SAKI SS19 Homework 3

Author: Maximilian Lattka

Program code: https://github.com/nakami/oss-saki-ss19-exercise-3/releases/tag/oss-saki-ss19-exercise-3

Summary

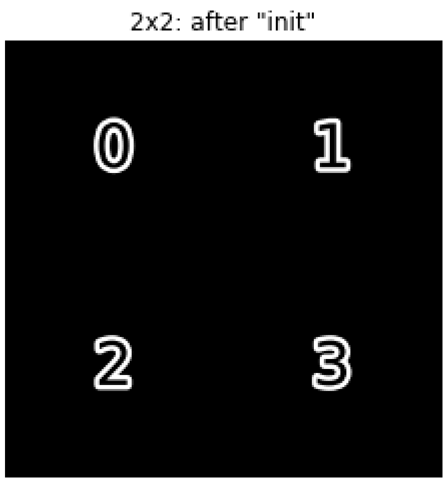
In contrast to exercise 1 and 2, exercise 3 is not about a classification task, but an optimization problem involving a storage warehouse. Given a 3x2 (or 2x2 for testing) storage layout and a training set, a Markov Decision Process (MDP) and Reinforcement Learning can be combined to optimize the routes of a robot arm. To solve the 2x2 and 3x2 MDPs the framework MDP Toolbox[1] came to use. As no strict requirements or skeletons were provided, this exercise required the implementation of an environment and a baseline algorithm to allow evaluation.

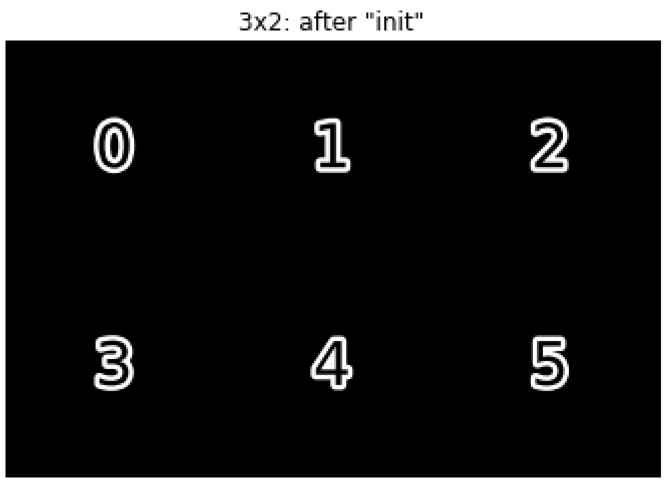
Overview and Data Exploration

Code: `overview.ipynb`, `GreedyAlgo.py`, `AIAlgo.py`

Data: `data_2x2/warehousetraining2x2.txt`, `data_3x2/warehousetraining3x2.txt`

First, I took a look at the provided data sets and planned the next steps. For these steps, please see the jupyter notebook `overview.ipynb`. In the four {2x2, 3x2}-{training, order} text files each line contains a tuple of a store/restore action and an item color (white, blue, red) which represent for robot arm commands. I calculated the distributions of each of the six tuples and rounded the resulting numbers to four decimals. These will be used to populate Transition Probability Matrices (TPMs) later. I define one state as a tuple of integers: four integers in the 2x2 (six for 3x2) represent the item storage fields and can each be one of four item states (0: empty, 1: white, 2: blue, 3: red). The last integer stands for the action the robot arm is supposed to do next (0..2: store white, blue, red; 3..5: restore white, blue, red). Therefore, there are 1536 possible states in total for the 2x2 and 24576 states for the 3x2 layout. The decisions to be determined by Reinforcement Learning are the positions where the next action (store/restore COLOR) should take place. For testing, evaluation and research purposes, I implemented a visualisation class in `WarehouseVisualizer.py` which outputs an image for a 2x2 or a 3x2 storage und its contents. The robot arm will start in field 0 every time an action is performed.





WarehouseVisualizer output of an empty storage warehouse (2x2 to the left, 3x2 to the right)

The warehouse contents are addressed by a list of fields which holds integers representing the color it holds (0: empty, 1: white, 2: blue, 3: red), similar to the data structure used for states. For the algorithms to run, I implemented classes for executing robot arm actions in two ways: `GreedyAlgo.py` for a greedy algorithm (always access the closest

[1] Markov Decision Process (MDP) Toolbox: https://pymdptoolbox.readthedocs.io/en/latest/api/mdptoolbox.html

storage field possible) and `AIAlgo.py` for a policy-based algorithm (use a policy to determine which storage field to access). Both classes store the total distance the robot arm moved which can be then inspected after executing a number of actions. The distance in my layouts works in a way that putting or taking something from the first field (0) already costs a distance of 1, operating on the adjacent fields costs a distance of 2 and so on.

MDP and Reinforcement Learning

Code: `mdp_2x2.ipynb`, `mdp_3x2.ipynb`

A MDP requires Transition Probability Matrices (TPMs) and a Reward Matrix (RM). See the respective jupyter notebook for the complete 2x2 or the 3x2 procedure. The AI algorithm is supposed to find the optimal field for every state and store/restore action, so I created a TPM for each possible position an action can take place in the storage warehouse. This way, I end up with four TPMs for 2x2 layout and six TPMs for the 3x2 layout. However, since the amount of states varies, the TPMs for 2x2 are of 1536x1536 dimension and those for 3x2 are of 24576x24576. The action distributions I calculated earlier (see jupyter notebook `overview.ipynb`) were used to populate the TPMs by finding successor states (having valid storage changes) at given action position and setting the TPM for the two states to the percentages of the corresponding action occurrences in the distribution. The TPMs have to be stochastically correct which means that every row has to sum up to one. For rows that are empty, this obviously is not the case, so I set a state's own transition TPM to 1 if doesn't suit this criteria. Next, the Reward Matrix with respective dimensions of 4x1536 and 6x24576 (action positions x states) was generated. The main idea is that the robot arm should be rewarded for small movements. That said, I created the RM to reflect that the further the robot arm has to go, the fewer rewards he will get. To avoid staying in the same state (in other words: the robot arm refusing actions), I punish it with a huge negative reward for such illegal state transitions. For all these operations I implemented a few helper functions which are also included in the particular jupyter notebook. Finally, the MDP Toolbox is called by providing the individual TPMs and RM. Here, a limit for the iterations and a discount factor can be chosen. I don't want to go too deep into the technical details, so I keep it brief. MDP Toolbox offers eight algorithms in total to solve MDPs. However, two in particular were recommended by the lecturer which I also limited myself to: `PolicyIteration`[2] which basically makes use of on one initial policy and improves its utilities until the policy converges [SB17, Chapter 4.3]. `ValueIteration`[3] on the other hand "[solves] Bellman's equation iteratively" and therefore avoids the policy evaluation every iteration which is done in `PolicyIteration` [SB17, Chapter 4.4].

[SB17]: Sutton, R. S., & Barto, A. G. (2015). Reinforcement Learning: An Introduction.

Evaluation

Code: `mdp_2x2.ipynb`, `mdp_3x2.ipynb`, `GreedyAlgo.py`, `AIAlgo.py`

For each layout I solved MDPs using the `PolicyIteration` (discount factor 0.999) and `ValueIteration` (discount factor 0.95) algorithms provided by the MDP Toolbox. After some experimenting on the 3x2 sets, the used resources (memory, calculation power/time) allowed me to adjust the discount factors to reasonable training times. On my workstation (Windows10, 16GB RAM, Intel Core i7 5820K) one single training takes about 2 hours. For the 2x2 layout, populating the necessary matrices and running the training was of no issue regarding resources. For completeness sake, I tested the resulting policies not only on the order sets, but also on the training data set as well. By comparing the total distance traveled in the greedy algorithm (`GreedyAlgo.py`) as baseline and the policy-based (`AIAlgo.py`) implementations, I was able to observe some underperforming scenarios, but also some benefits of the policy-based approaches over the greedy algorithm.

[2] PolicyIteration: https://pymdptoolbox.readthedocs.io/en/latest/api/mdp.html#mdptoolbox.mdp.PolicyIteration
[3] ValueIteration: https://pymdptoolbox.readthedocs.io/en/latest/api/mdp.html#mdptoolbox.mdp.ValueIteration

2x2 Layout

Code: `mdp_2x2.ipynb`, `GreedyAlgo.py`, `AIAlgo.py`

Data: `data_2x2/warehousetraining2x2.txt`, `data_2x2/warehouseorder2x2.txt`

For the 2x2 storage layout, both MDP solutions (`PolicyIteration` with a discount factor of 0.999; `ValueIteration` with a discount factor of 0.95) perform as good as the greedy implementation. To recall, on my 2x2 layout running an action on field 0 adds a distance of 1, an action on field 1 or 2 yields a distance of 2 and executing one on field 3 a distance of 3. On the training data set all implementations walked for a total distance of 14401 steps and 114 on the order data set. I also inspected the storage contents and found no behavior differences between the algorithms in decision making. However, this may be expected as the greedy algorithm may not provide any room for improvements on such a small layout. There may be differences however in edge-case scenarios: if a manufactured data set demands storing an item and it occupies a close storage field for a long time while using the greedy approach, the other MDP solutions will potentially take over - for the given data sets this was not the case.

3x2 Layout

Code: `mdp_3x2.ipynb`, `GreedyAlgo.py`, `AIAlgo.py`

Data: `data_3x2/warehousetraining3x2.txt`, `data_3x2/warehouseorder3x2.txt`

In comparison to the 2x2 storage, the results on the 3x2 layout differ. I trained both MDP algorithms with the same discount factors as before (PolicyIteration with a discount factor of 0.999; ValueIteration with a discount factor of 0.95). Not only did the generating process of the TPMs and RM turn out to be demanding in regards to resources (time, RAM storage), but also the training itself took a lot of time and scratched my workstation's capabilities.

	training data set	**order data set**
GreedyAlgo	26144	130
policy-based (PolicyIteration)	26456	126
policy-based (ValueIteration)	26422	126

Comparing the total distance the robot arm traveled using different algorithms and data sets

One can observe that the greedy algorithm performs best on the training data set and the algorithm paired with the policies does slightly worse with either of both policies:

training data set:

PolicyIteration: 26456/26144 ≈ 1.012 => **1,2% longer total distance** than greedy

ValueIteration: 26422/26144 ≈ 1.011 => **1,1% longer total distance** than greedy

However, the results for the order data set certainly look promising. Here both action executions of the policy-based algorithm do better than the baseline greedy implementation and also, by the same extent:

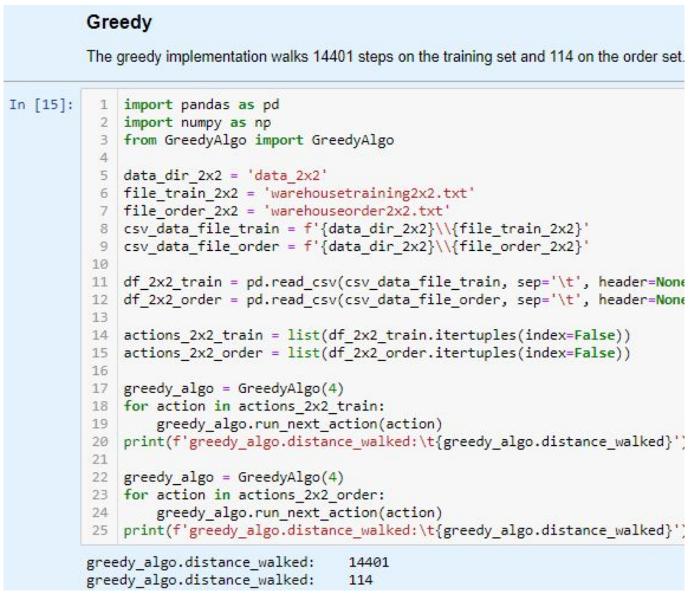
order data set:

PolicyIteration and ValueIteration: 1 - 126/130 ≈ 0.031 => **3,1% shorter total distance** than greedy

For in-depth understanding, I might use discount factors closer to 1.0 for further trainings. Also it may be worthwhile to inspect certain situations where the policy-based algorithm decides to take a different position side-by-side and investigate the outcome of such decisions on the long run.

Screenshots

2x2

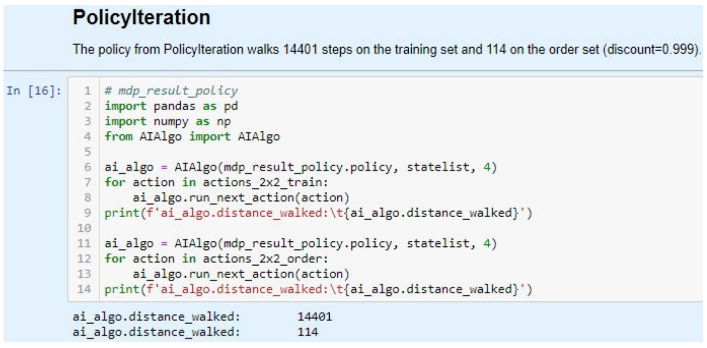


Greedy

The greedy implementation walks 14401 steps on the training set and 114 on the order set.

```
In [15]:
import pandas as pd
import numpy as np
from GreedyAlgo import GreedyAlgo

data_dir_2x2 = 'data_2x2'
file_train_2x2 = 'warehousetraining2x2.txt'
file_order_2x2 = 'warehouseorder2x2.txt'
csv_data_file_train = f'{data_dir_2x2}\\{file_train_2x2}'
csv_data_file_order = f'{data_dir_2x2}\\{file_order_2x2}'

df_2x2_train = pd.read_csv(csv_data_file_train, sep='\t', header=None
df_2x2_order = pd.read_csv(csv_data_file_order, sep='\t', header=None

actions_2x2_train = list(df_2x2_train.itertuples(index=False))
actions_2x2_order = list(df_2x2_order.itertuples(index=False))

greedy_algo = GreedyAlgo(4)
for action in actions_2x2_train:
    greedy_algo.run_next_action(action)
print(f'greedy_algo.distance_walked:\t{greedy_algo.distance_walked}'

greedy_algo = GreedyAlgo(4)
for action in actions_2x2_order:
    greedy_algo.run_next_action(action)
print(f'greedy_algo.distance_walked:\t{greedy_algo.distance_walked}'
```

```
greedy_algo.distance_walked:	14401
greedy_algo.distance_walked:	114
```

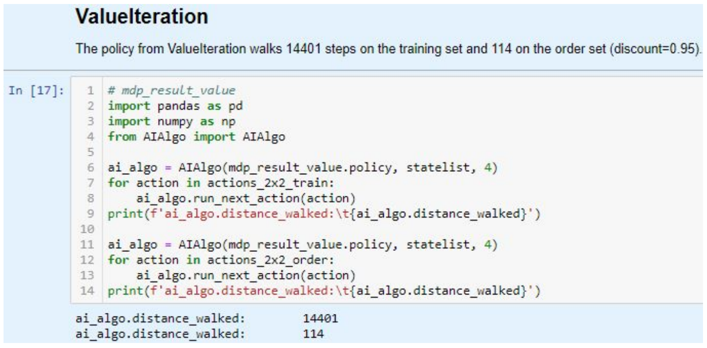


PolicyIteration

The policy from PolicyIteration walks 14401 steps on the training set and 114 on the order set (discount=0.999).

```
In [16]:
# mdp_result_policy
import pandas as pd
import numpy as np
from AIAlgo import AIAlgo

ai_algo = AIAlgo(mdp_result_policy.policy, statelist, 4)
for action in actions_2x2_train:
    ai_algo.run_next_action(action)
print(f'ai_algo.distance_walked:\t{ai_algo.distance_walked}')

ai_algo = AIAlgo(mdp_result_policy.policy, statelist, 4)
for action in actions_2x2_order:
    ai_algo.run_next_action(action)
print(f'ai_algo.distance_walked:\t{ai_algo.distance_walked}')
```

```
ai_algo.distance_walked:	14401
ai_algo.distance_walked:	114
```



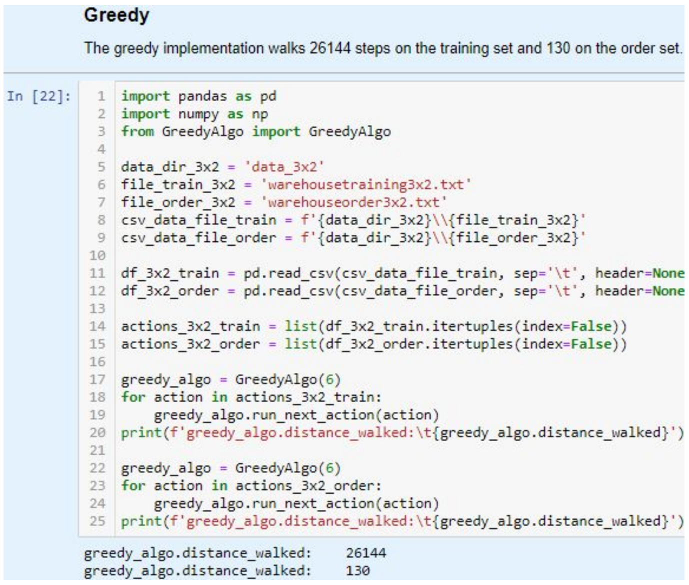
ValueIteration

The policy from ValueIteration walks 14401 steps on the training set and 114 on the order set (discount=0.95).

```
In [17]:
# mdp_result_value
import pandas as pd
import numpy as np
from AIAlgo import AIAlgo

ai_algo = AIAlgo(mdp_result_value.policy, statelist, 4)
for action in actions_2x2_train:
    ai_algo.run_next_action(action)
print(f'ai_algo.distance_walked:\t{ai_algo.distance_walked}')

ai_algo = AIAlgo(mdp_result_value.policy, statelist, 4)
for action in actions_2x2_order:
    ai_algo.run_next_action(action)
print(f'ai_algo.distance_walked:\t{ai_algo.distance_walked}')
```

```
ai_algo.distance_walked:	14401
ai_algo.distance_walked:	114
```

3x2

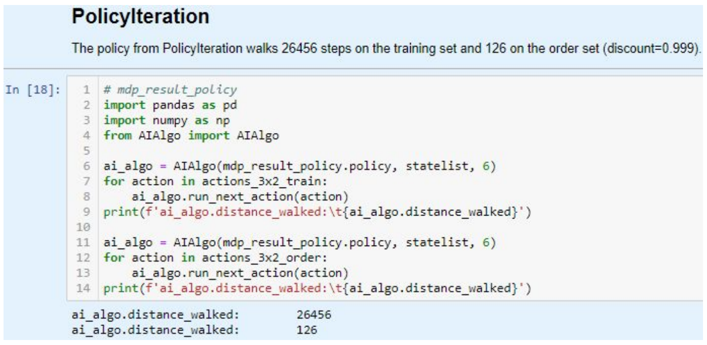


Greedy

The greedy implementation walks 26144 steps on the training set and 130 on the order set.

```
In [22]:
import pandas as pd
import numpy as np
from GreedyAlgo import GreedyAlgo

data_dir_3x2 = 'data_3x2'
file_train_3x2 = 'warehousetraining3x2.txt'
file_order_3x2 = 'warehouseorder3x2.txt'
csv_data_file_train = f'{data_dir_3x2}\\{file_train_3x2}'
csv_data_file_order = f'{data_dir_3x2}\\{file_order_3x2}'

df_3x2_train = pd.read_csv(csv_data_file_train, sep='\t', header=None
df_3x2_order = pd.read_csv(csv_data_file_order, sep='\t', header=None

actions_3x2_train = list(df_3x2_train.itertuples(index=False))
actions_3x2_order = list(df_3x2_order.itertuples(index=False))

greedy_algo = GreedyAlgo(6)
for action in actions_3x2_train:
    greedy_algo.run_next_action(action)
print(f'greedy_algo.distance_walked:\t{greedy_algo.distance_walked}')

greedy_algo = GreedyAlgo(6)
for action in actions_3x2_order:
    greedy_algo.run_next_action(action)
print(f'greedy_algo.distance_walked:\t{greedy_algo.distance_walked}')
```

```
greedy_algo.distance_walked:	26144
greedy_algo.distance_walked:	130
```

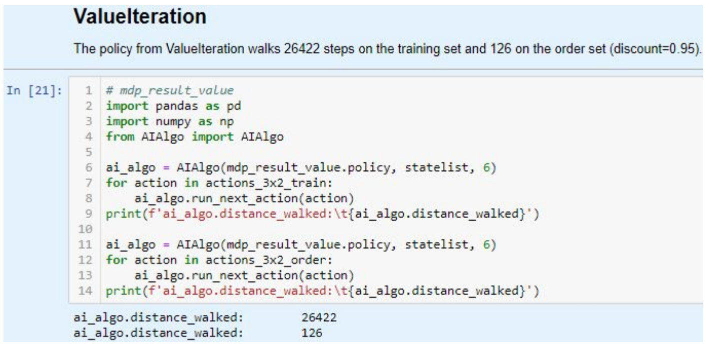


PolicyIteration

The policy from PolicyIteration walks 26456 steps on the training set and 126 on the order set (discount=0.999).

```
In [18]:
# mdp_result_policy
import pandas as pd
import numpy as np
from AIAlgo import AIAlgo

ai_algo = AIAlgo(mdp_result_policy.policy, statelist, 6)
for action in actions_3x2_train:
    ai_algo.run_next_action(action)
print(f'ai_algo.distance_walked:\t{ai_algo.distance_walked}')

ai_algo = AIAlgo(mdp_result_policy.policy, statelist, 6)
for action in actions_3x2_order:
    ai_algo.run_next_action(action)
print(f'ai_algo.distance_walked:\t{ai_algo.distance_walked}')
```

```
ai_algo.distance_walked:	26456
ai_algo.distance_walked:	126
```



ValueIteration

The policy from ValueIteration walks 26422 steps on the training set and 126 on the order set (discount=0.95).

```
In [21]:
# mdp_result_value
import pandas as pd
import numpy as np
from AIAlgo import AIAlgo

ai_algo = AIAlgo(mdp_result_value.policy, statelist, 6)
for action in actions_3x2_train:
    ai_algo.run_next_action(action)
print(f'ai_algo.distance_walked:\t{ai_algo.distance_walked}')

ai_algo = AIAlgo(mdp_result_value.policy, statelist, 6)
for action in actions_3x2_order:
    ai_algo.run_next_action(action)
print(f'ai_algo.distance_walked:\t{ai_algo.distance_walked}')
```

```
ai_algo.distance_walked:	26422
ai_algo.distance_walked:	126
```