

# hw6\_report

December 29, 2022

## 1 MAS Final Homework Assignment

Daniel Engbert, Dec 2022

```
[ ]: # helper function for use in code code below
import random
import numpy as np

def set_seed(seed=42):
    rng = random.Random()
    if seed is None:
        seed = random.randint(0, 9999)
    rng.seed(seed)
    np.random.seed(seed)
```

### 1.1 Monte Carlo Estimation of Shapley Value

How to fairly split a taxi fare across  $\{1, \dots, N\}$  players can be calculated using [Shapely values](#), which treats this scenario as a cooperative game. For this problem we assume that all players live on the way of the  $N$ th player, and that player  $A$  lives at distance of 1, player  $B$  at a distance of 2, etc.

#### 1.1.1 Computing Shapley values for $n = 4$ :

The code below computes all permutations (a.k.a. “coalitions”) of the  $N = 4$  players  $\{A, B, C, D\}$ , then for each permutation, `perm`, it assigns each player a share off the total payoff (the fare of total cost 4), by assuming player `perm[0]` showed up first at the taxi, followed by players `perm[1]`, `perm[2]`, `perm[3]`.

For example, for the permutation `perm = ['A', 'C', 'B', 'D']`, the respective payoffs (fare splits) will be `{'A': 1, 'B': 0, 'C': 2, 'D': 1}` (here player  $B$  pays 0 because by the time he arrives at the taxi, it was already going to go by his house anyways due to player  $C$ .)

The final computed Shapely values are simply the average payoffs of each player across all possible permutations.

```
[ ]: import shapley
import importlib
importlib.reload(shapley)
shapley.get_shapley(4, display=display)
```

```
player_vals = {'A': 1, 'B': 2, 'C': 3, 'D': 4}
there are 24 total permutations of 4 players:
```

```
[['A', 'B', 'C', 'D'],
 ['A', 'B', 'D', 'C'],
 ['A', 'C', 'B', 'D'],
 ['A', 'C', 'D', 'B'],
 ['A', 'D', 'B', 'C']]
```

(only the first 5 rows of permutations are shown above)

```
shapley_values: (for N = 4)
{'A': 0.25, 'B': 0.5833333333333334, 'C': 1.0833333333333333, 'D':
2.0833333333333335}
```

As can be seen from the output, the shapely values for  $n = 4$  players is {'A': 0.25, 'B': 0.583, 'C': 1.083, 'D': 2.083}

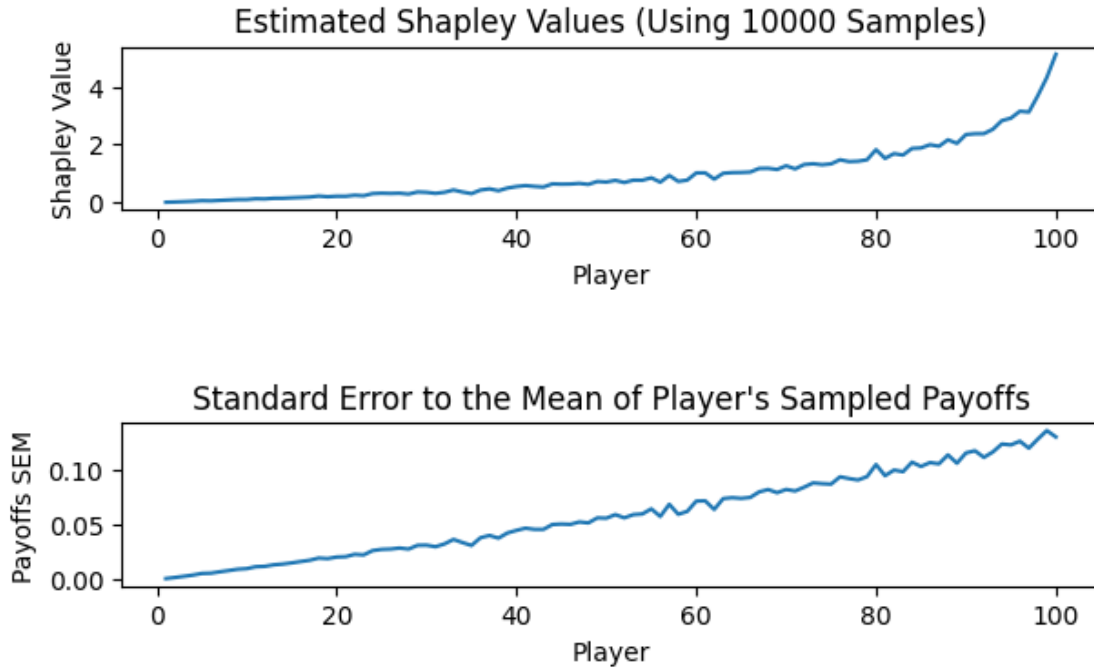
### 1.1.2 Estimating Shapley values for $n = 100$

Here we use [Monte Carlo sampling](#) to approximate the Shapely values for the taxi fare problem when  $n = 100$ .

```
[ ]: import shapley
import importlib
importlib.reload(shapley)
shapley.estimate_shapley(100, 10000)
```

```
estimated shapley values: (for N = 100 players, using 10000 sampled
permutations):
{1: 0.0102, 2: 0.0192, 3: 0.0306, 4: 0.0435, 5: 0.0641, 6: 0.0581, 7: 0.0743, 8:
0.0877, 9: 0.102, 10: 0.1034, 11: 0.1299, 12: 0.1246, 13: 0.1463, 14: 0.1481,
15: 0.1623, 16: 0.1745, 17: 0.1888, 18: 0.2236, 19: 0.2, 20: 0.2179, 21: 0.2164,
22: 0.2511, 23: 0.2335, 24: 0.3087, 25: 0.3216, 26: 0.3139, 27: 0.3215, 28:
0.292, 29: 0.3649, 30: 0.3516, 31: 0.3158, 32: 0.3531, 33: 0.4326, 34: 0.3621,
35: 0.3065, 36: 0.4308, 37: 0.4716, 38: 0.4078, 39: 0.4996, 40: 0.5518, 41:
0.5836, 42: 0.5561, 43: 0.5352, 44: 0.6429, 45: 0.6329, 46: 0.638, 47: 0.6602,
48: 0.6254, 49: 0.7262, 50: 0.7068, 51: 0.7648, 52: 0.6908, 53: 0.7716, 54:
0.7686, 55: 0.852, 56: 0.7033, 57: 0.9323, 58: 0.7278, 59: 0.7735, 60: 1.0214,
61: 1.0224, 62: 0.8086, 63: 1.0097, 64: 1.0318, 65: 1.037, 66: 1.0509, 67:
1.1761, 68: 1.1804, 69: 1.1405, 70: 1.2675, 71: 1.1605, 72: 1.3113, 73: 1.3365,
74: 1.3021, 75: 1.3328, 76: 1.4752, 77: 1.4125, 78: 1.4255, 79: 1.4755, 80:
1.8237, 81: 1.5231, 82: 1.6873, 83: 1.6377, 84: 1.8663, 85: 1.8829, 86: 1.9941,
87: 1.9405, 88: 2.1647, 89: 2.0406, 90: 2.3452, 91: 2.373, 92: 2.3764, 93:
2.5294, 94: 2.8259, 95: 2.911, 96: 3.1494, 97: 3.1216, 98: 3.6891, 99: 4.3161,
100: 5.1158}
```

```
sum of estimated shapley values: 100.00000000000003
```



Above we estimate the shapley values for  $n = 100$  players using 10,000 samples from the set of all permutations of player coalitions of size  $n$ .

As can be seen from the standard error to the mean graph, these estimated shapley values are very accurate (monte carlo) estimations of the exact shapley values for the players.

Additionally, the sum of the estimated shapley values is extremely close to 100.0 which is to be expected and is further evidence that these estimated shapley values are accurate.

---

## 1.2 Monte Carlo Tree Search (MCTS)

```
[ ]: import tree_search as ts
import importlib
importlib.reload(ts)

depth = 20
c = 2.0
mc = ts.MCTS(depth, c=c, B=25, draw=False, save=False)
target_value = mc.tree.nodes[mc.target_name]["value"]
print(f"initialized tree with depth={depth}, target_name = (node) {mc.
    ↪target_name} (with value {target_value:.3f})")

# run Monte Carlo Tree Search
tra = mc.run()
```

```

value = mc.tree.nodes[tra[-1]]["value"]
dist = ts.MCTS.edit_distance(
    mc.tree.nodes[tra[-1]]["address"], mc.tree.nodes[mc.target_name]["address"]
)
print(f"\noptimal trajectory =\n{mc.get_path(mc.target_name)}")
print(f"\nreturned trajectory =\n{tra}")
print(
    f"\nvalue of terminal node in trajectory: {value:.2f} ({100 * value /
    ↪target_value:.2f}% of target_value {target_value:.2f}), distance = {dist}"
)

```

initialized tree with depth=20, target\_name = (node) 837183 (with value 25.124)

optimal trajectory =  
 [1, 3, 6, 12, 25, 51, 102, 204, 408, 817, 1635, 3270, 6540, 13080, 26161, 52323, 104647, 209295, 418591, 837183]

returned trajectory =  
 [1, 3, 6, 12, 25, 51, 102, 205, 410, 821, 1643, 3286, 6572, 13144, 26289, 52578, 105157, 210314, 420628, 841257]

value of terminal node in trajectory: 6.98 (27.80% of target\_value 25.12),  
 distance = 4

Above we initialize a tree with *depth* = 20, a random leaf node is selected as the target node (assigned the highest value), and the other leaf nodes are assigned values such that nodes closer to the target have higher values.

Monte Carlo Tree Search (MCTS) is then used to search for the most optimal leaf node, and the resulting trajectory (final path through the tree found by MCTS) is reported above as well as the relative value of the leaf node found by MCTS vs the value of the target node.

```

[ ]: import math
from collections import defaultdict
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np
import random
import statistics
import importlib
import tree_search as ts
importlib.reload(ts)
MCTS = ts.MCTS

##### experiment params
#depth = 10
depth = 4    # depth of tree to search

```

```

B = 25          # for computing values of leaf nodes
c = 2           # hyperparam
#####

#mc = MCTS(depth, c=c, B=B, draw=True)

def make_graph(data, show=True, save=False, samples=None):
    plt.clf()
    fig, ax = plt.subplots(1, 1)
    ax.set_xlabel("UCB c Value")
    ax.set_ylabel("Average Payoff Received (%)")
    ax.set_title(
        f"Effect of UCB c Value on Payoffs (MCTS with Depth {depth}, Samples = {samples})"
    )
    ax.errorbar(data.keys(), [sum(d)/len(d) for d in data.values()],
        yerr=[statistics.stdev(d)/2 for d in data.values()])
    #ax.errorbar(data.keys(), [sum(d)/len(d) for d in data.values()], yerr=[50
    for d in data.values()])
    if save:
        plt.savefig("c_graphs.pdf", dpi=400)
    if show:
        plt.show()

data = defaultdict(list)
depth = 15
NUM_SAMPLES = 5
iter = 0
for c in np.linspace(0, 5, num=10):
    if iter % 4 == 0:
        print(f"testing with c = {c}")

    #if c % 0.25 == 0:
    for _ in range(NUM_SAMPLES):
        mc = MCTS(depth, c=c, B=B, draw=False)
        tra = mc.run()
        value = mc.tree.nodes[tra[-1]]["value"]
        target_value = mc.tree.nodes[mc.target_name]["value"]
        data[c].append(100 * value / target_value)
make_graph(data, samples=NUM_SAMPLES, show=True, save=True)

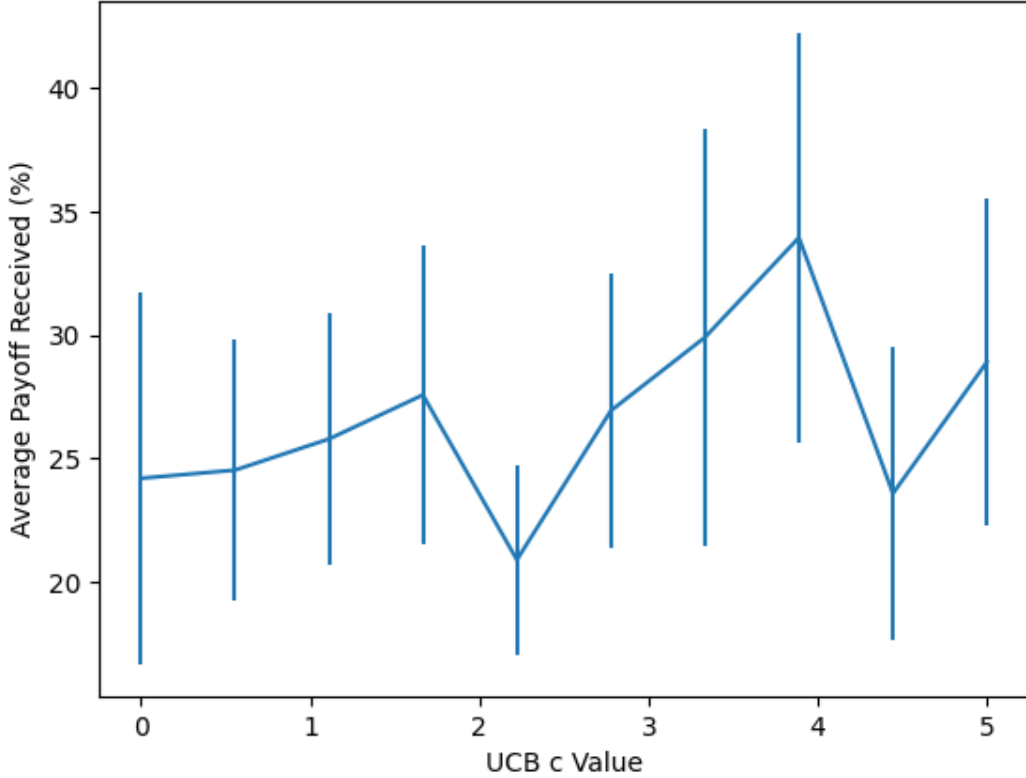
```

testing with c = 0.0

testing with c = 5.0

<Figure size 640x480 with 0 Axes>

Effect of UCB  $c$  Value on Payoffs (MCTS with Depth 15, Samples = 5)



Above we experiment with the effect of the param  $c$  on MCTS.  $c$  is a hyperparam for UCB calculation where higher values encourage more exploration, and lower values encourage exploitation.

UCB values in the range  $[0, 5]$  are tested, and for each UCB value, MCTS is conducted for `NUM_SAMPLES=5` independent iterations, and the average percent of the maximum payoff is reported. The vertical error bars depict the standard deviation of each sample set.

Based on these results,  $c = 4$  appears to be most efficient for this problem space. Additional experiments could be conducted to further explore the effect of the  $c$  param in trees of varying depths.

## 1.3 RL: SARSA and Q-Learning

### 1.3.1 Strategy 1: SARSA with Greedification

For the first experiment, SARSA is used in combination with greedification to search for an optimal policy.

During learning, the Q-table is updated while rolling out each episode path.

Specifically we simulate *iterations* = 350 episodes (starting at a random initial empty position each time) and perform learning with salsa and pure greedy selection) until each episode terminates.

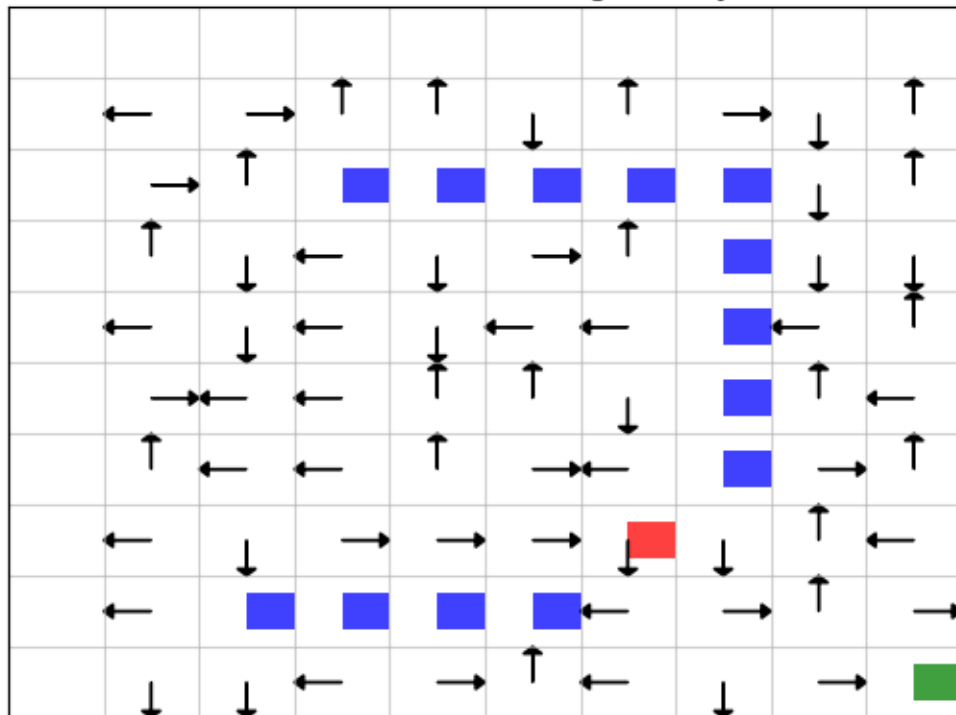
```
[ ]: import importlib
import qlearn as ql
importlib.reload(ql)
# sarsa with greedification
sim = ql.direct_updates(sarsa=True, epsilon=0.0, iterations=350,
↳display=display)
```

initial Q table:

	s	a	r	q
0	(0, 0)	2.0	-1.0	-0.009565
1	(0, 0)	0.0	-1.0	0.040704
2	(0, 0)	1.0	-1.0	-0.087852
3	(0, 0)	3.0	-1.0	0.091506
4	(0, 1)	2.0	-1.0	-0.100777
..	...	...	...	...
319	(8, 7)	3.0	-1.0	-0.060036
320	(8, 8)	2.0	50.0	0.000000
321	(8, 8)	0.0	-1.0	0.070333
322	(8, 8)	1.0	50.0	0.000000
323	(8, 8)	3.0	-1.0	0.215316

[324 rows x 4 columns]

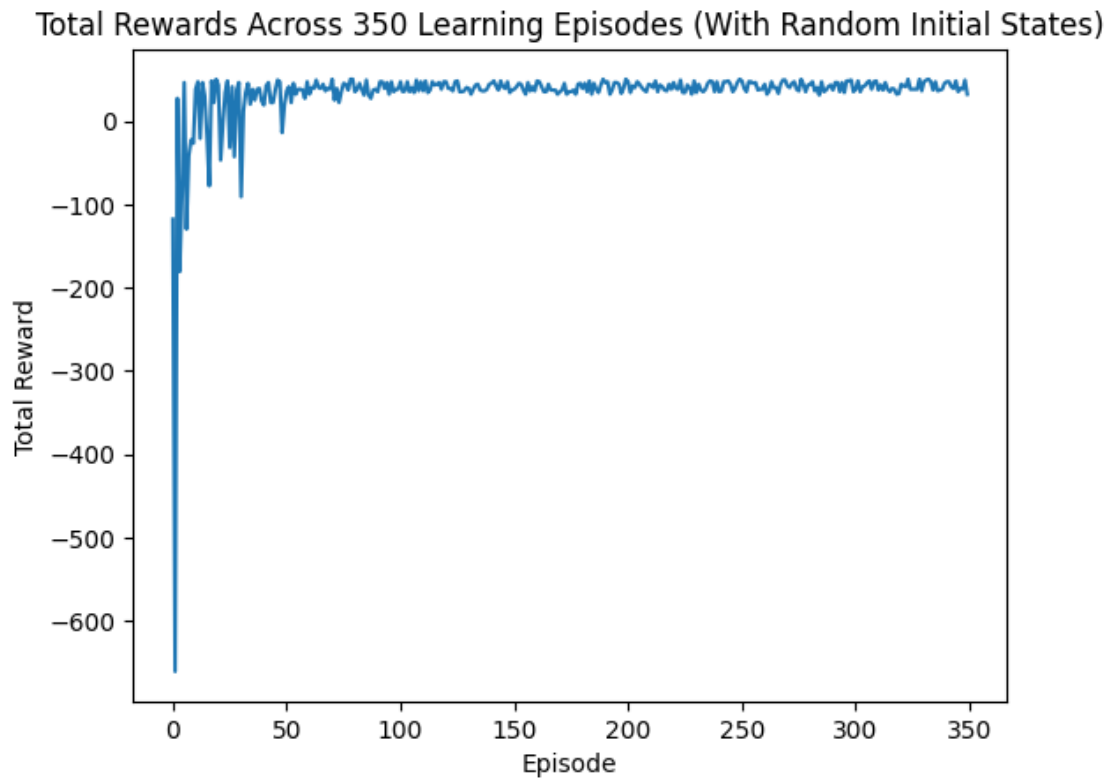
Initial Random Q table (Showing Greedy Actions)



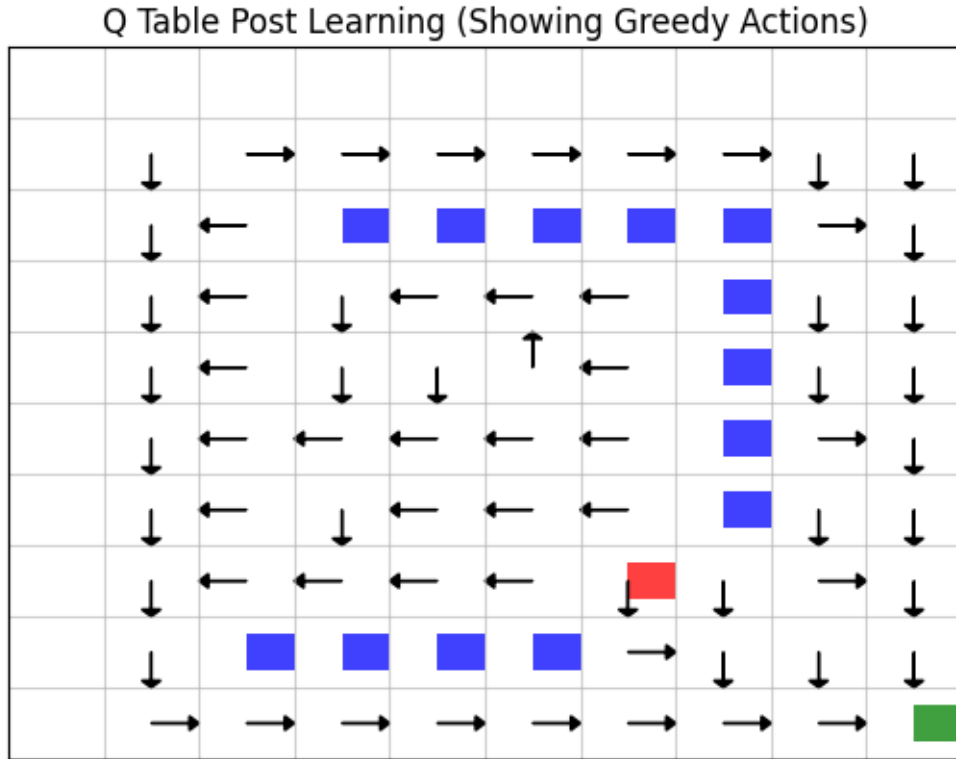
Q table (post learning):

	s	a	r	q
0	(0, 0)	2.0	-1.0	-2.879648
1	(0, 0)	0.0	-1.0	-2.985564
2	(0, 0)	1.0	-1.0	-2.984675
3	(0, 0)	3.0	-1.0	-2.985323
4	(0, 1)	2.0	-1.0	-2.980470
..	...	...	...	...
319	(8, 7)	3.0	-1.0	-0.060036
320	(8, 8)	2.0	50.0	0.000000
321	(8, 8)	0.0	-1.0	0.070333
322	(8, 8)	1.0	50.0	0.000000
323	(8, 8)	3.0	-1.0	0.215316

[324 rows x 4 columns]







**Strategy 2a: Q-Learning (Direct Updates)** Here we directly update the Q-table while rolling out each sample path.

Specifically we simulate  $iterations = 350$  episodes (starting at a random initial position each time) and perform q-learning with  $\epsilon$ -greedy selection) until each episode terminates.

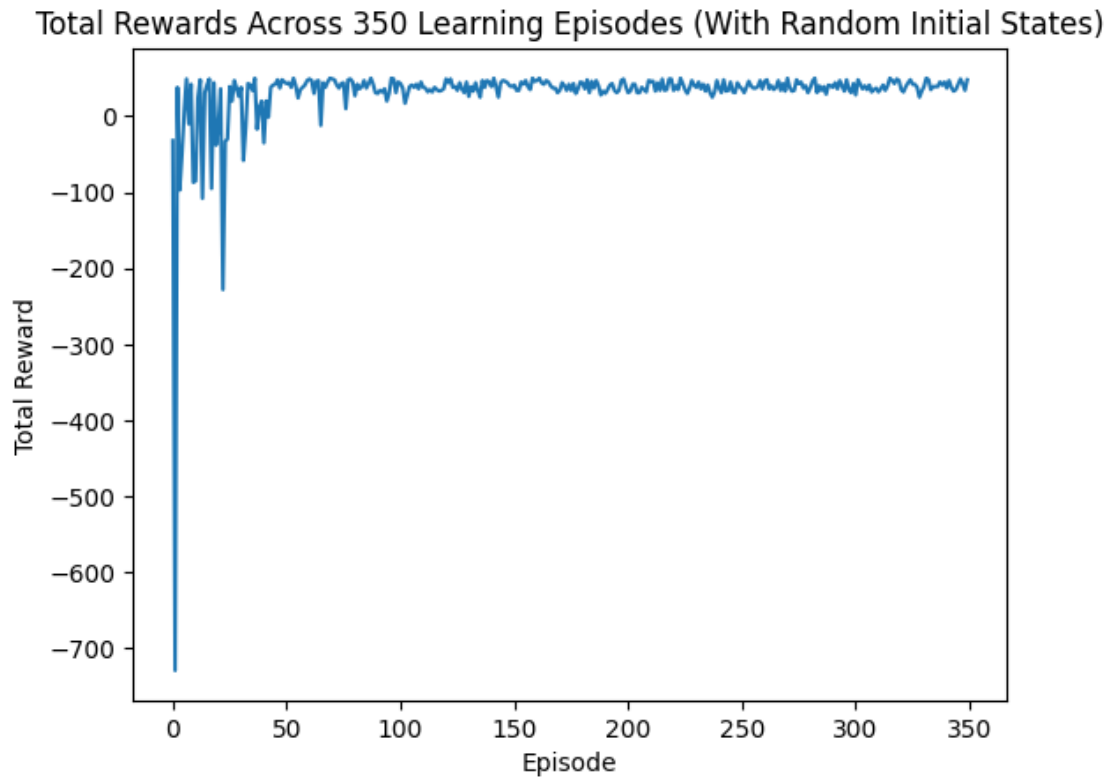
```
[ ]: import importlib
import qlearn as ql
importlib.reload(ql)
sim = ql.direct_updates(iterations=350, display=display, show_initial=False)
```

Q table (post learning):

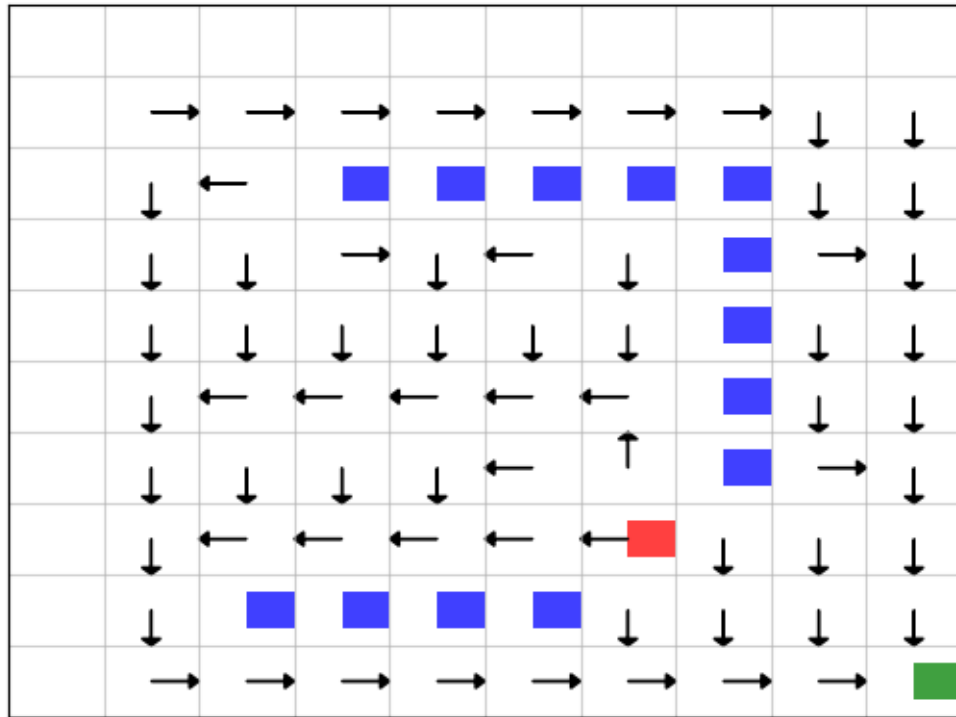
	s	a	r	q
0	(0, 0)	2.0	-1.0	-2.888844
1	(0, 0)	0.0	-1.0	-2.985318
2	(0, 0)	1.0	-1.0	-2.878611
3	(0, 0)	3.0	-1.0	-2.985552
4	(0, 1)	2.0	-1.0	-2.980773
..	...	...	...	...
319	(8, 7)	3.0	-1.0	20.624811
320	(8, 8)	2.0	50.0	0.000000
321	(8, 8)	0.0	-1.0	-0.013534

```
322 (8, 8) 1.0 50.0 0.000000
323 (8, 8) 3.0 -1.0 0.234438
```

```
[324 rows x 4 columns]
```



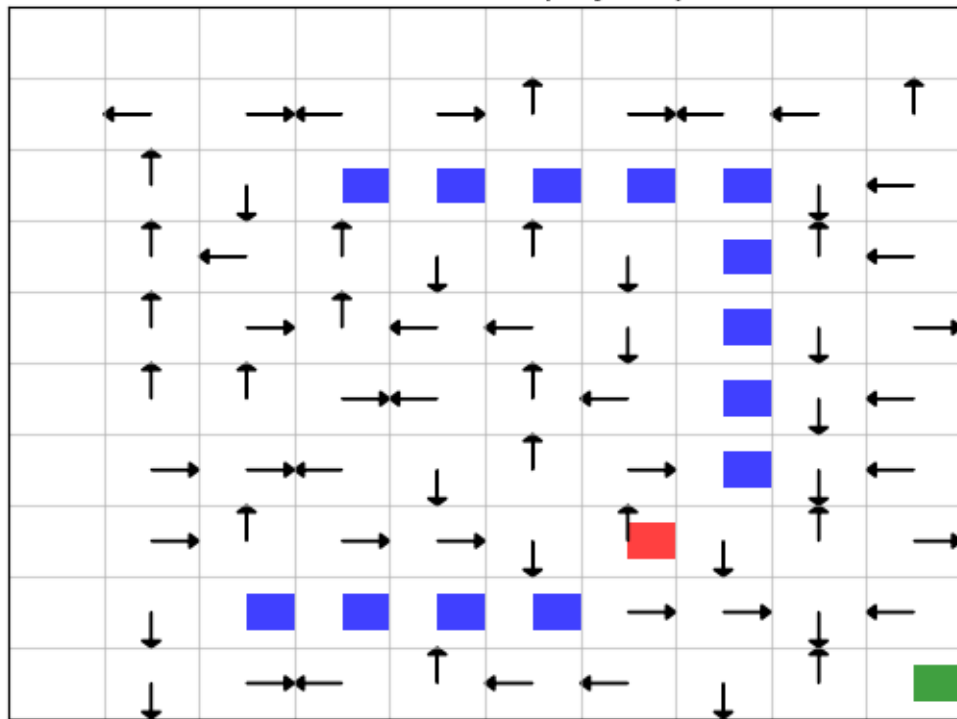
Q Table Post Learning (Showing Greedy Actions)



**Strategy 2b: Q-Learning (Replay Buffer)** Here we collect experiences by simulating episodes (without learning), and then sample from the stored experiences to perform Q-learning.

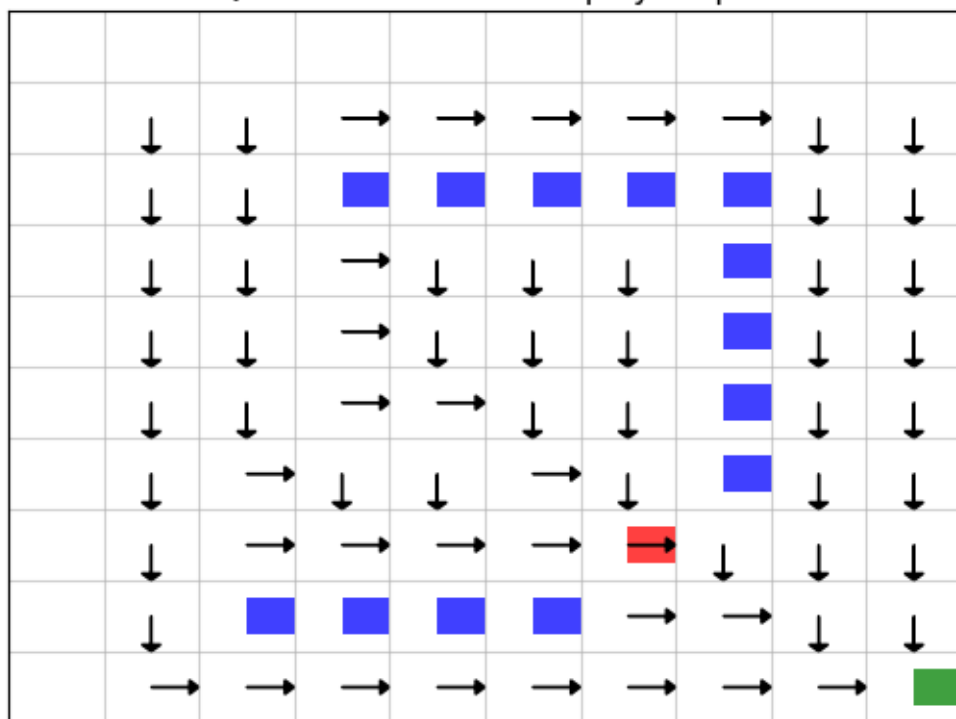
```
[ ]: import importlib
import qlearn as ql
importlib.reload(ql)
ql.strategy2b()
```

Q Table After 0 Replay Steps



collected 5000 experiences to sample from.

Q Table After 10000 Replay Steps



### 1.3.2 Discussion

#### Strategy 1 and 2a:

Comparing the graphs of rewards during learning in strategy 1 and 2a there doesn't seem to be a clear difference (at least in this implementation tested on this specific grid world example). Both strategies appear to converge after about 100 episodes.

Note that an “episode” refers to one play through the grid world game, starting at an initial random (empty) location, and stepping through the game (with a policy defined by the q-table), until the terminal state is reached (shown in green). The total rewards across the agents trajectory in this game is reported.

Potential future work would be to compare with more detail these two strategies, for example instead of plotting the total reward after each learning episode (which each start at random locations), we could sample  $n = 10$  (non-learning episodes) for the purposes of averaging their rewards for a more stable measure of the “fitness” of the q-table at a given moment in time.

#### Strategy 2b:

Strategy 2b is inherently harder to directly compare to the others as it doesn't train with sequential episodes, but rather with individual sampled game steps from a buffer of initial (non-learning) episode steps. Experience replay appeared to be not be as efficient as the other strategies (at least in this implementation tested on this specific grid world example).

This strategy could be improved by interleaving experience generation, and training. As the initial experience buffer occurs on a completely random qtable, it could be valuable to gradually replace the buffer later with experiences on a partially trained qtable as the buffer could contain more relevant state actions pairs for training.

Additionally, as the grid world map only has one terminal location (with a positive reward), its very important to the training initially that states are frequently sampled next to this location. As the q-table initially has random q values,  $(s, a)$  pairs near the terminal spot need to be updated to have relevant q values before this key information can accurately percolate to the rest  $(s, a)$  pairs of the q-table.