DIPLOMARBEIT

D*-Lite Algorithm vs Dyna $Q+$ Algorithm for Navigating Agents in a Railway Network

ausgeführt am

Institut für
Analysis und Scientific Computing
TU Wien

unter der Anleitung von

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Felix Breitenecker

und

Dipl.-Ing. Dr.techn. Martin Bicher

durch

Dominik Rothschedl
Matrikelnummer: 01425890
[redacted]

Wien, am 4. Februar 2022

Kurzfassung

Diese Arbeit untersucht Pfadfindungsalgorithmen, um den zeiteffizientesten Weg für Agenten in einem dynamischen Modell, das auf einem realen Eisenbahnnetz basiert, zu finden. Aufgrund von zeitabhängigen Eigenschaften, wie ausgelastete Bahnhöfe und somit der daraus resultierenden Verspätung, sind herkömmliche Pfadfindungsalgorithmen wie A* nicht zufriedenstellend.
Im Zuge dieser Fragestellung analysieren wir das Verhalten von zwei spezifischen Algorithmen: dem heuristischen D*-Lite Algorithmus und der Reinforcement Learning Methode Dyna $Q+$.
Der D*-Lite Algorithmus ist eine inkrementelle heuristische Suchmethode, die Informationen wiederverwendet, um in unbekanntem Terrain zielgerichtet zu navigieren. Dieser Algorithmus kann in seiner ursprünglichen Form blockierte Knoten mittels Überprüfung der Umgebung in jedem Schritt erkennen. Durch eine Anpassung bezieht er in unserem Fall auch in Zukunft mögliche Entwicklungen der Auslastung eines Knoten mit ein. Außerdem wird das Konzept des Reinforcement Learning vorgestellt, welches zu der Anwendung des sogenannten Dyna $Q+$ Algorithmus führt. Dieser lernt durch einen Markov Decision Process (MDP) die optimale Strategie, um sein Ziel zu erreichen. Er ermittelt für jeden Zustand die beste Aktion, genauer gesagt jene welche die größte Belohnung bringt, mittels einer Kombination aus bisher Gelerntem und Erkundung.
Einer der größten Schwächen von Dyna $Q+$ ist die Speicherung der Werte für jede mögliche Kombination eines Zustands und einer Aktion. Um dies zu umgehen, wird die Deep Variante von dieser Reinforcement Learning Methode eingeführt, welche einen Türöffner zur Theorie der Neuronalen Netze darstellt. Hierbei müssen nur die Modellarchitektur und die Parameter gespeichert werden.
Um die Unterschiede der Algorithmen aufzuzeigen, wird einerseits darauf geachtet, ob Pfade überhaupt gefunden werden können und andererseits die qualitativen Unterschiede aufgezeigt. Hierfür betrachten wir anhand zweier Aspekte ob globale oder lokale optimale Lösungen berechnet werden, nämlich die Lösung für den Agenten selbst und wie sich diese auf das Zugnetzwerk auswirkt. Beim ersten Aspekt werden wir die Länge des Pfades und die benötigte Dauer bis zum Erreichen des Ziels betrachten. Außerdem wird ein Augenmerk auf die Aktionen gelegt, die der Agent aufgrund von zeitlich blockierten Stationen nicht durchführen kann. Der zweite Aspekt, der die Unterschiede der Algorithmen aufzeigen soll, ist die Auswirkung des Agenten, der den berechneten Pfad folgt, auf das Schienennetz. Hier wird in Anzahl der blockierten Züge und Minuten der Verzögerung im gesamten System unterteilt.
Zusammenfassend beleuchtet diese Arbeit nicht nur die konkrete Aufgabenstellung, sondern versucht auch den allgemeinen Trend zur Nutzung künstlicher Intelligenzen zu begründen und sowohl die Vor- als auch Nachteile dieser Technologie in der theoretischen und praktischen Anwendung aufzuzeigen.

Abstract

This thesis studies path planning algorithms to find the most time efficient way for agents in a dynamic model based on a real-life railway network. Because of the dynamic nature of blocked nodes during various time steps while operating on an historical timetable, classical path finding algorithms such as A* are not sophisticated enough to allow agents to find a viable path through the system. Therefore, we analyze the behaviour of two specific algorithms: the heuristical D*-Lite algorithm and the Reinforcement Learning method Dyna $Q+$.
The D*-Lite algorithm is an incremental heuristic search method which reuses information from previous searches to navigate goal-directed in unknown terrain. It will be adapted to the specific setting with reusing blocked nodes. Furthermore, the concept of Reinforcement Learning is introduced, which leads to Dyna $Q+$. This one will find the optimal policy to achieve the goal by a Markov Decision Process (MDP). More precisely, it approximates the best action for each state in the environment through the right balance of exploitation and exploration, and draws its conclusions from the resulting outcomes. With respect to already taken actions in a state, Dyna $Q+$ will tackle the arising issues of the dynamically changing environment. To circumvent one of the biggest weaknesses of Dyna $Q+$, where the value of each state-action combination has to be stored in a matrix, Deep variants of Reinforcement Learning methods will be presented. Those are door openers to Neural Network theory, where only the model architecture and parameters need to be stored.
In order to show the differences between the algorithms, we consider on the one hand whether paths can be found at all and on the other hand the qualitative differences. For this purpose, we consider two aspects whether global or local optimal solutions are computed, namely the solution for the agent itself and how it affects the railway network. In the first aspect, we will look at the length of the path and the duration needed to reach the destination. We will also pay attention to the actions that the agent cannot perform due to time-blocked stations. The second aspect to show the differences of the algorithms is the impact of the agent following the calculated path on the railway network. Here it is divided into number of blocked trains and minutes of delay in the whole system.
This work does not only illuminate the concrete task but also tries to justify the general trend of using artificial intelligence and to shows both the advantages and disadvantages of this technology in the theoretical and practical application.

Keywords: Agent Based Model, heuristic path finding, D*-Lite, Reinforcement Learning, Q-Learning, Deep Reinforcement Learning, Deep Dyna Q, Neural Networks.

Danksagung

An dieser Stelle möchte ich mich allen voran bei Prof. Felix Breitenecker und Niki Popper bedanken. Aufgrund ihrer Lehrangebote auf der TU Wien wurde mein Interesse für mathematische Modellbildung und Simulation schon früh im Studium geweckt. Daraus resultierten nicht nur wissenschaftliche Arbeiten und Erfahrungen als studentischer Mitarbeiter, sondern seit kurzem auch die Möglichkeit meine berufliche Laufbahn in diesem Gebiet einzuschlagen.
Ganz besonders möchte ich Dominik Brunmeir hervorheben, der mein Interesse an diesem wissenschaftlichen Bereich noch verstärkt hat, für Fragen immer ein offenes Ohr hatte und mich zu jeder Zeit unterstützte, was in den durch COVID-19 geschuldeten turbulenten Zeiten nicht immer selbstverständlich war.
Ein besonderer Dank gebührt auch meiner Freundin und Korrekturleserin Alex, die mir in schwierigen Zeiten immer den Rücken stärkte und ohne die ich nicht das erreicht hätte, was ich bisher geschafft habe.
Außerdem gebührt auch meiner Familie, Susanne, Herbert, Katrin und Benedikt und bei meinen Großeltern ein außerordentlicher Dank, da sie mich nie unter Druck setzten, bei fehlgeschlagenen Prüfungen aufmunterten und auch vom Uni-Leben ablenkten.
Zu guter Letzt möchte ich auch meinen Freunden herzlich danken. Einerseits diejenigen, die ich auf der Universität kennen lernen durfte und mit deren Unterstützung ich die Herausforderungen des Studiums überwinden konnte. Andererseits die, die ich außerhalb des Studiums bereits kannte und mich motivierten früh in die Arbeitswelt einzusteigen.

Vielen Dank, Dominik

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am 4. Februar 2022

Dominik Rothschedl

Contents

1 Introduction

In this thesis two methods to navigate an agent through dynamic model based on a railway network will be introduced, namely the heuristic D*-Lite algorithm and the Reinforcement Learning method Dyna $Q+$. Due to the high complexity, the latter will result in the Deep version of the method, and we will scratch the surface of artificial intelligence in general and Neural Network in particular. With this specific example, the current trend to replace conventional methods with artificial intelligence (Hoog, 2017), (Yin et al., 2021), (Friesen and McLeod, 2014) can also be explained. However, it also shows some weaknesses of it, such as the need for certain hardware resources or non-trivial proofs of the convergence to global optimal solutions as well as the opaqueness of Neural Networks in general. In detail D*-Lite is easier to understand since it follows straightforward update and calculation rules, while even simple Neural Networks variants are non-transparent due to their architecture.

In the second chapter, the model will be presented. We will see how a railroad network with stations and tracks in between can be described by a graph. In this environment, trains (agents) move from one station to the next according to a certain schedule, which makes it an agent-based model (ABM). Since agent-based models are built from the bottom up, one of the most important properties can be observed: the emergence phenomenon. In our particular case this means, that the trains following simple rules and can cause congestion if at least one station reaches the maximum capacity. After introducing the model, two algorithms are presented to move an additional, but unscheduled agent from a starting point to a destination in this ABM with minimal disturbance of the existing schedule.
The first one is the heuristic approach in Chapter 3 which is called D*-Lite and follows the results of (Koenig and Likhachev, 2002) and (Koenig, Likhachev, and Furcy, 2004). This algorithm finds the shortest path between two nodes (stations) in an unknown environment by computing the shortest path to the destination in the starting station and navigating to the best neighbor. Then, the agent scans its immediate surroundings to see if a station has reached its maximum capacity of trains, uses the newly acquired information to calculate the shortest route to the destination and moves to the next 'best' station, where it again scans the surroundings, and so on until it reaches the destination station. D*-Lite uses the estimate $g(s)$ of the distance between a station s and the destination, the one-step look-ahead values $rhs(s)$, and a priority queue U to compute the shortest path. The latter stores nodes in the order of a key that depends on the two earlier mentioned values $g(s)$ and $rhs(s)$, only if $g(s) < rhs(s)$ or $g(s) > rhs(s)$ and will mandatory for proving the return of the shortest path. Additionally, the algorithm is applied to our use case with different scan ranges.
Since D*-Lite only detects new blocked nodes, but is not designed for such a dynamic environment where nodes can be unblocked after a certain time, an algorithm, named Dyna $Q+$, that can deal with these circumstances is presented in Chapter 4. This Reinforce-

ment Learning method follows the theory of (Sutton and Barto, 2018) and will find the optimal policy to achieve the goal in a Markov Decision Process (MDP). More precisely, it approximates the best action for each state in the environment through the right balance of exploitation and exploration, and draws its conclusions from the resulting outcomes. With respect to already taken actions in a state, Dyna $Q+$ will tackle the arising issues of the dynamically changing environment.
A major drawback of the Dyna $Q+$ algorithm is that each state and the possible actions are stored in a so-called Q-matrix, which poses storage and computational problems for applications with high dimension. Therefore, Deep variants of Reinforcement Learning methods are introduced, which are a door opener to Neural Network theory, where only the model architecture and parameters need to be stored. We will see that this is a powerful and easy to implement tool, which justifies the current trend, but also point out some drawbacks, such as the more complex proof to the optimal solution compared to D*-Lite, as well as the required deployment of special hardware components.

Path finding in dynamic environments is not only of high interest for researching, it can also be very well applied in practice. An example is the routing and scheduling of freight traffic in a railroad network. The implementation of a suitable path finding algorithm can be used to respond to ad hoc events, such as blocked routes and stations or weather changes. It will be shown that the algorithmically best solution is not always the best solution for real-world applications, since aspects such as storage, reuse and adaptability are relevant for business use. In the course of this work, we will observe that although the heuristic approach finds the more optimal solution, it only finds this one for one path at a time. This leads to the need of repeating the application to cover all possible paths. On the other hand, the Reinforcement Learning method, and by extension the Deep variant of it, offers a worse solution, but, given enough training, it computes a model that considers all possible paths and can be reused in any point of time. Therefore, as soon as this model is build for a business application with a certain network it will be more suitable for ad hoc requests.

2 The Model

Models describe real world problems in such a way that computer are able to solve them using certain methods. Depending on the description of these problems, the goals of the methods can be different. On the one hand, the present model describes the infrastructure of Austrian railways with the stations and the tracks in between. On the other hand, it describes the trains moving on this infrastructure, without disturbing other ones. Hence, the full breadth of Agent Based Modeling can be applied.

2.1 The Environment

Each railway network can be described as a graph $G = (V, E)$ where the set of nodes V and set of edges E represent stations or tracks between them. In our model each node is defined by the pair (s_{Lat}, s_{Lng}), which indicates its longitude and latitude. In our case, the edges $(s, s') \in E$ for vertices $s, s' \in V$ are undirected, thus $(s, s') = (s', s)$.

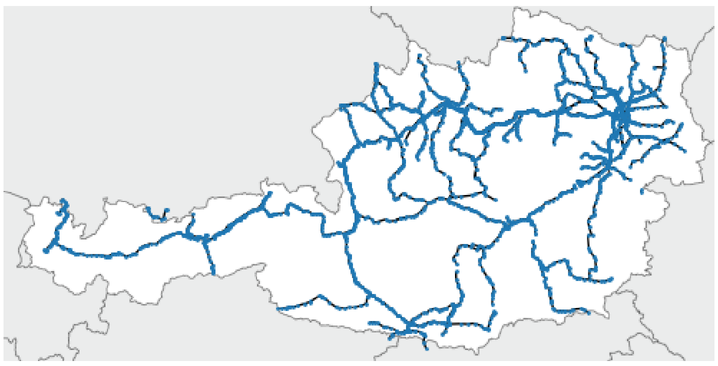

Figure 2.1: Austria's railway network considered according to the provided data. Map in the background is from (d-maps, n.d.)

As we can see in the created graph in Figure 2.1, it contains circles that can produce different paths in the different algorithms. Additionally, we can see some sections which are candidates for delays, for example in Tyrol or Salzburg. Another property of the agent based model, which is discussed in more detail in Section 2.2, is the maximum capacity of each station, which indicates the highest number of trains that can be placed in a given station.

Finally, we define the edge weights as the Euclidean metric (Kaltenbaeck, 2014)

$$d_E(s, s') := \sqrt{(Latitude_s - Latitude_{s'})^2 + (Longitude_s - Longitude_{s'})^2} \quad (s, s') \in E. \tag{2.1}$$

Due to the long distances, Vincenty's formula (Karney, 2013) is used to calculate (2.1) in meters, this gives an higher accuracy for this purpose.

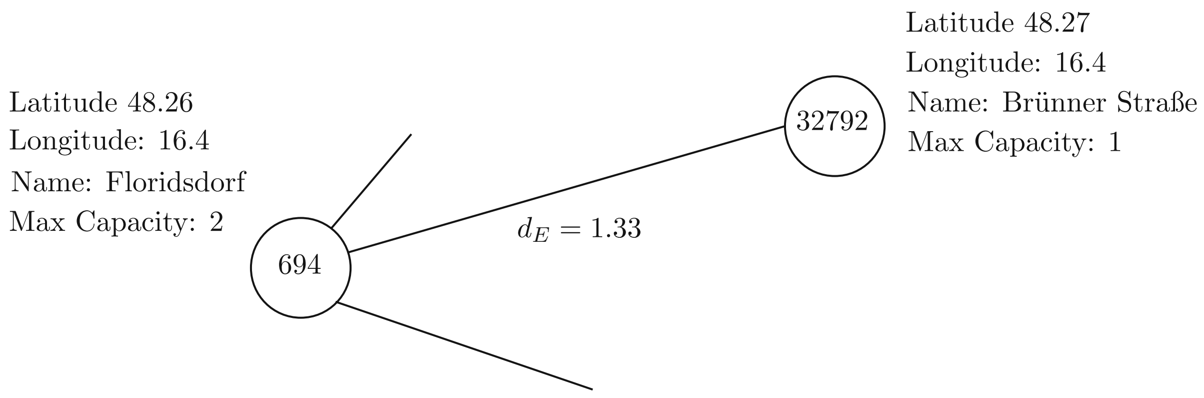

Figure 2.2: Example of two nodes and their stored information

2.2 Agent Based Model

Agent based applications are simulation techniques for real-world problems with eponymous agents as decision-making entities. Each of them evaluates its situation and chooses the next action according to a set of rules. Further, there exist a relation between the agents, which can be seen as communication. This leads to complex behavior patterns and provides valuable information about the dynamics of the real-world problems it emulates (Bonabeau, 2002). Therefore, the behavior of the model depends on the units and their communication at the microscopic level, but cannot be reduced to the respective individual system parts. The interactions between different, often simple parts of the model, we can observe complex behavior, which is called emergence and shows that the whole is more than the sum of its parts (Bonabeau, 2002).
A simple example of the emergent phenomenon can be found in (Bonabeau, 2002), where each member of the group has the nominate two individuals, one with label A, one with label B. After all choices are made, the persons should move in such a way that person A (the protector) is located between the interviewed person and person B. Immediately after that, people do not only move according to their choice, they are also taking into account the choice of other people. Thus, the agents will form in a tight knot, which is the emergent phenomenon in this example.
As we can see from the example, emergence is generated bottom-up in agent-based modeling. This draws the desired and characteristic property of building a dynamic and complex problem from simple rules.

2.2.1 The agents

In our case there are agents describing unique trains, which have a given track and time table, and one additional agent, which has to navigate through the environment without disturbing other agents. For consideration in this part of the thesis, we will focus on the agents with a defined timetable, before adding the latter.
As described in Section 2.2, agent based modeling allows the representation of real-world problems using self-deciding agents. In this thesis, the agents are assumed to be trains, which are only able to take two actions, namely, arrival and departure. The execution time for all these actions is stored in a time table, which is ordered by the trains, hence it is a discrete state event model .

Arrival	Train ID	Node	Departure	Workflow
24.06.2018 00:19	A	1073	24.06.2018 00:22	1
24.06.2018 00:22	A	524	24.06.2018 00:23	3
24.06.2018 00:23	A	1449	24.06.2018 00:24	5
24.06.2018 00:22	B	524	24.06.2018 00:23	4
24.06.2018 00:23	B	1656	24.06.2018 00:25	6
24.06.2018 00:21	C	1449	24.06.2018 00:25	2
24.06.2018 00:25	C	1656	24.06.2018 00:26	7

Table 2.1: Example of the timetable which stores actions of the trains. Each action stores the information about the train, train station and the time of arrival and departure. The first action is the one with the minimum arrival time, next, with the next smallest number and so on. If there are more actions with the same arrival time, they will processed top-down. The column Workflow displays the taken actions in the order they were executed.

Although, the action space is that limited, it has a huge impact on the model itself. Since, slots can be blocked in stations or nodes, a not executed action influences a lot of other actions that should be executed afterwards. If the maximum capacity of a node is reached and a new train should arrive, all of its actions will be delayed until the spot will be released.

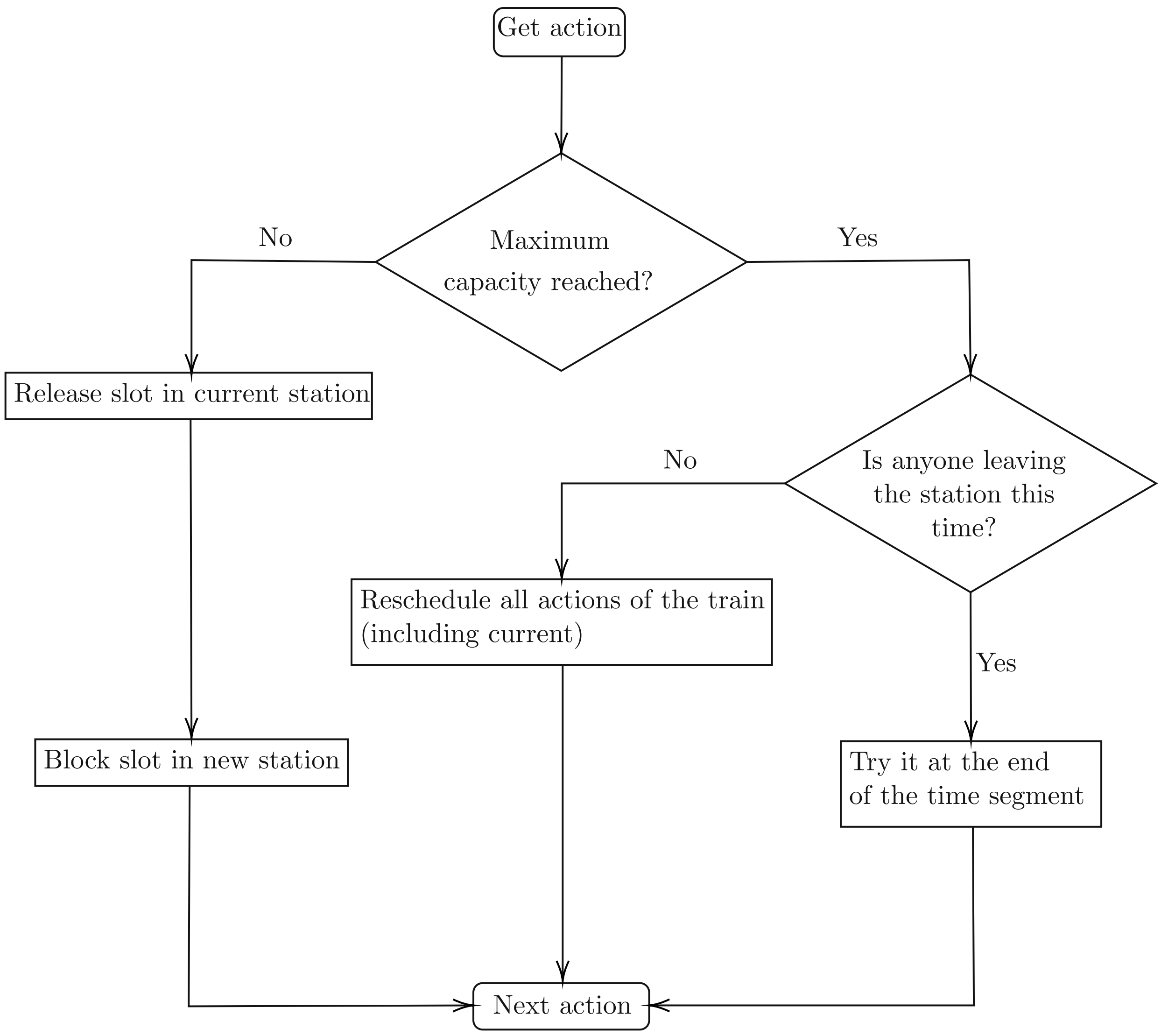
Get action
Maximum capacity reached?
No
Yes
Release slot in current station
Is anyone leaving the station this time?
No
Reschedule all actions of the train (including current)
Yes
Block slot in new station
Try it at the end of the time segment
Next action

Figure 2.3: Chart displays the actions of each train. If maximum capacity in station where a train will arrive is reached and another train leaves the station in future, the current action and all subsequent actions of the train will be rescheduled. The current one will be executed the next time a slot is released, successive actions relatively. On the other hand, if the capacity is full but another train is departing at the same time, the action will be attempted after all other actions have been performed in that time step.

We can observe an emergent behaviour, namely congestion. According to the policy described in Figure 2.3 any agent who cannot perform his action will remain in the current station until the desired one gets a free slot. This leads to the fact that in this station again one slot is occupied and thus another agent could not carry out its actions. All together this leads to the desired congestion.

Let us take a look at a simple example. A train A in station x, which is at full capacity, would like to arrive in station y at 9 AM, but the maximum capacity is reached in y and the next time a slot will be free is $9:05$ AM. As a consequence, all actions, including the one just tried to execute, of the agent will be delayed by five minutes and it stays in x for this occupation. At $9:03$ AM another agent B will arrive in x from station z, but no free slot is available in x. Hence, all actions of B will be delayed by two minutes, which is similar to the difference to the next time A will try to arrive in y. If there are any other agents, which will arrive in x or z, they will also be delayed and show us the phenomenon, how one agent can cause congestion.

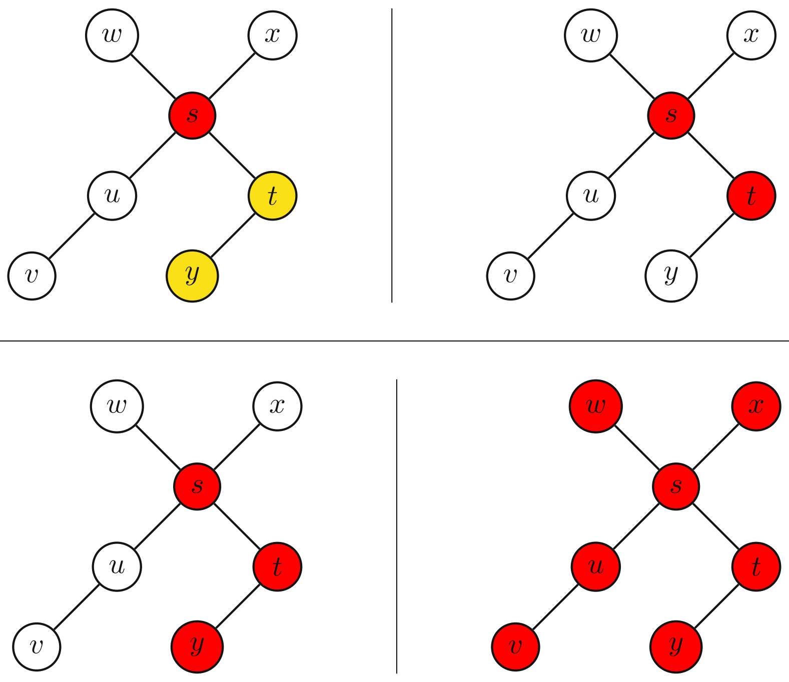

Figure 2.4: Another example to illustrate the phenomenon of congestion, where red nodes indicate full capacity, while yellow means that there is at least one train in the node, but it does not reach maximum capacity. The initial position (top left graph) is that all nodes have a maximum capacity of two and each train want to reach node s. Additionally s is fully utilized, while t and y each contain only one train. If now the train from node t wants to move to s, this action cannot be executed since lack of free places. Thus, only the train from y can move to t, which means that this node also reaches the maximum capacity (top right graph). All other trains that want to reach s via this route can only go to node y until it is full (bottom left graph). Since not only these, but also all other routes are used, the congestion can also be observed on these routes. (bottom left graph).

3 D*-Lite Algorithm

In this chapter we introduce the common algorithm for navigating through unknown terrain - the D*-Lite Algorithm. It is based on Lifelong Planning A* (LPA*) (Koenig and Likhachev, 2002), which uses the heuristic knowledge of the approximated goal distance to calculate the shortest path from a vertex s_{start} to s_{goal} by starting at the first vertex and moving to the "best" successor.
Consider a graph, where at least one path exists between s_{start} and s_{goal}. An agent (robot) starts at s_{start} with the task to reach a goal by the shortest path. To fulfill this, in each state the shortest path to the goal will be computed, which yields to an update of the values, as soon as an unknown blocked cell is found. Lets take an example for a better understanding. In Figure 3.1 we see a graph with known blocked vertices and after a few steps, the agent discovers a new blocked one, which is in the previous calculated shortest path. Hence, some values have to be updated and the shortest path changes. In this example $s_{start} = s_0$ and $s_{goal} = s_9$.

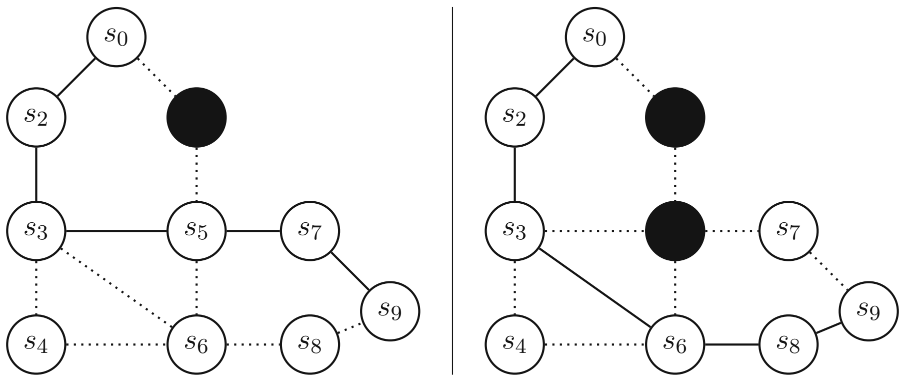

Figure 3.1: In this example all edge costs are one. The start node is s_0 and D*-Lite Algorithm calculates a path to the goal s_9. The black vertex is known to be blocked, so the shortest path is $s_0 - s_2 - s_3 - s_5 - s_7 - s_9$ but after a few steps, a new blocked vertex s_5 is discovered. After recalculating, the shortest path is $s_0 - s_2 - s_3 - s_5 - s_7 - s_9$.

In the following the D*-Lite will be introduced. Moreover, the similarities to the well known heuristic search algorithm A* will be pointed out below. This section follows the results of (Koenig and Likhachev, 2002) and proofs of (Koenig, Likhachev, and Furcy, 2004)

3.1 The algorithm

Before discussing the pseudo algorithm of D*-Lite in detail, we take a closer look at each node and on the related information stored. Assume s_{start} is the starting vertex and V the set of vertices. For each node $s \in V$ exists a shortest path (distance) $g^*(s)$ to the goal point s_{goal} and the edge weights $c(s, s')$, where s' is adjacent to s. It holds that

$$0 < c(s, s') < \infty.$$

Let $Succ(s)$ be the set of successors and $Pred(s)$ denotes the set of predecessors of $s \in V$. Since D*-Lite is a heuristic search algorithm, we need to specify its properties. First, the heuristic should be non-negative, satisfy triangular inequality, and be admissible

$$\begin{aligned} &h(s, s') \leq 0 \\ &h(s, s') \leq c^*(s, s') \\ &h(s, s'') \leq h(s, s') + h(s', s'') \qquad \forall s, s', s'' \in V. \end{aligned} \tag{3.1}$$

Therein, $c^*(s, s')$ denotes the costs of the shortest path from s to s'. If the vertices are adjacent, then $c^*(s, s') = c(s, s')$.

Algorithm A.1 D*-Lite

```
procedure CalculateKey(s)
    $return[min(g(s), rhs(s)) + h(s_{start}, s) + k_m; min(g(s), rhs(s))]$
procedure Initialize( )
    $U = \emptyset$
    $k_m = 0$
    for $s \in S$ do
        $rhs(s) = g(s) = \infty$
    $rhs(s_{goal}) = 0$
    $U.Insert(s_{goal}, CalculateKey(s_{goal}))$
procedure UpdateVertex(u)
    if $u \neq s_{goal}$ then
        $rhs(u) = min_{s \in Succ(u)}(g(s) + c(s, u))$
    if $u \in U$ then
        $U.Remove(u)$
    if $g(u) \neq rhs(u)$ then
        $U.Insert(u, CalculateKey(u))$
procedure ComputeShortestPath( )
    while $U.TopKey() < CalculateKey(s_{start})$ $or$ $rhs(s_{start}) \neq g(s_{start})$ do
        $k_{old} = U.TopKey()$
        $u = U.Pop()$
        if $k_{old} < CalculateKey(u)$ then
            $U.Insert(u, CalculateKey(u)$
        else if $g(u) > rhs(u)$ then
            $g(u) = rhs(u)$
            for $s \in Pred(u)$ do
                $UpdateVertex(s)$
        else
            $g(u) = \infty$
            for $s \in Pred(u) \cup \{u\}$ do
                $UpdateVertex(s)$
procedure Main( )
    $s_{last} = s_{start}$
    $Initialize()$
    $ComputeShortestPath()$
    while $s_{start} \neq s_{goal}$ do
        $s_{start} = argmin_{s \in Pred(s_{start})}(g(s) + c(s_{start}, s))$
        Move to $s_{start}$
        Scan graph for changed edge costs
        if any edge costs changed then
            $k_m = k_m + h(s_{last}, s_{start})$
            $s_{last} = s_{start}$
            for $(u, v) \in E$ $with$ $changed$ $costs$ do
                $Update$ $c(u, v)$
                $UpdateVertex(u)$
            $ComputeShortestPath()$
```

For each vertex there exists an estimation $g(s)$ for the distance between the vertex itself and the goal. Additionally, the rhs-value gives a second estimation of the distance. These are one-step look-ahead values of $g(s)$ for all $s \in V$ which leads to a better information in comparison to the g-values.

Regarding to Algorithm A.1 the procedure *Initialize()* sets the rhs-value of the goal state to zero and all others to infinity. The values of the vertices will be updated if the procedure *UpdateVertex(u)* is called, what can be seen at the update rule in line 12. Hence, the rhs-value satisfies

$$rhs(s) = \begin{cases} 0 & , \ s = s_{goal} \\ min_{s' \in Succ(s)}(g(s') + c(s', s)) & , \ otherwise \end{cases} \tag{3.2}$$

for all $s \in V$.
If the g-value of a vertex is equal to the rhs-value, we call the vertex locally consistent, in case of inequality, locally inconsistent. Local consistent in all nodes leads to the equality of g-values to the respective start values. Hence, the shortest path to s_{goal} from any vertex $s \in V$ can be computed by starting at s and taking the path to the successor $s' \in Succ(s)$, which minimize $g(s') + c(s', s)$ until the end point is reached.

Since the edge costs will be changed, not all vertices in D*-Lite will be locally consistent anymore. This results in considering the heuristics for focusing the search and updates g-values, which are relevant for the calculation of the shortest path. To store these relevant vertices, a priority queue U, which contains pairs of states and is sorted by their priority, is required.

Lemma 3.1.1. *The priority queue U contains all locally inconsistent vertices in each timestep.*

Proof. After initializing the priority queue in *Initialize()* the only locally inconsistent vertex s_{goal} is contained, since the other ones has g and rhs value infinity. The local consistency of the nodes changes if the g-value or rhs-value changes, which happens in the procedures *UpdateVertex(u)* and *ComputeShortestPath()*. It first updates the rhs-value so that Equation (3.2) is fulfilled and immediately removes the vertex if it is locally consistent or add it to the priority queue if not. In *ComputeShortestPath()* the vertex s with the smallest priority will be deleted from U and if it is locally overconsistent the g-value will be set equal to the rhs-value. Hence, s is locally consistent and not in the priority queue. But if s is locally underconsistent, *UpdateVertex(s)* will be called. Therein, the rhs-value is updated and if it is locally inconsistent, s is added to the priority queue again. None of these steps will be executed, when the priority key of s, is not the current one. Then, the vertex, which is still locally inconsistent, will be reinsert to U with an updated key.
Since we add locally inconsistent and delete locally consistent vertices,, the priority queue contains all locally inconsistent ones at each timestep. □

Now, we take a closer look on the keys in the priority queue U, which will be calculated in procedure *CalculateKey(s)* in Algorithm A.1. Therein a key is defined as a vector with two components $k(s) = [k_1(s); k_2(s)]$ where

$$\begin{aligned} k_1(s) &= min(g(s), rhs(s)) + h(s_{start}, s) + k_m \;\; and \\ k_2(s) &= min(g(s), rhs(s)). \end{aligned} \tag{3.3}$$

In (3.3), an unknown variable k_m appears that preserves the order of the priority queue after the agent moves from a node s to s_0 and the edge cost changes are detected. Since the keys are calculated if a vertex is inserted in the priority queue, the priority of each vertex $u \in U$ is equal to $k(u)$. Keys are compared according to a lexicographic ordering, i.e,

$$\begin{aligned} k(s) \leq k'(s) \iff & k_1(s) < k'_1(s) \;\; or \\ & k_1(s) = k'_1(s) \;\; and \; k_2(s) \leq k'_2(s). \end{aligned}$$

3.2 Termination and Correctness

To show that D*-Lite is correct and terminates, we will focus on local consistency of vertices, which are selected for expansion between line *23* and *30* in Algorithm A.1. First, we need to distinguish between keys before and after a vertex $u \in V$ is expanded in line *23*. So, $k_{b(u)}(s)$ and $k_{a(u)}(s)$ donating the keys of vertex $s \in V$ before and after the expansion of u. The first property to verify is, that the change from locally consistency to locally inconsistency of a vertex s, caused by select u for expansion in line *23*, will result in a higher key for s than for u after selection of u.

Lemma 3.2.1. *Assume vertex u is selected for expansion in line 23 with key $k_{b(u)}(u)$. Another vertex s is locally consistent before execution, but locally inconsistent after that, then*

$$k_{a(u)}(s) > k_{b(u)}(u)$$

Proof. Assume vertex u and s as required. Since u is locally inconsistent and s is locally consistent, we may assume that s and u are different.
Regarding to the definition, local consistency only changes if the rhs or g-value changes. According to (3.2) the rhs-value changes if the g-value or edge cost of the successors is changing. As we can see in Algorithm A.1, only the g-value of the, for expansion selected, vertex will change, if it is locally inconsistent. For calculating the change of the respective rhs-, g-values and keys, we have to distinguish between locally over- and underconsistency. Assume u is locally overconsistent. Therefore, the condition in line *23* of Algorithm A.1 is fulfilled, which leads to

$$g_{a(u)}(u) = rhs_{b(u)}(u) < g_{b(u)}(u). \tag{3.4}$$

Due to the requirements, s is locally consistent before and locally inconsistent after the expansion of u. As a consequence, u affects the consistency, more precisely the rhs- or g-value, of the other vertex. The latter does not depend on the predecessors, but as the rhs-values are defined by Equation (3.2), this value will change if u minimize the term $g(s') + c(s, s')$ for all $s' \in Succ(s)$,

$$rhs_{a(u)}(s) = g_{a(u)}(u) + c(s, u).$$

This together with Equation (3.4) will lead to a decreased rhs-value after selection of u

$$\begin{aligned} rhs_{a(u)}(s) &= g_{a(u)}(u) + c(u,s), \\ &< min_{s' \in Succ(s)}(g_{b(u)}(s') + c(s,s')), \\ &= rhs_{b(u)}(s). \end{aligned}$$

The local consistency of s before selection of u for expansion yields

$$rhs_{a(u)}(s) < rhs_{b(u)}(s) = g_{b(u)}(s) = g_{a(u)}(s). \tag{3.5}$$

Now, we can take a look on the keys and formulate the intended inequality. We start with the first component of those

$$\begin{aligned} k_{a(u)_1}(s) &\overset{(3.3)}{=} min(g_{a(u)}(s), rhs_{a(u)}(s)) + h(s) + k_m \\ &\overset{(3.5)}{=} rhs_{a(u)}(s) + h(s_{start}, s) + k_m \\ &\overset{(3.2)}{=} g_{a(u)}(u) + c(s,u) + h(s_{start}, s) + k_m. \end{aligned}$$

Since edge costs are positive and heuristics fulfill Equation (3.1), we obtain

$$\begin{aligned} k_{a(u)_1}(s) &\geq g_{a(u)}(u) + h(u) + k_m \\ &\overset{(3.4)}{=} rhs_{b(u)}(u) + h(s_{start}, u) + k_m \\ &\overset{(3.4)}{=} min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, u) + k_m \\ &\overset{(3.3)}{=} k_{b(u)_1}(u). \end{aligned} \tag{3.6}$$

Keys are compared in lexicographic ordering, which allows us to formulate the desired inequality, if strictness is fulfilled in (3.6). In case of equality, we have to investigate the second entry of the regarding keys

$$\begin{aligned} k_{a(u)_2}(s) &\overset{(3.3)}{=} min(g_{a(u)}(s), rhs_{a(u)}(s)) \\ &\overset{(3.5)}{=} rhs_{a(u)}(s) \\ &\overset{(3.2)}{=} g_{a(u)}(u) + c(s,u) \\ &> g_{a(u)}(u) \\ &\overset{(3.4)}{=} rhs_{b(u)}(u) \\ &\overset{(3.4)}{=} min(g_{b(u)}(u), rhs_{b(u)}(u)) \\ &\overset{(3.3)}{=} k_{b(u)_2}(u). \end{aligned} \tag{3.7}$$

Due to Equation (3.6) and (3.7) we receive

$$k_{a(u)}(s) > k_{b(u)}(u).$$

After obtaining the locally overconsistent case, the locally underconsistent one is partly the same. Here, the condition in line *23* in Algorithm A.1 is not fulfilled which leads to line *27* and the g-value of u is set to infinity. This will only affect the rhs-value of s if

$$rhs_{b(u)}(s) = g_{b(u)}(u) + c(s, u).$$

Hence, the increasing of $g_{a(u)}(u)$ to infinity results in

$$rhs_{a(u)}(s) > rhs_{b(u)}(s) = g_{b(u)}(s) = g_{a(u)}(s), \tag{3.8}$$

since s was locally consistent before selecting u. In the following, we take a look on the first component of the key

$$\begin{aligned} k_{a(u)_1}(s) &\overset{(3.3)}{=} min(g_{a(u)}(s), rhs_{a(u)}(s)) + h(s_{start}, s) + k_m \\ &\overset{(3.8)}{=} g_{a(u)}(s) + h(s_{start}, s) + k_m \\ &\overset{(3.8)}{=} rhs_{b(u)}(u) + h(s_{start}, s) + k_m \\ &\overset{(3.2)}{=} g_{b(u)}(u) + c(s, u) + h(s_{start}, s) \\ &\geq g_{b(u)}(u) + h(u) \\ &\overset{(3.8)}{=} min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, u) + k_m \\ &\overset{(3.3)}{=} k_{b(u)_1}(u). \end{aligned}$$

With the same argumentation like in the case above with the second components of the keys, we proved the lemma. □

In the next Lemma we investigate that the key of a, for expansion selected, locally overconsistent vertex remains the same after the selection, but the local consistency will change.

Lemma 3.2.2. *If a locally overconsistent vertex u with key $k_{b(u)}(u)$ is selected for expansion, then it is locally consistent the next time the line 20 will be executed and $k_{b(u)}(u) = k_{a(u)}(u)$.*

Proof. Assume u fulfills the requirements. Hence $g_{b(u)}(u) > rhs_{b(u)}(u)$, which yields to line *23* in Algorithm A.1 and local consistency

$$g_{a(u)}(u) = rhs_{a(u)}(u) = rhs_{b(u)}(u). \tag{3.9}$$

Now, we have to verify, that line *26* does not change the behaviour of the vertex anymore, if u is a predecessor of itself. According to (3.2), if u is the end vertex, it has a rhs value of zero. Assume that u is not the end vertex, then there exists a $w \in Succ(u)$ with

$$rhs_{b(u)}(u) = g_{b(u)}(w) + c(u, w). \tag{3.10}$$

We note that w cannot be the same vertex as u, since

$$rhs_{b(u)}(u) = g_{b(u)}(u) + c(u, u) \geq g_{b(u)}(u)$$

and this is a contradiction to local overconsistency. Combining (3.10) and (3.9) yields

$$\begin{aligned} g_{a(u)}(u) + c(u,u) &= rhs_{b(u)}(u) + c(u,u) \\ &> rhs_{b(u)}(u) \\ &= g_{b(u)}(w) + c(u,w) \\ &= g_{a(u)}(w) + c(u,w). \end{aligned}$$

Insertion into (3.2) leads to local consistency

$$\begin{aligned} rhs_{a(u)}(u) &= min(g_{a(u)}(w) + c(u,w), g_{a(u)}(u) + c(u,u)) \\ &= g_{a(u)}(w) + c(u,w) \\ &= rhs_{b(u)}(u) \\ &= g_{a(u)}(u). \end{aligned}$$

Now, we proof the equality of the keys before and after the selection of u.

$$\begin{aligned} k_{a(u)}(u) &= [min(g_{a(u)}(u), rhs_{a(u)}(u) + h(s_{start}, u) + k_m; min(g_{a(u)}, rhs_{a(u)}(u)] \\ &= [rhs_{a(u)}(u) + h(s_{start}, u) + k_m; rhs_{a(u)}(u)] \\ &= [rhs_{b(u)}(u) + h(s_{start}, u) + k_m; rhs_{b(u)}(u)] \\ &= [min(g_{b(u)}(u), rhs_{b(u)}(u) + h(s_{start}, u) + k_m; min(g_{b(u)}, rhs_{b(u)}(u)] \\ &= k_{b(u)}(u) \end{aligned}$$

□

In Lemma 3.2.1 we discussed vertices, which change local consistency after selection of another vertex for expansion. The next lemma shows the change of the keys, if the vertices remain locally inconsistent.

Lemma 3.2.3. *Assume vertex u is selected for expansion in line 23 in Algorithm A.1 with key $k_{b(u)}(u)$. If a vertex s is locally inconsistent at this point and the next time line 20 is executed, then*

$$k_{a(u)}(s) \geq k_{b(u)}(u)$$

Proof. Assume u and s as demanded. Due to the fact that the function *U.pop* returns the locally inconsistent vertex with smallest key, it holds

$$k_{b(u)}(s) \geq k_{b(u)}(u).$$

Now, we evaluate how the key of s is changing. First, we assume that $k_{a(u)}(s)$ does not change, then

$$k_{a(u)}(s) = k_{b(u)}(s) \geq k_{b(u)}(u). \tag{3.11}$$

If the key of s changes and $s = u$, then it was locally underconsistent. Lemma 3.2.2 shows that local overconsistency leads to local consistency the next time the line *20* is executed.

This violates our assumptions and hence, s was locally underconsistent. Therefore, we obtain that the condition of line *23* is not fulfilled and the g-value of $u = s$ is set to infinity. The rhs-value of this vertex will only change, if s is a predecessor of itself. Formulating the last two sentences as equations yields

$$\begin{aligned} \infty = g_{a(u)}(u) &\geq g_{b(u)}(u) \\ rhs_{a(u)}(u) &\geq rhs_{b(u)}(u) \end{aligned}$$

and

$$k_{a(u)}(s) = k_{a(u)}(u) \geq k_{b(u)}(u)$$

The third case is, that the key of s changes, $s \neq u$ and u was locally overconsistent. Then, line *23* will set

$$g_{a(u)}(u) = rhs_{b(u)}(u) < g_{b(u)}(u). \tag{3.12}$$

Equivalent to the proof of Lemma 3.2.1 the rhs value of s only changes if u is the minimizer of (3.2)

$$\begin{aligned} rhs_{a(u)}(s) &= g_{a(u)}(u) + c(s,u) \\ &\overset{(3.12)}{=} rhs_{b(u)}(u) + c(s,u) \\ &\overset{(3.12)}{=} min(g_{b(u)}(u); rhs_{b(u)}(u)) + c(s,u). \end{aligned}$$

As a consequence,

$$rhs_{a(u)}(s) \geq min(g_{b(u)}(u); rhs_{b(u)}(u)). \tag{3.13}$$

Consider the first component of the key of s

$$k_{a(u)_1}(s) = min(g_{a(u)}(s), rhs_{a(u)}(s)) + h(s_{start}, s) + k_m.$$

If $g_{a(u)}(s)$ is the minimum and we know that the heuristic fulfills Equation (3.1), it follows that

$$\begin{aligned} k_{a(u)_1}(s) &= g_{a(u)}(s) + h(s_{start}, s) + k_m \\ &= g_{b(u)}(s) + h(s_{start}, s) + k_m \\ &\geq min(g_{b(u)}(s); rhs_{b(u)}(s)) + h(s_{start}, u) + k_m \\ &= k_{b(u)_1}(s) \\ &\overset{(3.11)}{\geq} k_{b(u)_1}(u) \end{aligned}$$

If $rhs_{a(u)}(s)$ is the minimum and we know h fulfills (3.1), we obtain

$$\begin{aligned} k_{a(u)_1}(s) &= rhs_{a(u)}(s) + h(s_{start}, s) + k_m \\ &\overset{(3.13)}{\geq} min(g_{b(u)}(u); rhs_{b(u)}(u)) + h(s_{start}, u) + k_m \\ &= k_{b(u)_1}(u). \end{aligned}$$

The last two equations hold also for the second entry of the key, so

$$k_{a(u)}(s) \geq k_{b(u)}(u).$$

It remains to work on the case when the key of s changes, $s \neq u$ and u is locally underconsistent. Thus, the g-value of s does not change, but the rhs-value. Therefore, u is a predecessor of s, where the g-value is set to infinity in line *28*. This will yield to non decreasing rhs-values, since the other g-values remain at the same value. Therefore, we obtain

$$\begin{aligned} k_{a(u)}(s) &= [min(g_{a(u)}(s), rhs_{a(u)}(s)) + h(s_{start}, s) + k_m; min(g_{a(u)}(s), rhs_{a(u)}(s))] \\ &\geq [min(g_{b(u)}(s), rhs_{b(u)}(s)) + h(s_{start}, s) + k_m; min(g_{b(u)}(s), rhs_{b(u)}(s))] \\ &= k_{b(u)}(s) \\ &\overset{(3.11)}{\geq} k_{b(u)}(u), \end{aligned}$$

which finishes the proof. □

In the next theorem we will show one of the major similarities to the A* Algorithm. The f-values form the A* Algorithm of the expanded vertices in the latter are monotonically non-decreasing over time. In D*-Lite the keys of the expanded vertices are non-decreasing.

Theorem 3.2.1. *The keys of the, in line 23 for expanding chosen, vertices are monotonically non-decreasing.*

Proof. Recall the fact, that all vertices, which are candidates for expansion, are elements of U. According to Lemma 3.1.1 these are locally inconsistent. Assume that u is chosen for expanding with key $k_{b(u)}(u)$. The next time line *20* is executed, vertex s will be chosen. If those vertex was locally consistent before selecting u and became locally inconsistent during expansion, it follows that

$$k_{a(u)}(s) > k_{b(u)}(u)$$

according to Lemma 3.2.1. If s was locally inconsistent before and after expansion, then

$$k_{a(u)}(s) \geq k_{b(u)}(u)$$

regarding Lemma 3.2.3. □

In the next theorems we will discuss vertices which are locally consistent at the time, line *18* is executed. Theorem 3.2.2 ensures, that locally consistent vertices stay locally consistent as long as the condition in line *18* is not fulfilled. In Theorem 3.2.3 a for expansion selected locally overconsistent vertex will also stay locally consistent until the condition is not fulfilled anymore.

Theorem 3.2.2. *Let k be the smallest priority key of U which we get from $U.TopKey(U)$ in line 18. Additionally, we assume, that a vertex s is locally consistent at this time with*

$$k(s) \leq k.$$

Then, s remains locally consistent, until the condition in line 18 is not fulfilled anymore.

Proof. Let U be empty. Hence, all vertices are locally consistent, as U contains every locally inconsistent vertices, according to Lemma 3.1.1. So, $rhs(s_{start}) = g(s_{start})$ and $k = [\infty; \infty]$. The latter equality gives us the information, that $k \geq k(s_{start})$, which means the condition in line *18* is not fulfilled and the theorem is trivial.
In case that U is not empty, the proof is done by contradiction. Assume that s is locally consistent with key $k(s) \leq k$ and becomes locally inconsistent after expanding a vertex $u \neq s$, since locally inconsistent vertices can be candidates for expansion. According to Lemma 3.2.1 and Theorem 3.2.1 the equations $k_{a(u)}(s) > k_{b(u)}(u)$ and $k_{b(u)}(u) \geq k$ hold respectively. Hence,

$$k_{a(s)}(u) > k \geq k(u)$$

and further

$$\begin{aligned}[min(g_{a(u)}(s), &rhs_{a(u)}(s)) + h(s_{start}, s) + k_m; min(g_{a(u)}(s), rhs_{a(u)}(s))] \\ &> [min(g(s), rhs(s)) + h(s_{start}, s) + k_m; min(g(s), rhs(s))] \\ &= [g(s) + h(s_{start}, s) + k_m; g(s)].\end{aligned}$$

This follows according to the definition of the keys (3.3) and due to the fact that s is locally consistent. It follows that $g_{a(u)}(s) > g(s)$ which is a contradiction since there was no possibility to change the g-value of s. □

Theorem 3.2.3. *A locally overconsistent vertex u which is selected for expansion in line 23 is locally consistent the next time the condition in line 18 is queried. Furthermore, it remains in this property until the condition is not fulfilled.*

Proof. Assume u is a locally overconsistent vertex, which is selected for expansion. According to Lemma 3.2.2 it is locally consistent after expansion and $k_{a(u)}(u) = k_{b(u)}(u)$. Let $k_{b(u)}$ be the smallest key in priority queue U before and $k_{a(u)}$ after expansion of u. As shown in Theorem 3.2.1 the keys of the, for expanding selected, vertices are monotonically non-decreasing. Thus, $k_{a(u)} \leq k_{b(u)}$. Since u is the selected node in line *20*, $k_{b(u)}(u) = k_{b(u)}$ holds. Putting all the information together, u is locally consistent with key

$$k_{a(u)}(u) = k_{b(u)} \leq k_{a(u)},$$

which fulfills the requirement of Theorem 3.2.2 and yields to local consistency of u until the while loop breaks. □

The next result gives us the information, how often a vertex is selected for expansion, if we simplify the condition in line *18*.

Lemma 3.2.4. *If we change the condition in line 18 to "U is not empty", then each vertex will be selected for expansion at most twice, more accurate at most once when it is locally over- and underconsistent respectively. After termination of ComputeShortestPath() the g-values are equal to their respective start distances.*

Proof. Assume that we have changed the condition in line *18* like required. Consequently, U will be empty, if and only if there are no more locally inconsistent vertices anymore.
To prove, that each vertex is expanded at most twice, we have to split the cases where the node is selected for expansion. First, consider the vertex is locally overconsistent. As a result of Theorem 3.2.3, it is locally consistent after expansion and remains this property until the code terminates. Hence, a locally overconsistent vertex will only be selected for expansion once. If it is locally underconsistent, the g-value will be set to infinity in line *28*. Thus, the node is either locally consistent or locally overconsistent after expansion, where the latter leads to the previous case. Hence, every vertex will be selected for expansion at most twice.
To prove the second statement, recall that $g(s) = rhs(s)$ and (3.2) holds, which is equal to their start distances. □

The next lemma is about finding a shortest path from any vertex to s_{goal}.

Lemma 3.2.5. *By execution of line 18, let k be the smallest key in priority queue U and a vertex u locally consistent with $k(u) \leq k$. Then, the g-value of u equals their start distance. Additionally, we can trace back a shortest path from u to s_{goal}, by starting at u and moving to the successor s', that minimizes equation $g(s') + c(s, s')$, where s is the current node in the path.*

Proof. If the priority queue is empty, we are done by applying Lemma 3.2.4. So, we assume, that U is not empty.
Assume that u is locally consistent when line *18* is executed with key

$$k(u) \leq k. \tag{3.14}$$

First, we consider the case that $g(u) = \infty$. Since u is locally consistent, it holds that

$$g(u) = rhs(u) = \infty,$$

further $k(u) = [\infty; \infty]$. According to Assumption (3.14), the entries of k are infinity. Hence, the vertex with priority key k has to be locally consistent, which is a contradiction, since U contains locally inconsistent vertices as shown in Lemma 3.1.1. Hence, $g(u) < \infty$.
If u is equal to the end node, then $g(u) = rhs(u) = 0$ which yields to the equality of the g-value and the start distance. Thus, the lemmas proof is trivial.
Assume, that u is not s_{goal} and a successor s, which minimizes $g(s) + c(u, s)$. The aim is to prove the local consistency of s by showing $k(s) \leq k$, which finishes the proof since k is the smallest priority key of any locally inconsistent vertex. The assumptions for s lead to

$$\begin{aligned} g(u) = rhs(u) &= min_{s' \in Succ(u)}(g(s') + c(s', u)) && (3.15)\\ &= g(s) + c(u, s). && (3.16)\end{aligned}$$

Since the edge costs are positive and heuristics consistent, $g(s) \leq g(u)$ and $h(s) \leq c(u, s) +$

$h(u)$ hold. Hence,

$$\begin{aligned}
k(s) &= [min(g(s), rhs(s)) + h(s_{start}, s) + k_m; min(g(s), rhs(s))] \\
&\leq g(s) + h(s_{start}, s) + k_m; g(s)] \\
&< [g(u) + h(u); g(u)] \\
&= [min(g(u), rhs(u)) + h(s_{start}, u) + k_m; min(g(u), rhs(u))] \\
&= k(u) \\
&\leq k,
\end{aligned}$$

and therefore, s is locally consistent. Due to u and s are both locally consistent, Theorem 3.2.2 and Lemma 3.2.4 yield that the vertices remain locally consistent until the while-condition is not fulfilled and the g-value will be equal to their start distances. This together with (3.15) implies, that the edge between u and s is the last one in the path from u and the start node.
This procedure can be repeated to all vertices on the shortest path, starting at u, until s_{goal} is reached. □

Finally, we can prove the correctness of the algorithm.

Theorem 3.2.4. *The procedure ComputeShortestPath() in Algorithm A.1 expands every vertex at most twice, at most once when it is locally overconsistent and at most once if it is locally underconsistent. After termination, a shortest path from s_{start} to s_{goal} can be found, by starting at s_{start} and moving to the successor s, that minimizes $g(s) + c(u, s)$ until s_{goal} is reached.*

Proof. According to Lemma 3.2.4, *ComputeShortestPath()* expands each vertex at most twice, if the condition in line *18* is changed to *"U is not empty"*. Assume that the condition is not changed, then it will terminate, when U is empty, since $k = U.TopKey() = [\infty; \infty]$. Then, $k \geq k(s_{start})$ and $rhs(s_{start}) = g(s_{start})$ as all vertices are locally consistent. Therefore, the condition does not affect which vertices are expanding or the order and so it terminates after expanding each vertex at most twice. Then, s_{start} satisfy the requirements of Lemma 3.2.5 and we obtain the result. □

4 Reinforcement Learning

In this chapter we will discuss the approach to use Reinforcement Learning to navigate through a railway network. Its idea is to learn by interacting with an environment, like the learning we know from the nature, in a computational approach. The main aspect of this method is to learn behavior patterns, in other words, deciding in certain situations to obtain the best outcome. Simply put, Reinforcement Learning is nothing more than trial and error and learning from this. Iteratively, the actor tries different approaches (actions) and tries to maximize his output. Before discussing the basics of Reinforcement Learning, we will take a look on trivial examples made by (Sutton and Barto, 2018), for better a understanding.

Chess player Before making a move in a chess game, the player has to take a look on the board and the place of the figures. After evaluating this, the player can decide for a move, which is already known and will increase the chance to win or one he never tried before and learn for future games.

Robot in a maze A robot starts in an arbitrary place in a maze with full energy and attempts finding the exit of it, before the energy runs out. Of course, the more information the robot will gain from previous attempts, the nearer it will find the most efficient way to escape the maze.

4.1 Basics of Reinforcement Learning

The first part of introducing the Reinforcement Learning will be the definition of the most important vocabulary and formulas, while the first method is presented in 4.3.
The key terms in Reinforcement Learning are *states*, which describes a certain status in the environment, like the places of figures on a chess table. The transition from one state to another is called an *action*. Note, that the actions depending on the state and can be different for each one. The next term that needs to be specified is a *policy*. It defines the agent's way of behaving at a given time and is a possibility of an action, at a perceived state of the environment.
A *reward* defines the goal of a Reinforcement Learning problem. For each taken action, the environment will send a number, which can be positive or negative. This distinction also allows us to distinguish between good and bad decisions. To stay with the example of a chess player, a move which will arise the chance for a win will result in a positive reward, while a decreasing would be negative. In this example, but also for Reinforcement Learning in general, maximizing rewards is the goal. Thus, actions and the resulting rewards affects the policy. For example, if an action leads to lower rewards, then the policy is changed so that it is selected less often in the future.

Rewards indicate which action is good in a specific situation, but does not tell us anything for the long run. To obtain this information, we need a *value function* that gives us the total amount of reward an agent can expect to accumulate in the future for each state, starting from that state. As a result, it is possible for an action to have a small reward, but have a large value due to following high rewards or the reverse could occur. Thus, the values are primary for the action choices, but must be estimated and re-estimated from the observation an agent makes, while the rewards are given directly from the environment. This is why one of the most important components of Reinforcement Learning algorithms is a method for efficiently estimating values (Sutton and Barto, 2018).

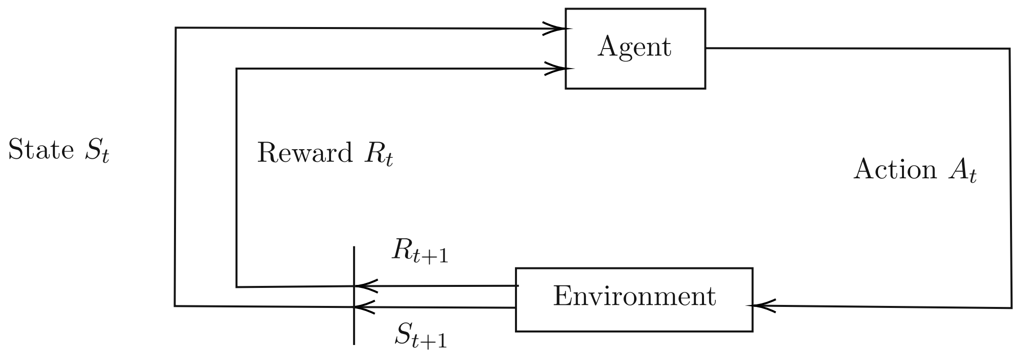

Figure 4.1: The agent-environment interaction in a Markov decision process. The agent chooses an action A_t and the environment responses with the state S_{t+1} and reward R_{t+1}. If the goal state is not reached, the agent will choose action A_{t+1} and the environment response with S_{t+2} and R_{t+2} and so on. (Sutton and Barto, 2018)

To understand how problems are solved, we first consider finite Markov Decision Processes, which are mathematically idealized form of Reinforcement Learning processes for which precise theoretical statements can be made (Sutton and Barto, 2018).
As illustrated in Figure 4.1, in state $S_t \in \mathcal{S}$ the agent chooses an action $A_t \in \mathcal{A}(s)$, where $\mathcal{A}(s)$ define the set of actions in a certain state. At the next step, where it should be mentioned that these are discrete steps $t = 0, 1, 2, ...$, the agent receives a numerical reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and is in the state S_{t+1}. Thus, rise to a trajectory that starts like

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, ... \tag{4.1}$$

The goal in Reinforcement Learning is to maximize the total amount of rewards an agent receives, precisely the expected return G_t at a timestamp t. The simplest form is the trivial sum of rewards

$$G_t = R_{t+1} + R_{t+2} + \ldots + R_T, \tag{4.2}$$

where T denotes a final step. This final step marks the end of a so-called episode, which are subsequences of the trajectory. But how to calculate the expected profit if the task is not episodic but continuous? The remedy for these tasks is a concept called discounting, where the rewards lose value, the further they are in the future. Hence, the expected return is defined as

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{4.3}$$

where the discount rate $\gamma \in [0, 1)$ ensures that rewards lying in the future are worth less. To use only one formula and refer to the close parallel between continuing and episodic tasks, the literature agrees on one formula for both variants,

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{4.4}$$

including the possibility $T = \infty$ or $\gamma = 1$, but not both simultaneously (Sutton and Barto, 2018).
The goal is to maximize Equation (4.4) and therefore the rewards, which depend on the actions the agent choose in a specific state. Recall the policy, which is a probability distribution over actions if the agent is at a certain state, mathematically written

$$\pi(a, s) = \mathbb{P}(a|S_t = s), \tag{4.5}$$

where $a \in \mathcal{A}$ denotes the action and $s \in \mathcal{S}$ the state. As already mentioned, a value function is needed to evaluate the decision possibilities in the long run. For this purpose, two functions are defined, which calculate either the value in a certain state or taking an action in a particular state under a policy π.
The first function we define, calculates the expected return when starting in state s and following policy π. The so-called state-value function is defined as (Sutton and Barto, 2018)

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t | S_t = s \right] \\ &\overset{(4.4)}{=} \mathbb{E}_\pi \left(\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \middle| S_t = s \right). \end{aligned} \tag{4.6}$$

To calculate the value function from choosing an action a in a certain state s under policy π is called the action-value function (Sutton and Barto, 2018)

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \\ &\overset{(4.4)}{=} \mathbb{E}_\pi \left(\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \middle| S_t = s, A_t = a \right). \end{aligned} \tag{4.7}$$

Both functions are defined for any policy π, to maximize the value function, the goal is to find the optimal policy π_*, its state-value function $v_*(s)$ and action-value function $q_*(s, a)$. Note, that there can be multiple policies, but the optimal value functions are unique. Policies are comparable by means of their state value functions (Sutton and Barto, 2018), i.e

$$\pi \geq \pi' \Leftrightarrow v_\pi(s) \geq v_{\pi'}(s), \quad \forall s \in \mathcal{S}.$$

Hence, the state-value function of the optimal policy is greater or equal in all states than state-value function of any other policy,

$$v_*(s) = \max_\pi v_\pi(s)$$

and therefore the action-value function satisfies

$$q_*(s,a) = \max_\pi q_\pi(s,a).$$

Finally, we can formulate Bellman optimality equation according to (Sutton and Barto, 2018) which defines the state-value and action-value functions of the optimal policy.

Theorem 4.1.1 (Bellman Equation)**.** *The state-value function of an optimal policy is defined as*

$$v_*(s) = \max_a \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \middle| S_t = s, A_t = a \right],$$

for all $s \in \mathcal{S}$. Moreover, the action-value function can be written as

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \middle| S_t = s, A_t = a \right].$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.

Proof. Assume π_* is an optimal policy. The state-value function of the optimal policy describes the biggest value function if we start in state s and follow policy π_*. The action-value function in turn provides the biggest value function if start in s, take action a and follow the optimal policy. We obtain that, $v_*(s)$ is nothing else than the action-value function, with the action, which maximizes the value function, i.e.

$$v_*(s) = \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a). \tag{4.8}$$

Thus,

$$\begin{aligned} v_*(s) &\overset{(4.8)}{=} \max_{a \in \mathcal{A}(s)} q_*(s,a) \\ &\overset{(4.7)}{=} \max_{a \in \mathcal{A}(s)} \mathbb{E}_{\pi_*}\left[G_t | S_t = s, A_t = a \right] \\ &\overset{(4.4)}{=} \max_{a \in \mathcal{A}(s)} \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s, A_t = a \right] \\ &= \max_{a \in \mathcal{A}(s)} \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a \right] \\ &= \max_{a \in \mathcal{A}(s)} \mathbb{E}_{\pi_*}\left[R_{t+1} | S_t = s, A_t = a \right] + \gamma \max_{a \in \mathcal{A}(s)} \mathbb{E}_{\pi_*}\left[G_{t+1} | S_t = s, A_t = a \right]. \end{aligned}$$

The last term denotes the expected reward, if the action in state s is taken, which maximizes the expected reward and follows the optimal policy afterwards. This value is equal to the unique optimal state-value function in state S_{t+1}. Hence,

$$v_*(s) = \max_{a \in \mathcal{A}(s)} \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a \right].$$

With (4.8) we receive

$$\begin{aligned} q_*(s) &= \mathbb{E}_{\pi_*}\left[G_t | S_t = s, A_t = a\right] \\ &= \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma G_t | S_t = s, A_t = a\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\right] \\ &= \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma \max_{a' \in \mathcal{A}(s)} q_*(S_{t+1}, a') | S_t = s, A_t = a\right]. \end{aligned}$$

□

4.2 Exploration and Exploitation

In the beginning of Chapter 4 we denoted Reinforcement Learning as learning of trial and error methods which we will now take a closer look at. In general, Reinforcement Learning methods want to take the actions, which were already tried in the past and effective to produce reward, called *exploitation*. But how do we discover the profitable actions or those that generate more rewards than the ones we already know? This is where *exploration* comes in, which means that with a certain probability the greediest choice is not taken, but a random one. It is obvious that neither pure exploitation nor pure exploration can solve the task satisfactorily, so an agent must try a variety of actions and favor the already discovered best actions. This dilemma is unique for Reinforcement Learning algorithms and is still unresolved (Sutton and Barto, 2018).
To ensure a good mix of the two decision policies, we will use an ϵ-greedy policy (Sutton and Barto, 2018) with $\epsilon < 1$. So, with probability $1-\epsilon$ the action is taken, which maximizes the reward, else it takes an arbitrary action. In general, the agent only knows the initialized values at the begin, thus, we will focus on exploration. The more the values are updated by (4.9), the more information about the environment the agent gains. Hence, the focus should be more on exploitation than exploration. Remedy provides us the Exponential decay formula

$$\epsilon(\mathcal{E}) = \epsilon_0 e^{-\lambda \mathcal{E}},$$

where $\mathcal{E}$ indicates the episode, ϵ_0 the initial value and $\lambda \in (0,1)$ a constant called decay constant.

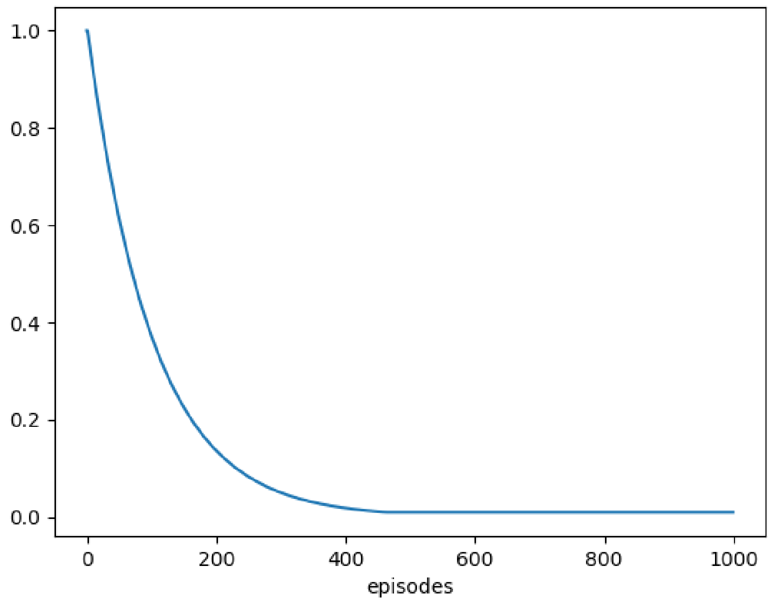

Figure 4.2: Exponential decay with initial value $\epsilon_0 = 1$ and decay $\lambda = 0.01$ over thousand episodes and minimal possible value 0.01

4.3 Q-Learning

Q-Learning (Watkins, 1989) is a simple algorithm for estimating the action-value function to get the greatest expected reward for an action in a certain state. The idea of this method is to estimate the value under a performed action a in a state s directly after receiving the reward, which reduces the length of the trajectory (4.1) and hence the amount of samples compared to Monte Carlo methods, which assume all values for the calculation of the optimal strategy (Sutton and Barto, 2018). The approach we are using belongs to the Temporal Difference learning algorithms.
Q-Learning updates the entries of the eponymous Q-matrix, which rows are the states and the columns the actions, by the rule (Sutton and Barto, 2018)

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_{a \in \mathcal{A}(S_{t+1})} Q(S_{t+1}, a) - Q(S_t, A_t) \right]. \tag{4.9}$$

$Q(S_t, A_t)$ defines the Q-value if the agent is in state S_t and choose action A_t. Obviously, the Q-value on the left hand side of Equation (4.9) presents the new calculated value, while occurrence on the right hand side defines the old value before update. The term in the brackets define the *Temporal Difference Error* which is calculated as the sum of the expected optimal reward, Theorem 4.1.1, and the current value. The learning rate $\alpha \in [0, 1]$ indicates how much the new value affect the old one. In the literature the default value is assumed to be 0.1 (Sutton and Barto, 2018).
In our case, the aim of Q-Learning is to train an agent to find an optimal path between a

start state/node/train station s_{start} and a terminal one, s_{goal}. Therefore, in each timestep the ϵ-greediest action, according to the Q-values, will be taken and the reward r and next step s' observed. Next, the Q-value of action a in state s is calculated by update rule (4.9). If s' is not equal to the terminal state s_{goal}, the steps are repeated with s', otherwise the episode successfully terminates and the next starts with the current Q-values.

Algorithm A.2 Q-Learning

```
1: procedure INITIALIZE( )
2:     for $s \in \mathcal{S}$ do
3:         for $a \in \mathcal{A}(s)$ do
4:             $Q(s,a)$ is arbitrarily
5:     $Q(s_{goal},\cdot) = 0$
6: procedure MAIN( )
7:     for each episode do
8:         $s = s_{start}$
9:         while $s \neq s_{goal}$ do
10:            Choose $a$ in $s$ using $\epsilon$-greedy policy on $Q(s,\cdot)$
11:            Observe $r$ and $s'$
12:            $Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a' \in \mathcal{A}(s')} Q(s',a') - Q(s,a)]$
13:            $s = s'$
```

As already mentioned, the selection of an action in a certain state follows an ϵ-greedy policy on $Q(s,\cdot)$, which means that with probability $1-\epsilon$ the action a, which maximizes $Q(s,\cdot)$ is taken, else an arbitrary action is selected. The first case indicates the exploitation part, we discussed in Section 4.2, the latter the exploration.

4.4 Dyna Q+

Q-Learning in Section 4.3 focus on improving the policy by learning due to episodes, so we select an action in a certain state based on the knowledge we gain through previous episodes. In this section a new method for agents will be introduced, the planning. In (Sutton and Barto, 2018), planning refers to the task of improving policy through interaction with a so-called modeled environment, which helps an agent to predict the expected return based on a taken action by using simulated experience. The model helps the agent predicting the reward and next state according to a certain state and taken action by producing a simulated experience. In (Sutton and Barto, 2018) two ways of simulating the environment are introduced. On the one hand, given a state and a given action, the probability of all next states and reward possibilities are calculated, which are called distributional models; on the other hand, the so-called sample models generate only one possibility according to the probability.
For better understanding and to draw attention to the differences, consider modeling the sum of two dices. A distribution model generates all possible sums and their probabilities,

while a sampling model generates only a single sum according to its probability distribution. Obviously, the first model variant is more accurate, but it is more difficult to implement all possible sums and their possibilities as a simulation of dice rolls and return of sums. At all when the environment becomes more complex, sampling models will be easier to implement which is why in this thesis sample models are used. State-space planning uses these models as input and computes value functions to optimize strategies using backup operations applied to simulated experiences. Thus, both planning, which uses simulated experience generated by a model, and learning, which use real experience generated by the environment, estimates value functions through supporting update operations.

Algorithm A.3 Q-Planning

```
1: **procedure** INITIALIZE( )
2:     **for** $s \in \mathcal{S}$ **do**
3:         **for** $a \in \mathcal{A}(s)$ **do**
4:             $Q(s,a)$ is arbitrarily
5:     $Q(s_{goal}, \cdot) = 0$
6: **procedure** MAIN( )
7:     **while** 1 **do**
8:         Select $s \in \mathcal{S}$, $a \in \mathcal{A}$ random
9:         Send $s$, $a$ to sample model and observe $s'$ and $r$
10:        $Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a' \in \mathcal{A}(s')} Q(s',a') - Q(s,a)]$
```

Learning affects the policy directly, so it is called direct Reinforcement Learning, while planning uses the information of the modeled environment and affects the policy via the model in small, incremental steps. Therefore, it is called indirect Reinforcement Learning. According to (Sutton and Barto, 2018) the advantages of direct Reinforcement Learning methods is to achieve better policies with fewer interactions with the environment, while indirect methods are simpler and do not depend on the model design. Agents that enjoy the benefits of both methods are called Dyna agents.

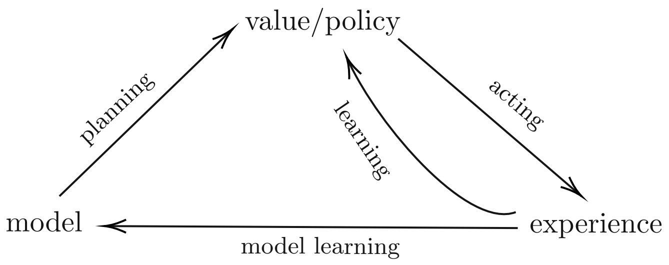

Figure 4.3: This figure shows the actions of a Dyna agent. The circle value/policy-acting-experience-learning marks the direct Reinforcement Learning part, while the circle, which includes the model, the indirect part (Sutton and Barto, 2018).

One example of Dyna agents, which includes all of the processes in Figure 4.3, is the Dyna Q algorithm.

Algorithm A.4 Dyna Q

```
 1: procedure INITIALIZE( )
 2:     for $s \in \mathcal{S}$ do
 3:         for $a \in \mathcal{A}(s)$ do
 4:             $Q(s, a)$ is arbitrarily
 5:             $M(s, a) = NULL$
 6:     $Q(s_{goal}, \cdot) = 0$
 7: procedure MAIN( )
 8:     for each episode do
 9:         $s = s_{start}$
10:         while $s \neq s_{goal}$ do
11:             Choose $a$ in $s$ using $\epsilon$-greedy policy on $Q(s, \cdot)$
12:             Observe $r$ and $s'$
13:             $Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a' \in \mathcal{A}(s')} Q(s', a') - Q(s, a)]$
14:             $M(s, a) = (r, s')$
15:             $s = s'$
16:             for $i \in \{1, \ldots, n\}$ do
17:                 select a random state $\hat{s}$, which was already observed
18:                 select a random action $\hat{a}$ which was already taken in $\hat{s}$
19:                 $(\hat{r}, \hat{s}') = M(\hat{s}, \hat{a})$
20:                 $Q(\hat{s}, \hat{a}) = Q(\hat{s}, \hat{a}) + \alpha[\hat{r} + \gamma \max_{\hat{a}' \in \mathcal{A}(\hat{s}')} Q(\hat{s}', \hat{a}') - Q(\hat{s}, \hat{a})]$
```

In Algorithm A.4 we can see the learning method from line 11 to 13, which is equal to the steps of our direct Reinforcement Learning method Q-Learning A.2. In line 14 we see one Dyna Q typical indirect Reinforcement Learning method called model learning, which indicates the mapping of a state-action pair (s, a) to a reward-state pair (r, s'), leading to the knowledge of gaining the reward and next state, if an action is selected in a certain state. Between lines 16 and 20 we obtain planning steps which use the information of the model to update the values of already chosen actions to get closer to the real environment, like the planning Algorithm A.3. Therefore, Dyna Q has a combination of both methods. As discussed, Dyna Q tries to model the environment as best it can, but reaches its limits when the environment is dynamic (Sutton and Barto, 2018). To solve this issue we will introduce Dyna Q+ (Sutton and Barto, 2018), where the model stores the information when the action was selected, which has an impact on the reward

$$r_{new} = r + \kappa\sqrt{\tau(s, a)}, \tag{4.10}$$

where $\tau(s, a)$ defines the time an action was not selected and κ a small number lower than one. This will force agents to choose actions which were not taken for a longer time and reminds on the exploration-exploitation approach of the direct Reinforcement Learning methods but on the model level. The last difference to Dyna Q is the initialization of the modeled environment, where each state-action pair is initialized with a reward of zero and returns the same state.

Algorithm A.5 Dyna $Q+$

```
 1: procedure Initialize( )
 2:     for $s \in \mathcal{S}$ do
 3:         for $a \in \mathcal{A}(s)$ do
 4:             $Q(s, a)$ is arbitrarily
 5:             $M(s, a) = (0, s, 1)$
 6:     $Q(s_{goal}, \cdot) = 0$
 7: procedure Main( )
 8:     for each episode do
 9:         $s = s_{start}$
10:         $t = 0$
11:         while $s \neq s_{goal}$ do
12:             Choose $a$ in $s$ using $\epsilon$-greedy policy on $Q(s, \cdot)$
13:             Observe $r$ and $s'$
14:             $Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a' \in \mathcal{A}(s')} Q(s', a') - Q(s, a)]$
15:             $M(s, a) = (r, s')$
16:             $s = s'$
17:             $t = t + 1$
18:             for $i \in \{1, \ldots, n\}$ do
19:                 select a random state $\hat{s}$, which was already observed
20:                 select a random action $\hat{a}$ which was already taken in $\hat{s}$
21:                 $(\hat{r}, \hat{s}', \tau) = M(\hat{s}, \hat{a})$
22:                 $r_{new} = \hat{r} + \kappa\sqrt{\tau}$
23:                 $Q(\hat{s}, \hat{a}) = Q(\hat{s}, \hat{a}) + \alpha[r_{new} + \gamma \max_{\hat{a}' \in \mathcal{A}(\hat{s}')} Q(\hat{s}', \hat{a}') - Q(\hat{s}, \hat{a})]$
```

4.5 Deep Dyna Q

One of the biggest limitations of the forms of Q-Learning is the calculation and storage of the Q-matrix since it has to be computed for each state. In our case a state is the composition of a feature vector x, where each entry represent a node, where

$$x_i = \begin{cases} 1, & \text{if } x_i = s \text{ or } s_{goal} \\ 0.5, & \text{if } x_i \text{ is blocked} \\ 0, & \text{else} \end{cases} \tag{4.11}$$

for all $i \in \{1 \dots n\}$, and the information of the date when this vector is valid. Therefore, the computation time for the Q-values will be unsatisfying for high dimensional applications. In (Sutton and Barto, 2018) a solution to this problem is presented by using the techniques from the increasingly widespread field of artificial intelligence, the so called Deep Q Networks (DQN) which are Neural Networks.

4.5.1 Basics of Neural Networks

A Neural Network consists of different layers of neurons, which are connected by edges. Since we are using Feed Forward Neural Networks, each layer gets the input of the previous one and provides the results to the next one, starting with an input layer, that has an argument, the feature vector x from (4.11). The layers between the input and the output are called hidden layers, where hidden layer l is calculated (Fan et al., 2020) by a function

$$\begin{aligned} f_l : \mathbb{R}^{d_{l-1}} \to& \mathbb{R}^{d_i} \\ f_l(\hat{x}) =& \sigma(W_l \hat{x} + v_l), \end{aligned} \tag{4.12}$$

where d_l defines the dimension of the current layer and d_0 indicates the size of the input feature vector x. Additionally, in (4.12) we will find the so called rectified linear unit (ReLU) activation function $\sigma(\cdot) = max(0, \cdot)$, the weight matrix $W_l \in \mathbb{R}^{d_l \times d_{l-1}}$ and the shift bias vector v_l. Hence, defining a ReLU network with L hidden layers, the composition of all layer functions and input vector $x \in \mathbb{R}^{d_0}$ is

$$\begin{aligned} f : \mathbb{R}^{d_0} \to& \mathbb{R}^{d_{L+1}} \\ f(x) =& f_{L+1} \circ f_L \circ \dots \circ f_1(x) \\ =& W_{L+1}\sigma(W_L\sigma(W_{L-1}\sigma(\dots\sigma(W1x + v_1)) + v_{L-1}) + v_L)), \end{aligned} \tag{4.13}$$

where we note, that the shift bias vector of the output layer v_{L+1} is zero and this layer does not require an activation function.

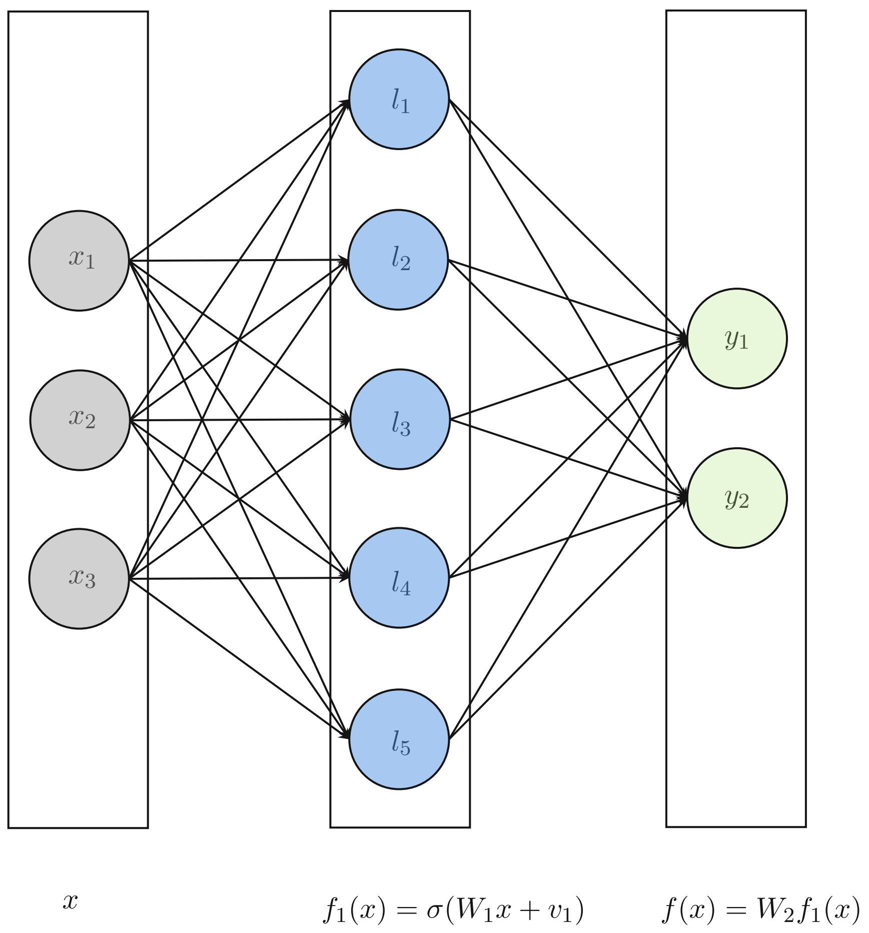

Figure 4.4: Example of a simple fully connected Feed Forward Neural Network with one hidden layer $L = 1$ with dimension $d_1 = 5$. x defines the input vector with dimension $d_0 = 3$ and the output layer is size $d_{L+1} = d_2 = 2$

4.5.2 The algorithm

In (Mnih et al., 2015) two steps, that are crucial for receiving the optimal result with Deep Dyna Q, are presented. The first one is called experience replay (Lin, 1992), where a replay memory M stores, like in Dyna $Q+$ A.5, the taken action in certain states and their corresponding following states and reward values, to train the network via stochastic gradient descent. With this method, temporal uncorrelated samples can be obtained to increase the accuracy of gradient estimation for the optimal result. The second step is the usage of a policy and a target network. While the first one will update its parameters after a certain time with a minibatch of M, the second one will only update for runs, where the goal is achieved. Both steps combined will update the parameter of the models correctly.
Let θ be the parameter of the Q_θ-network and θ^* of the target network. Selecting an independent set of samples $\{s_i, a_i, s'_i, r_i\} \in M \; \forall i \in 1, \ldots, n$, where n is the size of the minibatch, the target value Y_i is calculated quite similar to the Temporal Difference Error

in Bellman Equation in (4.9)

$$Y_i = r_i + \gamma \max_{a'_i \in \mathcal{A}(s'_i)} Q_{\theta^*}(s'_i, a'_i) \tag{4.14}$$

for updating θ by evaluating the gradient of the mean square error (MSE)

$$L(\theta) = \frac{1}{n} \sum_{i=1}^{n} [Y_i - Q_\theta(s_i, a_i)]^2. \tag{4.15}$$

Algorithm A.6 Deep Dyna Q

```
 1: procedure INITIALIZE( )
 2:     $M = \emptyset$
 3:     set time for up $T_{target}$
 4:     initialize network $Q_\theta$
 5:     initialize target network $Q_{\theta^*}$ with $\theta^* = \theta$
 6: procedure MAIN( )
 7:     for each episode do
 8:         $s = s_{start}$
 9:         $t = 0$
10:         while $s \neq s_{goal}$ do
11:             Choose $a$ in $s$ using $\epsilon$-greedy policy on $Q_\theta(s, \cdot)$
12:             Observe $r$ and $s'$
13:             store $(s, a, r, s')$ in $M$
14:             sample random minibatch $\{s_i, a_i, s'_i, r_i\}$ of $M$ with size $n$
15:             compute target $Y_i = r_i + \gamma \max_{a'_i \in \mathcal{A}(s')} Q_{\theta^*}(s'_i, a'_i)$ for all $i \in \{1, \dots, n\}$
16:             Update $\theta = \theta - \alpha \frac{1}{n} \sum_{i=1}^{n} [Y_i - Q_\theta(s_i, a_i)] \nabla_\theta Q_\theta(s_i, a_i)$
17:             if $t = T_{target}$ then
18:                 $\theta^* = \theta$
19:                 $t = 0$
20:             else
21:                 $t = t + 1$
```

4.5.3 Convergence to optimal solution

Since Neural Networks are learning the correlation between specific inputs and outputs, the question is how to prove the convergence to the optimal solution Q^*. As already mentioned, we consider networks with ReLU activation function, defined in (Fan et al., 2020), where most parameters are zero.

Definition 4.5.1 (Sparse ReLU Network)**.** *Let $L, s \in \mathbb{N}, V > 0$ and $d_l \in \mathbb{N}$ for all $l \in \{0, \dots, L+1\}$. Then, the family of sparse ReLU networks are bounded by V with L hidden* *layers, weight sparsity s is defined as*

$$\mathcal{F}(L,(d_i)_{i=0}^{L+1},s,V)=\left\{f:\max_{l\in\{0,\dots,L+1\}}\|(W_l,v_l)\|_\infty\leq 1,\sum_{l=1}^{L+1}\|(W_l,v_l)\|_0\leq s,\max_{l\in\{0,\dots,L+1\}}\|f_l\|_\infty\leq V\right\},$$

where f_l is defined equal to (4.12). Hence, (W_l,v_l) are the weight matrices and shift biased vector of the corresponding layer function.

In Definition 4.5.1 the norms indicates the ℓ_p-norm (Kaltenbaeck, 2014), especially the ℓ_0-norm which counts the entries not equal to zero.
Since the value functions v_π and q_π, which are defined in (4.6) and (4.7), are bounded by $V_{max}=\frac{R_{max}}{1-\gamma}$, the Sparse ReLU Networks are bounded by V_{max}. Therefore, we will write $\mathcal{F}(L,(d_i)_{i=0}^{L+1},s)$ instead of $\mathcal{F}(L,(d_i)_{i=0}^{L+1},s,V_{max})$ in the following. To calculate the error of the neural network with respect to the optimal solution, the problem must be rewritten into a value iteration (Sutton and Barto, 2018). For this purpose, instead of experience replay, a sample of independent transitions from a given distribution is taken by $\sigma\in\mathcal{P}(\mathcal{S}\times\mathcal{A})$, leading to the following representation of the Bellman Equation (4.9)

$$\mathbb{E}(Y_i|S_i,A_i)=(TQ_{\theta^*})(s_i,a_i) \tag{4.16}$$

where TQ_{θ^*} is called the Bellman Operator and describes the optimal state-action equation in Theorem (4.1.1) with Q-matrix Q_{θ^*} in a certain state S_i and taken action A_i. Next, we will examine the significance of the need for target network Q_{θ^*} based on the expected loss via bias-variance decomposition (Friedman, 2017), for which the target network is initially disregarded, i.e. $\theta^*=\theta$,

$$\mathbb{E}(L)=\|Q_\theta-TQ_\theta\|_\sigma^2+\mathbb{E}\left\{[Y_1-(TQ_\theta)(s_1,a1)]^2\right\}. \tag{4.17}$$

Therein, the first term indicates the so-called mean-squared Bellman Error (MSBE) (Sutton and Barto, 2018), which indicates the difference of the calculated Q matrix to the Bellman optimality operator. The second term describes variance of Y_1. In other words, loss function $L(\theta)$ is the empirical version of MSBE with a bias, which also depends on θ. Thus, minimizing $L(\theta)$ is not equal to minimizing MSBE. To eliminate this problem, the previously mentioned target network is used to make the bias dependent on the target network, which leads to

$$\mathbb{E}(L)=\|Q_\theta-TQ_{\theta^*}\|_\sigma^2+\mathbb{E}\left\{[Y_1-(TQ_{\theta^*})(s_1,a_1)]^2\right\} \tag{4.18}$$

and

$$\min_{\theta\in\Theta}L(\theta)\approx\min_{\theta\in\Theta}\|Q_\theta-TQ_{\theta^*}\|_\sigma^2, \tag{4.19}$$

where Θ is the parameter space. Note that, if $\{Q_\theta:\theta\in\Theta\}$ contains TQ_{θ^*}, (4.19) has solution $Q_\theta=TQ_{\theta^*}$, that is why the problem can viewed as an one-step of the previous mentioned value iteration.
Taking a function family of neural networks $\mathcal{F}$, defined on $\mathcal{S}\times\mathcal{A}$ and let $\tilde{Q}_k$ be the current

estimation of the optimal solution Q^* in the k-th iteration step, we define the target value according to (4.14)

$$Y_i = r_i + \gamma \max_{a_i' \in \mathcal{A}(s_i')} \tilde{Q}_k(s_i', a_i') \tag{4.20}$$

and the update of $\tilde{Q}_k$

$$\tilde{Q}_{k+1} = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} [Y_i - f(s_i, a_i)]^2 \tag{4.21}$$

leads to the neural fitted-Q iteration algorithm (Riedmiller, 2005), which is stated in A.7.

Algorithm A.7 neural fitted-Q iteration (neural FQI)

```
 1: procedure Initialize( )
 2:     $\mathcal{F}$ function class of neural networks
 3:     set sampling distribution $\sigma$
 4:     set number of iterations $K$
 5:     set number of samples $n$
 6:     set initial estimator $\tilde{Q}_0$
 7: procedure Main( )
 8:     for $k = 0, 1, \dots, K-1$ do
 9:         Sample i.i.d observations $\{(s_i, a_i, r_i, s_i')\}$ for $i \in \{1, \dots, n\}$
            with $(s_i, a_i)$ drawn by distribution $\sigma$
10:         $Y_i = r_i + \gamma \max_{a_i' \in \mathcal{A}(s_i')} \tilde{Q}_k(s_i', a_i')$
11:         Update $\tilde{Q}_{k+1} = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} [Y_i - f(s_i, a_i)]^2$
12:     Define policy $\pi_K$ as greedy policy with respect to estimator $\tilde{Q}_K$
```

Algorithm A.7 starts with an estimator $\tilde{Q}_0$ and learns the approximation of the Bellman optimality operator T and $\hat{T}_1$ via (4.21) and n samples $\{(s_i, a_i, r_i, s_i')\}$. We obtain that the next estimator $\tilde{Q}_1$ satisfies $\tilde{Q}_1 = \hat{T}_1 \tilde{Q}_0$. Observe that $\hat{T}_2$ is learned by another batch of samples, which yields to $\tilde{Q}_2 = \hat{T}_2 \tilde{Q}_1$ and so on. Applying these steps K times, the final estimator results in

$$\tilde{Q}_K = \hat{T}_K \hat{T}_{K-1} \dots \hat{T}_1 \tilde{Q}_0. \tag{4.22}$$

For proving the convergence of Algorithm A.6 and thus of the DQN to the optimal solution, we will need some assumptions (Fan et al., 2020). First, the state space $\mathcal{S}$ is a compact subset in $\mathbb{R}^r$ and without loss of generality $\mathcal{S} = [0, 1]^r$. Additionally, the action space $\mathcal{A}$ is finite. Last but not least, we will define the function class $\mathcal{F}$ and the approximation of the Bellman optimality operator in a mathematical formal way, which will be done for the first via Definition 4.5.1 and the latter via compositions of Hölder Smooth Functions (Fan et al., 2020).

Definition 4.5.2 (Hölder Smooth Function)**.** *Let $\mathcal{D}$ be a compact subset of $\mathbb{R}^r$, $r \in \mathbb{N}$ and the parameter $\beta, H > 0$, then the set of Hölder Smooth Functions on $\mathcal{D}$ is defined as*

$$\mathcal{C}_r(\mathcal{D}, \beta, H) = \left\{ f : \mathcal{D} \to \mathbb{R} : \sum_{\alpha : |\alpha| < \beta} \|\partial^\alpha f\|_\infty + \sum_{\alpha : \|\alpha\|_1 < \beta} \sup_{x,y \in \mathcal{D}, x \neq y} \frac{|\partial^\alpha f(x) - \partial^\alpha f(y)|}{\|x - y\|_\infty^{\beta - \lfloor \beta \rfloor}} \leq H \right\},$$

where $\lfloor \beta \rfloor$ is the floor function which returns the biggest integer smaller than β. Additionally, we are using the multi-index notation $\alpha = (\alpha_1, \dots, \alpha_r)^T \in \mathbb{N}^r$, which leads to $\partial^\alpha = \partial^{\alpha_1} \cdots \partial^{\alpha_r}$.

Since the final estimator is defined like in (4.22), compositions of Hölder Smooth Functions have do be defined (Fan et al., 2020).

Definition 4.5.3 (Composition of Hölder Smooth Functions)**.** *Let $q \in \mathbb{N}$, $p_j \subset \mathbb{N}$ and $a_j, b_j \subset \mathbb{R}$ such that $a_j < b_j$ for all $j \in \{1, \dots, q\}$. In addition, let*

$$g_j : [a_j, b_j]^{p_j} \to [a_{j+1}, b_{j+1}]^{p_{j+1}}$$

be a function, where the components g_{jk} are Hölder smooth for each $k \in \{1, \dots, p_{j+1}\}$ and they depend on t_j of its input variables, where $t_j \leq p_j$. Hence, $g_{jk} \in \mathcal{C}_{t_j}([a_j, b_j]^{t_j}, \beta_j, H_j)$. With this assumptions and $p_{q+1} = 1$, the family of compositions $\mathcal{G}(\{p_j, t_j, \beta_j, H_j\})$ can be defined, whenever $f \in \mathcal{G}(\{p_j, t_j, \beta_j, H_j\})$ can be written as

$$f = g_q \circ g_{q-1} \circ \ldots \circ g_2 \circ g_1,$$

with Hölder smooth $g_{jk} \in \mathcal{C}_{t_j}([a_j, b_j]^{t_j}, \beta_j, H_j)$ for each $k \in \{1, \dots, p_{j+1}\}$ and $j \in \{1, \dots, q\}$.

With previous mentioned assumptions and Definition 4.5.3 class of functions $\mathcal{F}_0$ and the set which contains Bellman optimality operator TQ can be defined (Fan et al., 2020).

Definition 4.5.4. *Let $\mathcal{F}(L, (d_i)_{i=0}^{L+1}, s)$ be the family of sparse ReLU networks defined on $\mathcal{S}$ with $d_0 = r$ and $D_{L+1} = 1$, following Definition 4.5.1. Then, the set of functions in Algorithm A.7 is defined by*

$$\mathcal{F}_0 = \{f : \mathcal{S} \times \mathcal{A} \to \mathbb{R} : f(\cdot, a) \in \mathcal{F}(L, (d_i)_{i=0}^{L+1}, s)\}.$$

Furthermore, let $\mathcal{G}(\{p_j, t_j, \beta_j, H_j\}_{j \in \{1,\dots,n\}}$ be a set of Hölder smooth function compositions from Definition 4.5.3. Similar to the previous definition, a function class $\mathcal{G}$ is defined as

$$\mathcal{G}_0 = \{f : \mathcal{S} \times \mathcal{A} \to \mathbb{R} : f(\cdot, a) \in \mathcal{G}(\{p_j, t_j, \beta_j, H_j\}_{j \in \{1,\dots,n\}} \text{ for any } a \in \mathcal{A}\}.$$

Now we will assume a property based on (Fan et al., 2020) of the function classes $\mathcal{F}_0$ and $\mathcal{G}_0$ that are important for the neural FQI Algorithm A.7, namely, that the Bellman optimality operator Tf is a composition of Hölder smooth functions if $f \in \mathcal{F}_0$.

Lemma 4.5.1. *For any $f \in \mathcal{F}_0$, there exists $Tf \in \mathcal{G}_0$, where T is the Bellman optimality operator, i.e. for any $f \in \mathcal{F}_0$ and $a \in \mathcal{A}$, $Tf(\cdot, a)$ can be written as composition of Hölder smooth functions as a function of $s \in \mathcal{S}$.*

Proof. The proof will be split into two cases. First, we assume the satisfaction of some smoothness condition. So, let $f \in \mathcal{F}_0$ and for any state-action pair $(s,a) \in \mathcal{S} \times \mathcal{A}$, $P(\cdot|s,a)$ the density of the next state. By definition of the Bellman optimality operator as function in Theorem 4.1.1, it can be written as

$$(Tf)(s,a) = r(s,a) + \gamma \int_{\mathcal{S}} \left[\max_{a' \in \mathcal{A}} f(s',a')\right] \cdot P(s'|s,a)ds'. \tag{4.23}$$

For any $s' \in \mathcal{S}$ and $a \in \mathcal{A}$ we define $g_1(s) = r(s,a)$ and $g_2(s,a) = P(\cdot|s,a)$ as Hölder smooth functions on $\mathcal{S} = [0,1]^r$ with parameter β and H. As already mentioned, f is limited by V_{max}, i.e. $\|f\|_\infty \leq V_{max}$. Thus, the order of differentiation and integration in (4.23) can be changed with respect to s, if $(Tf)(s,a)$ is differentiated, which yields to the affiliation of $s \mapsto (Tf)(s,a)$ to Hölder smooth class $C_r(\mathcal{S}, \beta, H')$, where $H' = H(1 + V_{max})$.
Now, take a look on the more general case, where we want to describe $P(s'|s,a)$ only for any fixed $a \in \mathcal{A}$. Then, $P(s'|s,a) = h_1[h_2(s,a), h_3(s')]$, where $h_2 : \mathcal{S} \to \mathbb{R}^{r_1}$ and $h_3 : \mathcal{S} \to \mathbb{R}^{r_2}$ are feature mappings and $h_1 : \mathbb{R}^{r_1 + r_2}$ is a bivariate function. Defining $h_4 : R^{r_1} \to \mathbb{R}$ as

$$h_4(u) = \int_{\mathcal{S}} \left[\max_{a' \in \mathcal{A}} f(s',a')\right] h_1[u, h_3(s')]ds' \tag{4.24}$$

will lead to

$$(Tf)(s,a) = g_1(s) + \gamma h_4 \circ h_2(s,a), \tag{4.25}$$

which is a composition of Hölder smooth functions if h_4 is Hölder smooth and g_1, h_2 can be represented as composition of Hölder smooth functions. It follows that $Tf \in \mathcal{G}_0$.

The second case is, when the transition density $P(s'|s,a)$ is not smooth, so consider the extreme case where s' is a function $s' = h(s,a)$. Hence,

$$(Tf)(s,a) = r(s,a) + \gamma \max_{a' \in \mathcal{A}} f[h(s,a), a']. \tag{4.26}$$

Due to

$$\left|\max_{a' \in \mathcal{A}} f(s_1', a') - \max_{a' \in \mathcal{A}} f(s_2', a')\right| \leq \max_{a' \in \mathcal{A}} \left|f(s_1', a') - f(s_2', a')\right|$$

for any $s_1', s_2' \in \mathcal{S}$ and the fact that $f(\cdot, a)$ is Lipschitz continuous for any $a \in \mathcal{A}$, $m_1(s) = \max_{a' \in \mathcal{A}} f(s,a')$ is Lipschitz on $\mathcal{S}$. Now assume $g_1(s) = r(s,a)$ and $m_2(s) = h(s,a)$ are compositions of Hölder smooth functions, so is $(Tf)(s,a) = g_1(s) + m1 \circ m_2(s)$. $\square$

Lemma 4.5.1 describes, that $T\tilde{Q}_k \in \mathcal{G}_0$ for each iteration step in neural FQN A.7. According to (Fan et al., 2020) the optimal solution Q^* is closed to $\mathcal{F}_0$ if $\mathcal{G}_0$ can be approximated by functions from $\mathcal{F}_0$. Thus, $\mathcal{F}_0$ is approximately closed under T.

The next assumption is a standard assumption in the literature of approximating policies and value iteration, i.e (Fan et al., 2020), (Munos and Szepesvári, 2008), (Scherrer et al., 2015) and (Farahmand, Munos, and Szepesvári, 2010), which measures the mismatch between a reference measure ν_2 and the distribution of future states starting with measure ν_1 and making m steps according to policies $\pi_1, \cdots, \pi_m$.

Assumption 4.5.1 (Concentrability Coefficients)**.** *Let $\nu_1, \nu_2 \in \mathcal{P}(\mathcal{S} \times \mathcal{A})$ absolutely continuous regarding to a Lebesgue measure on $\mathcal{S} \times \mathcal{A}$ and $\{\pi_t\}$ a sequence of policies for $t \in \{1, \ldots, m\}$. Starting at initial pair (s_0, a_0) with distribution ν_1 and taking action a_t according to policy π_t for $t \in \{1, \ldots, m\}$. Then, $\mathcal{P}^{\pi_m}\mathcal{P}^{\pi_{m-1}} \cdot \mathcal{P}^{\pi_1}\nu_1$ is the donated as the distribution of $\{(s_t, a_t)\}_{t=0}^{m}$, which allows us to define the m-th concentrability coefficient as*

$$\kappa(m, \nu_1, \nu_2) = \sup_{\pi_1, \ldots, \pi_m} \left[\mathbb{E}_{\nu_2} \left| \frac{d(\mathcal{P}^{\pi_m}\mathcal{P}^{\pi_{m-1}} \cdot \mathcal{P}^{\pi_1}\nu_1)}{d\nu_2} \right|^2 \right]^{1/2}.$$

Moreover, let σ be the sampling distribution of A.7 and μ a fixed distribution on $\mathcal{S} \times \mathcal{A}$, then we assume the existence of a constant $\phi_{\mu,\sigma} < \infty$ such that

$$(1-\gamma) \sum_{m \geq 1} \gamma^{m-1} \cdot m \cdot \kappa(m, \mu, \sigma) \leq \phi_{\mu,\sigma},$$

where $(1-\gamma)$ is a normalization term, in fact that $\sum_{m\geq 1} \gamma^{m-1} \cdot m = (1-\gamma)^{-2}$.

Assumption 4.5.1 requires a sufficient coverage of $\mathcal{S} \times \mathcal{A}$ through sampling distribution σ, which is, as shown in (Chen and Jiang, 2019), crucial for the success of batch reinforcement learning methods.
With Lemma 4.5.1 and Assumption 4.5.1 we can introduce the Lemma to estimate the error between the optimal solution and result of the neural FQI A.7.

Theorem 4.5.1. *With Lemma 4.5.1 and Assumption 4.5.1, $\mathcal{F}_0$, $\mathcal{G}_0$ are defined as in Definition 4.5.4 with constants $\{H_j\}_{j\in\{1,\ldots,q\}}$. Furthermore, for any $j \in \{1, \ldots, q-1\}$ define*

$$\beta_j^* = \beta_j \prod_{l=j+1}^{q} \min(\beta_l, 1),$$

where $\beta_q^* = 1$ and let

$$\alpha^* = \max_{j\in\{1,\ldots,q\}} \frac{t_j}{2\beta_j^* + t_j}.$$

Assume that batch sample size n is sufficiently large such that a constant $\xi > 0$ for the parameter of $\mathcal{G}_0$ exists, which satisfies

$$\max\left\{ \sum_{j=1}^{q} (t_j + \beta_j + 1)^{3+t_j}, \sum_{j=1}^{q} log(t_j + \beta_j), \max_{j\in\{1,\ldots,q\}} p_j \right\} \leq (\log n)^{\xi}. \tag{4.27}$$

Setting the hyperparameter $L^*, (d_i^*)_{i=0}^{L^*+1}$ *and* s^* *of* $\mathcal{F}_0$, $d_0^* = 1, d_{L^*+1}^* = 1$ *and some constant* $\xi^* > 1 + 2\xi$ *set*

$$\begin{aligned} L^* &\lesssim (\log n)^{\xi^*}, \\ r \leq \min_{j\in\{1,\dots,L^*\}} d_j^* &\leq \max_{j\in\{1,\dots,L^*\}} d_j^* \lesssim n^{\xi^*}, \\ s^* &\sim n^{\alpha^*}(\log n)^{\xi^*}, \end{aligned} \tag{4.28}$$

then there exists a constant $C > 0$ *such that*

$$\|Q^* - Q^{\pi_K}\|_{1,\mu} \leq C\frac{\phi_{\mu,\sigma}\cdot\gamma}{(1-\gamma)^2}|\mathcal{A}|(\log n)^{1+2\xi^*}n^{(\alpha^*-1)/2} + \frac{4\gamma^{K+1}}{(1-\gamma)^2}R_{max}. \tag{4.29}$$

Sketch of Proof. Remember that the neural FQI A.7 returns an estimator $\tilde{Q}_K$ after K iteration steps and define a policy with respect to it. For calculating the error to the optimal solution $\|Q^* - Q^{\pi_K}\|_{1,\mu}$ it is crucial to relate them to errors done in each step $\{\tilde{Q}_k - T\tilde{Q}_{k-1}\}_{k\in\{1,\dots,K\}}$. According to (Fan et al., 2020) and (Munos and Szepesvári, 2008) the error propagation in batch Reinforcement Learning provides an upper bound of $\|Q^* - Q^{\pi_K}\|_{1,\mu}$ using $\{\|\tilde{Q}_k - T\tilde{Q}_{k-1}\|_\sigma\}_{k\in\{1,\dots,K\}}$, i.e.

$$\|Q^* - Q^{\pi_K}\|_{1,\mu} \leq \frac{\phi_{\mu,\sigma}\cdot\gamma}{(1-\gamma)^2}\max_{k\in\{1,\dots,K\}}\|\tilde{Q}_k - T\tilde{Q}_{k-1}\|_\sigma + \frac{4\gamma^{K+1}}{(1-\gamma)^2}R_{max}, \tag{4.30}$$

where $\phi_{\mu,\sigma}$ is the constant from Assumption 4.5.1. In error propagation (4.30) the upper bound can be split into the statistical error, where $\max_{k\in\{1,\dots,K\}}\|\tilde{Q}_k - T\tilde{Q}_{k-1}\|_\sigma$ is essential, on the left hand side and an algorithmic error on the right hand side. The latter will decay to zero geometrically as the number of iterations K increases. For the convergence of the statistical error depending on the batch size n we have to bound the term $\|\tilde{Q}_k - T\tilde{Q}_{k-1}\|_\sigma$, which is shown in (Fan et al., 2020) under Lemma 4.5.1 for any $k \in \{1,\dots,K\}$. Hence,

$$\|\tilde{Q}_k - T\tilde{Q}_{k-1}\|_\sigma^2 \leq 4[dist_\infty(\mathcal{F}_0,\mathcal{G}_0)]^2 + C\frac{V_{\max}^2}{n}\log N_\delta + \delta C V_{\max}, \tag{4.31}$$

for any $\delta > 0$, constant $C > 0$ and the minimum number of balls to cover $\mathcal{F}_0$, with respect the ℓ_∞ norm, N_δ. Moreover,

$$dist_\infty(\mathcal{F}_0,\mathcal{G}_0) = \sup_{f'\in\mathcal{G}_0}\inf_{f\in\mathcal{F}_0}\|f - f'\|_\infty \tag{4.32}$$

indicates the ℓ_∞ error of functions in $\mathcal{G}_0$ which are approximated by functions in $\mathcal{F}_0$, i.e. using ReLU networks.
Equation (4.31) will be essential for the proof, so it is crucial to understand the boundary. As already mentioned, the first term with $[dist_\infty(\mathcal{F}_0,\mathcal{G}_0)]^2$ indicates the bias caused by approximating $\mathcal{G}_0$ functions with ReLU networks, while the remaining term $C(V_{\max}^2/n)\log N_\delta + \delta C V_{\max}$ measures the variance of the estimator.
For further steps we will set $\delta = 1/n$, which yields to

$$\begin{aligned} \|\tilde{Q}_k - T\tilde{Q}_{k-1}\|_\sigma^2 &\leq 4[dist_\infty(\mathcal{F}_0,\mathcal{G}_0)]^2 + C\frac{V_{\max}^2}{n}\log N_\delta + \frac{1}{n}CV_{\max} \\ &\leq 4[dist_\infty(\mathcal{F}_0,\mathcal{G}_0)]^2 + C\frac{V_{\max}^2}{n}\log N_\delta. \end{aligned} \tag{4.33}$$

The next and final step in this proof is to show that functions in $\mathcal{G}_0$ can be written as composition of Hölder smooth functions defined on a hypercube, which, according to (Schmidt-Hieber, 2020), can be approximate by a ReLU network and so also the functions in $\mathcal{G}_0$. Bounding the error of this approximation and applying the classical results on the covering number of neural networks (Anthony, Bartlett, and Bartlett, 1999) on $\log N_\delta$ concludes the proof.

First, let us recall Definitions 4.5.3 and 4.5.4, which states that for any $f \in \mathcal{G}_0$ and any $a \in \mathcal{A}$, $f(\cdot, a) \in \mathcal{G}(\{p_j, t_j, \beta_j, H_j\})$ can be written as composition $f(\cdot, a) = g_q \circ \cdots \circ g_q$ with Hölder smooth $g_{jk} \in \mathcal{C}_{t_j}([a_j, b_j]^{t_j}, \beta_j, H_j)$ for each $k \in \{1, \dots, p_{j+1}\}$ and $j \in \{1, \dots, q\}$. Now, define Hölder smooth functions on a hypercube like in (Fan et al., 2020)

$$\begin{aligned} h_1(u) &= \frac{g_1(u)}{2H_1} + \frac{1}{2}, \\ h_q(u) &= g_q(2H_{j-1}u - H_{q-1}) \text{ and} \\ h_j(u) &= g_j \frac{2H_{j-1}u - H_{j-1}}{2H_j} + \frac{1}{2} \qquad \forall j \in \{2, \dots, q-1\}. \end{aligned} \tag{4.34}$$

Thus,

$$f = g_q \circ \cdots \circ g_q = h_q \circ \cdot \circ h_1 \tag{4.35}$$

with Hölder smooth functions $h_{jk} \in \mathcal{C}_{t_j}([0,1]^{t_j}, \beta, W)$ where

$$W = \max \left\{ \max_{1 \le j \le -1} (2H_{j-1})^{\beta_j}, H_q (2H_{q-1})^{\beta_q} \right\}. \tag{4.36}$$

Applying the results of (Schmidt-Hieber, 2020) together with the assumptions from (Fan et al., 2020) namely, $m = \eta \lceil \log_2 n \rceil$, $\eta > 1$ sufficiently large constant, a $N \in \mathbb{N}$ sufficiently large number depending on batch size n and $L_j = 8 + (m+5)(1 + \lceil \log_2(t_j + \beta_j) \rceil)$, leads to a sparse ReLU network $\tilde{h}_{jk} \in \mathcal{F}(L_j, \{t_j, \tilde{d}_j, \dots, \tilde{d}_j, 1\}, \tilde{s}_j)$ with

$$\begin{aligned} \tilde{d}_j &= 6(t_j + \lceil \beta_j \rceil) N, \\ \tilde{s}_j &\le 141(t_j + \beta_j + 1)^{3 + t_j} N(m+6) \\ \|\tilde{h}_{jk} - h_{jk}\|_\infty &\lesssim N^{-\beta_j / t_j}. \end{aligned} \tag{4.37}$$

Defining $\tilde{f} : \mathcal{S} \to \mathbb{R}$ by $\tilde{f} = \tilde{h}_q \circ \cdot \circ \tilde{h}_1$ and setting

$$N = \left\lceil \max_{1 \le j \le q} C n^{t_j / (2\beta_j^* + t_j)} \right\rceil, \tag{4.38}$$

(Fan et al., 2020) shows that $\mathcal{F}(L_j, \{t_j, \tilde{d}_j, \dots, \tilde{d}_j, 1\}, \tilde{s}_j)$ can be embedded in $\mathcal{F}(L^*, \{t_j, \tilde{d}_j, \dots, \tilde{d}_j, 1\}, \tilde{s}_j + (L^* - \tilde{(L)} \tilde{d}_j))$ which is a subset of the required network $\mathcal{F}(L^*, \{d_j^*\}_{j=1}^{L^*+1}, s^*)$.
For bounding the difference between $\tilde{f}$ and $f(\cdot, a)$, we define

$$\lambda_j = \prod_{l=j+1}^{q} (\beta_l \wedge 1)$$

for any $j \in \{1, \dots, q-1\}$ and set $\lambda_q = 1$. Consequently, $\beta_j \lambda_j = \beta_j^*$ for all $j \in \{1, \dots, q\}$. Combining the result of (Fan et al., 2020) which estimates

$$\|f(\cdot, a) - \tilde{f}\|_\infty \lesssim \sum_{j=1}^{q} \|\tilde{h}_j - h_j\|_\infty^{\lambda_j}$$

and Assumptions (4.38) and (4.37) will lead to

$$[dist(\mathcal{F}_0, \mathcal{G}_0)]^2 \lesssim n^{\alpha^*-1}. \tag{4.39}$$

Last but not least we estimate the number of balls for covering $\mathcal{F}_0$ with classical results on the covering number of neural networks (Anthony, Bartlett, and Bartlett, 1999) and adapting them to our case (Fan et al., 2020), i.e.

$$\begin{aligned} \log N_\delta &\lesssim |\mathcal{A}| s^* L^* \max_{j \in \{1, \dots, L^*\}} \log(d_j^*) \\ &\lesssim |\mathcal{A}| n^{\alpha^*} (\log n)^{1+2\xi}, \end{aligned} \tag{4.40}$$

where $\delta = 1/n$.
Combining (4.33), (4.39) and (4.40) allows us to estimate (4.30) in the required way. □

After proving the estimation of the error to optimal solution in Theorem 4.5.1, we will discuss the boundary and why it will converge to zero. Similar to (4.30) it can be split into the a statistical error on the left hand side and an algorithmic error on the right hand side. The latter will geometrically converge to zero if iteration step K will be sufficiently large, while the convergence of the other is not that trivial to prove. According to (Fan et al., 2020) the statistical error dominates the algorithmic one if the iteration step satisfies

$$K \geq \frac{C'[\log |\mathcal{A}| + (1 - \alpha^* \log n)]}{\log(1/\gamma)}, \tag{4.41}$$

where C' is a sufficiently large constant. In this case consider $\phi_{\mu,\sigma}$ and γ as constants, ignore the logarithmic term, the error in (4.29) is

$$|\mathcal{A} n^{(\alpha^*-1)/2}| = |\mathcal{A}| \max_{j \in \{1, \dots, q\}} n^{-\frac{\beta_j^*}{2\beta_j^* + t_j}}, \tag{4.42}$$

which scales linear with the batch sample size n and converges to zero if n goes to infinity.

5 Results

The first task of navigating an agent through an existing railway network was to implement an agent based model. To this end, we constructed a data table layout, which was shown in Table 2.1, to store and work with the real data of the Austrian railway network. This table contains the schedule of each agent, i.e. the arrival and departure time and the corresponding station. For this purpose, the real data had to be adjusted from data errors first, which were, for example, unfeasible station sequences, incorrect date information and trains blocking each other. By developing simple rules, as shown in Figure 2.3, which we assigned to each agent and by combining these, as well as the communication of the individual agents, we were able to map phenomenon of emergence well and also observe it further on. This leads to congestion which was shown graphically in Figure 2.4.
To adapt the well-known D*-Lite algorithm to our model, we defined the cost-function $c(s, s')$ as the average time it takes a train to move from train station s to his neighbour s'

$$c(s, s') = \frac{d_E(s, s')}{average\ speed}. \tag{5.1}$$

We used a similar definition for the heuristic h and the costs of the shortest path c^*. Hence, then necessary inequalities, as mentioned in 3.1, are satisfied, which gives the correctness and termination of the algorithm.
As mentioned in Chapter 2 it is possible that the maximum capacity of a train station is reached and therefore the station will be blocked. If one blocked node is part of the calculated shortest path, the agent has two options to handle this issue. Either, like in Figure 3.1, the agent takes a new path without the blocked node or he will wait until the node is not blocked anymore. The latter option changes the costs of the incident edges from Equation (5.1) to

$$c(s, s') = \frac{d_E(s, s')}{average\ speed} + t_{blocked}(s') * 5, \tag{5.2}$$

where $t_{blocked}(s')$ indicates the time until s' is not blocked anymore. Equation (5.2) will force an agent to wait but the costs will increase at each timestep.
Unfortunately, we were not able to run our models on the entire Austrian railway network due to its computational complexity. Therefore, we made our observations based on the eastern region, to be more precise, the stations which have Longitude greater than 14.25 and Latitude greater than 47.25.

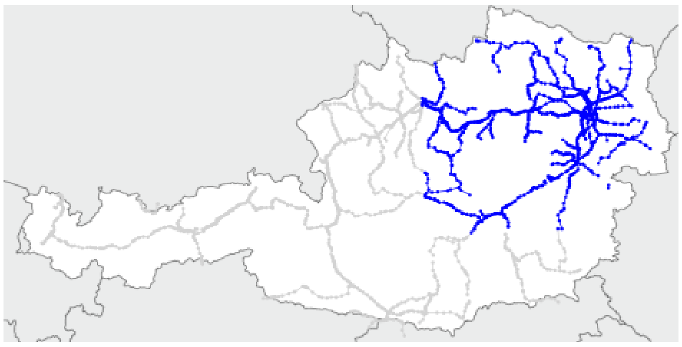

Figure 5.1: railway sub-network, where each node has Longitude greater than 14.25 and Latitude greater than 47.25

In order to make more general statements and draw comparisons to the upcoming algorithms, 300 random start and end nodes were chosen and their path was first computed using A*. Each calculation starts at $16.01.2017 - 00:00$, but D*-Lite agent begins to calculate its path only at $16.01.2017 - 07:00$, which means that around 20.000 actions have already been made and this leads to the fact that stations can be occupied. In addition, all paths were calculated with scan range one and five, which means that after each step of the D*-Lite agent once only the adjacent nodes and the other time even the neighbors over four nodes are examined for cost changes. Unfortunately, we were not able to run our models on the entire Austrian railway network due to its computational complexity. Therefore, we made our observations based on the eastern region. In addition, all paths were calculated with scan range one and five, which obviously leads to different results as shown in Table 5.1 and Figure 5.2.

	A^*	D_1^*	D_5^*
accuracy to A* paths	1	0.93	0.92
runs with delayed trains	0.182	0.151	0.144
runs with delayed D*-Lite agent	0.106	0.072	0.068
average no. of delayed trains p.r.w.d.	2.83	2.91	3.45

Table 5.1: Table displays results regarding the delays for agents following the A* path and D* path with scan range one (D_1^*) and five (D_5^*).
p.r.w.d = per run with delay

According to the results shown in Table 5.1, the agent following the D*-Lite path with scan range 5 interferes least with the existing trains and also avoids delays at the agent itself. These results can be explained due to the higher scanning range and thus the higher number of nodes that are scanned for updates after each step. This behavior tries to avoid delays, but when they occur more trains are hindered in comparison to the other algorithms, as stated in Table 5.1. It also shows that a higher scan range and the associated higher probability of deviating from the trunk routes will block more trains on their designated routes, causing more delayed trains.
Another aspect that should not be neglected is the shortest path that the agent should take, and here, D*-Lite with scan range 1 performs best compared on 300 paths in total. Defining the average speed in (5.2) as $100km/h$ yields that the sum of the travel time over all 300 paths is 2.5 minutes faster than A* and even 27.8 minutes faster than D*-Lite with scan range 5. This leads us to conclude that D*-Lite with scan range 1 is best suited for our application.

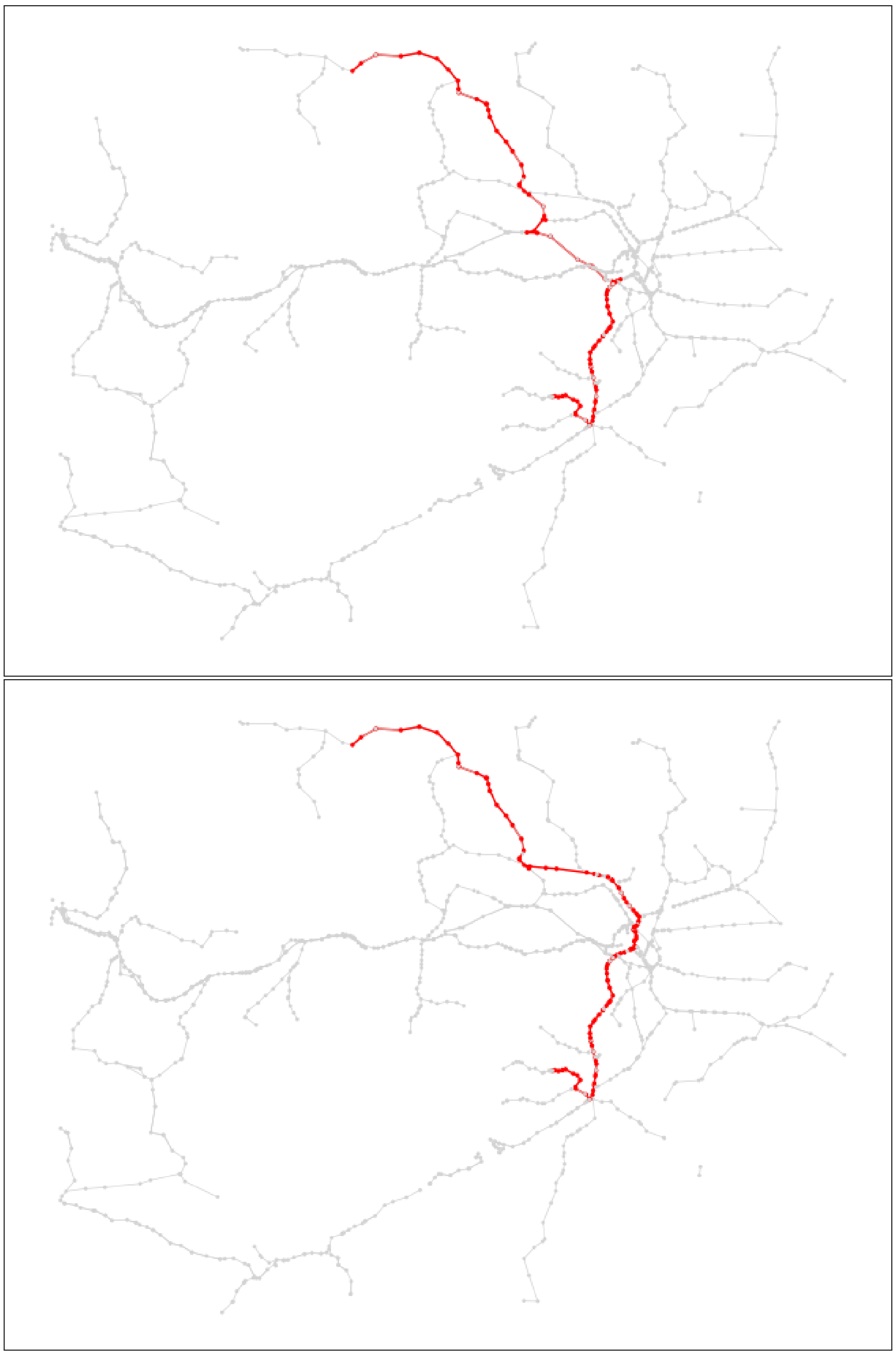

Figure 5.2: Example of calculated paths with scan range 1 (top) and 5 (bottom). Since the longer range also takes into account nodes that are further away, the agent can decide to take a different action, which leads to a different and also longer path. In this example, the D*-Lite agent with scan range 1 got a travel time of 88 minutes until the goal is reached, while the variant with the higher scan range needs 108 minutes.

D*-Lite only detects new blocked nodes, but we have observed, that it is not designed for such a dynamic environment where nodes can be unblocked after a certain time. Furthermore, the algorithm can only compute one path at a time, which makes it useless for a business application.
To deal with the time depending component of our model, we first studied Reinforcement Learning theory, namely the Q-Learning and Q-Planning methods. It turned out, that a combination of those two was a good candidate for path finding in our considered dynamic railway network. This combination is known as Dyna $Q+$. Therein, actions that have already been performed are stored in a memory buffer and the q-values that are updated taking into account the time that has elapsed since the action was performed.

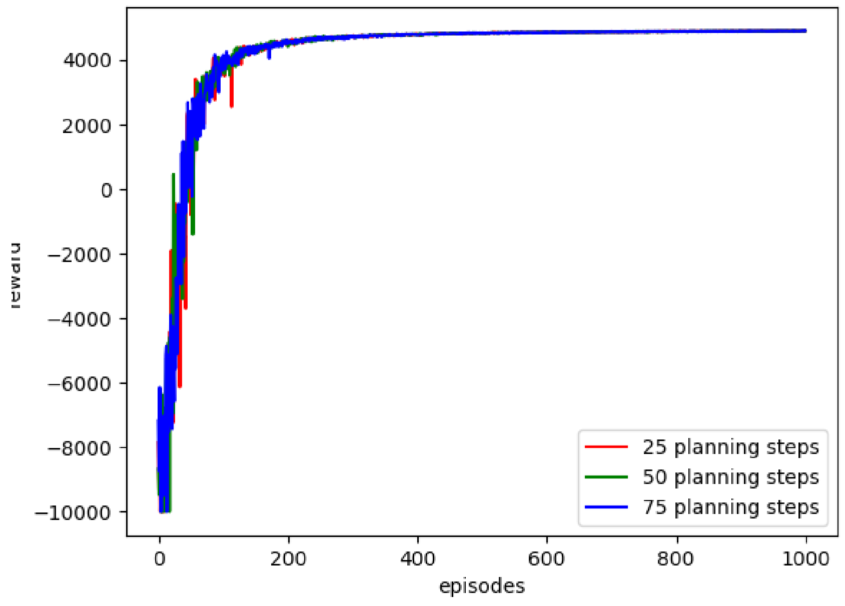

Figure 5.3: Average Dyna $Q+$ reward over 10 runs with different planning steps and 1000 episodes, where the reward function is as for Deep Dyna Q and a state is defined as feature vector x. We observe convergence to the optimal reward function, which is identical to convergence to the optimal path. Note that this does not indicate convergence to the global optimal solution; it may also be a local one. Another observation is that the number of planning steps does not dramatically affect convergence.

A major drawback of the Dyna $Q+$ algorithm is that each state and the possible actions are stored into the Q-matrix, which causes storage and computational problems for applications with high dimensions. This is the case for our model with 1198 nodes, where a specific action is that an agent can choose which neighbor it will visit next. We obtain $m(s) \in \mathbb{N}$ actions in each node/state, where $m(s)$ indicates the number of neighbors in state $s \in \mathcal{S}$. Therefore, Deep variants of Reinforcement Learning methods were used for implementing a Deep Dyna Q algorithm as it was done in Algorithm A.6. One of the first and most important thoughts about implementing Neural Networks is the choice of feature vector. This has to be chosen such that the individual states are unique and clear. In our case, if the agent is in a certain station and has a certain goal, we want to find the best action. Furthermore, it would be advantageous to include possible currently blocked track sections in the decision. Formally speaking, the agent is in node s with goal s_{goal} and therefore the feature vector $x = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$ is defined as

$$x_i = \begin{cases} 1, & \text{if } x_i = s \text{ or } s_{goal} \\ 0.5, & \text{if } x_i \text{ is blocked} \\ 0, & \text{else} \end{cases} \tag{5.3}$$

where $i \in 1, \dots, n$. Note that $n \in \mathbb{N}$ indicates the number of station in the selected eastern region. The order of the nodes in the feature vector remains the same in all calculations.
Another important aspect for Reinforcement Learning in general and the variants of Dyna $Q+$ in particular is the choice of the reward function. In path finding problems, one uses a positive return if the agent reaches the goal and to avoid infinite paths, a negative or neutral one for each other possible action (Sutton and Barto, 2018). In our setting this leads to

$$\begin{aligned} r :& \mathcal{S} \times \mathcal{A} \to \mathbb{R} \\ r(s,a) =& \begin{cases} 1000, & \text{if } s \text{ with } a \text{ leads to } s_{goal} \\ -1 \cdot c_{failed}(s), & \text{else} \end{cases} \end{aligned} \tag{5.4}$$

Note that $c_{failed}(s) \in \mathbb{N}$ indicates the amount of failed actions, as the last one was successful in state $s \in \mathcal{S}$. The reason for choosing the function that way was that if we are using a lower reward for reaching the goal, the actions that penalize the agents would clearly outweigh the positive reward. This leads to a model where the agent would never learn to reach the target node because it is an insignificant part of the total reward.

Due to time and resource constraints, it was not possible to compute all paths, as Neural Network training is time consuming and requires sufficiently powerful hardware. Nevertheless, the feasibility of this approach was demonstrated, but the results did not reach the expected quality, as a tendency for the gradient descent to get stuck in a local minimum was observed for the subset of paths where the algorithm successfully finished.

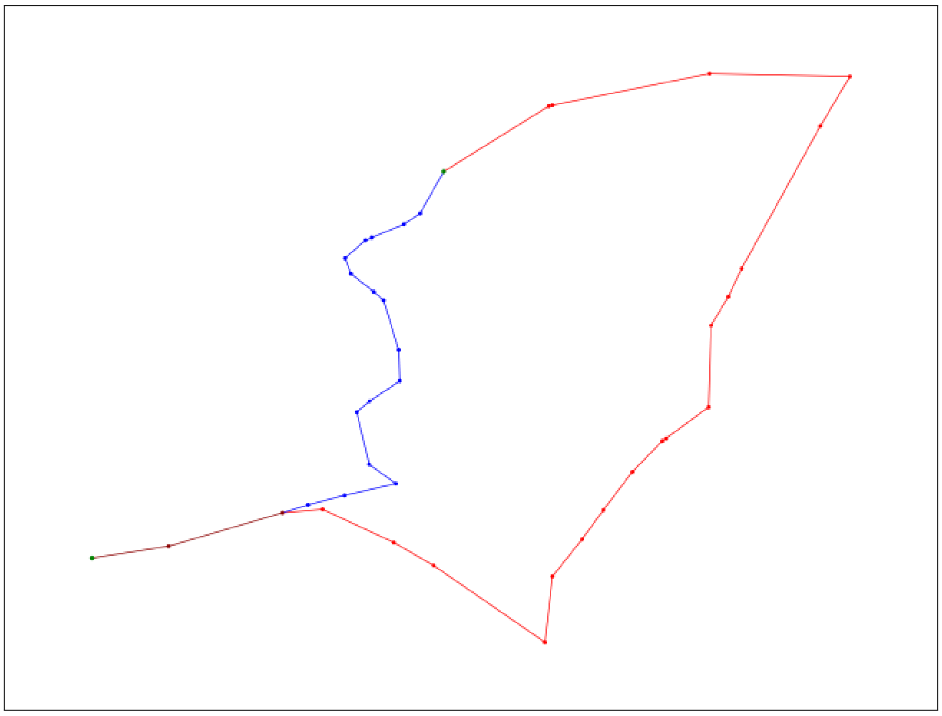

Figure 5.4: Example of a calculated Deep Dyna Q path in comparison with the calculated D* path. Red edges indicates the Deep Dyna Q solution which needs 22 minutes to reach the goal, while blue ones the D* result, which only needs 21. Start and goal are green.

Figure 5.4 shows an example of finding a local but not global minimum. As we can see, the Deep Dyna Q solution takes a different and longer path than the D* solution. Examining the solution using the delayed trains, as for D*-Lite agents, shows us that neither the agent nor other trains have delays. Therefore, the algorithm computes a solution that does not cause delays, but it does not find the global optimal path.
Despite all this, if the Neural Network finds a path, you can see in Figure 5.5 and 5.6 that it can then approximate the behavior of the Dyna $Q+$ algorithm well. Figure 5.5 shows that choosing the action with the maximum value in each state leads to the (local) optimal path. Furthermore, in the second part of the figure it is shown that when the agent starts from a nearby station, it finds the aforementioned optimal path to the same destination and follows it until it reaches the goal state. In 5.6, we see that the closer the agent is to the target node, the higher the approximated Q-values for the optimal solution are.

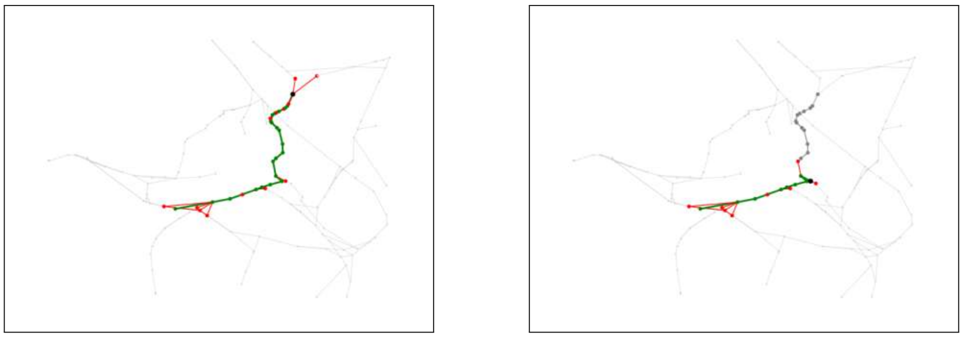

Figure 5.5: Example of a calculated Deep Dyna Q path from a starting point (black node), which displays the approximated Q-values for all steps. If an agent starts at the black start node, the edges are colored according to a traffic light system. This means that the edge with the best value is colored green and is therefore also the next one that the agent visits. Actions with at least 50% of the maximum value are colored orange, those below are colored red. The second image shows that the agent, starting from another node that is near the origin path, finds its way back to the path and follows it until the destination is reached. The path in grey is the previous one from the left picture.

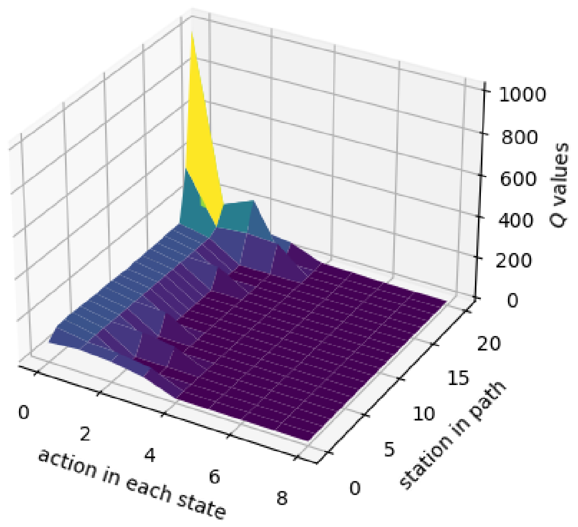

Figure 5.6: The figure shows the Q-values of each possible action for each station on a given calculated path. The closer the station is to the destination, the higher the value. This behavior shows the correct approach of the Dyna Q method.

Additionally, we proved the convergence to the optimal solution by rewriting the Neural Network as a sparse matrix in an iterative algorithm and define the Bellman Operator as composition of Hölder smooth function by following (Fan et al., 2020). Then the error in the optimal solution can be bounded by a statistical and an algorithmic error, the first converging to zero when the sample size approaches infinity, and the second when the iteration steps are sufficiently large.

5.1 Conclusion

To summarize this and bridge to the current trend, the reason for employing an artificial intelligence is the insufficient applicability to complex systems of the conventional method, in our case this was the heuristic approach D*-Lite. Simple machine learning methods, such as Dyna $Q+$, have their drawbacks, such as enormous computation times or memory problems at high dimensions, but serve their purpose once computed. Moreover, these methods have been thoroughly and comprehensibly proven, whereas neural networks, despite their large field of application, can be seen as a black box. Therefore, proving convergence to the optimal solution is nontrivial, which leads to the fact that proofs of already known applications have been published only recently or not at all. It is undisputed that Neural Networks are a powerful tool and also relatively easy to implement, but due to the high computational cost and opaqueness often simpler machine learning methods are at an advantage. However, since this field of research has experienced a real upswing in recent years, it is to be expected that new insights will be gained in both the applied and theoretical areas and that neural networks will be favored in the future.

6 Future Work

For all presented algorithms, implementation improvements can of course still be made and applied to the route network of the whole of Austria. Apart from these basic examples, more complex solutions can be implemented in the agent based model, Reinforcement Learning methods, and Neural Network. In ABM, for example, the individual agent's control system can be extended to include more data. This can be external factors such as weather influences or spontaneously closed edges, or internal ones such as maintenance times.
After that, one can tackle the problem of navigating another agent through the network, which leads us to transfer learning, i.e. learning the new target with the help of the already trained network (Torrey and Shavlik, 2010) and Multi Agent Reinforcement Learning (MARL) (Busoniu, Babuska, and De Schutter, 2008). The last method presented in this chapter for future work is using a so-called Graph Convolutional Network (GCN) (Kipf and Welling, 2016). Deep Dyna Q 4.5 learns while searching for the optimal result and is an unsupervised learning method by definition. Neural Networks are presented as supervised methods in the literature, that is why GCNs are worth mentioning to get a general insight into this topic.

6.1 Graph Convolutional Network

Similar to 4.5, the network consist of layers. Here, the input layer X consists of a matrix whose entries are the vectors of the individual nodes. Thus, the dimension is $n \times d$, where $n \in \mathbb{N}$ indicates the amount of nodes and $d \in \mathbb{N}$ the dimension of the feature vector x_i, $i \in \{1, \dots, n\}$. The dimension of the output layer Z is $n \times f$ where f is the number of the output features. According to (Kipf and Welling, 2016), each hidden layer can be written as non-linear function, which depends on the previous layer and the adjacency matrix A, i.e.

$$H^{(l+1)} = f(H^{(l)}, A), \tag{6.1}$$

where $H^{(0)} = X$, $H^{(L)} = Z$ and $l \in \{0, \dots, L-1\}$.

A common propagation rule of Graph Convolutional Networks is defined in (Kipf and Welling, 2016) as

$$H^{(l+1)} = \sigma(D^{-\frac{1}{2}} \hat{A} D^{-\frac{1}{2}} H^{(l)} W^{(l)}). \tag{6.2}$$

In (6.2) $H^{(l)}$ indicates the l-th neural network layer, while $W^{(l)}$ is a weight matrix for them. Now consider the term $D^{-\frac{1}{2}} \hat{A} D^{-\frac{1}{2}}$ and what information it provides to the propagation layer. First, the use of the matrix $\hat{A} = A + I$ ensures the sum up features of all adjacency nodes and the node itself. According to (Kipf and Welling, 2016), multiplying with $\hat{A}$ will chance the scale of the feature vectors, as a consequence we have to normalize the matrix, i.e. the sum of each row is one. Therefore, the diagonal node degree matrix D of $\hat{A}$ is multiplied by the matrix. Last but not least each layer is completed with an activation function $\sigma(\cdot)$ which categorizes the output values, for example either zero or one.

Applying GCN to our work, the probability to visit an adjacent node in a certain state is recommended. To this end, we follow the idea of (Osanlou et al., 2021). There, the input is an instance $I = (s_{start}, s_{goal})$, where s_{start} is the start node, while s_{goal} is the end node and the feature matrix is a pair of features for each node, more precisely

$$x_n = (s_n, g_n),$$

where

$$s_n = \begin{cases} 1 & \text{, if } n = s_{start} \\ 0 & \text{, else} \end{cases}$$

and

$$g_n = \begin{cases} 1 & \text{, if } n = s_{goal} \\ 0 & \text{, else} \end{cases}$$

for each node $n \in V$.
In this thesis 3 convolution layers are used, where the activation function for each layer is the so called ReLu function $\sigma(\cdot) = ReLu(\cdot) = max(0, \cdot)$, (Osanlou et al., 2021). According to (Osanlou et al., 2021), the softmax function

$$softmax(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{|V|} e^{z_k}}$$

is used after the last layer for calculating the probabilities of each vertex.

Similar to the thesis (Osanlou et al., 2021), we are using a learning method, which is called supervised learning. There, the optimal solution is already known and the loss of the trained Graph Convolutional Network will be calculated and effect the further learning sequences.
For generating training instances, the shortest path between each two nodes in the graph is calculated via A* algorithm. The longest ones are taken and stored as basis instance $I = (s_{start}, s_{goal})$, with an optimal shortest path
$p = \{s_{start}, s_1, s_2, \ldots, s_p, s_{goal}\}$. Each of the instance-solution pairs (I, p) is split into the components of the optimal path and creates more instance-solution pairs (I_i, p_i). According to (Osanlou et al., 2021) the split pairs satisfies, that p_i is the shortest path of I_i.

Lemma 6.1.1. *Assume instance $I = (s_{start}, s_{goal})$ with a corresponding optimal path $p = \{s_{start}, s_1, s_2, \ldots, s_p, s_{goal}\}$. Then, $p_i = \{s_i, s_{i+1}, \ldots, s_p, s_{goal}\}$ is an optimal solution for the instance $I_i = (s_i, s_{goal})$ for all $i \in 1, \ldots, p$.*

Proof. Assume $p = \{s_{start}, s_1, s_2, \ldots, s_p, s_{goal}\}$ is the optimal path of the instance $I = (s_{start}, s_{goal})$ and $p_1 = \{s_1, s_2, \ldots, s_p, s_{goal}\}$ is not the optimal solution for the instance $I_1 = (s_1, s_{goal})$. Then, there exits another path $\hat{p}$, which is a shorter path as p_1. Thus, p is not the optimal solution for I, which is a contradiction to the assumption. Applying this inductively, the Lemma is proved. □

According to Lemma 6.1.1 the train set contains root pairs (I_j, p_j) and the resulting pairs (I_{j_i}, p_{j_i}). Training the network to gain the next best node $\hat{s_{j_i}}$, the second entry of the pair is replaced by those. Our data set D contains all of those pairs $((I_{j_i}, \hat{s_{j_i}}))$ and is split into disjoint train and test sets

$$D = D_{train} \dot{\cup} D_{test}, \tag{6.3}$$

where 80% of the data is split between the first data set and the remaining 20% for validation.
First, the model is trained with the regarding data set and the loss function L is defined in (Osanlou et al., 2021) as the average of the logarithmic loss of the probability predicted by the neural network

$$L = \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} -t_{ij} f(x_i)_j, \tag{6.4}$$

where m indicates the number of examples in D_{train}, n the number of nodes and

$$t_{ij} = \begin{cases} 1 & \text{, if } n = \hat{s_{ij}} \\ 0 & \text{, else} \end{cases}$$

7 Acknowledgments

Parts of this thesis and the underlying research were conducted in the course of the project AundO (project number 867178) funded by the Austrian Research Promotion Agency (FFG).

Bibliography

Anthony, Martin, Peter L Bartlett, and Peter L Bartlett (1999). *Neural network learning: Theoretical foundations*. Vol. 9. cambridge university press Cambridge.

Bonabeau, Eric (2002). "Agent-based modeling: Methods and techniques for simulating human systems". In: *Proceedings of the national academy of sciences* 99.suppl 3, pp. 7280–7287.

Busoniu, Lucian, Robert Babuska, and Bart De Schutter (2008). "A Comprehensive Survey of Multiagent Reinforcement Learning". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 38.2, pp. 156–172. DOI: `10.1109/TSMCC.2007.913919`.

Chen, Jinglin and Nan Jiang (2019). "Information-theoretic considerations in batch reinforcement learning". In: *International Conference on Machine Learning*. PMLR, pp. 1042–1051.

d-maps (n.d.).

Fan, Jianqing et al. (2020). "A theoretical analysis of deep Q-learning". In: *Learning for Dynamics and Control*. PMLR, pp. 486–489.

Farahmand, Amir Massoud, Rémi Munos, and Csaba Szepesvári (2010). "Error propagation for approximate policy and value iteration". In: *Advances in Neural Information Processing Systems*.

Friedman, Jerome H (2017). *The elements of statistical learning: Data mining, inference, and prediction*. springer open.

Friesen, Marcia R. and Robert D. McLeod (2014). "A Survey of Agent-Based Modeling of Hospital Environments". In: *IEEE Access* 2, pp. 227–233. DOI: `10.1109/ACCESS.2014.2313957`.

Hoog, Sander van der (2017). *Deep Learning in (and of) Agent-Based Models: A Prospectus*. arXiv: `1706.06302 [q-fin.EC]`.

Kaltenbaeck, Michael (2014). *Fundament analysis*. ger. Berliner Studienreihe zur Mathematik. Lemgo: Heldermann.

Karney, Charles FF (2013). "Algorithms for geodesics". In: *Journal of Geodesy* 87.1, pp. 43–55.

Kipf, Thomas N and Max Welling (2016). "Semi-Supervised Classification with Graph Convolutional Networks". In: *arXiv preprint arXiv:1609.02907*.

Koenig, Sven and Maxim Likhachev (2002). "D* lite". In: *Aaai/iaai* 15.

Koenig, Sven, Maxim Likhachev, and David Furcy (May 2004). "Lifelong Planning A*". In.

Lin, Long-Ji (1992). "Self-improving reactive agents based on reinforcement learning, planning and teaching". In: *Machine learning* 8.3-4, pp. 293–321.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *nature* 518.7540, pp. 529–533.

Munos, Rémi and Csaba Szepesvári (2008). "Finite-Time Bounds for Fitted Value Iteration." In: *Journal of Machine Learning Research* 9.5.

Osanlou, Kevin et al. (2021). "Constrained shortest path search with graph convolutional neural networks". In: *arXiv preprint arXiv:2108.00978*.

Riedmiller, Martin (2005). "Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method". In: *European conference on machine learning*. Springer, pp. 317–328.

Scherrer, Bruno et al. (2015). "Approximate modified policy iteration and its application to the game of Tetris." In: *J. Mach. Learn. Res.* 16, pp. 1629–1676.

Schmidt-Hieber, Johannes (2020). "Nonparametric regression using deep neural networks with ReLU activation function". In: *The Annals of Statistics* 48.4, pp. 1875–1897.

Sutton, Richard S and Andrew G Barto (2018). *Reinforcement learning: An introduction*. MIT press.

Torrey, Lisa and Jude Shavlik (2010). "Transfer learning". In: *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, pp. 242–264.

Watkins, Christopher John Cornish Hellaby (1989). "Learning from delayed rewards". In.

Yin, Yuan et al. (2021). *Augmenting Physical Models with Deep Networks for Complex Dynamics Forecasting*. arXiv: 2010.04456 [stat.ML].