SAKI SS 2021 Homework 4

Author: Nicolas Webersinke

Program code: https://git.io/JCgAI

Summary

The task at hand is finding an optimal policy for a warehouse robot whose task is to store and restore items in a warehouse. The robot can be defined as the agent and the warehouse as the environment. Any optimal policy should minimize the distance the robot must move. This policy is to be found by modelling the problem as a markov decision process (MDP) and solving it by using the value iteration algorithm, which can be attributed to the field of reinforcement learning. The warehouse stores 3 different items which are represented as colors white, blue and red. The robot is in a start/stop position outside the warehouse to pick up or drop off new items. The size of the warehouse is variable in principle, but has the greatest impact on the number of states and, consequently, on the computational demand. The state space of a warehouse, which considers only the next color but no other future colors, is given by

$$numberOfStates = fillingStates^{warehouseSize} \times numberOfActions \times numberOfNextColors$$

Where *fillingStates* is the number of possible states for each shelf (which is 4 in this case, since there are 3 colors and the empty state), *warehouseSize* is the total number of shelves, *numberOfActions* is the number of possible actions (2 in this case, store and restore) and *numberOfNextColors* is the number of possible next colors (or items, which is 3 in this case). For a 2x2 warehouse there are 1,536 states, for a 3x3 warehouse already 1,572,864, which means that I would quickly suffer from a lack of memory. This is the reason why I stick to a 2x2 warehouse in the following, because for a 3x3 warehouse I would have to sample the transition probability matrix somehow.

First, I determine the state matrix. This has the shape 1,536x6 because there are 1,536 states and 4 shelves for a 2x2 Warehouse, and the action and next color for each state must be considered. All colors, fields and actions are encoded in numbers, e.g. 0 for the store action. Then the states are obtained by finding the cartesian product of the 6 columns. The resulting state matrix is used to determine the transition probability matrix.[1] The transition matrix represents the probabilities $P(s' \mid s, a)$ which are the probabilities for a given state *s* and action *a* to get into a new state *s'*. Therefore state *s* only depends on the next state *s'* and not on any previous state. Consequently this matrix is of shape $a \times s \times s'$ or in this case 5x1,536x1,536, since there are 4 fields and the start/stop position and 1,536 states in total. The probabilities are equal to the probability of the next color whenever a transition is possible, e.g. when the action is to store an item and a shelf is empty in the current state and filled with the respective color in the next state. At the beginning, the probabilities of the colors are equal. Later, for evaluation, different distributions are tested. The resulting transition probability matrix must be stochastic, which means that all rows must sum up to one. To achieve this property, I normalize each row if the sum of the row is not one and I set the diagonal to one if there is no non-zero value at all, effectively sinking the agent in that state. Finally, I need to derive the reward matrix with all rewards $R(s, a, s')$ which gives the rewards for each state *s* for an action *a* to get into a new state *s'*. The reward is a hyperparameter for which multiple values are applicable. I base the rewards on the Manhattan distance in the warehouse to the shelf at the defined entry point, since the robot is not allowed to move diagonally. For all possible transitions there is a reward depending on this distance, so that storing or restoring an item in a more distant shelf gives less reward and vice versa. Staying in the start/stop position also gives a reward, but a lower one than for all others.

[1] More precisely, the transition matrix and the reward matrix are multiple matrices or tensors. I stick to the singular as this is common.

This gives the robot the incentive to move only when it should, but not enough reward to stay permanently lazy in the start/stop position. I tried several rewards systematically, however, none led to a different result. Later, for evaluation, the different color distributions are also considered for rewards, e.g. the robot gets more reward for storing a more likely color closer to the entry point.

Finally, with the transition probability matrix and reward matrix at hand I now have the inputs to solve the Bellman equation.[2] The only thing left to define is the discount factor, which specifies the importance of the future reward for the current state. For the discount factor, different values did not yield to different results, which is why I stick to the common 0.9. I then eventually use the value iteration algorithm implemented by the mdptoolbox library to solve the Bellman Equation.[3] The method of the library returns the optimal policy, which gives us the optimal field the robot should move to for each of the 1,536 states in our warehouse environment.

Evaluation

To evaluate the optimal policy found, I perform a test run with several store and restore instructions for different items and compare the distance moved by applying the optimal policy found with the distance moved by following a greedy procedure. In general, I would expect an optimal policy to outperform the greedy strategy the bigger the warehouse is. The reason for this is obvious: In smaller warehouses, there are fewer paths to optimize. For a 2x2 warehouse, this means that the optimal policy is unlikely to perform much better than the greedy strategy of always choosing the shortest distance.

Indeed, both strategies perform equally well, with a total distance moved of 114 measured by Manhattan distance from the defined entry point. This implies that for a 2x2 warehouse there isn't much to optimize. However, this could be different for a larger warehouse.

I also repeat the evaluation for a different color probability distribution, e.g. red items are more likely than others. The reason for this is that I would expect that in this case a greedy strategy would no longer perform better. However, looking at the screenshot, the optimal policy now performs worse than the greedy strategy. This may again depend on the test set, the selected probability distribution or the selected rewards. I tried different values for this systematically, but did not get a better result.

[2] See Bellman, 1957 (https://www.jstor.org/stable/24900506).
[3] I also tried other algorithms such as policy iteration but got the same results.

Screenshot

Initialize warehouse with properties.

```
[144]: my_warehouse = Warehouse(layout=(2,2), possible_fillings=4, actions=2, possible_next_colors=3)
```

Get transition matrices.

```
[145]: my_warehouse.generate_transition_matrix()
```

```
[145]: array([[[1.00e+00, 0.00e+00, 0.00e+00, ..., 0.00e+00, 0.00e+00,
                0.00e+00],
               [0.00e+00, 1.00e+00, 0.00e+00, ..., 0.00e+00, 0.00e+00,
                0.00e+00],
               [0.00e+00, 0.00e+00, 1.00e+00, ..., 0.00e+00, 0.00e+00,
                0.00e+00],
               ...,
```

Get reward matrices.

```
[146]: my_warehouse.generate_reward_matrix()
```

```
[146]: array([[[0., 0., 0., ..., 0., 0., 0.],
               [0., 0., 0., ..., 0., 0., 0.],
               [0., 0., 0., ..., 0., 0., 0.],
               ...,
```

Results and evaluation for items with equal probability

Find optimal policy.

```
[152]: np.array(my_warehouse.get_optimal_policy(discount=0.9, verbose=False))
```

```
[152]: array([4, 4, 4, ..., 4, 4, 4])
```

Evaluate it against a greedy strategy.

```
[153]: my_warehouse.evaluate("test.txt")
```

```
Total Manhattan distance moved using optimal policy: 114
Total Manhattan distance moved using greedy strategy: 114
```

Results and evaluation for items with unequal probability (red being more likely)

Find optimal policy.

```
[158]: np.array(my_warehouse.get_optimal_policy(discount=0.9, verbose=False))
```

```
[158]: array([4, 4, 4, ..., 4, 4, 4])
```

Evaluate it against a greedy strategy.

```
[159]: my_warehouse.evaluate("test.txt")
```

```
Total Manhattan distance moved using optimal policy: 135
Total Manhattan distance moved using greedy strategy: 114
```