ORML: Project 2

Rory Kolster, SNR: 2070539, ANR: 736326

November 2024

1 Q-learning with Manager Game

For all parts of this question regarding the Manager game, I switch the t variable to start from 0 and end at 48, as this was easier for indexing lists in python. Note, also, that in each code file, there are code flags which dictate which question or part the code runs. They can be found in line 390-396 for Q1 and 326-329 for Q2.

1.1 Some simple heuristics

I implemented three simple heuristics to play the Manager game.

1. The first simply did only project C since its profit dominated the others if $v <= 6$. This meant that we had profits of 0 50% of the time and for the rest it would be 500, 1000 or 1500. This is naive but it yields a high average profit of roughly 435.

2. The second algorithm tries to estimate v by doing project A once, and then based on the duration it chooses whether to fire or continue with project C. If the duration of project A was less than a value x, then it would do project C, and otherwise it would fire after completing the first project. Testing different values of x turned up that 5 gave the best results. Then also, once in the later stages of the game, it observes the time and whether it is too far to do project C or B, i.e. if $t > T - 14$ (14 is the minimum time needed to complete C) don't do C, or if $t > T - 7$ (7 is the minimum time for B) don't do B. This felt like a smarter idea, but yielded worse results with an average profit of 385. Though it did reduce the standard deviation from 500 to 400.

3. The third heuristic tries to use both the first and second heuristics by always doing C unless the profit yields 0 after the first attempt, but also switching jobs once it's later in the game. This gave slightly better results than the first heuristic, since it gains some more profit in the last stages by doing B or A. The heuristic had an average of 445, with a similar standard deviation to the first algorithm.

1.2 Reinforcement learning states, actions and rewards

For the state space of the Manager game, I had 4 states for each time t, being one start state, only accessed at $t = 0$, and the rest were for each project. In the start state, there are 4 actions: fire (index 0), do project A (index 1), do project B (index 2), and do project C (index 3). For all states for $t < 11$, the action 0 is to fire, and the rest are to move to do each project respectively. During the training, the agent doesn't pass through every time for a single run, it only passes through the time related to durations of the projects. For instance, if the duration of project C for the first time is 20, then we move from $[t = 0][state = 0]$ to $[t = 20][state = 3]$, and so on. As for the rewards, once a project is completed, the reward is the profit obtained for that project, or if terminating the run by firing, the reward is the profit from firing.

1.3 Q-learning algorithm and fine-tuning of hyperparameters

In one run of the Q-learning algorithm, I initialize the variables (v, durations) of the game. Then while the time limit has not been reached, we go through states updating the Q-tables according to the rewards, prior values and potential next-step values. I initialised the Q-table with 0 since every reward/profit was greater than or equal to 0.

After achieving a Q-learning algorithm that yielded decent outcomes, I began fine-tuning the hyperparameters, α, γ and ϵ. This problem is non-deterministic so setting α lower is intuitively preferred since we want to still remember past events. Furthermore, the problem is stationary as the probabilities do not change over time. This then indicates that α can be both decaying and constant. By experimenting with both constant and decaying learning rates,defined as the reciprocal of the number of times the state has been visited, I observed that a combination of using a constant value of 0.01 for the first 80% of the training and then a decaying rate for the rest was preferred.

As the agent had no problem converging, a γ of less than one is not intuitive. Nevertheless, I tried some values and they did not have any consistent effect ($\gamma = 0.8$ was poorer and $\gamma = 0.4$ was similar). Thus, I stuck with $\gamma = 1$.

For the exploration parameter ϵ, it is recommended to have a small value like $0.01 - 0.1$, but I found that $\epsilon = 0.25$ was most effective, perhaps at escaping learning to just fire immediately.

I was still disapointed in the performance, so I tried implementing double Q-learning, to see if this improved it. It seemed to have minimal impact, sometimes outperforming the regular Q-learning agent, but sometimes performing worse, though these differences were minimal. I guess that this is due to the problem structure and the reward system.

1.4 Comparison of results

I ran each training for 100000 episodes and then tested the agents for 100000 games. Running the entire code takes about 110 seconds. I have enclosed both Q tables in the zip file and can be retrieved if the training flag is set to 0. I tested both the Q-learning and double-Q agents, and also implemented a random element test, where the Q tables were converted into discrete probabilty distributions, where the probabilty of choosing an action was the probabilistic Q value. This was possible since all the rewards are positive. The following table presents the results.

Measure	**Q-learning**	**Double Q-learning**
Average Profit	441.63429	438.64168
Profit Standard Deviation	509.070819	503.760226
Average Random Profit	322.27071	346.11512
Random Profit Standard Deviation	414.616398	427.536493

Table 1: Results of Q-learning and Double Q-learning.

From the table, we can see that Q-learning and double Q-learning were very similar in results, around the 440 mark. They also had similar characteristically high standard deviations. The random element testing naturally yielded poorer results, which could imply that the strategy learned is not bad. Its standard deviation was lower, which was because it would sometimes try other strategies like firing immediately, gaining some profit if the profit for project C was 0. Comparing these to the simple heuristics, we see that the agents performed almost equivalently to the first and last heuristics where project C dominated.

1.5 Strategy of trained agent

In the Appendix, I present the Q-table from both the Q-learning and double Q-learning, converted into an interpretable format, saying for at each time t, for each state, which action should be taken. Firstly, "-" represents states which are never visited. From this we can see the initial state being state 0 at $t = 0$. Furthermore, since the minimal time needed to complete project B and C are 7 and 14 respectively, the states for those times are not visited.

In general, both Q-learning agents utilise very similar strategies, namely to always start with project C. Also, later in the game the agents learn that it becomes futile to do the longer projects; beyond $t = 34$, only A and B are advised and beyond $t = 41$ only project A. This is the same as the simple heuristic. Also in the last time state, nothing is advised since there is nothing that can be done to gain any profit (by way of how I defined the state space), and it is represented by "No action". In some cases, the agents switch from project C to B or A if doing project C has taken too long indicating a high v and thus no profit.

The main differences between the Q-learning agent and the Double Q-learning agent, is that the former does fire, in state 1, which imitates the second simple heuristic, where if v is too large after trying project A, firing is advised. But since the agent never does project A in testing, we don't see the result of this. The other differences that have a small impact are differences in project choice in state 3.

2 Hyper-Heuristics with the 1CBP

2.1 Implementing the Greedy Heuristic

(Change question flag in line 327 to 1) I implemented the Greedy Heuristic as given in the brief. However, I noticed one slight inconsistency between the algorithm/formulation for the 1CBP given in the brief and the parent paper Akçay and Delorme, 2024. Here in the second constraint, $x_{iq'}$ is summed over the subscript $q' \in \mathcal{H}_i$. So in this we should sum until $K(i)$, and not until K. All that changes in the code, is that I then add an if statement to filter for this. Important to note, no pre-sorting is done.

The easy instance is as follows:

$$\begin{aligned} n &= 3 \\ W &= 10 \\ w &= \begin{bmatrix} 4 & 6 & 1 \end{bmatrix} \\ l &= \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \end{aligned}$$

The optimal assignment (found by gurobi) was using 3 active bins, an assignment of $\begin{bmatrix} 3 & 1 & 1 \end{bmatrix}$ with bin weights of $\begin{bmatrix} 7 & 7 & 5 \end{bmatrix}$. However, the implemented Greedy Heuristic yields 4 active bins, an assignment of $\begin{bmatrix} 1 & 1 & 2 \end{bmatrix}$ with bin weights of $\begin{bmatrix} 10 & 7 & 1 & 1 \end{bmatrix}$. The following illustrates the solution.

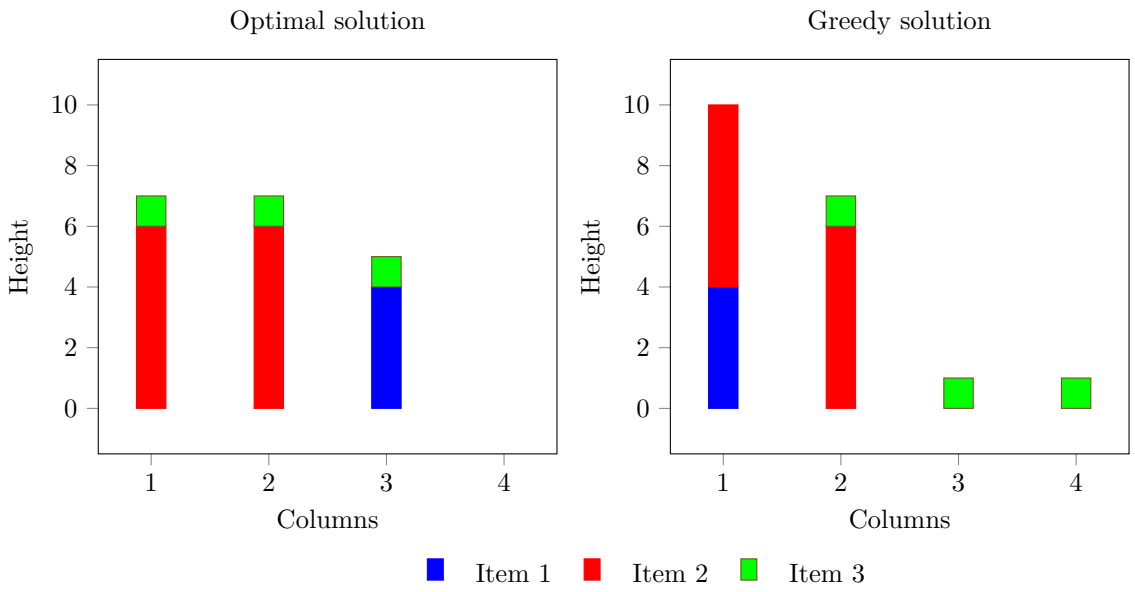

Figure 1: Simple example of failure.

2.2 Gurobi implementation

(Change question flag in line 328 to 2) I implemented the problem formulation into python to be solved by the gurobi solver. Like for the greedy implementation, constraint 2 had to be slightly altered. Also there was a bit of a struggle working with the different indices, as the formulation starts with 1, while python indexing starts with zero. The only main change was such that the k' should be greater than or equal to one, but in my implementation, I take the maximum between zero and $k - l_i + 1$.

The difficult example I chose was just a randomly generated instance with

$$\begin{aligned} n &= 60, \\ W &= 100, \\ w &= \begin{bmatrix} w_1 & \cdots & w_n \end{bmatrix}, \\ l &= \begin{bmatrix} l_1 & \cdots & l_n \end{bmatrix}, \end{aligned}$$

where $w_i \sim Discrete\ Uniform(20, 50)$ and $l_i \sim Discrete\ Uniform(1, 10)$. I set the time limit parameter for the gurobi solver to 5 seconds, and these instances always stopped due to the time limit constraint. To generate this instance, I used the same function as for the data generation for the later sections.

2.3 Q-learning framework

Given the brief's idea about combining the greedy heuristic and the ILP solver, by fixing the first N items using the greedy heuristic, I designed my state space as follows. I wanted to differentiate between instances where the greedy performed well and poorly. Between the greedy and ILP solver (restricted to 5 seconds solve time), as instance size grew, greedy does better. But I found instances where the contiguity constraints were high was where greedy performed better than ILP with 5 seconds, but often the last bins would be less full. Therefore, using the ILP solver on the last items would improve the last bins while being potentially solvable in 5 seconds or close. Hence, I divided the instances into 5 buckets depending on the mean of the contiguity constraints. The buckets were $\begin{bmatrix} [0,3) & [3,6) & [6,9) & [9,14) & [14,20) \end{bmatrix}$.
Then, the agent had 10 choices for N, the number of items to be fixed by the greedy heuristic, created by the

"*linspace*" function in *numpy*, from 0 (only ILP solver) to $\frac{2}{3}\times$ max(*instance size*). The resulting N values are as follows

$$\text{"N values"} = \begin{bmatrix} 0 & 5 & 11 & 17 & 23 & 29 & 35 & 41 & 47 & 53 \end{bmatrix}.$$

I wanted to have a further differentiation, such that the agent could also distinguish instance sizes, hence the effectiveness of the strategy learnt would increase, however this would increase the number of states and actions, meaning the number of episodes would need to be higher which poses a time constraint. Therefore, instead this framework tries to accommodate all sizes without distinction, impeding its success.

2.4 Instance generation

For my Q-learning algorithm to train, I generated 3 sets of random instances; I split the 2000 training instances into 3 sizes with 40, 60, and 80 items respectively. For the weights, a lower bound (1) and upper bound ($\frac{1}{2}\times$ *instance size*) were given and then randomly drawn from a discrete uniform distribution. Similarly, the contiguity constraints had a lower bound of 1. Although, for each separate instance, a new upper bound for the contiguity constraints was drawn from between 2 and 20, giving more variation to the instances. Then the contiguity constraints were also sampled from the discrete uniform distribution using those bounds. The capacities W were explicitly computed for each instance as $1.5\times$ the maximum weight of an item from the same instance.

2.5 Q-learning implementation

(Change question flag in line 328 to 3. Also note the other flags for training and saving) I implemented the Q-learning framework described in section 2.3. Given that the problem is non-stationary due to the varying of the instances, I opted for a constant α of 0.1. For the exploration parameter, I decided to go for a large ϵ of 0.4, such that more of the state space would be explored. I thought this to be important since I was working with such a low number of training episodes. For rewards, I put - (number of bins for hybrid - number of bins for greedy), since we are maximising the Q table values, so a lower Hybrid solution yields a higher reward.

In the framework, the last bucket was for when the mean of the contiguity constraints was above 14. Given that the maximum upper bound for the contiguity constraint random sampler was 20, then for sampling the upper bound N times, and then sampling N sets of samples of size n, the probability of obtaining a mean contiguity constraint higher than 14 is near-zero. Thus I made an error in designing the state space, making states for events that almost never happen. Although there is practically no negative implication.

2.6 Comparison of results

I trained the Q-learning agent for 2000 episodes, where one-third were instances of size 40, one-third were of size 60 and the last third were of size 80. I then tested the hybrid heuristic, along with the greedy heuristic and the 5-seconds ILP solver for 200 instances. Again the test instances were evenly split into the three instance sizes. I have enclosed the Q table in the zip file, and can be loaded if the training flag is set to 0.

The mean improvement on the greedy heuristic was 3.5 bins, while the mean improvement on the ILP solver was 130.17. The performance of the greedy heuristic is far more consistent, than that for the ILP solver, when it is bounded by time. This is illustrated by the mixed heuristic sometimes outperforming gurobi by 500 bins. This happens when the instance size is larger and the contiguity constraints are too, meaning the ILP solver does poorly in the first 5 seconds.

Out of the 200 test instances, the hybrid heuristic performed worse than the greedy in 35 instances, and performed worse than the ILP in 30 cases. In my implementation of the hybrid heuristic, it would be easy and not much added runtime to put a clause that returns the greedy solution if it is better than the one found by the hybrid heuristic. Something similar can be done for the 5-second ILP solver, but this would then double to

runtime of the algorithm.

For the instances with 40 items, the average improvement was 2.86 on the greedy and 113 on the ILP. For the 60-item instances, on average, the hybrid solution was 4.89 bins better than greedy and 92 bins better than ILP. For $n = 80$, the average improvement was 4.7 on greedy and 173 for ILP. Therefore, we can see that, on average, the mixed heuristic performs better in larger instances.

I also ran a training on just instance sizes of 60 to see if this improved the strategy and results learned by the agent. I trained for 200 episodes and tested against 100. The average improvement on greedy was 4.89 and was on 12 instances worse, while the average improvement on ILP was 103.46 and was on 21 instances worse. I have also enclosed this Q table.

2.7 Agent's strategy

Each instance is filtered into a bucket depending on the mean of the contiguity constraints. Then in each bucket, the agent has 10 actions as described above, choosing a number N of items to fix using the greedy heuristic. Below, I present the maximal actions learnt by the agent.

Buckets (mean l)	$[0, 3)$	$[3, 6)$	$[6, 9)$	$[9, 14)$
Best Action (N number to fix with greedy)	0	17	41	47

Table 2: Results of Q-learning.

Hence, the general trend is to fix more items using greedy as the mean of contiguity constraints increases. This makes sense as using greedy to fix more items, and using the ILP solver to focus on the lower capacity bins helps with improving marginally on the greedy heuristic. Notice that for instances that fall into the first bucket, where the mean of the contiguity constraints is less than 3, the hybrid heuristic opts to solve it only with the 5-seconds ILP solve. These instances are easier and thus ILP tends to do better over the limited time. Also, as we have seen, the larger the instance, the worse ILP performs in five seconds, hence the agent may be trying to reduce the instance size of the reduced problem to make it easier for the ILP solver.

Appendix

Time t	**State 0**	**State 1**	**State 2**	**State 3**
0	Project C	-	-	-
1	-	Project C	-	-
2	-	Project C	-	-
3	-	Project C	-	-
4	-	Project C	-	-
5	-	Project B	-	-
6	-	Fire	-	-
7	-	Fire	Project C	-
8	-	Project B	Project C	-
9	-	Fire	Project C	-
10	-	Fire	Project C	-
11	-	Project A	Project C	-
12	-	Project C	Project C	-
13	-	Project C	Project B	-
14	-	Project C	Project C	Project C
15	-	Project B	Project C	Project C
16	-	Project C	Project A	Project C
17	-	Project C	Project B	Project C
18	-	Project C	Project B	Project C
19	-	Project C	Project A	Project C
20	-	Project C	Project B	Project C
21	-	Project C	Project C	Project C
22	-	Project C	Project C	Project C
23	-	Project B	Project C	Project C
24	-	Project C	Project B	Project C
25	-	Project B	Project C	Project A
26	-	Project B	Project B	Project A
27	-	Project C	Project B	Project A
28	-	Project A	Project C	Project A
29	-	Project C	Project C	Project C
30	-	Project C	Project C	Project C
31	-	Project B	Project C	Project C
32	-	Project B	Project B	Project C
33	-	Project C	Project B	Project C
34	-	Project B	Project A	Project A
35	-	Project A	Project B	Project A
36	-	Project B	Project A	Project B
37	-	Project B	Project A	Project B
38	-	Project A	Project B	Project B
39	-	Project A	Project A	Project A
40	-	Project A	Project A	Project A
41	-	Project A	Project A	Project A
42	-	Project A	Project A	Project A
43	-	Project A	Project A	Project A
44	-	Project A	Project A	Project A
45	-	Project A	Project A	Project A
46	-	Project A	Project A	Project A
47	-	No action	No action	No action

Table 3: Q-learning table

Time t	**State 0**	**State 1**	**State 2**	**State 3**
0	Project C	-	-	-
1	-	Project C	-	-
2	-	Project C	-	-
3	-	Project B	-	-
4	-	Project B	-	-
5	-	Project C	-	-
6	-	Project B	-	-
7	-	Project B	Project A	-
8	-	Project C	Project C	-
9	-	Project C	Project A	-
10	-	Project C	Project C	-
11	-	Project C	Project C	-
12	-	Project A	Project C	-
13	-	Project A	Project C	-
14	-	Project A	Project B	Project B
15	-	Project C	Project C	Project C
16	-	Project C	Project C	Project C
17	-	Project C	Project A	Project C
18	-	Project C	Project A	Project C
19	-	Project C	Project C	Project C
20	-	Project A	Project A	Project C
21	-	Project C	Project C	Project C
22	-	Project A	Project C	Project C
23	-	Project C	Project A	Project C
24	-	Project C	Project C	Project B
25	-	Project C	Project C	Project C
26	-	Project C	Project C	Project A
27	-	Project B	Project C	Project B
28	-	Project B	Project C	Project A
29	-	Project C	Project C	Project A
30	-	Project C	Project A	Project A
31	-	Project A	Project A	Project C
32	-	Project C	Project C	Project C
33	-	Project B	Project A	Project B
34	-	Project B	Project B	Project B
35	-	Project B	Project A	Project B
36	-	Project A	Project B	Project A
37	-	Project B	Project B	Project B
38	-	Project B	Project A	Project B
39	-	Project B	Project B	Project B
40	-	Project A	Project A	Project B
41	-	Project A	Project A	Project A
42	-	Project A	Project A	Project A
43	-	Project A	Project A	Project A
44	-	Project A	Project A	Project A
45	-	Project A	Project A	Project A
46	-	Project A	Project A	Project A
47	-	No action	No action	No action

Table 4: Double Q-learning table

References

Akçay, F. B., & Delorme, M. (2024). Solving the parallel processor scheduling and bin packing problems with contiguity constraints: Mathematical models and computational studies. *European Journal of Operational Research*. https://doi.org/https://doi.org/10.1016/j.ejor.2024.09.013