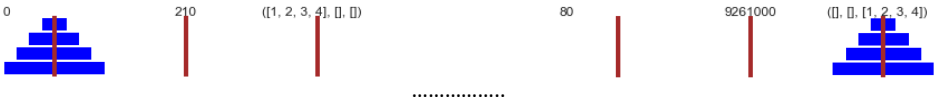
Settings:

4 disk problem. Normal Q learning, no blocking/unblocking.

4 different parameters (all alpha, epsilon combinations 0.25 and 0.75) averaged over 5 iterations.

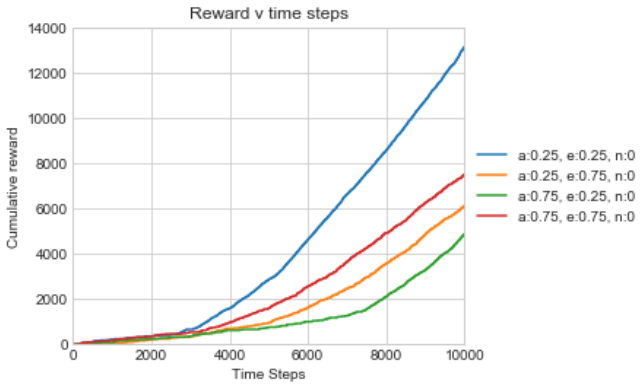
Graphical output:

Solution:

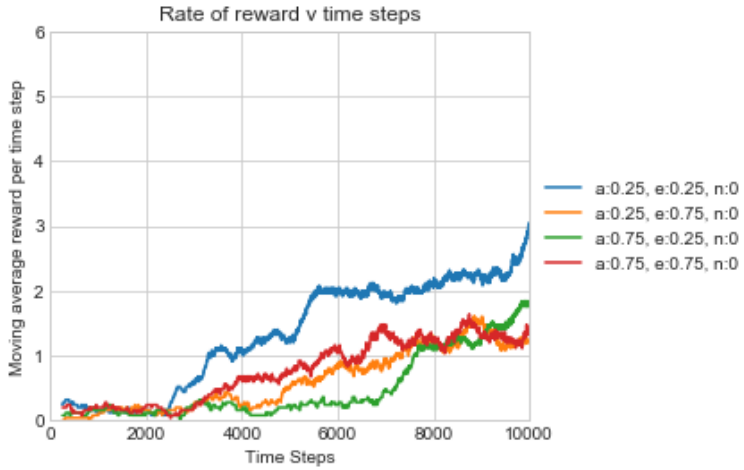


.....

Reward v timesteps



Rate of reward v timesteps



Hanoi_Final3_Qlearning_with_Blocking_Rgraphs.py

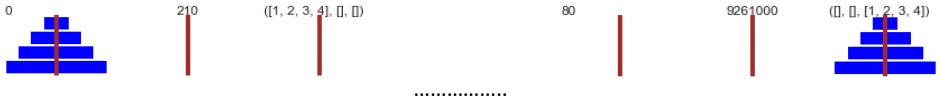
Settings:

4 disk problem. Normal Q learning, but with added blocking (at 20,000 steps) /unblocking (at 40,000 steps) of optimal path.

4 different parameters (all alpha, epsilon combinations 0.25 and 0.75) averaged over 5 iterations.

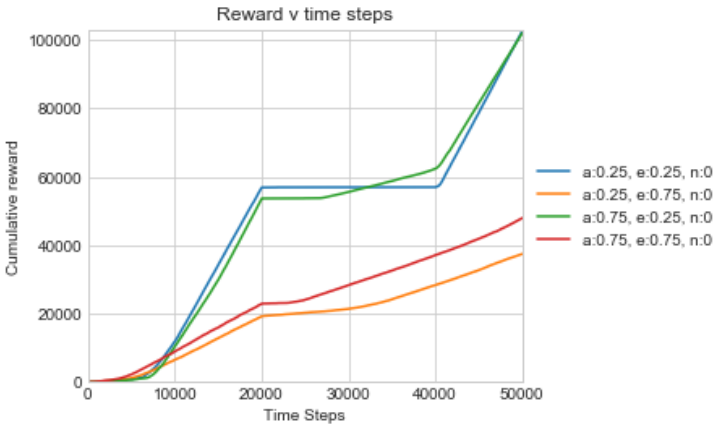
Graphical output:

Solution:



.....

Reward v timesteps



Rate of reward v timesteps

Hanoi_Final4_DynaQlearning_Rgraphs.py

Settings:

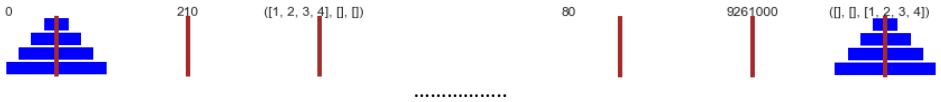
4 disk problem. Normal Q learning, no blocking.

But experience replay added with n=0 (normal Q learning), n=1 and n=3.

Only 1 setting for alpha and epsilon (0.75) averaged over 5 iterations.

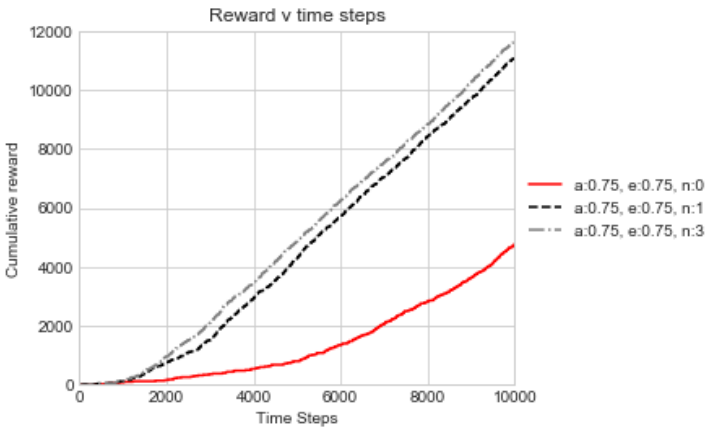
Graphical output:

Solution:



.....

Reward v timesteps



Rate of reward v timesteps

Hanoi_Final5_Qlearning_FunctionApproximation.py

Settings:

3 or 4 disk problem. Normal Q learning, no blocking.

No Q or R matrix but Q and R functions. The Q function is defined by a function approximator

There are 9 function approximators defined (f1 to f9). Changing the function involves changing the code in:

```
def f_approx(nDisks, Z, states, s, a):
    return f4(nDisks, Z, states, s, a, normalise=True)
```

All parameters settings hardcoded but changeable.

Output:

Whether the algorithm has converged or not:

```
solution from state #0 to state #26 in 8 steps
([1, 2, 3], [], [])
([2, 3], [], [1])
([3], [2], [1])
([3], [1, 2], [])
([], [1, 2], [3])
([], [2], [1, 3])
([1], [2], [3])
([1], [], [2, 3])
([], [], [1, 2, 3])
```

Graphical output:

Results of the learning run showing the norm of the weight vector and number of steps episodes (1st graph), value of all individual weights and value of epsilon (2nd graph) and value of final weights (last graph)