Deep Reinforcement Learning for Solving Train Unit Shunting Problem with Interval Timing

Wan-Jui Lee[1], Helia Jamshidi[2], and Diederik M. Roijers[3]

[1] R&D Hub Logistics, Dutch Railways, Utrecht, The Netherlands
`wan-jui.lee@ns.nl`
[2] CiTG, TU Delft, Delft, The Netherlands
`Helia.jamshidi@ptvgroup.com`
[3] Microsystems Technology, HU Univ. of Applied Sciences Utrecht, the Netherlands
AI Research Group, Vrije Universiteit Brussel, Brussels, Belgium
`diederik.yamamoto-roijers@hu.nl`

Abstract. The Train Unit Shunting Problem (TUSP) is a hard combinatorial optimization problem faced by the Dutch Railways (NS). An earlier study has shown the potential to solve the parking and matching sub-problem of TUSP by formulating it as a Markov Decision Process and employing a deep reinforcement learning algorithm to learn a strategy. However, the earlier study did not take into account service tasks, which is one of the key components of TUSP. Service tasks inject additional time constraints, making it an even more challenging application to tackle. In this paper, we formulate the time constraints of service tasks within TUSP to enable deep reinforcement learning. Using this new formalization, we compare two learning strategies, DQN and VIPS, to evaluate the most suitable one for this application. The results show that by assigning extra triggers to agents at fixed time intervals, the agent accurately learns based on VIPS to send the trains to the service tracks in time to comply with the departure schedule.

Keywords: Train Unit Shunting · Deep Reinforcement Learning · Value Iteration.

1 Introduction

The Dutch Railways (NS) operates a daily number of 4,800 domestic trains serving more than 1.2 million passengers each day. When trains are temporarily not needed to transport passengers they are maintained and cleaned at dedicated *shunting yards*. Here, NS is dealing with the so-called shunting activities [2]. The Train Unit Shunting Problem (TUSP) is a computationally hard sequential decision-making problem. At a service site, train units need to be inspected, cleaned, repaired and parked during the night. Furthermore, specific types of train units need to depart at a specific times. Together, these requirements and constraints make the TUSP a highly challenging scheduling and planning problem.

In 2018 NS has started to explore the possibility to solve the TUSP problem with Deep Reinforcement Learning [5], and this has shown great potential in solving the parking problem using Deep Q-learning. In [5] the authors only focused on matching incoming trains to outgoing trains and sending trains to parking locations. However, the service tasks, which form a key component and often are the bottleneck, are not yet taken into account. More generally, time constraints were out of scope.

For a complex planning and scheduling problem like TUSP, a proper design of the environment, reward functions, state representation and actions is critical to keep the problem not only tractable, but even learnable at all. To better understand and monitor the learning of an agent on the TUSP problem, we keep track of specific movements and violations during training. For instance, we monitor: correct departures, sending trains to the service track if there is undone service task, and parking at tracks with sufficient space. These visualizations and monitoring of the agent behaviors have helped a lot in iteratively improving upon the design of the agent, as well as the environment representing the TUSP problem. We believe that this strategy of monitoring shall be also helpful for other real-world applications which are as complex as TUSP.

In the following sections, we first describe the TUSP problem and its environment in detail, followed by the Deep Reinforcement Learning (DRL) methods our agents employ. Afterwards, we empirically compare different learning methods. We close with a discussion of the effectiveness of our formulation and learning methods.

2 Train Unit Shunting Problem (TUSP)

First, let us discuss an example to get a feel for what TUSP problems are. Figure 1 and Figure 2 illustrate a small TUSP problem. Figure 1 shows a service site. Trains enter and exit the site over gateway track G and can be parked on track 1 to 4. The tracks are connected by the switches a and b. The length of the parking tracks is displayed in meters. A cleaning platform allows internal cleaning tasks to be performed on trains positioned on track 2.

Figure 2 is a planning instance. Figure 2(b) shows the lists of incoming and departing trains together with their incoming and departing time. The departure trains specify the composition of subtypes instead of the train units, since the assignment is part of the matching problem. The ordering of the train units or subtypes indicates from left to right the order of the train units of subtypes in the train on the service site. Figure 2(a) describes the type of service tasks required for specific train units and the duration of such service tasks. The goal is to find a feasible shunting plan to move train units onto a shunting yard and make sure they can be serviced and parked in suitable locations, and subsequently depart at the desired time.

Please note that this example is only for illustration purposes, and real shunting yards and problem instances faced by NS are larger and more complex.

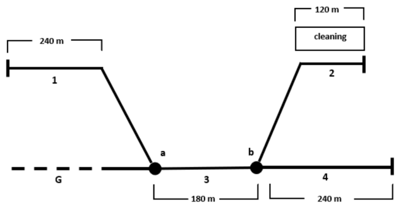
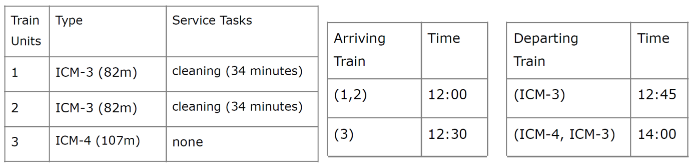


Fig. 1. An example of a service site.



Train Units	Type	Service Tasks
1	ICM-3 (82m)	cleaning (34 minutes)
2	ICM-3 (82m)	cleaning (34 minutes)
3	ICM-4 (107m)	none

(a) Train units

Arriving Train	Time
(1,2)	12:00
(3)	12:30

Departing Train	Time
(ICM-3)	12:45
(ICM-4, ICM-3)	14:00

(b) Arrivals and departures

Fig. 2. An example scenario

2.1 Design of the TUSP Environment

The design of the environment is essential for an agent to solve the TUSP problem effectively and efficiently. Specifically, we need to encode the essential components in the sequential decision making process of making a shunting plan in a compact, yet informative way.

The process starts with a train arrival at the gate track, and a decision needs to be made to move the arrival train immediately from the gate track to a parking track. If the train requires certain services, it will need to be moved to the corresponding service track from its current position before its expected departure time. If there is no service needed, the train will park on a parking track and wait for departure. At a certain moment, a train with a specific type and composition will be requested for departure, and the chosen train will move from the parking track to the gate track to depart. When all the arrival trains are handled properly and all departure trains depart in time with all service done, the planning process terminates and all the decisions form a feasible solution.

Therefore, the TUSP environment has to provide (1) proper triggers to indicate agents when to take an action, (2) a proper action set to allow agents to make necessary decisions, (3) rewards and punishment to teach agents whether a action is good or bad, and (4) yard status and expected train arrival/departures to provide agents with sufficient information for decision making. All these elements will be introduced in the following:

Triggers The environment maintains a trigger list in which each trigger has a countdown timer indicating how much time left to its execution. After a trigger is processed, it is deleted from the trigger list. The types of trigger events include:

1. An *Arrival* trigger is described with the train and time. When it is processed, the corresponding train appears on the gate track.
2. A *Departure* trigger is described with the train type and time.
3. An *End-Of-Service* trigger is described with the train unit and time. It can only be added to the trigger list by starting a service.
4. An *End-of-Movement* is also a trigger added after each movement state changes. When *End-of-Movement* is triggered the agent can take a new action.

These trigger events alone are not sufficient for agent to learn to put trains to service. Therefore, we propose the *wait* action to wait until the next trigger. In this setting, the wait action can result in time being moved forward with anywhere from a minute to several hours. While full information of how far the next trigger is, is encoded in the state, in early training the agent cannot access that information. This will make outcome of the wait action highly unpredictable and agent is less likely to learn it, despite it being crucial to solving an episode.

To address this phenomenon, it is best to make a trade-off between uniform and reactive sampling of time. This can be achieved by limiting the maximum interval between triggers. If the duration between triggers is long, extra triggers will be sampled with an uniform time interval. It should be mentioned that this condition will not change or enhance optimum solution to the planing problem. However, it proved to push training of the agent in the direction of the optimal policy.

Action space In this work, the action space consists of track-to-track movements, plus the wait action. No combination or splitting actions of train units are considered in this study and the compositions of arrival and departure trains are assumed to remain the same. Each movement has a start track and an end track. Movement actions only exist when these tracks are inherently connected; hence the agent can never violate routing between tracks. We note that this holds given that no simultaneous movements can be performed. Since all track pairs (start track to end track) are either connected to each other with a movement to the right or a movement to the left, there can only be maximum of one train which can move from the start track to the end track. Wait actions will forward the state of the yard in time to a fixed time interval or the next train arrival, a required train departure, or the end of service (cleaning) of a train. Via this action space, the agent has the freedom to move each train around.

Note that if a train unit has an undone service task and is sent to the corresponding service track, the service task is assumed to be started automatically after the movement.

Rewards and violations When the agent chooses an action that causes one of the following violations, the episode ends.

1. Choosing a start track that is empty and therefore there is no train to move.
2. Choosing to wait when there is an arrival or departure train.

3. Parking a train on a gate track or relocation track.
4. Choosing the wrong time or a wrong train type for departure.
5. Choosing a train with undone service for departure.
6. Moving a train while in service.
7. Violating the track length.
8. Missing a departure or arrival while doing other movements.

For actions that result in certain situations, rewards will be given:

- Moving to the relocation track: -0.3
- Move to the service track while no service is required: -0.5
- Right departure: +2.5
- Wait for service to end:+(duration in minutes)/60
- End service: +(duration in munites)/60
- Find a solution: +5
- Others: 0

State representation For the state representation, all data describing train arrival and departure schedules, and their required service time on each track is incorporated. The current positions and order of trains on tracks are also included. The type of a train unit is not communicated directly in the state, however to match leaving train units with required train type, for each train in the yard and each train which has not arrived yet, opportunities when this train can leave the yard are indicated. To summarize, the following information are encoded into the state representation and the concatenated into a vector:

- Position of train units on the track.
- Required service time of train units.
- Whether a train unit is under service.
- Length of train units.
- Time to arrival of train units.
- Is it the arrival time of a train unit?
- Next 3 departure opportunities of the same train type.
- Is it the departure time of the same train type?

3 Deep Reinforcement Learning for TUSP

In this section we describe our methodology for learning initial solutions (i.e., policies) for the improved formulation of the TUSP, as described in the previous section. As mentioned earlier, using DRL methods to solve the parking and matching subproblem of TUSP have been proposed [5]. However, the new TUSP formulation is significantly harder, due to the inclusion of temporal constraints as well as service tasks.

Consider the general setting shown in Figure 3 where an agent interacts with an environment. At each time step t, the agent observes a state signal s_t, and performs an action a_t. Following the action, the state of the environment transitions to

s_{t+1} and the agent receives reward r_t. We note that in our formulation, the state signal is sufficiently Markovian to model the problem as fully observable, i.e., the state transitions and rewards can be stochastic and are assumed to have the Markov property; i.e. the state transition probabilities and rewards depend only on the state of the environment s_t and the action taken by the agent a_t. It is important to note that the agent can only control its actions, and has no prior knowledge of which state the environment would transition to or what the reward may be. By interacting with the environment, during training, the agent can learn about these quantities. The goal of learning is to maximize the expected cumulative discounted reward: $\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t]$, where $\gamma \in (0, 1]$ is the factor discounting future rewards. In our application γ is set to 0.99.

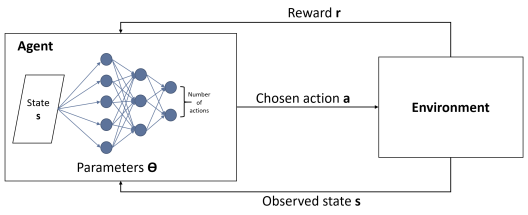


Fig. 3. Showing interaction of DRL agent with the environment

The agent picks actions based on a policy, defined as a probability distribution over actions: $\pi(s, a)$ which is the probability that action a is taken in state s. As there is at least one optimal deterministic stationary policy in a fully observable infinite-horizon discounted MDP [3] (as our TUSP formulation is), policies are often written as a mapping from states to actions, $\pi(s)$. In many popular reinforcement learning methods [8], this policy is derived from an state-action-value function $Q(s, a)$, i.e., $\pi(s) = \arg\max_a Q(s, a)$. However, in most problems of practical interest, there are many possible (state, action) pairs. Hence, it is impossible to store the state-action-value function in tabular form and it is common to use function approximators. A function approximator has a manageable number of adjustable parameters, θ; we refer to these as the policy or network parameters. The justification for approximating the Q-function is that similar states should have similar state-action values. Deep Neural Networks (DNNs) are a popular choice and have been shown to be usable as Q-function approximators for solving large-scale reinforcement learning tasks [4]. An advantage of DNNs is that they do not need hand-crafted features.

3.1 DQN

In the previous study of [5], the Deep Q-Network (DQN) [4] is adopted to solve the parking and matching subproblem of the TUSP. Their architecture is divided into two parts. First, a series of convolutional layers learns to detect increasingly abstract features based on the input. Then, a dense layers map the set of those features present in the current observation to an output layer with one node for every possible action possible in environment. The Q-values correspond to how good it is to take a certain action given a certain state. This can be written as $Q(s, a)$. Deep neural networks can be adopted as a function approximator for the Q-values of each (state,action) pair. This enables generalization from seen states to unseen states.

For (their subproblem of) TUSP [5] needed to make slight changes to standard DQN. The reason for this is that many actions are invalid at given time-steps. As DQN needs an equal number of outputs for each timestep, and must learn not to perform invalid actions, taking an invalid action is modelled as: receiving a highly negative reward, and ending the episode. However, because the action-space in TUSP is very large, this preempts learning efficiently: invalid actions are taken too often, so actually ending an episode with only valid actions because a rare occurrence using an ε-greedy exploration strategy. Therefore, the first three times an invalid action is chosen, the state is reverted to the previous state (before the offending action), and the strategy is switched to greedy, excluding the invalid action that was just taken. Note that the greedy action according to the network might also be invalid. In this case, the second-best action according to the action-value network Q is chosen. [5] found that this was an essential adaptation to make DQN work for their subproblem of TUSP. Because our formulation of TUSP has similar action spaces, but is even more complex (i.e., has more challenging constraints), we make use of this adaptation as well.

3.2 Value iteration with post-decision state (VIPS)

In DQN, the agent had difficulty handling transition function for TUSP. This is mainly due to the large action space, and the many invalid actions (leading to terminal states). This made learning slow. As our TUSP formulation is more complex, we found that DQN became too slow again. However, there some observations we can exploit.

In general the transition function determines how likely it is that we reach next state s_{t+1}. However, if we are in s_t in a deterministic problem like TUSP, the probability of a certain action is either zero or one. Furthermore, the transition function is known to us. Therefore, rather than using a Q-learning algorithm, we can employ a planning algorithm that exploits this. We extend value iteration for TUSP, to show the deep learning agent the deterministic transition function in advance.It should be emphasized though, that unlike DQN, after the agent has converged in value iteration, it cannot be used independent of the environment simulator to pick out actions, as it requires the transition function to operate.

Value function Value function $V(s)$ measures how good it is to be in a specific state. By definition, it is the expected discounted rewards that collect totally following the specific policy:

$$V^{\pi}(s_t) = \sum_{t'=t}^{T} \mathbb{E}_{\pi\theta}[\gamma^{t'-t} r(s_{t'}, a_{t'})|s_t] \tag{1}$$

Approximate value iteration First, dynamic programming can be used to calculate the optimal value of V iteratively.

$$V^*(s) = \max_a \mathbb{E}[r_{t+1} + \gamma V^*(s_{t+1}|s_t = s, a_t = a)] \tag{2}$$

Then, the value function can be used to derive the optimal policy.

The Post-Decision State Variable The post-decision state variable is the state of the system after we have made a decision (deterministic) but before any new information (stochastic in general) has arrived. For a wide range of applications, the post-decision state is no more complicated than the pre-decision state, and for many problems is it much simpler [6]. For the remainder of this section s_t is *pre-decision state variable* and s_t^a is *post-decision state variable*, in case of deterministic information, or using a forecast for the information s_t^a is the same as s_t.

3.3 Training procedures of DQN and VIPS

In this paper, we propose Value Iteration with Post-States (VIPS) as a method to solve TUSP. The main differences between VIPS and DQN are in the action selection. Specifically, VIPS either takes an exploratory action, or the greedy action with respect to the sum of the immediate reward, and the value of the next state. Note that the agent takes exploratory actions (even though it is a planning problem) for the function approximator to learn how to generalize across more of the reachable state-space. The target value is now based upon the VI Bellman backup rule [1] rather than the Q-learning update rule [9].

4 Experiment Setup

4.1 Instance generation

To test the performance of DRL model in various scenarios, 5,000 problem instances are generated for 4, 5, 6 and 7 trains, separately, resulting in a total of 20,000 problem instances. From these 20,000 problem instances, 1,000 are randomly withdraw as the test instance while the rest are used for training the DRL agents. The shunting yard studied in this work is 'de Kleine Binckhorst' which is one of the smallest yards of NS. In all the instances, there is at least one internal cleaning task assigned.

4.2 Neural network architecture

For both DQN and VIPS, Dense Neural Networks are constructed. The input to the neural network is a 1610 dimensional vector. The first and second hidden layers are fully connected layer with 256 and 128 units that use the ReLU activation function. The difference between DQN and VIPS on the nerual network architecture is in the output layer; DQN has a fully connected linear layer with 53 outputs (one for each action) while VIPS has a fully connected linear layer with 1 output. Other hyperparameters for DQN and VIPS are the same with the batch size of 128, memory length of 400,000, random exploration of 400,000 steps, discount factor of 0.99, and ADAM optimizer.

4.3 Off-policy action filtering

The main challenge of training in TUSP problem is that most actions (e.g. possible movements in the yard) are invalid at any given state. Therefore it is interesting to know what would be the effect of filtering on feasible actions while training the DRL agent to constrain its search space.

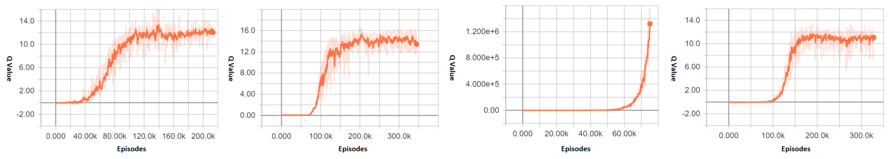
As mentioned earlier, there are many violations possible in the TUSP environment that will lead to immediate termination of the episode. Therefore, an off-policy rule is proposed to filter out actions that would result in immediate violations. It should be noted that the neural network will still estimate a value for the filtered action; hence theoretically it could be chosen as the optimal action (especially during early training).

5 Results

5.1 Convergence and problem solving capability

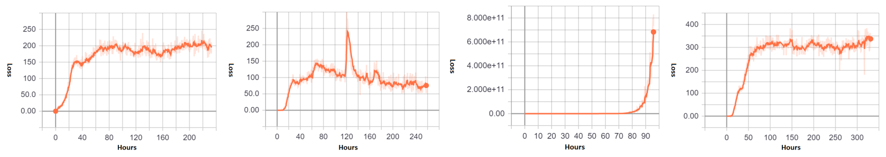
Figure 4 and Figure 5 show the convergence of Q values and losses of DQN and VIPS learned on either filtered actions or all actions. The light orange lines are the actual values and the dark orange lines show the exponential moving average of those values. It is very clear to see that when DQN learned on filtered actions, its Q values and losses explode after 60,000 episodes which is about 90 hours of execution. Because of this apparently bad result, it was terminated earlier compared to other models. VIPS, on the other hand, did converge when only learned on filtered actions. However, when learned on all actions, VIPS converges faster than DQN. VIPS learned on all actions and VIPS learned on filtered actions both converge around 40 hours of execution and therefore there is no clear benefit to learn only on filtered actions.

The capability to solve TUSP problems is essential for the evaluation of different models. All these four off-policy models are used to find shunting plans for 5 sets of test instances which are randomly withdrawn from 1,000 test instances. Their average performance is listed in Table 1. Models learned on all actions clearly have a better problem solving capability, and VIPS learned on all actions significantly outperforms the others. Figure 4 also supports the results



(a) VIPS learned on filtered actions (b) VIPS learned on all actions (c) DQN learned on filtered actions (d) DQN learned on all actions

Fig. 4. Q values of different models in episodes



(a) VIPS learned on filtered actions (b) VIPS learned on all actions (c) DQN learned on filtered actions (d) DQN learned on all actions

Fig. 5. Losses of different models in hours of execution

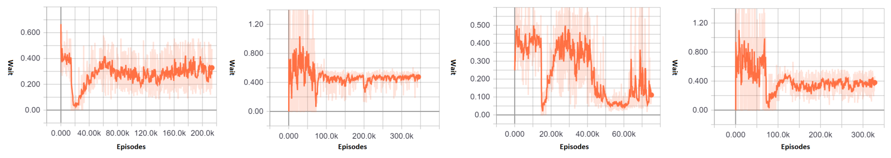
of Table 1. VIPS learned on all actions converges to a higher Q value compared to other methods which indicates a better strategy is learned. DQN learned on filtered actions, on the other hand, fails to converge and therefore fails to learn a proper strategy.

5.2 Monitoring on learning behaviour and violations

	Percentage of solved instances
VIPS, filtered actions	0.601±0.030
VIPS, all actions	0.905±0.037
DQN, filtered actions	0±0
DQN, all actions	0.782±0.016

Table 1. Average percentage of solved instances and standard deviations of different models on solving 5 sets of 200 test instances.

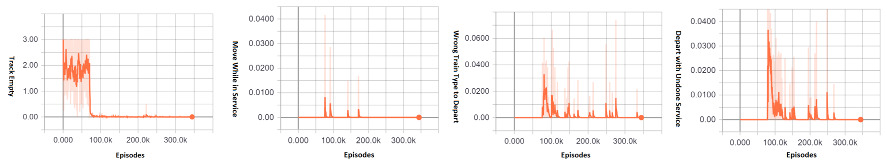
As discussed earlier, the main target of this work is to enable DRL agents to learn the concept of time in order to send the trains to service tracks for service at a proper moment. More specifically, the agent should generate a policy that trains should only move around when necessary and if there is no need or it is not a good moment to move, they should choose to wait on the tracks. Therefore, the learning behaviours of agents on learning the concept of waiting (which also implies the concept of time) are also monitored during the training process as given in Figure 6. After the random exploration period VIPS learned on all actions very quickly converges on learning the concept of waiting and it also learns to wait for often compared to other models. This suggests that it does not only solve more problems as shown in Table 1, but it also solves the problems with a better strategy.



(a) VIPS learned on filtered actions
(b) VIPS learned on all actions
(c) DQN learned on filtered actions
(d) DQN learned on all actions

Fig. 6. Wait actions of different models in episodes

In addition to monitoring on movement/wait actions, learning behaviours on violations are also monitored during training in this work. The monitoring on the violations and actions has helped a lot on the iterative adjustment on the design of TUSP environment and in understanding the decisions made by DRL agents. Figure 7 show some examples of the learning behaviours of VIPS learned on all actions on some violations. For instance Figure 7(d) shows that the agent did not learn to send the trains to service tracks at the beginning (after random exploration) and therefore trains often leave with undone service. After some episodes this violation starts to drop and is fully learned by the agent after 300,000 episodes. By monitoring various violations at the same time, we can better understand which are tasks are easy or difficult to learn by the agent, and ultimately to help us understand the bottleneck of the problem instance or the yard itself.



(a) Request train at an empty track
(b) Move a train under service
(c) Depart a wrong train type
(d) Depart a train with undone service

Fig. 7. Monitoring of different violations during the training (in episodes) of VIPS learned on all actions

6 Discussion and Conclusion

In this paper, we formulate the time constraints of service tasks within TUSP to enable deep reinforcement learning. Using this new formalization, we compare various learning strategies to evaluate the most suitable one for this application. The results show that by assigning extra triggers to agents at fixed time intervals, the agent accurately learns to send the trains to the service tracks in time to comply with the departure schedule.

Specific movements and violations are monitored during training to keep track on the capability of the agent on learning certain tasks effectively. These visualizations and monitoring of the agent behaviors have helped a lot in iteratively improving upon the design of the agent and the environment for the TUSP problem. We believe that this strategy of monitoring shall also be helpful for other real-world applications which are as complex as TUSP.

In this paper, we have focused on how many instances can be solved successfully, i.e., adhering to all (hard) constraints. However, when this is not possible, there are going to be trade-offs between different constraints. To empower the users of our planning system, we aim to model this using multiple objectives [7] in future work.

Acknowledgements

This research was in part supported by funding from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen".

References

1. Bellman, R.: A Markovian decision process. Journal of mathematics and mechanics pp. 679–684 (1957)
2. Boysen, N., Fliedner, M., Jaehn, F., Pesch, E.: Shunting yard operations: Theoretical aspects and applications. European Journal of Operational Research **220**(1), 1–14 (2012)
3. Howard, R.A.: Dynamic programming and Markov processes. (1960)
4. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M.A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature **518**, 529–533 (2015)
5. Peer, E., Menkovski, V., Zhang, Y., Lee, W.J.: Shunting trains with deep reinforcement learning. In: 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC). pp. 3063–3068. IEEE-SMC (2018)
6. Powell, W.B.: Approximate Dynamic Programming: Solving the Curses of Dimensionality: Second Edition. Wiley Series in Probability and Statistics (2011)
7. Roijers, D.M., Whiteson, S.: Multi-objective decision making. Synthesis Lectures on Artificial Intelligence and Machine Learning **11**(1), 1–129 (2017)
8. Sutton, R.S., Barto, A.G.: Introduction to Reinforcement Learning. MIT Press, Cambridge, MA, USA, 1st edn. (1998)
9. Watkins, C.J., Dayan, P.: Q-learning. Machine learning **8**(3-4), 279–292 (1992)