Gabor Mihucz

2239495m

Artificial Intelligence Assessed Exercise

Introduction

The task of this exercise was to design, implement, evaluate and document three virtual agents which are (potentially) able to reach a goal in a custom Open AI Gym environment derived from FrozenLake-v0[i].[ii]

The motivation for this exercise is to evaluate how a Q-learning agent learns and compares to a completely random/senseless agent and an optimal agent.

PEAS Analysis

The **performance measure** is the same for all three agents: to reach the goal without falling into a hole. We can measure this by counting the number of times the agent was able to reach the goal out of a number of trials/episodes, or by comparing the sum of rewards.

The **environment** is an 8x8 grid world representing 64 states where the agent can move to. The starting point and all other states labelled 'F' gain no rewards. The terminal states are labelled 'H', which incurs a negative reward, while 'G' gains a positive reward of 1. The environment is *deterministic*, for the simple agent, and *stochastic* (i.e. there is a certain probability that the outcome of an action in a given state will not be the intended outcome) for the random and RL-agent.

The **actuators** are the same for each agent: the step() function of the LochLomond instance, which triggers the agent to move to left, right, up or down.

The random agent has no **sensors,** knows nothing of its environment, it is just taking random actions. The simple agent is an oracle, it is fully aware of its environment, knows exactly where the holes are, and knows the distance between each state and the goal state (but still needs to search a way to get to the goal state.) The RL agent only knows what actions it can take in each state but has no prior knowledge about the state-space. It iteratively builds a Q-table, a table of state-action pairs, based on their perceived utility (how good they are for the purpose of reaching the goal), which in a way serves as a sensor.

Task Environment Characterisation

Type of Agent	Observable	Agents	Deterministic	Episodic	Static	Discrete
Random	Unobservable (no sensors)	Single	Stochastic	Sequential	Static	Discrete
Simple	Fully	Single	Deterministic	Sequential	Static	Discrete
RL	Partially	Single	Stochastic	Sequential	Static	Discrete

Method/design

Random Agent

The random agent is an **uninformed search** agent, it has no knowledge about the state space, all it can do is take random actions and distinguish between terminal and non-terminal states.

Simple Agent

I implemented an **A* search** approach, which is an informed search agent as it has complete knowledge about the state space. “The A* search is a form of best-first search. It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:

$$f(n) = g(n)+h(n)$$

Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have

$$f(n) = \text{estimated cost of the cheapest solution through } n$$

[...] provided that the heuristic function h(n) is consistent, A* search is both complete and optimal.”[iii] Therefore, an A* search agent can be used an optimal baseline, since it always finds the goal (and the shortest path to the goal as well).

Reinforcement Learning Agent

I implemented a **Q-learning** agent. In Q-learning, “Q(s,a) denotes the value of doing action *a* in state *s*. Q-values are directly related to utility values as follows:

$$U(s) = \max_a (Q(s,a))”[iv]$$

In each iteration, the algorithm updates Q(s,a) with the following equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma(\max_{a''}Q(s',a'')) - Q(s,a))$$

Where α is the learning rate, γ is the discount factor, and R(s) is the reward received for state s.

The agent iteratively builds a table representing the approximation of the utility of each state-action pair. A state-action pair, that would land the agent close to a hole will have low utility, while a state-action pair that would land the agent closer to the goal will have higher utility.

However, “repeated experiments show that the greedy agent very seldom converges to the optimal policy for this environment and sometimes converges to really horrendous policies. [..] An agent therefore must make a trade-off between exploitation to maximize its reward—as reflected in its current utility estimates—and exploration to maximize its long-term well-being.”[v]

Implementation

Random Agent

In each episode the environment is reset, and in each iteration within an episode, the agent takes a random action (chooses randomly between left, right, up or down). If the agent falls in a hole or reaches a goal, the episode is over, and we start a new episode.

Simple Agent

The agent knows the state space completely, hence the provided parser removed hole states ensuring that the agent would never end up in a terminal state with negative reward. I used the parsed values to create a GraphProblem instance to run the A* search implementation of the AIMA toolbox[vi]. By default, the GraphProblem's h() function returns the "straight-line distance from a node's state to goal"[vii], thus, I could utilise the *astar_search* function with standard settings. The *a_star_search* function is an extension of the *best_first_graph_search*. In the latter, I inserted a new variable action_list, which consists of the sequence of optimal actions based on the sequence of locations the search identified as optimal path for the given problem. I use action_list to demonstrate the Simple Agent's performance in the LochLomond environment.

Reinforcement Learning Agent

I initialised a Q-table as a (state space) x (action space) array of all 0s. In each iteration thereafter, the agent either follows the policy decided by the Q-table or takes a random action based on whether a randomly generated number is greater than a threshold value (epsilon).

Taking that action results in a new state and reward, with which the Q-table is updated as follows:

```
Q[s, a] = Q[s,a] + alpha * (reward + gamma*np.max(Q[new_state,:]) - Q[s,a])
```

After some experimentation, I found that the following parameters gave me reasonably good results: alpha = 0.1, gamma = 0.99, reward_hole = -0.06

I set epsilon to 1 for the first episode (no policy, just take random actions), then to 0.015 after each episode to follow the policy most of the time. Before each 100th episode, I set epsilon to 1-(episode_number/max_episodes) in order to ensure an occasional higher yet decreasing chance for exploration. This is an additional measure to avoid being stuck in a bad policy.

Experiments/Evaluation

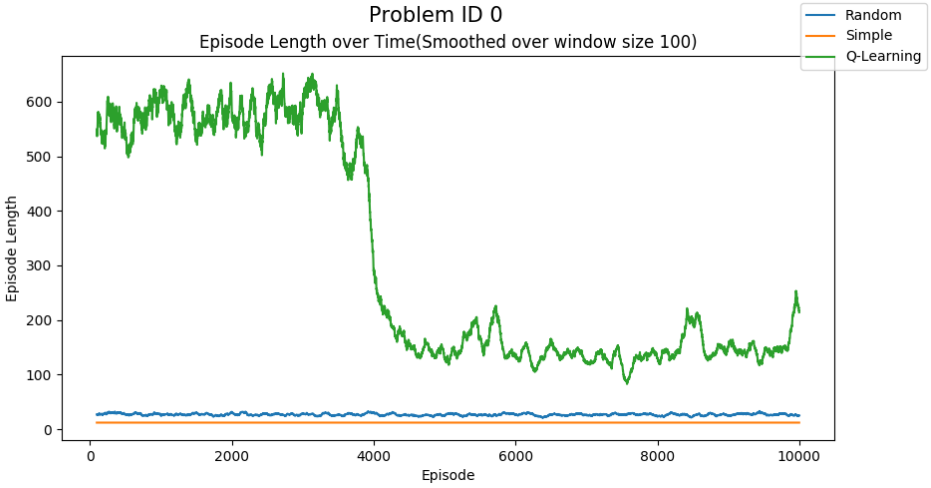
I created three graphs for each Problem ID, plotting:

- The episode length over episodes (to see how quickly the agent reached a terminal state)
- The episode reward over episodes
- The number of episodes executed per time step

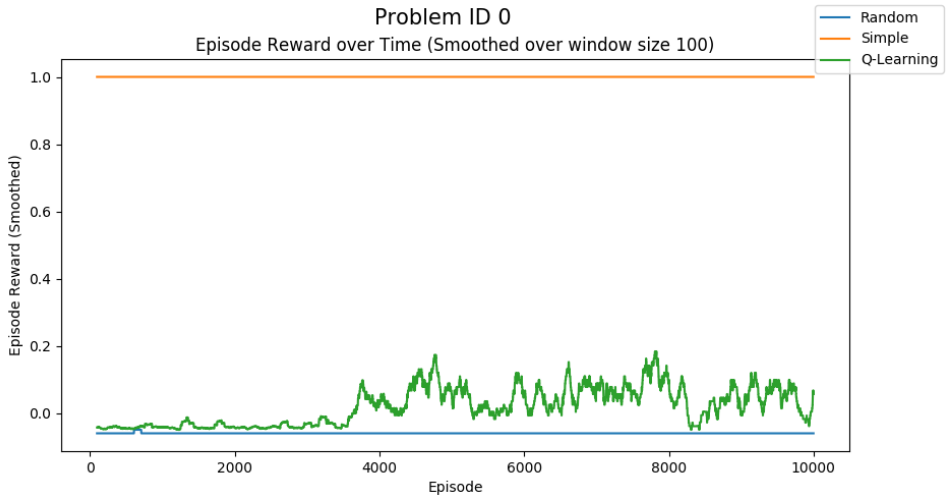
On these graphs, the episode length of the random agent and of the simple agent are usually low (for opposite reasons). The simple agent gets a steady 1 as a reward, as it knows the correct path, while the reward of the random agent is usually below 0. The simple agent runs slightly faster than the random agent, but the Q-learning runs much slower than both.

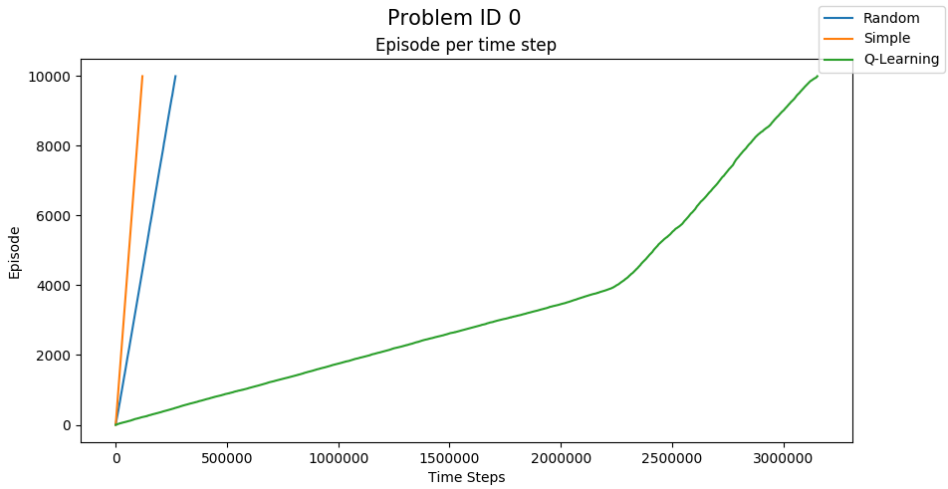
Graphs for Problem ID 0

The hardest problem of all was Problem ID 0. In the beginning, the Q-Learning agent needed 500-600 episodes to reach a terminal state: this changed rapidly after 4000 iterations, after which generally about 100-200 iterations were sufficient.



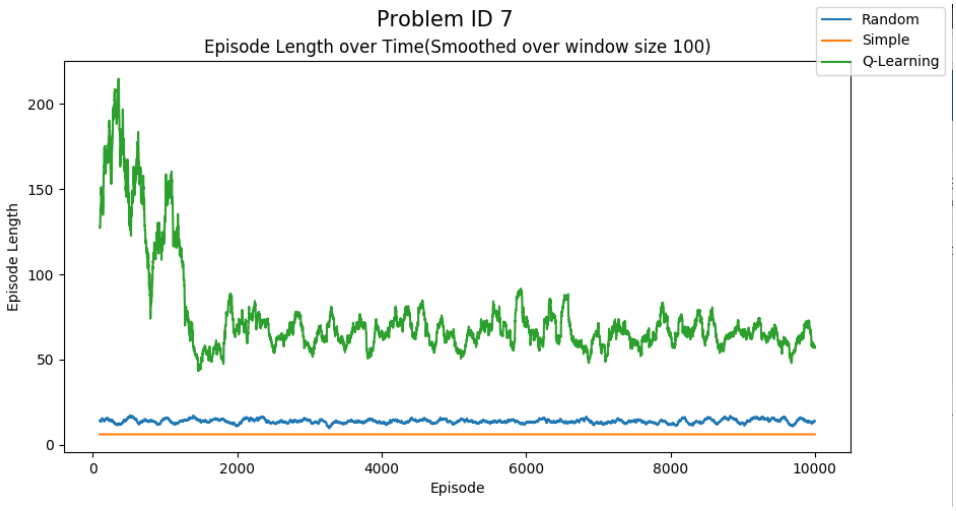
The graph below shows that approximately around 4000 iterations, the average reward value rose from below 0 to about 0.10 on average: this is when the agent found a route to the goal state. Since the environment is stochastic however, the agent is not always capable of reaching the goal, even when following the correct policy, hence the low values compared to the simple agent.



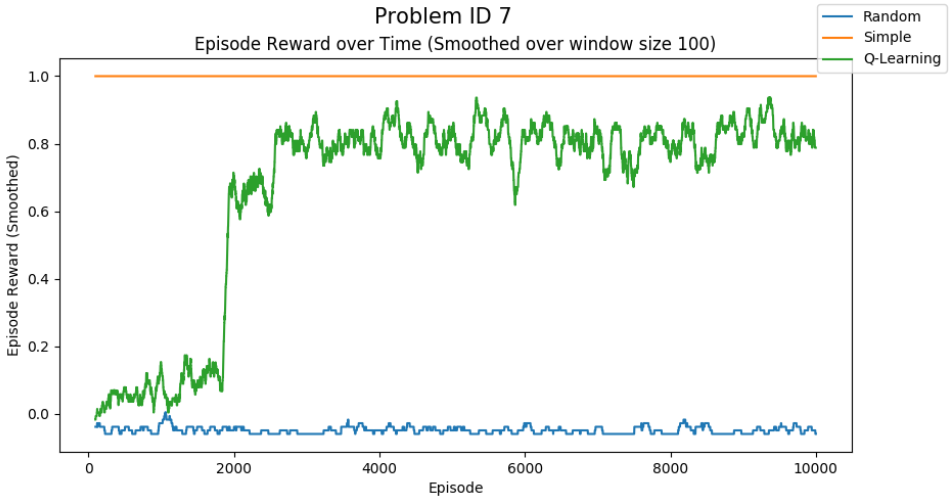


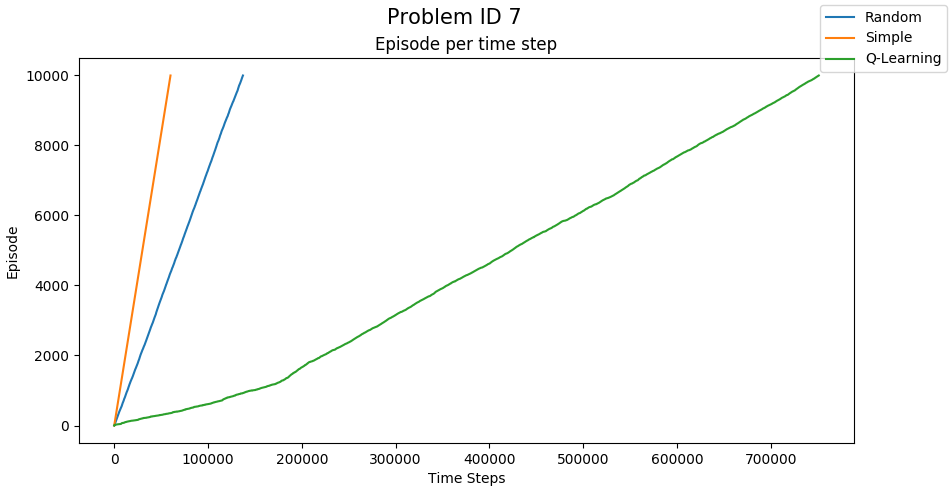
Graphs for Problem ID 7

The easiest problem was Problem ID 7. Each episode of the Q-Learning Agent lasts about 150 iterations for the first 1000 episodes, when this value drops to about half.



Similarly, the average reward increases sharply before 2000 iterations, from about 0.1 to 0.85. The agent found the goal state quickly and after that reached it again with high success rate over the following episodes and converges. This is because the goal state was close to the original state and there were no holes on the way.





The other graphs follow similar patterns. The sooner the agent can find the goal state, the higher success rate it can achieve in the environment.

Average number of times the agent found the goal in 10000 iterations after training

Agent\Problem ID	0	1	2	3	4	5	6	7
Random	1.6	1	39	6.3	6	10.6	0.6	112
Simple	10000	10000	10000	10000	10000	10000	10000	10000
Q-Learning	1570	2182	8292	8201	8205	8486	7328.6	9079.3

Discussion and conclusion

As the graphs above indicate, the Q-Learning Agent's performance is much better than the random agent: after training it can achieve 70-90% success rate in problems [2-7], and 15-20% on problem IDs 0-1.

It never gets as good as the optimal solution, but this is to be expected, since the optimal baseline environment is deterministic, and the agent has full knowledge of the state space.

Q-Learning therefore demonstrates an impressive ability to learn the optimal path in an unknown, stochastic environment. However, training a 64x4 table with 10000 episodes already takes a significant amount of time, therefore larger state and action spaces require function approximation methods such as radial basis functions or neural networks.

Appendix

Setup:

In order to run the scripts, please create a virtual environment and install the dependencies using the following command:

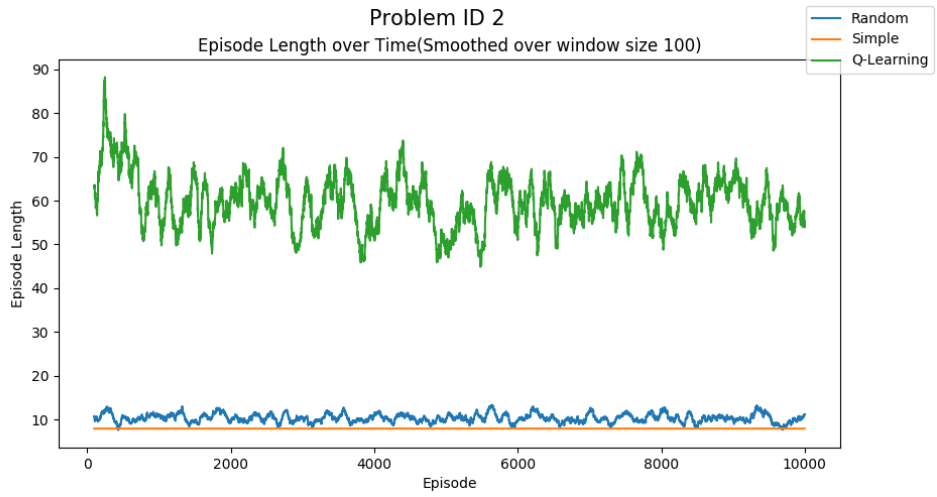
```
pip install -r dependencies.txt
```

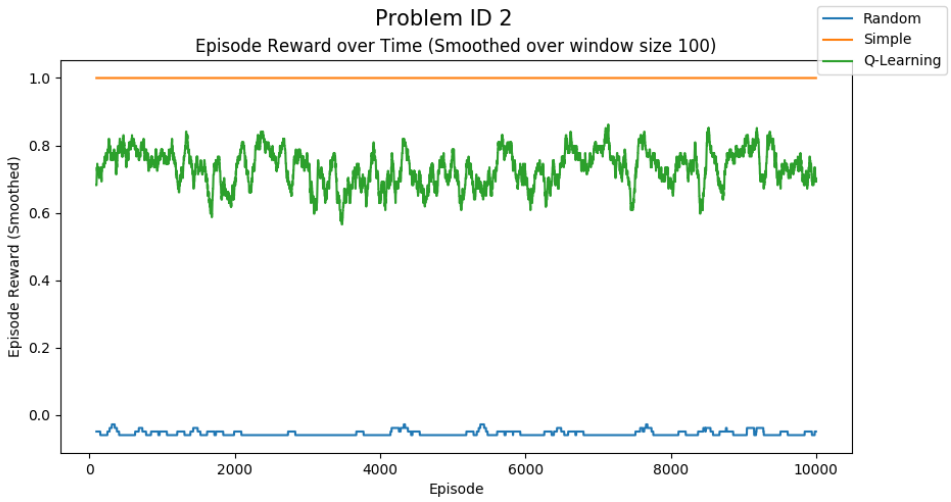
Run:

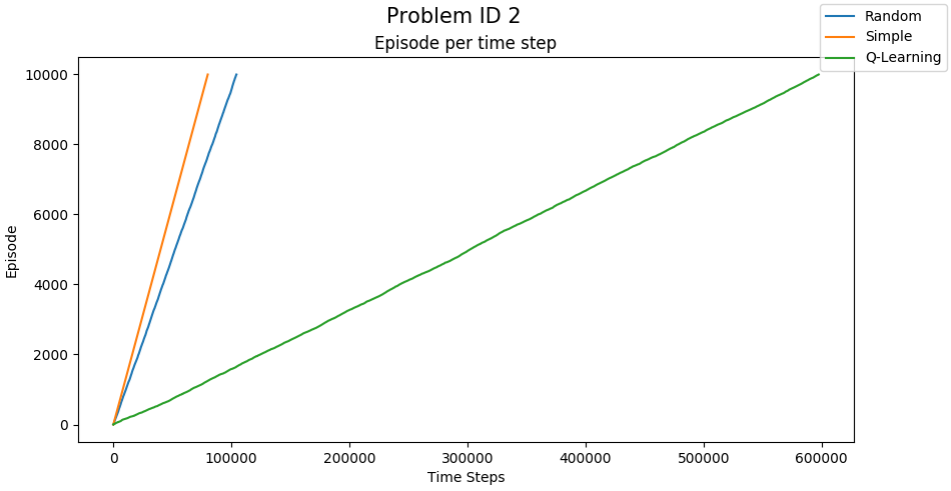
Each run_xxx.py file expects a problem_id ([0-7]) as input, including run_eval.py.

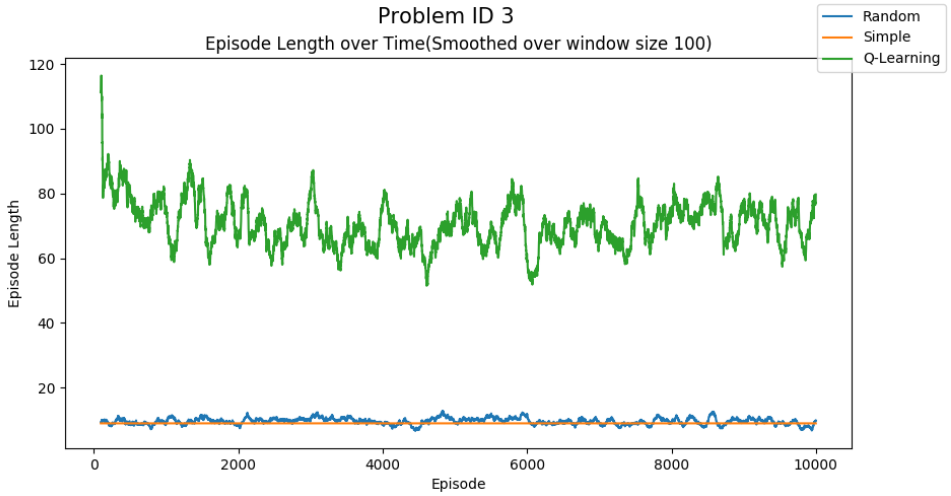
Graphs

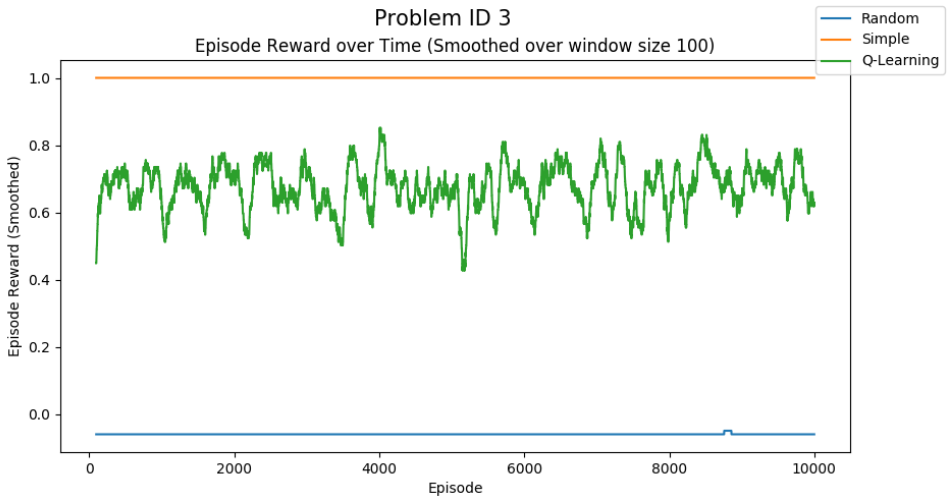
Problems IDs 2, 3 and 5 converge very soon after the start.

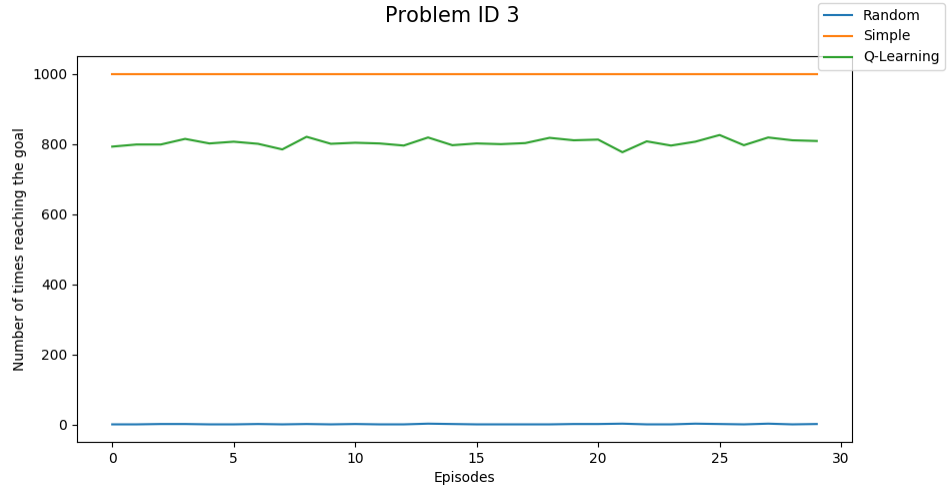


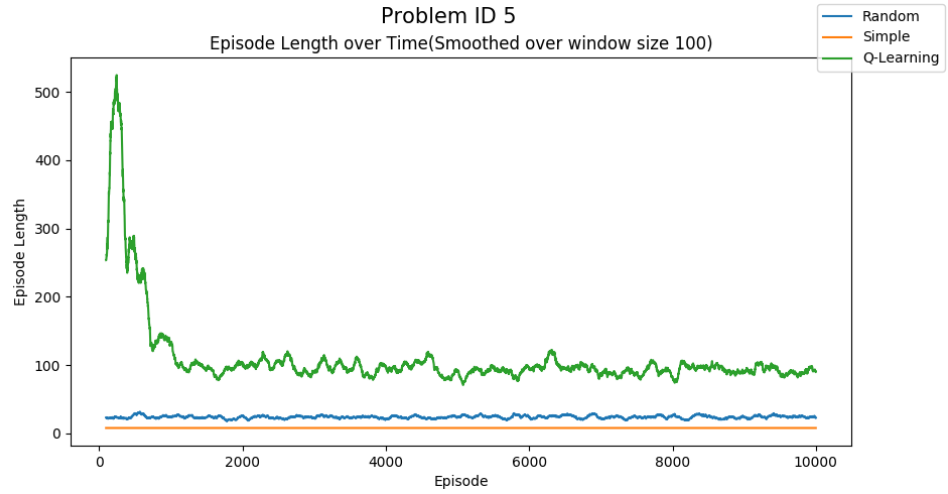

Problem ID 2
Episode Reward over Time (Smoothed over window size 100)
Random
Simple
Q-Learning
Episode Reward (Smoothed)
1.0
0.8
0.6
0.4
0.2
0.0
0
2000
4000
6000
8000
10000
Episode

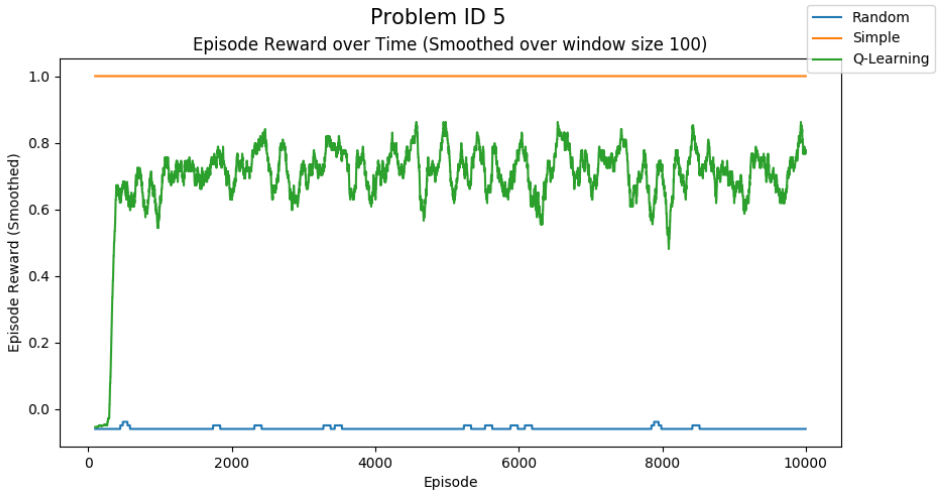

Problem ID 2
Episode per time step
Random
Simple
Q-Learning
Episode
10000
8000
6000
4000
2000
0
0
100000
200000
300000
400000
500000
600000
Time Steps


Problem ID 3
Episode Length over Time(Smoothed over window size 100)
Random
Simple
Q-Learning
Episode Length
120
100
80
60
40
20
0
2000
4000
6000
8000
10000
Episode


Problem ID 3
Episode Reward over Time (Smoothed over window size 100)
Random
Simple
Q-Learning
Episode Reward (Smoothed)
1.0
0.8
0.6
0.4
0.2
0.0
0
2000
4000
6000
8000
10000
Episode


Problem ID 3
Random
Simple
Q-Learning
Number of times reaching the goal
1000
800
600
400
200
0
0
5
10
15
20
25
30
Episodes


Problem ID 5
Episode Length over Time(Smoothed over window size 100)
Random
Simple
Q-Learning
Episode Length
500
400
300
200
100
0
0
2000
4000
6000
8000
10000
Episode



[i] https://gym.openai.com/envs/FrozenLake-v0/

[ii] https://moodle.gla.ac.uk/pluginfile.php/1873981/mod_resource/content/4/ai_h_2018-2019-ae_v2019a.pdf

[iii] Russell, S. J., Norvig, P., & Davis, E. (2010). *Artificial intelligence: a modern approach.* 3rd ed. Upper Saddle River, NJ: Prentice Hall., p. 93

[iv] Russell, S. J., Norvig, P., & Davis, E. (2010). *Artificial intelligence: a modern approach.* 3rd ed. Upper Saddle River, NJ: Prentice Hall., p. 843

[v] Russell, S. J., Norvig, P., & Davis, E. (2010). *Artificial intelligence: a modern approach.* 3rd ed. Upper Saddle River, NJ: Prentice Hall., p. 839-840

[vi] https://github.com/aimacode/aima-python/blob/master/search.py

[vii] https://github.com/aimacode/aima-python/blob/master/search.py