Intelligent Elevator Saga

Eran Nussinovitch
302186408
eran.nussinovitc@mail.huji.ac.il

Gregory Pasternak
327148417
grygoriy.pasternak@mail.huji.ac.il

Asaph Shamir
200122588
asaph.shamir@mail.huji.ac.il

I. Introduction

Elevators are one of the most used means of transport for both people and goods. Elevators were initially operated manually by a dedicated person which was always present in the elevator. Today, elevators are operated by a controller, which given the pressed buttons in the elevator and the different floors uses some algorithm to decide where the elevator will go. A particularly bad example of such mechanism can be found in the Rothberg B building.

The automation of elevators presents a problem. Many users (passengers) are requesting use of the same resource (elevator), and all are seeking to minimize their wait time. A good controller will give an overall low wait times. In this project we use different approaches, Reinforcement Learning (RL) and Adversarial Search (AS), to model the controller. We evaluate the performance of each approach and compare them.

Reinforcement Learning models the controller as an agent operating within a world. Agent types compared are Q learning, deep Q learning and Multi Agent. Adversarial Search models the problem as a two player game, the elevator agent vs. the world. The agent types compared in AS are Reflex, Alpha-Beta, and Expectimax.

The platform on which we train and test our agent is the ElevatorSaga[1] challenge. It is a coding game, which simulates a building with an arbitrary number of floors and elevators. The player's goal is to code the algorithm that decides the elevators actions. Instead of writing explicit code, we train different agents to control the elevators.

Finally, we will also compare our agents to some explicit algorithms: random agent, Shabat elevator, and a hand crafted solution to the ElevatorSaga game, taken from ElevatorSaga wiki.

II. Problem Description

1. *Problem Definition*

An ElevatorSaga challenge simulates a building with floors and elevators (Figure 1). Users spawn on different floors, each user has a destination floor he/ she wishes to go to. Each floor has an up button and a down button, users can press the up/ down button signaling that their destination floor is above/ below the current floor (the bottom floor and the top floor have only an up/ down button respectively). The elevators may move to any floor (including the floor they're on). Upon reaching a floor the elevator stops, and all the users in the elevator whose destination floor is the current floor will exit. Any user waiting at that floor will enter the elevator if there is room.

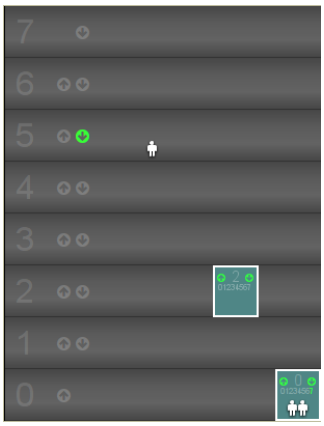


Figure 1: Example of an ElevatorSaga challenge with 8 floors and 2 elevators. The down button on the 5th floor is pressed. The 7th floor button in the second elevator is pressed.

We begin the formalization of the elevator saga game by defining:

$$|FLOORS| = N \;\; |ELEVATORS| = M$$

In each run of the simulator there is a spawn factor p_s governing the generation of new users. In every time step, a user is generated with probability p_s. The probability for creating a user in a certain floor is:

$$p(floor) = \begin{cases} 0.5 + \frac{0.5}{N}, & floor = 0 \\ \frac{0.5}{N}, & floor > 0 \end{cases}$$

Meaning most users spawn at floor 0.
If a user spawns at floor 0 its destination floor is randomly sampled out of $\{1, \dots, N-1\}$. Otherwise 90% of the time the user destination floor is 0, and 10% of the time it is randomly selected out of the other floors.
Each elevator has a maximum capacity. In our scenarios all elevators capacities were set to 4. An elevator with 4 users inside is full, and floor users cannot enter the elevator until users inside the elevator exit.

2. *Complexity Analysis*

State space:
We use a realistic approach, in which our agent only sees information available for a controller in real life. That is, it knows which buttons are pressed in each floor and in the elevators, and it knows the elevator locations. The agent does not know how many users are waiting on each floor and how many users are inside each elevator. However, the agent does

[1] https://play.elevatorsaga.com/

know if an elevator is full, simulating an overload sensor which lets the elevator know when it is too heavy to move. Such sensors exist in elevators today.
For each floor, excluding the top, there is an up button which can be pressed or not pressed, giving 2^{N-1} combinations. Each floor besides the bottom has a down button which also gives 2^{N-1} combination, in total, 2^{2N-2} cases.
Within every elevator, each floor button could be pressed or not, beside the floor the elevator is currently in, giving $2^{(N-1)}$ cases for each elevator. In total: $(2^{(N-1)})^M = 2^{M(N-1)}$.
Each elevator can be in any given floor so we have N^M combinations for locations. In addition, the load factor of each elevator is represented as a single Boolean giving in total 2^M cases.
All of the above gives us a state space of size:

$$2^{(2+M)(N-1)} \cdot N^M$$

Action Space:
Each elevator can go to any floor, including to stay on the floor it is on right now, resulting in N^M possible actions.

State-Action Space: Considering all the possible states and actions, the full space size is:

$$2^{(M+2)(N-1)} \cdot N^{2M}$$

The size of the state-action space is exponential in both the number of floors and the number of elevators, and is the main obstacle for our agents. For example, Rothberg building B front side has 2 elevators, each of them can reach 7 floors (-2, -1 and 1 through 5) giving a state-action space of size $\approx 2^{35}$

3. Evaluation & Metrics

We define the wait time of a user as the time passed since the user spawned until the user exits an elevator at the destination floor. Two metrics are used in our evaluation:

- Average user wait time
- Maximal user wait time

Average wait time reflects overall model performance, while maximal wait time shows fairness of the model. If a model shows significantly higher maximal wait times, it means there were users that experience starvation, which should be avoided.

4. Scenarios

We tested our agents on three scenarios (buildings):

- 3 floors, 1 elevator
- 7 floors, 1 elevator
- 7 floors, 2 elevators (Rothberg building B)

III. Solution Approaches

1. Reinforcement Learning

Due to the Markovian nature of the world (past states are insignificant when deciding on an action), Value Estimation is the most natural go-to approach. However, our world is stochastic, and the agent doesn't know the probability function for state-action-state (we don't know when and where users will be generated). This makes RL, and particularly Q-learning, a model-free agent, most suitable for this problem.
Even though Q-learning is model free, it still suffers from the huge state-action space in this problem. To lessen this we unified the up and down buttons in each floor to a single "call elevator" button reducing the number of state by a factor of 2^{N-2}.
The space size is particularly problematic for the scenario with 7 floors and 2 elevators. The state-action space is so large that it is computationally impossible to use regular Q-learning. We therefore use two variations of Q-learning for this scenario: Deep-Q-learning using a neural network and Multi Agent Q learning (see IV for further details).

Lastly, an RL algorithm requires feedback from the environment. Since the ElevatorSaga game does not have rewards, we design a reward system that supports the training process of our algorithms.

The reward for a state-action-nextState triplet is made up of several parts:
Let n_i^{exit} be the number of users exiting elevator i after it moved, then the reward for exiting users is:

$$r_{exit} = \sum_{i=1}^{M} n_i^{exit} \cdot 2$$

The movement penalty for each elevator is given by:

$$p_i^{move} = \Big(\big((a_i - l_i) - 1\big) \cdot 0.3 + 0.4\Big)$$

Where a_i is the floor elevator i reached after moving, and l_i is the floor it left. The total movement penalty is:

$$p_{move} = \sum_{i=1}^{M} p_i^{move} \cdot -\frac{1}{2}$$

The users' penalty is given by:

$$p_{users} = -1 \cdot \left[\sum_{i=1}^{M} n_i^{stay} + \min\left(M \cdot C, \sum_{j=1}^{N} n_j^{wait}\right)\right]$$

Where n_i^{stay} is the number of users still in elevator i after it moved, C is the capacity of the elevators and n_j^{wait} is the number of users waiting in floor j.
Additionally, we have special penalties. Let F be the set of floors with "call" buttons pressed. Let b_i be the set of buttons pressed in elevator i (the floors requested by users in that elevator). Let n_i^{in} be the number of users in elevator i before it moved. Then we have:

$$p_i^{full} = \begin{cases} -200, & n_i^{in} = C \wedge a_i \notin b_i \\ 0, & else \end{cases}$$

$$p_i^{empty} = \begin{cases} -200, & n_i^{in} = 0 \wedge a_i \notin F \wedge F \neq \emptyset \\ 0, & else \end{cases}$$

$$p_i^{bad} = \begin{cases} -100, & 0 < n_i^{in} < C \wedge a_i \notin F \cup b_i \\ 0, & else \end{cases}$$

The total reward is given by:

$$R = r_{exit} + p_{move} + p_{users} + \sum_{i=1}^{M} p_i^{full} + p_i^{empty} + p_i^{bad}$$

The design of the reward system plays a major part in determining the performance of our agents, and is further discussed in VI.1.

2. Adversarial Search

We note that this problem could be modeled as a 2-player game where the first (max) player is the elevator agent and the second player is the world. The world player always chooses an action by some stochastic process.
Like Q-Learning, adversarial search agents suffer greatly from the state-action space size which translates directly to a very large branching factor. This limits greatly the depth to which an AS agent can go during runtime. The elevator should operate in real-time, i.e. the agent's computation time to decide on the next action should seem instantaneous to the user.

For AS agents, the utility of a state is computed using simple evaluation function (adversarialSearch.py:340):

$$u = \#\ users\ outside\ the\ elevator \cdot (-1.1) + \#\ users\ in\ the\ elevator \cdot (-1)$$

Note: when a passenger arrives to its destination, he is immediately removed from the world, and stops being counted in consecutive rewards.

IV. Implementation Details

Reinforcement Learning Agents

All training was performed in episodes. Each episode started from an empty world and runs for 300 iterations (during training iterations are game steps, a single game step comprises of an action taken by the agent, and the world's reaction to that action). In each episode the spawn factor was randomly chosen from the segment [0.4, 0.65]. The number of training episodes varied, but was no less than 30,000 and no more than 100,000[2]. We used another parameter N_{exp} to set the number of episodes during which exploration occurs. The exploration is linearly decreased during the exploration episodes from its initial value until it reaches 0 after N_{exp} episodes. After the exploration phase, the exploration rate stays 0 for the rest of the training. Different exploration (ϵ), discount (γ), and learning (α) rates was used. Best results were achieved by setting initial values $\alpha = 0.5$, $\epsilon = 0.5$, $\gamma = 0.9$. We used exponential decrease for the learning rate and linear decrease for the exploration rate. The discount rate is constant.

1. Q learing Agent

We used our implementation from exercise 4.

2. Q learing Multi-Agent

Given a scenario with 2 elevators, each elevator is a separate Q-learning agent. The two agents share the Q table. During training both update the same Q table (this also "halves" the learning time required to achieve good results). During testing each agent operates independently from the other. To allow some "cooperation" between the agents, the state was expanded to include the location of the other elevator.

3. Deep Q-learing Agent

This variant replaces the discrete Q table with a parameterized Q value estimation function. Specifically, the estimation function used is modeled by a neural network.
The network receives a pair of state action. The state is represented as a concatenation of binary vectors for each of the state properties (elevator location, buttons pressed, requested floors, etc...). The action is represented as a one hot vector of size # legal actions. The output of the network is a scalar representing the Q value of the input state action pair.
For the case of 3 floors 1 elevator 2 fully connected (FC) hidden layers of size 256 was used. For 7 floors 1 elevators 2 FC hidden layers of size 512 was used. For 7 floors 2 elevators 3 FC hidden layers of size 512 was used. All the activations were Leaky Relu. A few different architectures were considered for each case, all gave similar results, the reported architectures gave the best performance.
The networks were trained using ADAM optimizer with a learning rate of 5e-4. The exact deep Q learning algorithm can be found here[3].

Adversarial Search Agents

4. Reflex

This is a greedy agent, which in each state selects the most rewarding action. The rewards to select from are computed as if the elevator would make its action, and the world would react (e.g. new passengers arrive, existing passengers exit if possible) (i.e. it looks one step ahead)

5. Alpha-beta pruning

Standard implementation of α-β pruning algorithm. We choose the depth such that the algorithm will yield results in real time and therefore only used depth 2 for 7 floors and depth 3 for 3 floors.

6. Expectimax

Standard implementation of Expectimax algorithm, with probability distribution of the chance player as it was defined by the world. Since branching factor is $M \cdot (N^2 + N)$, we only use this algorithm with depth 2 for 3 floors 1 elevator scenario.

Additional Agents

7. Random Agent

Chooses uniformly a random floor to go to on each action. In case of multiple elevators, they are controlled by a single controller (the agent), which chooses random floor for each of them independently.

8. Shabbat Elevator

This agent traverses all N floors from 0 to $N-1$ in a cycle, stopping on each one. After reaching the Nth floor, the elevator goes straight to the 0 floor. In case of multiple elevators,

[2] 40,000 training episodes took ~20 minutes for Q learning and ~60 minutes for deep Q learning.

[3] https://moodle2.cs.huji.ac.il/nu16/pluginfile.php/319819/mod_resource/content/2/RL_APML_2017.pdf (slide 23)

their starting floors are distributed uniformly between 0 and $N-1$.

V. Results

Testing was performed by running the ElevatorSaga game for 10,000 time steps (unlike game steps used in training), with a fixed spawn rate of 0.4. At the end of the 10,000 steps the average and maximal wait times were recorded.

Comparison of average wait times and maximal wait times of different agents for 3 floors 1 elevator (3F-1E) is presented in Figure 2, comparison for 7 floors 1 elevator (7F-1E) in Figure 3, and for 7 floors 2 elevators (7F-2E) in Figure 4. In figures 3 and 4 the random agent is omitted for presentation purposes, as its wait times are 2 orders of magnitude larger than the other agents. Table 1 summarizes all the results including the random agent.

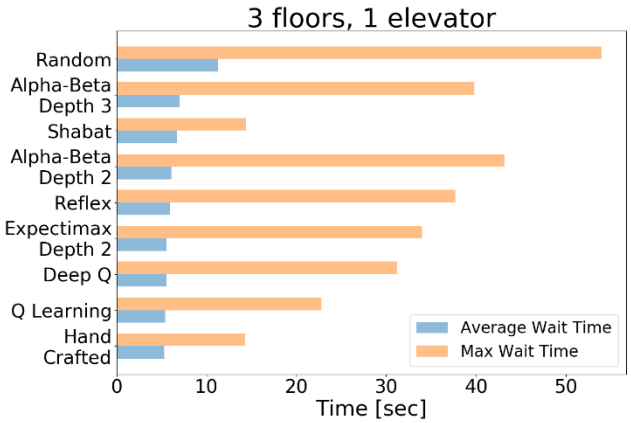


Figure 2: Comparison of average and maximal wait times for the 3 floors, 1 elevator scenario.

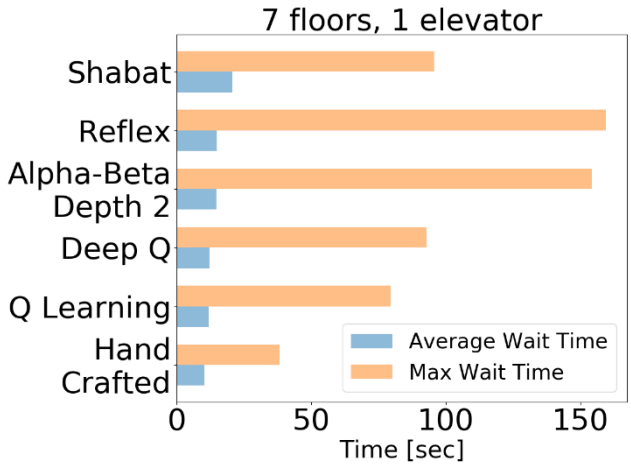


Figure 3: Comparison of average and maximal wait times for the 7 floors, 1 elevator scenario.

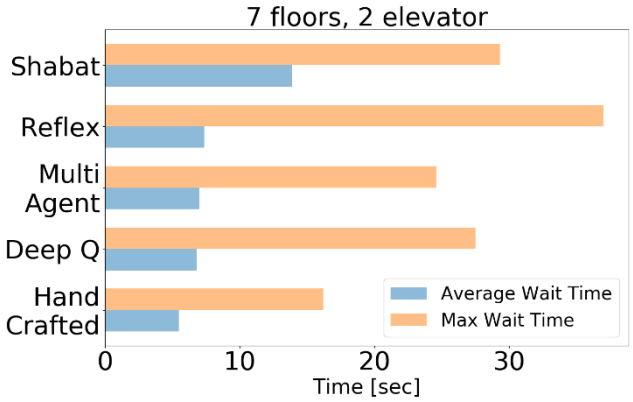


Figure 4: Comparison of average and maximal wait times for the 7 floors, 2 elevator scenario.

The best average and maximal wait times, over all scenarios, are achieved by the hand crafted algorithm from ElevatorSaga wiki. The second best performing agents were the RL agents, followed by AS agents. Differences in average wait times between hand crafted, RL and AS are small, and all performed much better than random, which in 7 floors scenarios had average wait times up to 2 orders of magnitude larger.

The poor performance of the Adversarial agents is caused by the real-time constraint. Due to the large branching factor, increasing the depth parameter for Expectimax above 2 and for Alpha-Beta above 3 caused noticeable delays in the elevator movement, and therefore was bounded. This decreases the performance of the algorithms.

Interestingly, Alpha-Beta agent with depth 3 underperformed Alpha-Beta with depth 2. This, and the fact that Expectimax outperformed both Alpha-Beta agents seems reasonable when considering that the game's min player is random 100% of the time, which contradicts the basic assumption of the Alpha-Beta algorithm (playing against an optimal player).

The dominance of the hand crafted algorithm over RL agents can be attributed to three main reasons. First, the hand crafted algorithm uses information unavailable to the RL agents, as it sees the exact number of users both in the elevators and on the floors (the hand crafted algorithm was taken as is). Second, as will be discussed later, the reward system has a major influence over the RL agents' performance. It is possible that with further tinkering of the rewards the performance of RL agents would increase. This is especially noticeable when looking at the maximal waiting times. Our reward system has no reward that is directly connected to wait times. Using our current reward system an RL agent is implicitly trying to maximize throughput[4], which is directly related to average wait times, but not to maximal wait times. Adding a negative reward based on wait times is not possible however, since such reward will depend on previous states and therefore will break the Markovian assumption in RL.

Lastly, during testing our Q-learning agent encounters states not seen before during training. This does not happen often, but every time it does the agent's best option is to randomly

[4] Throughput is the ratio of users transported to time.

choose an action. Even with varying spawn factors and high exploration rates, the agent is not able to visit all the edge cases during training and therefore performs poorly when it encounters these edge cases during testing. The hand crafted algorithm on the other hand, does not need to be trained and handles all possible states through explicit logic, which even if not optimal is better than random choice.

It is interesting to note that the Shabat agent has very low maximal wait time in the 3F-1E scenario. In this scenario the Shabat agent guarantees a maximum of 2 elevator movements for pickup of a user, and another 2 elevator movements for drop off. Additionally, the relatively low spawn rates combined with the Shabat algorithm and the fact that there are only 3 floors, ensures that the elevator is never full. This means that in the 3F-1E scenario a user will wait at most 4 elevator movements before reaching its destination floor. RL and AS agents on the other hand might choose to ignore a user waiting a long time in order to achieve better throughput, which leads to starvation and high maximal wait times.

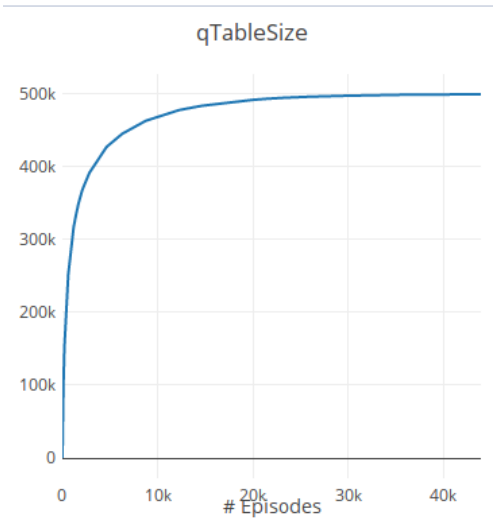
Once the number of floors is increased to 7, the Shabat agent maximal wait time increases beyond other agents. In 7 floors scenarios the Shabat agent might encounter a situation where the elevator is full and a user is forced to wait 2 or more cycles before it can board the elevator. Consequently, maximal wait times increase.

VI. Discusion

1. *Rewards effect*

The reward system for Q learning can change the performance of the agent dramatically. Initially the only reward was $\#\ users\ in\ the\ world \times -1$. However, agents trained with this reward made a lot of redundant actions, moving the elevator to floors were no user got on or off. To discourage this we added the movement penalty. This caused the agent not to move the elevator at all, because the weight of this penalty was too high. Later the weight of the movement penalty was set such that the agent should always gain from moving the elevator to let a user exit. Still, this meant that the agent is "allowed" to make unnecessary stops. Finally, we added the special penalties for empty and full elevators. An empty elevator should never move to a floor without users waiting, and a full elevator should always go to a destination floor of one of the users inside it. By penalizing "wrong" actions heavily we discourage the agent from making these bad moves.
This specific reward system can be considered as kind of implicit programming. Instead of programming the controller directly to address those cases, we penalize the agent and let it learn what to avoid.

2. *Training Analysis*



As mentioned above, during training the Q-learning agent encounters only a portion of all possible states. In the case of

Figure 5: # of state-action pairs seen by a Q-learning agent as a function of the number of episodes in the 7F-1E scenario.

7F-1E (using the unified "call elevator" button) the state action space size is approximately $1.6 \cdot 10^6$. However, looking at Figure 5, we can see that the actual number of state actions the agent sees tends to $5 \cdot 10^5$, less than half the possible number or state action pairs. This is the case despite high exploration rates during training. One reason for the discrepancy is that we count impossible states when calculating the state space size. A state where the elevator is on the 2nd floor, and the 2nd floor button in the elevator is pressed cannot happen in our simulator, however it is still counted. Another reason might be that the simulator requires longer episodes with higher spawn rates in order to see states with many users in the world, meaning these states have low probability to be seen in the state distribution of our simulator. Therefore, our agent doesn't encounter them.

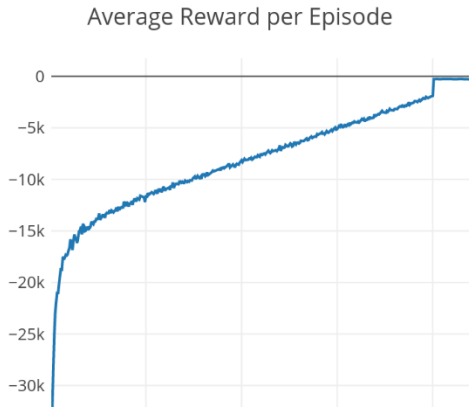


Figure 6: Average sum of rewards during training of Q learning agent for the 7F-1E scenario. $\#\ training\ episodes = 45{,}000$, $N_{exp} = 40{,}000$, $\epsilon = 0.5, \alpha = 0.5, \gamma = 0.9$

Figure 6 shows an increase in mean sum of episode rewards throughout training. At first the graph increases rapidly as the agent learns quickly many new state-action approximate values. It then enters a linear increase until $N_{exp} = 40{,}000$ episodes are done, and then jump sharply. This linear increase is governed by the linear decrease of the exploration factor. Starting with 0.5 exploration rate, the average sum of episode rewards is dominated by the random action the agent chooses. As the exploration rate decreases so does the "bad" random actions the agent chooses and the reward increase. We can see that even for very low exploration rates, the reward is dominated by the "bad" random actions which get heavily penalized by our reward system. The last sharp "step" in the graph happens when the exploration rate drops to zero, then the agent uses its learned policy only.

VII. CONCLUSIONS

We have presented a comparison of different approaches to implement an elevator controller. We have showed that RL is more suitable for this task than AS. RL demonstrates good performance in this task and probably could be improved further with more research.

As an experiment it was interesting to compare the performance of RL methods to other, and although RL gave good results better performance can be achieved by using a relatively simple algorithm. We therefore conclude that the controller algorithm in Rothberg building B needs to be replaced.

Table 1: Complete list of results for all agents and all scenarios. Best results are marked with bold font, second best results are highlighted in yellow

Agent	3F-1E		7F-1E		7F-2E	
	Average	*Max*	*Average*	*Max*	*Average*	*Max*
Random	11.28	54.0	1681.05	4670.0	64.52	239.0
Shabat	6.72	14.4	20.64	95.6	13.86	29.3
Reflex	5.93	37.7	14.9	159.3	7.37	37.0
α-β Pruning Depth 2	6.33	55.7	14.79	154.1	-	-
α-β Pruning Depth 3	7.03	40.8	15.8	-	-	-
Expectimax Depth 2	5.58	34.0	-	-	-	-
Multi Agent	-	-	-	-	7.00	24.6
Deep Q	5.56	31.2	12.19	92.2	6.84	27.5
Q Learning	5.42	22.8	11.86	79.5	-	-
Hand Crafted	**5.3**	**14.3**	**10.4**	**38.2**	**5.5**	**16.2**