UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Learning Shepherding Behavior

Michael Baumann

Dissertation
in Computer Science

submitted to the

Faculty of Computer Science,
Electrical Engineering and Mathematics

University of Paderborn

in partial fulfillment of the requirements for the degree of

doctor rerum naturalium
(Dr. rer. nat.)

Paderborn, October 2015

Acknowledgements

Although this work is—unless otherwise stated—a work of my own, many people helped to finish this thesis. Thus, it is my pleasure to express my gratitude to all those who accompanied me on this journey.

First, I would like to thank my advisor Prof. Dr. Kleine Büning for his patience, his constant support, and for always making time for discussions. His invaluable advice helped tremendously to shape this project. I am also grateful to Jun.-Prof. Dr. Heiko Hamann who agreed to review this thesis. Additionally, I would like to thank Prof. Dr. Friedhelm Meyer auf der Heide, Prof. Dr. Eyke Hüllermeier, and Dr. Matthias Fischer for being on my examination committee.

I would like to thank my (former) colleagues Dr. Uwe Bubeck, Dr. Asmir Vodenčarević, Felix Mohr, and Yan Yuhan for creating such an inspiring working atmosphere in the research group *Knowledge-based Systems* at the University of Paderborn. Additionally, I would like to thank Dr. Thomas Kemmerich, Dr. Markus Eberling, and Timo Klerx for fruitful discussions, for proof-reading papers, and for collectively drinking all the coffee. Additionally, I am grateful to Dr. Theodor Lettmann for his advice on research and beyond. I am indebted to Thomas, Timo, and Felix for proof-reading parts of this thesis and presenting valuable suggestions. Furthermore, I thank Simone Auinger, Elisabeth Lengeling, Gerd Brakhane, Christa Stoll, and Christina Lange for their help in all organizational and technical matters.

I am grateful to the International Graduate School *Dynamic Intelligent Systems* at the University of Paderborn for funding parts of my research and providing me with the opportunity to attend all those interesting conferences. In particular I like to thank Prof. Dr. Eckhard Steffen and Astrid Canisius for their organizational support.

I would like to express my deepest gratitude to my parents Susanne and Bernd for their continuous support and for making all of this possible in the first place.

Most importantly, I am very grateful to Yvonne for improving the readability of this thesis, for her lovely support and understanding, and for enduring all the time it took to finish this project.

Michael Baumann
Paderborn, October 2015

Abstract

Artificial shepherding strategies, i.e. using robots to move certain individuals in a controlled manner to a given location, have many applications in different situations. For example, people can be guided by mobile robots from dangerous places or swimming robots may be used to assist in cleaning up oil spills. Independent of an actual deployment of the strategies on real robots, investigating such strategies can help to improve the response of stewards or other security service personnel.

This thesis uses a multiagent system as model for the robots in which the sheep are modeled with reactive behaviors. We analyze the complexity of the shepherding task and present a greedy algorithm that only needs linear time to compute a solution that is close to optimal. In fact, the worst case length differs from the optimal solution in a term linear in the size of the sheep's viewing range. In addition to this, we analyze to what extend such strategies can be *learned* as learning usually provides powerful solutions. This thesis focuses on reinforcement learning as learning method.

To enable reinforcement learning agents to use their knowledge more efficiently in continuous or large state spaces as it is the case in the shepherding task, methods to transfer knowledge to unseen but similar situations are required. We investigate two different approaches to achieve such behavior: *State space abstraction* combines "similar" states to derive a smaller, abstract state space while *function approximation* directly approximates the value function of the reinforcement learning agent.

The approaches developed in this thesis, *GNG-Q* and *I-GNG-Q* combine reinforcement learning with adaptive neural algorithms and enable the agent to learn behavior in parallel with its representation. Both are based upon the growing neural gas, which is an unsupervised learning approach that learns a vector quantization by placing units in areas of the input space from which input data can be expected. *GNG-Q* groups states that are spatial close and share the same behavior while *I-GNG-Q* combines the learned behavior from a larger area of the approximation which results in a smoother value function. Thus, *GNG-Q* performs a state-space abstraction and *I-GNG-Q* approximates the value function. Both approaches monitor the agent's policy during its interaction with the environment to find regions of the approximation that have to be refined. Amongst many others, the core advantages of the developed approaches are that they do not need the model of the environment and that the resolution of the approximation is determined automatically and does not depend on domain knowledge. Additionally, such approaches omit storing values for all state-action pairs and are thus directly applicable to continuous state spaces.

The experimental evaluation underlines that the behaviors learned using our approaches are highly efficient and that the storage needed for the computed approximation is much smaller compared to an extensive representation.

Zusammenfassung

Roboter, die Schafe hüten sowie die dazu notwendigen Strategien zum kontrollierten Bewegen von Individuen zu einem gegebenen Ziel, bieten vielfältige Anwendungen in unterschiedlichsten Situationen. So wäre der Einsatz von mobilen Robotern bei Rettung von Menschen aus gefährlichen Lagen ein durchaus realistisches Einsatzfeld. Darüber hinaus könnten schwimmende Roboter bei der Beseitigung von Ölteppichen eingesetzt werden. Unabhängig von der tatsächlichen Umsetzung dieser Strategien durch Roboter, kann die Untersuchung solcher Taktiken helfen, die Logistik im Rettungswesen, im Umweltschutz oder in anderen Bereichen deutlich zu verbessern.

In dieser Arbeit verwenden wir ein Multiagentensystem mit aktiven Agenten als Modell für die Roboter und reaktiven Agenten als Pendant für die Schafe. Wir untersuchen die Komplexität des Schafehütens und entwickeln einen Greedy-Algorithmus, der in linearer Laufzeit eine fast optimale Lösung berechnet. Tatsächlich ist die maximale Abweichung von der optimalen Lösung linear in der Größe der Sichtweite des Schafs. Zusätzlich zu diesen Ergebnissen analysieren wir, wie solche Strategien *gelernt* werden können, da maschinelles Lernen oftmals zu effizienten und effektiven Lösungen führt. Im Folgenden nutzen wir Reinforcement Learning als Lernmethode.

Damit Reinforcement Learning Agenten ihr gelerntes Wissen auch in kontinuierlichen oder sehr großen Zustandsräumen vorhalten können, werden Methoden zur Wissensabstraktion benötigt. Diese erlauben dem Agenten bereits erlerntes Wissen auf ähnliche Situationen zu übertragen. Eine solche Fähigkeit wird beispielsweise beim maschinellen Schafehüten benötigt. Dazu untersuchen wir zwei unterschiedliche Ansätze: Die *Zustandsraumapproximation* aggregiert ähnliche Zustände und erzeugt einen kompakteren, abstrakten Zustandsraum, während *Funktionsapproximationen* die Bewertungsfunktionen des Reinforcement Learning Agenten approximieren.

Die Ansätze in dieser Arbeit kombinieren Reinforcement Learning mit adaptiven neuronalen Algorithmen. Sie ermöglichen dem Agenten sowohl seine Strategien als auch die Repräsentation dieses Wissens gleichzeitig zu lernen. Beide Verfahren basieren auf dem unüberwachten Lernverfahren Growing Neural Gas, das eine Vektorquantisierung lernt, indem es neuronale Einheiten in Regionen des Eingaberaums platziert, in denen Eingabedaten erwartet werden können. *GNG-Q* gruppiert benachbarte Zustände die das gleiche Verhalten erfordern; *I-GNG-Q* wiederum kombiniert Wissen aus größeren Gegenden der Approximation, um eine gleichmäßige Bewertungsfunktion zu erhalten. Beide Verfahren überwachen das Verhalten des Agenten während der Interaktion mit der Umgebung, um die Stellen der Approximation aufzufinden, die noch verfeinert werden müssen. Die Hauptvorteile der entwickelten Verfahren sind u.a., dass sie kein Modell der Umgebung benötigen und dass die Auflösung der Approximation automatisch und ohne Vorwissen bestimmt wird. Darüber hinaus müssen solche Approximationsmethoden nicht für jedes Zustands-Aktions-Paar Wissen vorhalten womit sie direkt für kontinuierliche Zustandsräume

anwendbar sind.

Die experimentelle Analyse unterstreicht, dass die Verhaltensmuster, die mittels unserer Ansätze gelernt wurden, sehr effizient sind und dass gleichzeitig der Speicherbedarf für die berechnete Approximation deutlich kleiner ist als für eine erschöpfende Repräsentation.

Contents

1 Introduction

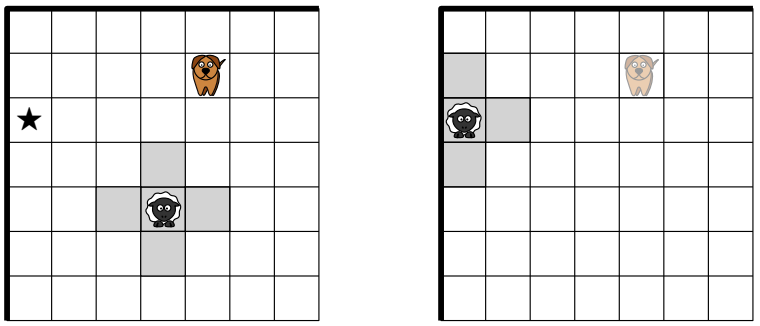
The questions whether robots can *learn* how to successfully control sheep and how well they would perform are one of the two main subjects of this thesis. The second and more vital topic is developing methods that allow agents (as computational model of robots) to store learned knowledge efficiently and investigates how generalizing from historic knowledge can improve the performance of the learning agent.

The dissection of the initially posed questions will reveal the importance and the benefits of dealing with artificial shepherding and point out how "shepherding" strategies could be applied to real world tasks. Additionally, such tasks immediately call for adequate means of storing and (re)using knowledge.

We consider the following task with two kinds of protagonists: The dogs have to drive the reactive sheep to a designated target area (cf. Figure 1.1 that shows a dog, a sheep with its viewing range, and the goal ★). To do so, the dogs approach the sheep, which triggers the sheep's flight instinct. In order to perform well, the dogs have to carefully walk around the viewing ranges of the sheep and only enter it with caution to ensure a controlled movement. This model abstracts from the real interaction between sheep and dogs as it can be found in the real world.

Although shepherding is an interesting task from which we could learn a lot for other real world situations, shepherding has not yet been investigated thoroughly: Many everyday tasks can be formulated in a way such that shepherding strategies (i.e. strategies that enable a robot to control (artificial) sheep) are able to support their solution. The following examples demonstrate alternative applications that could exploit theories developed in this thesis. Remember e.g. the "Deepwater Horizon Oil Spill" (also called "BP Oil Spill") from 2010 where roughly 4.9 million barrels oil were leaked into the Gulf of Mexico (United States Coast Guard, 2011). In addition to the conventional way of using dispersants or controlled burning (Fingas, 2002), small swimming devices could have been used to encircle oil slicks with containment booms. These "oil clusters" could then have been processed by skimmer ships.

A more frequent application is the task of guiding people through e.g. exhibitions or museums as it was tested by Thrun et al. (2000). Such a service robot would



(a) **(b)**

Figure 1.1: The task for the dog is to drive the sheep into the target (★) by reasonably entering its viewing range (grey). Note, that it does not matter where exactly the dog ends up after its task is completed (b).

be able to lead the visitors from exhibit to exhibit while simultaneously offering information about the attraction at hand.

Another particularly useful method is to use such robots to evacuate places, halls or even complete buildings. Deployed (initially passive) robots at relevant places could bee activated in case of an emergency which allows them to help guide people to safety. This idea is supported by research of human crowds whose behavior has been shown to be very similar to that of animal herds (Dyer et al., 2008). Independent of an implementation of the strategies on real robots, such strategies can help to improve the reaction behavior of stewards or other security service personnel. Additionally, the layout of places or buildings may be optimized due to results of this task.

Relying on robots is particularly useful in situations where *real* dogs (that are highly skilled at controlling *real* sheep) cannot be used due to the possibly hazardous surrounding environment: Clearly dogs are not supposed to swim in oil-impacted waters nor to sit for hours in a room waiting for an (hopefully never) occurring crisis.

This thesis uses a multiagent system (Ferber, 1999; Russell and Norvig, 2010) as model for the robots: Such systems consist of entities called *agents* that act autonomous in the system's *environment*. A simple example is a robotic vacuum cleaner (agent) that cleans an apartment (environment). In our scenario, the sheep are modeled with reactive behaviors and for the dog we propose a greedy algorithm as well as different strategies learned with reinforcement learning.

Other authors investigated shepherding tasks using neural networks (Potter et al., 2001), potential fields (Vaughan et al., 1998), rules learned by a genetic algorithm (Schultz et al., 1996), or algorithms using hand-coded strategies (Lien et al., 2004). Unfortunately, existing approaches do neither present provable optimal solutions nor assessments of how good the found solution is. Additionally, no theoretical investigation of the shepherding task or its solutions have been conducted.

This thesis offers a greedy algorithm with linear computational complexity that computes solutions for one sheep and one dog within close proven bounds (i.e. the length of the worst case solution only differs from the lower bound by a term linear in the sheep's viewing range). Furthermore, we model the shepherding task as learning task and compare the learned behaviors with the solutions computed by the greedy

algorithm. Thus, we know how much these strategies possibly deviate from an optimal solution.

With this we are back to our initial questions of whether or not such strategies can be *learned*. When discussing reasons for learning, Russel (2010) argued that "We need learning not only for erudition, but also because it improves our ability to generate effective behavior". In fact, learning is particularly useful if the solution of a task is not known beforehand and thus, i.e. the agent can neither follow a predetermined way nor can it be instructed. Additionally, learned behaviors enable the agent to solve tasks where algorithms or hand-coded strategies are difficult or even impossible to design. This thesis discusses how learning processes might not only generate effective but also efficient behavior.

In our opinion, a very suitable concept is *reinforcement learning* (RL) (Sutton and Barto, 1998) that enables agents to learn how to solve a given task from interaction with an environment. This environment has to offer a *reward function* that states the merit of performing an action in a given state without revealing the "optimal" behavior. The agent's goal is to learn a *policy*—i.e. a mapping from *states* of the environment to *actions* the agent can perform—that maximizes the sum of these rewards. This can be thought of as a teacher that does not tell the correct solution as in supervised learning but provides more feedback than in unsupervised learning. The core idea of reinforcement learning is to advise the agent *what* it should do without telling it *how* to do it. This presents a powerful strategy, similar to learning in nature, i.e. children touch the oven usually at most once, and animals learn by repeating behavior that leads to reward and avoid behavior that leads to punishment. Furthermore, designing a reward function is often more straightforward than presenting samples of how to behave correctly, efficiently, or effectively.

Although reinforcement learning is a very potent method, some challenges do exist. Delayed or sparse rewards pose the first challenge; it refers to the problem of determining which action triggered a later reward. Special attention is required to minimize ambiguity when giving feedback—although this does in general not solve the issue. The second challenge is called exploration vs. exploitation trade-off and refers to the necessary decision whether to explore new strategies instead of following the current best behavior. Exploring new ways potentially allows the agent to improve its behavior and prevents it from getting stuck in local optima. Third, there is the curse of dimensionality, which implies that the search space grows exponentially in number of states and actions. This latter challenge is what this thesis focuses on and aims to overcome with the help of *generalization*.

This concept of generalization can be used to tackle three problems with large or even continuous state spaces: Firstly, at the beginning of learning, the agent usually has no knowledge about the environment and does not know, where the "valuable" states are located in the state space. As the agent generally does not know what it has to learn—the reward function is designed to provide the agent with this information but only during the agent's interaction with the environment—it does not have a clue of how it should behave. Thus, in the beginning the agent usually stumbles in a trial-and-error manner through the environment until it finds states that provide feedback and eventually "lead" to the sought behavior (*1. "search problem"*).

As already said, we focus on the curse of dimensionality which is responsible for

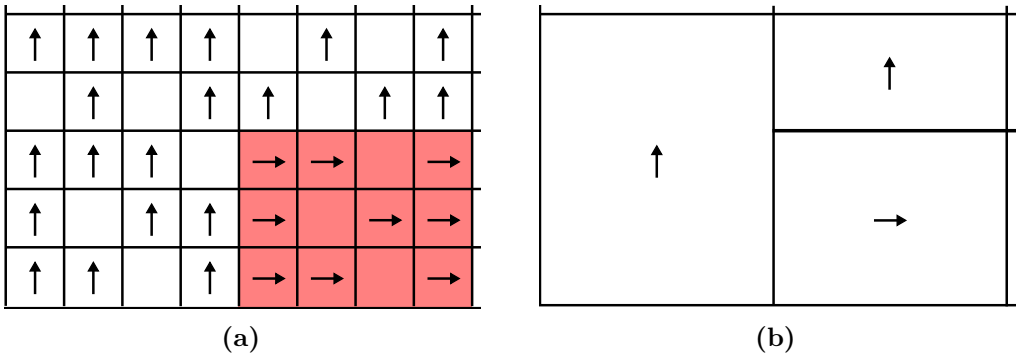


Figure 1.2: Usually, the agent is faced with a situation as in (a): It has knowledge (i.e. a direction like ↑ or →) for some states, only. Generalization offers a means to transfer knowledge to similar states (b).

the other two problems tackled in this thesis: Most reinforcement learning algorithms assume a tabular representation of the acquired knowledge which demands enormous or even infeasible requirements for storage and runtime for the RL algorithms (*2. "computational problem"*). Additionally, the higher the number of states the smaller the probability to experience a particular state more than once—a fact that counteract the need of witnessing the required high number of repetitions of each state-action combination that are needed to derive useful strategies (*3. "repetition problem"*).

In continuous state spaces (i.e. state spaces in which the state description contains continuous values), some form of discretization has to be performed in order to allow the application of table-based algorithms. Here the goal is to find the smallest resolution that is fine enough to capture all necessary details of the environment but that is also coarse enough to allow compact storing. Of course, with domain knowledge, the discretization can be tailored to fit the current state space but usually, this information is not accessible.

A widely used solution for these problems is the concept of generalization that allows to transfer knowledge from a small set of experiences to a larger set of unknown situations. See e.g. Figure 1.2 where the agent has knowledge about some states (depicted by the arrows in Figure 1.2(a)). It can be seen that only the arrows in the red area are pointing right while the others point upwards. Of course, there are some states for which no knowledge is available (the empty cells) but one could assume that these unknown states need the same behavior as their neighbors. Thus, in order to reduce the number of states and to transfer knowledge to unknown states, combining single states to larger groups of states is beneficial (cf. Figure 1.2(b)). With clever chosen approximation methods, the agent is able to use historic experience on states that it has never seen before—a fact that helps to increase the efficiency of the learning process while simultaneously decreasing the amount of needed storage.

This thesis investigates two different kinds of generalization: The first one, *state-space abstraction*, combines "similar" states to derive a smaller, abstract state space. With this approximated (discrete) state space, the agent can then perform standard reinforcement learning approaches even in continuous state spaces. The second field directly approximates the value functions of the reinforcement learning agent and is thus called *function approximation*. Both approaches omit storing values for all

state-action pairs and are thus directly applicable to continuous state spaces.

Possible approaches to aggregate states include tile coding (Sherstov and Stone, 2005; Whiteson et al., 2007; Lin and Wright, 2010), tree-based approaches (Chapman and Kaelbling, 1991; McCallum, 1995) or approximations based on vector quantization (Lee and Lau, 2004). Also, different approaches to approximate the value function exist, including the use of radial basis functions (RBFs) (Menache et al., 2005; da Motta Salles Barreto and Anderson, 2008) or using other approximation methods (e.g. (Konidaris et al., 2011; Whiteson and Stone, 2006)). Unfortunately, these approaches often assume domain knowledge or have computational issues as they have to solve the reinforcement task for different resolutions of the aggregation.

The challenge in generalization is the underlying task of finding a proper representation of the agent's knowledge: For state aggregation methods, the goal is to find an abstract state space that has the minimal number of states while still allowing to find a policy that is (close to) optimal in the *original* state space. Thus, often a large portion of domain knowledge is necessary to obtain useful results. Without domain knowledge, finding an optimal abstract state space is NP-hard (Even-Dar and Mansour, 2003b). The challenge in function approximation is to find parameters for the approximation scheme that allow the agent to learn a proper policy.

In this work we introduce two adaptive approximation methods to offer generalization for reinforcement learning. Both are based upon the growing neural gas (Fritzke, 1994b) that is an unsupervised learning approach that learns a vector quantization by placing units in areas of the input space from which input data can be expected. We extend this approach to correctly approximate the state space in parallel with the agent's trajectories through the environment. The learning of the behavior is done by the well-known reinforcement learning algorithm Q-Learning (Watkins, 1989). During the interaction of the agent with the environment, its policy is monitored to find areas that have to be refined. The result is an adaptive online approximation approach that uses feedback from learning to adjust the agent's knowledge representation.

The first algorithm is called *Growing Neural Gas Q-Learning* (*GNG-Q*) that aims at finding states that are both "similar" (i.e. they are spatial close) and that additionally require the same behavior. Such states are aggregated to abstract states and treated identically. This results in a smaller abstract state space in which Q-Learning can work efficiently and effectively. *GNG-Q* starts with a very coarse approximation consisting of just two abstract states that is refined based on information gathered from the agent's interaction with the environment. For every abstract state, the agent keeps track of how often it has to change its policy and refines abstract states in which too many changes occur. Amongst many others, the core advantages of the *GNG-Q* are that the approach does not need the model of the environment and the granularity of the approximation is determined automatically and does not depend on domain knowledge. Clearly, this approach assumes that the environment contains neighboring states that share the same action—but usually, this is true in many reinforcement learning scenarios in particular in continuous environments like the real world.

In our second approach, we use the growing neural gas to build an adaptive function approximation algorithm called *Interpolating Growing Neural Gas Q-Learning*

(*I-GNG-Q*). Here, we use a feature description for every state of the environment and the agent directly updates its approximation of the state-action value function. The core idea of *I-GNG-Q* is to place units in the domain of the value function (i.e. the state space) and to equip each neuron with a vector of values for the state-action value function at this position. These combinations resemble the data points of the function that should be interpolated (i.e. the value function). For each state description, *I-GNG-Q* selects the value vectors of the k most similar neurons and uses these values to compute the value at this point as a combination that is inversely weighted by the intermediary distances. This approach is very performant as no (rather slow) exponential functions have to be computed. As *GNG-Q*, the *I-GNG-Q*-approach starts with a very "rough" approximation of the value function that only consists of two neurons (i.e. two data points). During the agent's interaction with the environment, the resolution of the value function's approximation is refined by adding neurons in erroneous areas.

The distinction between *GNG-Q* and *I-GNG-Q* is primarily the layout of the resulting value function: While in *GNG-Q* the aggregation of similar states leads to a piecewise linear value function, the value function of *I-GNG-Q* is very smooth. Secondly, *GNG-Q* delivers information about the topology of the state space. On the other hand, *I-GNG-Q* does not equalize states but computes a separate value for every possible state.

To conclude, the parallel learning of behavior and its representation is highly valuable but unfortunately, hard to achieve (van Otterlo, 2009). Nevertheless, the approaches developed in this thesis offer a great way to deal with this issue.

Contributions of This Thesis

This work basically consists of two topics: Analyzing to what extend agents are able to *learn* shepherding strategies as well as developing adaptive approximation methods for reinforcement learning. In the following, we describe the main contributions of this thesis.

Shepherding

For the shepherding task we

- give a formal definition of the Shepherding task in Section 4.3, model the Shepherding task as (multi-)agent system, and relate this system to existing agent system taxonomies in Section 4.4.
- carefully analyze the state-space complexity of the Shepherding task in Section 4.6 and show that the number of states grows exponentially in the number of agents. Furthermore, we show how the dog's shepherding behavior can be analyzed using Manhattan geometry in Section 5.1. To the best of our knowledge, this is the first theoretical analysis on such tasks.
- present the *greedy coordinate correction* (*GCC*) approach that solves obstacle-free instances with one target, one dog, and one sheep in Section 5.2 and prove its linear runtime in Section 5.3.4.
- provide close upper and lower bounds on the solution lengths computed by the *GCC* approach in Section 5.3. For this, the maximal length of an solution

computed by *GCC* as well as the minimal length of any solution are proven. Our analysis reveals that the solutions computed by the *GCC* algorithm are close to the optimal solution, i.e. the upper and the lower bound differ by a term linear in the size of the sheep's viewing range.
- model the SHEPHERDING task as reinforcement learning task in Chapter 6 and compare the learned behaviors to solutions that were computed by the *GCC*-approach in Chapter 10.

Adaptive State Aggregation

The state aggregation approach developed in this thesis, *GNG-Q*, is presented by

- showing how the growing neural gas approach can be extended to work as an online adaptive state aggregation in Section 7.4. Additionally, two different interpretations of the neighborhood connections are discussed in Section 7.5.
- describing a means of monitoring the agent's policy in order to identify areas of the state space that need to be refined.
- introducing the concept of regional states in Section 7.6 that allows a secure adaptation of the approximation although the agent's behavior is a moving target.
- defining a new operation to refine the approximation based on the current error in Section 7.7. Furthermore, Section 7.8 provides criteria for the approximation's refinement and adaptation and argues how these criteria lead to an implicit stopping condition for adjustments.
- analyzing the computational complexity of the proposed approach in Section 7.11 and pointing out limitations of state aggregating approximations.

Adaptive Function Approximation

For the adaptive function approximation approach, *I-GNG-Q*, we

- motivate the use of a distance-based interpolation to respect the distance of a state to its nearest prototype Q-vectors and why it is useful to incorporate several prototype Q-vectors in Section 8.1.
- describe when it is useful to change the approximation and when this adjusting should be stopped in Section 8.3.
- present an *inverse distance weighting* approach to derive a smooth value function in Section 8.4.
- show how the *I-GNG-Q*-approach modifies the agent's policy with an improved update rule that is proven to avoid divergence (Reynolds, 2002) in Section 8.5.
- analyze the computational complexity of the *I-GNG-Q*-approach in Section 8.7 and in Section 8.8 we compare *GNG-Q* and *I-GNG-Q*.

Structure of the Thesis

Before we start presenting our results, we briefly introduce existing concepts in Chapter 2. In particular, we review (multi-)agent systems as the core model used in this thesis and give a short overview of single-agent reinforcement learning. Additionally, the growing neural gas approach is described which is later used to compute the

approximations for the reinforcement learning agent. Chapter 3 surveys related concepts from literature both for shepherding and for approximations in reinforcement learning.

As said before, our main contributions are two fold. We begin with the introduction and the analysis of the shepherding task in Chapter 4. Then, in Chapter 5 we present our greedy solution for tasks with one sheep and one dog, prove bounds on the quality of the solutions computed by this approach, and investigate its runtime. Furthermore, we model the shepherding task as reinforcement learning task in Chapter 6.

The second part of our contribution is centered around approximations for reinforcement learning. Thus, Chapter 7 is devoted to the presentation of our adaptive state-space aggregation method *GNG-Q*, while Chapter 8 introduces our adaptive function approximation approach *I-GNG-Q*. Each of these chapters cover the description of the algorithm, the underlying ideas as well as an analysis of the computational complexity.

Chapter 9 recapitulates what we have achieved in the Shepherding task and in Chapter 10 we experimentally evaluate our approximation approaches and investigate their performance in the shepherding task. Finally, Chapter 11 summarizes the key results of this thesis and points out suggestions for future work.

2 Background

This chapter reviews some background knowledge necessary for this thesis. In particular, Section 2.1 presents the idea of agents and their enclosing environment while Section 2.2 surveys reinforcement learning. Finally, Section 2.3 reviews the growing neural gas algorithm for vector quantization. These concepts are used in the remainder of this thesis: We introduce how to use the growing neural gas approach to obtain an approximation for reinforcement learning and show how this hybrid learning approach can be applied to a shepherding task implemented in an agentsystem.

2.1 (Multi-)Agent Systems

This thesis uses a multiagent system to model the task of robots shepherding flocks of (artificial) sheep. In the following, we concisely introduce agentsystems while for deeper insights we refer e.g. to (Lettmann et al., 2011; Wooldridge, 2009; Ferber, 1999; Russell and Norvig, 2010).

Agents (and agentsystems) can be divided into *hardware agents* and *software agents* (Müller, 1999):

- Hardware agents (e.g. robots) interact with a *physical* environment and are characterized by having a physical representation.
- Software agents (e.g. programs) may interact with either a physical or a *virtual* environment.

As already mentioned above, we use an agentsystem to simulate the task of shepherding and thus, we restrict ourselves to software agents.

Note, that there is a strong connection between agents and environments: Each agent is situated in and interacts (e.g. perceive or change different details) with an environment of some sort. Everything outside that agent is combined in the environment and thus, every agent can perceive other agents (or at least the results of their actions) that are possibly present in the same environment.

Several models to implement multiagent systems exist that—although having various similarities—differ in some details. In this thesis we use agents as representa-

tives of real robots (that in turn are representatives of sheep and dogs) and we do not delve too deeply into agent systems per se. Instead, we here present the ideas of such systems that are needed for this work and refer to (Lettmann et al., 2011) where existing models were compared and a universal sight on multiagent systems was introduced.

2.1.1 Agents

Several definitions of agents exists, we here contrast the most prominent ones, only (for a more thorough comparison, we again refer to (Lettmann et al., 2011)). First, the agent definition of Wooldridge:

> "An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous actions* in this environment in order to meet its delegated objectives." (Wooldridge, 2009)

A different view on agents can be found in Ferber's (1999) definition of agents:

> "An agent is a physical or virtual entity
>
> (a) which is capable of acting in an environment,
> (b) which can communicate directly with other agents,
> (c) which is driven by a set of tendencies (in the form of individual objectives or of a satisfaction/survival function which it tries to optimise),
> (d) which possesses resources of its own,
> (e) which is capable of perceiving its environment (but to a limited extent),
> (f) which has only a partial representation of this environment (and perhaps none at all),
> (g) which possesses skills and can offer services,
> (h) which may be able to reproduce itself,
> (i) whose behaviour tends towards satisfying its objectives, taking account of the resources and skills available to it and depending on its perception, its representations and the communications it receives." (Ferber, 1999)

It is obvious that Wooldridge's agent definition is rather concise while Ferber's definition is very detailed. Nevertheless, it is obvious that these definitions share some commodities: Both include the fact that agents are autonomous, i.e. they are not controlled by some external command but rather follow some objectives or goals. Additionally, both definitions contain the facts, that agents are situated in an environment and that they are able to interact with it by performing various actions.

We here use agents that possess all features of this intersection and—for the investigation of reinforcement learning—require agents that are capable of *learning*.

2.1.2 Environments

Briefly speaking, the environment is everything outside the agent. Nevertheless, the agent is closely connected to the environment as the environment contains the agent. An example of an interaction between an agent and an environment would be a cleaning robot that has to clean the floor of a room: In that scenario, the agent represents the robot and the environment represents the room.

Due to the close connection between agents and their surrounding environment, we continue our introduction with a short insight into this part of an agent system. Russell and Norvig (2010) presented several dimension along which environments could be classified:

Fully Observable vs. Partially Observable In a fully observable environment, each agent can precisely perceive all information about its environment that are needed to fulfill its task. Partially observable environments only offer noisy and/or missing information.

Single Agent vs. Multiagent At first sight, the differentiation between single- and multiagent systems is quite obvious: A multiagent system is an agent system with more than one agent. Surely, for some parts of the environment it can be argued whether or not they are to be considered as agents. Multiagent systems can be further divided into cooperative (e.g. all agents work together in some kind of a team), competitive (e.g. at least some agents have contradicting goals), or mixed (a combination of both) environments.

Deterministic vs. Stochastic In deterministic settings, performing an action in a state always results in the same succeeding state while in stochastic environments the next state is drawn from a (usually unknown) probability distribution.

Episodic vs. Sequential Episodic environments are characterized by the fact that in each episode the agent perceives its environment and performs a single action that does not affect later episodes. Sequential environments on the other hand are more complex. The agent has to think ahead as its actions may influence the future. Note, that the terminology is different to that used in reinforcement learning: There, an episode usually consists of several timesteps in each of which the agent is allowed to perform an action.

Static vs. Dynamic Static environments do not change while the agent is deployed in it while dynamic environments may change—perhaps due to actions performed by the agent. Static environments are obviously easier to deal with as they will not change during the time the agent needs to decide upon its action.

Discrete vs. Continuous The distinction between discrete and continuous is relevant to the perception of the agent (i.e. can its perceptions be described by discrete or continuous variables), the actions (i.e. is there a finite set of actions or not), and the time (i.e. are there distinctive timesteps or is the time continuous).

Known vs. Unknown In known environments, the agent knows how the environment works and has information about the outcome of all its actions before performing them while in unknown environments, the agent has to somehow figure out how the environment reacts to its action.

Obviously, the environment strongly influences the agent and the requirements it has to fulfill in order to successfully work in it. Additionally, the characteristics of the environment determine how hard it is to solve the task.

2.1.3 (Multi-)Agent Systems

Technically, every agent system that consists of more than one agent is considered a multiagent system. Nevertheless, we here only deal with single-agent systems although more agents are present: In this work, only one agent is capable of learning

(the shepherd) and the other agent(s) (the sheep) react(s) to the learning agent's actions.

Thus, it suffices to consider a multiagent system as follows: The environment encapsulates the computation of the sheep's reaction to the shepherd's actions. The shepherding agent is capable of observing the current state of the environment and based on this perception, it selects an action which may change the environment. Note, that the only possibility of performing an action that does not directly change the environment is to stand still as otherwise any movement of any agent changes the state of the environment.

To sum up, we here consider a *fully observable* but *unknown*, *discrete single-agent* system. Whether or not the here employed environment is *deterministic* or *stochastic* as well as *static* or *dynamic* depends on the sheep behavior: A purely reactive sheep that only moves if approached by the agent results in a *deterministic* and *static* environment while sheep that react randomly call for a *stochastic* and *dynamic* environment. In the terminology of Russell and Norvig (2010), the considered environment is *sequential*.

We here omit the definition of a complete formal model as it was presented e.g. by Lettmann et al. (2011) and refer to Figure 2.1 for the parts of an agentsystem that are relevant in this thesis. Note, that in Figure 2.1 the agent is drawn separately from the environment to stress the *interaction* of the agent with the environment.

2.2 Single Agent Reinforcement Learning

"An agent is learning if it improves its performance on future tasks after making observations about the world" (Russell and Norvig, 2010). In this work, we consider one particular type of learning that is called *reinforcement learning*. Especially for learning shepherding behavior, we deal with reinforcement learning tasks with discrete time and actions. This section gives a brief introduction to reinforcement learning in environments with *discrete* state signals while environments with *continuous* state signals are considered in the part of this work that deals with approximation methods.

Reinforcement learning can be considered as a method in between *supervised* and *unsupervised* learning:

- In supervised learning the learner (an agent) has access to labeled data and tries to learn the concept behind these training data. The goal is to interfere a function that maps a given input vector to the correct output vector and that gives correct results even for unknown samples. Generally, in supervised learning one distinguishes classification and regression tasks: Classification deals with assigning a category to a given sample based on the concept derived from the training data while in regression the relationship between variables from the training data has to be transferred to unknown data.
- In unsupervised learning the structure of the completely unlabeled data has to be learned. The learner cannot "compare" its estimate with a correct solution and there is no external error or reward signal that helps the agent to improve its estimation. Here, the goal is to identify patterns depart from unstructured noise (Ghahramani, 2004). One prominent example is *clustering* where samples are grouped according to their similarity that is defined by some distance

measure. Here, it often cannot be checked if the solution is "right" or "wrong".

Reinforcement learning deals with an agent learning which actions to perform in order to fulfill a certain task—solely from interaction with an unknown environment (Sutton and Barto, 1998). This interaction is composed of the agent performing actions and the environment answering with a reward signal. This reward expresses the value of performing this action in the current environment state. The *reward function* has to be designed such that the agent can learn a useful behavior by maximizing these rewards over time without having access to the "optimal" behavior. Hence, the goal of the agent is to learn a *policy*—i.e. a mapping from states of the environment to the agent's actions—that maximizes the sum of rewards over time. This policy then models the behavior of the agent.

In contrast to supervised learning, the agent is not told which actions are advisable; it has to discover their utilities in possible environment states on its own. Comparing to unsupervised learning, in reinforcement learning the agent gets some sort of feedback, although the rewards may be delayed (e.g. awarded after reaching a target or winning a game) and each action performed may of course affect later rewards.

In other words, reinforcement learning only states *what* to do without telling the learner *how* and is thus well applicable if designing a reward function is easier than generating training samples for supervised learning.

2.2.1 Reinforcement Learning Tasks

Reinforcement learning tasks usually consist of

- an environment with a *set of states* S that may e.g. be described by discrete or continuous values.
- a *set of actions* A the agent can perform.
- a function T that models the system's dynamics, i.e. the influence of actions on the states of the environment. This function is called *transition function*.
- a function r that offers the agent a *reward* (positive feedback) or a punishment (negative feedback) for being in a given state of for performing an action in a given state.

Additionally, we assume that the complete system (i.e. environment and agent) has a discrete time signal and that the agent can somehow "see" the state $s_t \in S$ of the environment at time t.

Figure 2.1 shows an example of the agent-environment interaction (Sutton and Barto, 1998) in reinforcement learning: The agent observes the current state s_t of the environment (1) and selects and performs (2) an action a_t. The environment responds (3) with a reward $r(s_t, a_t)$ and transitions (4) into the succeeding state s_{t+1} with probability $\mathrm{T}(s_t, a_t, s_{t+1})$. Some tasks involve delayed rewards, that are e.g. awarded after achieving a goal which makes such tasks hard to solve due to this sparse feedback. In those tasks it is particularly unclear which actions were good and which not.

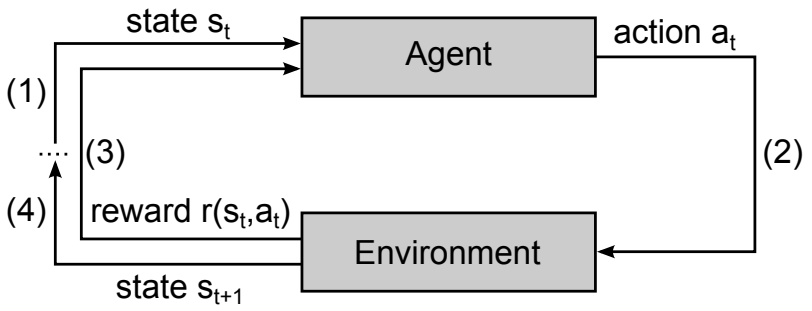


Figure 2.1: Agent-environment interaction, adapted from (Sutton and Barto, 1998)

2.2.2 The Reward Function

A crucial part of each reinforcement learning tasks is the reward function as this function formalizes the task or the goal of the agent. The reward is modeled as a single number and can be positive, negative or neutral. Here, we consider an agent that maximizes its reward and thus, the interpretation of the reward r will be as follows:

$\mathbf{r > 0}$ This is a signal, that actually is considered a *reward* in terms of "gain something". It is appropriate e.g. for winning a game or more generally for reaching a (sub-)goal.

$\mathbf{r < 0}$ A negative reward is considered a *punishment* and is attributed e.g. for losing a game or in a more general sense for "misbehavior".

$\mathbf{r = 0}$ Neutral rewards are used to assess behavior that is neither good nor bad. For example all situations in a game that are not winning or losing may be valued with a neutral reward.

Nevertheless, we always refer to this signal as "reward", even if it expresses a punishment. The usage of such rewards is very convenient as it only demands to rate the agent's behavior instead of specifying samples of the desired behavior beforehand.

The most important aspect of the reward function is that the agent must be able to learn *what* the designer wants by maximizing the rewards over time. Although the reward function may be used to incorporate domain knowledge (Matarić, 1994) it should not state *how* the agent should fulfill its task (Sutton and Barto, 1998). Additionally, great care has to be taken when paying reward for reaching subgoals: On the one hand, this may improve the speed of learning as the agent gets feedback earlier and its learning may become more targeted. On the other hand, this might lead to unwanted behavior: If the reward function is not well designed, the agent may collect a huge cumulative reward by fulfilling subgoals over and over again without ever achieving the actual goal.

In contrast to experiencing rewards in nature, where rewarding signals are at least partially produced *inside* the individual's body, the rewards used here are generated outside the agent in the environment. Thus, the entity giving the rewards is out of the agent's reach to prevent the agent from "manipulating" the overall task. Furthermore, it is somewhat imaginary for an agent to strive for rewards but one can clearly think of an agent that tries to maximize a function (here the reward function).

The agent receives the reward $r(s, a)$ after performing action a in state s. Other definitions may offer a reward for being in a particular state or for a particular transition between two states (van Otterlo, 2009). Clearly, both of these deviations can be expressed by the notation used here.

2.2.3 Returns

After introducing the reward function, we are now going to further specify the objective of learning. As mentioned before, the task of the agent is to learn an optimal policy—but which policy is considered optimal?

Over the time, the agent gathers the value $r_t + r_{t+1} + r_{t+2} + \ldots$ where $r_t = r(s_t, a_t)$ is the reward received at the end of time t after performing a_t in s_t. The value to be maximized is called *expected return* (Sutton and Barto, 1998) and is defined in its easiest form as the sum of every reward received after time t:

$$R_t = r_t + r_{t+1} + \ldots + r_T$$

where T is the final time step (e.g. after reaching some terminal state). This form of return is well applicable in reinforcement learning tasks that have a defined end, e.g. a game that is either won or lost. One iteration of the task from the beginning to the end is called *episode*.

In *continuing* tasks—e.g. control tasks—that have no definitive terminal state this definition is fraught with problems as the final time step as well as the sum of rewards may become infinite. A more general concept in these situations is called *expected discounted return* (Sutton and Barto, 1998) and is defined as:

$$\begin{aligned} R_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \\ &= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \end{aligned} \tag{2.1}$$

with an *discount factor* $0 \leq \gamma \leq 1$. This discount factor is used to rate the current value of a reward that is received in the future. A reward with a value of $\hat{r}$ that is received at time $t + k$ is worth $\gamma^{k-1}\hat{r}$ at the current time t. For $\gamma < 1$, the sum in Equation (2.1) is finite for bounded rewards. If γ approaches zero, the agent is more interested in immediate rewards (i.e. it tries to maximize sooner rewards) while if γ approaches one, the agent tries to maximizes the reward on the long run.

2.2.4 The Markov Assumption

In the most general case, the probability of transitioning from s_t to a particular successor state s' and the reception of a certain reward r after performing action a_t depends on the complete "state-action-reward" history since the beginning of the current episode:

$$\mathbb{P}\{s' = s_{t+1}, r = r(s_t, a_t) \mid s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_{t-1}, a_{t-1}, r_{t-1}, s_t, a_t\} \tag{2.2}$$

A very handy relief—if supported by the environment—is to make the so-called *Markov* assumption that allows that the current state and the current reward only

depends on a finite fixed number of previous states (Russell and Norvig, 2010). With that assumption, it suffices to consider only elements from the k last time steps as the probability of ending up in state s' and receiving reward r after performing action a_t in s_t only depends on the most recent history:

$$\mathbb{P}\{s' = s_{t+1}, r = r(s_t, a_t) \mid s_{t-k+1}, a_{t-k+1}, r_{t-k+1}, s_{t-k+2}, a_{t-k+2}, r_{t-k+2}, \ldots, s_t, a_t\} \tag{2.3}$$

The simplest form are first-order Markov processes (as often used in reinforcement learning (Sutton and Barto, 1998; Mitchell, 1997)) where the succeeding state and the received reward only depends on the current state s_t and the action a_t performed in it:

$$\mathbb{P}\{s' = s_{t+1}, r = r(s_t, a_t) \mid s_t, a_t\} \tag{2.4}$$

Signals for which Equation (2.2) is equivalent to Equation (2.3), are called kth-order Markov processes (Russell and Norvig, 2010). In the reinforcement learning literature it is often assumed that the current state contains enough information to completely disregard the past by only considering state and reward signals that are first-order Markov processes (i.e. processes for which Equation (2.2) equals Equation (2.4)). In this work, we also focus on first-order Markov processes.

An environment has the Markov property if its responses (i.e. the succeeding state as well as the reward) are Markov processes (Sutton and Barto, 1998). Such environments allow the agent to select an action based on its current observation without knowing the complete history of states, actions and rewards it has received before. Summing up, we can now introduce the Markov decision processes (MDP) (Puterman, 2005), a common model for dealing with reinforcement learning tasks:

Definition 1 (Markov Decision Process). *A Markov decision process (MDP) is a tuple* $\mathrm{M} = (S, A, \mathrm{T}, r)$ *where the transition function* $\mathrm{T} : S \times A \times S \to [0, 1]$ *is a probability distribution over the succeeding states, i.e.* $\mathrm{T}(s_t, a_t, s_{t+1})$ *is the probability of transitioning to state* s_{t+1} *after performing action* a_t *in state* s_t*. To guarantee a proper probability distribution,* $0 \leq \mathrm{T}(s, a, s') \leq 1$ *has to hold for all states* s, s' *and all actions* a*. Additionally,* $\sum_{s' \in S} \mathrm{T}(s, a, s') = 1$ *has to hold for all states* s *and all actions* a*. The reward function* $r : S \times A \to \mathbb{R}$ *reflects the immediate merit* $r_t = r(s_t, a_t)$ *of performing* a_t *in* s_t*.*

Note that a deterministic transition function can be described by $\mathrm{T}(s, a, s') = 1$ for at most one s' for all state-action pairs (s, a) and zero for all other s'. In these settings, we write $\mathrm{T}(s, a) = s'$ for the state s' having $\mathrm{T}(s, a, s') = 1$.

Summing up, the key advantage of Markovian environments is that each time the agent performs a specific action a in one particular state s, the outcome depends on the same probability distribution every time the last k time steps were identical. In the most common model of first-order Markov environments the probability distribution for the succeeding state only depends on the current state and action. For deterministic MDPs, the current state together with the current action definitively determines the following state regardless what has happened before.

A short example for the Markov property is the following: Consider a door that opens only after a button has been pressed before. So, in order to go through that door, the agent has to first press the button and can then approach the door. An

environment that only offers a state perception that contains the current position of the agent is not Markovian: The agent would know where it is and would thus know if it is close to the door but it could not know if the button was pressed before (if it was not pressed indeed or the agent had not stored this information in some way). In a Markovian environment, the description of the current state contains the information of whether or not the button was pressed and then, the agent can actually learn the concept of passing through that door.

As mentioned before, the agent's policy is a function that "contains" the behavior of the agent. It does so by mapping a state s to an action a which tells the agent what to do in the current situation. Thus, the policy has to be defined over all states the agent can possibly perceive to allow for a correct behavior in all circumstances. Usual models for the policy are look-up tables or, when the amount of possible states renders this approach infeasible, other means of storage have to be considered. Although it is possible to define a *stochastic policy*[1], we here only consider a *deterministic policy* $\pi : S \to A$ that is used by the agent to determine which action $a_t = \pi(s_t)$ to perform in a given state s_t.

The agent's goal is to learn an *optimal* policy to make a useful action selection based on the current state of the environment. This policy should not focus on the immediate rewards the agent may receive but maximize the expected discounted return in order to reach the overall goal. To get a more precise view on what the agent should maximize, reinforcement learning makes use of so-called value functions.

[1] A stochastic policy $\pi : S \times A \to [0, 1]$ expresses the probability of performing action a in s.

2.2.5 Value Functions

Reinforcement learning algorithms usually center around functions that estimate the *value* (i.e. the accumulated expected discounted return) of being in a particular state (state-value functions) or the value of performing a particular action in a given state (action-value functions).

State-Value Function

We start by introducing the *state-value function* that denotes the value of a state s if the agent starts its trajectory in s and follows the policy π afterwards. This value uses the expected return given in Equation (2.1). If the reinforcement learning task is modeled as MDP, the state-value function is defined (Sutton and Barto, 1998) as

$$\begin{aligned} V^{\pi}(s) &:= \mathbb{E}_{\pi}\big[R_t \mid s_t = s\big] \\ &= \mathbb{E}_{\pi}\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} \;\middle|\; s_t = s\right] \qquad (2.5) \\ &= \mathbb{E}_{\pi}\left[r_t + \gamma \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \;\middle|\; s_t = s\right] \\ &= \mathbb{E}_{\pi}\Big[r_t + \gamma V^{\pi}(s_{t+1}) \;\Big|\; s_t = s\Big] \\ &= \sum_{s' \in S} \mathrm{T}(s, \pi(s), s') \Big(\underbrace{r(s, \pi(s))}_{r_t} + \gamma V^{\pi}(s') \Big) \qquad (2.6) \end{aligned}$$

where $\mathbb{E}_{\pi}[\cdot]$ is the expected value for the returns produced by starting in state s and following π at any timestep t. Note, that the expectation operator is only necessary in stochastic environments as then $\mathrm{T}(s, a, s')$ and/or $r(s, a)$ may have several outcomes (i.e. the agent may end up in different states s' after performing action a in state s and/or receive different rewards for the same transition). For deterministic environments (i.e. environments where performing an action a in a state s always results in a transition to only one state s' and an unambiguous reward), Equation (2.5) simplifies to

$$\begin{aligned} V^{\pi}(s) &:= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \qquad (2.7) \\ &= r_t + \gamma \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \\ &= \underbrace{r(s, \pi(s))}_{r_t} + \gamma V^{\pi}(s_{t+1})\,. \end{aligned}$$

With Equation (2.5) (or Equation (2.7) respectively), we can now precisely state the learning goal of the agent. It has to learn an *optimal policy*

$$\pi^{\star} = \arg\max_{\pi} V^{\pi}(s), \forall s \in S$$

that maximizes the sum of discounted rewards over time. Simultaneously, the *optimal state-value function* is derived by following the optimal policy:

$$V^\star(s) = \max_\pi V^\pi(s)\,.$$

The optimal state-value function $V^\star(s)$ is the maximal accumulated expected discounted return the agent can collect by starting in state s and following an optimal policy in any state of its trajectory.

With this state value function, we can now describe an optimal policy for the agent in more detail:

$$\pi^\star(s_t) = \arg\max_a \left(r(s_t, a) + \gamma V^\star(s_{t+1})\right) \tag{2.8}$$

Unfortunately, to compute this optimal policy, the agent would need to have access to the reward function and the state-transition function to compute the outcome of its actions—an assumption that is unfeasible in most reinforcement learning tasks. A possible remedy is the use of the action-value function described next that can e.g. be learned by Q-Learning (Watkins, 1989).

Action-Value Function

Since the agent chooses an action in a given state, it can use the *action-value function* $Q(s_t, a_t)$ that expresses the expected accumulated reward for performing action a_t in state s_t and then selecting the following actions from an optimal policy afterwards:

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma V^\star(s_{t+1}) \tag{2.9}$$

Thus, the function $Q(s, a)$ covers the immediate reward assigned by the environment for performing the action a in state s and the by γ discounted return that can be achieved if the agent adheres to an optimal policy thereafter.

Comparing Equation (2.8) and Equation (2.9), it can be seen, that the right-hand part of Equation (2.9) is exactly what is maximized in Equation (2.8):

$$\pi^\star(s_t) = \arg\max_a Q(s_t, a_t) \tag{2.10}$$

Thus, an optimal policy $\pi^\star$ for the agent can be derived by learning an optimal action-value function Q without the need of knowing the reward function or the state-transition function. Instead, the agent can behave optimally by choosing an action a that maximizes $Q(s, a)$ in the current state s. This is of course only possible in a Markov decision process as defined in Definition 1 that presents the agent with sufficient information in the state description s.

A big advantage of the action-value function Q is that it allows the agent to learn a globally optimal policy by repeatedly updating a local part $Q(s, a)$ of the function (Mitchell, 1997). This is exactly what Q-Learning (Watkins, 1989) does to obtain an optimal policy for the agent.

In the following we denote by Q and V the actual—but usually unknown—value functions. The value of a state if the agent selects all its actions from a policy π is denoted by V^π or by $V^\star$ if the policy is optimal. Similarly, $Q^\star$ is the action-value

function when following an optimal policy. During learning (especially during the execution of Q-Learning), the agent only possesses an estimation $\widehat{Q}$ of the original action-value function.

Summing up, the goal of the agent is to learn an action policy that maximizes the total reward it will receive from any starting state. This can be done in several ways, but we here focus on Q-Learning (Watkins, 1989).

2.2.6 Overview of Approaches that Solve Reinforcement Learning Tasks

As discussed before, the agent's goal is to maximize these rewards over time. A great variety of methods to solve such RL tasks exists (see e.g. (Sutton and Barto, 1998) or (Szepesvári, 2010) for comprehensive and in-depth surveys of relevant approaches). We here just mention the most prominent ones.

If the agent has access to all information of the reinforcement learning and especially to the reward function and the state-transition function, an optimal policy can be computed offline by using *dynamic programming* to solve the associated Bellman equations (van Otterlo and Wiering, 2012). Usually, either the policy or the value function is iteratively evaluated and improved to derive the optimal value function. Fundamental downsides of this approach are the enormous time- and space-consumption for realistic reinforcement learning models and, most prevailing the absence of knowledge of the underlying model.

Monte Carlo methods do not require knowledge of the reinforcement learning task's model but instead learn from interaction with the environment. They do so by assuming the expected return to be a random variable to approximate the value function V by averaging rewards obtained by the agent after the end of an episode. As the agent uses a fixed policy, it may not experience sufficiently many state-action transitions per se but needs to sample every transition possible with a non-zero probability (Sutton and Barto, 1998).

A third category of approaches is called *temporal difference* methods that iteratively reduce the difference between estimates for any state-action pair over time. Temporal difference learning does not need the model of the environment and updates the agent's approximation based on experiences like Monte Carlo methods. These updates are similar to the idea of updates in dynamic programming approaches as they use intermediate estimates after each step instead of waiting for the actual return that is used by Monte Carlo methods (Sutton and Barto, 1998). Especially, in temporal difference approaches the updates of the agent's estimates are partly based on other, previously computed estimates.

Reinforcement learning approaches can be classified along several dimensions of which the most prevailing are *model-free vs. model-based*, *on-policy vs. off-policy* and *online vs. offline* approaches.

In a model-based setting, the agent has access to the underlying model—i.e. to the reward function and the state-transition function—of the reinforcement learning tasks as e.g. employed in dynamic programming approaches. Model-free approaches on the other hand assume no such knowledge which forces the agent to start learn everything from interaction with the surrounding environment. This is usually the case for Monte Carlo or temporal difference methods.

Online learning approaches as e.g. temporal difference methods update the

agent's behavior during its interaction with the environment (or they are at least able to do so) while methods that compute the value functions based on a complete given model of the environment are called offline approaches. Methods from the first category are able to adjust the agent's behavior during the agent's deployment while methods from the second category usually need a new simulation on the adjusted environment model.

The distinction between on-policy and off-policy methods boils down to the question of which policy is affected by performed updates to the approximation: On-policy algorithms as e.g. SARSA (Rummery and Niranjan, 1994) update the value function along the currently followed policy while off-policy methods update the policy that the agent currently assumes to be the best (e.g. Q-Learning). The difference between these two characteristics is plainest in the presence of *exploration*[2]: On-policy methods update the estimate for the state-action pair that actually was experienced while offline methods would update the state-action pair that would have been experienced if the agent had followed the best policy even if the agent did not follow this policy at the current timestep.

2.2.7 Q-Learning

In this thesis, we use Q-Learning (Watkins, 1989), one frequently employed algorithm to learn an optimal policy by incrementally updating an estimation $\widehat{Q}$ of the action-value function Q during interaction with the environment (and which is thus a model free algorithm). The learning is done by iteratively reducing the differences between the Q-value of the current state s_t and that of the succeeding state s_{t+1} and thus, Q-Learning falls into the class of temporal difference learning (Mitchell, 1997) as these values are estimated at different times. Additionally, Q-Learning is an off-policy method as it always updates the agent's estimates according to the policy that is currently optimal.

The key idea of Q-Learning (Watkins, 1989) is the recursive character of the action-value function Q: Given $Q(s, a)$, $V^\star(s)$ can be derived as

$$V^\star(s) = \max_a Q(s, a)$$

and thus, the Q-function given in Equation (2.9) can be expressed solely based on Q itself:

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a) \tag{2.11}$$

After learning, the optimal action-value function $Q^\star(s, a)$ tells the agent the value of performing action a in state s and then following an optimal policy. Thus, the Q-function provides the agent with an ordering of its actions in a given state regarding its value. Even during the learning process, i.e. when the agent may not have obtained the final estimates for the action-value function, the agent can derive a greedy policy from its current estimates by performing the action with the highest value.

The estimation $\widehat{Q}$ of the agent is initially set to zero (or random values) for all state-action pairs and then updated based on the agent's interaction with the

[2] Exploration occurs if the agent sometimes tries an action that deviates from the current policy to get broader knowledge of the environment.

environment: In each timestep t, the agent observes the current state s_t and selects (e.g. with a greedy policy as described above) an action a_t which is the being performed. After performing this action, the agent receives the immediate reward $r(s_t, a_t)$ and perceives the succeeding state s_{t+1} (cf. Figure 2.1). With these information, the agent updates its estimate for the state s_t:

$$\widehat{Q}_t(s_t, a_t) = r(s_t, a_t) + \gamma \max_a \widehat{Q}(s_{t+1}, a) \tag{2.12}$$

This rule closely resembles Equation (2.11) but Equation (2.12) works solely on the agent's estimation $\widehat{Q}$ while Equation (2.11) is defined over the actual action-value function Q that is in general unknown to the agent and is the target of the estimate $\widehat{Q}$.

Note, that the update rule in Equation (2.12) only works for deterministic environments (Mitchell, 1997). If the transition function and/or the reward function are non-deterministic (but still Markovian) their unstable outcomes may lead to oscillating Q-values which would prohibit the agent to obtain a useful policy. In Equation (2.6) we already saw the state-value function for non-deterministic MDPs and similarly, the following action-value function for non-deterministic MDPs can be formulated:

$$Q(s, a) = \sum_{s' \in S} \mathrm{T}(s, \pi(s), s') \Big(r(s, \pi(s)) + \gamma \max_{a'} \widehat{Q}(s', a') \Big) \tag{2.13}$$

where $\mathrm{T}(s, a, s')$ is the probability that the state s' is reached after performing action a in state s. To apply Q-Learning in non-deterministic environments, a more general update rule (that is also applicable in deterministic environments) can be used (Watkins and Dayan, 1992):

$$\widehat{Q}_{t+1}(s_t, a_t) := (1 - \alpha_t)\widehat{Q}_t(s_t, a_t) + \alpha_t \big[r(s_t, a_t) + \gamma \max_{a' \in A} \widehat{Q}_t(s_{t+1}, a') \big] \tag{2.14}$$

with the learning rate $\alpha_t \in (0, 1]$ which is the most prevailing difference to the update rule in Equation (2.12). Its purpose is to decrease the impact of updates over time and thus, a learning rate α_t for each state-action pair could e.g. be chosen to diminish as the number of updates for that pair increases. If α was set to one, Equation (2.14) would turn into the learning rule given in Equation (2.12).

Notice, that although both Q-updates make use of the reward and the succeeding state, the agent does not need to know the reward function or the transition function: The update for the state-action pair (s_t, a_t) is performed after transitioning to the succeeding state s_{t+1} and receiving the reward $r(s_t, a_t)$. The complete Q-Learning algorithm can be found in Algorithm 1.

Q-Learning is proven to converge to the true Q-function given that each state-action pair is updated infinitely often, an exact representation of the policy is used (i.e. tabular with one cell for each state-action pair), the rewards for each state-action pair are bounded, a discount factor $0 \leq \gamma < 1$ is used, and the learning rate α_t fulfills $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$ (Watkins and Dayan, 1992).

An additional advantage of Q-Learning is the fact that it can be used to learn behavior even if the learning agent has no information about the effects of its actions (Mitchell, 1997). Nevertheless, Q-Learning shares problems that are common to reinforcement learning tasks and that are discussed later in Section 2.2.9.

Algorithm 1: Q-Learning (Watkins and Dayan, 1992)

$\forall s, a$ initialize $\widehat{Q}(s, a)$ to zero
loop
  observe state s
  select action a and execute it
  receive immediate reward r
  observe succeeding state s'
  update $\widehat{Q}(s, a)$:

$$\widehat{Q}_{t+1}(s_t, a_t) := (1 - \alpha_t)\widehat{Q}_t(s_t, a_t) + \alpha_t[r(s_t, a_t) + \gamma \max_{a' \in A} \widehat{Q}_t(s_{t+1}, a')]$$

2.2.8 Exploration

From the learned Q-function, it is easy to derive an optimal (or during learning a greedy) policy by performing the action with the highest Q-value in each state. To avoid local optima, it is crucial to try actions other than those with the highest Q-value during learning, i.e. to *explore* different behaviors: In the beginning of learning, the agent usually has no knowledge about how to act correctly (i.e. all entries of its Q-table are zero or randomly set to small values) and thus, it "wanders" through the environment in a trial-and-error manner. After receiving some rewards, always sticking to a greedy policy would lead the agent to fixating on these early found solutions which would in turn increase the Q-values of the affected transitions and ignore Q-values of the rest.

Surely, this may lead to a solution but it may only be a local optimum of the value function: As the first steps of the agent in the environment are basically a random walk, it is highly unlikely that the first found solution is indeed optimal. Instead, the agent should try variations of solutions found so far which will most probably lead to better and eventually to optimal solutions. In fact, the aforementioned convergence proof for Q-Learning only requires sufficiently many visits of each state-action pair and makes no assumption of the order in which these pairs are visited. Q-Learning is thus exploration-insensitive, meaning that the optimal solution can be found using an arbitrary exploration method as long as the criteria stated above are met (van Otterlo, 2009).

We here use ε-greedy action selection, i.e. with probability $1 - \varepsilon$ the action with the highest Q-value and with probability ε, a random action is performed. Other approaches implement exploration e.g. by performing an action a in a state s with a probability that partly depends on the Q-value $\widehat{Q}(s, a)$. This approach uses a so-called Boltzmann (or softmax) distribution that assigns higher probabilities to actions with high Q-values while assuring that every action has a non-zero probability. For other approaches we refer e.g. to (Wiering, 1999).

2.2.9 Challenges in Reinforcement Learning

Unfortunately, besides the benefits of reinforcement learning in general and Q-Learning in particular, this concept of learning faces some challenges:

Delayed Reward The agent gets rewards after each state transition but these rewards may be mostly neutral. This means that the agent cannot always clearly associate the received reward with a particular state-action pair: Although a reward given at timestep t was triggered by the action in that timestep but all previous actions may have influenced this reward, too. In an extreme case, the agent only gets a positive reward after the episode is over and—especially in the beginning of learning—the agent does not know, which actions eventually have led to a valuable outcome.

Exploration vs. Exploitation During learning, the agent has to balance the urge to gather lots of returns (i.e. to exploit its currently learned behavior by selecting actions that offer the highest outcome) and the need to explore the environments to possibly find an even more valuable behavior by also selecting actions that initially do not look as promising as the greedy policy. While only using the greedy policy would most probably restrict the agent to finding a local optimum, only choosing exploratory actions would hinder the agent to collect sufficient returns. Thus, the agent has to appropriately trade-off its exploration and exploitation.

Curse of Dimensionality The "curse of dimensionality" (Bellman, 1957) denotes the fact that the search space grows exponentially in the number of states and actions as this space is generally composed as Cartesian product of all dimensions. This implies a tremendous demand for storage (remember that e.g. the convergence proof for Q-Learning requires that the agent can store its behavior in a table-based format) and usually long times for the learning as the agent has to visit each state-action pair sufficiently often.

Partially Observable States Most algorithms assume that the agent has perfect knowledge of its current state. While this assumption is pretty convenient to analyze the algorithms theoretically, in practice it is often hard to present the agent (or the robot) with means to observe its environment without any noise or distortion. The "worst case" is a Partially observable MDP (POMDP) (Kaelbling et al., 1998).

The next section reviews vector quantization and presents the growing neural gas; an approach which can be used to compute an approximation of the input space.

2.3 Growing Neural Gas for Vector Quantization

In this section, we present *vector quantization* as one method to reduce data and we introduce Fritzke's *growing neural gas* (Fritzke, 1994b) to adaptively compute the centers of a Voronoi partition that is one method to accomplish this. We will come back to these ideas in Chapter 7 and Chapter 8 as the growing neural gas (GNG) approach forms the base of the approximation algorithms developed in this work.

2.3.1 Vector Quantization

Vector quantization is a data compression method often used in communication or storage (Gray, 1984) that eliminates the necessity of transmitting or storing every signal from a possibly continuous sequence (e.g. speech or image data). It uses a finite set

$$C = \{\vec{w}_0, \ldots, \vec{w}_m\}$$

called *codebook* of *codeword* vectors $\vec{w}_i \in \mathbb{R}^d$ to map input vectors to the scalar index of a codeword and thus reduces the problem of dealing with vectors to dealing with scalars instead[3].

In a *nearest neighbor* or *Voronoi* quantizer, every codeword is the representative for all input vectors that are closer to it than to all other codewords. Consequently, this introduces a *Voronoi region*[4] for every codeword $\vec{w}_i$ (cf. Figure 2.2) that consists of all points that are closer to its generating codeword $\vec{w}_i$ than to any other codeword:

$$\mathfrak{R}(\vec{w}_i) = \{x \in \mathbb{R}^d \mid \mathrm{d}(x, \vec{w}_i) \leq \mathrm{d}(x, \vec{w}_j)\ \forall \vec{w}_j \in C\}$$

using e.g. the Euclidean distance

$$\mathrm{d}(u, v) = \|u - v\|_2 = \sqrt{\sum_{i=1}^{d} (u_i - v_i)^2} \tag{2.15}$$

to compute the distance between two d-dimensional vectors u and v. Thus, a nearest neighbor quantizer partitions the input space into disjoint Voronoi regions.

After defining the codewords, any vector x from the input space is mapped with the nearest neighbor rule

$$\mathrm{nn}(x) = \arg\min_{\vec{w}_j \in C} \mathrm{d}(x, \vec{w}_j) \tag{2.16}$$

to its closest codeword that is called *nearest neighbor* of x. Thus, the approximation consists of the codebook and the function nn that maps all input vectors to their codeword (Linde et al., 1980).

Unfortunately, such compressions cause distortion as information of the "real" inputs is lost. The example in Figure 2.2 points out advantages and drawbacks of vector quantization: The vectors x_1 and x_2 are mapped to the same codeword i as they are both in the region $\mathfrak{R}(i)$. The codeword i is obviously a good estimation for x_1 as the distance $\mathrm{d}(x_1, \vec{w}_i)$ is small and thus, the distortion is neglectable. The vector x_2 on the other hand has a much larger distance to i and, although being closer to i than to any other codeword, it is also very close to the border of the neighboring codeword j. Thus, the distortion caused by this vector is much larger.

Usually, a codebook C should minimize the *overall squared quantization error* for a set of datapoints D:

$$E(D, C) = \sum_{x \in D} \mathrm{d}(x, \mathrm{nn}(x))^2 \tag{2.17}$$

[3] In the following we will use the term "codeword i" to refer to the vector $\vec{w}_i$.

[4] Throughout this work, we will use the term *region* to refer to all positions that are closer to the center (e.g. the codeword) of this region than to any other center even though the polytope will not be flat for state spaces with more than two dimensions.

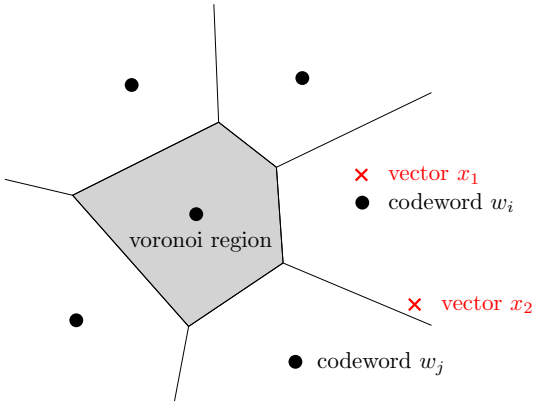


Figure 2.2: Example codebook consisting of six codewords (•) together with two sample vectors x_1, x_2 (×) from the input space. The lines indicate the borders of the induced Voronoi regions defined by the codewords.

The goal of a useful quantization is to distribute the codewords in a way that the overall quantization error is minimized. Unfortunately, this is often made difficult by the distribution of the input data: For every distribution of the input data several optimal quantizer layouts may exist. Usually, the distribution is unknown and thus, an approximation of the optimal codebook has to be computed. One widely used approach is the LBG algorithm (Linde et al., 1980) that is named after its inventors. However, this approach is computationally expensive as it requires several iterations over a set of training vectors. The following section presents an approach that *iteratively* learns the centers of a Voronoi quantizer.

2.3.2 Core Idea of Growing Neural Gas

The growing neural gas (GNG) algorithm (Fritzke, 1994b) is an unsupervised learning approach which employs a network of interconnected units. These units are called *neurons* and are successively inserted in an initially small network according to samples from the input space. Additionally, existing neurons are moved to better match the distribution from which the input data is sampled. Each neuron n is assigned a *reference vector* $\vec{w}_n \in \mathbb{R}^d$ that resembles—in the here considered application—the codewords described before, and a local error variable $\text{error}(n)$. The goal of GNG is to place neurons in areas of the input space where most data can be expected to minimize the quantization error.

The GNG approach is based upon three principles (Fritzke, 1998):

1. The information accumulated in $\text{error}(n)$ respects the overall optimization goal.
2. The error measure is chosen such that the insertion of a new neuron decreases the error in its proximity.
3. The largest decrease in the error can be expected if the new neuron is inserted in a region with maximal local error.

The general approach of GNG is as follows: It starts with a minimal network consisting of two randomly positioned neurons and iteratively it adds new neurons in areas where the greatest reduction of the distortion can be expected.

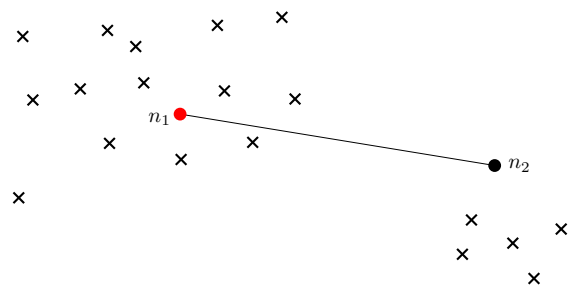


Figure 2.3: The neuron n_1 on the right (red) has a high error value as it has been the nearest neuron for many inputs of which some have a relatively large distance to the neuron.

In each step of training, a randomly chosen vector x from the input space is presented to the network and the neurons $n_1 = \mathrm{nn}(x)$ with the nearest and $n_2 = \mathrm{nn}_2(x)$ with the second nearest reference vector are computed:

$$\mathrm{nn}(x) = \underset{n \in N_t}{\arg\min}\ \mathrm{d}(x, n) \tag{2.18}$$

$$\mathrm{nn}_2(x) = \underset{n \in N_t \setminus \{\mathrm{nn}(x)\}}{\arg\min}\ \mathrm{d}(x, n) \tag{2.19}$$

where N_t is the current set of neurons. Note, that we often talk about a *neuron* n but depict the neurons reference vector $\vec{w}_n$ as this is its position. Additionally, we use the notation $\mathrm{d}(x, n)$ to refer to the distance between some vector x and the reference vector of neuron n, i.e. $\mathrm{d}(x, n) := \mathrm{d}(x, \vec{w}_n)$.

Obviously, nn resembles the nearest neighbor rule from Equation (2.16) and nn_2 can be considered as a second application of the nearest neighbor rule that ignores $\mathrm{nn}(x)$. The neurons $n_1 = \mathrm{nn}(x)$ and $n_2 = \mathrm{nn}_2(x)$ are called *nearest* and *second nearest* neurons to x. This notion of nearest and second nearest neuron is in terms of the similarity measure (e.g. based on the Euclidean distance) in the state space while the terms neighbors and neighborhood are defined over the topology induced by the connections of the neurons in the growing neural gas.

2.3.3 Error Measure

As mentioned before, each neuron is assigned a variable $\mathrm{error}(n)$ that accumulates local error information for n's region. This error is used by the GNG approach to detect regions that have to be refined by inserting additional neurons into the network. This error has to respect the optimization goal and it shall be locally reducible by adding a new neuron in that region (Fritzke, 1996). The concrete design of the error measure depends on the purpose of the growing neural gas: For the here employed vector quantization, the quantization error (cf Equation (2.17)) is most appropriate while for classification or entropy maximization other measures should be considered (Fritzke, 1996).

The error can become especially high if e.g. a neuron n "is responsible" for a large portion (either by size of the space or by the number of samples) of the input space. Then again, input vectors may have a large distance to its closest neuron, which introduces a large distortion, too. To respect this, the error variable of the

nearest neuron is increased by the squared distance between x and n_1's reference vector to incorporate the distortion caused by the recent sample:

$$\Delta\text{error}(n_1) = \mathrm{d}(x, n_1)^2 \,.$$

Regions with larger error values can then be easily identified and relieved by the insertion of a new neuron. For example, neuron n_1 (red) in Figure 2.3 has a much larger error value than neuron n_2 as it has been the nearest neuron n_1 for more input vectors and some of them have a relatively large distance to it. Thus the distortion in this region is large which results in a large error value.

The neurons' error values endure over several training cycles and "memorize" where errors occur. To emphasize recent errors, after each training step all error values are multiplied with a factor $\beta \in (0, 1)$ to obtain an exponential decay.

2.3.4 Neighborhood Connections

If the two nearest neurons n_1 and n_2 to a sample x are not connected yet, a *neighborhood connection* is established and n_1 and n_2 become topological neighbors. Each connection is an undirected edge $\{n_u, n_v\}$ that is equipped with an *age* to remove outdated connections. If the nearest neurons to the current sample are already connected (i.e. they were the nearest neighbors to a sample presented before), the age of that connection is reset to zero. By way of illustration one can think of a decreasing strength for each connection that is renewed every time the adjacent neurons are the closest neurons to an input sample.

These connections form the edges of the induced *Delaunay* triangulation that is the dual graph to the Voronoi tesselation: In this triangulation, every pair of nodes (here neurons) that have neighboring regions is connected with an edge. In the GNG approach, these connections are established during learning and thus only consider the layout present at the time of creation. Over time, regions that were neighboring can of course become separated by the adaptation of the network (cf. Section 2.3.5). In fact, the induced Delaunay triangulation in the GNG is a slowly moving target (Fritzke, 1994b).

A connection between two neurons implies that these two neurons had the closest and the second closest distance to a sample during learning. Thus, their respective regions were neighboring at the time the connection was created. As the neurons are moved during learning and additional neurons are inserted into the network, the layout of the regions changes. To consider this, the ages of the connections is used: If the age of a connection $\{n_u, n_v\}$ exceeds the predefined maximal age this is evidence, that the layout of the network has changed in a way that the respective edge is no longer part of the Delaunay triangulation. These overage connections are removed which can cause isolated neurons that are removed as well. Note, that it is always assured that at least two neurons remain in the network.

The approach to handle the aging is as follows: In every training step, the age of all connections emanating from the nearest neuron n_1 is increased while the age of the connections between n_1 and n_2 is reset to zero. Thus, every time, two neurons are the nearest and the second nearest neurons to one sample, the connection between them is renewed while all other edges are aged.

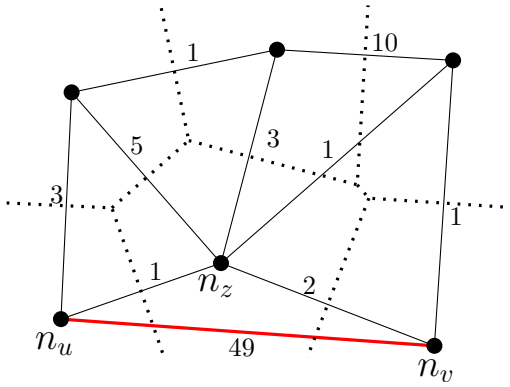


Figure 2.4: Layout of a network with six neurons (•), solid lines depicting neighborhood connections, and dotted lines being the borders of the Voronoi regions. The numbers labeling the connections are the ages. It can be seen, that the connection between n_u and n_v has a high age: Thus, the two neurons' regions were neighboring at some point in time (i.e. they were the nearest and the second nearest neuron to at least one sample) but due to changes in the layout the regions are no longer neighboring. Thus, the connections between n_u and n_v will be deleted soon.

Figure 2.4 shows an example of how the connections' ages are used to detect outdated edges: The regions of the neurons n_u and n_v were the nearest and second nearest neighbors to some input. Now, after refinement and movement, the region $\mathfrak{R}(n_z)$ lies in between. The age of the edge $\{n_u, n_v\}$ is increased every time $n_u = \text{nn}(x)$ is the nearest neuron to an input x while n_v is not the second nearest neuron (or vice versa). Thus, the age of this edge is never reset to zero and after age_{max} training rounds in which exactly one neuron of n_u and n_v was the nearest neuron to some input x, the edge is removed.

2.3.5 Adaptation of the Network

The idea of the adaption that moves the nearest neuron and all of its topological neighbors towards the current sample is to position the neurons in areas of the input space from which data can be expected. Neurons in those "producing" subspaces help to reduce the distortion and thus to minimize the quantization error. This is especially useful in high-dimensional spaces in which the data only comes from a low-dimensional subspace.

The movement is carried out by moving the reference vector of the nearest neuron n_1 towards the sample x by moving it by

$$\Delta\vec{w}_{n_1} = \epsilon_b(x - \vec{w}_{n_1})\,. \tag{2.20}$$

Furthermore, the reference vectors of all neighbors j (defined by the neighborhood connections) of n_1 are moved by

$$\Delta\vec{w}_j = \epsilon_n(x - \vec{w}_j) \tag{2.21}$$

towards x with movement strengths $0 \leq \epsilon_n \ll \epsilon_b \leq 1$. These movements provide an adaptation towards the samples from the input space as can be seen in Figure 2.5.

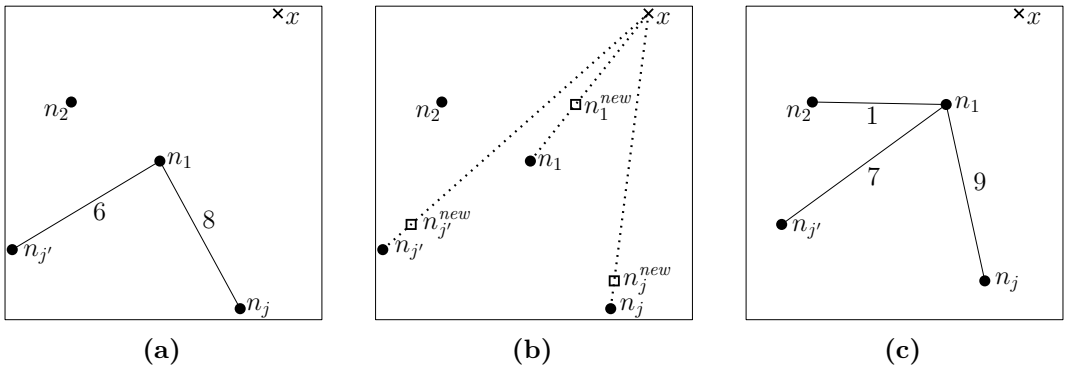


Figure 2.5: Example of the movement phase: In (a), the nearest and the second nearest neuron $n_1 = \text{nn}(x)$ and $n_2 = \text{nn}_2(x)$ to x are determined. The solid lines are the neighborhood connections with their respective age. After this, the nearest neuron n_1 and all of its neighbors n_j and $n_{j'}$ are moved towards the input vector x by using Equation (2.20) and Equation (2.21). The dotted lines in (b) are the direct connections between the neurons and the sample x. The new positions for the neurons are depicted by a square on this line. In (c), the final layout of this area is shown: The neurons n_1 and n_2 are now connected by a neighborhood connection and the ages of all connections are increased.

The above mentioned movement of neurons is only one part of the network's adaptation to the distribution of the input space. The second and even bigger portion is caused by inserting new neurons in areas where the distortion is high. These inserted neurons interpolate their positions from existing neurons and acquire a portion of the error value in the respective area.

Every λ_{insert}'th step, a new neuron is added in a region with maximal distortion, i.e. the new neuron n^+ is added halfway between the neuron n_q with largest accumulated error and the neuron n_f with n_f the largest accumulated error in n_q's neighborhood. The insertion procedure is as follows (depicted in Figure 2.6):

1. The neuron n_q with the largest accumulated error is selected. This neuron is the center of the region $\mathfrak{R}(\vec{w}_q)$ with maximal distortion (cf. Figure 2.6(a)).
2. All of n_q's topological neighbors are inspected and the neighbor n_f with the largest accumulated among them is selected (cf. Figure 2.6(b)).
3. A new neuron n^+ is created and positioned halfway between n_q and n_f (cf. Figure 2.6(c)):
$$\vec{w}_{n^+} = \frac{\vec{w}_q + \vec{w}_f}{2}\,. \tag{2.22}$$
4. The neighborhood connection between n_q and n_f is removed. Simultaneously, the neuron n^+ is connected to n_q and n_f, i.e. the connections $\{n^+, n_q\}$ and $\{n^+, n_f\}$ are established (cf. Figure 2.6(d)).

The newly inserted neuron n^+ will relieve n_q and n_f because a portion of the input will now be mapped to n^+. Thus the errors of n_q and n_f is decreased by a factor α. This increases the probability that the next insertion will take place in another region (Fritzke, 1996). The error of the new neuron is then initialized with n_q's new error value.

The motivation behind the insertion is the following: Every neuron accumulates the quantization error for all samples in its region. Thus, the sum of all error variables

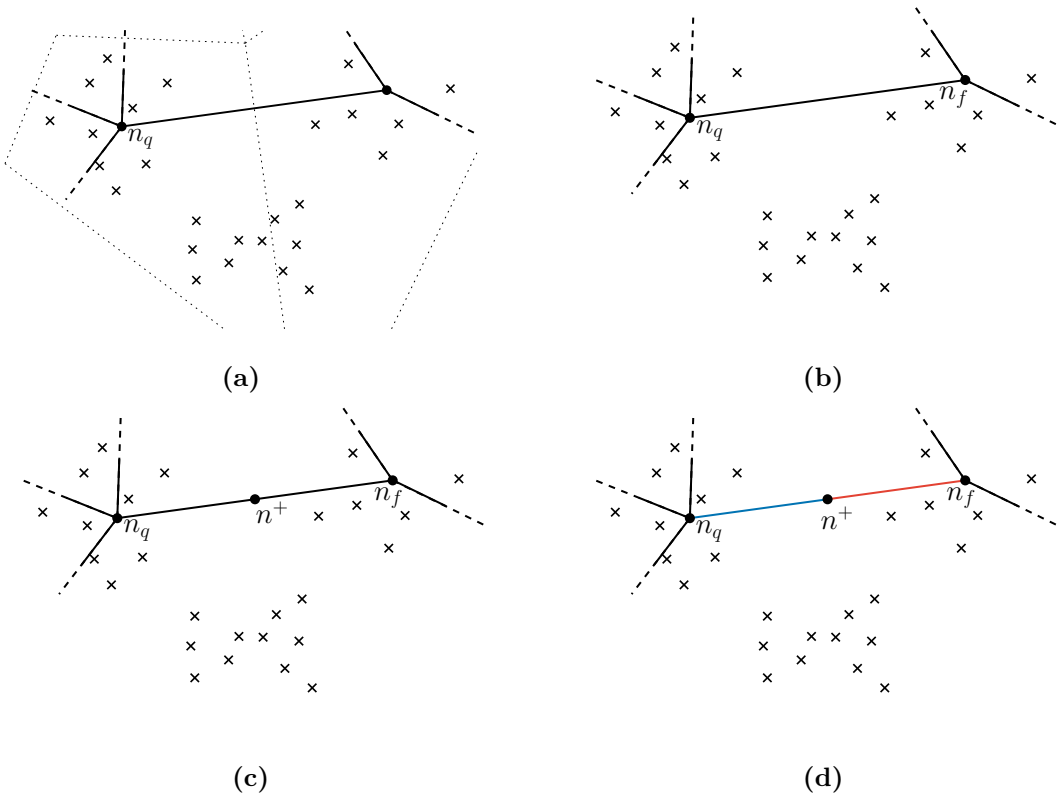


Figure 2.6: In this example, the crosses are samples and the solid and dashed lines are neighborhood connections. The dotted lines in (a) indicate the Voronoi borders for the regions of the left and the right neurons (•) and illustrate which samples are assigned to which neuron. For the insertion procedure, the left neuron n_q is selected because it has the highest error value in this example (a). The neuron n_f has the highest error value among n_q's neighbors (b). The high error values for both n_q and n_f are particularly caused by the group of samples in the bottom as the distances to their respective reference vectors are relatively large. In (c), the new neuron n^+ is placed halfway between n_q and n_f. Finally, the connection between n_q and n_f is removed and new connections between n^+ and n_q as well as between n^+ and n_f are established (d).

is the quantization error for all samples seen so far (except for the reduction by β). If a new neuron is inserted, the overall quantization error is prone to decrease. Without knowledge of the distribution of the input space, the most promising position is in close proximity of the neuron with the highest error (Fritzke, 1998).

2.3.6 Growing Neural Gas Algorithm

Algorithm 2 shows the complete growing neural gas approach. Note, that the pseudo code is organized such that it is explanatory and thus, the performance could clearly be improved by combining the loops.

Algorithm 2: GNG (Fritzke, 1994b)

create two neurons n_a and n_b with random positions $\vec{w}_a, \vec{w}_b \in \mathbb{R}^d$
$i \leftarrow 0$ `// initialize counter for training rounds`
repeat
  generate random sample x from the input space
  compute nearest $n_1 = \mathrm{nn}(x)$ and second nearest neuron $n_2 = \mathrm{nn}_2(x)$ to x
  `/* update error of nearest neuron` n_1 `*/`
  $\mathrm{error}(n_1) \leftarrow \mathrm{error}(n_1) + \mathrm{d}(x, n_1)^2$
  `/* move nearest neuron and its neighbors */`
  $\vec{w}_{n_1} \leftarrow \vec{w}_{n_1} + \epsilon_b(x - \vec{w}_{n_1})$
  foreach *neighbor* j *of* n_1 **do**
    $\vec{w}_j \leftarrow \vec{w}_j + \epsilon_n(x - \vec{w}_j)$
  `/* create or reset neighborhood connection */`
  if n_1 *and* n_2 *are connected* **then**
    set age of connection $\{n_1, n_2\}$ to zero
  else
    create new connection $\{n_1, n_2\}$
  `/* aging / removal of connections emanating from` n_1 `*/`
  foreach *neighbor* j *of* n_1 **do**
    increment age of connection $\{n_1, j\}$
    if *age of* $\{n_1, j\}$ *is larger than threshold* age_{max} **then**
      remove connection $\{n_1, j\}$
      remove possible isolated neurons if at least two neurons remain
  `/* insertion of new neuron */`
  if $i \mod \lambda_{insert} \equiv 0$ **then** `/* every` λ_{insert} `training round */`
    select neuron n_q with maximal error
    select neuron n_f that has the maximal error among n_q's neighbors
    insert new neuron n^+ halfway between n_q and n_f:

$$\vec{w}_{n^+} = \frac{\vec{w}_q + \vec{w}_f}{2}$$

    remove connection $\{n_q, n_f\}$ between n_q and n_f
    connect n^+ with n_q and n_f
    $\mathrm{error}(n_q) \leftarrow \alpha \cdot \mathrm{error}(n_q)$
    $\mathrm{error}(n_f) \leftarrow \alpha \cdot \mathrm{error}(n_f)$
    $\mathrm{error}(n^+) \leftarrow \mathrm{error}(n_q)$
  `/* error decay */`
  foreach *neuron* $n \in N_t$ **do**
    $\mathrm{error}(n) \leftarrow \beta \cdot \mathrm{error}(n)$
  $i \leftarrow i + 1$ `// increase counter for training rounds`
until *stopping criterion is fulfilled*

2.3.7 Concluding Remarks

Using the reference vectors of all neurons as codewords, GNG builds a Voronoi quantizer. The movement and the insertion of the neurons provides an adaptation of the nearest neuron and its neighbors to the underlying input space with a fixed intensity. The algorithm is executed until some convergence criterion is met (e.g. size restriction, a quality threshold, or maximal number of iterations).

All parameters used in this approach are constant over time and thus allow a continuous adaptation to the input space. In fact, there are five parameters necessary for the GNG approach:

Movement The parameters ϵ_b and ϵ_n control the movement of the nearest neuron and its neighbors. Thus, these values influence the strength of the adaptation to the currently presented sample.

Insertion Delay λ_{insert} is the periodicity in which new neurons are inserted into the network. If its value is to high, insertions for rare classes may be prohibited (Heinke and Hamker, 1998).

Maximal Age The maximal age of connections age_{max} determines how fast connections are removed which has an influence on the currency of the topology.

Error Decay β is used to highlight more recent errors and thus, its value determines how fast the impact of errors diminishes.

These parameters are well investigated and it turned out that GNG is relatively insensitive to their values (Heinke and Hamker, 1998).

The growing neural gas approach relies on two earlier presented works: Kohonen (1982) introduced the Self Organizing Feature Map that computes a usually two-dimensional representation of a given input space with an arbitrary dimensionality and thus usually incorporates a dimensionality reduction while Martinetz and Schulten (1991) created a vector quantization by using units with the same dimensionality as the input space.

The parameters in the neural gas approach (Martinetz and Schulten, 1991) are decreased over time, which on the one hand leads to a stabilization of the network but on the other hand prohibits a continual execution of the learning algorithm. In GNG the network is able to grow by adding new neurons or to shrink by removing isolated neurons, the size of the network does not have to be predefined as for example in the neural gas (Martinetz and Schulten, 1991) and neither size nor topology has to be fixed beforehand in contrast to self-organizing maps (Kohonen, 1982). Thus, no knowledge about the input space despite the dimensionality is needed. During learning, the network computed by GNG is always a good approximation of the input space while Martinetz and Schulten's neural gas has intermediate states in which the approximation is relatively poor due to unevenly distributed centers and a high number of redundant connections (Fritzke, 1997).

3
Related Work

This chapter gives a short introduction to approaches that are to some extend similar to the work presented in this thesis. Section 3.1 reviews works that deal with computational methods for shepherding tasks and in Section 3.2 we compare the approaches. Section 3.3 surveys approaches that obtained generalization by aggregating states or by approximating the value functions of the reinforcement learning agent. We conclude this chapter in Section 3.4 where we categorize the mentioned approaches and contrast them to the algorithms developed in this thesis.

3.1 Shepherding Approaches

We here review related work on algorithmic or machine learning solutions for shepherding or shepherding-related tasks. Unfortunately, only a small amount of such literature exists. For overviews of the biological background of shepherding or possible applications of such approaches we refer to Section 4.2 or Section 4.1, respectively.

The first investigation of learning shepherd behaviors was performed by Schultz et al. (1996). There, the authors use a genetic algorithm to learn rules in a simulator that were later implemented in a real robot. This dog robot had to drive sheep-robots with a fixed behavior into a predefined area. The sheep initially wander around randomly until the dog comes into sight at which point the sheep switched to a fleeing behavior and tried to get away from the approaching dog. The dog-robot had several sensors that captured its orientation and its distance to the target as well as to the sheep. Additionally, it had information about the orientation of the sheep. These values were matched against the rules which commanded the dog to move or adjust its orientation by turning.

Vaughan et al. (1998) presented a potential field based approach to let a robot gather a group of ducks (whose herding behavior can be considered similar to that of sheep) in a target area. This approach was evaluated in simulation and transferred to a mobile robot. The shepherding robot had access to information of an overhead camera that overlooked the complete scene and thus, the robot had (nearly) perfect

knowledge of its position, the positions of the ducks as well as the position of the target. The potential field algorithm used three forces to compute the movement of the shepherd:

1. An attraction of the shepherd to the center (i.e. their average position) of the flock with a magnitude proportional to the distance in between.
2. A repulsion of the shepherd from the center of the flock to prevent collisions. The magnitude of this force was proportional to the inverse square of the distance.
3. A repulsion of the shepherd from the target with constant magnitude.

These forces lead to a minimum behind the flock in relation to the target and thus, the shepherd tends to move to this point. This movement drives the flock away from the shepherd and towards the target. Additionally, it is assured that the shepherd does not crash into walls.

Sigaud and Gérard (2000) used Learning Classifier Systems (LCS) in the same task as Vaughan et al. (1998) with three shepherds and six ducks. The authors manually designed features to describe situations of the task: These features captured e.g. the positioning of the shepherds itself (e.g. "isOnWay"), information about other shepherds (e.g. "nobodyBehindFlock"), as well as their orientation to the ducks (e.g. "isBehindFlock") and the target (e.g. "isAtTarget"). Additionally, high-level actions like "go behind the flock" or "drive closest duck to flock" were implemented. LCS used this expert knowledge and learned a mapping between features and actions to steer the flock to the target.

Potter et al. (2001) introduced a scenario with up to three shepherds, that had to drive one sheep into a target area. The sheep followed a fixed strategy that kept it away from other agents and additionally let it avoid the target and instead trying to flee to the open side of the environment. Optionally, a fox was introduced that also followed a fixed behavior and tried to kill the sheep. The behavior of the shepherds were implemented with neural networks whose weights were evolved with a genetic algorithm. The authors investigated homogeneous (i.e. all shepherds used the same network) and heterogeneous (i.e. each agent learned a different network) behaviors of the shepherds and observed that in the heterogeneous case one shepherd learned to guard the sheep by blocking the fox. Features for the dog(s) were derived in relation to the sheep; they included the sheep's orientation and distance to the target as well as to the dog(s). From these values, the neural network determined the translation and the bearing of the shepherd.

Most recently, Gomes et al. (2015) applied cooperative coevolution algorithms (CCEAs) in the same task as Potter et al. (2001). CCEAs allow the evolution of a heterogeneous set of agent behaviors in which each agent develops a specialized behavior. To improve the scalability of CCEAs, Gomes et al. propose a hybrid approach Hyb-CCEA in which agents that behave similar share a controller. They argue that their approach is able to compute solutions with similar performances more efficiently than standard CCEAs and that this gain is higher with growing numbers of agents.

Lien et al. (2004) presented several strategies to approach and steer a flock of sheep with one shepherding agent. They argued that a side-to-side motion of the shepherd—i.e. moving from left to right and back while approaching the flock—is

most powerful. Additionally, the agent stays outside the bounding box of the flock to avoid too much disturbance. The authors mention that a similar strategy is actually used by Border Collies. One benefit of this strategy is that in addition to steering the flock the shepherd compresses the flock structure. Lien et al. (2009) extended the approach to include human interaction with the shepherd.

Strömbom et al. (2014) implemented a heuristic shepherding algorithm to work on agents that are modeled as particle model. The sheep agents try to remain their flocking structure while avoiding to be too close to the shepherd. The shepherd's task is to collect all sheep and drive the herd to the lower left corner of the environment. To do so, the shepherd-agent uses three rules:

1. If the shepherding agent is closer than a given "interaction distance" to any sheep-agent, it remains on its position to avoid disturbance.
2. If all sheep agents are within a given radius around their center of mass—i.e. the flock structure is sufficiently dense—the shepherding agent drives the sheep towards the target by approaching them from a position behind the flock in relation to the target.
3. If at least one sheep-agent is farther apart, the shepherding agent starts collecting these agents to improve the flocking structure.

These rules lead to a behavior that is similar to the hand-coded strategies reported by Lien et al. (2004): The shepherd uses a side-to-side movement to collect separated sheep and otherwise drives the sheep towards the target. Strömbom et al. compared results of their approach with empirical data derived from an actual sheep dog and note that both behaviors are similar.

A different approach to solve the shepherding task was followed by Razali et al. (2012) who presented an algorithm that used a memory-based artificial immune system (an approach that share some commodities with neural networks) to optimize the control for the shepherding agents. An also slightly different task was investigated by Çelikkanat and Sahin (2010): There, some members of the flock that should be controlled were informed about the intended direction. These individuals could not be identified as leaders but helped "anonymously" steer the flock into the right direction.

Thakkar and Wesley (2005) described an approach where a robot supported a human in a shepherding task. Although only restricted to herding robots, this work is similar to (Lien and Pratt, 2009) as both approaches incorporate human interaction into the process. Thakkar and Wesley used gestures to communicate with the shepherding robot while Lien et al. used laser pointers.

3.2 Discussion of Shepherding Tasks and Approaches

Here, we relate our work to the approaches described before.

The shepherding task considered in Chapter 4 does not include informed sheep as in the approach presented by Çelikkanat and Sahin (2010) and does not rely on human assistance as in the works of Lien et al. (2009) or Thakkar and Wesley (2005).

Schultz et al. (1996), Sigaud and Gérard (2000), and Potter et al. (2001) also used learning approaches to solve several instantiations of a shepherding task but none of these works compared their performance with an optimal solution. These works

used various learning approaches but neglect the usage of reinforcement learning—a learning scheme that is in our opinion well suited for these kinds of tasks. Another problem may occur as Sigaud and Gérard present hand-coded high-level actions that highly depend on the expertise of the designer.

Vaughan et al. (1998) and Razali et al. (2012) applied different optimization approaches to solve the shepherding task. Especially Vaughan et al.'s idea of computing a solution with a vector field is very interesting although its performance depends on an observing camera. Furthermore, neither approach offers a means of rating how good the computed solution actually is.

The hand-coded strategies in the works of Lien et al. (2004) and Strömbom et al. (2014) appear to be well suitable and the authors argue that real shepherding dogs use similar strategies. Nevertheless, these works omit the comparison with an optimal solution, too.

In this thesis, we present an algorithm that computes a solution that is within proven bounds around the optimal solution and we can thus not only make claims about the quality of the algorithm's solution but also compare the learned strategies to this baseline. This is a valuable contribution to the interesting field of learning and computing shepherding strategies.

Furthermore, the shepherding task introduced in Chapter 4 can be adjusted to match the formulations of Schulz et al. (1996), Vaughan et al. (1998), Lien et al. (2004) and with the slight extension of including a predator even to that of Potter et al. (2001).

3.3 Approximations for Reinforcement Learning

This section gives an overview of other approaches that use various machine learning methods to obtain approximation in reinforcement learning. We describe their core ideas and point out major differences to the approaches proposed in this thesis. Table 3.1 presents a short overview of the approximation approaches described hereafter. We here mostly focus on approaches, that aggregate states adaptively or that directly approximate the value function of the reinforcement learning task. We refer the reader e.g. to (van Otterlo, 2009) or (Buşoniu et al., 2011a) for even broader overviews of approximations in reinforcement learning.

3.3.1 State Aggregation Approaches

Many state aggregating methods are built upon *vector quantization* (e.g. clustering or more generally groups of states that are somehow "similar"). Bertsekas and Castañon (1989) introduced an adaptive approach that groups states with similar magnitude of their Bellman residuals. Since these residuals are prone to change over the time of learning, the state groups may be readjusted repeatedly.

Smith (2002) used two self-organizing maps to deal with continuous state and action spaces. This setup results in a discretization of both spaces so that standard Q-Learning could be applied: The neural units approximating the continuous state vector are interpreted as discrete states and the units representing the action space are treated as discrete actions. Thus, a tabular implementation of Q-Learning can be applied. One general drawback of using self-organizing maps may be that the

dimensionality and the number of neurons is predefined and does not change during learning. Without knowledge of the environment, these values have to be determined experimentally in order to cover the state space sufficiently.

The approach of Gomes et al. (Gomes et al., 2015) is similar in spirit: They try to share agent controllers among sub-teams of homogenous agents inside a group of heterogeneous agents. Thus, they perform a structural approximation on the system level by grouping agents that behave similar in one group while the approaches presented in this thesis perform approximations on the level of agents by approximating knowledge inside one agent.

Lee and Lau (2004) used an adaptive vector quantizer to partition the state space while the agent is learning. The resulting partitioning is based on proximity in the state space and it respects similarity of Q-vectors computed by Q-Learning. The approximation is refined if the reward accumulated in one region is exceeding some threshold and a predefined minimal distance to all neighboring centers is kept. If after an episode ends two cells are nearest neighbors and both Q-vectors are more similar than a given threshold, they are merged. In this approach, the centers of the created regions are not able to move and domain knowledge especially about the values and the shape of the reward function is required to determine useful values for the thresholds used. Similar to the *GNG-Q* approach described in Chapter 7, all states in one region are treated identically. *GNG-Q* and (to some degree) *I-GNG-Q* (introduced in Chapter 8) also respect similarity in the state and action space; but both approaches do this by counting changes in the local policy instead of predefined thresholds. Additionally, *GNG-Q* and *I-GNG-Q* adapt their approximation to the state space while the centers in the work of Lee and Lau (2004) are fixed. One problem with Lee et al.'s approach may be the unpredictable effect of the merging operation on all surrounding regions.

Fernández and Borrajo (2000) presented the VQQL model that consists of the generalized Lloyd algorithm for vector quantization and Q-Learning. It uses vector quantization to obtain a set of codebook vectors that represent the state space. In a subsequent step, Q-Learning is used to learn a policy based on this reduced representation. The key difference to the *GNG-Q* and *I-GNG-Q* approaches is that Fernández and Borrajo construct the state-space representation independently from learning.

A different field of aggregation methods use space partitioning data structures to split the state space into portions of states that are treated equally. Chapman and Kaelbling (1991) introduced the *G-algorithm* that groups states with the same reward and identical Q-values with the use of a decision tree. It starts with one abstract state for the complete state space and recursively splits all abstract states wherever incompatibilities arise. A similar approach was proposed by McCallum (1995): The *Utile Suffix Memory* also uses a decision tree and aims at finding groups of states with identical optimal actions. It does so by performing a Kolmogorov-Smirnov test to compare the distributions of future rewards for the same action performed in different path through the tree. These approaches were extended by Uther and Veloso (1998) to also work in reinforcement learning tasks with continuous state spaces. Pyeatt and Howe (1998) also introduced a means of dealing with continuous state spaces to the G-algorithm and additionally investigated other criteria of when and

where to split.

Often, *tile coding* approaches are used to deal with large or continuous state spaces. Figuratively, overlapping tilings are laid over the state space and each state activates all tiles, it appears in. These activations then form a new description of the state. Sherstov and Stone (2005) investigated the influence of parameter choices for tile coding and pointed out the need of adjusting these parameter values over time. Following these results, several adaptive tile coding approaches were proposed.

Adaptive Tile Coding (ATC) (Whiteson et al., 2007) is an approach that learns a policy in parallel with an appropriate state-space abstraction. Starting with a very coarse approximation consisting of just one tile, it is refined based on information (i.e. the Bellman error) from learning (model based dynamic programming) until the task can be learned adequately on this piecewise constant approximation. The refinement operation splits one tile evenly in half, which will often lead to problems as it may happen that many splits are needed until incompatible states are separated. Obviously, this might result in a too fine approximation in some parts of the state space. Whiteson et al. (2007) present two criteria for selecting the tile that should be split: One refines the approximation to obtain a finer policy; the other refines areas where the value function changes rapidly.

Lin and Wright (2010) present evolutionary tile coding (EvoTC), a model free approach that uses a genetic algorithm to decide where and when tiles should be split. Contrary to the ATC approach of Whiteson et al. (2007), the approach of Lin and Wright (2010) is able to split tiles unevenly and thus avoids some unnecessary splits. However, the splits remain parallel. The genetic algorithm optimizes the performance of the RL algorithm (e.g. SARSA) and uses this performance to evaluate the chromosomes' fitness. New tilings are created by the mutation operation that affect the structure and the resolution of tilings. A drawback of this algorithm may be that the RL algorithm has to be executed for each chromosome and that the approximation is not computed in parallel with the learning of the RL task as it is done in *GNG-Q* and *I-GNG-Q*.

A different direction is taken by model minimization approaches that usually work on the MDP model itself and are thus generally applied prior to learning the reinforcement learning task. The reduced model can then be solved with standard reinforcement learning algorithms. Dean and Givan (1997) used stochastic bi-simulation to create the coarsest refinement of the state space into stable blocks of states. Givan et al. (2003) extended the work of Dean and Givan to handle stochastic transition and reward functions. A similar approach was presented by Ranvindran and Barto (2002) who proposed a model minimization method that exploits symmetry and redundancy of the MDP to obtain a more compact representation. These approaches preserve the actual state transitions and the corresponding rewards.

3.3.2 Function Approximation Approaches

While the approaches discussed until now dealt with finding a suitable aggregation of somehow compatible states, we now focus on methods that strive to approximate the value function(s) of the reinforcement learning agent directly.

Ormoneit and Sen (2002) present a kernel-based approximation for reinforcement learning with value iteration in continuous state spaces. The model based approach

interpolates the action-value function by a sum of weighted kernels. The probability that it converges to the true action-value function increases if the kernel bandwidths are decreased appropriately during learning and the number of samples increases. Drawbacks of this approach include the fact that the learning is offline and a set of sample trajectories in the MDP is needed. Furthermore, no exploration is considered.

Šter and Dobnikar (2003) use a neural network constructed of radial basis functions to approximate the action-value function of Q-Learning. The adaptive radial basis decomposition (ARBD) approach starts with medium sized (regarding the width) basis functions and finds areas, where they have to be replaced by functions with smaller widths. Two processes control this refinement: The first process adds new basis functions when the current state is not sufficiently covered, the temporal difference error is high and a given minimal resolution is respected. The second process employs a vector quantization algorithm that approximates the temporal difference error distribution. It uses conditions on this error to decide when to refine the approximation by placing new basis functions on the centers of the vector quantization. Similar to the *GNG-Q* and *I-GNG-Q* approach, vector quantization is employed to compute centers for the approximation. The decision to refine the approximation in (Šter and Dobnikar, 2003) is based on a predefined threshold and the approximation has to adhere to a given minimal resolution. While Šter and Dobnikar (2003) base the quantization on the temporal difference error, the quantization in *GNG-Q* and *I-GNG-Q* is based on the similarity in the state-action space.

Menache et al. (2005) present two model free methods to adjust the parameters of predefined basis functions during learning with a fixed control policy: The first uses gradient descent optimization while the second uses cross-entropy optimization. Both approaches optimize the Bellman error and approximate the value function. The approaches consists of two interleaved phases: One phase updates the behavior of the reinforcement learning agent using LSTD(λ) while all RBF-network parameters were fixed while the second phase updates the parameters of all RBFs. A difference to *I-GNG-Q* is that in the adjustment step of (Menache et al., 2005) all basis functions are changed while *I-GNG-Q* only performs local adjustments.

Mahadevan (2005) introduced so-called proto-value functions, that are learned by analyzing the structure of the state space. This approach models the MDP as a graph and uses least-squares policy iteration to learn the agent's policy. Here, the learning is done by analyzing the topology of the state space instead of learning from the rewards the agent receives.

In the work of Keller et al. (2006), a model based approach to construct a linear approximation of the value function with basis functions is presented. This approach builds on the work of Bertsekas and Castañon (1989) and uses neighborhood component analysis together with dynamic programming to combine states which have similar Bellman errors. The goal is to reduce the dimensionality of the state space by placing basis function in those clusters of "similar" states.

In (da Motta Salles Barreto and Anderson, 2008), an approach to automatically construct a RBF network similar to the work of Ormoneit and Sen (2002) is used to approximate RL value functions. Its restricted gradient descent (RGD) method is designed to reduce divergence of the approximation. The approach conservatively

adjusts the centers of the RBFs as well as their widths during learning with SARSA. The basic idea is similar to *I-GNG-Q* but the criterion for the refinement is different. Da Motta Salles Barreto and Anderson (2008) add a new basis function when the activation of the most activated RBF is below a predefined threshold while *I-GNG-Q* refines the region where the policy changed most frequently.

Buşoniu et al. (2011b) introduce a direct search approach to approximate a policy for continuous state RL tasks. The presented algorithm computes a policy that maximizes the empirical reward from on a representative set of start states using the cross-entropy method. Each basis function of a predefined set is assigned to one discrete action and these assignments as well as their shapes and locations are adjusted during learning. Unfortunately, this approach requires a predefined set of basis functions while *I-GNG-Q* starts with a very coarse approximation that is refined whenever necessary.

Konidaris et al. (2011) approximate the value function of continuous environments with a method that employs Fourier series and SARSA. They compare their approach empirically to various other basis function approaches and conclude that its performance is similarly good. However, this approach seems to be rather runtime consuming.

A slightly different direction was followed by Engel et al. (2003) who approximated the value function with a Bayesian probabilistic model defined over a Gaussian prior. This approach was extended by Engel et al. (2005) to work with stochastic state transitions.

Munos and Moore (2002) present a model-based approach for continuous state RL tasks with continuous time. They approximate the state space with different kinds of binary trees that are iteratively refined by splitting cells. In (Munos and Moore, 2002), different splitting criteria are investigated, ranging from local ones that are e.g. based on the policy to global ones that respect the impact of one particular split on surrounding cells. The MDP for each level of approximation is solved with dynamic programming, which is computationally expensive. Furthermore, this approach works only offline and needs the model of the RL task that should be solved. Munos and Moore's method may suffer from a too fine resolution as (Whiteson et al., 2007) since both methods refine cells in a similar manner. The *I-GNG-Q* and the *GNG-Q* approaches described in this thesis work online, i.e. at any given time during learning, the agent can exploit knowledge gained so far.

Ratitch and Precup's (2004) model free approach places *Sparse Distributed Memory* units as centers for the approximation. The value of a state is then interpolated between the values of activated units. New units are added during learning with SARSA if for the current state the number of activated nearby units is below some threshold. In contrast to the approach presented here, Ratitch and Precup need predefined thresholds and the units are not able to adapt to the distribution of the state space by moving. If an adjustment of the approximation is needed, new units are added and others are possibly deleted. Ratitch and Precup's method is partly similar to the approaches of e.g. Fernández and Borrajo (2000), Lee and Lau (2004) or *GNG-Q* but in (Ratitch and Precup, 2004) and *I-GNG-Q* the values are interpolated between different prototype states.

Ernst et al. (2005) follow a similar idea as Ormoneit and Sen (2002) and intro-

duced fitted Q-iteration, an approach that approximates the action-value function. It uses a set of samples that reflect the agent's experience within the environment. By using this set of samples, the RL task is transformed into a number of supervised learning tasks that are approached with several tree-based methods. The storage of the samples for this approach may become an issue as the agent constantly interacts with the environment.

Whiteson and Stone (2006) combine neuroevolutionary optimization with Q-Learning to build an evolutionary function approximation: A population of candidate neural networks is trained based on information from Q-Learning and crossover and mutation operators are used to create new generations of hopefully fitter networks. The value function is then modeled by the neural network. Although this approach finds good approximations, it requires a rather long time for training.

Bradtke and Barto (1996) proposed to apply least-squares algorithms for temporal difference approaches (LSTD) in reinforcement learning. These methods are able to efficiently compute the value function for a given (fixed) policy. Lagoudakis and Parr (2003) enhanced LSTD to allow the computation of a good policy with the *least-squares policy-iteration* (LSPI) algorithm. Later, these approaches were augmented with regularization to prevent over-fitting (Kolter and Ng, 2009).

3.4 Discussion of Approximation Approaches

The approximations from before can be classified and distinguished along several dimensions. For this analysis we incorporate the following:

Different Ways of Refining the Approximation focuses on the method that is used to alter the resolution of the approximation.

Layout and Shape of Abstract States compares whether the layouts of the abstract state spaces are predetermined. Clearly, this dimension is only relevant for state-space abstraction approaches.

Necessary Knowledge of the MDP Model points out if the approach makes use of the reinforcement learning task's MDP or if it is capable of learning from the agent's interaction with its environment.

Required Batches of Experience means that representative samples of an agent interacting with the environment have to be recorded. This dimension states whether the approach needs these information or if the agent learns "live" from its own experiences.

Online vs. Offline Learning compares whether the agent can use its knowledge at any time during training, with some delay, or only after training.

Needed Domain Knowledge measures how much domain knowledge of the designer has to be incorporated in order to derive useful approximations.

Runtime compares the approaches regarding the overhead that is caused by the approximation method.

3.4.1 Different Ways of Refining the Approximation

There are several ways of adapting the resolution of the approximation: Usually methods start with a very coarse resolution (e.g. only consisting of one abstract

Table 3.1: Overview of related approaches

	Approach	Type of approximation	Online	Used RL method	Model free
State Aggregation	IA-DP (Bertsekas and Castañon, 1989)	piecewise constant	no	DP	no
	Neighbourhood Q-Learning(Smith, 2002)	piecewise constant	✓	Q-Learning	✓
	VQQL (Fernández and Borrajo, 2000)	piecewise constant	no	Q-Learning	✓
	G-Algorithm (Chapman and Kaelbling, 1991)	piecewise constant	no	Q-Learning	✓
	Continuous G-Algorithm (Pyeatt and Howe, 1998)	piecewise constant	✓	Q-Learning	✓
	U-Tree (McCallum, 1995)	piecewise constant	✓	DP	learns approximation of model
	Continuous U-Tree (Uther and Veloso, 1998)	piecewise constant	✓	DP	learns approximation of model
	TD-AVQ (Lee and Lau, 2004)	piecewise constant	✓	Q-Learning	✓
Tile Coding	ATC (Whiteson et al., 2007)	piecewise constant	✓	DP	no
	EvoTC (Lin and Wright, 2010)	piecewise constant	no	SARSA	✓
Function Approximation	Variable Resolution Discretization (Munos and Moore, 2002)	piecewise linear	no	DP	no
	SDM RL (Ratitch and Precup, 2004)	SDM	✓	SARSA	✓
	NEAT+Q (Whiteson and Stone, 2006)	neural network	✓	Q-Learning	✓
	kernel based RL (Ormoneit and Sen, 2002)	Kernel based	no	value iteration	no
	ARBD (Šter and Dobnikar, 2003)	RBF	✓	Q-Learning	✓
	Menache et al. (Menache et al., 2005)	RBF	no	LSTD(λ)	✓
	RGD (da Motta Salles Barreto and Anderson, 2008)	RBF	✓	SARSA	✓
	CE policy search (Buşoniu et al., 2011b)	RBF	✓	policy search	✓
	Proto-Value (Mahadevan, 2005)	Fourier	no	LSTDQ	✓
	Fourier Basis (Konidaris et al., 2011)	Fourier	✓	SARSA	✓
	GPTD (Engel et al., 2003, 2005)	Gaussian	✓	SARSA	✓
	Fitted Q Iteration (Ernst et al., 2005)	various	no	value iteration	✓
	LARS-TD (Kolter and Ng, 2009)	RBF	no	LSTD	✓

state) and based on different criteria this resolution is refined.

The approaches from (Munos and Moore, 2002) and (Whiteson et al., 2007) both make use of the policy to determine areas that have to be refined. Munos and Moore (2002) search for places where the policy changes by comparing the policy derived from the original MDP with the approximated policy and Whiteson et al. (2007) locate changes in the policy by computing the influence on the value function for potential splits. The other tree-based approaches (Chapman and Kaelbling, 1991; McCallum, 1995; Uther and Veloso, 1998; Pyeatt and Howe, 1998) use statistical measures mostly on differences in the distributions of expected (future) rewards.

Lee and Lau (2004) refine the approximation if the reward accumulated in one region is larger than a predefined threshold and a given minimal resolution is kept. (Ratitch and Precup, 2004), (Šter and Dobnikar, 2003), and (da Motta Salles Barreto and Anderson, 2008) use similar approaches to determine the time and place of a refinement: Ratitch and Precup add a new element whenever less than a given number of elements is activated while Barreto and Anderson refine the approximation if the maximal activation of the network is below some threshold. Šter and Dobnikar add a new unit each time the current temporal difference error is larger than a predefined threshold and the activation of the network is below a different threshold.

Other approaches such as e.g. (Menache et al., 2005), (Whiteson and Stone, 2006), (Lin and Wright, 2010) implicitly refine the approximation without directly using feedback from the learning agent. For example, the mutation operator of the evolutionary algorithm that computes the weights for the neural network in Whiteson and Stone's approach is responsible for adding new neurons while Lin and Wright's approach directly implements the refinement of the tilings in the mutation operator.

In contrast, *GNG-Q* and *I-GNG-Q* count how often the action with the maximal $\widehat{Q}$-value changes in order to determine when and where to refine the aggregation or the approximation. This information can be achieved during learning by monitoring the maximal policy of the agent while emphasizing more recent changes. To us adapting the approximation to the input space by movements is more promising than only placing centers in sparse regions and keeping their positions fixed as in the work of Lee and Lau (2004).

3.4.2 Layout and Shape of Abstract States

The approaches (Whiteson et al., 2007) and (Lin and Wright, 2010) define abstract states by borders that are always parallel. Nevertheless, both approaches differ in the way of where tiles are split: Whiteson et al. always split the tiles in half while Lin and Wright's approach looks for the best place to split.

Tree-based approaches are usually also bound to orthogonal splits. Additionally, approaches that split (abstract) states according to the relevance of one specific dimension of the state space at a time as e.g. (Chapman and Kaelbling, 1991; McCallum, 1995; Uther and Veloso, 1998; Pyeatt and Howe, 1998) are not able to recognize the significance of feature combinations. Munos and Moore (2002) also use tree structures to store the knowledge of the learning agent but use information of the value function or the agent's policy to decide on the need of refining the approximation.

Approaches based on vector quantization on the other hand do not predetermine the shape of abstract states: (Lee and Lau, 2004), (Fernández and Borrajo, 2008), (Smith, 2002), and *GNG-Q* allow arbitrary layouts of the abstract states.

Clearly, hard-coded shapes of the abstract states may become problematic if the predefined shape does not directly match the shape of irregularities in the actual model, or when several splits are needed just to cover one particular feature of the environment.

3.4.3 Necessary Knowledge of the MDP Model

Some approaches need access to the model of the reinforcement learning task. In general, this includes the complete MDP with the transition function and the reward function.

The model-minimization approaches (Dean and Givan, 1997; Givan et al., 2003; Ravindran and Barto, 2002) clearly need the model that should be minimized which restricts the applicability to tasks where this information is available. Undoubtedly, this may allow to find even compacter approximations than approaches that do not have access to the reinforcement learning task's MDP.

McCallum's (1995) and Uther and Veloso's (1998) approaches perform dynamic programming on approximations of the actual MDP model that is derived from interaction with the environment. Other approaches that use dynamic programming and are thus also dependent on access to the MDP include (Ormoneit and Sen, 2002), (Munos and Moore, 2002), (Whiteson et al., 2007), and (Lin and Wright, 2010).

All other approaches mentioned in this section and especially the approaches presented in this thesis (*GNG-Q* and *I-GNG-Q*) do not need the model of the reinforcement learning task at hand.

Although such approaches usually perform well, the need for the complete model of a reinforcement learning task is a rather tough assumption. Often such a model is not available and agents that are able to work successfully even when faced with this absence appear to be very powerful.

3.4.4 Required Batches of Experience

Another, somewhat less demanding requirement is the access to batches of experience: This means that the learning agent gets data of possible interactions with the environment including the states, the performed actions, the received rewards, and the resulting state.

The difference to the approaches that learn an approximation of the model as e.g. (McCallum, 1995; Uther and Veloso, 1998) is that those approaches collect data *during* the agent's interaction with the environment while the following approaches need the data *prior* to learning. Similarly, Engel et al. (2003; 2005) rely on batches of transitions that have been experienced by the agent before. The approach in (Mahadevan, 2005) uses sampled trajectories to build an approximation of the state space's topology while Ernst et al. (2005) use a regression algorithm to induce the action-value function from a set of training samples. Other sample-based approaches include e.g. (Lagoudakis and Parr, 2003) or (Kolter and Ng, 2009).

A problem with learning from samples may be the increasing need for storage to collect enough sample trajectories to allow successful learning. The approaches presented in this thesis do not need to collect samples of the interaction with the environment.

3.4.5 Online vs. Offline Learning

Online learning means that the learning of the task is done while the agent interacts with the enclosing environment. Particularly, the agent can use the current state of its knowledge at any time even though this knowledge may be not be very expressive in the early stages of its interaction with the environment. Offline learning occurs if the agent learns in advance.

The batch-based approaches mentioned before (e.g. (Lagoudakis and Parr, 2003) or (Munos and Moore, 2002)) as well as the evolutionary tile-coding approach (Lin and Wright, 2010) work offline. Additional approaches that are intended to work offline include (Fernández and Borrajo, 2000), (Menache et al., 2005) or (Mahadevan, 2005).

Advantages of such offline approaches are the separation of the exploration from the actual learning of the behavior (which improves the dealing with the exploration-exploitation tradeoff) as well as a sometimes more efficient use of the experience. The downside is that these approaches share the drawbacks in the previous section (Required Batches of Experience).

Conversely, the other approaches reviewed in this chapter as well as *GNG-Q* and *I-GNG-Q* refine the approximation in parallel with the RL algorithm and thus, newly gained knowledge can directly be exploited in the learning process.

3.4.6 Needed Domain Knowledge

Some approaches need domain knowledge to function satisfying: For example, the SOM-based approach by Smith (2002) needs a predefined dimension of the approximating network. Other approaches employ thresholds to refine the approximation (e.g. (Ratitch and Precup, 2004), (Lee and Lau, 2004), or (da Motta Salles Barreto and Anderson, 2008)). Lagoudakis and Parr (2003) used a predefined parametric approximation whose parameters were estimated using a least squares method while Ormoneit and Sen (2002) used a non-parametric kernel-based approximation. In (Konidaris et al., 2011), the number and the order of the Fourier bases had to be determined.

The tree-based approaches (McCallum, 1995; Uther and Veloso, 1998; Chapman and Kaelbling, 1991; Pyeatt and Howe, 1998) that refine the state-space abstraction based on thresholds for statistic tests rely on thresholds whose values influence the layout of the approximation and thus the performance of the learning agent.

Since the performance of these approaches usually depends on appropriate values for these thresholds, often deep domain knowledge and/or repeated executions of the approaches are necessary.

Our *GNG-Q* approach only requires parameters that are well explored and GNG turns out to be quite insensitive to its parameter values (Heinke and Hamker, 1998) which essentially abandons the requirement of knowledge on the RL task. The

I-GNG-Q approach additionally needs two parameters that affect the amount of smoothing of the value function.

3.4.7 Runtime

The approaches (Munos and Moore, 2002; Lin and Wright, 2010) are computationally expensive as both need to solve the RL task for several grades of approximation. Especially, Lin and Wright's (2010) approach uses the solution quality of the reinforcement learning task as fitness for the evolutionary algorithm which implies solving the task for each individual. Ernst et al. (2005) employ numerous runs for the regression algorithm that extrapolates the knowledge beyond the training set. Similarly, Mahadevan's (2005) approach contains one execution of LSTDQ for each iteration of the RBF parameter improvement.

The G-Algorithm (Chapman and Kaelbling, 1991) removes knowledge every time a state is split. This may result in several redundant computations for some parts of the state space.

Although computing smoother approximations, the evaluation of exponential or trigonometric functions (e.g. exp, cos, sin) is in general computationally more expensive than piecewise constant approaches. This applies to all approaches that rely on RBF or Fourier bases: (Šter and Dobnikar, 2003), (Menache et al., 2005), (Mahadevan, 2005), (da Motta Salles Barreto and Anderson, 2008), (Buşoniu et al., 2011b), (Konidaris et al., 2011), (Kolter and Ng, 2009).

The online state-aggregating approaches as e.g. (Lee and Lau, 2004) or the *GNG-Q* approach developed in Chapter 7 do not need the computation of such functions and they interleave the learning of the task with the adjustment of its representation. Similarly, the *I-GNG-Q* approach in Chapter 8 learns online and uses a much more performant base than the RBF approaches.

4 The Shepherding Task

This chapter introduces the SHEPHERDING task in a multiagent system with two different types of agents—*sheep* and *dogs*. The dogs' goal is to drive the sheep to a designated target area (cf. Figure 4.1). Practical applications of such shepherding behaviors include—in addition to using robots to steer flocks of real animals as e.g. investigated by Vaughan et al. (1998) or Evered et al. (2014)—evacuation tasks (Dyer et al. (2008) found that large crowds of people show a similar behavior as herds of animals) or swimming robots that encircle oil spills to improve the manual strategies mentioned in (Fingas, 2002).

Although other works have already investigated similar shepherding tasks (see e.g. (Vaughan et al., 1998) or (Lien et al., 2004)), existing work does not present a theoretical analysis of the task as we present in Section 4.6.

The main results given in this chapter are the following:

- We formally define the SHEPHERDING task in Section 4.3.
- We describe a (multi-)agent system for the SHEPHERDING task and relate it to existing taxonomies in Section 4.4.
- We carefully analyze the state-space complexity of the SHEPHERDING task in Section 4.6 and show that the size of the state space grows exponentially in the number of dogs and sheep.

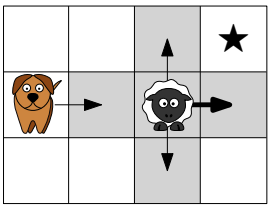


Figure 4.1: Example of a SHEPHERDING instance with one dog, one sheep with a viewing range of one, and one target (★). The dog enters the viewing range of the sheep that now has three possible actions to maximize the distance to the dog. According to the model presented in Section 4.5, it performs the action "forward" (relative to the dog).

In addition to the content mentioned before, Section 4.1 presents a motivation and applications for our work while Section 4.7 concludes this chapter. Some of the results presented here are partly based on previously published material (Baumann and Kleine Büning, 2013).

4.1 Motivation

In recent years, the number of robots sold has steadily risen and an increasing variety of services have been provided by robots (IFR Statistical Department, 2013). Application areas include assembly (e.g. in the automotive industry), explosive ordnance disposal (Kron et al., 2004), surveillance tasks (Parker and Touzet, 2000), or service robots that e.g. autonomously vacuum the floor. For further applications of (multi-)robot systems, we refer e.g. to (Parker, 2003).

In certain situations, robots have advantages over living beings, especially in hazardous or inaccessible places, or in settings where the creatures are prone to be hurt (Casper and Murphy, 2003). Due to the increasing number of tasks that could be assigned to robots and the concludingly growing number of robots needed to undertake these tasks the industry would immensely benefit from well-developed strategies. We here consider a multiagent system to simulate multiple robots and their tasks. This approach of developing strategies in a simulation and to possibly later transfer the knowledge is often used in order to prevent the robots from exhaustion (Michel, 2004).

In this work, we investigate a shepherding task, a task that is usually accomplished by shepherding dogs. The shepherding task is a type of *manipulation planning* (Lien and Pratt, 2009) or *group motion control* (Harrison et al., 2010; Vo et al., 2009), i.e. a group of units (the sheep) is manipulated by another group of units (the shepherds or dogs) in order to fulfill a given task (e.g. reaching a predefined target area). We model both the dogs and the sheep as agents to simulate several behaviors and to investigate effective strategies. Our agents abstract real robots and thus, the strategies discussed here are well adaptable to work on robots in the real world. Although little work has been done to use robots to control a herd of sheep, the results are encouraging: Vaughan et al. (2000) used real robots to control a group of ducks, whose behavior is similar to that of sheep. In a more recent work, the flight distance of sheep in response to a mobile robot was investigated and it turned out, that the sheep got accustomed to it which allows for new ways of steering the sheep (Evered et al., 2014).

But even apart from using robots in agriculture, strategies to control sheep have manifold areas of important applications:

Evacuation tasks, i.e. leading people from dangerous places to safety could be accomplished by previously placed robots that are activated in case of emergency (Martiínez-García et al., 2005). Kenny et al. (2001) emphasized the need to better understand crowds and their behaviors. Dyer et al. (2008) found that large crowds of people show a behavior similar to those of herds of animals and thus, shepherding strategies could be used to *simulate* the flow of units (animals or people) in a building or an urban area to help architects in creating save setups (Kirkland and Maciejewski, 2003). Additionally, those simulations could be used to organize and

improve evacuations or the responses to riots.

Guidance tasks, as e.g. studied by Burgard et al. (1998), Thrun et al. (2000) or Nourbakhsh et al. (2003) deal with people that are e.g. moved through a museum or around touristic attractions. These tasks are comparable to the above mentioned evacuation tasks but generally they can disregard critical situations and the intended movement of the people aims at entertaining the guests or visiting a given set of places of interest.

(Box-)*Pushing* (see e.g. (Donald et al., 1994) or (Matarić et al., 1995)) is another application of shepherding strategies: There, an object has to be moved to a destination by a set of robots. The object is sometimes too heavy or too large to be handled by one robot, only. Thus, they have to cooperate to accomplish the task. Also, construction operations or bulldozing assignments (Lau et al., 2011) could be solved with the help of shepherding behaviors. An additional example of pushing strategies are swimming robots that could encircle oil after oil spills and thus improve the manual strategies mentioned by Fingas (2002).

In the *entertainment area*, Cowling and Gmeinwieser (2010) discuss the usage of shepherding strategies in various *video games*. Further applications of shepherding strategies could e.g. be found in robot soccer (Lau et al., 2011).

The shepherding task is computationally hard as the size of the state is extremely large and requires multiagent cooperation (Vo et al., 2009). Although there already exists some research on shepherding in (multi-)agent or (multi-)robot systems using neural networks (Potter et al., 2001), potential fields (Vaughan et al., 1998), rules learned by a genetic algorithm (Schultz et al., 1996), or algorithms using hand-coded strategies (Lien et al., 2004), existing work does neither consider the theoretical background of the shepherding task nor offer considerations on the optimal solution.

In this work, we investigate the size of the state space and measure the influence of the numbers of the different agents types. To the best of our knowledge, this is the first theoretical investigation of such tasks. Later in Chapter 5, we develop strategies to solve the shepherding task, provide a measure to relate the performance to an optimal solution (i.e. we provide an upper and a lower bound for any optimal solution), and explore the power of learning in this particular task.

4.2 Biological Background

In order to highlight the authenticity of our model a brief biological background of shepherding needs to be provided.

Like several other animals (e.g. cattle, goats, poultry, or pigs) sheep were kept to provide for food, milk, or wool. In this work we restrict ourselves to the shepherding of sheep but note the transferability to other herded animals. Sheep are graminivore and are characterized as gregarious animals with a strong herd instinct and therefore individual members of the herd tend to follow the group. Flocking provides both each sheep and the group as a whole with safety (Hamilton, 1971; McFarland, 2006):

- While some members may graze, others can “stand guard” and watch out for predators.
- Furthermore, a large group even of weak animals can discourage attackers: Due to the size of the herd carnivores are often irritated because they cannot decide

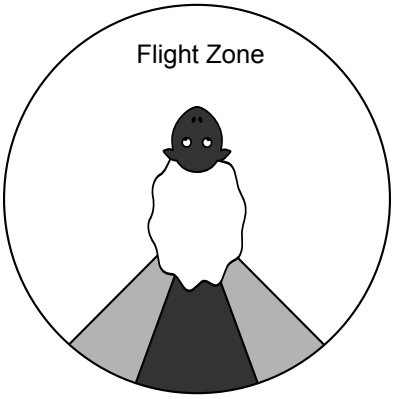


Figure 4.2: Flightzone of sheep: The dark grey area is the blind spot of a sheared sheep while the lighter grey area is the blind spot of a wooly sheep. The white area constitutes a coverage of 270 to 320 degrees (Shulaw, 2005).

which individual to hunt.

- Even if some members of the herd are caught the group itself survives.

Due to their good eyesight sheep are able to detect threats early. Additionally, sheep have a visual field of 270 to 320 degrees (Shulaw, 2005) which allows to detect danger from behind without even turning their heads. Sheep react to other approaching individuals (e.g. people or animals) that enter the sheep's *flight zone* by backing away. This area is determined by the visual field which is influenced by the anatomy (e.g. form of head or the amount of wool) of the sheep and is depicted in Figure 4.2. The size of the flight zone is connected to the tameness of the individual animal (Grandin, 1994).

Dogs, which are often used to drive and protect sheep, have a visual field of 250 degrees (Walls, 1963). Trained working dogs are able to act on commands of a human shepherd or even partly independently. Their tasks include the protection of the herd against predators as well as maintaining a compact flock structure even in the absence of fences. Additionally, they are used for guiding and moving the sheep. This is done in several ways that include entering the flight zone, barking, or snapping.

On the one hand, herding is crucial to protect the flock against theft, flight, or attacks. On the other hand, it may be necessary to move the flock e.g. to other pastures, to the shearer, or to the slaughterhouse. In this work we focus on the *movement* of sheep and therefore use the terms herding or shepherding only for this purpose.

In general, sheep are driven by entering their flight zone (cf. Figure 4.2): This will result in the sheep moving away from the intruder[1]; the direction and the speed of movement can be influenced by the angle or the speed of approaching the sheep (Grandin, 1994). If the intruder leaves the flight zone the sheep usually stops moving and it continues its previous activity as e.g. grazing. The behavior of dogs that is exploited to use them for shepherding emerged from predation which is present to some extend in any dog (Coppinger and Coppinger, 2007). Selective breeding has

[1] Note, that a sheep (mostly rams or female sheep with lambs) may choose to attack the intruder if it comes to close or if the animal is cornered.

lead to effective herding dogs: Although the dogs' hunting instinct—or, to be more precise, the *prey drive*—is still present, they no longer see the sheep as food and thus refrain from serious attacks (Sundance, 2009).

Due to the flocking behavior of sheep, even those sheep that are not directly affected will follow the collective flight. This allows effective shepherding as in general the number of dogs is by magnitudes smaller than the number of sheep (Green and Woodruff, 1993).

4.3 Description of the Shepherding Task

The Shepherding task for a group of n agents (the *dogs*) is to drive a herd of m reactive agents (the *sheep*) into a designated target area with as little steps (for the dogs) as possible (cf. Figure 4.1). The sheep have a limited viewing range and are afraid of the dogs, i.e. they try to *flee* if the distance between sheep and dog is less than the sheep's viewing range. Although the dogs may have a limited view as well, they are basically allowed to always perceive the sheep's positions which could be seen as an infinitely large viewing range. All agents are able to perceive the position and the type of all agents and obstacles inside their viewing range.

The implementation of the dogs and the sheep differ in their discriminability: Dogs can be identified and discriminated (they "look different" and each dog has a unique ID) while each sheep can only be identified as "the sheep at position (x, y), i.e. they "look identical".

The *environment* is modeled as a grid world with cells that can be occupied by at most one agent or an obstacle and neither the dogs nor the sheep can leave the environment. Obstacles are modeled such that they cannot be passed but agents are able to look over them (e.g. think of obstacles as being pits).

In the following definition, we introduce *instances* of the Shepherding task:

Definition 2 (Shepherding Instance (Baumann and Kleine Büning, 2013)). *An instance of* Shepherding(n, m) *is described by a tuple* $(w, h, r_{sheep}, r_{dog}, \mathcal{G}, \mathcal{D}, \mathcal{S}, \mathcal{O}, \mathcal{C}_{dog}, \mathcal{C}_{sheep})$ *where*

- n, m *are the numbers of dogs and sheep.*
- $w, h \in \mathbb{N}$ *are the numbers of cells in* x*- and* y*-direction. These values define a set* $A = \{(x_a, y_a) \mid x_a, y_a \in \mathbb{N}_0 \wedge x_a < w \wedge y_a < h\}$ *of all cells in the area.*
- $r_{sheep}, r_{dog} \in \mathbb{N}_0$ *are the viewing ranges of the sheep and the dogs, respectively. We use* ∞ *to denote an unlimited viewing range (i.e. the agent(s) can perceive the complete environment).*
- $\mathcal{G} \subset A$ *is a set of target cells.*
- $\mathcal{D}$ *is a set of* n *dog agents.*
- $\mathcal{S}$ *is a set of* m *sheep agents.*
- $\mathcal{O} \subset A$ *is a set of cells that are occupied by obstacles and that cannot be entered by agents.*
- $\mathcal{C}_{dog} = \{(x_i, y_i) \mid x_i, y_i \in (A \setminus \mathcal{O}),\ i \in \mathcal{D}\}$ *is a set of* n *cells that are occupied by dogs.*
- $\mathcal{C}_{sheep} = \{(x_j, y_j) \mid x_j, y_j \in (A \setminus \mathcal{O}),\ j \in \mathcal{S}\}$ *is a set of* m *cells that are occupied by sheep.*

The following example shows the instance that is depicted in this chapter's introduction:

Example 1. *The instance in Figure 4.1 can be described as* Shepherding$(1, 1) = (w, h, r_{sheep}, r_{dog}, \mathcal{G}, \mathcal{D}, \mathcal{S}, \mathcal{O}, \mathcal{C}_{dog}, \mathcal{C}_{sheep})$ *with* $w = 4$, $h = 3$, $r_{sheep} = 1$, $r_{dog} = \infty$, $\mathcal{G} = \{(3, 2)\}$, $\mathcal{O} = \varnothing$, $\mathcal{C}_{dog} = \{(0, 1)\}$, $\mathcal{C}_{sheep} = \{(2, 1)\}$.

Note, that we slightly abstract from the biological model of dogs and sheep by e.g. allowing a visual field of 360 degrees although the visual field in sheep is between 270 to 320 degrees (Shulaw, 2005) and around 250 degrees in dogs (Walls, 1963). Additionally, we do not discriminate between the visual field and the flight zone (Shulaw, 2005) of sheep: As all agents are without orientation, there is no natural way of defining places "behind" the sheep. Thus, the flight zone here is circular instead of conical as in real sheep and is considered to be identical to the visual field. Furthermore, we find the term "viewing range" most appropriate as the agents only have access to information within this distance. On the contrary, the reaction of the sheep in our task is coherent to "real" sheep: In both cases the sheep moves as long as there is an intruder in its flight zone.

In this work, we only deal with initially solvable instances, and to ensure this solvability, every Shepherding instance is initialized such that

- all agents are placed randomly in the environment.
- each cell contains at most one agent.
- no agent is placed upon an obstacle.
- no dog agent is visible to any sheep.
- no sheep is on a target cell.
- is has at least as many target cells as sheep (i.e. $|\mathcal{G}| \geq |\mathcal{S}|$).
- there is an unblocked path for any sheep such that all sheep may end up on a target cell.

We consider a discrete time model and thus we can look at "snapshots" (in the future called *states*) of the system at any given time step. To formally capture this, we define states[2] of Shepherding instances:

Definition 3 (States of Shepherding Instances). *A* state *of a* Shepherding *instance* $(w, h, r_{sheep}, r_{dog}, \mathcal{G}, \mathcal{D}, \mathcal{S}, \mathcal{O}, \mathcal{C}_{dog}, \mathcal{C}_{sheep})$ *is a tuple* $(dog_1, \ldots, dog_n, sheep_1, \ldots, sheep_m)$ *where*

- $dog_1, \ldots, dog_n$ *are the positions of the dogs in fixed order (e.g. ordered by the agents' ids), and*
- $sheep_1, \ldots, sheep_m$ *are the positions of the sheep sorted increasing by their x-coordinates (with increasing y-coordinates as tie breaker)*

at the current time step and thus comprises all information that may change. A state is called a goal state *if all sheep are placed on a target cell and thus the instance is solved.*

[2] The difference between instances and states lies in the contained information: A state, although maybe correct in several instances, depends on its instance as the state includes e.g. no information on the size of the environment.

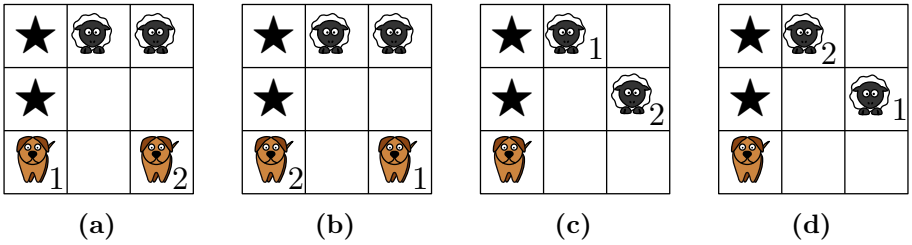


Figure 4.3: The two states in (a) and (b) should be distinguishable as dog 1 and dog 2 may have learned different strategies to deal with this situation. On the contrary, since all sheep behave identically and follow a fixed strategy, sheep 1 and sheep 2 do not need to be distinguished and the states depicted in (c) and (d) can be treated equally.

As mentioned before, the states of Shepherding instances allow the distinction of dogs whereas sheep are indistinguishable. This reflects the fact, that we are concerned to derive a useful behavior for the more sophisticated dog agents (e.g. by learning different strategies as it was discussed e.g. by Potter et al. (2001)). The sheep's behavior can be considered identical for each sheep and should thus be considered as part of the environment. Figure 4.3 shows an example of several states that should be distinguishable (Figures 4.3(a) and 4.3(b)) as well as states that do not need to be distinguishable (Figures 4.3(c) and 4.3(d)). The sorting with respect to the coordinates as described in Definition 3 ensures determinism. Note, that due to this, there may be several goal states even if the number of target cells equals the number of sheep: Consider the set of target cells that is occupied by a set of sheep. We can now interchange the sheep on the same set of target cells while the state that describes the situation remains the same. But, as argued before, this is not necessary.

If each sheep should be able to have a different behavior it may be necessary to store the sheep's positions as described for the dogs.

Example 2. *The states depicted in Figure 4.3 can be described as*

$$\begin{aligned} \textit{(a)} \quad & (\underline{(0,0)}, \underline{(2,0)}, (1,2), (2,2)). \\ \textit{(b)} \quad & (\underline{(2,0)}, \underline{(0,0)}, (1,2,)(2,2)). \\ \textit{(c) and (d)} \quad & (\underline{(0,0)}, (1,2), (2,1)). \end{aligned}$$

where the dog's positions are underlined.

As the dogs and the sheep move around, the sets $\mathcal{C}_{dog}$ and $\mathcal{C}_{sheep}$ change over time but have to always be legal, i.e.

- each cell is occupied by at most one agent (dog or sheep).
- no agent is located on or has crossed an obstacle.
- the maximal distance of one step is respected.
- no agents collided during movement.
- all agents are within the boundaries of the environment.

This means that the state transition function (i.e. the function that models the impacts of the agent's actions) has to consider these rules. In fact, the system only changes its state s_t at time t to a different succeeding state s_{t+1} if the action a_t

performed in s_t was legitimate[3].

To describe what we are looking for, we give the following definition for solutions of Shepherding instances:

Definition 4 (Solution of a Shepherding Instance). *A solution of a given* Shepherding *instance is a sequence of states, that are visited to transform the initial state of the instance into a goal state. This sequence includes the goal state and all transitions between each two succeeding states have to regard all constraints mentioned before. The number of these transitions is called* solution length*.*

Based on Definition 4 we can compare solutions regarding their solution lengths. We are interested in an *optimal* solution, i.e. a solution that has minimal length as this implies a solution with the smallest number of steps needed by the dogs.

Although we only consider solvable instances, an initially solvable instance can result in a state that is not solvable. Depending on the sheep behavior, the sheep can be moved to a position from which it cannot be moved to a target cell.

With this formulation in mind, we are now able to implement the Shepherding task with in a multiagent system.

4.4 Modeling the Shepherding Task as Multiagent System

As mentioned above, the Shepherding task is modeled in a (multi-)agent system (cf. Section 2.1). Although both the dogs as well as the sheep are agents that are situated in an environment, there is an important distinction between them: The sheep are *part of* the environment and are thus controlled by it whereas the dogs are not controlled by the environment. In the remainder of this work, we will consider two ways of controlling the dogs; namely an algorithmic approach (cf. Section 5.2) as well as the learning of appropriate behavior (cf. Chapter 6).

4.4.1 Agent Architecture

Here, we describe the used agent models and relate them to existing models and paradigms from literature.

In the taxonomy of Russel and Norvig (2010), the sheep agents are *simple reflex* agents as they only utilize behavior that breaks down to condition-action rules. Furthermore, they only react to external events (this could be a dog entering the sheep's viewing range or a random event that triggers the flee instinct). The dog agents are *model-based, goal-based* agents[4] and are more sophisticated as they have knowledge of the environment's model and know about their goals.

According to Stone and Veloso's (2000) description of agent architectures, the agents employed here are *homogeneous* and *non-communicating*. Another characteristic of the agent architecture measures the sophistication of agents: While *reactive*

[3] In Section 4.4.2 and Section 4.4.3 we describe the possible actions and the transition function in more detail.

[4] Even the learning approach in this work does not completely fit Russel and Norvig's view of a learning agent, as their model is rather an *online* learning agent whereas reinforcement learning agents usually learn *offline* and use the learned policy the same way as any other action selection method would be used.

agents only react (i.e. choose a predetermined behavior) to a certain situation or to an external stimulus, *deliberative* agents behave more "intelligently". The latter model includes an internal representation of the enclosing environment and the ability to predict the effects of performing specific actions in certain situations. Applying these characterizations to the agents of the Shepherding task, it is quite obvious, that the sheep can be considered reactive (they only move if they are externally triggered to do so) while the dogs are deliberative as they possess a model of the environment and move reasonably to pursue their goal of collecting all sheep in the target area. Note that the sheep may be considered to be not purely reactive as the sheep try to maximize the distance to all visible sheep (thus, they have some kind of a goal). This "fuzziness" in classifying agents as reactive or deliberative has already been mentioned by Stone and Veloso:

> "Although the line between reactive and deliberative agents can be somewhat blurry, an agent with no internal state is certainly reactive, and one which bases its actions on the predicted actions of other agents is deliberative."
>
> (Stone and Veloso, 2000)

Iocchi et al. (2001) present a taxonomy for multirobot systems that consists of four different levels:

- *Cooperation*, i.e. the execution of a given task depends on or is improved by the presence of several robots.
- *Knowledge* of the other robots' presence in the system.
- *Coordination*, i.e. the robots take into account the other robots' actions in order to guarantee a performant system.
- *Organization*, i.e. whether the decision making is centralized or distributed.

Additionally, there are two dimensions that are orthogonal to the previous four: The usage of *communication* as well as the distinction between *homogenous and heterogeneous* agents, where homogenous agents have differences either in the hardware or in the software.

Although our multiagent system consists of two types of agents and would thus be heterogeneous per se, we classify the agents considered in this work separately[5]. The sheep agents are

- *non-cooperative*, as the sheep neither operate together nor do they follow a global task.
- aware of the other agents inside their viewing range (*knowledge*).
- *not coordinated*, i.e. they do not base their decision on the actions performed by other sheep.
- *organized* completely distributed, i.e. there is no central instance that dictates their movement.
- *neither communicating* among each other nor with the dog(s).
- *homogeneous*, i.e. every sheep has the same behavior and in a given situation every sheep will react identically.

[5] Clearly, all agent characteristics can be altered to resemble a different interpretation of the given task. The sheep for example may of course be coordinated or even communicating if this becomes necessary. Additionally, they may e.g. follow a flocking behavior.

The dogs on the other hand

- are *cooperative* as all dog agents have the common goal to drive the sheep into the target area.
- have *knowledge* of the other agents inside their viewing range (that is in general assumed to be infinite but may of course be restricted).
- can be *coordinated* depending on the behavior.
- form some kind of *organization* that highly depend on the fact of whether they are aware of each other and what the do with this potential knowledge.
- do not *communicate* if the approaches described in this work are used but in general, the dogs could clearly make use of communication.
- *homogenous* if they use the algorithmic approach and may be *heterogeneous* if the use the learning approaches.

Iocchi et al. (2001) also distinguish between *reactive* and *deliberative* behavior: The sheep agents are reactive because of their behavior based controller that is activated by the agents' sensors. The dogs are more complex and their architecture is a hybrid between the *behavior-based* controller of the sheep and a *Sense-Model-Plan-Act* architecture that consists of four steps:

1. *Sensing* to collect sensor information of the environment.
2. Building a *model* of the current state of the environment.
3. Based on the model created in the last step, a planner builds a *plan* that contains the actions to be performed to reach a certain goal.
4. After an appropriate sequence of actions has been found, the commands are converted in low level commands that allow the robot to *act* accordingly.

The distinction of our approaches lies in the fact, that they only compute the next action to perform instead of a sequence, that is needed to reach the goal. A clear advantage of our approach is the fact that the system is able to react quickly to changes in the environment.

Figure 4.4 adapts the view of Lettmann et al. (2011) that builds upon Russel and Norvig's representation (Russell and Norvig, 2010): We use a box to represent the sheep's *reflex controller* and a cloud to depict the *deliberative controller* of the dogs. While Russel and Norvig place the agents outside the environment, in (Lettmann et al., 2011) we argue to embed the agents into the environment. We here follow the same approach but we emphasize the clear differences between dogs and sheep regarding their capabilities and their knowledge of the environment or of the task.

One could of course argue, that the sheep are only entities that add dynamic to the environment and are not "real" agents. Nevertheless, we modeled the sheep as agents to provide the possibility of implementing more elaborate sheep behavior. The close connection of the reactive sheep to the environment is expressed by the greenish color of the agents in Figure 4.4.

Although the representation in Figure 4.4 is two-dimensional, only, we point out that the mental processes (i.e. interpretation of perceptions and selection of appropriate actions) of the agents take place on the *agent model level*. This level can be thought above the *system level* where the "real" interaction between agents and environment take place (see e.g. Fig. 3 in (Lettmann et al., 2011)).

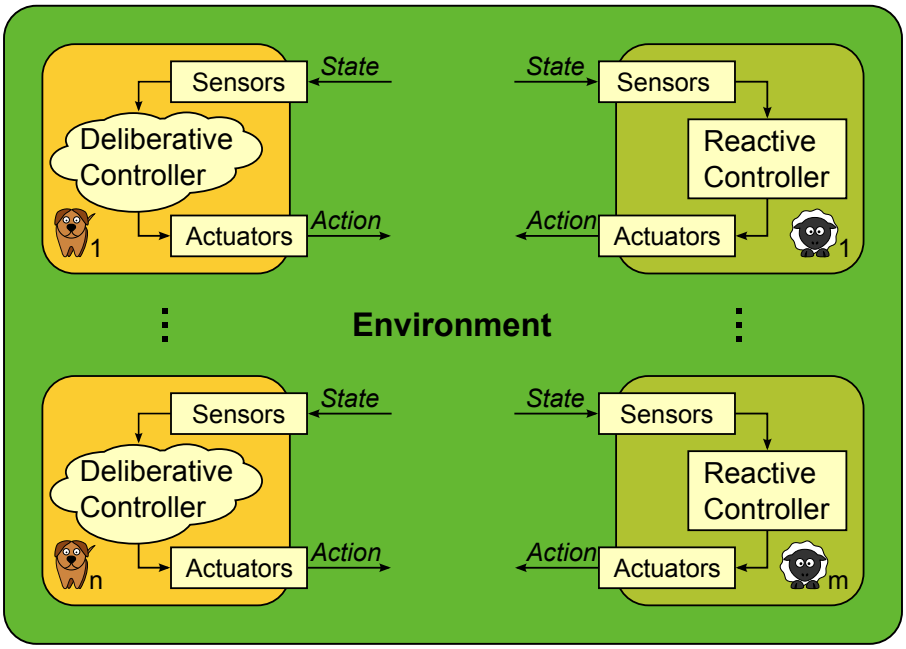


Figure 4.4: The shepherding task modeled as agent system (following the notion of Lettmann et al. (2011))

4.4.2 The Agents' Action Space

The agents can move one step in any cardinal direction or decide not to move at all. Thus, a *von Neumann neighborhood* is used. As already mentioned, all agents in this work are considered to be without orientation (i.e. they move in any direction without having to turn before). Whenever we talk about an orientation like "the dog is moving left", we give the orientation relative to another object of the environment.

To globally describe the actions, we chose the cardinal points (i.e. "north" corresponds to upwards on the grid and "east" corresponds to right). Thus, the action sets for the sheep agents and the dog agents are $A_{sheep} = A_{dog} := \{step_{north}, step_{east}, step_{south}, step_{west}, stand\}$, respectively. See e.g. Figure 4.7 on page 62: If the sheep goes to its left, the global action would be $step_{north}$ and it goes forward, the global action would be $step_{east}$. Note that an agent can be forced to remain on its position although it decided to move, e.g. if it would collide with another agent or tries to leave the environment and thus, there may be a difference between the desired action and the one that is actually performed.

The joint action is build in the same order as mentioned in Definition 3. Note, that the joint action to perform in one round is built step by step as first the dogs decide upon their action and the sheep react to these actions.

4.4.3 State Transition Function

As we have already seen, the environment passes several states from its initialization until an instance is solved. We call the transition between two consecutive states a *round* (cf. Figure 4.5). With Definition 4 in mind, the solution length of a given solution equals the number of rounds needed for this solution.

Figure 4.5: Schematic image of the relation between states and rounds

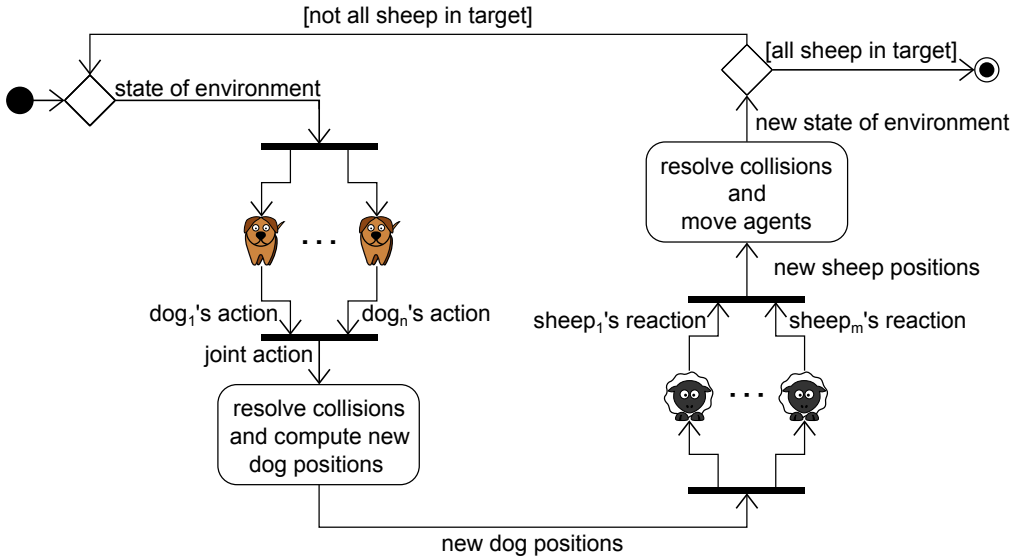


Figure 4.6: Sequence of one round where the dogs compute their actions individually and the sheep react to the possibly corrected movement.

Figure 4.6 depicts the procedure of one round and is the content of the "blackbox" drawn in Figure 4.5. Every round (we here consider the agent system of being in timestep t at the beginning of the round) consists of:

1. The dogs perceive the current state s_t of the environment.
2. Based on s_t each dog decides upon its preferred action to perform next.
3. The joint action that comprises all actions selected by the dogs is passed to the environment.
4. The environment resolves all collisions and ensures that no agent tries to leave the environment.
5. After executing the resulting joint action of the dogs, the new dog positions are passed to the sheep.
6. Based on the sheep's behavior (cf. Section 4.5), the sheep possibly move.
7. If the new state of the environment is neither a goal state nor a game-over state, the new current state is passed to the dogs again, and the cycle is restarted at Item 1.

Note, that the dogs decide in parallel and, if they are not communicating, they do not take into account actions selected by other agents in the very same round. From all these individual actions of the dog agents, the joint vector is built and then fed into the environment.

This joint-action vector is checked by the environment to guarantee that each transition from a state to its successor is legal. First, the environment ensures that

no dog tries to leave the environment or clashes with an obstacle. Second, collisions between agents (dogs and sheep) are resolved by forcing all involved agents to remain on their current positions. Additionally, the immediate position change of two agents is prohibited.

After all collisions are resolved, the dogs' positions are passed over to the sheep which can then react to the dog's movement. Each sheep[6] takes the positions of the dogs *after* movement and the positions of the sheep *before* movement and locally computes a new position. These positions are then checked for collisions as described before and, after all collisions are resolved, all agents move to their new positions.

After the movement of the sheep, the environment checks, if the instance is solved (i.e. all sheep are on target cells) or if the goal can never be achieved from the current state (depending on the sheep behavior, game-over states are possible). Otherwise, the resulting state of the environment is passed to the dogs for the next round.

Up do now, we only said that the sheep *somehow* move. In the next section, we take a deeper look into the decision process that leads to the movement.

4.5 Sheep Behavior

The *deterministic* behavior models a reactive sheep that only moves if a dog entered its viewing range (i.e. at least one dog is closer than r_{sheep} to it). The main goal of each sheep is to maximize the sum of (Manhattan-)distances to all visible dogs. Note, that this computation uses the positions of the dogs *after* their movement in round t. Although other dogs may come into sight (or others diminish) by moving to a new position, only the dogs visible *before* the sheep's movement are considered (i.e. the positions of the dogs computed in round $t-1$).

The sheep are allowed to move one step to a neighboring cell. Thus, the potential new positions of the sheep are all in the von Neumann neighborhood of range one around the current position (cf. Figure 4.8(a)). Of course, the movement has to be legal, i.e. the new position has to be inside the boundaries of the environment, and must not be occupied by another agent or an obstacle. If several positions with identical sums of distances exist, the order of preference is (relative to the nearest dog[7]): *straight away*, *left*, *right*, *standing still*, and *towards the dog* as can be seen in Figure 4.7. Note that the movement of other sheep can block the movement as all sheep decide individually and move in parallel (cf. Section 4.4.3). Albeit being reasonable, this preference (except the evasive movement) is arbitrary but to guarantee a deterministic behavior, some tie-breaking rules had to be established.

Figure 4.8 shows an example of the sheep's tie-breaking preferences: Generally, the sheep has five possible targets (the four neighboring cells as well as the current position, cf. Figure 4.8(a)). In the considered situation, two dogs approach the sheep resulting in the situation depicted in Figure 4.8(b). There, the sums of distances to the dogs are denoted. Obviously, there are two positions, that both maximize the distances and, according to the sheep preferences described before and illustrated in Figure 4.7, the sheep takes one step to its left (in relation to the dog).

[6] The order of the sheep is the same as described in Definition 3.

[7] Or the dog with the smallest ID, if there are several dogs with identical distance to the sheep.

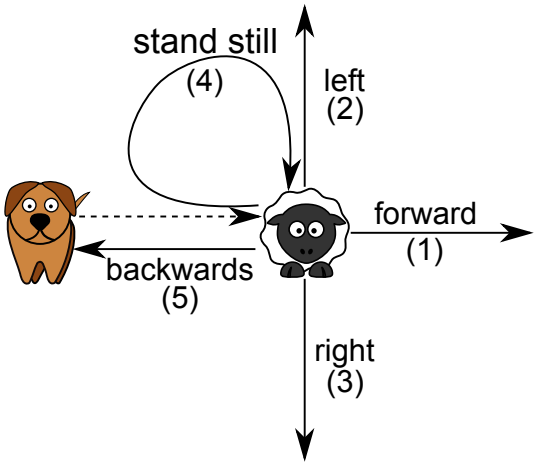


Figure 4.7: Movement preferences of the sheep (relative to the dog that approaches the sheep from the left): The sheep would move forward by performing the global action $step_{east}$. In this example the relations to the other "global" actions in the agent system are: "left" $\widehat{=}$ $step_{north}$, "right" $\widehat{=}$ $step_{south}$, and "backwards" $\widehat{=}$ $step_{west}$.

Note, that a deterministic reactive sheep can be moved to a border cell from where it cannot be freed by a single dog agent. Thus with this behavior, initially solvable instances can result in states from which the goal (i.e. all sheep are situated on target cells) cannot be achieved.

4.6 Complexity of the Shepherding Task

In this section we investigate the state-space complexity of the Shepherding task. In the following, we use $n = |\mathcal{D}|$ and $m = |\mathcal{S}|$ to denote the numbers of dogs and sheep, respectively (cf. Definition 2). Additionally, we use O to refer to the Landau notation.

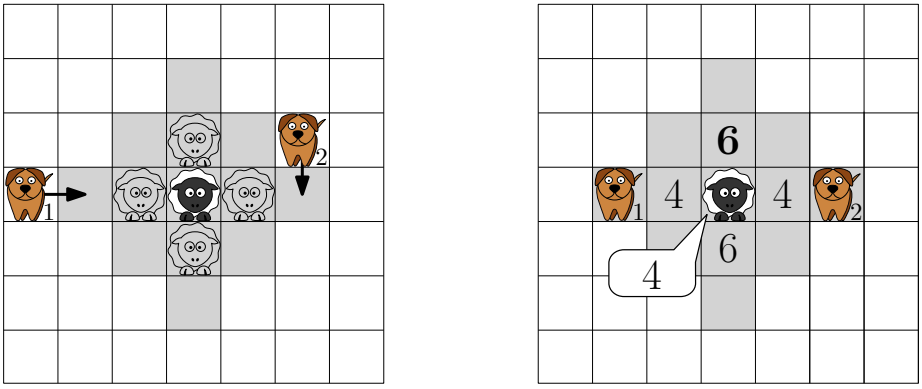
One metric to measure the complexity of tasks is the *branching factor* for the nodes of the corresponding search tree. The branching factor is the number of children this particular node has (Edelkamp and Korf, 1998). As the behavior of the sheep depends deterministically on all actions of the dogs (and thus on the joint-action vector) for each state there exist $|A|^n = 5^n$ possible following states which serves as upper bound for the branching factor.

In the following, we compute the *state-space complexity* of the Shepherding task. This measure of complexity is defined as the number of legal states that may emerge from the initial state of an instance (Allis, 1994).

For an instance $(w, h, r_{sheep}, r_{dog}, \mathcal{G}, \mathcal{D}, \mathcal{S}, \mathcal{O}, \mathcal{C}_{dog}, \mathcal{C}_{sheep})$ of the Shepherding task, the size of the state space, i.e. the number of possible states is bounded by

$$(w \cdot h)^{(n+m)} .$$

Note that this number contains invalid states, e.g. states in which several agents are positioned on one cell and obstacles are not considered, either. A closer bound can be derived by considering two crucial facts that follow from Definition 3: First,



(a) **(b)**

Figure 4.8: Example showing the possible new positions for the sheep in (a) as well as their corresponding sums of distances to the dogs in (b). The bold number in (b) marks the target of the sheep according to the preference in Figure 4.7.

dog agents are distinguishable and second, sheep agents are not distinguishable. The number of legal states is thus given by

$$\underbrace{\binom{w \cdot h - |\mathcal{O}|}{n} \cdot n!}_{\text{possible dog positions}} \cdot \underbrace{\binom{w \cdot h - |\mathcal{O}| - n}{m}}_{\text{possible sheep positions}} \tag{4.1}$$

as the dogs are distributed with considering the order (i.e. it does matter "which dog sits where") and the sheep are distributed without taking into account the order (i.e. it is disregarded "which sheep sits where").

Before we move on to the simplification of Equation (4.1) and to a further analysis, we first introduce some helping statements. To begin with, Lemma 1 shows that we can change the order of permutation (i.e. selection that disregards order) and combination (i.e. selection that considers the order) on a given set without changing the result:

Lemma 1 (Interchangeability of Permutation and Combination). *Given a set with n elements and $k_1, k_2 \in \mathbb{N}_0$ with $k_1 + k_2 \leq n$, the number of possible combinations for the following approaches are the same:*

- *First selecting k_1 from n elements with order taken into account (permutation, ${}_nP_{k_1}$) and then selecting k_2 elements from the remaining $n-k_1$ elements without considering the order (combination, $C_{k_2}^{n-k_1}$).*
- *First selecting k_2 from n elements without considering the order (combination, $C_{k_2}^{n}$) and then selecting k_1 elements with order taken into account from the remaining $n - k_2$ elements (permutation, ${}_{n-k_2}P_{k_1}$).*

Thus, the following two terms are equivalent:

$$\underbrace{\binom{n}{k_1} \cdot k_1!}_{{}_nP_{k_1}} \cdot \underbrace{\binom{n-k_1}{k_2}}_{C_{k_2}^{n-k_1}} = \underbrace{\binom{n}{k_2}}_{C_{k_2}^{n}} \cdot \underbrace{\binom{n-k_2}{k_1} \cdot k_1!}_{{}_{n-k_2}P_{k_1}}$$

Proof. Reformulation and rearrangement gives:

$$\begin{aligned}
\binom{n}{k_1} \cdot k_1! \cdot \binom{n-k_1}{k_2} &= \frac{n! \cdot k_1! \cdot (n-k_1)!}{k_1! \cdot (n-k_1)! \cdot k_2! \cdot (n-k_1-k_2)!} \\
&= \frac{n! \cdot k_1!}{k_1! \cdot k_2! \cdot (n-k_1-k_2)!} \\
&= \frac{n! \cdot k_1!}{k_1! \cdot k_2! \cdot (n-k_2-k_1)!} \\
&= \frac{n! \cdot k_1! \cdot (n-k_2)!}{k_1! \cdot k_2! \cdot (n-k_2-k_1)! \cdot (n-k_2)!} \\
&= \frac{n!}{k_2! \cdot (n-k_2)!} \cdot \frac{(n-k_2)!}{k_1! \cdot (n-k_2-k_1)!} \cdot k_1! \\
&= \binom{n}{k_2} \cdot \binom{n-k_2}{k_1} \cdot k_1!
\end{aligned}$$

□

Next, Corollary 1 allows to rewrite certain fractions that consist of factorials as a product:

Corollary 1. *Let $n, k \in \mathbb{N}_0$ be two numbers with $k \leq n$, then*

$$\frac{n!}{(n-k)!} = \prod_{i=n-k+1}^{n} i\,.$$

Proof. Reformulation gives:

$$\begin{aligned}
\frac{n!}{(n-k)!} &= \frac{1 \cdot 2 \cdot \ldots \cdot (n-k-1) \cdot (n-k) \cdot (n-k+1) \cdot \ldots \cdot n}{1 \cdot 2 \cdot \ldots \cdot (n-k-1) \cdot (n-k)} \\
&= (n-k+1) \cdot \ldots \cdot n \\
&= \prod_{i=n-k+1}^{n} i
\end{aligned}$$

□

The next corollary gives an estimate for products of numbers from a given range:

Corollary 2. *Let $k, l \in \mathbb{N}_0$ be two numbers with $k \leq l$, then*

$$\prod_{i=k}^{l} i \leq l^{(l-k+1)}$$

with equality for $k = l$.

From Lemma 1 it follows, that it is irrelevant whether one first places the dogs or the sheep and thus, it is sufficient to consider Equation (4.1). Next, we derive a more compact term for the size of the state space (Note, that we use Corollary 1 to derive Equation (4.3) from Equation (4.2) and Corollary 2 to obtain Equation (4.4)

from Equation (4.3).):

$$\begin{aligned}
&\binom{w \cdot h - |\mathcal{O}|}{n} \cdot n! \cdot \binom{w \cdot h - |\mathcal{O}| - n}{m} \\
&= \frac{(w \cdot h - |\mathcal{O}|)! \cdot n! \cdot (w \cdot h - |\mathcal{O}| - n)!}{n! \cdot (w \cdot h - |\mathcal{O}| - n)! \cdot m! \cdot (w \cdot h - |\mathcal{O}| - n - m)!} \\
&= \frac{(w \cdot h - |\mathcal{O}|)!}{m! \cdot (w \cdot h - |\mathcal{O}| - n - m)!} \qquad (4.2) \\
&= \frac{1}{m!} \cdot \prod_{i=w \cdot h - |\mathcal{O}| - n - m + 1}^{w \cdot h - |\mathcal{O}|} i \qquad (4.3) \\
&\leq \frac{1}{m!} \cdot (w \cdot h - |\mathcal{O}|)^{(n+m)} \qquad (4.4)
\end{aligned}$$

The bound on the number of possible states given in Equation (4.3) is clearly in $\mathrm{O}((w \cdot h - |\mathcal{O}|)!)$ where $w \cdot h - |\mathcal{O}|$ is the number of (potentially) accessible cells[8]. For an increasing number of agents (i.e. n and/or m increase), the number of states increases exponentially, while for small n, m the number of states approaches the number of allowed cells. Clearly, the number of dogs is much more influential[9] as the faculty of the number of sheep is also in the denominator of Equation (4.2) and thus reduces the growth of the number of states.

Corollary 3 (Number of Possible States for Shepherding Instances). *For an instance* $(w, h, r_{sheep}, r_{dog}, \mathcal{G}, \mathcal{D}, \mathcal{S}, \mathcal{O}, \mathcal{C}_{dog}, \mathcal{C}_{sheep})$ *of the* Shepherding *task, the size of the state space is bounded by*

$$\frac{(w \cdot h - |\mathcal{O}|)!}{m! \cdot (w \cdot h - |\mathcal{O}| - n - m)!} \leq \frac{1}{m!} \cdot (w \cdot h - |\mathcal{O}|)^{(n+m)}$$

where $w \cdot h - |\mathcal{O}|$ *is the number of (potentially) accessible cells. Thus, the bound on the number of possible states grows exponentially in the number of agents.*

Figure 4.9 illustrates the connection between the number of the sheep and the number of the dogs: The considered environment had $w = 10$ cells times $h = 10$ cells without any obstacle which results in 100 cells. For this setting, legal instances with varying numbers of agents were evaluated. Thus, for all combinations of dogs and sheep with at most 100 agents in total the size of the state space was computed. The maximal number of $9.407 \cdot 10^{157}$ states is achieved with $n = 98$ and $m = 1$ (98 dogs and one sheep). It is obvious, that the number of dogs has a much larger impact on the size of the state space than the number of sheep.

After this analysis, we like to relate the above results to other tasks. To compare, the game of Go on a 10×10 board has a search space complexity of $9.64 \cdot 10^{46}$ while Go with a by-the-book board of 19×19 fields has a search space complexity

[8] Note that there may e.g. exist cells that are not accessible because they are "encircled" by obstacles and thus, the effective number of accessible cells may be lower than $w \cdot h - |\mathcal{O}|$.

[9] Roughly speaking, for each additional dog the number of states is multiplied by the number of accessible cells.

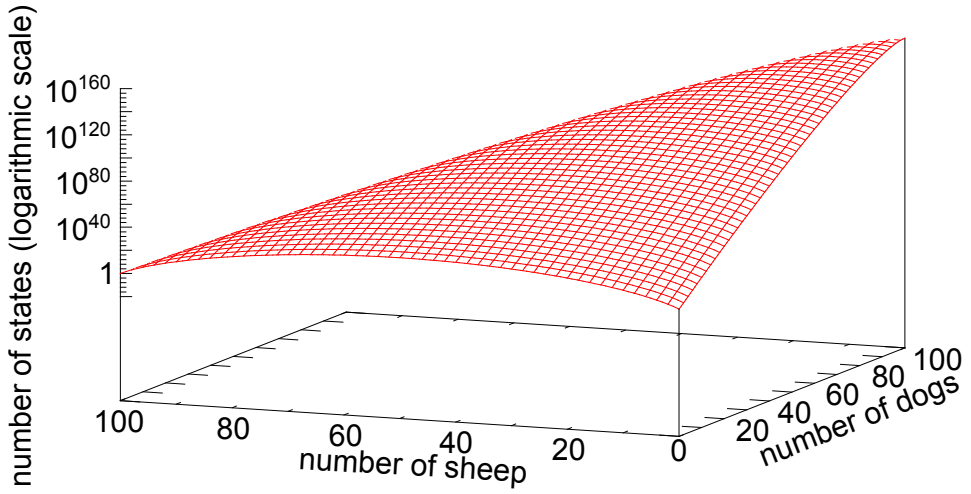


Figure 4.9: Number of states for an instance with 100 cells and varying numbers of dogs and sheep.

of $2.082 \cdot 10^{170}$ (Tromp and Farnebäck, 2007). Similarly, chess on a standard 8×8 board has a state-space complexity of about $5 \cdot 10^{52}$ (Allis, 1994).

In 2007, checkers with approximately $5 \cdot 10^{20}$ states was solved (Schaeffer et al., 2007) but even with this result, the solving of chess is not yet in sight. The next game to be solved is most probably Othello (also named Reversi) with a state-space complexity of about 10^{28} (Allis, 1994) although even this rather moderate step towards the state-space size of chess will require substantially more resources than the solution of checkers (Schaeffer et al., 2007).

4.7 Conclusion

In this chapter, the Shepherding task for agents was introduced. We highlighted the importance of this particular task and pointed out similar tasks that would benefit—either directly or with some adjustments—from shepherding strategies.

First, we had a look at the biological background of shepherding and the behavioral interaction of dogs and sheep that allow the control of large flocks by a small number of dogs. With this foundation, we formalized the task of dogs driving sheep to a target area. The Shepherding task was then implemented as multiagent system and we related the resulting system to existing taxonomies from literature.

We carefully analyzed the state-space complexity of the Shepherding task and showed that it grows exponentially in the number of agents. These results were compared to the complexities of other games and we saw that even the most recent results in solving board games are far away from state space with sizes similar to the task of controlling sheep.

In the succeeding chapter, we present a greedy approach that solves the Shepherding task with one sheep and one dog in linear time and whose solution are proven to be within tight bounds around the optimal solution.

5
Single Agent Shepherding

After having already pointed out the significance of dealing with shepherding(-related) tasks in Section 4.1, we here focus on means to solve such tasks.

In this chapter, we present solutions for instances of SHEPHERDING$(1, 1)$ using a greedy algorithm whose worst-case solutions are close to optimal and whose runtime is linear for plausible scenarios. The core idea of our *greedy coordinate correction* (*GCC*) approach is to iteratively correct the sheep's coordinates: From Section 4.3 we know that the only way to reposition the dog to change the orientation of the sheep without disturbance is to walk around the viewing range. As every such circumnavigation costs additional steps we strive to minimize the need of repositioning the sheep by computing a shortest path with the minimal number of bends. The central approach of *GCC* is depicted in Figure 5.1: The dog selects the closest feasible position to control the sheep and then drives it along one axis (the x-axis in the example) until the first coordinate (the x-coordinate in the example) of the sheep is equal to the corresponding target coordinate. Finally, the remaining coordinate (the y-coordinate in the example) is corrected until the sheep is on the target. This behavior prevents the usage of an arbitrary shortest path (dashed lines) with up to four orientation changes but instead results in a path for the sheep with only one bend.

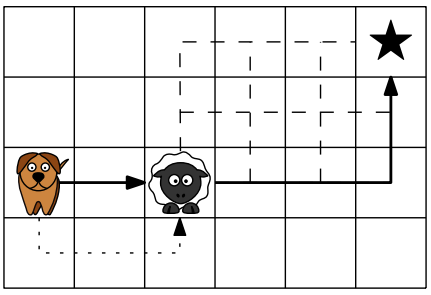


Figure 5.1: The dog goes to the nearest driving position of the sheep and moves the sheep to the target in such a way that at most one repositioning is necessary.

We analyze the proposed algorithm and prove that its solution lies within close bounds, i.e. the lower and the upper bound for any Shepherding$(1,1)$ instance only differ by a term linear in the sheep's viewing range. Thus, for any obstacle-free instance of Shepherding$(1,1)$, *GCC* guarantees a solution from within this bound without knowing the actual setup of the instance.

Additionally, we model the Shepherding task as reinforcement learning task and later in Section 10.7 we compare the behavior computed by the *GCC* algorithm to learned strategies. There, we also analyze in which situations which approach is superior.

Some authors investigated the learning of shepherding tasks using neural networks (Potter et al., 2001), or rules learned by a genetic algorithm (Schultz et al., 1996). Others used potential fields (Vaughan et al., 1998), algorithms with hand-coded strategies (Lien et al., 2004), or heuristics imitating the behavior of real dogs (Strömbom et al., 2014). Unfortunately, existing works do not give provably optimal solutions and the quality of the learned behavior is hard to assess.

In essence, the intention of this chapter is to build a theoretical basis to assess the *quality* of learned strategies, which subsequently aggravates the possibility of assessing the learned behaviors.

The main results given in this chapter are the following:

- We analyze the shepherding task from Chapter 4 using Manhattan geometry in Section 5.1 where we review existing results as well as providing some new statements to help us analyze the proposed algorithm.
- We introduce the *greedy coordinate correction* (*GCC*) algorithm that solves obstacle-free instances with one target of Shepherding$(1,1)$ in Section 5.2.
- In Section 5.3, we provide close upper and lower bounds on the solution lengths computed by the *GCC* approach: The maximal length of any optimal solution is proven in Lemma 3 and Lemma 4 while the minimal length (that might even not be reached by an optimal solution) is proven in Lemma 5. The main result is then provided in Theorem 1 which shows that the solutions computed by the *GCC* algorithm are close to the optimal solution, i.e. the upper and the lower bound differ by a term linear in the sheep's viewing range.
- Section 5.3.4 shows that any instance of Shepherding$(1,1)$ can be solved with the aforementioned quality in linear time using the *GCC* algorithm.
- We model the Shepherding task as reinforcement learning task in Chapter 6

Apart from the above mentioned contributions, we analyze the *GCC* algorithm and explain instances that are hard to solve for *GCC* in Section 5.3.2. Finally, Section 5.4 concludes this chapter.

Some of the results presented here are extensions of previously published material (Baumann and Kleine Büning, 2013) which, to the best of our knowledge, was the first theoretical investigation of this kind of tasks.

5.1 Foundations

This section reviews the concept of Manhattan geometry to analyze the *GCC* approach and to derive the bounds on the optimal solution in Section 5.3. We review properties of paths in Manhattan geometry in order to formulate how the dog agent

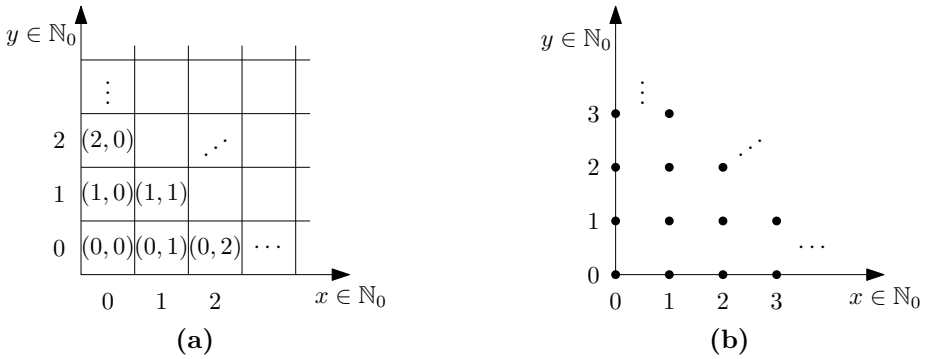


Figure 5.2: Enumeration of cells in our model: (a) shows how the cells are enumerated while (b) displays the equivalent lattice in the $\mathbb{N}_0^2$ space.

should move best. As the viewing range of the sheep defines a circle, we review the concept of circles and circumferences in Manhattan geometry. The reader familiar with those concepts may want to skip this section (and continue reading on page 78) and return when necessary.

5.1.1 Manhattan Geometry

In this work we model the environment surrounding the agents as a two-dimensional grid with discrete cells. We enumerate the cells of the grid as depicted in Figure 5.2(a) with the cell in the lower left being the origin. In Figure 5.2(b) the corresponding lattice in the coordinate system of the $\mathbb{N}_0^2$ space is shown.

We employ the following convention: For a position or a point p from a two-dimensional space we write x_p for its x-value and y_p for its y-value. Furthermore, a cell is part of the environment and an agent has a position that is located on a cell.

In order to measure distances, a *norm* is needed to assign a positive length (or zero) to a given vector. An important class of vector spaces is the L^p space, in which the length of a vector $x = (x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$ is usually given by the p-norm:

$$\|x\|_p = \left(\sum_{i=1}^{d} |x_i|^p \right)^{1/p} \tag{5.1}$$

with $p \geq 1 \in \mathbb{R}^d$. With this, the general metric $\mathrm{d} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ between two points u, v can be constructed:

$$\mathrm{d}_p(u, v) = \|u - v\|_p \tag{5.2}$$

The probably most prominent example of this metric is the Euclidean norm for $p = 2$ that was mentioned in Section 2.3 and that is used in later parts of this thesis, as well. Since we are dealing with a grid world and the agents are limited to only make steps to directly neighboring cells, we use the Manhattan distance ($p = 1$) between two given points.

Definition 5 (Manhattan Distance). *The Manhattan Distance between two positions $u, v \in \mathbb{R}^d$, is defined as*

$$\mathrm{d}_M(u, v) = \|u - v\|_1 = \sum_{i=1}^{d} |u_i - v_i| \ .$$

In our grid representation for the environment, this definition is adjusted to consider the fact that the cells' coordinates are two-dimensional and integer:

Definition 6 (Manhattan Distance in a 2D-Grid). *Given two positions* $p_1 = (x_1, y_1)$ *and* $p_2 = (x_2, y_2)$ *with* $x_1, x_2, y_1, y_2 \in \mathbb{N}_0$*, the Manhattan distance between these two positions is*

$$\mathrm{d}_M(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2|$$

with $\mathrm{d}_M(p_1, p_2) \in \mathbb{N}_0$.

With this definition we are able to talk about paths and especially, shortest paths in the considered grid environment.

5.1.2 Manhattan Paths

In the following, we introduce two possibilities to express paths in the Manhattan metric. The first one, *rectilinear paths* will be the mostly used concept while the second, *lattice paths* is presented solely to prove some properties of such paths. Other means of dealing with paths in different applications exist although they are out of scope for this thesis.

As described in Section 4.4.2 we are dealing with a von Neumann neighborhood in Manhattan geometry and thus, the agent is allowed to take steps to its directly neighboring cells. In the context of analyzing paths we define steps as follows:

Definition 7 (Steps in Gridworlds). *A step* $\vec{s}$ *between two neighboring positions* p_1 *and* p_2 *in a grid with* $\mathrm{d}_M(p_1, p_2) = 1$ *is given by the vector*

$$\vec{s}(p_1, p_2) = \overrightarrow{p_1 p_2} = \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix}.$$

Taking into account the properties of the von Neumann neighborhood, the above definition results in four possible steps that will be abbreviated as $step_{north} := (0, 1)$ for a step up, $step_{east} := (1, 0)$ for a step to the right, $step_{south} := (0, -1)$ for a step down, and $step_{west} := (-1, 0)$ for a step to the left (cf. Figure 4.7). These steps are collected in the set $\mathfrak{S}$ of all allowed steps.

Rectilinear Paths

Our ultimate goal is to find paths that require as few turnings of the robot as possible. The motivation for this approach stems from practical considerations as every turn of a real robot would require slowing down, turning, and accelerating again (de Berg, 1991). One highly appropriate concept to model such paths is called *rectilinear paths* (see e.g. (Larson and Li, 1981; de Berg, 1991)):

Definition 8 (Rectilinear Paths). *A rectilinear path*

$$P(u_0, u_n) = ((u_0, u_1), (u_1, u_2), \ldots, (u_{n-1}, u_n))$$

is a sequence of connected segments $seg_i = (u_i, u_{i+1})$ *with* $0 \leq i \leq n - 1$ *such that each segment* seg_i *connecting the points* u_i *of the path and* u_{i+1} *is axis-parallel.*

The most efficient approach is clearly to store segments of maximal length, i.e. combining consecutive segments with the same direction. We make use of this assumption in the following.

Lattice Paths

A different approach for dealing with rectilinear paths is their representation as lattice paths as e.g. presented by Hilton and Pedersen (1991). Its main difference to rectilinear paths as mentioned before is the modeling: While the first approach uses line segments, the latter one stores every point of the path appearing between the start and the end:

Definition 9 (Lattice Path). *A lattice path is a sequence of positions $u_0, u_1, \ldots, u_n$ where each position u_i is located on the grid and for all u_i, u_{i+1} where $0 \leq i < n$ the vector $\overrightarrow{u_i u_{i+1}}$ is in the set $\mathfrak{S}$ of allowed steps.*

As we will see later, this concept easily allows to count the number of possible shortest paths. Other definitions of lattice paths store the sequence of steps taken but clearly, these approaches are easily transferable. Furthermore, additional extensions like diagonal steps are possible but beyond the scope of this thesis.

Comparison of the Above Concepts

Both concepts introduced before are here used to model paths in Manhattan geometry that only consist of segments that are each parallel to the coordinate axes. Thus, the transformation from one approach to the other (and vice versa) is straightforward: Given a rectilinear path as defined in Definition 8, a lattice path can be constructed by storing every position that is visited while follwing the path. The other way round is similar as segments are created from sequences of points during which the direction does not change.

5.1.3 Path Metrics

In this thesis, the most relevant measures of paths are their lengths and their numbers of bends. The general challenge that we are confronted with is the task of finding a path that has the minimal length and, if there are several shortest paths, selecting one that has the minimal number of bends[1].

Length of a Path

We start with the introduction of lengths for given paths:

Definition 10 (Length of a Path). *Given a path $P(u_0, u_n) = ((u_0, u_1), (u_1, u_2), \ldots, (u_{n-1}, u_n))$ and some distance measure $\mathrm{d} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, the length of P is given by*

$$\mathrm{L}(P) = \sum_{(u_i, u_{i+1}) \in P} \mathrm{d}(u_i, u_{i+1}) \,.$$

[1] This problem was also dealt with e.g. by Yang et al. (1991) where the authors used the term "minimal bend shortest path".

Analogously, the length of a path equals the number of steps needed to go from the start point to the target.

With the above definition, one can easily define the shortest path, i.e. a path connecting two positions with the shortest length among all possible paths between those points:

Definition 11 (Shortest Paths). *Given two positions $s, t \in \mathbb{R}^d$ and some distance measure $\mathrm{d} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, the shortest path $P_L^\star$ from s to t is given by*

$$P_L^\star(s,t) := \underset{P(s,t)\in\mathcal{P}}{\arg\min}\, \mathrm{L}(P(s,t))$$

where $\mathcal{P}$ is the set of all allowed paths.

In Definition 11 as well as in Definition 12, the set $\mathcal{P}$ is used to model only allowed paths as in the general case, obstacles may prohibit some path layouts.

Additionally, we need the number of bends of a path as this value tells how often the agent has to be repositioned.

Bends of a Path

In addition to the length of a path two other (closely related) measures can be expressed: The *number of segments* counts the axis-parallel segments and the *number of bends* expresses how often the agent following the path has to turn.

As already mentioned before, we assume all segments to be of maximal length, i.e. a segment covers all positions in the same direction and is not divided into several "sub-segments".

As the number of segments $|P|_{\text{seg}}$ of a rectilinear path can easily be derived from the size of the sequence representing the path, the most obvious way to derive the number of segments for a lattice path is to first transform it into the rectilinear representation as described before.

The connection between the number of segments and the number of bends $|P|_{\text{bend}}$ is obvious: Obviously, a path with n segments needs $n-1$ turns and thus:

$$|P|_{\text{bend}} = |P|_{\text{seg}} - 1$$

Clearly, a minimization task for the number of bends can be formulated similar to the definition of shortest paths in Definition 11.

Definition 12 (Minimal Bend Path). *Given two positions $s, t \in \mathbb{R}^d$, the minimal bend path $P_{bends}^\star$ from s to t is given by*

$$P_{bends}^\star(s,t) := \underset{P(s,t)\in\mathcal{P}}{\arg\min}\, |P(s,t)|_{bend}$$

where $\mathcal{P}$ is the set of all allowed paths.

This concept of minimizing bends in paths is highly relevant for the development of chips or circuits (see e.g. (Raghunath et al., 1986)). Other names for the number of segments include e.g. link distance (Mitchell et al., 1992).

5.1.4 Monotone Paths

One property of shortest paths in Manhattan geometry (and also in other metrics) is *monotonicity*: When following a shortest path in environments without obstacles, the distance of the agent to the target never increases. We use this concept of monotonicity to enable the transfer of results for lattice paths to rectilinear paths.

Other definitions for monotonicity of paths include e.g. argumentations with non-negative inner products (Arkin et al., 1989) or orderings of the orthogonal projections of all path vertices on the vector $\overrightarrow{st}$ from the start s to end t (Lee and Preparata, 1984). Here, we use the following definition of monotone paths:

Definition 13 (Monotone Paths). *A path $P(u, v)$ between positions u and v is called* uv-monotone *if for every segment $seg_i = (u_i, u_{i+1})$ of the path*

$$\begin{cases} x_i \leq x_{i+1} \wedge y_i \leq y_{i+1} & \text{if } x_u \leq x_v \wedge y_u \leq y_v \\ x_i \leq x_{i+1} \wedge y_i \geq y_{i+1} & \text{if } x_u \leq x_v \wedge y_u \geq y_v \\ x_i \geq x_{i+1} \wedge y_i \leq y_{i+1} & \text{if } x_u \geq x_v \wedge y_u \leq y_v \\ x_i \geq x_{i+1} \wedge y_i \geq y_{i+1} & \text{if } x_u \geq x_v \wedge y_u \geq y_v \end{cases}$$

holds. Thus, a path for which the above holds for every segment between the start s and the end t is completely monotone and is called st-monotone.

Whenever we talk of monotone paths we mean st-monotone paths unless otherwise stated. Additionally, we assume that by employing the concept of monotonicity the agent does not walk beyond the target in any direction.

5.1.5 Properties of Shortest Manhattan Paths

We now state some properties of shortest paths in Manhattan geometry that become important in the remainder of this chapter. The first one is the fact that every shortest path fullfils the monotonicity requirements stated in Definition 13.

Remark 1 (Every Shortest Path in Manhattan Geometry Without Obstacles is Monotone). *Given a Manhattan environment without any obstacle, then any shortest path $P(s, t)$ between any two positions s, t is monotone as defined in Definition 13.*

Proof. We prove this statement by contradiction: Let us assume a shortest path $P(s, t)$ that is not monotone. Without loss of generality let this path go "up and right", i.e. P fulfills $x_s \leq x_t \wedge y_s \leq y_t$ (the other seven cases can be proven analogously). Since path P is not monotone, P contains at least one segment $seg_i = (u_i, u_{i+1})$ that does not fulfill the monotonicity requirements from Definition 13. Thus, for such a segment seg_i it holds $x_i > x_{i+1} \vee y_i > y_{i+1}$. Since $\mathrm{d}_M(s, u_{i+1}) < \mathrm{d}_M(s, u_i) + \mathrm{d}_M(u_i, u_{i+1})$ we can adjust the path P and obtain a shorter path P' by going directly to u_{i+1} instead of going the detour over u_i. This is contradiction to the assumption that P was a shortest path. □

With the above mentioned fact, we can go on and investigate the number of different shortest paths between two positions. It is worth to note that we know from the proof of Remark 1 that every shortest path in our setting contains at most two directions; namely the ones that are directed towards the target.

Ambiguity of Shortest Manhattan Paths

Whereas a shortest path between two points in the (obstacle-free) Euclidean plane is unique, in the Manhattan metric this is not the case (Gardner, 1997). We here investigate the number of such shortest Manhattan paths and argue that there exist paths with a certain property, namely to have at most one bend.

As a first step, we give the number of shortest path from $(0, 0)$ to (x, y) with $x, y \in \mathbb{N}_0$ when the movement is restricted to going up (along the positive y-axis) and right (along the positive x-axis). As we have already seen earlier during the investigation of monotonicity, it is sufficient to only consider this kind of paths.

Due to the symmetry of Manhattan geometry, we show the number of paths only for the upper right quadrant. Later, the straightforward transformation to shortest paths with arbitrary start and target positions will be shown.

Proposition 1 (Number of Obstacle-Free Shortest Lattice Paths with Restricted Set of Directions (Hilton and Pedersen, 1991)). *Given a position $p = (a, b)$ with $a, b \in \mathbb{N}_0$ in a Manhattan grid without obstacles, the number of shortest paths from $(0, 0)$ to p with allowed movements "going up" ($\uparrow$) and "going to the right" ($\rightarrow$) is*

$$\binom{a+b}{a}.$$

Proof. We here follow the idea of Hilton and Pedersen (1991). Clearly, we need $|a|+|b|$ steps to go from $(0, 0)$ position $p = (a, b)$. It is also obvious, that we need $|a|$ steps along the x-axis and $|b|$ steps along the y-axis. To get the number of possible shortest paths between $(0, 0)$ and (a, b), we need to determine the number of combinations of "right" steps and "up" steps. This is equivalent to the question "how many ways to choose which of the steps goes up (right) exist" which is

$$\underbrace{\binom{a+b}{b}}_{\text{“up”}} = \underbrace{\binom{a+b}{a}}_{\text{“right”}}.$$

□

For a path that does not start in $(0, 0)$ an easy transformation can be applied such that the above proposition holds.

Corollary 4 (Number of Shortest Paths Between Arbitrary Positions). *Given two positions $p_1 = (a_1, b_1)$ and $p_2 = (a_2, b_2)$ in an obstacle-free Manhattan grid having $a_1 \leq a_2$ and $b_1 \leq b_2$, the number of shortest paths from p_1 to p_2 with allowed movements "going up" ($\uparrow$) and "going to the right" ($\rightarrow$) is*

$$\binom{(a_2 - a_1) + (b_2 - b_1)}{(b_2 - b_1)}.$$

Because of the symmetry of the Manhattan distance the above formula holds for any two positions in a Manhattan grid.

Corollary 5 (Number of Shortest Paths in Manhattan Geometry Without Obstacles). *Given two positions $p_1 = (a_1, b_1)$ and $p_2 = (a_2, b_2)$ in a Manhattan grid without obstacles having $a_1, a_2, b_1, b_2 \in \mathbb{Z}$, then the number of different shortest paths between p_1 and p_2 is given by*

$$\binom{|a_1 - a_2| + |b_1 - b_2|}{|b_1 - b_2|} = \frac{(|a_1 - a_2| + |b_1 - b_2|)!}{|a_1 - a_2|! \cdot |b_1 - b_2|!} = \frac{\mathrm{d}_M(p_1, p_2)!}{|a_1 - a_2|! \cdot |b_1 - b_2|!} .$$

Since we have several shortest paths between any two positions in Manhattan geometry, we can select the one that best fits our needs. In our case this will be the shortest path with the minimal number of bends, i.e. among all the shortest paths between two positions, we select one with the smallest number of turnings necessary.

Clearly, the smallest number of bends between two positions p_1, p_2 with $p_1 \neq p_2$ in an obstacle-free environment is one for a straight line (i.e. p_1 and p_2 only differ in one coordinate) and two if p_1 and p_2 differ two coordinates.

Shortest Manhattan Paths with at Most One Bend

For Manhattan geometry, de Berg (1991) stated that between any two points in a polygon there always exists a shortest path with the minimum number of bends. Since we deal with an obstacle-free environment, there exist two shortest paths each with the minimal number of bends (i.e. at most one).

Corollary 6 (Existence of Single-Bend Shortest Paths). *In an obstacle-free environment with Manhattan geometry between any two positions p_1, p_2 there exists two shortest paths that each have at most one bend.*

Proof. For the case when both points only differ in one coordinate the shortest path is clearly a straight line without any bends.

If the positions differ in two coordinates, we follow the idea from Proposition 1. We need to make $|x_1 - x_2|$ steps in along the x-axis and $|y_1 - y_2|$ steps along the y-axis. Without loss of generality let us assume that $x_1 < x_2$ and $y_1 < y_2$ and thus we only need to go up ($\uparrow$) and right ($\rightarrow$). Once again, we follow the same argumentation as in Proposition 1 and state that we can derive a shortest path in exactly two different ways:

- First go $|x_1 - x_2|$ steps to the right and then go $|y_1 - y_2|$ steps upwards, or
- first go $|y_1 - y_2|$ steps upwards and then go $|x_1 - x_2|$ steps to the right.

Clearly, each of these paths is a shortest path as the number of steps equals the Manhattan distance between p_1 and p_2 and both paths each consist of one bend. □

In the remainder of this thesis we will talk about "paths" without further specifying which concept we refer to. Since we pointed out the equivalency of both approaches we will use rectilinear paths unless otherwise stated.

5.1.6 Circles in Manhattan Geometry

After having talked about norms and distances, we can now begin to talk about circles, i.e. sets of all points that have the same distance (the radius) to the circle's

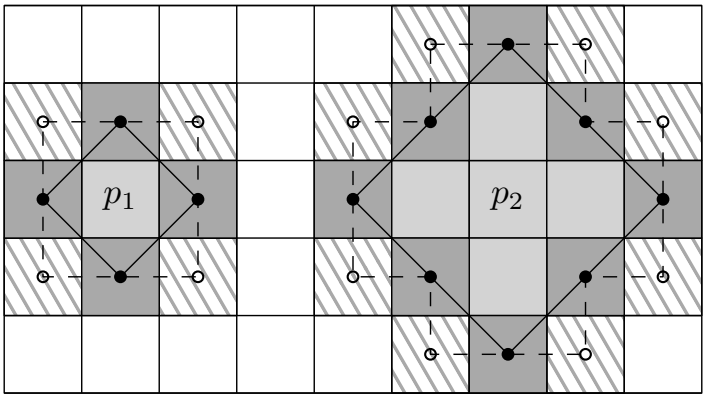


Figure 5.3: Circle with radius $r = 1$ around p_1 and radius $r = 2$ around p_2. The dashed lines indicate the circumference that consists of the points of the circle (filled points) as well as the outlined points. The hatched fields are cells that are part of the circumference but are not included in the circle.

center. This is necessary as the viewing range of the sheep defines a circle in which it reacts to approaching dogs. Later, we will use the circumference to count the number of steps to walk around a given position (the sheep) without coming closer than a predefined distance (the viewing range) to that position.

Although this concept is valid for arbitrary dimensions, we here restrict ourselves to the two-dimensional case. A two-dimensional circle with radius $r \in \mathbb{R}$ around the center $c_\circ$ in p-norm is defined as

$$\circ_p(c_\circ, r) := \{(x, y) \in \mathbb{R}^2 \mid \mathrm{d}_p(c_\circ, (x, y)) = r\} \tag{5.3}$$

$$:= \left\{(x, y) \in \mathbb{R}^2 \;\middle|\; (|x_\circ - x|^p + |y_\circ - y|^p)^{1/p} = r\right\} \tag{5.4}$$

With the Manhattan metric, a circle becomes a square that is rotated by 45 degrees relative to the coordinate axes as can be seen in Figure 5.3. Note, that for grids as considered here, the radius r can only be an integer which is dictated by the definition of the Manhattan distance (cf. Definition 6):

$$\circ(c_\circ, r) := \{p \in \mathbb{Z}^2 \mid \mathrm{d}_M(c_\circ, p) = r\} \tag{5.5}$$

$$:= \{p \in \mathbb{Z}^2 \mid |x_\circ - x_p| + |y_\circ - y_p| = r\} \tag{5.6}$$

In the following we will use the term "circle" interchangeably for the boundary of the figure (dark gray in Figure 5.3) as well as for all positions inside this boundary (dark gray *and* light gray cells in Figure 5.3).

In addition to the definition of a circle, the *circumference* of a given circle is defined as the distance around that circle. In Figure 5.3 it can be seen, that the circumference consists of all the points that are in the "circle set" from Equation (5.5) (marked in dark gray) as well as some points outside the circle (gray shaded).

In Manhattan geometry, the formula differs from the well-known formula in Euclidean metric:

Proposition 2 (Circumference of a Circle in Manhattan Geometry). *The circumference of a circle with radius r in Manhattan distance is $C(r) = 8r$.*

Proof. We adopt Adler and Tanton's (2000) approach in which

$$\int_0^r \left(\left| \frac{dx}{du} \right|^p + \left| \frac{dy}{du} \right|^p \right) du$$

was presented as formula for the arc length of the first quadrant and was used to compute values of π in different p-norms. Accordingly, we follow Adler and Tanton and parametrize the equation for circles as given in Equation (5.4) as follows:

$$x = u^{1/p}$$
$$y = (r-u)^{1/p}$$

With this, we are now able to express the circumference $C(r)$ of a circle with radius r which—due to the symmetry of the Manhattan geometry—is four times the arc length given above:

$$\begin{aligned} C(r) &= 4 \cdot \int_0^r \left(\left| \frac{dx}{du} \right|^p + \left| \frac{dy}{du} \right|^p \right) du \\ &= 4 \cdot \int_0^r \left(\left| \frac{u^{1/p-1}}{p} \right|^p + \left| -\frac{(r-x)^{1/p-1}}{p} \right|^p \right)^{1/p} du \\ &= \frac{4}{p} \cdot \int_0^r \left(|u|^{1-p} + |-(r-x)|^{1-p} \right)^{1/p} du \end{aligned} \tag{5.7}$$

In this thesis, we are interested in the Manhattan geometry (i.e. $p = 1$) and thus, Equation (5.7) becomes:

$$\begin{aligned} C(r) &= 4 \cdot \int_0^r (1+1)^1 du \\ &= 4 \cdot [2u]_0^r = 4 \cdot (2r - 0) = 8r \end{aligned}$$

□

5.1.7 Properties of Circles in Manhattan Geometry

We now recall some properties of circles in Manhattan geometry that are used in the remainder of this thesis:

We start with the (shortest) distance of arbitrary positions to given circles:

Proposition 3 (Distance to a Circle in Manhattan Geometry). *The Manhattan distance from a position p_1 to a circle of radius r around a position p_2 is*

$$\mathrm{d}_M(p_1, p_2) - r\,.$$

Proof. Consider a different circle, centered in p_1 whose radius r' is increased until both circles meet. Let $p^\star$ be one of those closest meeting points (cf. Figure 5.4). The distance from p_1 to this point $p^\star \in \circ(p_2, r)$ on the original circle can clearly be expressed as $\mathrm{d}_M(p_1, p^\star) = r'$. Since all positions in the circle set $\circ(p_2, r)$ have distance r to p_2, the distance from p_1 to p_2 can be expressed as $\mathrm{d}_M(p_1, p_2) = \mathrm{d}_M(p_1, p^\star) + r$. Thus, the claim follows. □

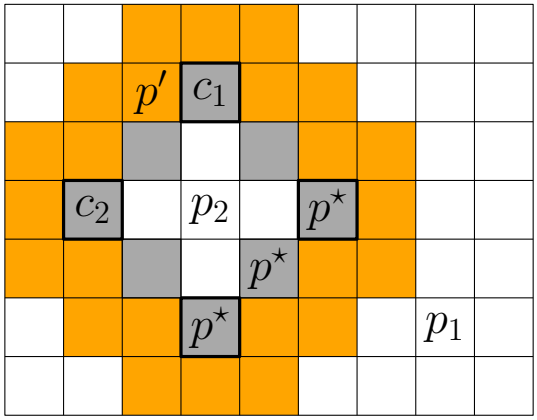


Figure 5.4: Example of a circle with radius $r = 2$ around position p_2 (gray cells). The cells $p^\star$ have the shortest distance to cell p_1 while p' is an arbitrary position on the circumference (orange cells) with distance $r' = 3$ to p_2. The thick framed cells are called *corners* of which c_1 and c_2 are two examples.

Obviously, Proposition 3 only applies to (one of) the nearest position(s) in the circle. With the following proposition, we are able to express distances to arbitrary positions in the circle $\circ(p_2, r)$.

Proposition 4 (Distance to any Point on a Circle in Manhattan Geometry). *Given a position p_1, any position p' with distance r to a position p_2 can be reached with at most*

$$\mathrm{d}_M(p_1, p_2) - r + \frac{1}{2}C(r)$$

steps without coming closer than r to p_2.

Proof. From Proposition 3, it follows, that at least one position with distance r to p_2 can be reached in $\mathrm{d}_M(p_1, p_2) - r$ steps. Starting at this position any other position p' with distance r to p_2 can be reached with at most $\frac{1}{2}C(r)$ steps by either following the circumference of the circle around p_2 clockwise or counterclockwise on the outside (cf. Figure 5.4). □

Figure 5.4 shows examples for the terms used in the preceding propositions. With these facts, we are now equipped to present our greedy shepherding algorithm in the following section.

5.2 A Greedy Shepherding Algorithm

In this section we present a greedy algorithm that solves obstacle-free Shepherding-instances with one dog and one sheep, i.e. $(w, h, r_{sheep}, \infty, \{target\}, \varnothing, \{dog\}, \{sheep\})$. Narrowing down the task allows us to establish a theoretical basis for the analysis of such tasks.

The core idea of our *greedy coordinate correction* (*GCC*) algorithm in Algorithm 3 is to drive the sheep until one coordinate equals the according target coordinate and then to steer the sheep to the target along the remaining axis as can be seen in Figure 5.5. To steer the sheep in a controlled way, the dog approaches the sheep

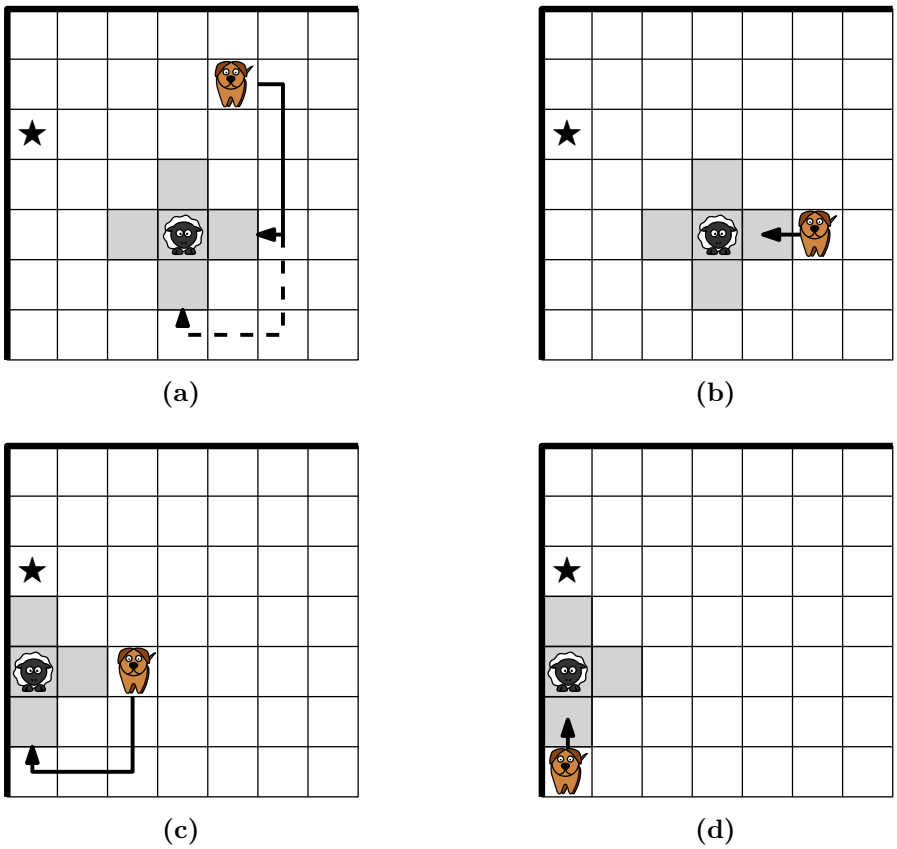


Figure 5.5: Example of how the dog drives the sheep to the target (★): It walks around the viewing range of the sheep (marked in gray) and then pushes the sheep until it is exactly below the target ((a)-(c)). Then, the dog walks again around the viewing range and drives the sheep to the target ((c)-(d)).

only from the corners of its viewing range. From the four corners, only two (the east and the south corner in Figure 5.5(a)) are feasible in order to move the sheep towards the target; approaching the sheep from any other corner would drive it farther away from the target. The dog chooses the closest corner (in this example the east corner) and walks to this position. In Figure 5.5(b), the dog approaches the sheep until the x-coordinates of the target and the sheep are identical (i.e. the sheep's first coordinate is corrected). After this, only the y-coordinate needs to be corrected and just one corner (in Figure 5.5(c) the south corner) remains feasible to approach. Finally, the dog has to walk around the viewing range of the sheep and drives the sheep towards the target until the last coordinate of the sheep is corrected and the sheep reaches the target (Figure 5.5(d)).

The benefits of the *GCC* approach include a runtime linear in the solution length and that it requires only constant storage (cf. Section 5.3.4).

5.2.1 Assumptions for the Greedy Shepherding Algorithm

To describe our algorithm and to prove its properties, we need several assumptions. Some of them are directly implied by the model while Assumption 5 (that is tighter

than Assumption 1 but we later show how to extenuate it) eases the proofs.

1. The sheep does not start on a border cell, i.e. $x_{sheep} \in [1, w-2]$ and $y_{sheep} \in [1, h-2]$.
2. At the beginning the dog is outside the viewing range of the sheep, i.e. $\mathrm{d}_M(dog, sheep) > r_{sheep}$.
3. In each step the dog has perfect knowledge about the sheep's position, e.g. $r_{dog} = \infty$.
4. The environment is larger than the sheep's field of view: $\min\{w, h\} \geq r_{sheep} - 2$.
5. In the beginning, the sheep has a distance of $r_{sheep} + 1$ to the border.

Note that Assumption 1 is necessary as the (deterministic reactive) sheep cannot be freed from a border cell and due to the sheep's model situations where the sheep is on a border cell, while the target is not, are unsolvable.

The second assumption's purpose is to ease the argumentation and can easily be (re-)established: As the sheep tries to get away from the dog it would start walking until the dog is outside the sheep's viewing range. This would result in an addition of r_{sheep} steps if the dog remains on its position or in $r_{sheep}/2$ steps if the dog actively tries to get out of the sheep's viewing range.

Assumption 3 is crucial to the later presented algorithm. Nevertheless, the task of finding the sheep in the first place is a different task and could be solved by environment exploration strategies as e.g. presented by Batalin and Sukhatme (2003). For the sake of argumentation we here rely on the dog's sensory superiority.

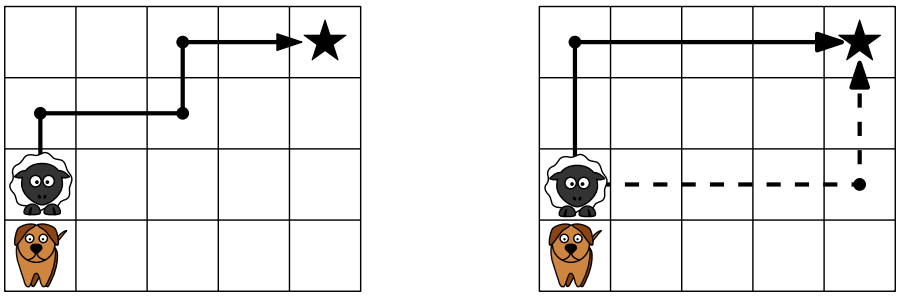
The fourth assumption ensures, that the sheep can be encircled by the dog without driving the sheep to the border. As a counterexample one can assume that the width or the height of the environment would exceed the viewing range of the sheep only by one. Then, the sheep's viewing range would "bounce" from one border to the other as only one field would be free for the dog to pass by. On the contrary, if the viewing range and the size of the environments satisfy the condition given in Assumption 4, the sheep can be driven such that the dog can freely walk around the sheep's viewing range.

As already mentioned above, the last assumption is used to ease the proof of the upper bound on the solution length in Lemma 3. Furthermore, in Lemma 4 we describe, how the sheep that is not on a border cell (and thus only fulfilling Assumption 1) can be moved such that Assumption 5 holds afterwards and how many steps are necessary for this. Although Assumption 1 is required only for initial states, we call the sheep *detached* if this assumption holds at any time during the solution.

5.2.2 Core Idea: Correcting One Coordinate at a Time

As mentioned before, we are interested in an optimal solution, i.e. a shortest sequence of states to transform the initial state of the instance into a solved one. Thus, we are clearly looking for a shortest path from the sheep's initial position to the target and, additionally, the dog should move as few steps as possible.

As the sheep reacts to the dog if it is inside the sheep's viewing distance, the dog can only move without disturbing the sheep if it is outside this distance. This results in the following challenge: If the sheep should be moved on a way that involves



(a) (b)

Figure 5.6: The *GCC* algorithm selects a shortest path with the minimal number of turnings (b) instead of using an arbitrary shortest path that requires several repositionings of the herding dog (a).

a turn (i.e. a change of direction), the dog has to step back and walk around the viewing distance of the sheep and then continue the driving. Since this results in additional steps, the dog has to carefully decide where and when it approaches the sheep. Thus, our approach attempts to move the sheep with minimal number of turnings, i.e. driving the sheep in one direction as long as possible.

The idea of the *greedy coordinate correction* algorithm is to first drive the sheep to a position where one of the sheep's coordinate equals the according target coordinate and then driving the sheep to the target by *correcting* the second coordinate (cf. the example given in Figure 5.5). Thus, we define coordinate corrections as:

Definition 14 (Coordinate Correction). *A coordinate of the sheep is* correct*, if its value equals the value of the according target coordinate.* Correcting *a coordinate is to steer the sheep along the respective axis until this coordinate is correct as just defined.*

After introducing this central idea, we continue to describe the objective of our *GCC* approach.

Figure 5.6 depicts the core idea of our algorithm: In order to move the sheep in Figure 5.6(b) from its initial position to the target (★) the dog first drives the sheep to the top (i.e. correcting the first coordinate) and then to the right (i.e. correcting the second coordinate) instead of driving the sheep along an arbitrary shortest path as indicated in Figure 5.6(a). Note, that the dashed line in Figure 5.6(b) is also a viable solution but the positioning of the dog and the sheep clearly requires the path depicted by the solid line. Both of the paths in Figure 5.6(b) contain the minimal number (i.e. one) of turnings (marked by the discs) while the path in Figure 5.6(a) would need three turnings. Remember, that each turning in the path would require the dog to reposition itself by walking around the viewing distance of the sheep.

We begin with showing that the idea of our algorithm is legitimate, i.e. that we can solve every instance of Shepherding that adheres to the assumptions mentioned before. Particularly, we transfer the findings from Section 5.1.1 to the scenario at hand.

Corollary 7 argues that this approach is legitimate, i.e. the sheep can be moved to the target by successively correcting its coordinates. The absence of the dog is

easy to achieve as due to the assumptions given before the dog starts outside of the viewing range of the sheep and thus, the dog can "move out of the way" without disturbing the sheep.

Corollary 7 (Existence of Single-Bend Shortest Paths in Shepherding). *Given a grid in Manhattan geometry without any obstacles and only one moving entity (an agent), then every shortest path between two positions p_1 and p_2 can be arranged such that the agent has to change its direction at most once without increasing the path's length (see Figure 5.6 for an example).*

Proof. Follows directly from Corollary 6 on page 75 that states, that any shortest path between two positions can be arranged in such a way that it contains at most one bend. □

After having talked about how the sheep has to move we now analyze in detail how the dog can drive the sheep to the target by applying the *GCC* approach.

5.2.3 Safely Controlling the Sheep

In order to drive the sheep in a controlled way the dog approaches the sheep from the corners of the viewing range (cf. Figure 5.8). We define *driving positions* in relation to the sheep's coordinates and indicate the positioning of allowed driving positions in Definition 16. The driving positions ensure, that approaching the sheep from those positions decreases the sheep's distance to the target. Additionally, they guarantee, that the sheep is driven to the border cells only if the target is on a border cell.

We start with the definition of a spatial relation to later select driving positions that are viable for the current position of the sheep.

Definition 15 (Spatial Relations). *Given two positions a, b, position a is behind b in relation to the target if both of the following conditions hold:*

$$\begin{aligned} \mathrm{d}_M(b, \mathit{target}) &< \mathrm{d}_M(a, \mathit{target}) \\ \mathrm{d}_M(a, b) &< \mathrm{d}_M(a, \mathit{target})\,. \end{aligned}$$

Thus, position a is behind b (in relation to the target) if b is closer to the target and the distance between a and b is smaller than the distance between a and the target (cf. Figure 5.7). Clearly, a more general condition for behindness in relation to an arbitrary position can easily be defined.

With this definition, we can now define the set of (allowed) driving positions:

Definition 16 ((Allowed) Driving Position). *The* driving positions *dp_i are located at the corners (cf. Figure 5.4 on page 78)* outside *the sheep's viewing range and have distance $r_{sheep} + 1$ to the sheep, i.e.*

$$\begin{aligned} dp_0 &= (x_{sheep}, y_{sheep} + r_{sheep} + 1) \\ dp_1 &= (x_{sheep} + r_{sheep} + 1, y_{sheep}) \\ dp_2 &= (x_{sheep}, y_{sheep} - r_{sheep} - 1) \\ dp_3 &= (x_{sheep} - r_{sheep} - 1, y_{sheep})\,. \end{aligned}$$

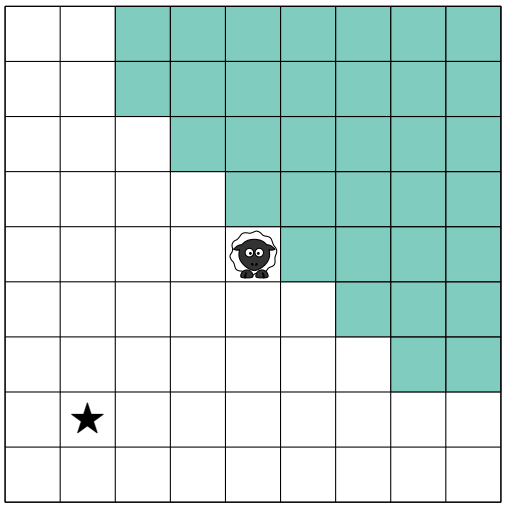


Figure 5.7: The green fields mark the cells that are behind the sheep regarding to Definition 15.

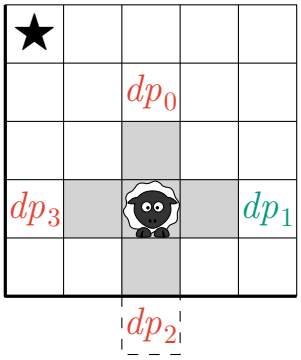


Figure 5.8: Example of *allowed driving positions*: If the sheep is e.g. below and right of the target (★), the only allowed driving position dp_1 is on the right of the viewing range (gray cells). Note, that dp_2 is not allowed as it is outside the environment and dp_0 and dp_3 are not allowed as approaching the sheep from these directions would drive the sheep away from the target.

The set $\mathcal{ADP}$ of allowed driving positions *contains all driving positions that are behind the sheep relative to the target (cf. Figure 5.8) and inside the environment (cf. Definition 15). Additionally, the corresponding coordinate does not have to and must not be corrected.*

Now we are ready to introduce the complete approach of our *GCC* algorithm.

5.2.4 Complete Approach

The *GCC* approach in Algorithm 3 first ensures that all allowed driving positions are accessible, i.e. the sheep has a distance of at least $r_{sheep}+1$ to all borders. There exist two possible violations: One with the sheep being closer than $r_{sheep}+1$ to *one* border and one with the sheep closer than $r_{sheep}+1$ to *two* borders. The first situation can be solved by walking between the border and the sheep (cf. Figure 5.9) which is due

Algorithm 3: Greedy Coordinate Correction (GCC)

input : instance of Shepherding$(1, 1) = (w, h, r_{sheep}, \{target\}, \varnothing, \{dog\}, \{sheep\})$

output : greedy solution for the given instance if the instance adheres to the assumptions given before

```
1 (select target ∈ 𝒢 closest to the sheep)
2 if not all allowed driving positions are inside the environment then
3     move clockwise around sheep
4 dp* = ⊥
5 while the sheep is not in the target do
6     if dp* = ⊥ ∨ current coordinate correct then
7         select closest allowed driving position dp* as in Definition 16
8         move to dp* ∈ 𝒜𝒟𝒫 without penetrating the viewing range
9     approach the sheep in direction dp* sheep (overarrow)
```

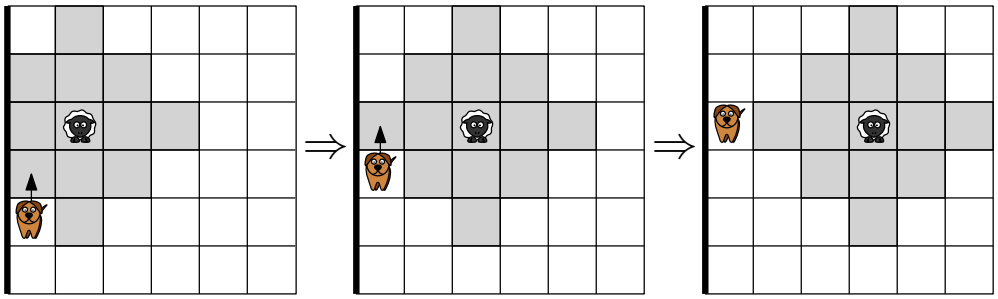


Figure 5.9: If the sheep is closer than $r_{sheep} + 1$ to one but not on the border, it can be driven from a wall if the dog approaches it clockwise until the respective coordinate is equal.

to Assumption 1 always possible. The solution for the second case is achieved by two repetitions of the behavior just mentioned (cf. Figure 5.11). Thus, the first lines of Algorithm 3 ensure that Assumption 5 holds if the initial instance only fulfills the first four assumptions. In addition to the correct behavior to free the sheep from the wall as depicted in Figure 5.9, Figure 5.10 shows a negative example in which the sheep will due to its behavior never be freed from the wall.

Obviously, at most two coordinates have to be corrected in any instance as the environment is a two-dimensional plane. As already mentioned, our approach corrects the sheep's coordinates consecutively: The algorithm chooses the closest allowed driving position $dp^\star \in \mathcal{ADP}$, and moves the dog to this position without penetrating the viewing range of the sheep. Clearly, such a path exists as the sheep has now a distance of $r_{sheep} + 1$ to any border which leaves a gap of at least one between the viewing range and any border.

From the selected driving position, the according direction is directly implied by the vector $\overrightarrow{dp^\star\ sheep}$. Then, the dog approaches the sheep and it walks straight away as depicted in Figure 5.5(b). The dog approaches the sheep to the left or right (up or down) until the x-values (the y-values) of the sheep and the target are equal.

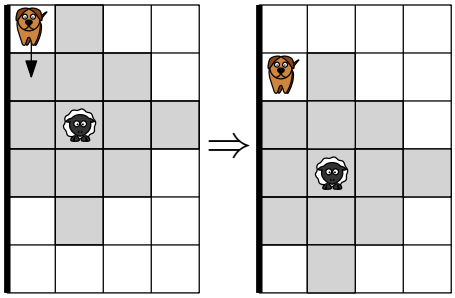


Figure 5.10: Contrary to the correct example in Figure 5.9 where the sheep was approached clockwise, the sheep walks in the same direction as the dog (cf. Section 4.5) and would keep its distance to the border if it is approached counterclockwise.

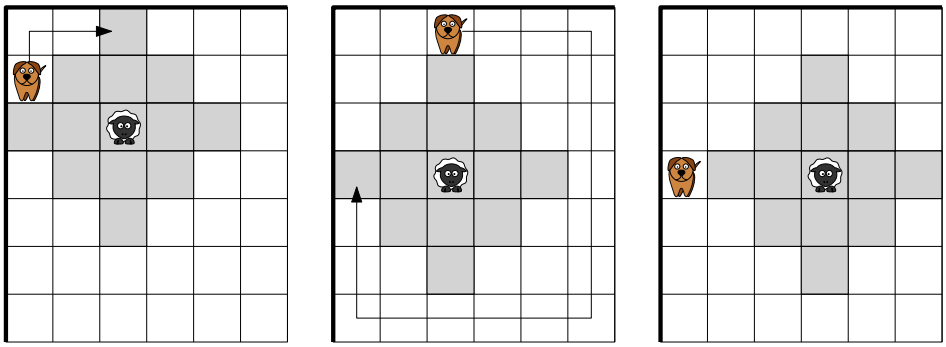


Figure 5.11: The dog can free the sheep from a corner (i.e. if the sheep is closer than $r_{sheep} + 1$ to two borders but not on any border) of the environment by circling the viewing range (b) and entering the corners of the viewing range ((a) and (c)).

After the current coordinate is corrected, the dog stops approaching the sheep and walks to the next allowed driving position if the sheep is not in the target, yet. This next driving position is on the other axis (cf. Figure 5.5(c)) which means that the dog has to travel a distance of $\frac{1}{4}C(r_{sheep} + 1)$ around the viewing range[2]. Finally, the dog approaches the sheep again until both coordinates are corrected and the sheep is in the target.

5.3 Analysis of the *GCC* Algorithm

In the following, we analyze the properties of *GCC* for instances with one target but the extension to dealing with several targets has a straightforward approximation: For every instance the closest target can be considered. Clearly, a global optimal solution could be computed by taking into account an optimal distribution of sheep to targets. Nevertheless, realistic scenarios would include one cohesive target area that may of course consist of one cell per sheep.

[2] From Proposition 2 on page 76 we know that the circumference of a circle in Manhattan geometry is $C(r) = 8r$ and thus the dog can walk on a $C(r_{sheep} + 1)$ steps long circle with radius $r_{sheep} + 1$ around the sheep's viewing range r_{sheep} without being noticed by the sheep.

5.3.1 Correctness of the Approach

We now continue with the analysis of the *GCC* algorithm. In Lemma 2, we prove that our approach finds a solution in cases in which the sheep is farther away than r_{sheep} to all borders (i.e. lines 5–9 of Algorithm 3).

Lemma 2 (*GCC* Solves Shepherding Instances). *Given the Assumptions 1–5, the dog can steer the sheep from any cell to any target cell by subsequently correcting the sheep's coordinates.*

Proof. Due to Corollary 7, every shortest path of the sheep from its initial position to the target can be arranged such that at most one turning is necessary. As the sheep has a distance of at least $r_{sheep} + 1$ to any border, the dog can reach any position outside the sheep's viewing range without moving the sheep. Particularly, the dog can reach all driving positions (cf. Definition 16). From the task definition in Section 4.5 follows that the sheep prefers moving straight away from the dog in case there are several positions with equal distance to the dog (cf. Figure 5.5). Thus, the dog can access any driving position and correct one of the sheep's coordinates using the nearest allowed driving position $dp^\star$.

After this, the dog can reach three or four driving positions:

- Three driving positions are reachable if the target is closer than r_{sheep} to the border and now the sheep is closer than r_{sheep} to the border, too. Then the driving position opposing $dp^\star$ is no longer reachable since it is outside the environment.
- Otherwise, four driving positions are accessible.

Nevertheless, the dog has to access one of the driving positions on the other axis as the first coordinate of the sheep is now identical to the corresponding target coordinate. Thus, due to Assumption 4, the dog can walk around the viewing range to correct the last coordinate. □

As next step, we begin the detailed analysis with Lemma 3 that provides an upper bound on the number of steps that are needed to solve a given Shepherding instance.

5.3.2 Upper Bound for the Solution Length

As for Lemma 2, we assume that all of the assumptions mentioned before hold and thus, the sheep has a distance of at most $r_{sheep}+1$ to every border of the environment.

Lemma 3 (Upper Bound for Detached Sheep). *Any legal instance* $(w, h, r_{sheep}, \infty, \{target\}, \varnothing, \{dog\}, \{sheep\})$ *of* Shepherding$(1, 1)$ *that fulfills Assumptions 1–5 can be solved with at most*

$$\mathrm{d}_M(dog, sheep) + 4(r_{sheep} + 1) + \mathrm{d}_M(sheep, target)$$

steps using lines 5–9 of Algorithm 3.

Proof. Due to Assumption 5, the sheep has a distance of $r_{sheep}+1$ to any border and thus, the dog can reach at least one allowed driving position as defined in Definition 16.

Let $dp^\star \in \mathcal{ADP}$ be the nearest such position. From Proposition 4 on page 78 we know that the distance from position p_1 to any position on the circumference of a circle with radius r around p_2 can be reached with at most

$$\mathrm{d}_M(p_1, p_2) - r + \frac{1}{2}C(r)$$

steps without coming closer than r to p_2. As all driving positions dp_i have distance $r_{sheep} + 1$ to the sheep, the distances of the dog to any (allowed) driving position $dp_i \in \mathcal{ADP}$ is given by

$$\mathrm{d}_M(dog, dp_i) \leq \mathrm{d}_M(dog, sheep) - (r_{sheep} + 1) + \frac{1}{2}C(r_{sheep} + 1)\,.$$

Now, the dog can correct one of the sheep's coordinates by moving from $dp^\star$ towards the sheep until the sheep's currently corrected coordinate equals the according target coordinate. If the target is not reached, the last coordinate has to be corrected and only one allowed driving position remains. As this driving position is on the other axis, it can be reached by following the sheep's viewing range's circumference for $\frac{1}{4}C(r_{sheep}+1)$ steps (for an example see Figure 5.5(c)). Finally, the current coordinate is corrected as described before until the sheep reaches the target.

Now, we give the number of steps that are needed:

$$\begin{aligned}
&\mathrm{d}_M(dog, sheep) - (r_{sheep} + 1) + \tfrac{1}{2}C(r_{sheep} + 1) + \tfrac{1}{4}C(r_{sheep} + 1) + \mathrm{d}_M(sheep, target)\\
&= \mathrm{d}_M(dog, sheep) - (r_{sheep} + 1) + \tfrac{5}{8} \cdot 8(r_{sheep} + 1) + \mathrm{d}_M(sheep, target)\\
&= \mathrm{d}_M(dog, sheep) + 4(r_{sheep} + 1) + \mathrm{d}_M(sheep, target)
\end{aligned}$$

Note, that $\mathrm{d}_M(sheep, target)$ covers all corrections of the sheep's coordinates (i.e. the number of all steps that the sheep performs until it reaches the target). □

We now show, how Assumption 5 for Lemma 2 and Lemma 3 can be established, i.e. how many steps are needed to free the sheep from the border cells. For the sake of argumentation, we assume, that the dog is directly outside the viewing range of the sheep.

Lemma 4 (Number of Steps to Detach the Sheep). *In any admissible instance* $(w, h, r_{sheep}, \infty, \{target\}, \varnothing, \{dog\}, \{sheep\})$ *of* Shepherding$(1, 1)$ *with the sheep closer than* $r_{sheep} + 1$ *to at least one border but having Assumptions 1–4 intact, and* $\mathrm{d}_M(sheep, dog) = r_{sheep} + 1$ *the sheep can be moved with at most*

$$C(r_{sheep} + 1) = 8(r_{sheep} + 1)$$

steps such that the sheep has a distance of at least $r_{sheep} + 1$ *to all borders (i.e. Assumption 5 holds afterwards).*

Proof. Due to $\mathrm{d}_M(sheep, dog) = r_{sheep} + 1$ the dog is directly at the sheep's viewing range and by the formula for the length of the circumference of circles in Manhattan geometry given in Proposition 2 on page 76 the dog can circumnavigate the sheep and return to its initial position in

$$C(r_{sheep} + 1) = 8(r_{sheep} + 1)$$

steps. As the dog walks outside the viewing range's circumference with a distance of $r_{sheep} + 1$ to the sheep, the sheep does not move.

To see that the sheep can be moved from the wall by this maneuver, consider the situation depicted in Figure 5.9: If the dog approaches the sheep's viewing range at positions different from the corners the sheep flees—as described in the sheep behavior in Section 4.5—to the right (relative to the dog). Thus, the dog can go clockwise between the border and sheep to push the sheep away from the wall.

In the worst case, i.e. if the sheep is in a corner (cf. Figure 5.11) it needs a complete circumnavigation and $C(r_{sheep} + 1) = 8(r_{sheep} + 1)$ steps are needed. □

Although Lemma 4 only considered the case where the dog is directly at the viewing range, the bound holds for the general case as well: As the dog has to move to the viewing range only once, the number of steps are already covered by Lemma 3. If line 3 of our *GCC* approach in Algorithm 3 has to be executed, the dog is directly at the viewing range afterward and thus, the number of steps to get to the driving position in lines 5–9 is accordingly smaller.

Corollary 8 (Upper Bound for Shepherding Instances). *The upper bound, i.e. the maximal number of steps needed to solve any instance* $(w, h, r_{sheep}, \infty, \{target\}, \varnothing, \{dog\}, \{sheep\})$ *of* Shepherding$(1, 1)$ *that fulfills the given assumptions is*

$$\mathrm{d}_M(dog, sheep) + 12(r_{sheep} + 1) + \mathrm{d}_M(sheep, target)\,.$$

Proof. From Lemma 3 we know that at most $\mathrm{d}_M(dog, sheep) + 8(r_{sheep} + 1) + \mathrm{d}_M(sheep, target)$ steps are needed to drive the sheep to the target if Assumption 5 holds (i.e. all corners of the sheep's viewing range are accessible). Otherwise, due to Lemma 4 $8(r_{sheep} + 1)$ steps are needed to establish the accessibility of the corners. Thus, any instance $(w, h, r_{sheep}, \infty, \{target\}, \varnothing, \{dog\}, \{sheep\})$ of Shepherding$(1, 1)$ can be solved with at most $\mathrm{d}_M(dog, sheep) + 12(r_{sheep} + 1) + \mathrm{d}_M(sheep, target)$ steps using *GCC* (Algorithm 3). □

The bound given in Corollary 8 is the worst case for all possible (allowed) instances of Shepherding$(1, 1)$. With a more detailed treatment of special situations (i.e. situations in which the sheep is close to a corner of the environment), one could e.g. formulate a more narrow upper bound for these cases. Nevertheless, we will see in the following that the given bound is pretty close.

Finally, in Example 3 we describe instances in which *GCC* (Algorithm 3) needs a number of steps equal to the upper bound proven before:

Example 3 (Worst Case Situation for *GCC*). *The worst case for* GCC *(cf. Algorithm 3) are instances, in which*

1. *the target is located at one border and has a distance of* r_{sheep} *to another border*
2. *the sheep is positioned such that it has a distance of* $r_{sheep} - 1$ *to the same borders as in Item 1*
3. *the dog is positioned outside the viewing range*

The instances given in Example 3 are expensive as the sheep has to be driven away from the target before it can be driven towards the target (cf. Figure 5.12). Thus, the viewing range of the sheep has to be circumnavigated completely before its coordinates can be corrected.

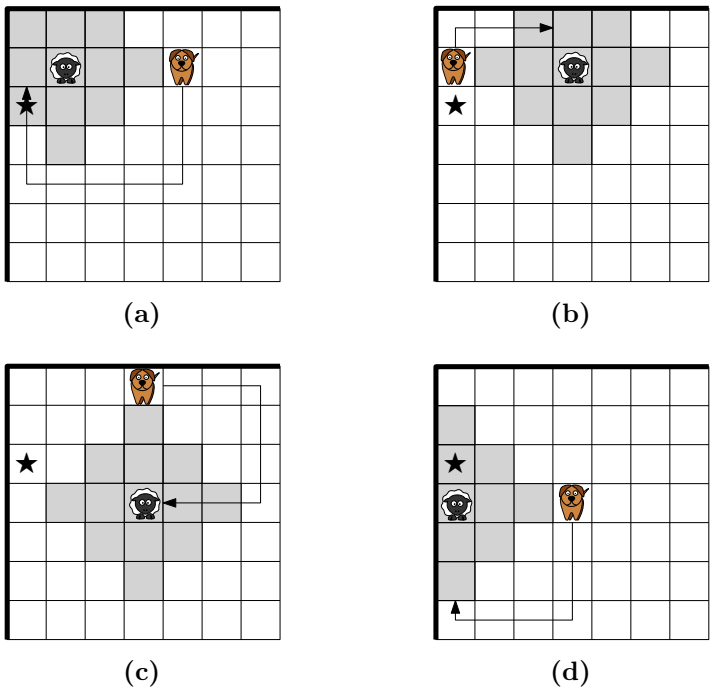


Figure 5.12: Explanatory execution of *GCC* for a worst case situation with $r_{sheep} = 2$ and the target positioned at ★.

5.3.3 Length of Any Optimal Solution

First we give a *lower bound* on the number of steps of an optimal solution for any instance with one sheep, one dog, and no obstacles. The lower bound is the minimal number of steps, in which a setting could potentially be solved. For a given setting, even the *optimal* solution might of course need more steps than the lower bound for the very same instance; it only means that there cannot be a solution with fewer steps than stated by the lower bound.

Lemma 5 (Lower Bound for Shepherding Instances). *The minimal length of any optimal solution sequence for any legal instance* $(w, h, r_{sheep}, \infty, \{target\}, \varnothing, \{dog\}, \{sheep\})$ *of* Shepherding$(1, 1)$ *is*

$$l_{min} = \mathrm{d}_M(sheep, target) + \mathrm{d}_M(dog, sheep) - (r_{sheep} + 1)\,.$$

Proof. The sheep needs $\mathrm{d}_M(sheep, target)$ steps to move from its initial position to the target and it only moves if a dog is in its viewing range. By Proposition 3 on page 77 $\mathrm{d}_M(dog, sheep) - (r_{sheep} + 1)$ steps are needed to get the dog in distance $r_{sheep} + 1$ to the sheep such that it can be controlled. Thus, l_{min} is the minimal number of steps needed to drive the sheep to the target. □

The bound in Lemma 5 is tight e.g. for instances in which one coordinate of sheep and target are equal and the dog is "behind" the sheep relative to the target as can be seen in Figure 5.13.

With this value for the lower bound we can now conclude our analysis with our main result, i.e. bounds on the number of steps for any optimal solution based on the

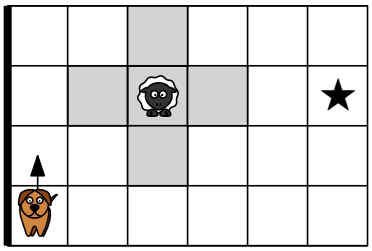


Figure 5.13: Example in which the sheep can be driven with the minimal number of steps given in Lemma 5.

lower bound established in Lemma 5 and the upper bound of *GCC* (Algorithm 3) given in Corollary 8.

Theorem 1 (Length of Any Optimal Solution). *The length l_{opt} of any optimal solution for any instance $(w, h, r_{sheep}, \infty, \{target\}, \varnothing, \{dog\}, \{sheep\})$ of* Shepherding*(1,1) is*

$$l_{min} \leq l_{opt} \leq l_{min} + 11(r_{sheep} + 1)$$

and the two bounds differ by $11(r_{sheep} + 1)$.

Proof. From Corollary 8 we have the upper bound of

$$\mathrm{d}_M(dog, sheep) + 12(r_{sheep} + 1) + \mathrm{d}_M(sheep, target)$$

and subtracting the lower bound

$$\mathrm{d}_M(sheep, target) + \mathrm{d}_M(dog, sheep) - r_{sheep} - 1$$

given in Lemma 5, leads to the stated boundaries. □

Theorem 1 states that *GCC* computes a solution that needs at most $11(r_{sheep}+1)$ steps more than any optimal algorithm. This value covers the circumnavigation of the sheep to make all corners of the viewing range accessible as well as the repositioning of the dog in case both coordinates have to be corrected: $11(r_{sheep}+1) = C(r_{sheep} + 1) + \frac{1}{2}C(r_{sheep} + 1) - (r_{sheep} + 1)$ where the last subtrahend results from the fact that the dog only has to move to the viewing range.

5.3.4 Computational Complexity of *GCC*

After having analyzed the upper and lower bounds of Shepherding$(1, 1)$ instances in terms of steps needed, we now turn our attention to the computational complexity of the *GCC* approach.

The first line of *GCC* in Algorithm 3 is only necessary if more than one target is present. Then, for each target in the set of targets $\mathcal{G}$ the distance has to be computed which each takes $O(1)$ and resulting in $O(|\mathcal{G}|)$.

Checking if all four allowed driving positions are accessible can be done in $O(1)$: For each driving position *dp* as defined in Definition 16 it has to be checked, if *dp* is behind the sheep in relation to the target to determine if *dp* is allowed. This can be done with three distance computations per driving position as given in Definition 15.

Additionally it only has to be assured that the allowed positions are inside the boundaries of the environment.

If the sheep has to be detached from the border (i.e. if line 3 of *GCC* in Algorithm 3 has to be executed), this can be done in $O(r_{sheep})$ due to Lemma 4.

As argued above, the computation of $\mathcal{ADP}$ can be done in $O(1)$ and the check whether the sheep is in the target is also possible in constant time. Thus, the time spend in the loop is bounded by the number of steps $\mathrm{d}_M(dog, sheep) + 4(r_{sheep} + 1) + \mathrm{d}_M(sheep, target)$ as given in Lemma 3.

Summing up, the total computational complexity of *GCC* is $O(|\mathcal{G}|+w+h+r_{sheep})$. Generally (i.e. in settings with several targets), the size of $\mathcal{G}$ is bounded by the number of sheep m. Thus, the complexity is given by $O(w + h + r_{sheep})$.

The storage needed for *GCC* is constant as only the position of the sheep as well as the position of the dog have do be provided.

Corollary 9 (Complexity of the *GCC* approach). GCC *can solve any instance of* Shepherding$(1, 1)$ *in time* $O(w + h + r_{sheep})$ *and is thus linear in the length of the solution.*

5.4 Conclusion

In this chapter we proved close upper and lower bounds on the optimal solution and showed that these upper and lower bounds differ by a term linear in the viewing range of the sheep. This difference is caused by the possibly necessary circumnavigation of the sheep's viewing range without disturbance (cf. Figure 5.5(c)). In order to derive aforementioned bounds, we geometrically analyzed the strategy followed by the *GCC* approach with the results presented in the beginning of this chapter.

We introduced the Greedy Coordinate Correction (*GCC*) approached, a greedy algorithm that solves Shepherding$(1, 1)$-instances within these bounds. In Section 5.3.4 we showed that the computational complexity of *GCC* is linear and, more precisely, that the runtime only depends on the length of the solution. Additionally, the algorithm only needs a constant amount of storage. This result is especially interesting considering the state-space complexity of the Shepherding task as analyzed in Section 4.6.

The performance of *GCC* as well as the tightness of the bounds could be further improved by taking into account special cases as e.g. presented in Example 3 and e.g. derive separate bounds for situations in which the sheep is close to the border in contrast to situations in which the sheep is remote from the border. Nevertheless, this is dispensable as the bounds are already sufficiently close and the overhead would be disproportionate.

6
Learning Shepherding Behavior

Coming back to the initially posed question of whether agents can *learn* shepherding behaviors and how good the learned behavior actually is, we here model the SHEPHERDING task as reinforcement learning task.

The *GCC* algorithm described before requires domain knowledge as the approach needs to know how the sheep moves and from which positions the sheep can be controlled safely. Thus, we investigate how learning can be used to supersede this requirement and to possibly find even more effective behaviors.

We formulate the SHEPHERDING task as RL task by giving the following MDP $\mathrm{M} = (S, A, \mathrm{T}, r)$:

Transition Function The transition function T models the behavior of the environment described in the description given in Section 4.4.

Reward A reward of 0 is given to the shepherding agent for the action that pushed the sheep to the target while every other step is punished with -1.

Action Set The action set

$$A := \{step_{east}, step_{north}, step_{west}, step_{south}, stand\}$$

for the shepherding agent is also dictated by the task definition from Chapter 4 where each step takes the agent to an adjacent cell.

State Space The state description consists of the exact positions of the sheep and the shepherding agent (cf. Definition 3):

$$S := \{(d_x, d_y, s_x, s_y) \mid 0 \leq d_x, s_x < w \text{ and } 0 \leq d_y, s_y < h\}\,.$$

As the SHEPHERDING task is episodic (i.e. it consists of several episodes that each ends when a goal state is reached), we use the approach described by Sutton and Barto (1998) to adjust the concept of MDP to handle such tasks: We introduce so called *absorbing* states that are entered whenever a goal state is reached and an episode has ended. Each absorbing state only transitions to itself with an neutral

reward of zero. After such an absorbing state is reached, the system is reset to a new start state.

As described in Section 2.2, the transition function and the reward function are unknown to the reinforcement learning agent. In fact, the agent does not even know what exactly it should do. The only information "known" to the agent is the fact that it has to maximize its rewards.

The analysis of the complexity of the Shepherding task in Section 4.6 revealed that the state spaces are huge. Although reinforcement learning has many benefits and is well suited for learning such behaviors, the curse of dimensionality becomes challenging. Thus, in the following chapters, we present two approaches that help to deal with large state spaces: First, we introduce the Growing Neural Gas Q-Learning (*GNG-Q*) that aggregates neighboring states that can be treated equally. The second approach is the Interpolating Growing Neural Gas Q-Learning (*I-GNG-Q*) that approximates the value function of the reinforcement learning agent.

With the above model, any reinforcement learning algorithm can be employed to learn the desired behavior. In this thesis, we use Q-Learning (cf. (Watkins, 1989)) as well as *GNG-Q* and *I-GNG-Q* (described in Chapter 7 and Chapter 8) for this purpose. Finally, we compare the behavior computed by the *GCC* algorithm to those learned strategies in Section 10.7.

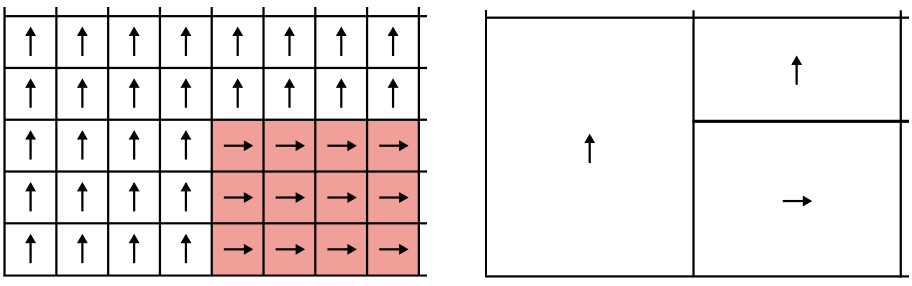
7

Adaptive State Aggregation

In this chapter, we consider the automatic aggregation of states to build an abstract state space. Such aggregations are useful for both discrete as well as for continuous state signals: In the first setting, several states are combined to form one *abstract state* (cf. Figure 7.1) while in the second setting some kind of discretization is inherently necessary to use tabular representations for the learned behavior in RL as already mentioned in Section 2.2. Of course, one can think of the continuous case as a discrete case with an infinitely small resolution.

The creation of useful aggregations is hard to accomplish: On the one hand, it should be as compact as possible to reduce the needed storage and the required learning. On the other hand it has to be fine enough to cover all relevant information of the original state space which may again result in a large abstract state space. Indeed, Even-Dar and Mansour (2003b) showed that the computation of a minimal aggregation is NP-hard.

There are severe problems with large (abstract) state spaces: First, the huge number of states results in an immense memory usage. Second and even worse is the fact that the higher the number of state-action pairs, the lower the probability



(a) Fine grained policy **(b)** Aggregated states

Figure 7.1: In (a) the policy is stored on a "per state" granularity while in (b) several states with the same action are combined to form blocks of states.

to visit a certain state again and thus most states appear to be unknown although possibly very similar states have been already seen. Thus, it is necessary to enable the transfer of past knowledge to similar states to speed up learning and to exploit past experience.

Possible approaches to aggregate states include tile coding (Sherstov and Stone, 2005; Whiteson et al., 2007; Lin and Wright, 2010), tree-based approaches (Chapman and Kaelbling, 1991; McCallum, 1995) or vector quantization (Lee and Lau, 2004). However, these approaches often assume domain knowledge or have computational issues as they have to solve the reinforcement task for different resolutions of the aggregation.

To us, the most promising approach is to use an adaptive approximation that is updated during learning to incorporate knowledge gained so far. Thus, we present *Growing Neural Gas Q-Learning* (*GNG-Q*) that combines Q-Learning (Watkins, 1989) and the unsupervised growing neural gas (Fritzke, 1994b) to automatically compute a state aggregation while learning the respective reinforcement learning task. This approach solves reinforcement learning tasks in continuous or discrete state spaces with a discrete time signal and discrete actions of the agent. *GNG-Q* respects similarity in the state and action space and its approximation is refined during learning based on information achieved from interaction with the environment.

To summarize, the *GNG-Q* approach

- computes an approximated state space using information achieved during learning without the need of labeled data.
- shares the advantage of off-policy reinforcement learning approaches, i.e. following a policy and learn about many (other) policies in parallel.
- respects similarity in state and action space.
- is an online learning approach, i.e. at any time during learning, the agent can rely on the knowledge gained so far.
- enables the reinforcement learning agent to learn the behavior and its representation in parallel.
- efficiently computes a useful policy in terms of training episodes and storage needed.
- is easy to implement.
- does not need a model of the considered reinforcement learning task (i.e. *GNG-Q* is *model free*).
- uses local criteria and computes a convenient discretization without domain knowledge.
- allows flexible and adaptive shapes of the discretization, which results in a compact representation.

The main results given here are the following:

- We review the theoretical model of state aggregation in Section 7.2.
- In Section 7.3, we introduce the main idea of the *GNG-Q* approach from a high level perspective.
- Section 7.4 shows how the growing neural gas approach can be extended to work as an adaptive state aggregation. Additionally, we discuss two different interpretations of the neighborhood connections in Section 7.5.

- In Section 7.6 the concept of regional states is introduced to allow a secure adaptation of the approximation although the movement of the agent introduces a moving target for the approximation.
- We introduce a new operation to refine the approximation that is based on the current error in Section 7.7. Furthermore, Section 7.8 provides criteria for the approximation's refinement and adaptation and argues how these criteria lead to an implicit stopping condition for adjustments.
- Section 7.11 analyzes the computational complexity of the proposed approach and points out limitations of state aggregating approximations.
- Eligibility traces are incorporated in Section 7.9 to speed up learning by better distributing Q-updates among earlier actions that lead to a particular reward.

Additionally, Section 7.1 motivates the use of an adaptive state aggregation algorithm while Section 7.12 concludes this chapter. Parts of this chapter are based on or are partly reformulated from (Baumann and Kleine Büning, 2011) and (Baumann et al., 2012).

7.1 Motivation

The ability to learn is a crucial requirement for agents to qualify as intelligent (Russell and Norvig, 2010). *Learning* agents can solve problems where pre-programmed strategies are hard or even impossible to create. Additionally, they are able to deal with changes in the environment and are able to improve their performance over time.

In reinforcement learning (RL), an agent is situated in an (possibly unknown) environment and has to learn a *policy* (i.e. a mapping from states of the environment to actions of the agent) to fulfill a given task[1]. This learning is carried out by interaction, i.e. the agent performs actions and observes the environment's responses in form of numerical rewards. Reinforcement learning is well applicable in tasks for which it is easier to describe *what* should be achieved than to state *how* it should be done.

In order to introduce and to theoretically investigate algorithms that compute optimal (or near optimal) solutions for such RL tasks it is usually assumed to store the learned behavior for each state-action pair in tabular representations. Thus, those approaches are often not directly applicable to difficult tasks with large or continuous state spaces, as they tend to appear in reality. To apply such algorithms in continuous state spaces, some kind of *discretization*[2] or some other approximation method has to be used to allow the agent to store its knowledge[3]. Often, a discretization sufficiently fine to cover all relevant information results in an extremely large state space, again.

[1] For a more comprehensive introduction to RL we refer to Section 2.2 and the references therein.

[2] A discretization transforms continuous values into discrete values e.g. by partitioning a given range into a small set of intervals (Kotsiantis and Kanellopoulos, 2006).

[3] In Chapter 8 we consider an approach that directly approximates the RL value function and is thus directly applicable to RL tasks with continuous state signals.

7.1.1 Challenges of Large State Spaces

Confronted with such large state spaces, three problems arise: The first problem was termed "needle-in-a-haystack problem" (van Otterlo, 2009) and deals with the issue of finding states (e.g. terminal states) that provide feedback to guide the agent that initially only "stumbles" randomly through the environment. Since this problem can be dealt with e.g. by the use of sensible exploration strategies (Thrun, 1992), reward shaping (Ng et al., 1999), or offering guidance (Driessens and Dzeroski, 2004), we here strive to tackle the following two major problems: The curse of dimensionality (Bellman, 1957) (i.e. the search space grows exponentially in the number of states and actions) induces high memory requirements to store the learned behavior and hinders a performant learning even for algorithms with runtimes linear in the number of states. Additionally, and even worse, the large amount of state-action pairs complicates becoming familiar with each possible state in order to act reasonably: Usually, algorithms that compute value functions or policies need tremendously many iterations through the state and action space in order to converge or to even derive a useful solution.

With this argument, Sutton and Barto motivate the use of *generalization*:

> "In many tasks to which we would like to apply reinforcement learning, most states encountered will never have been experienced exactly before. This will almost always be the case when the state or action spaces include continuous variables. The problem is not just the memory needed for large tables, but the time and data needed to fill them accurately. In other words, the key issue is that of generalization. How can experience with a limited subset of the state space be usefully generalized to produce a good approximation over a much larger subset?" (Sutton and Barto, 1998)

In other words, the higher the number of state-action pairs, the lower the probability to visit a certain state again and thus most states appear to be unknown although possibly very similar states have been already seen. Thus, it is necessary to enable the transfer of past knowledge to new situations to speed up learning and to reuse past experience. This ability to generalize knowledge is especially valuable in online reinforcement learning, i.e. if the agent has to fulfill tasks interleaved with its learning (Buşoniu, 2008).

7.1.2 Generalization as a Means of Dealing with Large State Spaces

Basically, two different approaches exist: state-space abstractions and function approximations. In this chapter we present an adaptive approach for state aggregation to automatically compute a compact representation of the original state space while in Chapter 8 we adapt some of the ideas employed here to create an adaptive function approximation approach.

The core idea of state-space abstraction (or state aggregation) is to combine several equivalent or at least similar[4] states of the original Markov decision process (MDP) to form a smaller abstract MDP. In this abstract MDP several states of the original MDP are represented by one abstract state and due to the smaller number of states the learning agent usually finds good solutions much faster. Additionally, this

[4] We will later discuss what exactly qualifies states to be combined.

approach offers generalization: All states of the original MDP that are combined in the same abstract state are treated equally. Thus, the agent can transfer knowledge to unseen but similar states.

Such abstractions usually introduce a trade off between sample complexity and computational complexity (Kakade, 2003): The sample complexity of a reinforcement learning algorithm is the amount of experience the agent has to obtain in order to learn a (near)optimal policy. Basically, the general goal of approximation schemes is to reduce the sample complexity without increasing the computational complexity—i.e. the computational costs of creating, maintaining, and using the approximation—too much.

Spatial aggregation—i.e. aggregations that at least to some extend respect similarity between states—are especially effective if the environment includes areas in which the same action is useful. An example for such aggregations can be seen in Figure 7.1.

7.1.3 Finding the Right Generalization

The goal of state-space abstraction is to compute an optimal aggregation, i.e. an aggregation that has a minimal number of states but still allows to learn a policy on the abstract MDP that leads to the optimal behavior in the original MDP. Note, that both criteria have to be respected: Of course, an aggregation that only consists of one state would be very compact but would in general never allow to learn any useful behavior.

Additionally, it is important to choose the right method for partitioning the state space: Using a regular partitioning approach is easy to implement but often results in a too large number of abstract states (Bertsekas and Tsitsiklis, 1996). Even if the environment allows a rectangular partition, the resolution has to be chosen appropriately.

An optimal state-space aggregation could be constructed *a posteriori* by first learning (an approximation of) the optimal Q-function and aggregate similar states that share the same optimal action afterwards. Unfortunately, this approach neither reduces the storage needed nor offers generalization during *learning*, since estimations for each state-action pair have to be computed. In fact, finding a minimal state aggregation is NP-hard (Even-Dar and Mansour, 2003b).

On the other hand, the *manual* creation of a state-space aggregation *a priori* requires immense domain knowledge; otherwise, it may be

- too coarse to capture all relevant features of the environment, or
- too fine and will thus result in a very large abstract state space.

Both cases would lead to unsatisfactory or ineffective learning. See e.g. Figure 7.2(a) where the resolution allows the agent to learn a useful policy while the resolution in Figure 7.2(b) does not: The size of the grid is too coarse to represent all necessary features of the environment (i.e. the cell that contains the target also includes states that are not part of the target) and thus, the agent would have problems to identify the real goal. On the other hand, a too fine approximation as e.g. in Figure 7.2(c) would allow for correct learning but this learning would be rather slow due to the large state space.

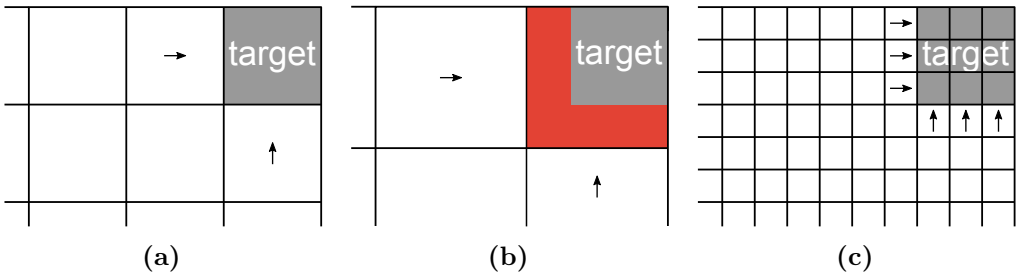


Figure 7.2: Different discretizations for a task in which the agent has to find a target: The resolution in (a) is correct as it allows the agent to capture all necessary information. The one in (b) is too coarse and would hinder the agent to learn an appropriate behavior since the agent would not be able to select one optimal action in the red area around the target. Finally, the resolution in (c) is too fine which would result in long learning times.

On the other hand, if one is in the possession of domain knowledge, then this knowledge can be used to greatly improve the performance of a state-space abstraction: The first possibility of incorporating prior knowledge about the reinforcement learning task at hand is to choose features (i.e. how the state signal is presented to the learning agent) that are appropriate for the given task (Sutton and Barto, 1998). An alternative is to refine the resolution in important parts of the approximation (Santamária et al., 1997).

In the general (and the most difficult) case the designer of an approximation scheme has no prior knowledge of the task's properties and can thus not rely on this helpful information. When building a general purpose aggregation approach (i.e. an approach that finds (sub-)optimal solutions without knowing the reinforcement learning task beforehand) the easiest way to derive an appropriate resolution would be to try several granularities and select the most performant approximation (trial and error). Obviously, this would result in many iterations as there exists several "dimensions" that can be changed:

Uniformity of Shape: i.e. whether or not all abstract states have the same geometric shape

Uniformity of Size: "all abstract states have the same size" vs. "the size of abstract states differs"[5]

Temporally Constancy: if the layout of the approximation is fixed throughout the execution or if it is subjected to change

This thesis presents an adaptive method to derive an effective resolution of the state-space aggregation that automatically detects areas in which the granularity of the approximation has to be refined. Thus, this approach computes non-uniform shaped and differently sized abstract states that change over time *without* domain knowledge.

[5] With the size of an abstract state $\widehat{\mathbf{s}}$ we denote the number of states of the original MDP that are mapped to $\widehat{\mathbf{s}}$.

7.1.4 Benefits of Adaptive Generalization

Humans are usually more skilled in abstracting knowledge and in recognizing structures or concepts in their perception. In fact, humans often excel in fulfilling complex tasks by ignoring irrelevant details (Goldstone and Barsalou, 1998) while computers struggle due to the overwhelming amount of data.

For an agent, deciding which suitable scheme is most appropriate for a given task or how to set its parameters is a difficult open problem (Ponsen et al., 2009). Often, these decisions are made by humans which biases the learning to a particular task and limits the reusability of this distinct combination. If the agent is presented with the "tools" to build and use an approximation, it is able to create an approximation that is highly specialized to the environment it is currently confronted with. Simultaneously, the agent is able to adjust to new challenges if necessary.

An adaptive approximation that is refined and adjusted by the agent without external control is highly valuable as this capability equips the agent with autonomy. Thus, the agent can adapt to changes in its task or the environment with no or at most minimal human supervision. Additionally, if agents are capable to "find" their own abstraction their learning will not be affected negatively by superficial intervention. It has been argued that the capability to autonomously create abstractions and to use generalization to solve tasks is the essence of intelligence (Brooks, 1991).

The gradual refinement allows an efficient usage of knowledge: In the beginning the learning affects large portions of the state space as the resolution of the approximation is very coarse and only consists of few abstract states. Each newly created abstract state is a local refinement and the behavior for the new state is derived from the split state. Additionally, a broader generalization in the beginning is valuable to allow quick learning of the rough idea while later a much finer generalization allows to discover all the details. Sherstov and Stone (2005) argued that the best results are to be expected if the generalization is gradually reduced.

Otterlo (2009) claims that automatically finding a suitable approximation is hard to achieve and that the agent should be supported by as much prior knowledge as possible. Although we completely agree that this task is very complicated, we note that easing the approximation process by adding domain knowledge undermines the agent's autonomy.

7.1.5 Learning Goal in Adaptive Generalization

The overall goal of the *GNG-Q* approach is to learn a policy defined on the abstract state space that is (near-)optimal in the original MDP. Particularly, the learning of the approximation (which results in the abstract state space) should be done concurrently to the agent's interaction with the environment (i.e. the original MDP).

While adapting the approximation in parallel with the learning process is highly beneficial, one additional challenge is introduced: Reinforcement learning without approximation is *one* learning task that has a (set of equally good) solution(s), namely the optimal policy(ies) (Sutton and Barto, 1998). Adding approximation to reinforcement learning introduces a new learning task, i.e. the interleaved learning of the knowledge representation which can e.g. be seen in the schematic depiction of our approach in Figure 7.3.

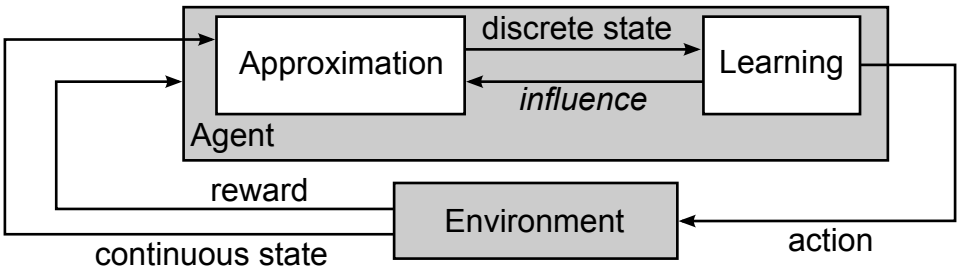


Figure 7.3: Abstract framework of our approach.

Often, learning approaches (supervised as well as unsupervised as e.g. the growing neural gas approach) assume a fixed set of training samples that is used to train the approximation by probably numerous iterations over this set until a desired solution quality is achieved. This learning then only affects the task of estimating the parameters for this particular method.

When using approximation methods *inside* reinforcement learning, two major challenges appear:

1. The policy depends on the value function that is learned by the reinforcement learning agent. As such, the target values are not known beforehand but computed and updated during the interaction with the environment and thus, the learning goal (i.e. the value function) is non-stationary. This is a severe difference to the general assumption prevalent in supervised learning where "correct" answers are present for the training data.
2. Also, the learned policy is used by the agent to interact with the environment which influences the sampling of states to the agent. Thus, even the sampling is non-stationary as opposed by the existence of a fixed training set in supervised learning.

7.1.6 Core Idea of Our Approach

To us, the most promising approach is to use an adaptive approximation that is updated during learning to incorporate knowledge gained so far. Figure 7.3 summarizes the key idea of our approach: In each learning step, the agent learns using the current state-space approximation. Simultaneously, the applied learning method influences the state-space approximation as knowledge gained during learning is used to find and refine too coarse areas. Since GNG computes a graph consisting of Voronoi centers (cf. Section 2.3), a nearest neighbor rule introduces generalization as every state of the original state space is treated equivalent to the most similar reference vector.

Our combination of Q-Learning and GNG offers many advantages: The approach operates online, does not need the MDP of the RL task, and is easy to implement. The usage of an unsupervised learning method abandons the requirement of experience tuples as e.g. in the work of Ernst et al. (2005) and is especially useful if the training samples are presented successively and not known in their entirety (van Otterlo, 2009). Furthermore, the GNG is insensitive against changes in its parameters (Heinke and Hamker, 1998) and is thus well applicable without deeper domain knowledge.

GNG-Q is able to efficiently compute a useful policy in parallel with a compact state-space approximation that respects similarity in state and action space. The goal is to find a partition of the state space in so-called *state regions* so that each region contains states which are similar and can be treated equally. In most reinforcement learning tasks we can assume that there are areas of spatially close states that require the same behavior (cf. Figure 7.4 on page 105 where a robot has stored the shortest path to a target). Neither the number nor the positioning of the reference vectors has to be predetermined but is instead derived from interaction with the environment: The initially coarse approximation is locally refined in areas that need a finer resolution. Thus, there is no need to decide on the granularity of approximation beforehand.

After learning, the policy can be stored very efficiently as only the Voronoi centers of the approximation and the associated action values are needed. During learning, the additionally stored information is linear in the number of centers. The mapping of one state of the original state space to its abstraction is realized by a nearest neighbor rule and is thus very fast and easy to implement.

7.1.7 Concluding Remarks

Practical applications of adaptive approximation methods are manifold.

For example behaviors for robots can be learned by simulation: An agent (representing the robot) learns in a simulated world (i.e. the environment) in order to protect the hardware of the real robot and to accelerate learning. This (initial) knowledge can then be implemented in the real robot and the adaptive approximation method may be used to adjust the behavior in order to compensate for changes in or unknown dynamics of the real environment (Kakade, 2003). Storage-efficient methods also are interesting for the application in (mobile) robots: As their storage capacity is usually limited, compact approximations are helpful to save expenses on hardware.

The fact that the approximation is adapted during learning allows the agent to use its generalization capabilities while interacting with the environment and additionally, the interleaved learning of behavior and its representation allows an efficient fitting to the learning task at hand.

7.2 Theoretical Model of State Space Abstraction

As pointed out earlier, large state spaces introduce severe issues and it is thus highly beneficial to introduce some kind of generalization (Sutton and Barto, 1998). Amongst many others (for detailed overviews see (van Otterlo, 2009) or (Buşoniu et al., 2011a)), one approach to deal with large state spaces is the use of *state-space abstractions*. Following (van Otterlo, 2009), we define an abstract state space as follows:

Definition 17 (State Space Abstraction). *Let* $\mathrm{M} = (S, A, \mathrm{T}, r)$ *be a deterministic Markov decision process as defined in Definition 1. We define the corresponding* abstract MDP $\widehat{\mathrm{M}} = (\widehat{S}, A, \mathrm{T}, r)$ *where* $\widehat{S}$ *is a partition of the actual state space* S *and usually* $|\widehat{S}| \ll |S|$ *holds. Each abstract state* $\widehat{\mathbf{s}} \in \widehat{S}$ *is defined as a set* $\widehat{\mathbf{s}} := \{s \mid \psi(s) =$

$\widehat{\mathbf{s}},\ s \in S\}$ *where the* abstraction function ψ *is a mapping* $\psi : S \to \widehat{S}$ *that maps each state of* S *to one of the states of the abstract state space* $\widehat{S}$*. Thus,* ψ *provides a partition of* S *with* $\bigcup_{\widehat{\mathbf{s}} \in \widehat{S}} \widehat{\mathbf{s}} = S$ *and* $\widehat{\mathbf{s}}_i \cap \widehat{\mathbf{s}}_j = \varnothing,\ \forall\ \widehat{\mathbf{s}}_i \neq \widehat{\mathbf{s}}_j \in \widehat{S}$*.*

The value functions in RL for an abstract MDP $\widehat{\mathrm{M}}$ can be learned from interactions with the original MDP M: The agent observes a state $s_t \in S$ and performs action a_t that takes it to the subsequent state $s_{t+1} = \mathrm{T}(s_t, a_t)$ and results in a reward $r(s_t, a_t)$. This information can be used e.g. in a Q-Learning update for the abstract MDP (van Otterlo, 2009):

$$\begin{aligned}\widehat{Q}_{t+1}(\psi(s_t), a_t) := (1 - \alpha_t)\widehat{Q}_t(\psi(s_t), a_t) \\ + \alpha_t\Big[r(s_t, a_t) + \gamma \max_{a' \in A} \widehat{Q}_t(\psi(s_{t+1}), a')\Big]\end{aligned} \tag{7.1}$$

Note, that the update for one abstract state $\widehat{\mathbf{s}}$ affects all states $s \in S$ that are abstracted to $\widehat{\mathbf{s}}$, i.e. all states s for which $\psi(s) = \widehat{\mathbf{s}}$ hold. This is a major advantage as one update affects several states and each (maybe unseen) state is treated as any other state abstracted to the same abstract state.

7.3 General Approach

This section presents the general approach as well as the core idea of *GNG-Q*, which is detailed in the following sections. The intention of *GNG-Q* is to learn the behavior and its representation in parallel using a combination of Q-Learning (cf. Section 2.2) and the unsupervised growing neural gas (GNG) vector quantizer (cf. Section 2.3). Our approach assumes that similar states need similar behavior—an assumption that is often true in large portions of reinforcement learning state spaces. If the state space does not contain similar states that need similar behavior one region per state is needed which would be identical to storing the complete value function.

We consider units called *neurons* as described in Section 2.3 that each have a reference vector that can be seen as position, an index, and a variable for the accumulated error associated with this unit. This error contains information gathered by monitoring the agent's policy. Additionally, each neuron is assigned a *prototype Q-vector* that comprises all updates that were performed in the respective region by Q-Learning. In fact, the prototype Q-vector can be thought of as Q-vector for the abstract state created by its corresponding region. The nearest neuron is the neuron with the most similar reference vector to a given state and, similarly the second nearest neuron has the second most similar reference vector to the same given state. For clarity, we here assume a state space consisting of real-valued vectors, otherwise, the methods to measure the similarity and to adapt neurons to states have to be altered.

Typically, reinforcement learning deals with single states, but to improve the efficiency of learning we search for groups of neighboring states in which the RL agent's behavior is similar. These groups are identified with the help of GNG that computes a Voronoi tessellation of the state space. Then, our approach treats all states in one region equally (i.e. all states in one region as defined in Section 2.3 are seen as one abstract state). The Q-function in our approach is defined as a mapping from those regions to the Q-vectors.

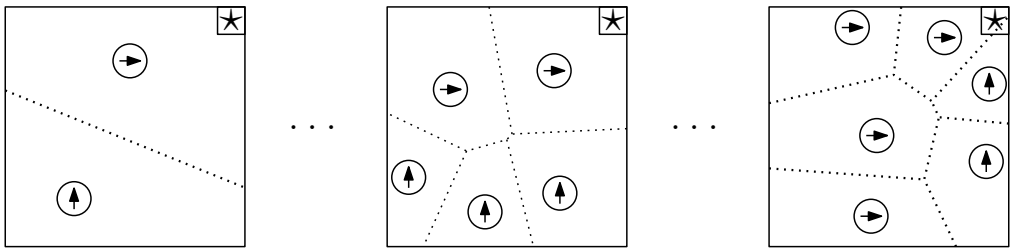


Figure 7.4: *GNG-Q* refines the approximation based on information gained during learning. The neurons (circles with arrows that depict the action with the maximal Q-value) are as described in Section 2.3, but additionally, each neuron has one Q-vector whose values are updated during learning. The dotted lines indicate the borders of the Voronoi regions for each layout.

The core idea of *GNG-Q* is as follows: As in the generic GNG approach, the approximation is initially very coarse and it refines regions that contain incompatible states. In each learning step in the reinforcement learning environment, the agent uses the current approximation to update its estimated policy using Q-Learning. Simultaneously, changes in the learned policy point out regions that have to be refined. Thus, an *abstract state space* is built by aggregating compatible states into so-called *state regions* which are represented by neurons. The size, the shape, as well as the positions of these state regions are adjusted based on the interaction during learning without knowing the environment in advance.

The goal is to partition the state space in regions of similar states such that all states in one region can be treated equally. Q-Learning updates its estimations of the Q-values iteratively which may influence the behavior in a larger area. Thus, it is favorable to adjust the approximation by inserting and moving neurons in parallel with changes in the policy defined by the estimated Q-values. In particular, *GNG-Q* uses those changes to determine areas of the approximation that are too coarse and need to be refined. As *GNG-Q* starts with a coarse approximation and regions that are not visited are removed, the learned layout of the abstract state space is very compact.

Figure 7.4 shows an example of how the state regions are adjusted during learning: *GNG-Q* starts with a very coarse approximation, initially consisting of two regions (i.e. the abstract state space consists of only two states). In each step the learner's estimation of the Q-values $\widehat{Q}$ is updated with the Q-Learning update rule based on the regions of the abstract state space's current layout. The learner's policy is monitored during learning and these information are then used to adjust the layout of the abstract state space. These adjustments (movement of neurons and refinement) carry on until a sufficiently fine approximation is found.

Thus, the abstraction function ψ from Section 7.2 and the agent's approximation $\widehat{Q}$ for the abstract MDP are learned in parallel: The abstraction function ψ is derived from the layout (i.e. the number of the neurons and their positions) and the nearest neighbor rule. At any time during learning, the agent can make use of the information and experience gathered so far although in most cases, the approximation of the state space as well as the learned Q-values need some time to evolve.

Figure 7.5 adapts the agent-environment cycle presented by Sutton and Barto

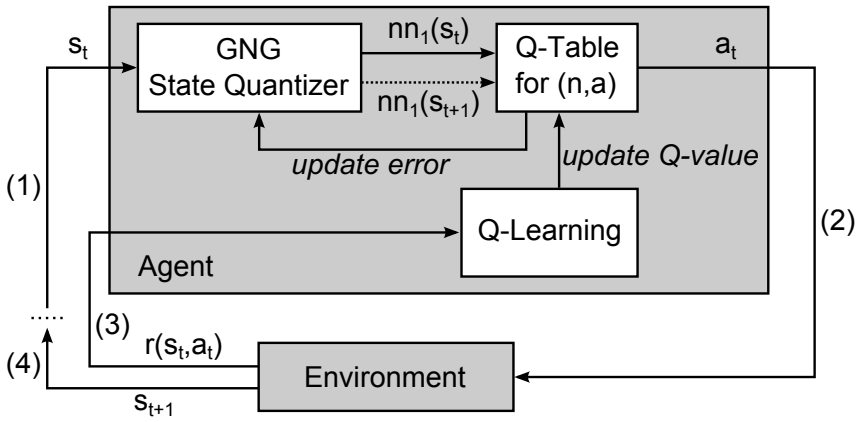


Figure 7.5: *GNG-Q* in Sutton and Barto's (1998) agent-environment interaction.

(1998): In each step, the agent perceives the current state of the environment $s_t \in S$ and uses the state-space approximation to compute an internal representation of the state (1). The nearest neuron $\text{nn}(s_t)$ to the present state s_t is used to perform a look up in the Q-table consisting of Q-values for every neuron-action pair (and thus for every Voronoi region). This results in the prototype Q-vector of the nearest neuron's region from which the action with the maximal Q-value can be derived (i.e. the agent can use the approximation to get a *maximal* or *greedy* policy by performing the action with maximal Q-value from the region of the nearest neuron to the current state). During learning, the agent may use some exploration strategy (e.g. ε-greedy) to determine the next action a_t. After performing (2) the action a_t, the environment responds (3) with a reward $r_t = r(s_t, a_t)$ that represents the immediate value of performing a_t in state s_t and simultaneously transitions (4) into a subsequent state s_{t+1}. While learning, the agent uses this reward to update its estimation of the Q-function at $\widehat{Q}(n_1, a_t)$ taking into account the estimate for the succeeding state s_{t+1} computed with the same approximation (depicted as dotted line in Figure 7.5). If this update leads to a change in the maximal policy, the error variable of n_1 is updated to indicate the possible need for refinement in this region.

Additionally to the nearest neurons to the current state s_t and the subsequent state s_{t+1}, the GNG component of our approach computes the second nearest neuron to the current state, too. This information is used to build neighborhood connections as described in Section 7.5. While the generic GNG approach updates the approximation (i.e. the positions of the neurons) after every presented training sample (in our case a sample would be the state s_t although the update takes place after performing the action a_t that leads to the succeeding state s_{t+1}), our approach updates the positions of the neurons *after* one complete reinforcement learning episode, i.e. after a goal state is reached[6]. To consider all states that have been visited during one episode, each neuron has an additional set to collect states that are visited during the current episode. After each episode, every neuron adapts to the centroid of all states that have been visited in its region (cf. Section 7.6). By doing so, the *GNG-Q* approach ensures that the approximation is stationary during each

[6] In non-episodic reinforcement learning tasks a comparable solution can be easily defined by e.g. using a fixed interval for the updates.

episode and only adjusts the approximation in between two succeeding episodes.

Thus, our approach computes an approximation that respects similarity in both the state and the action space in parallel with the learning of the behavior. The Q-function in our approach is defined over neurons and actions and can be learned with tabular Q-Learning using one entry for every neuron-action pair. All states in one state region are treated equally, i.e. they all share one prototype Q-vector. The *GNG-Q* approach leads to a piecewise constant approximation of the Q-function as all states in a region are treated equally. The nearest neighbor rule introduces generalization: Each neuron defines a region of states that are treated equally and an update of the Q-vector for one region affects all states in this region. Simultaneously, an unseen state can be handled as similar states seen before as it is treated as any other state in its region.

7.4 From States to State Regions

In this chapter, we present an approach to aggregate several states to an abstract state. This state aggregation is one possibility for function approximation in reinforcement learning. A very concise formulation of the goal for approximation in reinforcement learning was presented by Sutton and Barto (1998):

> "How can experience with a limited subset of the state space be usefully generalized to produce a good approximation over a much larger subset?"
>
> (Sutton and Barto, 1998)

Usually, reinforcement learning works with single *states*, but here, similar states are aggregated to so-called *state regions*. From the theoretical model presented in Section 7.2 we know that we need a function ψ that maps from the set of all possible states of the reinforcement learning task to a set of abstract states. The theoretical model, however, does not specify *how* to realize such a function. Clearly there are several possibilities including very rudimentary methods like a set approach: Each abstract state is modeled as a set for all states from the "real" state space and the ψ would reduce to the set membership relation $\in$. Using this naïve approach, the only improvement to a standard tabular approach would be that one Q-update would influence several states with (i.e. all states in the set for the current abstract state). Unfortunately, the amount of storage needed is identical to the amount needed without this kind of approximation. Additionally, this approach would neither be directly applicable to continuous reinforcement learning tasks nor exploiting the fact that reinforcement learning tasks often have neighboring states that require the same behavior.

We present an approach that allows to efficiently compute the abstraction function ψ and additionally, this function can be stored very compactly. To achieve this, we present two crucial requirements for aggregated states:

1. The states have to be similar regarding some similarity measure (e.g. constructed based on the Euclidean distance) in the state space.
2. The (currently) optimal behavior in those states has to be the same.

Figure 7.6 shows an example which emphasizes the importance of both requirements for aggregated states: The states s_1 and s_2 are obviously spatial close but they cannot

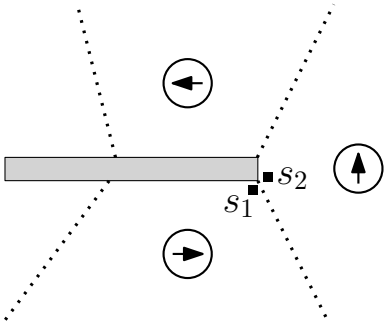


Figure 7.6: Example of state regions for a policy to walk around a wall

be aggregated to one abstract state as the needed behavior is different.

We employ the ideas described in Section 2.3 to create the state regions. *GNG-Q* uses neurons that are inserted and distributed in the state space[7]. The abstraction function ψ is thus defined by the set N_t of neurons present at time t and the nearest and second nearest neuron functions[8] that are derived from the nearest neighbor rule. As a reminder to Section 2.3, these functions determine the neuron $\mathrm{nn}(s)$ with the most and the neuron $\mathrm{nn}_2(s)$ with the second most similar reference vectors to a given state $s \in S$ (ties are broken by selecting the neuron with minimal index):

$$\mathrm{nn}(s) = \operatorname*{arg\,min}_{n \in N_t} \mathrm{d}(s, \vec{w}_n)$$
$$\mathrm{nn}_2(s) = \operatorname*{arg\,min}_{n \in N_t \setminus \{\mathrm{nn}(s)\}} \mathrm{d}(s, \vec{w}_n)$$

We use the neurons' reference vectors as representatives of the state regions[9] and adapt the Q-update from Equation (2.14) to define the reinforcement learning agent's estimation $\widehat{Q}$ over the neurons of the network and actions. We can then use a tabular representation for $\widehat{Q} : N_t \times A \to \mathbb{R}$.

All states in one state region are treated identically and they all share one *prototype* Q-vector that comprises all Q-updates performed in the respective region. This Q-update only depends on the Q-value of the region containing the current state s_t and the Q-value of the region containing the succeeding state s_{t+1}:

$$\begin{aligned}\widehat{Q}_{t+1}(\mathrm{nn}(s_t), a_t) := (1 - \alpha_t)\widehat{Q}_t(\mathrm{nn}(s_t), a_t) \\ + \alpha_t \big[r(s_t, a_t) + \gamma \max_{a' \in A} \widehat{Q}_t(\mathrm{nn}(s_{t+1}), a') \big]\end{aligned} \tag{7.2}$$

See Figure 7.7 where the agent performs action a_t in state s_t and transitions to state s_{t+1}. This transition leads to an update of $\widehat{Q}(\mathrm{nn}(s_t))$ that uses the estimate

[7] For now we just assume, that the neurons are placed *somehow* in the state space; in Section 7.6 and Section 7.7 we will describe how the neurons are repositioned as well as how the approximation is refined by adding new neurons.

[8] Since we made the assumption that we deal with states that are described by real-valued vectors we here use the same distance d as in Section 2.3. Thus, we use $\mathrm{d}(s, \vec{w}_n)$ to measure the distance between the vector describing state s and the reference vector of neuron n.

[9] Remember that the neurons' reference vectors can be used as codewords as described in Section 2.3.

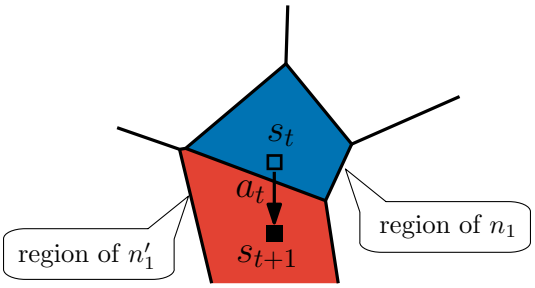


Figure 7.7: After performing action a_t in state s_t that is in the region of neuron n_1, the agent transitions to state s_{t+1}. The Q-update affects all states in the region of neuron n_1 and incorporates the maximal Q-value of the region that contains state s_{t+1} (i.e. defined by neuron n_1').

$\widehat{Q}(\text{nn}(s_{t+1}))$. Note, that this update influences the values of all states in the region of neuron $n_1 = \text{nn}(s_t)$ but no values outside this region. The aggregation creates a partition of the state space and thus leads to a piecewise constant approximation of Q and the generalization is achieved by the nearest neighbor rule as all states in one region are treated equally.

From the fact that all neurons in one region share a single prototype Q-vector it directly follows that the agent's policy also depends on the current state region: Thus, the *maximal policy* (or greedy policy) is defined as

$$\widehat{\pi}^\star(s_t) = \arg\max_a \widehat{Q}_t(n_1, a) \tag{7.3}$$

with $n_1 = \text{nn}(s_t)$ being the neuron with the most similar reference vector to s_t. This is the learner's approximation of the *optimal policy* as defined in Equation (2.10) that requires the actual, but usually unknown, optimal Q-function $\widehat{Q}^\star$. Thus, using in each region the action with highest Q-value results in a policy defined over state regions as depicted in Figure 7.4 or Figure 7.6. Of course, during learning it is useful to choose the actions incorporating some kind of exploration strategy as e.g. described in Section 2.2.8.

7.5 Neighborhood Connections

In this work we explore two different interpretations for the neighborhood connections of the GNG approach: First, they can be used identically to the intention in the generic GNG approach (i.e. a connection between two neurons is evidence that the two corresponding Voronoi were neighboring at the time of their creation). Second, we discuss how to use neighborhood connections to derive an abstraction of the transition function for the abstract MDP from Section 7.2.

Note, that the connections in our approach are not essential but merely used to gain insights in the layout of the state space. While in the generic GNG approach the connections are used to adjust the topological neighbors of a neuron as well, we here use the neighborhood connections to update the network's topology or to determine "dead" abstract states, i.e. regions that have not been visited for a long time.

7.5.1 Topological Connections

The first method that can be used to create neighborhood connections was introduced in (Baumann and Kleine Büning, 2011) and stems from the creation process in the generic GNG. A connection in this context states that the two connected neurons were the nearest and second nearest neurons to a given input at the time of its creation. Thus, this approach creates a network of neurons, whose connections give insights into the *topology* of the state space.

As in the generic GNG, we compute the nearest neuron n_1 and the second nearest neuron n_2 to the current input (in our case the current state s_t)

$$n_1 = \mathrm{nn}(s_t)$$
$$n_2 = \mathrm{nn}_2(s_t)$$

with the nearest and second nearest neighbor rules as described in Section 2.3. These neurons n_1, n_2 are connected with a neighborhood connection $\{n_1, n_2\}$ and are now *topological neighbors*.

With this method, clues about the topology of the state space can be derived: As two neurons n_1 and n_2 are connected if they were the nearest and second nearest neurons to a sample (in our case a state) and thus, the respective regions $\mathfrak{R}(n_1)$ and $\mathfrak{R}(n_2)$ are neighboring[10]. From this fact, we can infer that the abstract states represented by these particular neurons are *spatial* neighboring but we do not know, if there is an action with which the agent transitions from a state in region $\mathfrak{R}(n_1)$ to a state in region $\mathfrak{R}(n_2)$ (or vice versa). Nevertheless, one might want to analyze the Q-vectors of neighboring neurons in order to investigate whether there are areas in the state space that are spatially close but require different behavior. A clue for this might be found in severe differences in the maximal policy.

To keep the representation of the state space's topology up to date, each neighborhood connection is equipped with an age. If two neurons n_1, n_2 are the nearest and second nearest neurons but are already connected with a neighborhood connection the age of that existing connection is reset to zero. Furthermore, the ages of all connections emerging from the nearest neuron n_1 are increased.

Due to the adaptation and the refinement of the network's layout (or due to changes in the environment), some neighborhood connections may become outdated. This would result in high ages of the relevant connections as they would not be reset to zero for a given amount of iterations.

In the topological context, the removal of a neuron n results in a coarsening of the approximation: The states of the original MDP that fell into the region of n would afterwards be approximated by surrounding neurons. Nevertheless, it is more probable that no or only very few states will be affected by this; otherwise, the region would not have been removed.

7.5.2 Abstraction of Transition Function

The second approach for the creation of connections is more suitable for reinforcement learning as the quantization motive described before only covers a portion of what

[10] To be precise, the neurons' regions *were* neighboring at the time of the connection's creation.

is desirable in this particular kind of approximation (Baumann et al., 2012). Here, two neurons $n_1 = \text{nn}(s_t)$ and $n'_1 = \text{nn}(s_{t+1})$ are connected if an action performed in the region $\mathfrak{R}(n_1)$ of neuron n_1 resulted in a state in the region $\mathfrak{R}(n'_1)$ of neuron n'_1. Thus, the connections can be used to approximate the abstract transition function of the original MDP as each connection between two neurons n_1 and n'_1 implies that an action performed in a state of n_1's region ended in a state in the region of neuron n'_1 (or vice versa as the connections are undirected).

In each learning step, the nearest neuron to the current state s_t and the nearest neuron to the succeeding state s_{t+1} are determined:

$$n_1 = \text{nn}(s_t)$$
$$n'_1 = \text{nn}(s_{t+1})$$

and both neurons are connected with a *neighborhood connection* to become *topological neighbors*. Note, that in this approach loops (i.e. edges $\{n_u, n_v\}$ with $n_u = n_v$) may occur as the agent might transition between two different states that are in the same region. This cannot happen in the first approach described before or in the generic GNG approach as there the nearest and second nearest neuron to a given input is connected and the network always consists of at least two neurons.

To completely capture the abstract transition function, two extensions are needed:

1. For each action that can be performed, one unique connection has to be stored: If an action a performed in the current state s_t in $\mathfrak{R}(n_1)$ results in a succeeding state s_{t+1} in region $\mathfrak{R}(n'_1)$, the connection between n_1 and n'_1 has to be labeled with the performed action a.
2. The connections have to be oriented to consider the direction of the transition from region $\mathfrak{R}(n_1)$ to region $\mathfrak{R}(n'_1)$ and vice versa.

Thus, an abstract transition function $\widehat{\mathrm{T}} : \widehat{S} \times A \to \widehat{S}$ for the abstract MDP $\widehat{\mathrm{M}}$ can be created from the neighborhood connections.

These two extensions result in an upper bound of $|N_t| \cdot |A|$ connections for a deterministic MDP. In case of non-determinism in the abstract MDP (cf. Section 7.11.2 for an explanation of how non-determinism can occur even in deterministic environments), a neuron may have several emanating edges for one action. This results in an upper bound of $|A| \cdot |N_t| \cdot |N_t|$ connections for this scenario.

To identify outdated connections, each neighborhood connection is equipped with an age which is initialized with zero. If two neurons n_1, n'_1 are already connected with a neighborhood connection but are again designated to be connected, the age of that existing connection is reset to zero. Additionally, the ages of all connections emerging from the nearest neuron n_1 are increased. If the age of any connection exceeds age_{max}, the connection is removed as this is evidence, that this connection is outdated: Consider the latest age_{max} transitions (s, a, s') that involved n_1 or n'_1 (i.e. transitions for which s and/or s' was in $\mathfrak{R}(n_1)$ or $\mathfrak{R}(n'_1)$), then there was no action that led from a state in region $\mathfrak{R}(n_1)$ to a state in region $\mathfrak{R}(n'_1)$ (or vice versa). In other words: For the last age_{max} transitions (s, a, s') that involved at least one of the regions $\mathfrak{R}(n_1), \mathfrak{R}(n'_1)$ there was no transition such that the agent "traveled" from one of the two regions to the other. Thus it is save to assume that the approximation

changed in a way that no action leads from any state abstracted by n_u to any state abstracted by n_v (or vice versa).

If the deletion of connections results in isolated neurons, these may be removed as well, as they are most likely unreachable following the same argumentation: Consider a neuron n that has no neighborhood connections left. At some point in time, there was an action that led from a state s to a state s' in $\mathfrak{R}(n)$ or that led from a state in $\mathfrak{R}(n)$ to another region. Due to the current layout of the approximation, this is no longer the case and thus, the region can be considered to be dead. Reasons for the development of such "redundant" abstract states could be changes in the environment or changes in the approximation due to adaptations or refinements.

7.6 Adapting the Approximation

The basic idea of the movement of neurons in the generic GNG is to adapt the network's topology to the probability distribution by which the samples of the input space are drawn (Fritzke, 1994b). In other words, the neurons should be placed in areas in which input samples can be expected. Although we follow the same intention we have to respect a certain property of the agent-environment interaction in reinforcement learning.

7.6.1 Adaptation of the Approximation in Reinforcement Learning

While learning from such episodic interaction with the environment the agent passes numerous state sequences: It starts in an initial state and transitions repeatedly to subsequent states until a goal state is reached. This time dependence (i.e. the agents wanders through the environment and thus the visited states depend on the current time and the transition model of the environment instead of some underlying topology) would "pull" the network towards the goal states, as in each time step the neurons would adapt to the current state trying to follow the agent's trajectories. Furthermore, a constant movement of the approximation would hinder the learning of a steady policy as Q-Learning concurrently operates on the current approximation.

To prohibit this moving target and to guarantee a static approximation during each episode, an additional set is introduced for each region: For every neuron n the set of *regional states* $\mathcal{R}_n$ stores all states the agent visited in n's region $\mathfrak{R}(n)$ in the current episode, i.e. all states s for which $n = \mathrm{nn}(s)$ held during the visit (cf. Figure 7.8). After each episode, the centroid $\overline{s_n}$ of $\mathcal{R}_n$ is computed for each neuron n:

$$\overline{s_n} = \frac{1}{|\mathcal{R}_n|} \sum_{s' \in \mathcal{R}_n} s' \tag{7.4}$$

The actual movement after each episode is derived from the adaptation of the generic GNG algorithm:

$$\Delta \vec{w}_n = \epsilon_b \cdot (\overline{s_n} - \vec{w}_n) \tag{7.5}$$

Here, each neuron n is moved towards the centroid $\overline{s_n}$ from Equation (7.4) with $0 \leq \epsilon_b \leq 1$ determining the strength of the adaptation. Thus, each centroid $\overline{s_n}$ of all states visited in neuron n's region $\mathfrak{R}(n)$ during one episode resemble the samples as targets of the adaptation. In this context, each visited state s of the reinforcement

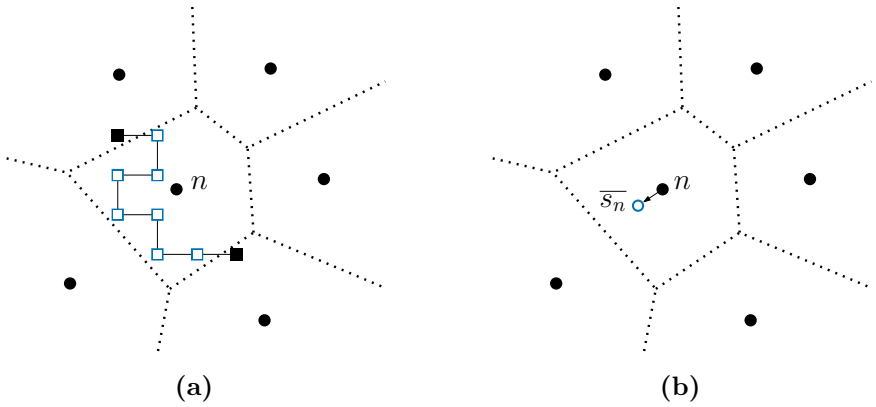


Figure 7.8: The neuron n is moved towards the centroid $\overline{s_n}$ (○ in (b)) of the regional states to adapt to states that the agent has visited in the current episode (□ in (a)).

learning task is part of the centroid $\overline{s_{n_1}}$ of its nearest neuron $n_1 = \text{nn}(s)$ and has thus also impact on the adaptation. Note, that in a first step all centroids are computed and in a second step all neurons are moved towards their centroids. Due to the resulting change in the layout it may of course happen that after or even during the movement a state that was in the region $\mathfrak{R}(n)$ of a neuron n during the episode (and thus contributed to the centroid $\overline{s_n}$) is no longer in $\mathfrak{R}(n)$ afterwards.

In the reinforcement learning context, the overall performance of the learning agent depends on the layout (i.e. the positions of the neurons) of the approximation as well as on the learned behavior on the current approximation. Too extensive changes in this layout may temper the performance and thus, the delayed adaption helps to increase the stability of the approximation as the approximation's layout remains static throughout each episode.

7.6.2 Differences to the Adaptation in the Generic GNG

While in the generic GNG every neuron is moved regardless of any condition, we here move a neuron n only if its associated error value is larger than a threshold Δ (e.g. $\Delta = 1$). The intention is to not move a neuron which is well positioned and has a useful Q-vector. It is intuitive to consider the neuron's error for this purpose as this error value is increased every time the policy in its region changes. Thus, the performance in areas with high error values should increase by repositioning the included neurons whereas neurons whose local policy was stable shall keep their position. Conversely, if the errors of all neurons are small the policy has stabilized.

Furthermore, in the generic GNG approach, each neuron is moved towards every presented sample x directly after the presentation of that very sample x while in *GNG-Q* only the nearest neuron to a state s is influenced by s. Additionally, in GNG each sample x has an (much smaller) influence on neurons outside the region of its nearest neuron $n_1 = \text{nn}(x)$ as also the topological neighbors of n_1 are moved towards x whereas in *GNG-Q* no state has an influence on the adaptation of neurons outside the region of its nearest neuron.

7.6.3 Analysis of Our Strategy

Having neurons in areas that are visited frequently is beneficial as these regions are important for reaching the goal of the reinforcement learning task at hand. Theoretically the idea of regional states can be motivated with the help of the so-called centroid condition: The *centroid condition* (Gersho and Gray, 1991) states that the quantization error for a set of reference vectors is minimized if each reference vector lies on the centroid of all data in its Voronoi region[11]. As we do not know all states in one region, our approach of storing all states the agent visited in one region $\mathfrak{R}(n)$ can be seen as an approximation of all states s that are actually in the region $\mathfrak{R}(n)$ (i.e. $\mathcal{R}_n \subseteq \mathfrak{R}(n)\ \forall n \in N_t$). To respect the approximative character of the regional states, we only move neurons a fraction of the distance towards the centroid of all those states instead of moving the reference vector directly on the centroid. Additionally, regions are split if the policy changes too frequently which indicates incompatible states. Especially in the beginning, the policy changes during learning e.g. caused by the needed exploration, but after some time high error values will most likely occur in impure regions. Thus, moving the neurons towards the centroids of their regional states helps to minimize the quantization error which helps to improve the performance of the learned policy.

As *GNG-Q* updates the approximation during learning, we have a moving target for the learning goal. In the adaptation phase of our approach new neurons may be added or existing neurons may be moved. Thus, these adaptions also change the abstraction function because ψ is defined by the positions of the neurons and the used distance measure.

After moving the neurons, one state s may be abstracted by a different abstract state than before because it may now be in a different region. Additionally, if a new neuron is added, the number of regions change and thus, states may be in a different region after the refinement, too. The refinement also changes the domain of the estimated Q-function but the influence is rather low as the Q-vector in the two new regions is the same as before the insertion. Dead regions that are deleted also change the domain of $\widehat{Q}$ but this does not influence the approximation as these regions were not visited for a long time.

7.6.4 Remarks for the Implementation

To avoid the storage of all visited states in one region, a *cumulative moving average* can be used, i.e. $\overline{s_n}$ is updated according to

$$v_{k+1} = \frac{1}{k+1} \cdot (k \cdot v_k + s_t)$$

to compute the centroid of all states visited until the current step (Welford, 1962) as presented by Knuth (1997).

[11] This is the core idea of the *Linde–Buzo–Gray* algorithm (Linde et al., 1980) that successively moves each reference vector to the centroid of its Voronoi region until the layout of the complete network stops changing. This approach finds a local minimum of the squared-error distortion measure.

7.7 Refining the Approximation

Finding suitable criteria of when and where to refine the approximation is one of the central challenges in adaptive resolution approaches for reinforcement learning (van Otterlo, 2009). We here present the adjustments to the original GNG-appraoch as it is described in Section 2.3.

As the generic GNG approach, the *GNG-Q* algorithm starts with a network consisting of two neurons. Thus, at the beginning, the reinforcement learning agent can only distinguish two regions and has two Q-vectors available to store its estimations (cf. Figure 7.4). On the one hand, it is desirable to have as few neurons as possible but on the other hand, there are of course situations when the network has to grow. Especially in the beginning, the network needs to grow in order to improve the learner's performance.

It is important that the approximation consists of regions that need only one Q-vector to express a useful behavior for every state in this region. We call a state region *pure*, if all states mapped into that region are similar and require similar behavior, i.e. the prototype Q-vector of this region's neuron is appropriate for all contained states. *Impure* regions contain states that require different Q-vectors to derive a useful policy.

We follow a top-down approach in which an initially coarse approximation is repeatedly refined in areas where the agent's behavior can be improved. While the Q-Learning algorithm learns the RL task on the approximated state space, the gained experience is used to determine impure regions that need a higher resolution.

7.7.1 Error Measure

The growing neural gas algorithm described in Section 2.3 maintains a local error variable for each neuron to determine where new neurons should be inserted. Usually, the quantization error is used for this purpose. In general, the error has to be a measure that shall be reduced and should indicate regions whose error will be decreased by inserting further neurons into that region (Fritzke, 1996). For state aggregation, the quantization error is not sufficient as it would only consider the similarity in the state space but disregard the similarity in the action space.

The goal of the *refinement* is to split impure regions with incompatible states. To design the error measure, we make use of the following observation: If the policy in the region of one specific neuron changes often, this is evidence that the states in this region require different behaviors and cannot be treated equally.

In *GNG-Q*, this error is expressed by counting changes in the local policy of each neuron's region. Initially, the resolution of the approximation is very coarse and in each learning step, Q-Learning is applied to the current approximation (cf. Figure 7.4). Regions that need refinement are identified by monitoring changes in the policy learned so far: Every time, a Q-update causes that

$$\arg\max_a \widehat{Q}_t(n_1, a) \neq \arg\max_a \widehat{Q}_{t+1}(n_1, a) \tag{7.6}$$

holds, the error for the current state's region (i.e. defined by neuron n_1) is increased because the agent would now prefer a different action for this region. Thus, frequent changes in the policy indicate the need to split.

Regions with high error values are periodically refined as these region contain states that have to be separated to learn a good policy. Regions with low error values consist of states, that are compatible and a useful behavior for all of them can be expressed with a single Q-vector. In each step, the error values of all neurons are decayed by multiplication with a factor $0 < \beta < 1$. This exponential decay emphasizes recent errors.

7.7.2 Inserting New Neurons

As the error depends on the changes of the maximal policy in the corresponding region, it is beneficial to add a new neuron close to the neuron with the maximal error value. Thus, the newly inserted neuron can help to reduce the error in the respective region. This intention is identical in the generic GNG approach.

In our work, we explore two different methods to insert a new neuron:

1. The new neuron is inserted identical to the insertion in the generic GNG.
2. New neurons are inserted by cloning an existing neuron.

For the first method, the reference vector of the a newly inserted neuron n_+ is positioned halfway between the neuron n_e with the largest accumulated error and the neuron n_f that has the maximal accumulated error value of n_e's topological neighbors. To exploit knowledge gained so far, the Q-vectors of those two neurons are averaged to form the initial Q-estimation for the new neuron:

$$\widehat{Q}(n_+, a) = \frac{\widehat{Q}(n_e, a) + \widehat{Q}(n_f, a)}{2} \ \forall a \in A \tag{7.7}$$

Additionally, the connection between n_e and n_f is removed and the new neuron n_+ is connected to n_e and n_f.

The second method works by cloning the neuron n_e with the highest error value and perturbate n_e and the new neuron n_+ by a small amount to ensure that their initial positions differ slightly. This splits the former erroneous region and supports the creation of pure regions. The new neuron n_+ is initialized with the Q-vector of n_e and connected to the same topological neighbors.

Both methods relieve regions in which the local policy changes often (e.g. oscillating between different actions) by adding a new neuron n_+ in the proximity of the neuron n_e with the highest accumulated error value. This results in a local refinement of the approximation as a portion of the states formerly assigned to n_e will now be assigned to n_+. The idea is that the new regions tend to be more pure. Especially in the beginning, the policy changes during learning e.g. caused by the needed exploration, but after some iterations high error values will most likely occur in impure regions.

The difference between the presented methods is rather marginal: The first method places the new neuron on the border of n_e's region while the second method places the new neuron inside the region close to n_e (cf. Figure 7.9). One advantage of the second method is that it does not depend on the neighborhood connections. In Figure 7.9(b) it can be seen that the region of neuron n_e is split approximately in half while the surrounding regions are only marginally affected.

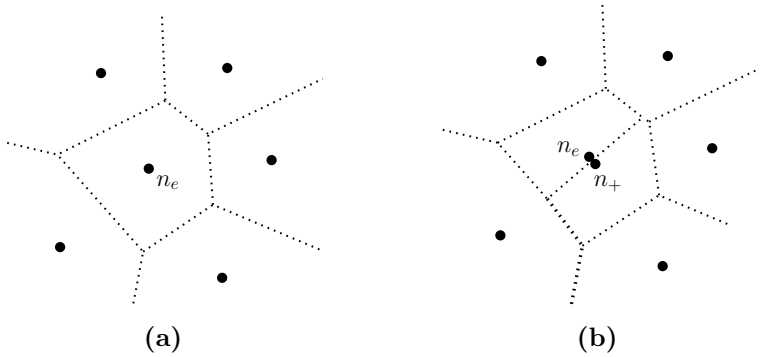


Figure 7.9: The region of the neuron n_e in (a) is refined by inserting the neuron n_+ into the vicinity of neuron n_e.

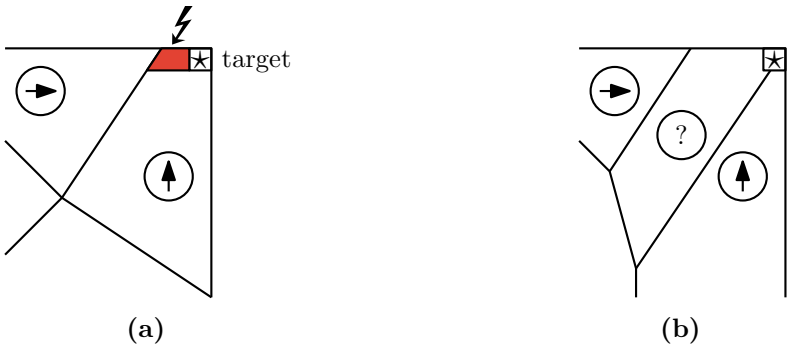


Figure 7.10: The approximation in (a) contains a small area (marked with ↯), in which the policy learned so far is erroneous. If this area is visited often enough, the Q-updates cause frequent changes in the local policy. After some time, this problem is resolved by adding a new neuron in that region (b).

The generic GNG approach redistributes the error information by initializing the error value of the new neuron n_+ with a value derived from the values of n_e and n_f. In contrast, *GNG-Q* resets the error values of all neurons present to zero to reflect the fact that the abstraction function ψ has changed.

In both insertion methods, the Q-vector of the new neuron reuses knowledge gained so far: In the method that "clones" the neuron with the error, it is useful to also copy the Q-vector as it contains values that have been learned for this area. For the second method, the new Q-vector is interpolated from the Q-vectors of the neurons between which the new neuron is inserted.

See e.g. Figure 7.10 where the current approximation contains a small region (marked with ↯) that causes problems with the currently learned policy (Figure 7.10(a)). The policy changes frequently as the agent learns that in that region "going right" is better than "going up" as the policy suggests. This is evidence that the current approximation includes regions with incompatible states. After the refinement (which is done identically to the insertion of the generic GNG), the approximation allows the agent to learn a proper policy (Figure 7.10(b)).

7.8 Stopping Criteria

As mentioned before, finding the right time to refine the approximation is a severe challenge. Clearly, this includes finding the time of when to *stop* the adaptation process (van Otterlo, 2009). Otterlo terms this the *stability-plasticity* dilemma as it is a trade-off between allowing the approximation to adapt (plasticity) and keeping the approximation stable in order to reduce variance (stability). In the generic GNG algorithm (Fritzke, 1994b) all parameters are constant over time to allow an adaption to changes in the distribution of the input space. This behavior can introduce over-fitting as more and more neurons would be added to the network.

A simple stopping criterion could be constructed by limiting the maximal number of neurons. However, the determination of a useful bound would require either much domain knowledge or extensive experiments with different thresholds. The goal of an adaptive state-space abstraction should therefore be to automatically stop growing when the approximation is fine enough to represent all relevant information of the state space.

One common approach in machine learning to avoid over-fitting in supervised learning tasks is the use of a validation set to monitor the current performance of the learning algorithm. Although we usually have no information about the correct behavior in RL tasks, a stopping criterion based on the validation set approach could be constructed by drawing an expressive (or in the absence of domain knowledge a large enough randomly chosen) set of initial states and to follow the maximal policy until a terminal state is reached. If the performance on this set is well enough, the insertion of neurons can be stopped. Furthermore, the movement and the removal of isolated neurons are stopped and the state-space approximation is fixated. The error information however is updated continuously to enable the network to resume movement and growing if the performance decreases.

In the *GNG-Q* approach, we refine the approximation *after* an episode, if

$$\sum_{n \in N_t} \text{error}(n) > |N_t| \tag{7.8}$$

holds and at least λ_{insert} episodes have passed since the last insertion of a new neuron. Thus, the refined approximation can be adapted for some time and the Q-vectors for the new approximation can be learned accordingly. Of course, one could refine the approximation whenever the sum of all errors is larger than zero. However, this might cause a too fine approximation, as sometimes a change in the policy is inevitable. The motivation for the condition above is, that on average each neuron is "allowed" to change its policy once per episode.

The condition stated above implicitly provides a *stopping criterion* for adjustments to the approximation: If the errors of all neurons are small, this is evidence, that the overall policy has not changed often since the last insertion and the current policy can be expressed sufficiently with the current resolution. *GNG-Q* uses the above criteria on the error to decide when the approximation should be refined or moved.

After the state-space approximation is fixed, our algorithm reduces to the generic Q-Learning algorithm and updates the Q-vectors for the neuron-action pairs on the approximation.

7.9 Eligibility Traces for State Regions

One way to deal with reinforcement learning tasks that have scarce or long delayed rewards is the use of *eligibility* traces (Dayan and Sejnowski, 1994). Although requiring more computation time per iteration they usually offer faster learning (Sutton and Barto, 1998). Eligibility traces combine features of both temporal difference methods and Monte Carlo methods.

Eligibility traces keep track of all state-action pairs the agent encountered during the current episode and offer a means to distribute immediate reward to all state-action pairs (s, a) that have been visited before. The strength of this "backpropagation" is done according to each state-action pairs' eligibility $e(s, a)$. This counter is increased by 1 every time the action a_t is performed in the current state s_t and if a_t is the action with the highest Q-value for state s_t. If a_t is not the maximal action, the eligibility traces for all state-action pairs are reset to zero. Additionally, each $e(s, a)$ is decayed by a factor $\lambda \in [0, 1]$ for all state-action pairs:

$$e_{t+1}(s,a) = \begin{cases} \gamma\lambda(e_t(s,a)+1) & \text{if } s = s_t \text{ and } a = a_t = a^\star \\ 0 & \text{if } a_t \neq a^\star \\ \gamma\lambda e_t(s,a) & \text{if } s \neq s_t \text{ or } a \neq a_t \end{cases} \tag{7.9}$$

with $a^\star = \arg\max_{a'} \widehat{Q}_t(s, a')$. Thus, reward or punishment can be credited for all state-action pairs that were "responsible" for it. If the agent performs an exploratory action (i.e. an action that has not the highest Q-value for the current state), the eligibility traces are cut off.

The method used here is called *Watkins's Q(λ)* (Sutton and Barto, 1998); for a comparison of other approaches see (Sutton and Barto, 1998; Dayan and Sejnowski, 1994). In every update, the temporal difference error δ_t is computed as $\delta_t = r - \widehat{Q}_t(s_t, a_t) + \gamma \max_{a'} \widehat{Q}_t(s'_t, a')$ between the current state s_t and the succeeding state s_{t+1}. This value is added to the Q-value of every state-action pair:

$$\widehat{Q}_{t+1}(s,a) = \widehat{Q}_t(s,a) + \alpha_t \delta_t e_t(s,a), \ \forall s \in S, \forall a \in A \tag{7.10}$$

Thus, the reward received for performing a_t in state s_t influences the learned Q-function along the actions derived from the greedy policy instead of only directly affecting $\widehat{Q}(s_t, a_t)$.

The transformation of this approach to neurons is straightforward: For each neuron n, we use $e_t(n, a)$ to express the eligibility for this neuron-action pair (compare Algorithm 4). In fact, this is identical to using eligibility traces on the abstract MDP $\widehat{\mathrm{M}}$.

7.10 Complete Algorithm

Algorithm 4 summarizes the pseudo code for our *GNG-Q* approach. To deal with different sized dimensions, it is useful to scale the values of the states to be from the same interval. A common approach is to normalize (Chakrabarti et al., 2008) a value $x \in [x_{min}, x_{max}]$ to a value $x_{scaled} \in [x^{min}_{scaled}, x^{max}_{scaled}]$ such that

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}} \cdot \left(x^{max}_{scaled} - x^{min}_{scaled}\right) + x^{min}_{scaled}.$$

Thus, the distance function employed in the nearest neighbor rule weights all dimensions equally and no dimension will be favored just because its values are from a larger scale. In our approach we use a normalization to the interval $[0, 1]$.

In line 7 of Algorithm 4, the agent selects its next action. At this point, it can either use the best action at this point (derived from the greedy policy as given in Equation (7.3)) or a random action to explore possibly better strategies. Usually we use the standard ε-greedy approach (i.e. the agent chooses a random action with probability ε and the maximal action with probability $1 - \varepsilon$) but clearly, any exploration scheme may be employed.

7.11 Analysis

In this section, we analyze the *GNG-Q* approach. First, we analyze the computational complexity of the *GNG-Q* approach. And second, we investigate the possible creation of non-determinism that may happen even in deterministic MDPs. Finally, we show an example of a state aggregation performed by *GNG-Q*.

7.11.1 Computational Complexity

We here analyze the complexity of our *GNG-Q* approach that is depicted in Algorithm 4. Note that due to the learning mode of reinforcement learning, several episodes each with an unknown number of steps are required to learn a satisfiable behavior. Our analysis thus only covers the runtime independent of these factors. For the interaction with the environment we assume that the agent can perceive the current state in time $O(1)$.

The initialization in line one can be done in $O(|A|)$ which is—due to the fact that the number of actions is constant—constant time. For each episode, the lines three and four have to be executed which both cost $O(|N_t|)$ where $|N_t|$ is the number of neurons at timestep t.

The rest of the approach can be split into four parts (we only discuss the instructions that cannot be done in constant time):

Maintaining the Error Values In the process of refining the approximation (line 33) *after* an episode, the neuron with the highest error value has to be determined. Concurrently, the errors of the neurons are adjusted *in every* episode: In line 13, the errors are discounted and in line 27 the error is potentially increased due to changes in the policy. Thus the naïve approach results in a list with a linear runtime of $O(|N_t|)$ for both the updates as well as the selection of the neuron with the highest error value. Concurrently, the ages of the neighboring neurons can be adjusted as well.

One could improve the error handling by using a *lazy heap* (Fišer et al., 2013): The combination of a heap with a "waiting" list allows the update to be performed in $O(\log |N_t|)$ and the selection of the neuron with the highest error value in $O(1)$.

Policy The computation of the nearest neuron necessary for the policy (lines 7–8) and the computation of the estimation for the succeeding state (line 10) can easily be done with a linear search that has a runtime of $O(|N_t|)$. Tree structures (as

Algorithm 4: *GNG-Q*

add two neurons n', n'' with random reference vectors and
$$\widehat{Q}(n', a) = \widehat{Q}(n'', a) = 0,\ \forall a \in A$$
foreach *episode* **do**
  initialize regional states $\mathcal{R}_n = \varnothing,\ \forall n \in N_t$
  initialize eligibility traces: $e(n, a) = 0,\ \forall n \in N_t,\ \forall a \in A$
  while *episode not finished* **do**
    `/* interaction with environment */`
    observe current state s_t and determine nearest neuron $n_1 = \mathrm{nn}(s_t)$
    select and perform action a_t
    observe subsequent state s_{t+1}, identify nearest neuron $n'_1 = \mathrm{nn}(s_{t+1})$
    `/* update neurons */`
    $\mathrm{visits}(n_1, a_t) \leftarrow \mathrm{visits}(n_1, a_t) + 1$
    store s_t in n_1's regional states: $\mathcal{R}_{n_1} \leftarrow \mathcal{R}_{n_1} \cup \{s_t\}$
    discount errors for all neurons
    connect neurons n_1, n'_1
    increase age of all neighborhood connections of n_1
    `/* update` $\widehat{Q}$ `*/`
    $\alpha_t = \frac{1}{\mathrm{visits}(n_1, a_t)^{\omega}}$
    $\delta_t = r - \widehat{Q}_t(n_1, a_t) + \gamma \max_{a'} \widehat{Q}_t(n'_1, a')$
    $e_{t+1}(n, a) \leftarrow e_t(n, a) + 1$
    foreach *neuron* $n \in N$ **do**
      foreach *action* $a \in A$ **do**
        $\widehat{Q}_{t+1}(n, a) \leftarrow \widehat{Q}_t(n, a) + \alpha_t \delta_t e_t(n, a)$
        if $a_t = \arg\max_{a'} \widehat{Q}_t(s_t, a')$ **then**
          $e_{t+1}(n, a) \leftarrow \gamma \lambda e_t(n, a)$
        else
          $e_{t+1}(n, a) \leftarrow 0$
    `/* Monitor changes in policy */`
    if $\arg\max_a \widehat{Q}_t(n_1, a) \neq \arg\max_a \widehat{Q}_{t+1}(n_1, a)$ **then**
      increase error(n_1)
  `/* Adaptation of approximation */`
  foreach *neuron* $n \in N_t$ **do**
    if $\mathrm{error}(n) > \Delta$ **then**
      compute centroid $\overline{s_n}$ of regional states for neuron n:
      $$\overline{s_n} = \frac{1}{|\mathcal{R}_n|} \sum_{s' \in \mathcal{R}_n} s'$$
      adaptation of neuron n to $\overline{s_n}$:
      $$\vec{w}_n \leftarrow \vec{w}_n + \epsilon_b \cdot (\overline{s_n} - \vec{w}_n)$$
  `/* Refinement of approximation */`
  if $\sum_{n \in N_t} \mathrm{error}(n) > |N_t|$ **then**
    insert new neuron in most erroneous region

e.g. a k-d tree) that partition the space into a binary tree are not applicable since the neurons are continuously moved, which would require many (rather expensive) insertions.
Fišer et al. (2013) suggest to store the neurons in a grid structure that allows linear runtime for the updates and "near constant" time for the search of the nearest neuron.
Additionally, $O(|A|)$ time is needed to select the action with the highest Q-value (line 8).

Eligibility Traces The computation of the eligibility traces needs time $O(|N_t||A|)$ and is thus rather costly but they usually improve the learning time in terms of episodes for the agent.

Adaptation The adaptation of the neurons after each episode (i.e. the movement in lines 28–31) can clearly be done in linear time.

Thus, each episode (that consists of e steps) needs $O(e \cdot (4 \cdot |N_t| + |A| + |N_t| \cdot |A|) + |N_t|)$ with the naïve approach and $O(e \cdot (|N_t| + |A| + \log |N_t| + |N_t| \cdot |A|) + |N_t|)$ with Fišer et al.'s approach.

With these results, each episode can be done in $O(e \cdot |N_t| \cdot |A|)$ with a naïve implementation. Using the improvements of Fišer et al. (2013) would also lead to an asymptotic runtime of $O(e \cdot |N_t| \cdot |A|)$ which could be improved to $O(e \cdot |N_t|)$ if the eligibility traces were omitted. Nevertheless, even with eligibility traces the overall performance of Fišer et al.'s approach is clearly an improvement.

Especially after learning, the usage of a tree-based structure to store the knowledge is highly beneficial in terms of storage and computation time for policy.

Corollary 10 (Complexity of *GNG-Q*). *With a naïve implementation, each episode with e steps of* GNG-Q *can be done in time $O(e \cdot |N_t| \cdot |A|)$ where $|N_t|$ is the number of neurons at time t and $|A|$ is the number of actions of the reinforcement learning task. This could be improved to $O(e \cdot |N_t|)$ using Fišer et al.'s (2013) approach and omitting the usage of eligibility traces.*

7.11.2 "Induced Non-Determinism"

Even in deterministic environments—i.e. the state transition function as well as the reward function is deterministic—the aggregation of states can introduce non-determinism in the abstract MDP: Consider the situation in a shortest path scenario depicted in Figure 7.11: If the agent performs the action "go right" in one of the states abstracted by the region $\widehat{\mathbf{s}}_1$, then, depending on its location in this region, the subsequent state can be "in" the abstract state $\widehat{\mathbf{s}}_2$ or $\widehat{\mathbf{s}}_3$. As the agent updates its estimates of $\widehat{Q}(\widehat{\mathbf{s}}_1, \rightarrow)$ depending on the Q-vector of the succeeding state, the Q-values are prone to oscillate. This problem occurs, if there are at least two states s_1, s_2 in one region $\widehat{\mathbf{s}}$ that result in states that are abstracted by different abstract states after performing the same action, formally:

$$\exists s_1 \neq s_2 \in \widehat{\mathbf{s}}, \exists a \in A : \ \psi(\mathrm{T}(s_1, a)) \neq \psi(\mathrm{T}(s_2, a)). \tag{7.11}$$

This kind of non-determinism can be caused by irregularly shaped regions (as in the *GNG-Q* approach, cf. Figure 7.11(a)) but may also occur whenever transitions between differently sized abstract states are possible (cf. Figure 7.11(b)).

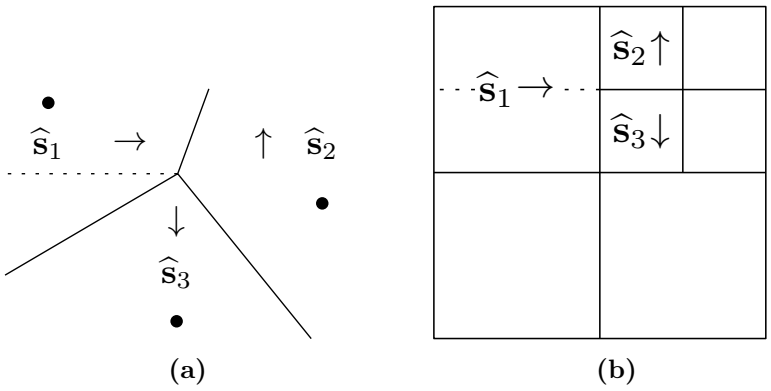


Figure 7.11: Induced non-determinism in different approximation schemes: The action $\rightarrow$ in $\widehat{\mathbf{s}}_1$ may lead to different succeeding states depending on the actual state that is abstracted to $\widehat{\mathbf{s}}_1$. If the action $\rightarrow$ is performed above the dotted line in (a) or (b) then the agent ends up in a state that is abstracted by $\widehat{\mathbf{s}}_2$ and in a state that is abstracted by $\widehat{\mathbf{s}}_3$ if the action was performed below the dotted line.

In (Fernández and Borrajo, 2008), this non-determinism is also called "the loss of the Markov property" as the subsequent state $\widehat{\mathbf{s}}_{t+1}$ now not only depends on $\widehat{\mathbf{s}}_t$ but also the actual state s_t of the original MDP and on the (actual) states visited before time t.

To improve the dealing with this non-determinism in the abstract MDP, we equip *GNG-Q* with a decreasing learning rate α_t that guarantee

$$\sum_t \alpha_t = \infty \qquad \text{and} \qquad \sum_t \alpha_t^2 < \infty\,. \tag{7.12}$$

Note, that this is the same condition on the learning rate as given by Watkins and Dayan (1992) for the convergence of Q-Learning.

Following the idea of (Even-Dar and Mansour, 2003a), such a learning rate can be constructed as

$$\alpha_t = \frac{1}{1 + \text{visits}(\widehat{\mathbf{s}}, a)^\omega} \tag{7.13}$$

where $\text{visits}(\widehat{\mathbf{s}}, a)$ refers to the number of how often action a was executed in the abstract state $\widehat{\mathbf{s}}$ and ω is a constant to regulate the decrease of the learning rate over time. To fulfill the condition above, $0.5 < \omega \leq 1$ has to hold. This learning rate decreases the influence of updates of each $(\widehat{\mathbf{s}}, a)$ over time and helps to reduce the oscillation of the Q-values.

Abstracting states transforms the learning task into a partially observable MDP (POMDP) (Singh et al., 1994; van Otterlo, 2009). Unfortunately, the improvement mentioned above does not provide a solution for the POMDP, yet it introduces more stability to the learning process: The core difference to "normal" non-determinism (that can successfully be solved by reinforcement learning) is the fact that the abstract successor state is not drawn identically distributed for a given *abstract* state and a given action but this distribution differs for different *actual* states that are abstracted by the same state (cf. Figure 7.11).

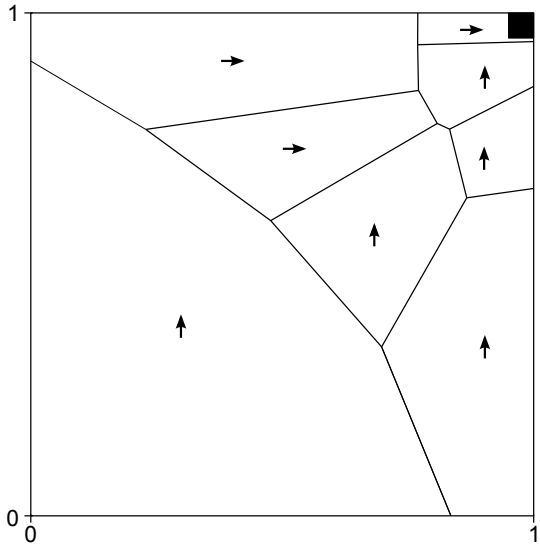


Figure 7.12: Example policy learned by *GNG-Q*.

7.11.3 Example Abstract State Space

Finally, we show an example of the state spaces computed by *GNG-Q*. The learning task is similar to that used later in the evaluation: The agent is situated in a grid world as in Figure 10.16 on page 180 with the target located at $(1, 1)$. The goal of the agent is to find the shortest paths from any position in the world to the target. For this, it perceives the continuous state that consists of its position (x, y) with $x, y \in [0, 1]^2 \subset \mathbb{R}^2$. It can perform steps of length 0.05 in any of the cardinal directions.

In Figure 7.12, an example policy of *GNG-Q* is shown. It can be seen that the neurons arrange more closely around the target resulting in a finer resolution while the region farther away from the target is broader. This makes sense as for the states more distant from the target it is useful to first "somehow" walk in the correct direction while the closer the agent gets to the target, the more accentuated the agent's behavior becomes.

Figure 7.13 shows a heat map that plots the distribution of all neuron positions in the final episode of 100 runs. It can be seen that most of the neurons are located on the diagonal towards the target. The heat map underlines the explanation given before: More neurons are located close to the target while the area farther away from the goal has in general a broader generalization.

7.12 Conclusion

We presented *GNG-Q*, a combination of Q-Learning and growing neural gas (GNG) that builds a state-space aggregation for reinforcement learning while the agent interacts with its enclosing environment. Our core idea is to use the GNG quantizer to aggregate similar states into regions that can be treated equally. *GNG-Q* refines regions where the learner's estimated policy changes often as this is evidence for a region that consists of incompatible states. Thus, similarity in both the state

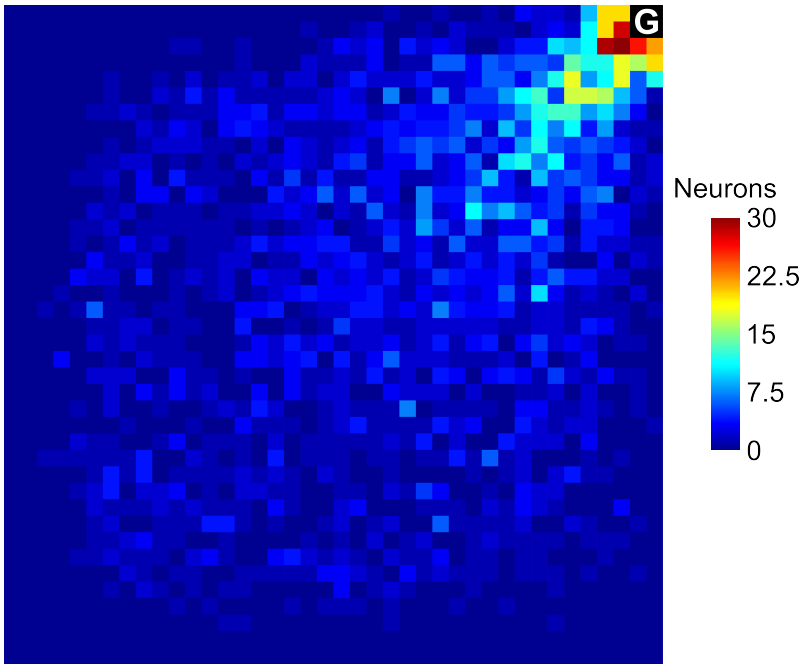


Figure 7.13: Heat map of the neurons' distribution over 100 runs.

and action space is respected. Of course, such approaches are only applicable if the reinforcement learning task contains states that are neighboring and share the same behavior—fortunately, this is usually the case.

The local refinement in erroneous regions improves the policy without affecting distant regions and is called "hard competitive learning" (Fritzke, 1998). The adaption of the approximation used in *GNG-Q* can be seen as parameter exploration as discussed e.g. by Rückstieß et al. (2010).

Our approach builds an approximation of the state space that is suitable to the particular reinforcement learning task in parallel with the learning. The Q-function in our approach is defined over neurons and actions and can be learned with tabular Q-Learning using one entry for every neuron-action pair. All states in one state region are treated equally, i.e. they all share one *prototype* Q-vector which leads to a piecewise constant approximation of the Q-function.

After finding a sufficiently fine approximation of the state space, each region consists of states that have the same optimal action. Then, our approach reduces to generic Q-Learning—now on a smaller and discrete state space. Advantages of *GNG-Q* include that knowledge achieved during learning is used to refine the approximation which supersedes the need of deciding on the granularity of approximation beforehand. Additionally, *GNG-Q* works online (i.e. at any time during learning, the agent can make use of the knowledge acquired so far) and does not need the model of the reinforcement learning task to compute an efficient discretization.

The agent needs to store solely the positions and the prototype Q-vectors of the neurons which results in a very compact representation. The state aggregation function is covered by the nearest neighbor rule and thus, this approach is well suitable for an implementation on robots. One additional possible application of

GNG-Q could be the usage as preprocessing step for other reinforcement learning algorithms: *GNG-Q* is used to compute an approximation for the RL task and then, every state-action pair is initialized with its estimation. After this, Q-Learning (or in fact any tabular reinforcement learning algorithm) can be used to adjust the values such that the optimal solution is found. This idea is similar in spirit to transfer learning as e.g. reported by Barrett et al. (2010) and might reduce the time needed while still ensuring to find the optimal solution.

8

Adaptive Function Approximation

As argued before, it is highly beneficial to add generalization to reinforcement learning and thus enabling the transfer of experience to unseen but similar states to speed up learning and to exploit gained experience more efficiently. Contrary to Chapter 7 where we aggregated states into state regions (i.e. we approximated the state space with a *state-space abstraction*), we here introduce the *Interpolating* GNG-Q (*I-GNG-Q*) that learns an approximation of the *value function* of the reinforcement learning task at hand.

Different approaches to approximate the value function exist, including the use of radial basis functions (RBFs) (Menache et al., 2005; da Motta Salles Barreto and Anderson, 2008) or by using other approximation methods (e.g. (Konidaris et al., 2011; Whiteson and Stone, 2006)). For detailed overviews of other approaches, we again refer to (Buşoniu et al., 2011a; van Otterlo, 2009).

Although *GNG-Q* offers many advantages and performs well in different reinforcement learning tasks, we here introduce an additional approach that uses the ideas of the *GNG-Q*-framework. Figure 8.1 illustrates the core idea of the new *I-GNG-Q* approach: While in *GNG-Q* all states in one region have the same Q-vector (which leads to a piecewise constant value function) here the Q-vector for a state is computed as combination of several prototype Q-vectors (that are assigned to neurons) and that the strength of each neuron's influence depends on the distance to the state.

In Figure 8.1(a) it can be seen, that *GNG-Q* would treat the states s_1 and s_2 identically as both are in the region of the neuron n_1. Nevertheless, the state s_2 is close to the boundary of neuron n_2's region and due to adaptations of the approximation (i.e. movement of neurons), the state may even fall into this region. It seems natural to assume, that the knowledge at the position of state s_2 may resemble a mixture of the knowledge encoded at the positions of neurons n_1 *and* n_2.

Our new approach also takes this into account: The Q-vector for state s_1 most strongly relies on the Q-vector associated with n_1 but also considers neuron n_2's Q-vector. Similarly, the Q-vector for s_2 consists of two nearly equal portions of both neurons' Q-vectors (although clearly the influence of n_1 is stronger since this neuron

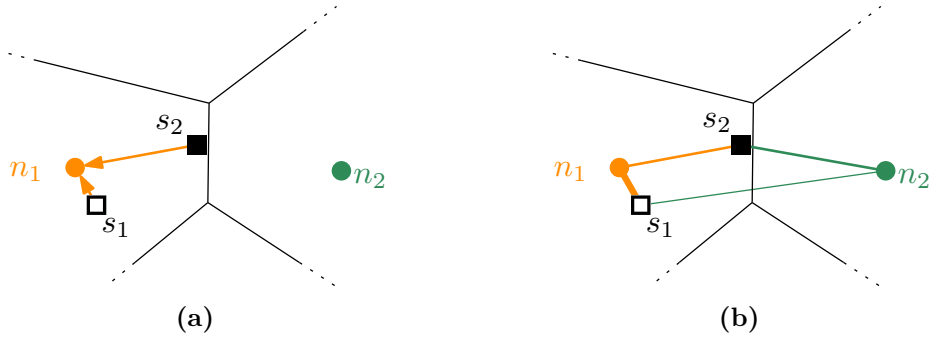


Figure 8.1: Two states s_1, s_2 in the same region of n_1 are treated differently in *GNG-Q* and *I-GNG-Q*: In (a) it can be seen that *GNG-Q* treats s_1 and s_2 identical while (b) shows that *I-GNG-Q* computes the estimation for a state by combining several neurons of this state's vicinity.

is the nearest neuron to s_2).

We implement this extension in a more general way: Instead of only using the two nearest neurons, we consider the k nearest neurons and derive the influence of each neurons' Q-vector with the method just mentioned. In fact, the borders depicted in Figure 8.1 are no longer present in the approximation for $k > 1$.

We use the idea of combining a growing neural gas and Q-Learning from the *GNG-Q*-framework and include the aforementioned ideas in order to further improve the performance of the approximation. In summary, the new *I-GNG-Q* approach has the following features:

- distance-based interpolation between learned prototype Q-vectors computes a smooth value function approximation:
- eligibility traces to features to speed up learning
- shares the advantages of *GNG-Q*: operates online and does not need the underlying model of the considered RL task
- as the computation of exponential functions used for RBFs is usually very slow (Schraudolph, 1999), the *inverse distance weighting* used here is a performant alternative
- incorporates an update rule that is proven to not diverge (Reynolds, 2002)

The key difference between *GNG-Q* and *I-GNG-Q* is that *GNG-Q* approximates the *state space* by aggregating similar states while *I-GNG-Q* directly approximates the *value function* of the reinforcement learning agent.

The main results in this chapter are the following:

- We motivate the use of a distance-based interpolation between Q-vectors to respect the distance of a state to its nearest prototype Q-vector and to incorporate several prototype Q-vectors in Section 8.1.
- In Section 8.3 we describe when it is useful to change the approximation and when this adjusting should be stopped.
- The core idea of the *I-GNG-Q* approach is described in Section 8.4 where we present the inverse distance weighting approach to derive a smooth value function.
- In Section 8.5, we show how the *I-GNG-Q* approach modifies the agent's policy

with an improved update rule that is proven to avoid divergence (Reynolds, 2002).
- We present a theoretical model of value function approximation in reinforcement learning and show how to include eligibility traces that allow a more efficient use of experience in Section 8.2.1.
- Section 8.7 analyzes the computational complexity of the *I-GNG-Q* approach and Section 8.8 compares *GNG-Q* and *I-GNG-Q*.

Additionally, Section 8.6 presents the pseudocode of the *I-GNG-Q* approach and Section 8.9 concludes this chapter. Parts of this chapter are based on (Baumann and Kleine Büning, 2014) which is the extend version of (Baumann and Kleine Büning, 2012).

8.1 Motivation

As Chapter 7, this chapter also deals with the approximation of reinforcement learning tasks with huge or even continuous state spaces. Nevertheless, we here employ a perspective that is different than before: *I-GNG-Q* directly approximates the value function V instead of aggregating states and concurrently learning on that approximated state space.

8.1.1 Benefits of Function Approximation

As already motivated in Section 7.1, the use of generalization highly improves the efficiency of reinforcement learning in complex tasks.

In the general reinforcement learning setting, for each state s and each possible action a the value $\widehat{Q}(s,a)$ is stored in some kind of tabular representation. The *GNG-Q* approach presented in the preceding chapter also stores the learned behavior for each *abstract* state and each action of the agent in e.g. a table.

One challenge while computing a policy on a state aggregation in parallel with updates to that aggregation is the fact that the value function may have steep transitions and changing the layout of the approximation may have a large impact on states close to the borders. Especially, two states of the original MPD that are neighboring but that are assigned to different abstract states may have Q-values that differ significantly even though the states in question may be arbitrarily close (in the case of continuous environments). Of course, this may also apply even to states that are not located at a border between abstract states. A potential cure for this exact problem is an approach called *soft partitioning* that allows the abstract states to overlap and to thus smoothen the borders between the abstract states (Bertsekas and Tsitsiklis, 1996).

We here implement the approach of directly approximating the value function of the reinforcement learning agent in an adaptive approximation scheme similar to the one introduced before. In that setup, the former Q-table with an entry for every state-action pair is replaced by a functional representation that is parameterized by a weight vector $\vec{\theta}$ (Sutton and Barto, 1998). Usually, in this approach the number of parameters is much smaller than the number of all possible state-action pairs the agent could encounter in the original reinforcement learning task and thus, the

complexity primarily depends on the *dimensionality* of the state space rather than on its size (van Otterlo, 2009).

In contrast to the table-based approach that is often employed in reinforcement learning, where each possible state of the environment has one entry in a table we here use a *feature vector* $\phi(s)$ to describe a state s. Usually, this allows a more compact representation by mapping the state space into a feature space (i.e. as the states are now represented by features, the associated search space is given by the combination of all features). This set of *features* can be built in diverse ways as we will discuss in the following.

8.1.2 Related Concepts

Although we here use a different approach to enrich the reinforcement learning agent with an approximation than in Chapter 7, the goal of approximating the value function is the same. We introduce generalization, i.e. we aim to enable the agent to transfer knowledge from experienced states to unseen states that may be similar to the ones already seen.

Approximating the reinforcement learning value functions can be done using different approaches, e.g. by using radial basis functions (RBFs) (Menache et al., 2005; da Motta Salles Barreto and Anderson, 2008) or by using other approximation schemes (e.g. (Konidaris et al., 2011; Whiteson and Stone, 2006)). For detailed overviews of other approaches, we refer e.g. to Section 3.3 or (Buşoniu et al., 2011a; van Otterlo, 2009).

As we will see in Section 8.2.1, each state is here represented by a feature vector. A convenient way to obtain such a feature description is using an approach similar to neural networks where the hidden neurons' activation functions are radial basis functions (for generic radial basis function networks see e.g. (Haykin, 1998)). Here, the feature value of the i-th center c_i for state s is computed by $\phi_i(s) = \frac{1}{\sqrt{2\pi{\sigma_i}^2}} e^{\frac{-\mathrm{d}(s,c_i)}{2{\sigma_i}^2}}$ where σ_i is the "width" of the bell-curved function and $\mathrm{d}(s, c_i)$ is the distance between the state s and the center c_i of the function. Although being influenced by all centers c_i of the network, the activation is highest in close vicinity of s.

Although similar in concept, (Konidaris et al., 2011) uses the Fourier basis to compute the features for a given state. Slightly more different is the approach presented by Whiteson and Stone (2006) that used a neural network trained by an evolutionary algorithm to approximate the agent's value function.

Drawbacks of such approaches often include runtime issues as the reinforcement learning task at hand has to be solved several times (i.e. once for each level of granularity of the approximation) or they require knowledge of the underlying MDP. Additionally, some approaches need batches of historic interactions of the agent: While this is sometimes useful, it cannot be used in online learning.

As the computation of exponential functions used for RBFs is usually time consuming (Schraudolph, 1999), the inverse distance weighting used here is a performant alternative. Additionally, the basis functions incorporated in approximation algorithms require to be designed a priori: Often, the algorithm heavily relies on proper choices of the relevant parameters (Ponsen et al., 2009). Additionally, information about the shape of the value function to be approximated is often required.

Without this information, such approaches often compensate with a (too) fine uniform resolution throughout the state space (Buşoniu et al., 2008).

8.1.3 Challenges in Function Approximation

A problem that may occur with approximations for reinforcement learning is called *exaggeration*: Learned values can be overestimated which can lead to divergence in the worst case (see e.g. (Thrun and Schwartz, 1993; Wiering, 2004)). This problem can occur if the estimats of the current state and the following state share parameters and thus errors in one state may lead to errors in other, depending states. Especially, the max operator in the temporal difference in Equation (8.3) can exaggerate the imprecise estimates of the learned value function. This problem is worsened if the agent stays (e.g. due to exploration) in specific areas of the state space.

Basically, there exists a trade-off between generalization and locality: On the one hand, a broader generalization affects more states and thus, the learner can use its experience more efficiently. On the other hand, updates that only affect small portions of the state space are less prone to overestimation: In the extreme case, every state of the state space has its own feature which is equivalent to having a table based approach.

A different trade-off has been noted by Sherstov and Stone (2005): There, approximation approaches are said to balance "representational power, computational costs, and ease of use". Otterlo (2009) addresses a similar trade-off; for him the relevant elements are computational costs to build and maintain the approximation, the possibility of allowing the agent to learn the correct task, and added benefits of the approximation.

8.1.4 Learning Goal in Adaptive Function Approximation

Similar to the goal in state-space aggregation, we here aim to find an approximation of the value function that allows the agent to learn a useful policy in the original MDP. One difference is the fact that we here only work on this original MDP instead of creating an abstract MDP as in *GNG-Q*.

Nevertheless, the learning of knowledge in parallel with its representation remains a challenging task. Indeed, one challenge in the training of adaptive function approximations arises from the fact that two learning tasks have to be solved: The first task is concerned with learning or updating the layout of the approximation. In our case this means the number as well as the positions of the neurons. The second task is to use the approximation to learn the best policy for the reinforcement learning task at hand. Here, this boils down to updating the weight vectors[1] $\vec{\theta}_n$ for each neuron n.

In classical function approximation this task is usually an instance of supervised learning which means that the correct values for a given set of data points is known beforehand. Usually supervised learning approaches iterate numerous times over this training set until the performance of the learned function is satisfying. In the cases of adaptive value function approximation in reinforcement learning, the target value (i.e. the value of the value function) for an input (i.e. a state) is not known in advance.

[1] See Section 8.2.1 for the general model of function approximation in reinforcement learning.

These values are updated during learning which leads to constant changes of the target. Since the trajectories of the agent depend on the value function as well, even the strategy that samples training data changes over the time of learning.

Note, that these two learning tasks are mutually dependent: Changes in the policy of the learning agent may lead to changes in the approximation (insertion of new neurons or repositioning of existing neurons, cf. Section 7.7 and Section 7.6). On the other hand, the performance of the learned policy also highly depends on the current approximation (see e.g. the example given in Figure 7.2) as the agent must have an appropriate approximation to cover all possible situations.

8.1.5 Benefits of Our Approach

Together with a rule that updates prototype Q-vectors depending on their influence on the current state, experience is generalized not only inside one region (as e.g. in the *GNG-Q* approach presented in Chapter 7) but also to Q-vectors outside the current region. In Section 8.5 we equip our approach with an *averaging* update rule inspired by the work of Reynolds (2002) that is proven to not exaggerate and that thus prevents the aforementioned problems for function approximation which have also been reported e.g. by Thrun and Schwartz (1993), Reynolds (2002), or Wiering (2004). Additionally, we incorporate eligibility traces to further increase the performance of the approximative reinforcement learning approach (cf. Section 7.9 where we implemented this approach for state aggregation).

The *I-GNG-Q* approach and *GNG-Q* share the advantage of using GNG that is quite insensitive to its parameter setting. One general advantage of using "growing" approximation approaches is the fact that they usually start with a coarse approximation that is refined which usually leads to smaller approximators.

New to *I-GNG-Q* is the combination of several prototype Q-vectors to from the value for a given state which offers several advantages. First, the interpolation between numerous prototype vectors offers a smoother value function than the piecewise constant function from state aggregation approaches. Second, the combination of more than one prototype Q-vector helps mitigating the negative impacts of possibly wrong estimations. Additionally, the distances between the state and the nearest neurons also play an important role in determining the estimated value which improves the performances especially for states that are rather far away from their nearest prototype Q-vector. Thus, nearby neurons have a larger impact on states than neurons farther away.

Figure 8.2 presents the *I-GNG-Q* approach in Sutton and Barto's (1998) agent-environment cycle. In each step, the agent perceives the current state s_t of the environment (1). This state is then fed into the *GNG state quantizer* where the k nearest neurons to s_t are computed. For each of these neurons, the associated weight vector is derived and all these information are fed into the *function approximator*. There, the estimate $\widehat{Q}(s_t, \cdot)$ for s_t is computed such that the influence of each of the k nearest neuron's weight vector depends on the distance of the respective neuron to s_t. From this resulting Q-vector, the agent can derive a greedy policy. During learning, the agent may use some exploration strategy (e.g. ε-greedy) to determine the next action a_t. After performing (2) this action a_t, the environment responds (3) with a reward $r_t = r(s_t, a_t)$ that represents the immediate value of performing a_t

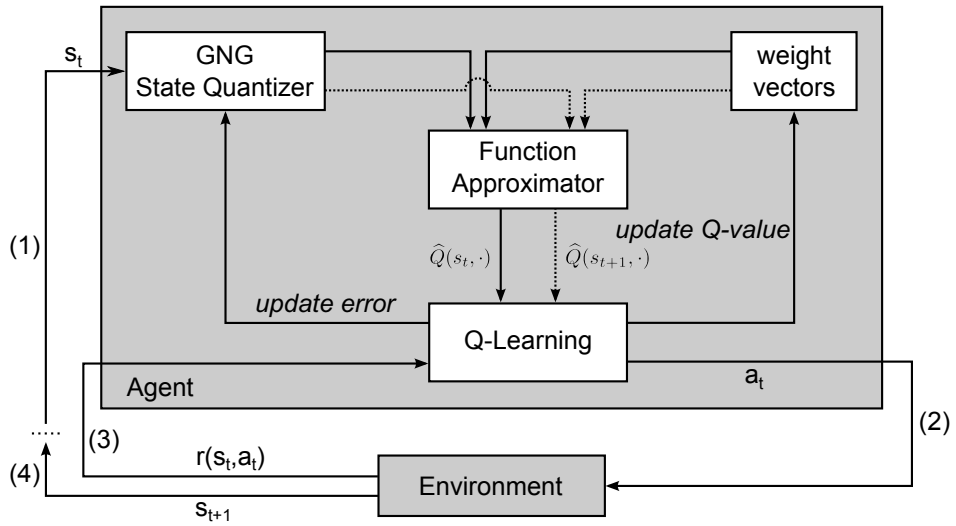


Figure 8.2: *I-GNG-Q* in Sutton and Barto's (1998) agent-environment interaction.

in state s_t and simultaneously transitions (4) into a subsequent state s_{t+1}. While learning, this reward and the estimate $\widehat{Q}(s_{t+1}, \cdot)$ for s_{t+1} is used by the agent to update its estimation of the Q-function (dotted lines in Figure 8.2). If this update leads to a change in the maximal policy, the error variable of the nearest neuron n_1 to s_t is updated to indicate the possible need for refinement in this region.

To summarize, the approximation computed by *I-GNG-Q* has the following properties:

- Estimated Q-values are built as combination of several prototype Q-vectors.
- The distance between the state and each of the nearest neurons affects their prototype Q-vector's influence on the estimation.
- Well balanced trade-off between locality and broadness of the influence both for the computation of Q-vectors and for their update.
- Neither exaggeration of the feature function nor of the update.

8.2 Function Approximation for Reinforcement Learning

Different to the approach presented in Chapter 7, we here strive to represent the agent's value function with a parameterized function. Usually, these function approximations have an exponentially smaller number of parameters than the number of state-action pairs in the original MDP (van Otterlo, 2009).

Solving the reinforcement learning task while simultaneously adapting the approximation is hard to achieve and thus, most methods use a fixed level of abstraction (e.g. a handmade setup of radial basis functions with predefined parameters) during learning (van Otterlo, 2009).

A crucial challenge with using function approximations (in general as well as in the context of reinforcement learning) is the requirement that a function should produce reasonable results throughout the function's domain given only a very limited set correct (labeled) data to rely on. In this context, it has to be ensured that the

samples from the known data are representative for the structure of the task such that anything that is learned with these data can be extended to usefully operate in the real task.

8.2.1 Theoretical Model

Amongst many others (for detailed overviews see e.g. (Buşoniu et al., 2011a) or (van Otterlo, 2009)), one approach to deal with large or continuous state spaces is to use a functional representation for the learned values that is parameterized by *weight vectors* $\vec{\theta}$ (Sutton and Barto, 1998). Depending on the behavior of the employed function, updating these vectors affects the values of numerous states and thus, the agent is able to generalize its knowledge. Often, linear functions

$$\widehat{Q}_{\vec{\theta}_t}(s,a) = \sum_{i=1}^{l} \phi_i(s)\vec{\theta}_t(i,a) \tag{8.1}$$

of $\vec{\theta}$ are used (Sutton and Barto, 1998; Melo and Ribeiro, 2007) where $\vec{\theta}_t \in \mathbb{R}^{l\times|A|}$ is a $l \times |A|$ matrix. The row vector $\vec{\theta}_t(i)$ is associated with feature i and $\vec{\theta}_t(i,a)$ is its component for action a. Here, instead of having a table cell for every state-action pair, l vectors $\vec{\theta}_t(i)$ with $|A|$ entries each are used as surrogate for the Q-vectors.

The *feature vector* $\phi(s)$ is a description of state s and consists of l features that are independent of the action:

$$\phi(s) = (\phi_1(s), \ldots, \phi_l(s)) \in \mathbb{R}^l\,. \tag{8.2}$$

Each of these features $\phi_i(s)$ measures the influence of the i-th component on the state s and can be computed with different means (common choices include RBFs or similar functions).

The learning of $\vec{\theta}$ can e.g. be performed based on gradient descent (Sutton and Barto, 1998; Melo and Ribeiro, 2007) (for brevity we write $\widehat{Q}_t(s,a)$ for $\widehat{Q}_{\vec{\theta}_t}(s,a)$):

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha_t \nabla_{\vec{\theta}} \widehat{Q}_t(s_t,a_t)\delta_t \tag{8.3}$$

with the *temporal difference*

$$\delta_t = r(s_t,a_t) + \gamma \max_{a'\in A} \widehat{Q}_t(s_{t+1},a') - \widehat{Q}_t(s,a) \tag{8.4}$$

and where $\nabla_{\vec{\theta}} \widehat{Q}(s_t,a_t) = \phi(s_t)$ holds, if ϕ is a linear approximation. This learning rule influences the values for all features and all actions where the strength of the influence is determined by the feature vector $\phi(s_t)$ of the current state s_t.

8.2.2 Function Approximation with Eligibility Traces

As in *GNG-Q* (cf. Section 7.9) we include eligibility traces to the *I-GNG-Q* approach.

The transformation of this approach to features is straightforward: For each feature (i.e. neuron) i, we use $e_t(i,a)$ to express the eligibility for this neuron-action pair and each $e_t(n,a)$ is increased by the value of $\phi_n(s_t)$ for all features:

$$e_{t+1}(n,a) = \begin{cases} \gamma\lambda(e_t(n,a) + \phi_n(s_t)) & \text{if } a = a_t = a^\star \\ \gamma\lambda e_t(n,a) & \text{if } a_t = a^\star \neq a \\ 0 & \text{if } a_t \neq a^\star \end{cases} \tag{8.5}$$

with $a^\star = \arg\max_{a'} \widehat{Q}_t(s, a')$. The value $\phi_n(s_t)$ can be considered as portion of eligibility of feature (i.e. neuron in the remainder of this chapter) n: Since the weight vector $\vec{\theta}(n)$ of feature n contributed with the strength of $\phi_n(s_t)$ to the Q-vector of the state s_t this fact should also be reflected in the distribution of the error. This transforms the concept of eligibility traces from the case where we have a counter for each state to feature based function approximation.

8.2.3 Classification of Approximation Approaches

In function approximation, the functions $\phi_i(s)$ are often referred to as *basis functions* that in combination build the approximation. Such approaches can be classified along several dimensions:

Fixed vs. Adaptable Fixed approximations are initialized once and never changed (i.e. the learning that takes place only affects the knowledge stored in the approximation and does not affect the approximation itself). On the other hand, adaptable approximators may also e.g. change the parameters or even the granularity of the approximation. This is a great advantage since it eliminates the need of defining the parameters that control the layout and the behavior of the approximation a priori. The downside is that adaptive architectures are usually more difficult to train (van Otterlo, 2009).

Local vs. Global Local approaches use only a small part of the approximation to derive a value for one state (most local "approximation": table) while global approaches use the complete approximation to compute the value for a given input (e.g. multi layer neural networks) (van Otterlo, 2009).

Global approximations are mostly more smooth, which can be both a curse and a blessing: On the one hand, we want the function to be smooth at some areas. On the other hand, a too smooth function may "smudge" soft details of the underlying data. Additionally, global approximations are prone to a phenomenon called *forgetting*: The learning approximator tries to minimize the deviation between the expected and the computed output by tuning its adjustable parameters. Due to the globality of the approximation scheme, these adjustments may not be limited to the close vicinity of the input but also affect portions of the approximation more distant. This might be desirable but often these distant adjustments "override" knowledge and thus, the approximation forgets historic data (French, 1999).

On the contrary, local approximators restrict updates to the learned knowledge to only a small part of the approximation. This might of course lead to the need of larger approximations to capture all necessary details of the input space.

Online vs. Offline Usually, function approximation methods are trained with repeated iterations over a batch of given training samples which is clearly an offline approximation. In reinforcement learning, no such training set is available as the samples (states with estimated Q-values) become available during the agent's interaction with the environment. This setting calls for online training methods.

The *I-GNG-Q* approach described in the following is *adaptable* and *online*. Additionally, the employed inverse distance weighting uses a trade-off between the

two extremes *local* and *global*.

8.3 Adjusting the Approximation

As in the *GNG-Q* approach, the approximation in *I-GNG-Q* can also be adjusted by two operations: The *adaptation* moves the neurons in such a way that they represent the approximated function as well as possible and the *refinement* is used to refine the resolution of the approximation. Once again, these changes to the approximation are performed after each episode to improve the stability of the learning progress.

8.3.1 Adaption in our Approach

The adaptation of the approximation in *I-GNG-Q* is performed identically to the adaption in *GNG-Q*: Each neuron n is moved towards the centroid of all states the agent has visited inside n's Voronoi region $\mathfrak{R}(n)$ in the current episode (cf. Section 7.6).

Here, the need for refining the approximation is also based on the current policy of the agent. In *GNG-Q*, the policy is monitored to figure out states that cannot be treated equally (which is resolved by creating a new abstract state) while in *I-GNG-Q* the refinement becomes necessary if a part of the value function cannot be represented sufficiently by the current approximation. Nevertheless, *GNG-Q* as well as *I-GNG-Q* use the same approach of deciding whether or not to refine the current approximation (i.e. both approaches use the amount of changes in the agent's policy as error measure).

Refining of the approximation is done by cloning the neuron n_e with the highest error value and perturbate n_e and the new neuron n_+ by a small distance to ensure that their initial positions differ slightly. The weight $\vec{\theta}_+$ of the new neuron n_+ is initialized with the weight vector $\vec{\theta}_e$ of n_e. After this refinement, the error values of all neurons are reset to zero, to reflect the fact that the approximation has changed. Having in mind that the weight vector of a neuron n can be interpreted as the Q-vector at the neuron's position, it is obvious that the adjustment operations are identical in both approaches.

8.3.2 Differences to the Generic Adaptation

Nevertheless, the motivation as well as the influence of these operations differ. In *GNG-Q*, we searched for "pure" regions of states that can be treated equally while in *I-GNG-Q* we refine the approximation in areas where the current value function causes an unsatisfactory policy. The repositioning of the neurons towards the centroid of the visited states is in *GNG-Q* used to support the creation of "pure" regions. In *I-GNG-Q* on the other hand, we move the neurons to areas that have been visited by the learning agent.

The intention of the adaptation in *I-GNG-Q* is that the bases of the approximation (i.e. the neurons) are moved to a different position while in *GNG-Q* the abstract states are moved. In general, the influence of moving one neuron is stronger in *GNG-Q* than in *I-GNG-Q* because in the latter approach, each approximated

value is computed from several bases and thus, the transition of the Q-value before the movement to the Q-value after the movement is smoother.

As stated above, the motivation for adaptation in GNG is to move neurons in areas where inputs can be expected. This behavior is useful only to a certain degree: As in *GNG-Q*, the neurons should find the subspaces of (probably high-dimensional) input spaces where samples can be expected (cf. (Fritzke, 1994a)) but we do not want the network to follow every state the agent has visited which would result in a highly unstable approximation. Thus, in *I-GNG-Q* the adaptation is also done after one episode has finished.

Adding a neuron in *I-GNG-Q* is also equivalent to refining the approximation but here, the approximation becomes more expressive while in *GNG-Q* a completely new state is created.

Nevertheless, *GNG-Q* and *I-GNG-Q* share similarity since both approaches work on the state space: *GNG-Q* works directly on the state space by grouping "similar" states while *I-GNG-Q* approximates the value function that is defined over the state space. The outcome is also similar as neurons are concentrated in areas that need a finer resolution.

8.3.3 Redundancy of Neighborhood Connections

In *I-GNG-Q*, the connections between neurons from the original GNG approach are not necessary. The insertion is handled by cloning the neuron with the highest error value and no further information can be drawn from maintaining the inter-neuron connections: *I-GNG-Q* does not learn an abstract MDP and maintaining a topological representation of the state space is possible but not very meaningful. Similarly, the approach presented by Fritzke (1994a) only uses the connections of the underlying GNG to determine the widths of the radial basis functions. As can be seen in Section 8.4, this is unnecessary in our approach.

8.4 Smoothing the Approximation

The *GNG-Q* approach as it is described in Chapter 7 treats every state in one region identically. Consider e.g. the value function in Figure 8.3(a) that has been learned on the same scenario as in Section 7.11.3: As all states in one region are treated equally, the value function is piecewise constant and thus, plateaus with steep transitions in between emerge. Although this approach is computationally fast, considering more than one Q-vector to compute the approximated values can improve the policy for states that are e.g. close to the regions' borders. Combining several Q-vectors results in a smoother value function and thus, the hard transitions tend to diminish (cf. Figure 8.3(b)). Having a smoother value function can especially help during the beginning of learning when the layout of the approximation is possibly not in its final form. Then, a state next to the border of its nearest neuron's Voronoi region may profit from not only depending on this region's value but also on those of its neighbors.

One example that illustrates the benefits of combining several prototype Q-vectors is given in Figure 8.1: In *GNG-Q*, the corresponding prototype Q-vectors are the same for both states s_1 and s_2 as they belong to the same region (Figure 8.1(a)).

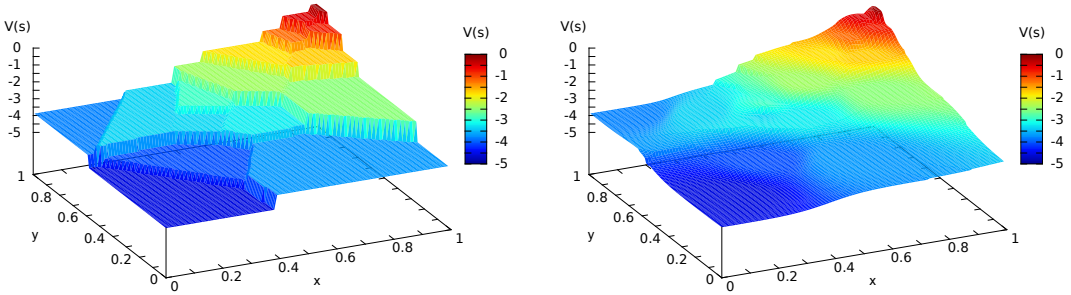


Figure 8.3: Value function comparison of a piecewise constant approximation (a) as computed by *GNG-Q* and an interpolated approximation (b) computed by *I-GNG-Q* in the same domain as in Section 7.11.3.

Since s_2 is close to the border of n_2's region, it would be more useful to compute the approximated Q-vector for s_2 as a combination of the Q-vectors associated with n_1 and n_2 inverse proportional to the distances of s_2 to n_1 and n_2. The same should hold for s_1, although the influence of n_1 will be much larger than the influence of n_2. In this context, the weight vector $\vec{\theta}(i)$ can be considered as the prototype Q-vector at the position of neuron i and the features $\phi_j(s) \in \phi(s)$ determine the influence of each neuron j on the state s (Figure 8.1(b)).

In this section, we implement the aforementioned approach to construct features for a given state while in Section 8.5 we add a rule that updates the same prototype Q-vectors that have contributed to the computation of the respective Q-value. Particularly, this means that

- the computation of the Q-vector for a state uses several Q-vectors outside the region of the nearest neuron.
- the update that is caused by the current state-action pair influences the Q-function outside the region of the nearest neuron.

The strengths of these updates will also be made according to the influences stated by the feature vector.

As already described earlier, the weight vector $\vec{\theta}(i)$ can be thought of as prototype Q-vector at the position of neuron i. Thus, at timestep t, the *I-GNG-Q* approach has $l = |N_t|$ neurons n, each equipped with one weight vector $\vec{\theta}_t(n)$. Note, that we assume that the ids of the neurons are updated such that all neurons present at timestep t can be addressed by $1, ..., |N_t|$.

8.4.1 Inverse Distance Weighting

In the *I-GNG-Q* approach, we compute the features $\phi_n(s)$ (cf. Section 8.2.1) that are used to describe a state s with a concept called *inverse distance weighting* (IDW) (Shepard, 1968).

The IDW approach was introduced to compute a continuous surface from irregularly spaced (empirical) data points. These data points may consist of a number of coordinates along with the associated value at the respective points. As the available data points might be thinly scattered over a large area, it is desirable to

interpolate the unknown function or process that generated the data to be able to obtain approximated values at arbitrary coordinates.

Shepard's approach is an interpolation that passes through all given data by computing a weighted average of the known samples. One possibility for such a weighting scheme is an inverse function that assigns large influence of data points nearby and less influence of data points farther away (Shepard, 1968):

$$f(\mathbf{x}) = \begin{cases} y_i & \text{if } \exists i : d(\mathbf{x}, \mathbf{x}_i) = 0 \\ \dfrac{\sum_{i=1}^{n} \frac{y_i}{d(\mathbf{x},\mathbf{x}_i)^p}}{\sum_{i=1}^{n} \frac{1}{d(\mathbf{x},\mathbf{x}_i)^p}} & \text{otherwise.} \end{cases} \tag{8.6}$$

In Equation (8.6) y_i are the values for the n data points at positions $\mathbf{x}_i$ and $d(\mathbf{x}, \mathbf{x}_i)$ is the distance of a point $\mathbf{x}$ in the plane to the known data point $\mathbf{x}_i$.

8.4.2 Inverse Distance Weighting Transferred to our Approach

Transferring Shephard's approach to our setting, we obtain the following weighting function:

$$u(s, n) = \frac{1}{\mathrm{d}(s, n)^p} \tag{8.7}$$

Using Equation (8.7) our approach assigns large influence of prototype Q-vectors close to the respective state and less influence of prototype Q-vectors farther away:

$$\mathrm{d}(s, n_i) < \mathrm{d}(s, n_j) \Rightarrow u(s, n_i) > u(s, n_j) \tag{8.8}$$

holds for all neurons $n_i \neq n_j \in N_t$ and for all $p \geq 1$. Thus, we can easily construct features for state s that respects the aforementioned properties:

$$\phi_n(s) = \frac{u(s, n)}{\sum_{n' \in N_t} u(s, n')} \tag{8.9}$$

The normalization in Equation (8.9) allows an improved generalization compared to non-normalized approaches (Bugmann, 1998). If not all distances between the state s and the neurons are greater zero (i.e. if at least one state is exactly located on a neuron n_0) $\phi_n(s)$ the estimated Q-vector is set to the prototype Q-vector of n_0.

Each feature value $\phi_n(s)$ indicates the influence of the Q-vector of neuron n for the computation of the Q-vector for the state s. Since we compute a feature value $\phi_n(s)$ for each neuron, the size l of the feature vector is determined by the current number of neurons $|N_t|$. Note that we also use $\phi_i(s)$ instead of $\phi_{n_i}(s)$ to refer to the feature value of neuron n_i.

Obviously all features computed as described above are greater or equal to zero and the sum of all features is one:

$$\phi_i(s) \geq 0 \ \forall i, \forall s \qquad \text{and} \qquad \sum_i \phi_i(s) = 1 \tag{8.10}$$

Thus, the approximated values for any state-action pair is always less than or equal to any value of the considered prototype Q-vectors. This guarantees that the feature function does not exaggerate.

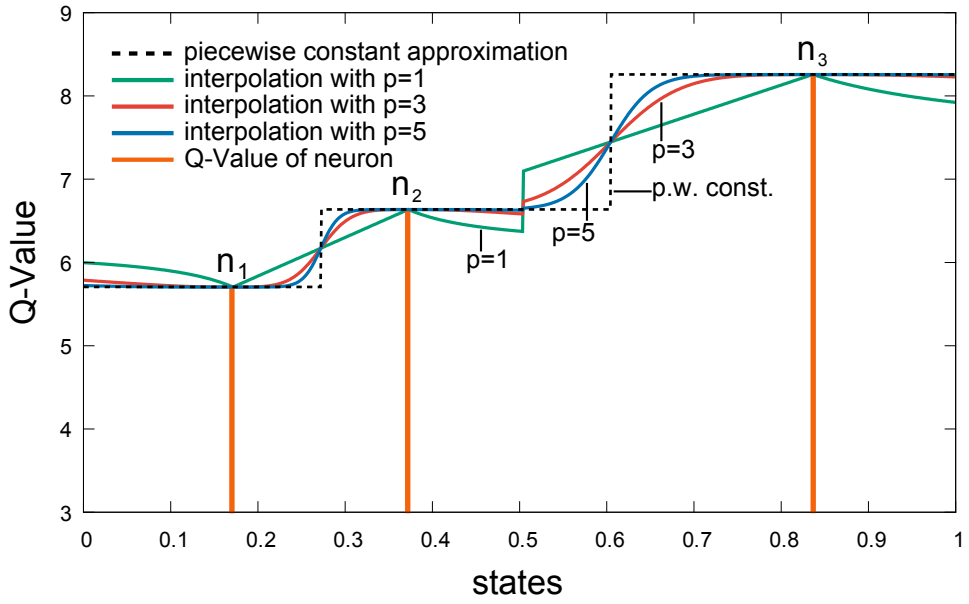


Figure 8.4: Different estimations of a value function for a one-dimensional state space: The Q-value for each neuron n_i is depicted as vertical bar and the dotted black line shows a piecewise constant estimation as e.g. computed by *GNG-Q*. The colored lines show estimations for the value function with $k = 2$ neurons and varying values for the exponent p of the feature function.

In general, the Q-vector for any state s would consist of a weighted combination of all $|N_t|$ Q-vectors associated with the currently present neurons. Clearly, this would lead to tasks as then every state would depend on every neuron in the current approximation and any update caused by a state would affect all prototype Q-vectors. To respect the desired locality, we only consider the $k \geq 1$ nearest neurons to state s to compute the features $\phi(s)$. Accordingly, $u(s, i) = 0$, if neuron n_i is *not* among the k nearest neurons to s. Thus, k elements of $\phi(s)$ are greater than zero (i.e. the features related to the k nearest neurons) while the other $l - k$ elements of $\phi(s)$ are zero. The Q-value for a state-action pair (s, a) is then derived by a combination of the prototype Q-vectors of the k nearest neurons weighted by $\phi(s)$.

Figure 8.4 shows different degrees of smoothness for the estimation of a one dimensional value function using $k = 2$ neurons for each estimation. The Q-value of each neuron is depicted as vertical bar and thus the approximation in this example incorporates three neurons. The dotted black line is a piecewise constant approximation e.g. computed by *GNG-Q*[2]. It can be seen that the estimation becomes smoother as p increases. The step at around 0.5 (visible for $p = 1$ and $p = 3$) is caused by the layout of the approximation: For every state less than 0.5, the neurons n_1 and n_2 are the nearest neurons and thus, for every state between n_2 and 0.5, the estimation interpolates between the Q-values of these two neurons. The value of k defines the number of prototype Q-vectors that are incorporated for the estimation of each Q-value and p influences the smoothness of the approximation: Higher values of the exponent p increase the influence of nearby prototype Q-vectors.

[2] Of course, this behavior can be invoked by setting $k = 1$.

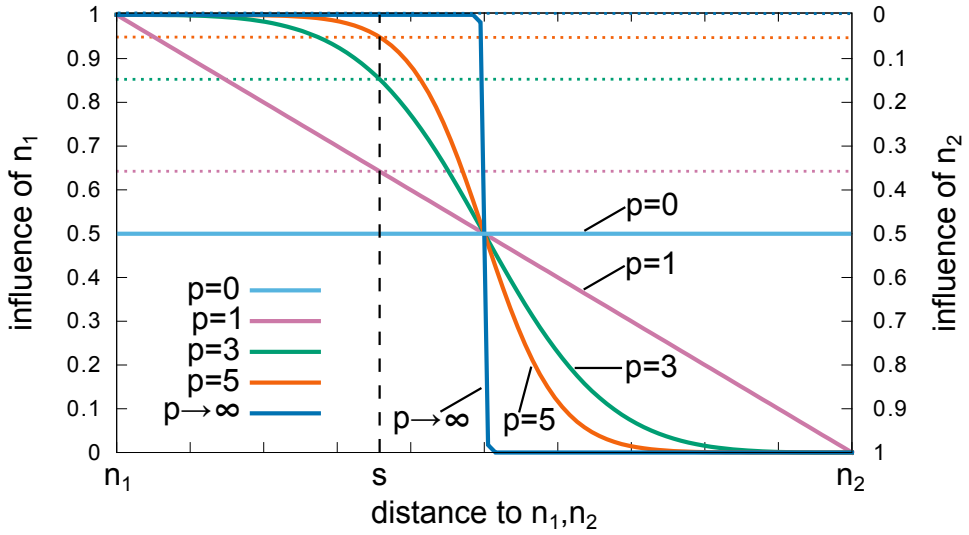


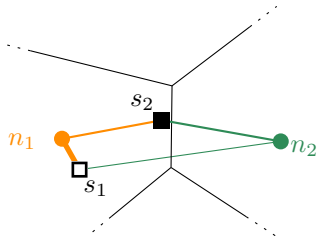
Figure 8.5: Influence of Q-vectors of $k = 2$ nearest neurons n_1 and n_2 with varying p for the computation of $\widehat{Q}(s, \cdot)$ for a state s. The dotted lines are located at the intersection of the dashed line indicating the state s and the plot for each p and show the influence for neuron n_1 on the left and the influence of neuron n_2 on the right. For $p = 1$, $\phi_1(s) \approx 0.64$ and $\phi_2(s) \approx 0.36$, for $p = 3$, $\phi_1(s) \approx 0.85$ and $\phi_2(s) \approx 0.15$ and for $p = 5$ we have $\phi_1(s) \approx 0.94$ and $\phi_2(s) \approx 0.06$.

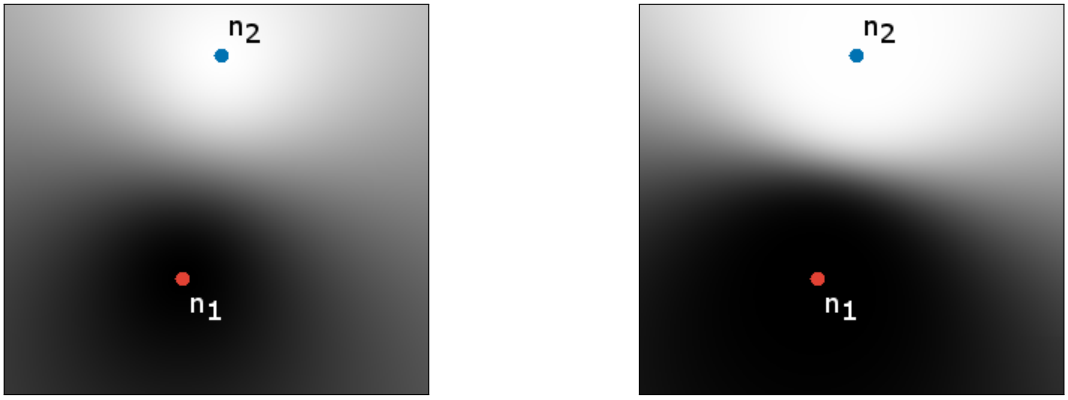
Figure 8.5 depicts the influence of the exponential parameter p slightly different: Consider a state s whose $\widehat{Q}$-value is approximated by the $\widehat{Q}$-values of $k = 2$ neurons. For $p = 0$ each of the nearest neurons' $\widehat{Q}$-vectors have an influence of 0.5 regardless of the state's distance to the neurons. For $p > 0$ the features ϕ_i depend on the distance of s to the neurons n_i: With $p = 1$ we have a linear influence and with increasing p, the influence of the nearest neuron increases. For $p \to \infty$, only the $\widehat{Q}$-vector of the nearest neuron is considered which is identical to the behavior in *GNG-Q*.

In Figure 8.6 we present a more geometrical interpretation of how p affects the influence of the $k = 2$ involved prototype Q-vectors. The two neurons are depicted as red (n_1) and blue (n_2) dots and the black areas are areas in which the prototype Q-vector of red dominates the term while the prototype Q-vector of blue dominates for states in the white areas.

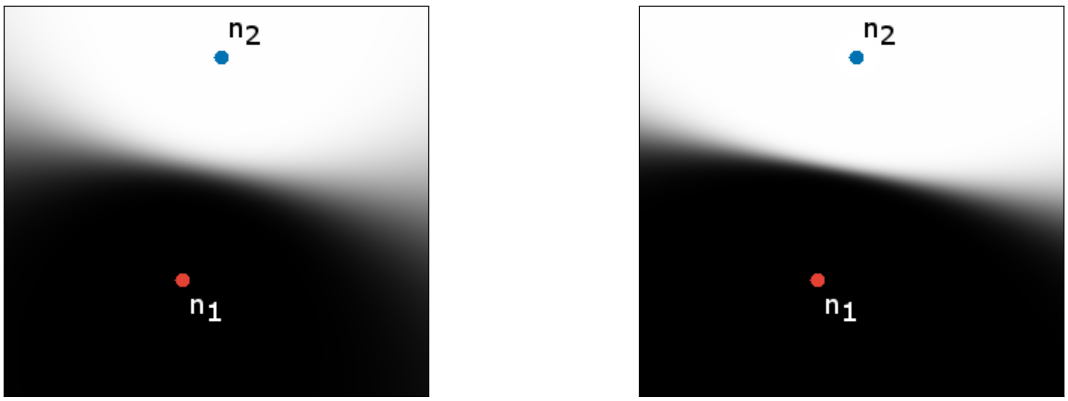
In the following, we present a short example of the computation that is performed by the *I-GNG-Q* approach.

Example 4 (Inverse Distance Weighting for Neurons). *Given the situation from Figure 8.1, we show the influence of each neurons' Q-vector on the approximated Q-vector for states s_1 and s_2:*





(a) $p = 2$ **(b)** $p = 4$



(c) $p = 8$ **(d)** $p = 16$

Figure 8.6: Influence of exponent p in a two-dimensional setting where the black colored areas mark the influence of the red neuron (n_1, lower) and the white areas show the influence of the blue (n_2, upper) neuron. The gray areas indicate areas where both neurons influence the outcome of the approximation (cf. (a)–(b)) while for increasing p the approximation depends more and more on the nearest neuron, only ((c)–(d)).

We here use $p = 3$ and $k = 2$ and show how the Q-vectors for the states s_1, s_2 are computed as weighted combination of the prototype Q-vectors of the neurons n_1, n_2.

We begin with Equation (8.7) *and compute the weighting $u(s, n)$ for each combination of states s_1, s_2 and neurons n_1, n_2. These values are used to compute the following features with the use of Equation* (8.9)*:*

$$\begin{aligned}\phi_1(s_1) &\approx 0.99\\ \phi_2(s_1) &\approx 0.01\\ \phi_1(s_2) &\approx 0.66\\ \phi_2(s_2) &\approx 0.34\end{aligned}$$

Thus, the feature vectors according to Equation (8.2) *are*

$$\begin{aligned}\phi(s_1) &= (0.99, 0.01)\\ \phi(s_2) &= (0.66, 0.34)\end{aligned}$$

Assume now that the approximation is made for a reinforcement learning task with four actions $A_e = \{a_1, a_2, a_3, a_4\}$ *and that the prototype Q-vectors*[3] *are* $\widehat{Q}_t(n_1) = (95, 100, 81, 81)$ *and* $\widehat{Q}_t(n_2) = (100, 81, 90, 81)$.

Using Equation (8.1)*, the Q-values* $\widehat{Q}_t(s_1), \widehat{Q}(s_2)$ *for all actions are computed:*

$$\begin{aligned}
\widehat{Q}_t(s_1, a_1) &= 0.99 \cdot 100 + 0.01 \cdot 81 = 95.05 \\
\widehat{Q}_t(s_1, a_2) &= 0.99 \cdot 90 + 0.01 \cdot 81 \;\; = \mathbf{99.81} \\
\widehat{Q}_t(s_1, a_3) &= 0.99 \cdot 81 + 0.01 \cdot 100 = 81.09 \\
\widehat{Q}_t(s_1, a_4) &= 0.99 \cdot 81 + 0.01 \cdot 90 \;\; = 81.00
\end{aligned}$$

$$\begin{aligned}
\widehat{Q}_t(s_2, a_1) &= 0.66 \cdot 100 + 0.34 \cdot 81 = \mathbf{96.70} \\
\widehat{Q}_t(s_2, a_2) &= 0.66 \cdot 90 + 0.34 \cdot 81 \;\; = 93.54 \\
\widehat{Q}_t(s_2, a_3) &= 0.66 \cdot 81 + 0.34 \cdot 100 = 84.06 \\
\widehat{Q}_t(s_2, a_4) &= 0.66 \cdot 81 + 0.34 \cdot 90 \;\; = 81.00
\end{aligned}$$

So, for s_1*, the action* a_2 *has the maximal value and would thus be chosen if the agent follows a maximal policy. This is identical to the action that would have been chosen with* GNG-Q*. For* s_2 *on the other hand, the action with the maximal value is* a_1 *which differs from the choice computed by* GNG-Q *(i.e. action* a_1 *as all states are treated identical as the neuron in whose region they are in).*

After the new approach of combining several prototype Q-vectors (represented by the weight vector $\vec{\theta}_t(n)$) has been introduced we show in the next section how these weights can be updated so that each update affects all prototype Q-vectors that have contributed to the approximated Q-vector of the current state.

8.5 Update Rule

Although the Q-update rule of the generic *GNG-Q* affected several states—to be precise all states in the region of the nearest neuron to the current state—no Q-value outside the region was updated even if the current state was close to the region's boundary. In *I-GNG-Q*, we update the weights (i.e. the prototype Q-vectors) of all neurons that contributed to the computation for the Q-vector of the current state. Generally, all neurons would be updated but we here follow the same approach as in Section 8.4: Every neuron n is updated by an action performed in a state s with the strength of the feature $\phi_n(s)$ as computed in Equation (8.9). In fact, each update triggered by an action a that was performed in a state s changes the values probably for many s' with $s' \neq s$ and a' with $a' \neq a$.

Unfortunately, this setup can be fraught with problems as function approximation with reinforcement learning can overestimate learned values and even lead to divergence in the worst case (see e.g. (Thrun and Schwartz, 1993; Reynolds,

[3] Note, that we here use the simplified concept of Q-vectors instead of the weights $\vec{\theta}$ as described earlier.

2002; Wiering, 2004)). This problem can occur if the values of the current state and the following state share parameters and thus errors in one state may lead to errors in other, depending states. The max operator in the temporal difference in Equation (8.3) can exaggerate the imprecise estimates of the learned value function. This problem is worsened if the agent stays (e.g. due to exploration) in specific areas of the state space.

As we have already seen in Section 8.4, the feature computation in Equation (8.9) guarantees that no weight is exaggerated by ensuring that every computed value is smaller/equal than any interpolation base.

I-GNG-Q updates the Q-vectors with a *gradient-descent approach* (i.e. moving the vector in the direction that will reduce the error the most) that aims to minimize the difference between each $\vec{\theta}$ and the Q-vector of the succeeding state. The standard update rule for function approximation in reinforcement learning moves the weight vector of the current state-action pair in direction of the reward plus the Q-value (that is computed using the complete weight matrix) of the next state.

In *averaging reinforcement learning*, the difference between the reward plus the Q-value of the succeeding state and the weight vector itself is used. The update rule from (Reynolds, 2002)

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha_t \left(r(s_t, a_t) + \gamma \max_{a'} \widehat{Q}(s_{t+1}, a') - \vec{\theta}_t \right) \phi(s) \tag{8.11}$$

is called *averaging update* and thus uses the difference between the Q-value of the succeeding state and the vector $\vec{\theta}$.

Reynolds (2002) proved that Q-Learning in combination with this update rule does not diverge. Later, Szepesvári and Smart (2004) proved the convergence of averaging Q-Learning for an interpolative mapping and following a stationary policy.

As mentioned before, we incorporate eligibility traces as described in Section 8.2.2, and thus the complete update in each step is given by

$$\vec{\theta}_{t+1}(i, a) = \vec{\theta}_t(i, a) + \alpha_t \delta_t e_t(i, a) \tag{8.12}$$

for all features i and all actions a with

$$\delta_t = r(s_t, a_t) + \gamma \max_{a'} \widehat{Q}(s_{t+1}, a') - \vec{\theta}_t(i, a) \tag{8.13}$$

being the temporal difference at time t.

8.6 Complete Algorithm

Here we combine the previously presented adaptations to *GNG-Q* to form the new *Interpolating* GNG-Q (*I-GNG-Q*) approach in Algorithm 5. For the sake of simplicity, we assume that the states consist of vectors with (real-valued) numbers. Otherwise, methods to measure the distance between neurons and states as well as to adapt the approximation have to be defined.

Similar to the *GNG-Q* algorithm, we also scale the state vectors as described in Equation (7.11). Identically, in line 8 any exploration approach can be employed.

Algorithm 5: *I-GNG-Q*

add two neurons n', n'' with random reference vectors and
$\vec{\theta}(n', a) = \vec{\theta}(n'', a) = 0,\ \forall a \in A$
foreach *episode* **do**
  initialize regional states
  initialize eligibility traces
  while *episode not finished* **do**
    `/* interaction with environment */`
    observe current state s_t
    determine nearest neuron $n_1 = \mathrm{nn}(s_t)$ to s_t
    select and perform action a_t
    observe subsequent state s_{t+1} and reward r
    determine nearest neuron $n_1' = \mathrm{nn}(s_{t+1})$ to s_{t+1}
    `/* update neurons */`
    store s_t in regional states: $\mathcal{R}_{n_1} \leftarrow \mathcal{R}_{n_1} \cup \{s_t\}$
    discount errors for all neurons
    connect neurons n_1 and n_1'
    increase age of n_1's neighborhood connections
    `/* update` $\widehat{Q}$ `*/`
    compute $\phi(s_t)$ using Equation (8.9)
    foreach *neuron* $n \in N$ **do**
      $e_{t+1}(n, a) \leftarrow e_t(n, a) + \phi_n(s_t)$
      foreach *action* $a \in A$ **do**
        $\delta_t = r + \gamma \max_{a'} \widehat{Q}_t(s_{t+1}, a') - \vec{\theta}_t(n, a)$
        $\vec{\theta}_{t+1}(n, a) \leftarrow \vec{\theta}_t(n, a) + \alpha_t \delta_t e_t(n, a)$
        if $a_t = \arg\max_{a'} \widehat{Q}_t(s_t, a')$ **then**
          $e_{t+1}(n, a) \leftarrow \gamma \lambda e_t(n, a)$
        else
          $e_{t+1}(n, a) \leftarrow 0$
    `/* Monitor changes in policy */`
    if $\arg\max_a \widehat{Q}_t(n_1, a) \neq \arg\max_a \widehat{Q}_{t+1}(n_1, a)$ **then** increase error(n_1)
  `/* Adaptation of approximation */`
  foreach *neuron* $n \in N$ **do**
    if error$(n) > \Delta$ **then**
      `/* compute centroid of` $\mathcal{R}_n$ `*/`
      $\overline{s_n} = \frac{1}{|\mathcal{R}_n|} \sum_{s_r \in \mathcal{R}_n} s_r$
      `/* adapt` n `to` $\overline{s_n}$ `*/`
      $\vec{w}_n \leftarrow \vec{w}_n + \epsilon_b \cdot (\overline{s_n} - \vec{w}_n)$
  `/* Refinement of approximation */`
  if $\sum_{n \in N} \mathrm{error}(n) > |N|$ **then**
    insert new neuron in most erroneous region

8.7 Computational Complexity

This section analyzes the complexity of the *I-GNG-Q* approach that is described in pseudocode in Algorithm 5. Once again, we analyze the algorithm's runtime per episode and assume that the agent can perceive the current state of the environment in time $O(1)$.

As in the *GNG-Q* approach, the initialization (line one) can be done in time $O(|A|)$ which is—due to the fact that the number of actions is constant or at least bound—constant time. For each episode, the lines three and four have to be executed which both cost $O(|N_t|)$ where $|N_t|$ is the number of neurons at timestep t.

The portion of the code that maintains the error values is identical to the on in the *GNG-Q* approach and thus takes time $O(|N_t|)$ for each update. Identically, the selection of the neuron with the highest error can be done in linear time. With the mentioned improvements (cf. Section 7.11.1), the update can be performed in time $O(\log |N_t|)$ and the selection of the neuron with the highest error in time $O(1)$. Adapting the neurons after each episode (i.e. the movement in lines 27–30) can clearly be done in linear time.

For the policy (lines 7–8), the k nearest neurons to the state s have to be computed. With a linear search, this can be done in time $O(k \cdot |N_t|)$. Using the approach of Fišer et al. (2013) as it was mentioned in Section 7.11.1, the updates can be done in linear time and the search of the nearest neurons would take "near constant" time. Additionally, $O(|A|)$ is needed to select the action with the highest Q-value (line 8).

Finally, the computation of the eligibility traces needs time $O(|N_t| \cdot |A|)$ and is thus rather costly but they usually reduce the number of learning episodes for the agent.

Summing up, each episode (with e steps) of *I-GNG-Q* needs $O(e \cdot (2 \cdot k|N_t| + |A| + 2 \cdot |N_t| + |N_t| \cdot |A|) + |N_t|)$ with the naïve approach and $O(e \cdot (2 \cdot k + |N_t| + |A| + \log |N_t| + |N_t| \cdot |A|) + |N_t|)$ using Fišer et al.'s update approach.

Corollary 11 (Complexity of *I-GNG-Q*). *Each episode with e steps of* I-GNG-Q *can be done in $O(e \cdot k \cdot |N_t| \cdot |A|)$ with a naïve implementation where k is the number of interpolation bases, $|N_t|$ is the number of neurons at time t and $|A|$ is the number of actions of the reinforcement learning task.*

8.8 Comparison *GNG-Q* vs. *I-GNG-Q*

One key difference between *GNG-Q* and *I-GNG-Q* is that *GNG-Q* approximates the state space by aggregating similar states (which leads to a piecewise constant value function) while *I-GNG-Q* directly approximates the value function. This difference can e.g. be seen in the frameworks in Figure 7.5 and Figure 8.2: For *GNG-Q*, a direct connection between the neurons and the Q-vectors exists while for *I-GNG-Q*, a Q-vector for each state is computed as combination of the weight vectors and the distances to the neurons. This is performed in the *Function Approximator* in Figure 8.2.

State aggregation methods usually compute a new abstract MDP on which learning is performed. The goal is to construct this MDP in a way that the derived

policy is also optimal (or at least useful) in the original MDP. In fact, aggregating states imply a loss of information: States from the original state space are combined in order to form a new abstract state and thus, a new representation is formed. Having this new representation, any suitable learning approach could be applied to derive a possibly different policy. Additionally, the abstract MDP may lead to further insights into the reinforcement learning task and point e.g. out areas that need special consideration. A drawback may be the fact that this operation cannot be reversed, i.e. it is in general not possible to deduce the original state from an abstracted state.

On the other hand, function approximation approaches do not create a new representation of the state space: The goal of function approximation is to formulate a hypothesis that allows to work for unseen samples. Thus, the aspect of generalizing knowledge is more prevalent than in state-space aggregation. Value function approximation does not perform a look up in order to derive the value for a given state but rather computes the value at this point (i.e. the state), instead.

Both concepts share the need for a suitable definition of *similarity*: State space abstractions need to know which states can be treated equally while the performance of function approximation usually gets worse for samples that are highly different from samples that have already been processed. In the latter case, this is often taken care of by the employed function approximation.

Coming back to the reinforcement learning value functions, it has to be noted that the value function on the abstract state space computed by *GNG-Q* is piecewise constant and thus discontinuous between two abstract states (cf. Figure 8.3(a)). After learning, the discontinuity may of course be unavoidable (and unproblematic), but during the course of learning, one state of the original MDP may be mapped to a different abstract state which in turn leads to a different Q-vector. On the contrary, the value function computed by *I-GNG-Q* is continuous (Shepard, 1968). This difference is also reflected in the way the learned values are stored: *GNG-Q* uses a tabular approach with a cell for each abstract state whereas *I-GNG-Q* has a weight vector $\vec{\theta}_n$ for every neuron n.

Note that the updates of both approaches influence numerous states: The update performed in *GNG-Q* influences all states of the original MDP that are mapped to the currently updated abstract state while *I-GNG-Q* influences states even beyond the Voronoi region of the nearest neuron. Additionally, *GNG-Q* treats every original state mapped into one abstract state identical regardless of its distance to the center (i.e. the neuron) of the abstract state while *I-GNG-Q* always respects the distance to all relevant neurons.

Using the terminology from Section 8.2.1, the feature function ϕ of *GNG-Q* is *binary* as it is defined as

$$\phi_i(s) = \begin{cases} 1 & \text{if } i = \text{nn}(s) \\ 0 & \text{otherwise.} \end{cases}$$

Thus, each original state s only activates one entry of the feature vector $\phi(s)$ while in *I-GNG-Q* k entries are non-zero.

Although the computational complexities of both approaches introduced in this thesis have the same asymptotic behavior of $O(|N_t|^2)$, *I-GNG-Q* is more expensive

as it has to compute the k nearest neurons instead of only computing nearest neurons. Then again, this clearly helps to improve the reinforcement learning agent's performance.

8.9 Conclusion

We saw that the GNG can also be used to create an adaptive function approximation that is adjusted during the interaction of the reinforcement learning agent with its environment. The enhancements in *Interpolating* GNG-Q (*I-GNG-Q*) reduced the time until stable behaviors are found and improved the regulation of the refinement and adaptation. Additionally, the approximated value function is smoother (Figure 8.3(b)) than the former piecewise constant approximation of *GNG-Q* (Figure 8.3(a)) because now Q-values are computed as weighted combinations of several prototype Q-vectors. *I-GNG-Q* is capable of learning compact approximations in parallel with an (nearly) optimal policy and its performance is well competitive with other approaches from literature without the need of knowing the considered RL task beforehand (cf. Chapter 10). Furthermore, we showed how to incorporate eligibility traces to speed up learning and to more efficiently use the agent's experiences and we formulated criteria for the adjustments of the approximation.

One major problem with function approximation in reinforcement learning is the exaggeration of the Q-values (e.g. (Thrun and Schwartz, 1993; Reynolds, 2002)). *I-GNG-Q* uses an update function and a feature function that are designed to prevent such an overestimation. Additionally, the combination of several prototype Q-vectors helps to stabilize the learning as possible erroneous knowledge can be corrected faster and does not have such high influence.

The *I-GNG-Q* approach has a computational complexity of $O(|N_t|^2)$ where N_t is the set of neurons that are present at timestep t. As the computation of exponential functions often employed for RBFs is usually very slow (Schraudolph, 1999), the *inverse distance weighting* used here is a performant alternative. The *I-GNG-Q* approach can rely on the fact, that the parameters for the GNG vector quantization only requires parameters that are well explored and that GNG is quite insensitive to these parameter values (Heinke and Hamker, 1998). Additionally, we only need two more parameters: The first, k, controls how much the approximation should generalize its knowledge while the second, p, is used to control emphasis of the nearest neuron's prototype Q-vector.

9 Evaluation

In this work, we presented the SHEPHERDING task and investigated solutions for instances with one sheep and one dog. For this, we presented the *GCC* algorithm that solves SHEPHERDING$(1, 1)$ tasks within close bounds (i.e. the upper and the lower bound differ in a term linear in the sheep's viewing range) on the solution length and present learned strategies in the following chapter.

Obvious extensions of the task are given by increasing the number of dogs n and the number of sheep m resulting in instances of SHEPHERDING(n, m).

As long as the instances involve one dog but several sheep (i.e. SHEPHERDING$(1, m)$), the task could still be solved with single-agent reinforcement learning methods. However, considering more sheep raises the question of how the sheep behave and interact with the dog as well as with each other (see e.g. the part on the biological background of shepherding in Section 4.2). The most simple approach would consider each sheep as being completely independent. In this case, the dog could employ the *GCC* algorithm for each sheep individually and drive the sheep one after another into the target. Possible models for herding behaviors include the *boid* approach by Reynolds (1987) that uses simple rules to derive quite realistic flocking behaviors. In this setting, the *GCC* approach might be used as a starting point by considering the complete flock as one "artificial" sheep that has to be controlled. Nevertheless, this would probably not lead to completely satisfying results as *GCC* is explicitly tailored to the behavior of one single sheep while controlling a flock of sheep would require to incorporate all additional dynamics introduced to the flocking behavior.

The major challenge in SHEPHERDING$(1, m)$ instances is the larger state space as for every sheep the dimensionality and thus the number of states increases. In general, this would result in longer times for learning useful behaviors as the agent has to become familiar enough with the states of the task.

When several dogs are included, the task becomes more complex as even the computation of the shortest paths for a cooperative multiagent system is PSPACE-hard (Hopcroft et al., 1984). Searching for an optimal solution with, e.g. A*, is often problematic as not only the number of states (as e.g. shown for the SHEPHERDING

task in Section 4.6) but also the branching factor grows exponentially (Wang and Botea, 2008).

Increasing the number of dogs n transfers the learning task from single-agent learning to multiagent learning. Especially for multiagent reinforcement learning, additional challenges arise (Buşoniu et al., 2010):

- The curse of dimensionality is intensified as the exponential growth of the state space is now in the number of states, actions, and agents.
- Exploration vs. exploitation is now in terms of the environment as well as in the other agents' behaviors.
- The learning goal becomes non-stationary as changes in the policy of one agent may affect the (optimal) policies of others.
- Coordination is needed since the effect of actions taken by one agent depends on actions of the other agents.

Thus, approximation approaches as often applied in single-agent reinforcement learning have to be evaluated regarding their applicability in multiagent reinforcement learning tasks.

A straightforward application of *GCC* in settings with multiple dogs and one sheep would let the nearest dog drive the sheep while the resulting dogs make sure to not get in the way. Nevertheless, this approach would not make use of the additional power provided by several dogs. A more sophisticated solution would introduce some means of coordination such that the dogs take useful positions to possibly even improve the upper bound on *GCC*'s solution length.

In the general case of Shepherding(n, m), the core idea of *GCC* might still be useful. However, major adjustments have to be made: First of all, the ideas mentioned before for dealing with some kind of flock need to be implemented. Particularly the dogs' interaction with a group of sheep has to be carefully modeled in order to not destroy the herding structure. Thus, cooperation and coordination of the dogs becomes crucially important. Second, the positioning of the dogs in the space as well as the paths the dogs use to get to their destination need in-depth considerations to allow proper behaviors of the dogs.

10 Experimental Results

In this section, we first experimentally evaluate the *GNG-Q* and *I-GNG-Q* approaches on benchmark tasks and later apply our adaptive approaches to the shepherding task.

We begin by describing the evaluation procedure, followed by a description of the different tasks. Then we describe and evaluate default parameter settings for *GNG-Q* and *I-GNG-Q*. Afterwards, we analyze the influences of the approaches' parameters individually as well as their combined influences and interdependencies. The results of this evaluation are compared to those of other approaches from literature. We close this part by pointing out the benefits of automatically created approximations.

After that we apply reinforcement learning to the shepherding scenario and investigate how this task can benefit from the adaptive approximation schemes developed in this thesis. There, the reinforcement learning approaches are compared with the solution of the greedy shepherding algorithm *GCC* and it is analyzed, in which situations of the scenario which approach is superior.

10.1 Experimental Setup

In this section we describe the experimentation procedure as well as the benchmark task used to evaluate our adaptive approximation approaches.

10.1.1 Experimentation Procedure

All algorithms were tested in several reinforcement learning tasks. For each algorithm and each considered learning task we evaluated the performance for different settings of the algorithm's parameters. In the following we call such a combination of algorithm, parameter settings, and reinforcement learning *experiment*. Each experiment was simulated for 50 runs and the results were averaged. In every run, we initialized the Q-tables or the vectors $\vec{\theta}$ with zero for every entry and used a different random seed.

We divided the evaluation in alternating learning and test phases:

Learning During learning, the agent performs updates according to the update mechanism of the tested approach and uses an ε-greedy approach (i.e. the agent uses an action chosen uniformly at random with probability ε and an action according to the greedy policy with probability $(1 - \varepsilon)$) to allow for proper exploration. After learning for 10 episodes the performance of the agent's current policy is *evaluated*.

Evaluation In the following evaluation phase the agent does not learn and is thus not allowed to change its policy. Instead, it always chooses the action with the highest Q-value without falling back to exploratory actions. The agent starts on randomly chosen start states and tries to reach a goal state. If the agent did not reach the goal after a fixed number of steps (that depends on the reinforcement learning task), the try was stopped. This is repeated for 100 different random start states and the results for all the start states are averaged to represent the performance at this point of time.

We measure the performance by counting the steps needed to reach the goal which obviously should be minimized. A different measure would be the received reward but as the agent should minimize the number of steps in any task considered here, we stick to the more intuitive way of reporting the number of steps[1]. In addition to this, we also evaluated the success ratio of the agent, i.e. how often the agent succeeds to reach the goal with less than the task dependent maximal number of steps. Nevertheless, we resort to this measure only when some additional insights can be gained. This measure is clearly also closely connected to the number of steps: For every situation from which the agent cannot reach the goal within the allowed number of steps, the (possibly) high number of maximal steps is added to the counter.

Additionally, the size of the approximation is measured: For tabular Q-Learning, the size is equal to the number of states of the environment while for *GNG-Q* and for *I-GNG-Q* the number of neurons constitute this measure. In *GNG-Q* the number of neurons is exactly the number of abstract states whereas in *I-GNG-Q* the number of neurons indicates the number of possible prototype Q-vectors from which the k nearest are chosen. Of course, the size of the approximation is not the primary measure as the smallest approximation consisting of one abstract state (or neuron) would almost never allow the agent to perform something useful. Nevertheless, for a given performance it is desirable to have the smallest approximation that offers this performance.

To sum up: Each run is evaluated after each ten episodes. The performance at episode t is evaluated on a set of 100 start states and the average number of steps to reach the goal from all these states is measured. Additionally, the size of the approximation as well as the percentage of reaching the goal at episode t is recorded. The performance of an experiment at episode t is the average of the respective measure of all 50 runs at episode t.

[1] Note, that the reward functions of the tasks are designed such that the agent *minimizes* the number of steps by *maximizing* the received reward.

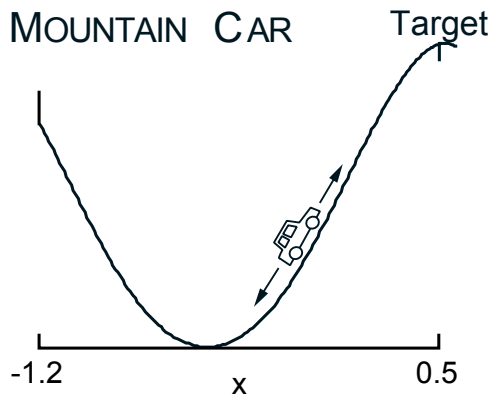


Figure 10.1: The mountain car task (figure taken from (Sutton and Barto, 1998))

10.1.2 Benchmark Task: The Mountain Car Domain

To evaluate the *GNG-Q* and the *I-GNG-Q* approaches, we use the well-known mountain car task (Boyan and Moore, 1994; Singh and Sutton, 1996). We use this task for the comparison of the base configurations for our approaches in Section 10.2 as well as for the detailed evaluations of *GNG-Q* and *I-GNG-Q* in Section 10.3 and Section 10.4. Unless otherwise stated, each approach was evaluated for 20,000 episodes.

In the mountain car domain, a car has to drive up a hill (cf. Figure 10.1). As it is too weak to accelerate up the slope at once, it has to drive in the opposite direction to gain velocity. The possible actions are *full throttle forward*, *full throttle backwards* or *neutral throttle*. The agent is rewarded with 0 for the action that leads to the target and punished with -1 else. The continuous state space has two dimensions and consists of the position $x \in [-1.2, 0.5]$ and the velocity $v \in [-0.07, 0.07]$. We use the physics as described in (Singh and Sutton, 1996):

- The valley is described by $\sin(3x)$.
- The allowed actions a_t are modeled as $+1, 0, -1$ for forward, neutral, backward.
- The state transition is given by $v_{t+1} = \text{bound}(v_t + 0.001a_t - g\cos(3x))$ and $x_{t+1} = \text{bound}(x_t + v_{t+1})$.

In the above description, the value $g = -0.0025$ is the force of gravity and the operation bound ensures that the value of each variable remains within the allowed bounds.

In each episode, the car is placed in a randomly chosen allowed combination of $x, \dot{x}$ and an episode is finished if the agent either reached the target or it passed 2000 unsuccessful steps. If the car hits the wall at $x = -1.2$ the velocity is set to zero.

The challenge in the mountain car task is the fact that the agent has to first depart from the target to build up enough momentum to finally reach the target (Gatti and Embrechts, 2013). Thus, the agent's reward has to first worsen before it can get better which is difficult for the agent and which causes a discontinuity in the value function (Drummond, 1996).

10.2 Comparison of Base Configurations for *GNG-Q* and *I-GNG-Q*

We here compare base configurations for *GNG-Q* and *I-GNG-Q* in the mountain car domain as described before. Later, these results will be used to evaluate the influences of the parameters in these approaches.

For the Q-Learning parameters, we chose the following values: discount factor $\gamma = 0.9$, exploration probability $\varepsilon = 0.05$, exponent for the time dependent learning rate[2] $\omega = 0.65$, and decay for eligibility traces $\lambda = 0.9$. The learning rate α_t was decreased over time for all approaches in relation to the number of visits of the state-action-pair or the feature-action pair, respectively.

The parameters of the base configuration for *GNG-Q* were:

- episodes between two insertions $\lambda_{insert} = 40$
- maximal age of neighbor connections $age_{max} = 300$
- adaptation strength $\epsilon_b = 0.05$
- the neuron's error decay $\beta = 0.9999$

For *I-GNG-Q*, the base configuration consists of all the parameter settings as described above for *GNG-Q* and additionally:

- exponent $p = 3$ of the inverse distance weighting function
- number $k = 3$ of neurons to be considered for the computation

As we can see from Figure 10.2, *GNG-Q* finds a good policy slightly faster in the beginning. Taking into consideration Figure 10.3, one can see that both *GNG-Q* and *I-GNG-Q* start by adding many neurons resulting in a rather fine piecewise constant approximation for *GNG-Q*. Each new state can be considered to be already partially learned because the Q-vectors of new neurons are initialized with Q-vectors of nearby neurons. Although this is true for both approaches, it appears as if *GNG-Q* could make more use of this initial information than *I-GNG-Q*. After around 2300 episodes, *I-GNG-Q* becomes more stable than *GNG-Q*. Nevertheless, *GNG-Q* has a final average value of steps of 58.7 while *I-GNG-Q* has a final average number of steps of 60.9.

Figure 10.3 shows the number of neurons needed to represent the learned policy for which the number steps are shown in Figure 10.2. In the end, *I-GNG-Q* needs 40 neurons on average whereas *GNG-Q* needs around 32 neurons on average. The number of neurons stabilizes after around 4200 episodes for *I-GNG-Q* and after around 7500 episodes for *GNG-Q*. The fact that *GNG-Q* needs more episodes to stabilize the approximation can also be seen in Figure 10.2: There, the number of steps also stabilizes at around 8000 episodes.

Although the average number of steps for both approaches is quite comparable, *I-GNG-Q* only needs 5870 episodes on average to always reach the goal (i.e. to have success rate of 1.0) while *GNG-Q* needs 13910 episodes to achieve this.

[2] We use the approach described in Equation (7.13) in Section 7.11.2.

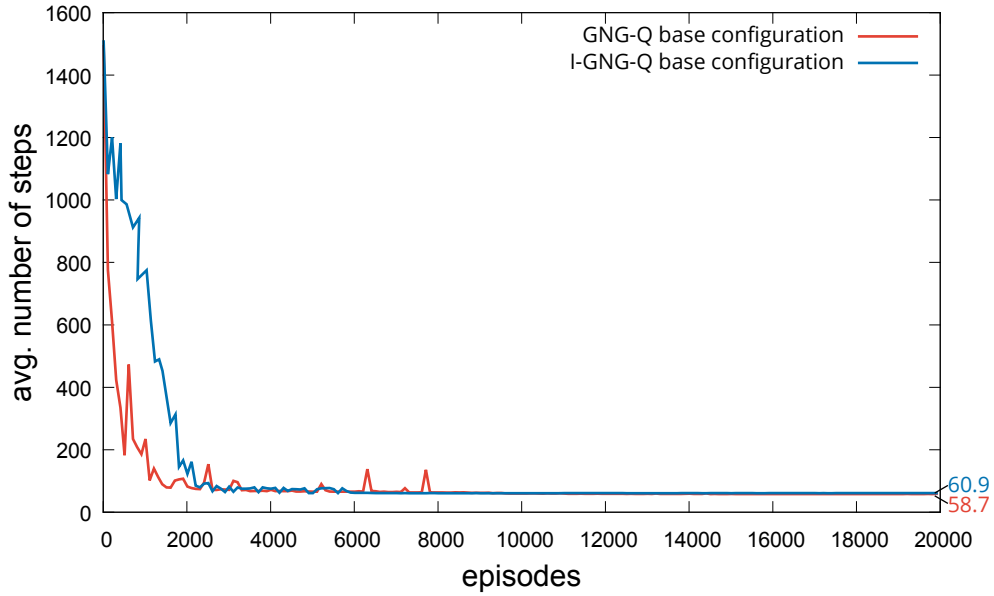


Figure 10.2: Average number of steps for the *GNG-Q* and *I-GNG-Q* base configurations in the mountain car task.

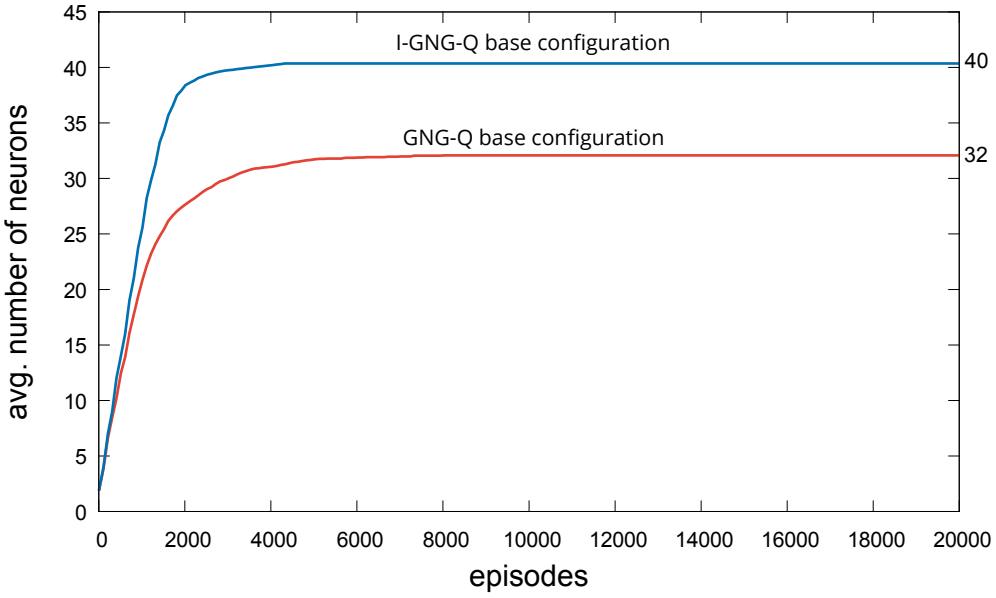


Figure 10.3: Average number of neurons for the *GNG-Q* and *I-GNG-Q* base configurations in the mountain car task.

10.3 Evaluation of *GNG-Q*

This section evaluates the performance in the mountain car domain (Section 10.1.2) if we change one parameter of *GNG-Q* while keeping the remaining parameters at the values of the basic configuration described in Section 10.2. For each variation we analyze the influence on the average number of steps needed to solve the mountain car task as well as the average sizes of the approximations. In each setting, we mark the value of the basic configuration by underlining it. Section 10.3.4 analyzes the effects of *GNG-Q*'s parameter on different metrics as well as possible interdependencies between these values.

10.3.1 Influence of the Insertion Delay λ_{insert}

Here, we investigate the influence of the insertion delay λ_{insert} for *GNG-Q* that defines the minimal number of episodes that have to pass since the last insertion before an additional neuron is added. We analyze the approach's behavior for $\lambda_{insert} \in \{10, \underline{40}, 80, 200, 400\}$.

As we can see in Figure 10.4, values for $\lambda_{insert} \in \{10, \underline{40}, 80\}$ result in finding a good approximation rather quickly: After around 1000 episodes, the average number of steps is already close to the final value. Nevertheless, the performance for $\lambda_{insert} = 80$ is less stable than the ones for $\lambda_{insert} \in \{10, \underline{40}\}$. Values of $\lambda_{insert} \in \{200, 400\}$ needs more episodes to learn a good policy and it remains more unstable as the others. This behavior is caused by the slower insertion of new neurons: As can be seen in Figure 10.5, the number of neurons for $\lambda_{insert} \in \{10, \underline{40}, 80\}$ increases relatively fast but stabilizes early. Thus, *GNG-Q* can distinguish more abstract states in the beginning which allows for a quick discovery of efficient policies. The disadvantage of such small insertion delays is clearly a probably a too fine approximation.

After 20,000 episodes, the average number of steps is between 55.72 (for $\lambda_{insert} = 10$) and 62.82 (for $\lambda_{insert} = 400$). The final average number of neurons varies from 13.88 with $\lambda_{insert} = 400$ to 54.42 with $\lambda_{insert} = 10$.

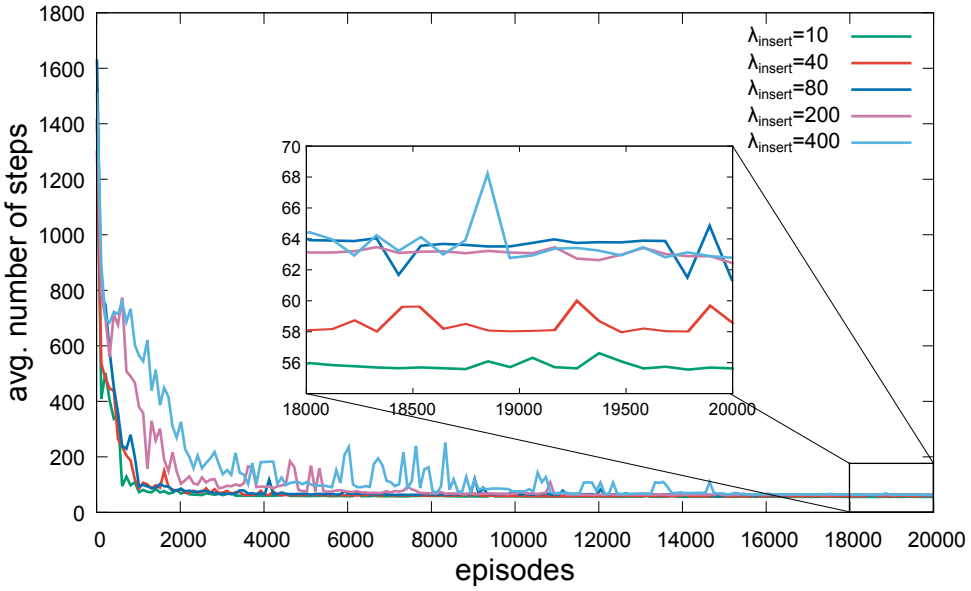


Figure 10.4: Average number of steps for *GNG-Q* with insertion delay $\lambda_{insert} \in \{10, \underline{40}, 80, 200, 400\}$ in the mountain car task.

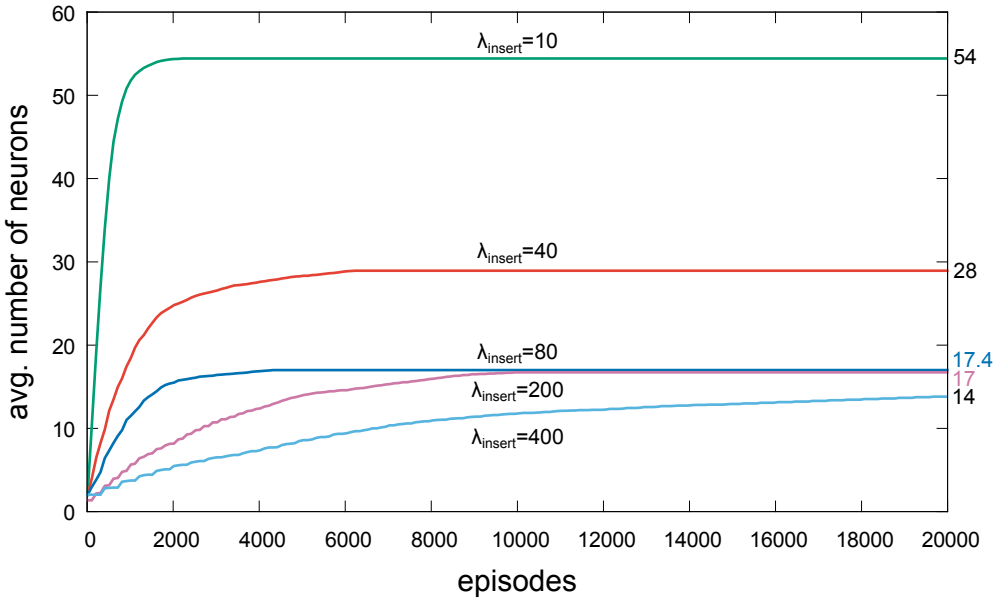


Figure 10.5: Average number of neurons for *GNG-Q* with insertion delay $\lambda_{insert} \in \{10, \underline{40}, 80, 200, 400\}$ in the mountain car task.

10.3.2 Influence of the Movement Strength ϵ_b

In this section we investigate the behavior of *GNG-Q* with varying values for the movement strengths of the neurons $\epsilon_b \in \{0.01, \underline{0.05}, 0.1\}$. Small values of this parameter result in a slow adaptation to the visited states while large values result in stronger adaptations: ϵ_b is the portion of how far the neuron is moved towards the centroid of all visited states in the neuron's region.

Figure 10.6 shows that the movement strength that adapts the neurons to the visited states has no clear influence on the time of finding a good approximation. All three values lead to a similar performances although $\epsilon_b \in \{0.01, 0.1\}$ leads to a more unstable behavior. The average number of steps in the last episode is 56.5 for $\epsilon_b \in \{0.01, 0.1\}$ and 58 for $\epsilon_b = 0.05$ and thus, these values result in a slightly better performance than the base configuration.

The average number of neurons can be seen in Figure 10.7: Here it is obvious that $\epsilon_b = 0.01$ needs more neurons on average (38) than $\epsilon_b = 0.1$ or $\epsilon_b = 0.05$ (32.36 and 32.06). Once again, the number of neurons is related to the stability of the performance: The setting with $\epsilon_b = 0.01$ only moves the neurons by very small portions which results in some peaks over time. These peeks are then responsible for the additional insertions of new neurons.

In the final episode, the final average number of steps is between 56.07 with $\epsilon_b = 0.01$ and 58.07 with $\epsilon_b = 0.05$. The average number of neurons is between 32.06 for $\epsilon_b = 0.05$ and 38.56 for $\epsilon_b = 0.01$.

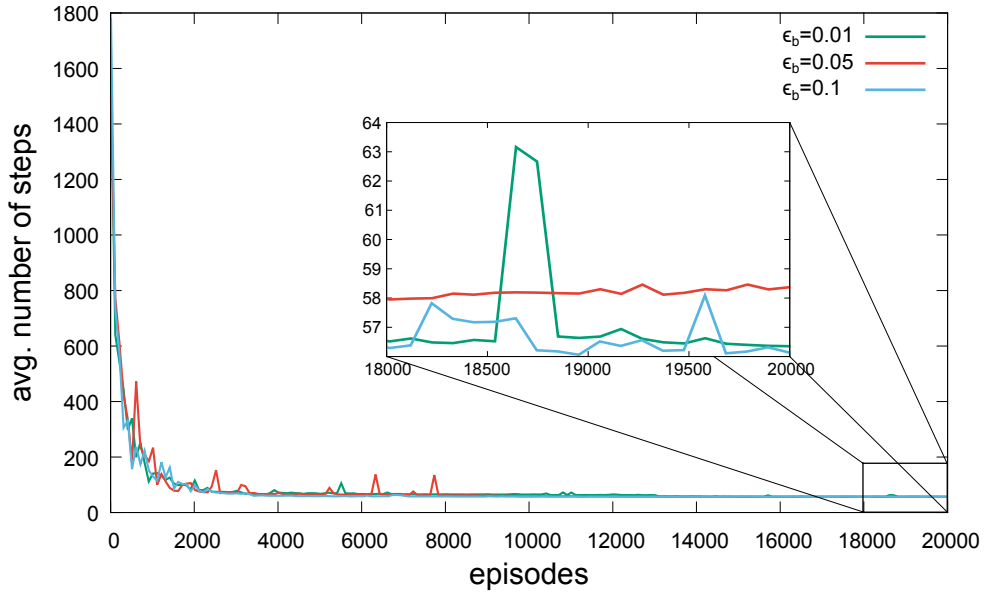


Figure 10.6: Average number of steps for the *GNG-Q* approach with movement strength $\epsilon_b \in \{0.01, \underline{0.05}, 0.1\}$ in the mountain car task.

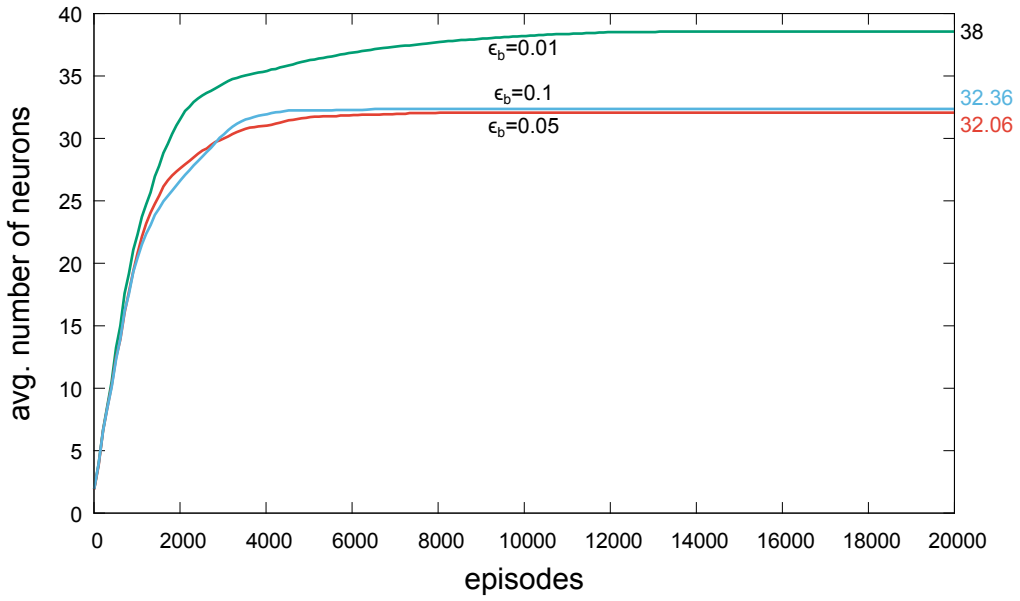


Figure 10.7: Average number of neurons for the *GNG-Q* approach with movement strength $\epsilon_b \in \{0.01, \underline{0.05}, 0.1\}$ in the mountain car task.

10.3.3 Influence of the Maximal Connection Age age_{max}

As final parameter for *GNG-Q* we analyze the influence of the maximal connection age $age_{max} \in \{50, 150, \underline{300}, 500\}$. This parameter controls how fast outdated connections and isolated neurons are removed.

Figure 10.8 reveals that once again, no clear influence on the speed of finding a good approximation can be established. For $age_{max} = 50$, the performance is rather unstable which continues until the end. The other values perform quite similar with the larger values (i.e. $age_{max} \in \{\underline{300}, 500\}$) leading to the most stable behaviors. This can be explained by taking into account the purpose of this parameter: Abstract states that have not been visited for a certain time should be removed. Nevertheless, the necessary and important exploration done by the agent may lead to some states not being visited for some time. A too small value for age_{max} may then result in prematurely removed states.

An investigation of the average number of neurons in Figure 10.9 shows that both "extreme" values for age_{max} result in rather small approximations (31.20–32.06 neurons on average) while $age_{max} \in \{150, \underline{300}\}$ results in average approximation sizes of 26.76–27.45 neurons. Although all four values are quite close, it is interesting to note that $age_{max} = 50$ results in a smaller approximation than $age_{max} \in \{150, \underline{300}\}$. The explanation for this is aforementioned premature removal of abstract states for small age_{max} that on the one hand reduces the size of the approximation but on the other hand results in a more unstable behavior.

After 20,000 episodes, the average number of steps is between 58.30 for $age_{max} = 500$ and 59.58 for $age_{max} = 150$. The final average number of neurons is between 26.76 for $age_{max} = 500$ and 32.06 for $age_{max} = 300$. Especially the final average number of steps is very close and thus, it can be assumed that the maximal connection age has little influence on the solution quality.

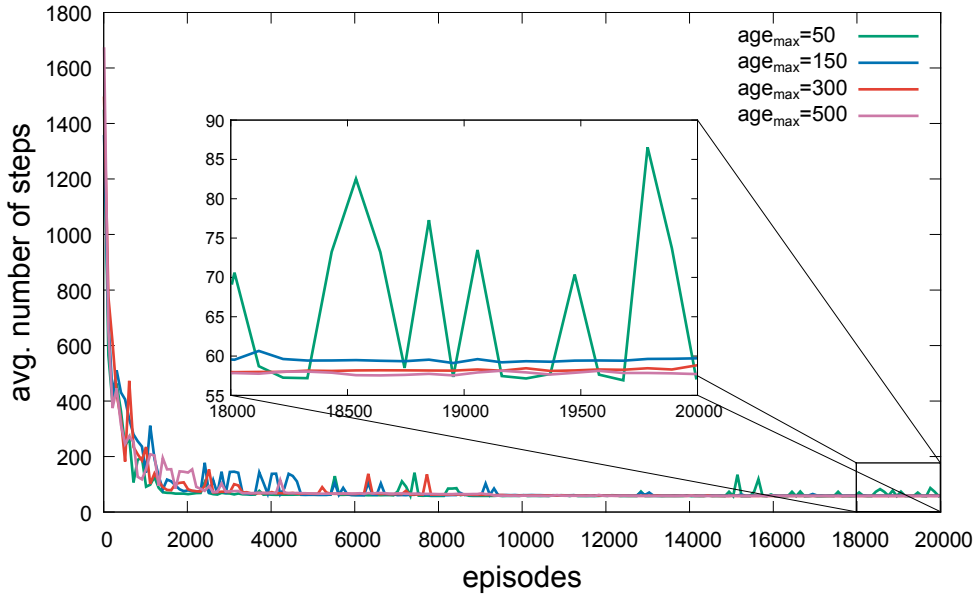


Figure 10.8: Average number of steps for the *GNG-Q* approach with maximal connection age $age_{max} \in \{50, 150, \underline{300}, 500\}$ in the mountain car task.

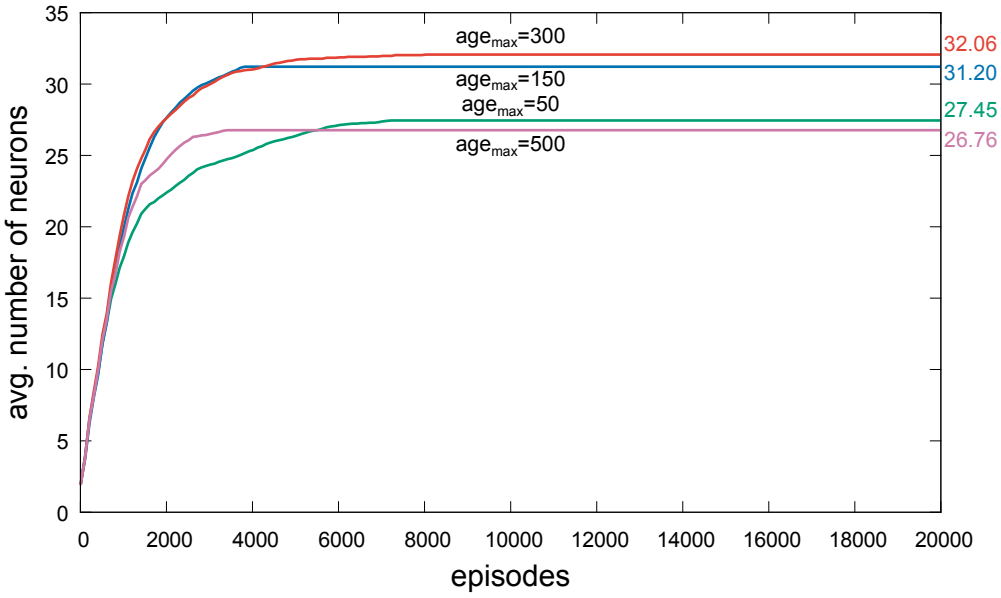


Figure 10.9: Average number of neurons for the *GNG-Q* approach with maximal connection age $age_{max} \in \{50, 150, \underline{300}, 500\}$ in the mountain car task.

Table 10.1: Levels for each factor in the 2^k factorial design for *GNG-Q*.

factor	low value (-1)	high value (+1)
λ_{insert}	10	400
age_{max}	50	300
ϵ_b	0.05	0.1

10.3.4 Interdependencies between Parameters

Here, we identify the influences of each parameter as well as the influences of its combinations by using a 2^k factorial design (Jain, 1991). This experimentation design investigates the effect of k *factors* (the parameters, here $\lambda_{insert}, \epsilon_b, age_{max}$) that each have two *levels* (i.e. high and low values of the parameters) on a *response variable* (i.e. the output of the experiment).

We investigate three response variables:

- the average number of steps in the last episode
- the number of episodes that is needed to first find a solution that differs by at most 5% from the best result in the complete run
- the size of the approximation

Clearly, all these response variables should be as small as possible.

In this section, we investigate the influences of the insertion delay λ_{insert}, the adaptation strength ϵ_b, and the maximal connection age age_{max}. For each of these factors, we specify low and high levels as given in Table 10.1.

For each combination of the high and low values of all parameters (i.e. one row of the result tables) we used the same experimentation procedure as described in Section 10.1.1 and thus, the values in the result columns are each averages of 50 repetitions for each 20,000 episodes. In the tables, +1 indicates the high value while -1 indicates the low value as given in Table 10.1. The row *Total/8* is the average effect of a factor, i.e. the change in the response variable created by changing the value of a factor from low to high averaged over all possible combinations of the other factors. The importance of a factor f can be expressed by how much of the variation it explains: This is given as the portion *Total/8* of f divided by the sum of all *Total/8* (Jain, 1991). The parameters are labeled *A, B, C* and combinations are labeled *AB, AC, BC, ABC*. For example the column *AB* measures the combined influence of parameter A and B.

Influence on the Quality in the Last Step

In Table 10.2 we can see that the largest influence on the performance in the last episode is the insertion delay for new neurons λ_{insert} with 78.36%. The second most influence has the combination of λ_{insert} and the strength of the adaptation ϵ_b with 9.38%.

From this we can conclude that high values of λ_{insert} will result in a higher number of steps needed and thus, smaller values of λ_{insert} are advisable. The influence of the combination of the factors λ_{insert} and ϵ_b is negative. In this case this means that for both high and low levels of λ_{insert} the value of ϵ_b has no influence while

for a low level of ϵ_b a low level of λ_{insert} is better. For a high level of ϵ_b the change between high and low level for λ_{insert} is not relevant.

Influence on Size of the Approximation

Obviously, λ_{insert} has the largest influence on the size of the approximation (cf. Table 10.3). This analysis was mainly done to investigate whether there are other factors that have an impact on the number of neurons. Nevertheless, it turned out that λ_{insert} has an (negative) effect of 97.69%: This means that small number of λ_{insert} result in large approximations and large values result in small networks.

Influence on the Speed of Finding Good Approximations

Table 10.4 shows that λ_{insert} has the largest influence (79.49%) on the speed of developing a good approximation. This can be explained by the behavior already investigated before: A small value of λ_{insert} allows to the approximation to be refined faster which results in a better performance early. So, too high values of λ_{insert} should be avoided.

The second large effect (9.79%) is the combination of λ_{insert} and the maximal age of the connections between neurons. This is also quite obvious as both factors have an influence on the number of new neurons added to the approximation.

10.3.5 Results *GNG-Q*

We saw that the insertion delay λ_{insert} has the largest influence on *GNG-Q*'s performance: The value affects the speed of finding a good solution, the size of the approximation and the final quality of the agent's performance. Lager values of λ_{insert} result in more compact approximations while small values allow the agent to find good solutions earlier. Additionally, smaller values of λ_{insert} result in a better solution quality both in terms of the final number of steps and of the steadiness of the performance. The analysis of the interdependencies confirmed that roughly 90% of the effects on all response variables can be explained by the insertion delay itself or by combined influences that include λ_{insert}. Nevertheless, the insertion delay had always the single largest influence.

The movement strength ϵ_b only has minor influences on the time of finding a good approximation and of the final solution quality. Still, too large and too small values influence the stability of the performance.

For the maximal connection age age_{max}, too small values (in relation to the insertion delay) result in less stable performances as well as in slightly longer times for finding a good solution.

Table 10.2: Results of the factorial design for the performance in the last episode with *GNG-Q*. The values in the result column are the average number of steps needed to solve the task in the final episode.

	λ_{insert}	age_{max}	ϵ_b					
I	**A**	**B**	**C**	**AB**	**AC**	**BC**	**ABC**	**Result**
+1	-1	-1	-1	+1	+1	+1	-1	55.64
+1	+1	-1	-1	-1	-1	+1	+1	61.05
+1	-1	+1	-1	-1	+1	-1	+1	54.68
+1	+1	+1	-1	+1	-1	-1	-1	60.65
+1	-1	-1	+1	+1	-1	-1	+1	57.92
+1	+1	-1	+1	-1	+1	-1	-1	58.46
+1	-1	+1	+1	-1	-1	+1	-1	55.77
+1	+1	+1	+1	+1	+1	+1	+1	60.76
464.93	16.91	-1.21	0.89	5.01	-5.85	1.51	3.89	**Total**
58.12	2.11	-0.15	0.11	0.63	-0.73	0.19	0.49	**Total/8**
	4.47	0.02	0.01	0.39	0.53	0.04	0.24	$\boldsymbol{SQ}$
	71.49	0.37	0.20	6.28	8.56	0.57	3.78	$\boldsymbol{SQ \cdot 8}$
Effect	78.36%	0.40%	0.21%	6.88%	9.38%	0.62%	4.15%	

Table 10.3: Results of the factorial design for the size of the approximation with *GNG-Q*. The values in the result column are the average numbers of neurons in the final episode.

	λ_{insert}	age_{max}	ϵ_b					
I	**A**	**B**	**C**	**AB**	**AC**	**BC**	**ABC**	**Result**
+1	-1	-1	-1	+1	+1	+1	-1	51.42
+1	+1	-1	-1	-1	-1	+1	+1	11.88
+1	-1	+1	-1	-1	+1	-1	+1	48.21
+1	+1	+1	-1	+1	-1	-1	-1	8.76
+1	-1	-1	+1	+1	-1	-1	+1	59.00
+1	+1	-1	+1	-1	+1	-1	-1	10.78
+1	-1	+1	+1	-1	-1	+1	-1	51.33
+1	+1	+1	+1	+1	+1	+1	+1	14.38
255.76	-164.16	-10.4	15.22	11.36	-6.18	2.26	11.18	**Total**
31.97	-20.52	-1.3	1.90	1.42	-0.77	0.28	1.40	**Total/8**
	421.07	1.69	3.61	2.02	0.60	0.08	1.95	$\boldsymbol{SQ}$
	6737.13	27.04	57.91	32.26	9.55	1.28	31.25	$\boldsymbol{SQ \cdot 8}$
Effect	97.69%	0.39%	0.84%	0.47%	0.14%	0.02%	0.45%	

Table 10.4: Results of the factorial design for the number of episodes until *GNG-Q* finds the first time a solution that is at most 5% worse than the best solution. The values in the result columns are the number of episodes after which the average behavior reaches the required quality.

	λ_{insert}	age_{max}	ϵ_b					
I	**A**	**B**	**C**	**AB**	**AC**	**BC**	**ABC**	**Result**
+1	-1	-1	-1	+1	+1	+1	-1	8750
+1	+1	-1	-1	-1	-1	+1	+1	10820
+1	-1	+1	-1	-1	+1	-1	+1	6300
+1	+1	1	-1	+1	-1	-1	-1	11300
+1	-1	-1	+1	1	-1	-1	+1	9200
+1	+1	-1	+1	-1	+1	-1	-1	12270
+1	-1	+1	1	-1	-1	+1	-1	6650
+1	+1	1	+1	1	+1	1	+1	12350
77640	15840	-4440	3300	5560	1700	-500	-300	**Total**
9705	1980	-555	412.5	695	212.5	-62.5	-37.5	**Total/8**
	3920400	308025	170156.25	483025	45156.25	3906.25	1406.25	$\boldsymbol{SQ}$
	62726400	4928400	2722500	7728400	722500	62500	22500	$\boldsymbol{SQ \cdot 8}$
Effect	79.49%	6.25%	3.45%	9.79%	0.92%	0.08%	0.03%	

10.4 Evaluation of *I-GNG-Q*

This section evaluates *I-GNG-Q*'s performance in the mountain car domain from Section 10.1.2 if we change one parameter while keeping the remaining parameters at the values of the basic configuration described in Section 10.2. For each variation we analyze the influence on the average number of steps needed to solve the task as well as the average sizes of the approximations. Once again, we underline the value of the basic configuration. Section 10.4.4 analyzes the effects of *I-GNG-Q*'s parameter on different metrics as well as possible interdependencies between these values.

10.4.1 Influence of the Insertion Delay λ_{insert}

Figure 10.10 shows the average number of steps for $\lambda_{insert} \in \{10, \underline{40}, 80, 200, 400\}$. The influence is similar to that in *GNG-Q*: Once again, the setting with $\lambda_{insert} = 10$ finds a good solution rather quickly and then remains relatively stable. Nevertheless, the final result of all other settings except $\lambda_{insert} = 400$ are better than the final result for $\lambda_{insert} = 10$. With increasing λ_{insert}, the time until the performance reaches its final level increases. This is similar to the results we saw before in the analysis of *GNG-Q*: The smaller λ_{insert} the faster new neurons are added which results in a finer approximation. In the case of *I-GNG-Q* more neurons mean more prototype Q-vectors that potentially influence the value of for a given state. Thus, for small values of λ_{insert} *I-GNG-Q* can learn a very nuanced policy even in the beginning.

This fact is also reflected in Figure 10.11 where the average number of neurons is plotted over the episodes. It can be seen that $\lambda_{insert} = 10$ results in the highest number of neurons. With increasing λ_{insert}, the average number of neurons decreases. Obviously, an increasing λ_{insert} delays the time needed to fix the size of the approximation.

Comparing the results to the ones of *GNG-Q*, it can be seen that *I-GNG-Q* is more sensible to the choice of λ_{insert} than *GNG-Q*: For the same values of λ_{insert} *GNG-Q* needed 14–54 neurons while *I-GNG-Q* needed 15–94 neurons (for $\lambda_{insert} = 400$ and $\lambda_{insert} = 10$). Nevertheless, *I-GNG-Q* is able to produce a more stable behavior earlier than *GNG-Q*.

After 20,000 episodes, the average number of steps range from 60.32 with $\lambda_{insert} = 80$ to 71.84 with $\lambda_{insert} = 400$. The final average number of neurons is between 15 for $\lambda_{insert} = 400$ and 94 for $\lambda_{insert} = 10$.

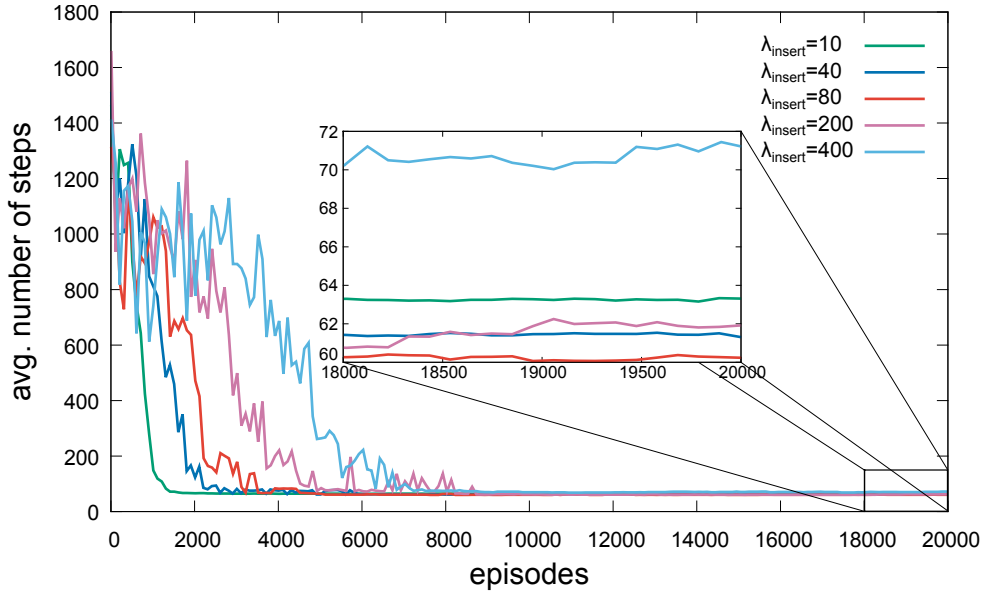


Figure 10.10: Average number of steps for *I-GNG-Q* with insertion delay $\lambda_{insert} \in \{10, \underline{40}, 80, 200, 400\}$ in the mountain car task.

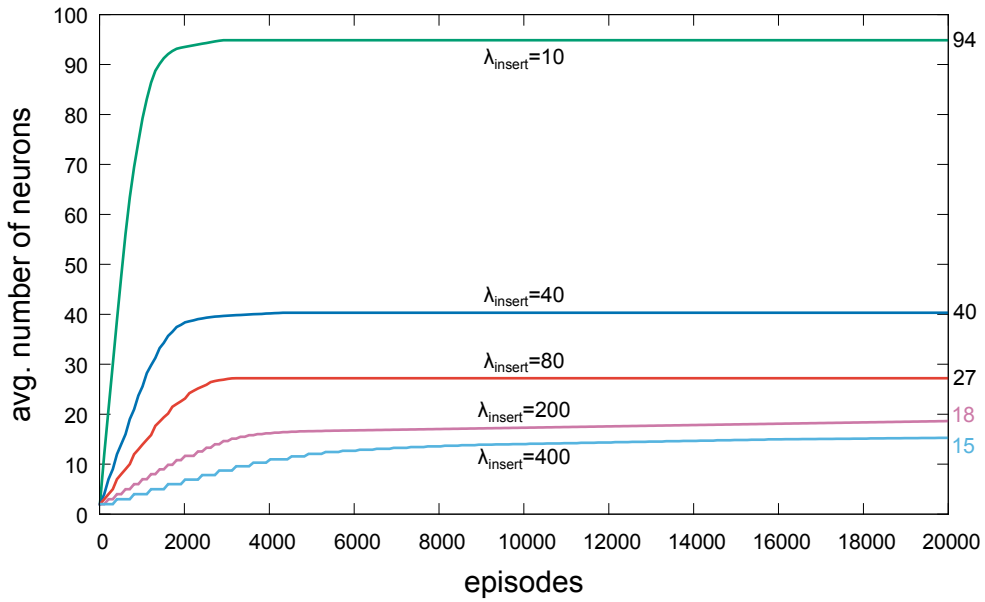


Figure 10.11: Average number of neurons for *I-GNG-Q* with insertion delay $\lambda_{insert} \in \{10, \underline{40}, 80, 200, 400\}$ in the mountain car task.

10.4.2 Influence of the Distance Exponent p

In Figure 10.12, the average number of steps for $p \in \{2, \underline{3}, 4, 5\}$ is shown. No clear influence on the speed of finding a good approximation can be reported. Nevertheless, higher values for p result in a more stable behavior. The quality of the solution also depends on the number of p: The higher p the smaller the average number of steps needed to solve the task. This can be explained by taking into account the purpose of the exponent: As explained in Section 8.4.2 and depicted in Figure 8.4, the exponent influences the smoothness of the interpolation between the sampling points (i.e. the neurons). Thus, the agent can develop better behaviors in certain areas of the environment.

In addition to this, Figure 10.13 shows the average numbers of neurons for the different settings. In general it holds that the higher p the smaller the average number of neurons although the effect is quite small: The smallest number of neurons is achieved with $p = 5$ where 39.01 neurons are needed. The maximal number is reached with $p = 2$ and 41 neurons on average. Thus, the sizes of approximations for varying the exponent p are quite close.

In the final episode, the average number of steps is between 59.21 for $p = 5$ and 66.83 for $p = 2$. The final average number of neurons varies from 39.04 for $p = 5$ to 41.69 for $p = 2$. Thus, the impact on the size of the approximation is quite limited.

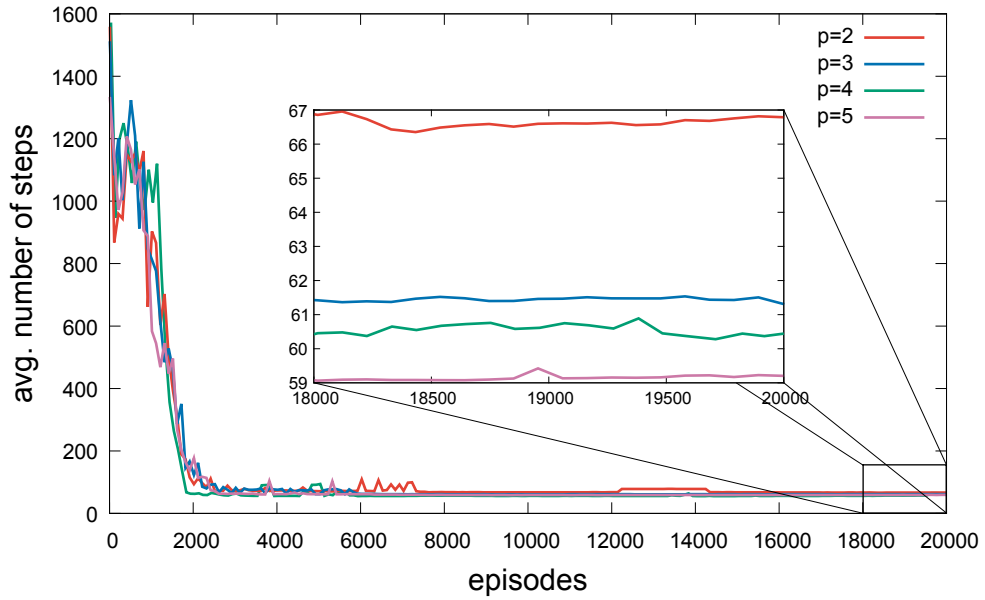


Figure 10.12: Average number of steps for *I-GNG-Q* with distance exponent $p \in \{2, \underline{3}, 4, 5\}$ in the mountain car task.

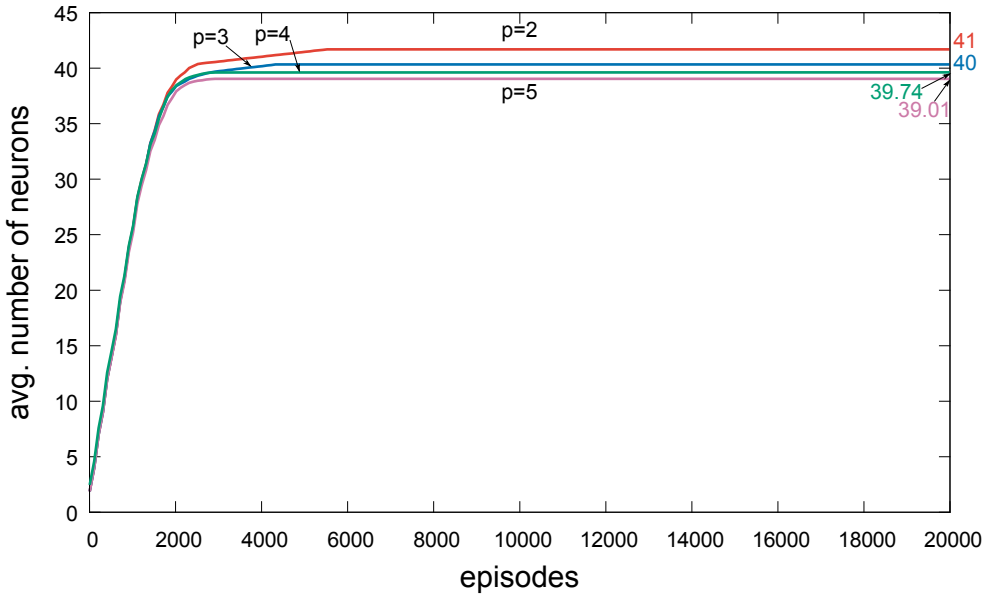


Figure 10.13: Average number of neurons for *I-GNG-Q* with distance exponent $p \in \{2, \underline{3}, 4, 5\}$ in the mountain car task.

10.4.3 Influence of the Number of Interpolation Bases k

As final parameter for *I-GNG-Q* we analyze the influence of the number of interpolation bases $k \in \{2, \underline{3}, 4, 5\}$. This parameter controls the number of prototype Q-vectors that are considered for the computation of the Q-value for a particular state, i.e. the prototype Q-vectors of the k nearest neurons are inversely weighted by the distances between the neurons and the state to make up the approximated Q-value.

Figure 10.14 shows the number of steps needed to solve the mountain car task. All four settings find a good behavior roughly equally fast and all converge to their finale performance after around 6000 episodes. From then on, all settings remain stable. Additionally, the final performance for all values of k are close: The best performing value ($k = 2$) needs 60.5 steps on average while the worst performing value ($k = 5$) needs approximately 62.3 steps on average. Thus, the higher k the higher the average number of steps. An explanation for this behavior is the following: While it is advantageous to compute the approximated Q-vector not only based on one prototype Q-vector, relying on too many prototype may be problematic as some of these additional neurons might be within the k nearest neurons but may be "responsible" for different portions of the state space. Obviously, the performance of the *I-GNG-Q* approach in different tasks might benefit from fine tuning this parameter.

The average number of neurons needed for different values of k are depicted in Figure 10.15. Once again, the difference is rather marginal: The smallest average number of neurons (39.21) is achieved with $k = 5$ and the highest average number of neurons (41.09) is achieved with $k = 2$. Thus, it can be assumed that the higher k, the smaller the final resulting approximation. Nevertheless, $k = 4$ converges to the final number of neurons later than the other settings.

After 20,000 episodes, the differences between the performances are close: For $k = 2$ the average number of steps is minimal (60.27) while for $k = 5$ the highest average number of steps (62.08) is reached. The final average number of neurons is between 39.21 for $k = 5$ and 41.09 for $k = 2$. Thus, neither the solution quality nor the size of the approximation depends seriously on the number of interpolation bases k.

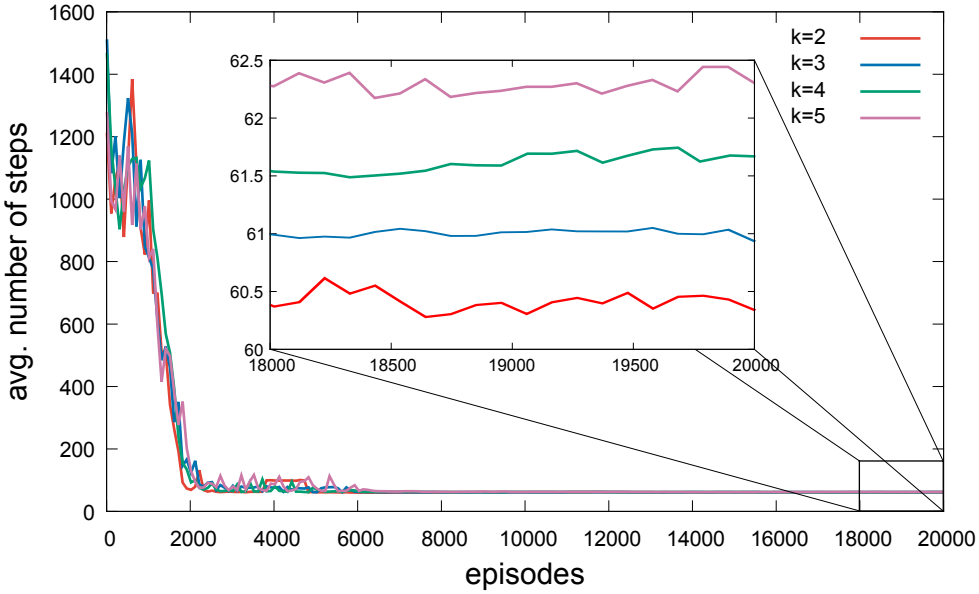


Figure 10.14: Average number of steps for *I-GNG-Q* with number of interpolation bases $k \in \{2, \underline{3}, 4, 5\}$ in the mountain car task.

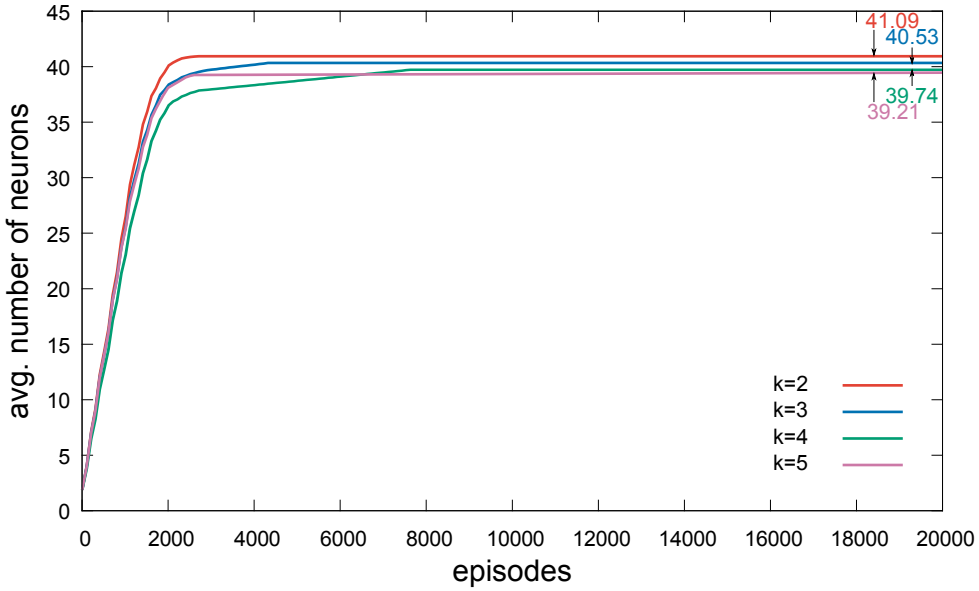


Figure 10.15: Average number of neurons for *I-GNG-Q* with number of interpolation bases $k \in \{2, \underline{3}, 4, 5\}$ in the mountain car task.

10.4.4 Interdependencies between Parameters

This section investigates the influences of relevant parameters on *I-GNG-Q*'s performance as well as their interdependencies. We use the 2^k factorial design as described in Section 10.3.4 on the parameters insertion delay λ_{insert}, number of interpolation bases k, and the exponet of the inverse distance weighting function p. For each of these factors, we specify low and high levels as given in Table 10.5.

As before, we investigate the three response variables which should be as small as possible:

- average number of steps in the last episode
- number of episodes needed to first find a solution that differs at most 5% from the best result in the complete run
- size of the approximation

Influence on the Solution Quality

Table 10.6 shows that the largest influence on the solution quality in the last episode is given by the exponent p of the inverse distance weighting function. Its effect is 45.72%. The second largest effect (31.62%) is due to the number of interpolation bases k. Interestingly, the influence of the insertion delay λ_{insert} is here only the fourth important factor while in *GNG-Q*, this was the parameter that had the largest influence on all considered response variables.

The influence of p is negative, i.e. large values of p result in a small number of steps. Contrary to this, the influence of k is positive which means that the higher the number of neurons considered for the computation of the Q-values, the higher the average number of steps in the last episode.

Influence on the Size of the Approximation

As in *GNG-Q*, the insertion delay λ_{insert} has the largest effect on the size of the approximation: Here, this effect is 84.56% and it is negative, i.e. the larger the value of λ_{insert}, the smaller the resulting approximation. The second large influence (12.56%) has the number k of neurons that are included in the computation of each approximated Q-vector. This effect is positive which means that a larger value of k results in a larger approximation.

Influence on the Speed of Finding Good Approximations

In Table 10.8 we can see, that the insertion delay λ_{insert} has the largest effect on the first time a good solution is found. This is similar to the results in *GNG-Q*

Table 10.5: Levels for each factor in the 2^k factorial design for *I-GNG-Q*.

factor	low value (-1)	high value (+1)
λ_{insert}	10	200
k	2	5
p	2	5

although there, the effect is larger: For *I-GNG-Q* the effect of λ_{insert} is 55.02% and for *GNG-Q* the effect of λ_{insert} is 79.49%. Both effects are positive which means that a smaller value of λ_{insert} results in finding an good approximation earlier. When we compare the values of the response variable for *GNG-Q* and *I-GNG-Q*, we can see that *I-GNG-Q* is generally faster than *GNG-Q*.

The number k of neurons that are included in the computation of the Q-values has the second largest effect on the speed of finding a good approximation with *I-GNG-Q*: The influence is 15.89% and the third largest influence is the combination of λ_{insert} and k. The parameter k has an negative effect on the number of episodes needed to find a proper approximation. This means that the more neurons are included in the computation, the faster a good approximation is found.

Table 10.6: Results of the factorial design for the performance in the last episode with *I-GNG-Q*. The values in the result column are the final average number of steps needed to solve the task.

	λ_{insert}	k	p					
I	**A**	**B**	**C**	**AB**	**AC**	**BC**	**ABC**	**Result**
+1	-1	-1	-1	+1	+1	+1	-1	62.19
+1	+1	-1	-1	-1	-1	+1	+1	58.99
+1	-1	+1	-1	-1	+1	-1	+1	64.04
+1	+1	+1	-1	+1	-1	-1	-1	61.64
+1	-1	-1	+1	+1	-1	-1	+1	57.52
+1	+1	-1	+1	-1	+1	-1	-1	57.72
+1	-1	+1	+1	-1	-1	+1	-1	60.14
+1	+1	+1	+1	+1	+1	+1	+1	60.08
482.32	-5.46	9.48	-11.4	0.54	5.74	0.48	-1.06	**Total**
60.29	-0.68	1.19	-1.43	0.07	0.72	0.06	-0.13	**Total/8**
	0.47	1.40	2.03	0.01	0.51	0.01	0.02	$\boldsymbol{SQ}$
	7.45	22.47	32.49	0.07	8.24	0.06	0.28	$\boldsymbol{SQ \cdot 8}$
Effect	10.49%	31.62%	45.72%	0.10%	11.59%	0.08%	0.40%	

Table 10.7: Results of the factorial design for the size of the approximation with *I-GNG-Q*. The values in the result column are the average number of neurons in the final episode.

	λ_{insert}	k	p					
I	**A**	**B**	**C**	**AB**	**AC**	**BC**	**ABC**	**Result**
+1	-1	-1	-1	+1	+1	+1	-1	69.12
+1	+1	-1	-1	-1	-1	+1	+1	39.78
+1	-1	+1	-1	-1	+1	-1	+1	73.48
+1	+1	+1	-1	+1	-1	-1	-1	51.19
+1	-1	-1	+1	+1	-1	-1	+1	68.6
+1	+1	-1	+1	-1	+1	-1	-1	39.06
+1	-1	+1	+1	-1	-1	+1	-1	75.84
+1	+1	+1	+1	+1	+1	+1	+1	55.29
472.36	-101.72	39.24	5.22	16.04	1.54	7.7	1.94	**Total**
59.05	-12.72	4.91	0.65	2.01	0.19	0.96	0.24	**Total/8**
	161.67	24.06	0.43	4.02	0.04	0.93	0.06	$\boldsymbol{SQ}$
	2586.74	384.94	6.8	64.32	0.59	14.82	0.94	$\boldsymbol{SQ \cdot 8}$
Effect	84.56%	12.58%	0.22%	2.10%	0.02%	0.48%	0.03%	

Table 10.8: Results of the factorial design for the number of episodes until *I-GNG-Q* finds the first time a solution that is at most 5% worse than the best solution in all runs. The values in the result columns are the number of episodes after which the average behavior reaches the required quality.

	λ_{insert}	k	p					
I	**A**	**B**	**C**	**AB**	**AC**	**BC**	**ABC**	**Result**
+1	-1	-1	-1	+1	+1	+1	-1	2960
+1	+1	-1	-1	-1	-1	+1	+1	4530
+1	-1	+1	-1	-1	+1	-1	+1	2930
+1	+1	+1	-1	+1	-1	-1	-1	3450
+1	-1	-1	+1	+1	-1	-1	+1	3410
+1	+1	-1	+1	-1	+1	-1	-1	4180
+1	-1	+1	+1	-1	-1	+1	-1	3420
+1	+1	+1	+1	+1	+1	+1	+1	3630
28510	3070	-1650	770	-1610	-1110	570	490	**Total**
3563.75	383.75	-206.25	96.25	-201.25	-138.75	71.25	61.25	**Total/8**
	147264.06	42539.06	9264.06	40501.56	19251.56	5076.56	3751.56	$\boldsymbol{SQ}$
	2356225	680625	148225	648025	308025	81225	60025	$\boldsymbol{SQ \cdot 8}$
Effect	55.02%	15.89%	3.46%	15.13%	7.19%	1.90%	1.40%	

10.4.5 Results *I-GNG-Q*

We saw again, that larger values of the insertion delay λ_{insert} causes *I-GNG-Q* to need more time for finding a good solution and that smaller values for λ_{insert} results in larger approximations in terms of the number of neurons. In contrast to *GNG-Q*, the largest influence on the final performance is not caused by the insertion delay λ_{insert} (except for $\lambda_{insert} = 400$, the final performances for varying λ_{insert} are quite close).

Here, the distance exponent p has the largest influence on the final performance while the insertion delay λ_{insert} has the fourth largest influence after the number of interpolation bases and the combined influence of λ_{insert} and p. In general it holds that the higher p the smaller the average number of steps needed to solve the task and the smaller the average number of neurons although the latter effect is quite small.

For the number of interpolation bases k, we saw that the higher k the higher the average number of steps which can be explained by the fact that then possibly too many neurons influence the Q-value for a given state. In addition, the higher k, the smaller the final approximation although this influence is very small.

Once again, the largest influence on the size of the approximation and the time to find a good solution is caused by the insertion delay λ_{insert}.

Comparing *GNG-Q* and *I-GNG-Q*, it can be seen that *I-GNG-Q* is able to produce a more stable behavior earlier than *GNG-Q* (cf. Table 10.8 and Table 10.4).

10.5 Comparison to Other Approaches

In this section we compare the results of *GNG-Q* and *I-GNG-Q* with each other as well as with results of other approaches from literature.

Table 10.9 shows a comparison of the results obtained after 20,000 episodes by *GNG-Q* and *I-GNG-Q* in the mountain car task. We report results for

- the base configurations as described in Section 10.2.
- the experiments for the evaluations of parameter changes (Sections 10.3.1–10.3.3 and Sections 10.4.1–10.4.3). We denote by base($par = val$) the base configuration with parameter *par* set to *val* while leaving all other parameters unchanged.
- the experiments for the factorial designs (Section 10.3.4 and Section 10.4.4).

For each of the latter two groups of experiments, we selected the parameter settings that led to the

- minimal average number of steps
- maximal average number of steps
- minimal average number of neurons
- maximal average number of neurons

over all runs with this parameter setting (i.e. one experiment) in the final episode. For every such set of runs, we report the best and the worst values achieved by one run for the final number of steps and the final number of neurons (in each case columns *Min* and *Max*). Additionally, for each experiment we analyze the mean, standard deviation, and the radii for the 95% confidence intervals for the average number of steps as well as for the average number neurons (columns *Mean*, *Std. Dev.*, *CI*). We mark the minimal average value of one group by underlining and the maximal value of one group with an overbar.

For example, the parameter setting base($\lambda_{insert} = 10$) hat the minimal average number of steps (55.72) and simultaneously the maximal number of neurons (54.42) from all the experiments in Sections 10.3.1–10.3.3.

To clarify, the columns *Min* and *Max* for the steps (similar for the number of neurons) needed are the final average number of steps needed by the agent to reach the goal from 100 randomly chosen start states (cf. Section 10.1.1). The columns *Mean*, *Std. Dev.*, *CI* are statistics over all final average numbers of steps of all runs for this setting (e.g. *Mean* is the average of all average numbers of steps for this setting).

The best average final number of steps for each approach is marked bold: 54.68 steps on average for *GNG-Q* with $\lambda_{insert} = 10, age_{max} = 500, \epsilon_b = 0.01$ and 57.52 steps on average for *I-GNG-Q* with $\lambda_{insert} = 10, k = 2, p = 5$.

We compare our results to the following approximation approaches from literature that used the same evaluation approach as in this work: In (da Motta Salles Barreto and Anderson, 2008), average numbers of steps of 52 (with nine RBF units) and 76 (with four RBF units) are reported for their adaptive RBF approximation. Their approach needs around 50,000 episodes to reach a stable solution. When using 9 RBF units, the approach finds a solution whose quality is close to the final performance after roughly 25,000 episodes.

Table 10.9: Comparison of the best and the worst parameter settings of *GNG-Q* and *I-GNG-Q* in terms of the average number of steps as well as the number of neurons in the last episode: Minimal and maximal values of all runs for that parameter setting, mean with standard deviation and radii of the 95% confidence interval for all runs of this parameter setting for both measures.

		Steps needed					**Number of neurons**				
	Parameter setting	Min	Max	Mean	Std. Dev.	CI	Min	Max	Mean	Std. Dev.	CI
GNG-Q	base	46.88	75.97	58.72	7.14	2.62	5	94	32.06	22.32	8.19
	base($\lambda_{insert} = 10$)	41.35	88.18	55.72	7.79	2.43	19	123	54.42	23.57	7.35
	base($\lambda_{insert} = 400$)	43.29	101.86	62.82	13.08	5.18	3	49	13.89	12.23	4.84
	$\lambda_{insert} = 400, age_{max} = 50, \epsilon_b = 0.01$	46.72	80.33	61.05	8.89	4.57	3	45	11.88	10.42	5.36
	$\lambda_{insert} = 10, age_{max} = 500, \epsilon_b = 0.01$	43.91	77.17	**54.68**	6.90	1.92	15	158	48.21	29.24	8.14
	$\lambda_{insert} = 10, age_{max} = 50, \epsilon_b = 0.1$	45.76	93.80	57.92	8.78	2.55	15	175	59.00	32.18	9.34
	$\lambda_{insert} = 400, age_{max} = 500, \epsilon_b = 0.01$	44.04	76.43	60.65	9.61	3.97	3	26	8.76	5.88	2.43
I-GNG-Q	base	41.95	102.23	60.90	11.22	3.06	20	107	40.33	14.28	3.90
	base($\lambda_{insert} = 10$)	47.24	92.19	63.32	9.62	2.51	31	154	94.28	29.08	7.90
	base($\lambda_{insert} = 400$)	48.10	139.60	71.84	22.12	7.97	6	51	15.28	9.21	3.32
	base($p = 5$)	46.45	89.81	59.21	8.94	2.66	12	71	39.04	12.31	3.66
	$\lambda_{insert} = 10, k = 5, p = 2$	45.15	89.70	64.04	11.09	3.75	54	95	73.48	11.39	3.85
	$\lambda_{insert} = 10, k = 2, p = 5$	46.55	81.21	**57.52**	8.18	2.43	50	97	68.61	10.59	3.14
	$\lambda_{insert} = 10, k = 5, p = 5$	48.46	76.60	60.14	6.97	2.09	34	150	75.84	24.03	7.21
	$\lambda_{insert} = 200, k = 2, p = 5$	44.91	76.37	57.72	7.88	2.42	12	64	39.06	10.76	3.31

Whiteson and Stone (2006) also report a number of approximately 52 steps for their adaptive tile coding approximation but because of a high sample complexity (Whiteson and Stone, 2006), this approach needs much more than 100,000 episodes to come to stable results. Lin and Wright (2010) evaluated their evolutionary tile coding approximation and report an average of around 50 steps to solve the task.

Especially the latter two approaches have the drawback of using the performance of reinforcement learning as fitness function. Thus, each of those approaches needs *number of generations* times *size of the population* executions of the RL task. Each of these executions may of course need several episodes of the RL task in order to finish learning.

Nevertheless, Lin and Wright (2010) report that their approach is able to find a state-space approximation consisting only of two abstract states which is highly efficient. For *GNG-Q* some of the experiments with insertion delay $\lambda_{insert} = 400$ included runs with a final number of three neurons (i.e. three abstract states). Each of those were able to solve the task in 44.04–48.64 steps on average which is quite remarkable.

We also evaluated a grid approximation for standard Q-Learning on the task where we experimented with different approximation sizes and allowed Q-Learning to run for 5,000,000 episodes. The best setting had 50×50 cells and needed an average of 43.14 steps on the 100 start states for evaluation. The best results of *GNG-Q* and *I-GNG-Q* were able to achieve better results (the lowest average number of steps for *GNG-Q* in the above set was 41.35 and 41.95 for *I-GNG-Q*. Thus, both approaches are able to compute efficient approximations during learning. Nevertheless, both measures (average number of steps and the size of the approximation) depend on the choice of the parameter. In addition to this, the initial placement of the first neurons may lead to results with inferior policies. This can be seen by the maximal values given in Table 10.9. Nevertheless, the values of the standard deviation and the radii of the 95% confidence intervals suggest that these high values are outliers and that the average performance is usually decent.

Comparing *GNG-Q* and *I-GNG-Q* it can be seen that *GNG-Q* is able to find slightly better and smaller approximations than *I-GNG-Q*. Nevertheless, *I-GNG-Q* is usually faster in finding good and stable approximations as can be seen in Table 10.4 and Table 10.8.

To sum up, the results of *GNG-Q* and *I-GNG-Q* are well comparable to similar approaches. Especially the approximations derived from evolutionary algorithms are much more computationally expensive than the adaptive approaches presented in this thesis.

10.6 Advantages of Adaptive Approximations in Unknown Environments

To stress the advantages of adaptive approximation methods, we evaluated our approaches in the following *random world* environment:

We adapt the continuous world from (Boyan and Moore, 1994) that was employed in (Baumann and Kleine Büning, 2011): The agent has to learn the shortest path from all positions to a target. The agent can perform actions that take it one step in any of the four cardinal directions and the state space is $S = \{(x, y) \mid x, y \in [0, 1]^2 \subset \mathbb{R}^2\}$.

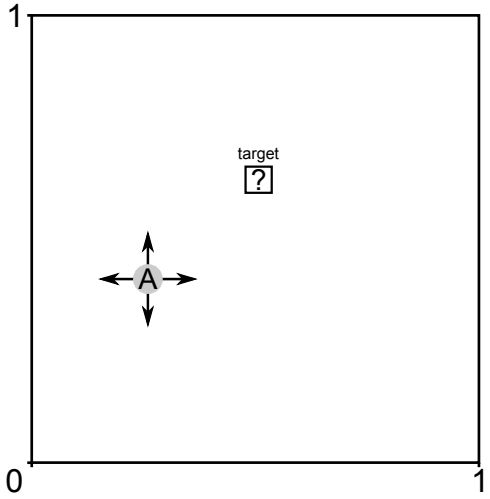


Figure 10.16: The world with a randomly placed target.

During creation of the world, the target is randomly placed inside the pane and fixed at this position until the end (cf. Figure 10.16). Thus, every instance looks different but the layout of the world is fixed during each run. The size of the target is equal to the size of the agent's step size s_{step}.

At the beginning of each episode, the agent is randomly placed inside the world and if it tries to leave the world in any dimension, it is positioned on the border of this dimension. For the action that leads the agent to the target, a reward of 0 is awarded, for all other action, the reward is -0.5 which motivates the agent to find paths as short as possible. Unless otherwise stated, we use $s_{step} = 0.05$ (i.e. the agent needs $1/s_{step} = 20$ steps to go from one border to the opposing border) and if the agent did not reach the target after a given amount of steps (for a stepsize of 0.05 we set this threshold to 400), the trial is stopped.

The challenge in this task is that even if one would know the underlying MDP, no information could be drawn from it: The target is positioned at the creation of the task and then fixated. Figuratively, the agent is put in the environment and then the position of the target is decided. Thus, it would be very hard to "guess" a good hand-made discretization of the state space (cf. Section 7.1.3 and especially Figure 7.2). Solutions to such tasks are especially interesting as often no knowledge is available about the state space or the related value function.

To compare the approaches, we use a grid approximation for Q-Learning such that the agent transitions into a new cell every time it performs a step. As the target can be located anywhere in the plane, situations similar to the one depicted in Figure 7.2(b) can occur (i.e. some parts of a cell may be overlapped by the target while others consists of non-goal states. We tested two grid sizes: For the first we divided each dimension in $\frac{1}{s_{step}}$ intervals and for the second we divided each dimension in $\frac{1}{2 \cdot s_{step}}$ intervals. In our scenario this results in 20×20, and 40×40 cells, respectively. Especially for the first case, it may happen quite often that the target is not completely inside only one cell of the approximation (cf. Figure 7.2(b) while for the second case, the resolution of the grid will be too fine (cf. Figure 7.2(c)).

For *GNG-Q* and *I-GNG-Q* we used our base configurations from Section 10.2

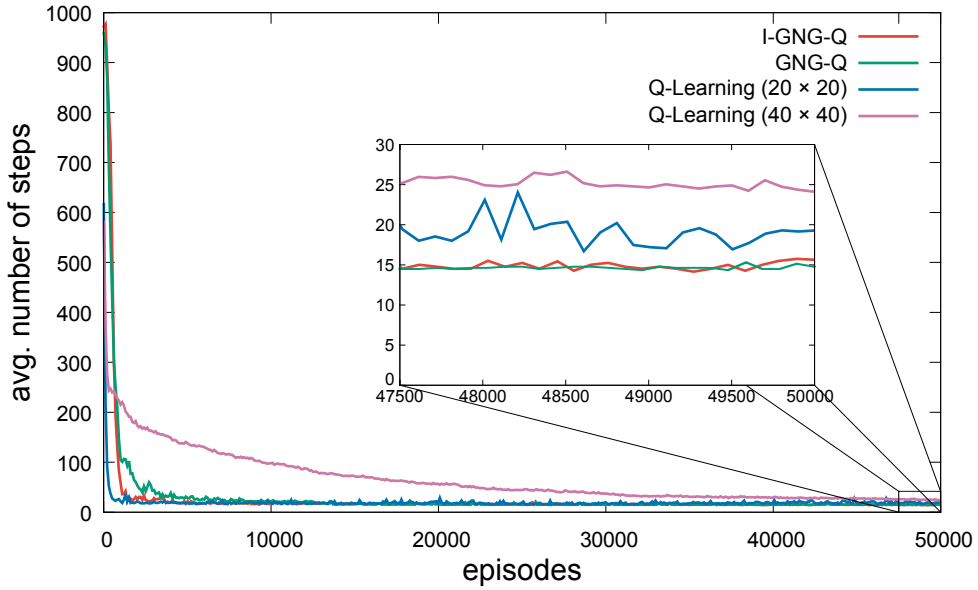


Figure 10.17: Average number of steps with *GNG-Q*, *I-GNG-Q*, and Q-Learning (20×20 and 40×40 grid) in the random world task.

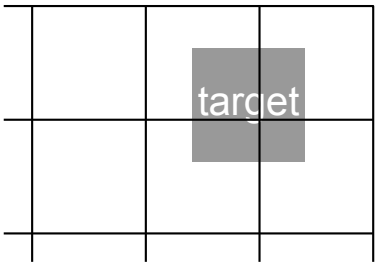


Figure 10.18: Additional problem when using a fixed grid approximation: Parts of the target may be in several cells.

on the same task. In this investigation, we allowed every approach to run for 50,000 episodes.

Figure 10.17 shows how the performance for *GNG-Q*, *I-GNG-Q* and the two grid discretizations for Q-Learning evolve over time. The Q-Learning approach with 20×20 cells is the fastest to find a good policy on average. This is due to the fact that this setup already starts with a quite fine approximation which allows the agent to often find the target somehow but maybe not in the best way. Thus, the learned behavior for this approach is unstable and its final policy needs close to 20 steps on average to reach the goal (cf. the zoomed area in Figure 10.17). In Figure 10.18 one explanation for this is shown: As the target can be anywhere in the environment, it may happen that it is partially contained in several cells. Thus, for such cells the agent cannot know for certain whether it is in the target.

The 40×40 grid approximation for Q-Learning results in a more stable behavior but it is much slower and the average number of steps is around 25. The finer approximation slightly reduces the problem depicted in Figure 10.18 but still faces the drawback of being too fine huge areas of the state space as depicted in Figure 7.2(c).

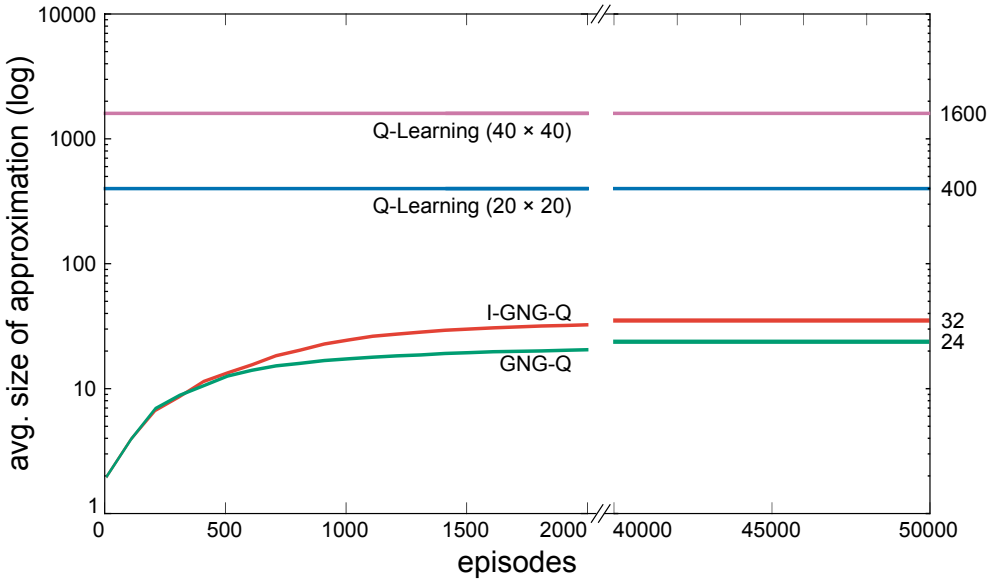


Figure 10.19: Size of the approximation (logarithmic scale) in number of needed neurons with *GNG-Q* and *I-GNG-Q* as well as in number of states for Q-Learning in the random world task.

Additionally, the larger state space results in a longer time to find a reasonable policy.

I-GNG-Q and *GNG-Q* find good approximations slightly slower than the 20×20 grid approximation for Q-Learning. Nevertheless the final results are better: Here, only around 15 steps (14.60 for *GNG-Q* and 14.75 for *I-GNG-Q*) on average are needed to reach the target. Additionally, the more stable behavior indicates that the automatically found approximations are well suited and allow the agent to efficiently and effectively store its behavior. The last argument can especially be seen in Figure 10.19 where the size of the approximations are compared. *I-GNG-Q* and *GNG-Q* need 32 and 24 neurons on average to store the learned behavior while the Q-Learning approaches need 400 states for the 20×20 and 1600 states for the 40×40 approximation. Thus, starting with a coarse approximation and inserting new neurons (i.e. abstract states) as it is done in *GNG-Q* and *I-GNG-Q* offers a means to find compact approximations in (possibly) unknown environments. Both approaches developed in this thesis offer the most stable behaviors, the smallest number of steps needed, as well as the most compact abstract state space.

For comparison, we employed the following baseline approach: We employ Q-Learning on a predefined uniform discretization with a similar (abstract) state-space size as computed by *GNG-Q* and *I-GNG-Q* after 50,000 episodes. Without any knowledge on the task at hand, it is advisable to use the same resolution for each dimension. Of course, this may not be optimal as some dimensions might require a finer resolution than others. As these approaches need 24 and 32 neurons on average we split each dimension in $\lceil \sqrt{24} \rceil = 5$ intervals for the first baseline and in $\lceil \sqrt{32} \rceil = 6$ intervals for the second baseline. Figure 10.20 shows the average number of steps needed by the baselines: It can be seen that both approaches fail to converge to a stable behavior and that the average number of steps needed is quite high.

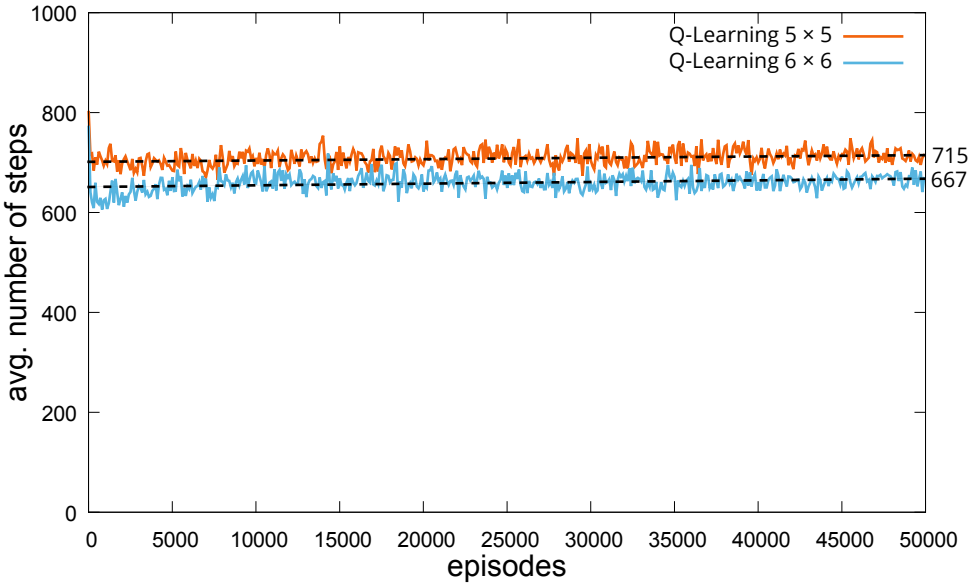


Figure 10.20: Comparison of the average number of steps for the baseline Q-Learning approaches in the random world task. The dashed lines are the trend lines for each approach.

Table 10.10: Comparison of the average success rates (i.e. how often the agent reached the goal) after 50,000 episodes with standard deviation (Std. Dev.) and radii of the 95% confidence interval (95% CI) for the considered approaches.

Approach	Avg. SR	Std. Dev.	95% CI
GNG-Q	0.9966	0.0148	0.0004
I-GNG-Q	0.9965	0.0077	0.0004
Q-Learning (20×20)	0.9933	0.0256	0.0001
Q-Learning (40×40)	0.9919	0.0190	0.0002
Q-Learning (5×5)	0.3161	0.2861	0.0009
Q-Learning (6×6)	0.3241	0.3286	0.0011

In Table 10.10 we report on the average success rates (i.e. how often the goal is reached on average) of all approaches. It can be seen that *GNG-Q* and *I-GNG-Q* have the highest success rates (0.9966 and 0.9965) and that the 95% confidence interval is tight. The 20×20 and the 40×40 approximation for Q-Learning have slightly lower success rates of 0.9933 and 0.9919, also with close endpoints of the 95% confidence interval. Additionally, the (sample) standard deviations of *I-GNG-Q*, *GNG-Q*, as well as of the 20×20 and the 40×40 approximations are quite small and thus, the success rates for each single run are quite similar. The baseline approaches perform really bad as the agent only has success rates of 31% and 32%.

Finally, we let the Q-Learning approaches complete 1,000,000 episodes to investigate if they improve their performances with longer training time. The baseline approaches cannot be improved by allowing the agent more time to learn. Obviously, the approximation only insufficiently captures the features of the environment. The 20×20 and the 40×40 approximations on the other hand slightly improve over

time: For the 20×20 grid, the average success rate increases by 0.0002 while the average number of steps remains the same. The finer 40×40 resolution results in a policy with an average number of steps of 14.69 after 1,000,000 episodes and is thus in between the performances of *GNG-Q* and *I-GNG-Q*. Additionally, the success rate increases to 0.9959. After around 500,000 episodes, the 40×40 grid approximation reaches a similar performance as *GNG-Q* and *I-GNG-Q*.

In scenarios where no information about the environment is available, it is hard to create a suitable approximation by hand. Especially for such situations, adaptive approximation schemes as the ones presented in this thesis are well suited to help the agent find efficient and compact representations of the state space. Although the 20×20 discretization for Q-Learning is faster to find a good policy in the beginning, *GNG-Q* and *I-GNG-Q* compute a better and more stable policy. Additionally, the approximations of our approaches are very compact: *GNG-Q* needs 24 neurons (i.e. abstract states) on average and *I-GNG-Q* uses 32 neurons on average. Grid approximations that divide each dimension in equally sized intervals are not able to compute useful strategies with the same number of abstract states.

10.7 Shepherding

This section analyzes the performance of reinforcement learning in the Shepherding task. We first compare the solution of the standard tabular Q-Learning as described in Section 2.2.7 and later we investigate how the approximation methods developed in this thesis perform.

10.7.1 Learning Shepherding with Q-Learning

In the following, we compare the solutions computed by the reinforcement learning approach (cf. Chapter 6) with the *GCC* algorithm from Section 5.2. Note, that the RL agent has no information about the goal for the task and it does not know the position of the target. It only knows its own position as well as the position of the sheep. During learning it interacts with the environment and based on the reward it learns after some time that it is valuable to reach certain states (i.e. the states in which the sheep is in the target). This is contrary to *GCC* where the agent needs explicit knowledge about the location of the target.

For Q-Learning we used a discount factor $\gamma = 0.95$, a decreasing learning rate $\alpha_t = \frac{1}{1+\text{visits}(s,a)}$ with $\text{visits}(s,a)$ counting the number of performing a in s, and varied the exploration probability $\varepsilon \in \{0.05, 0.1, 0.2, 0.4, 0.6\}$. The environment had 21×21 cells with the target in the center and the viewing ranges for the sheep were $r_{sheep} \in \{1, 2, 3, 4, 5\}$. We allowed Q-Learning to learn for 1,000,000 episodes for each parameter combination. Each combination was repeated 100 times with different random seeds resulting in 100 policies per combination.

We evaluated every RL policy π by comparing its performance with that of *GCC*: For each possible (i.e all states where the sheep is neither placed on the target nor directly positioned at a border, cf. Figure 10.22) start state s we computed the number of steps $\#steps_{GCC}(s)$ with *GCC*, the number of steps $\#steps^{\pi}_{QL}(s)$ needed

Table 10.11: Results of the comparison of *GCC* and Q-Learning: Mean values (with 95% confidence interval radii), standard deviation, minimal, and maximal values of the differences $\#steps^{\pi}_{diff}(s)$ as well as success rates (SR) of the Q-Learning policies for varying viewing ranges r_{sheep} of the sheep and varying exploration probabilities ε.

r_{sheep}	ε	Mean	Std. Dev.	Min	Max	SR
1	0.05	0.70 ± 0.002	3.29	-30	14	0.9997
	0.1	1.70 ± 0.001	2.79	-30	14	0.9998
	0.2	2.73 ± 0.001	2.11	-22	14	0.9999
	0.4	3.32 ± 0.001	1.69	-12	14	0.9999
	0.6	3.38 ± 0.001	1.67	-4	14	0.9999
2	0.05	2.88 ± 0.002	4.46	-24	24	0.9997
	0.1	4.05 ± 0.002	4.00	-28	24	0.9998
	0.2	5.26 ± 0.002	3.49	-18	24	0.9999
	0.4	6.02 ± 0.002	3.16	-10	24	0.9999
	0.6	6.13 ± 0.002	3.14	-4	24	0.9999
3	0.05	5.53 ± 0.003	5.98	-32	34	0.9997
	0.1	6.83 ± 0.003	5.55	-30	34	0.9999
	0.2	8.14 ± 0.003	5.07	-22	34	0.9999
	0.4	8.98 ± 0.002	4.64	-12	34	0.9999
	0.6	9.12 ± 0.002	4.64	-10	34	0.9999
4	0.05	8.35 ± 0.004	7.51	-34	44	0.9997
	0.1	9.79 ± 0.004	7.03	-30	44	0.9998
	0.2	11.23 ± 0.003	6.49	-28	44	0.9999
	0.4	12.11 ± 0.003	5.97	-14	44	0.9999
	0.6	12.29 ± 0.003	5.95	-8	44	0.9999
5	0.05	11.29 ± 0.004	8.76	-36	54	0.9998
	0.1	12.78 ± 0.004	8.29	-34	54	0.9998
	0.2	14.39 ± 0.004	7.65	-32	54	0.9999
	0.4	15.31 ± 0.004	7.02	-18	54	0.9999
	0.6	15.52 ± 0.004	6.98	-6	54	0.9999

by the investigated policy π, and analyzed the difference

$$\#steps^{\pi}_{diff}(s) = \#steps_{GCC}(s) - \#steps^{\pi}_{QL}(s)\,. \tag{10.1}$$

The difference $\#steps^{\pi}_{diff}(s)$ is positive if the number of steps needed by the policy is smaller (i.e. the policy is better) and negative if *GCC* needed less steps than Q-Learning.

In Table 10.11 we report the mean difference (with 95% confidence interval radii) for each combination of viewing range r_{sheep} and exploration probability ε (i.e. the mean difference of all policies for that combination evaluated on all allowed start states). In addition to this, we show the standard deviation, minimal, and maximal values of the differences $\#steps^{\pi}_{diff}(s)$ as well as success rates (SR) of the Q-Learning policies.

It turned out, that Q-Learning solved at least 99.97% (for $\varepsilon = 0.05$ and $r_{sheep} = 1$) of the start states and that the success ratio increased to 99.99% for increasing ε and fixed r_{sheep}. Additionally, the success ratio increased with increasing r_{sheep} and fixed

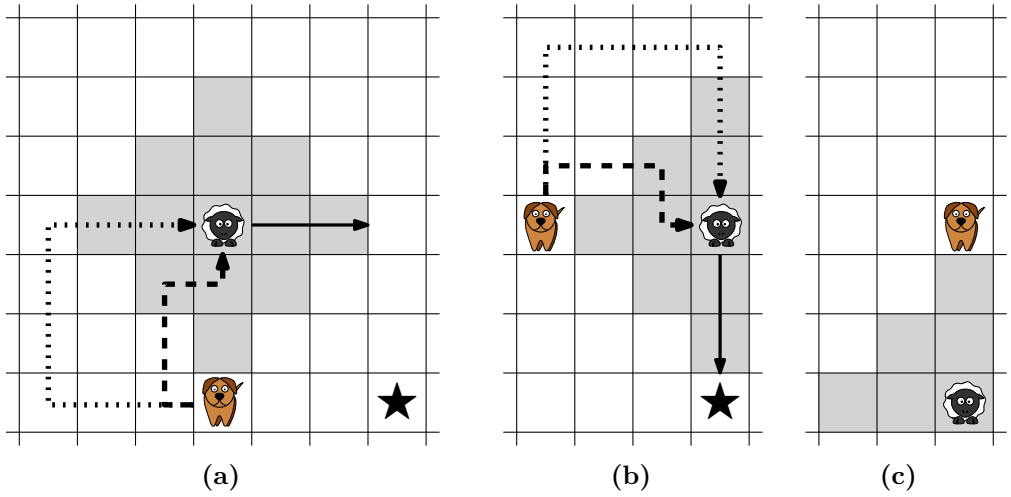


Figure 10.21: Comparison of the *GCC* strategy (dotted) and a learned behavior (dashed). *GCC* needs nine steps for each correction while the Q-Learning strategy only needs five by approaching the sheep at the edge of its viewing range instead of only approaching the corners.

ε. A closer inspection revealed that the reason for not solving some start states is due to the lack of visits which could be improved by e.g. implementing more elaborate exploration strategies for RL. With increasing ε (i.e. allowing the agent to explore more) and fixed viewing range, the mean value and the minimal value increases while the standard deviation decreases which results in better policies. Additionally, the width of the 95% confidence intervals decreases.

As the mean value of the differences is always positive, learning turned out to be superior. Especially for larger viewing ranges of the sheep, the learned policies are much more powerful than the solutions computed by *GCC*. Figure 10.21 shows an example policy for an instance with viewing range $r_{sheep} = 2$ for the sheep: It can be seen that the dog has learned a very efficient way of driving the sheep by approaching the edges of the viewing range instead of only steering the sheep from the corners as it is done in *GCC*.

In addition to this, we analyze the performance for different distributions of the start states. Instead of allowing the sheep to start from any admissible position, we consider different *settings* (cf. Figure 10.22):

- The first setting fulfills Assumption 5 (i.e. the sheep has a distance of $r_{sheep} + 1$ to the border which allows the dog to directly control the sheep without having to detach it first) and thus, the sheep is only placed on *inner* cells (but not on the target).
- The second setting investigates the behavior if Assumption 5 does not hold (i.e. the dog has to first detach the sheep), and thus, the sheep is placed on an *outer* cell but not on the border.

In both settings, the dog is allowed to start on any free cell.

We used the same procedure to evaluate the policies as described before with the restriction that the start states are generated according to the above settings. Again,

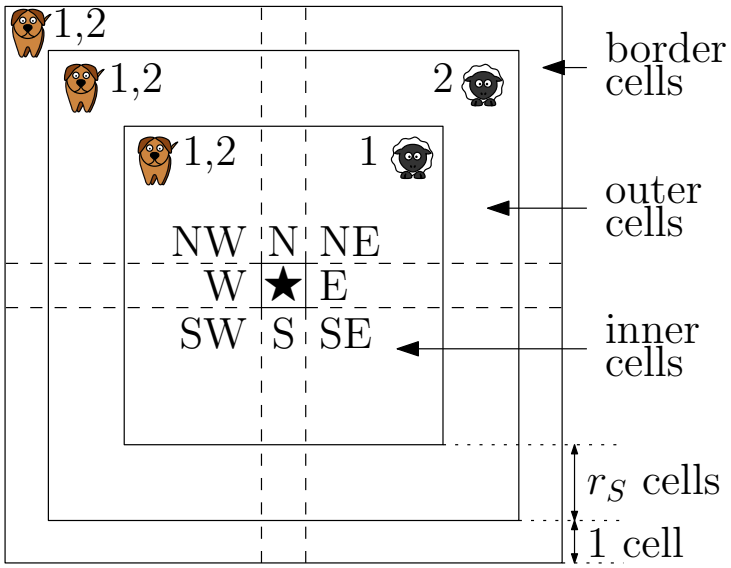


Figure 10.22: Schematic overview of the settings we used in the evaluation: The first setting has the sheep on an *inner cell* (but not on the target (★)) and the second setting has the sheep on an *outer cell*. In both settings, the dog is allowed to start on any free cell. The dashed lines indicate sectors that we use to explain the learned behaviors.

we computed the *average difference* over 100 repetitions for both settings and each combination of the sheep's viewing range $r_{sheep} \in \{1, 2, 3, 4, 5\}$ and the exploration probability $\varepsilon \in \{0.05, 0.1, 0.2, 0.4, 0.6\}$.

Table 10.12 summarizes the results of this analysis. Once again, the mean value of the differences is always positive, and thus, learning turned out to be superior. With a fixed viewing range and increasing ε (i.e. allowing the agent to explore more), the mean value and the minimal value increases while the standard deviation decreases which is evidence that the RL policies are better than *GCC*. For fixed values of ε, and increasing values for the viewing range, the RL policies become much better than the *GCC* solutions which can again be explained by the performant learned behavior (cf. Figure 10.21).

The average superiority of the learning agent over *GCC* in the *outer* settings is larger than in the *inner* settings: The states in the inner part of the environment are "easier" as the sheep needs not to be freed from the border. Thus, *GCC* already presents a powerful solution. Conversely, the situations with the sheep close to the border are harder to solve: Here, the shepherding agent has to figure out how to get behind the sheep in relation to the target. Especially such situations where the sheep has to be detached from the border are solved highly efficiently by the learning approach. The RL agent does not walk around the sheep's viewing range as it is done by *GCC* (depicted in Figure 5.9 or Figure 5.11) but instead uses the same behavior as shown in Figure 10.21.

The results show that increasing the viewing range of the sheep leads to an increase of the average difference and an increase of the standard deviation. Once again, the mean difference is larger in the *outer* settings. This allows the conclusion that learning is particularly beneficial for settings with larger viewing ranges and

Table 10.12: Results of the comparison of *GCC* and Q-Learning with different settings of the start states: Mean values (with 95% confidence radii), standard deviation, minimal, and maximal values of the differences $\#steps^{\pi}_{diff}(s)$ as well as success rates (SR) of the Q-Learning policies.

		Only inner cells					**Only outer cells**				
r_{sheep}	ε	Mean	Std. Dev.	Min	Max	SR	Mean	Std. Dev.	Min	Max	SR
	0.05	0.50 ± 0.002	2.99	-30	4	0.9997	1.48 ± 0.005	4.22	-30	14	0.9997
	0.1	1.47 ± 0.001	2.47	-30	4	0.9998	2.59 ± 0.004	3.68	-26	14	0.9998
1	0.2	2.45 ± 0.001	1.71	-22	4	0.9999	3.84 ± 0.003	3.03	-20	14	0.9999
	0.4	2.99 ± 0.001	1.14	-10	4	0.9999	4.66 ± 0.003	2.62	-12	14	0.9999
	0.6	3.04 ± 0.001	1.09	-2	4	0.9999	4.75 ± 0.003	2.62	-4	14	0.9999
	0.05	2.07 ± 0.003	3.98	-28	8	0.9997	4.20 ± 0.004	4.87	-28	24	0.9997
	0.1	3.22 ± 0.002	3.53	-28	8	0.9998	5.42 ± 0.004	4.34	-24	24	0.9998
2	0.2	4.38 ± 0.002	2.98	-18	8	0.9999	6.70 ± 0.003	3.77	-16	24	0.9999
	0.4	5.09 ± 0.002	2.46	-10	8	0.9999	7.56 ± 0.003	3.59	-6	24	0.9999
	0.6	5.17 ± 0.002	2.40	-4	8	0.9999	7.70 ± 0.003	3.51	-2	24	0.9999
	0.05	3.36 ± 0.004	5.58	-30	12	0.9997	7.43 ± 0.004	5.66	-32	34	0.9997
	0.1	4.64 ± 0.004	5.16	-26	12	0.9998	8.75 ± 0.004	5.15	-30	34	0.9998
3	0.2	5.96 ± 0.003	4.61	-22	12	0.9999	10.03 ± 0.003	4.69	-16	34	0.9999
	0.4	6.88 ± 0.003	3.84	-12	12	0.9999	10.80 ± 0.003	4.60	-8	34	0.9999
	0.6	7.02 ± 0.003	3.70	-10	12	0.9999	10.96 ± 0.003	4.51	-4	34	0.9999
	0.05	3.81 ± 0.006	7.40	-34	16	0.9998	10.60 ± 0.004	6.48	-32	44	0.9997
	0.1	5.39 ± 0.006	6.89	-30	16	0.9998	11.97 ± 0.004	6.00	-30	44	0.9998
4	0.2	7.02 ± 0.005	6.20	-28	16	0.9999	13.30 ± 0.003	5.56	-20	44	0.9999
	0.4	8.24 ± 0.004	5.02	-14	16	0.9999	14.02 ± 0.003	5.56	-8	44	0.9999
	0.6	8.45 ± 0.004	4.75	-8	16	0.9999	14.18 ± 0.004	5.46	-4	44	0.9999
	0.05	3.26 ± 0.010	9.00	-36	18	0.9999	13.54 ± 0.004	7.24	-30	54	0.9998
	0.1	5.00 ± 0.009	8.60	-34	18	0.9999	14.96 ± 0.004	6.75	-28	54	0.9998
5	0.2	7.16 ± 0.009	7.68	-32	18	0.9999	16.10 ± 0.004	6.38	-16	54	0.9999
	0.4	8.92 ± 0.006	5.78	-18	18	0.9999	16.41 ± 0.004	6.29	-8	54	0.9999
	0.6	9.26 ± 0.006	5.26	-6	18	1.0	17.27 ± 0.004	6.25	-2	54	0.9999

situations in which the sheep is close to the border. Nevertheless, the impact of the viewing range on the performance was already investigated in Section 5.3 where we showed that the upper bound on the lengths of solutions computed by *GCC* depends on the size of the viewing range: As an agent that uses *GCC* has to encircle the sheep completely in order to detach it from the wall the overhead in terms of needed steps grows with the size of the viewing range. Learning on the other hand is able to derive a powerful strategy without relying on domain knowledge. In fact, the learned strategies performed better even in the inner setting where the dog can immediately start controlling the sheep.

A more detailed investigation revealed that learning is substantially better than the *GCC* approach for situations in which the target is somewhere between the dog and sheep, e.g. the dog is south (in sectors *SW* or *SE*, cf. Figure 10.22) and the sheep is north (*NW* or *NE*) of the target. In situations where the sheep starts in one of the cardinal sectors (i.e. *N, E, S, W*) the performance is usually identical, i.e. $\#steps^{\pi}_{diff}(s) \approx 0$.

10.7.2 Learning Shepherding with *GNG-Q* and *I-GNG-Q*

We use the same setting and procedure as described before to evaluate *GNG-Q* and *I-GNG-Q* on the Shepherding task.

In initial experiments we found that neither *GNG-Q* nor *I-GNG-Q* showed a relevant influence of the exploration parameter ε on the performance. We thus used a medium value of $\varepsilon = 0.2$ for the following analysis. For *I-GNG-Q* we used two interpolation bases ($k = 2$) and a distance exponent of $p = 5$. *GNG-Q* was set to the base configuration described in Section 10.2 except for the insertion delay that was set to $\lambda_{insert} = 200$. This value for λ_{insert} was also used in *I-GNG-Q*.

Both approaches were allowed 500,000 episodes of training. It turned out, that *GNG-Q* needed around 50,000 episodes to compute stable solutions while *I-GNG-Q* found stable solutions after around 20,000 episodes. For both approaches, settings with smaller viewing ranges of the sheep were solved earlier. In comparison, the solutions computed by *I-GNG-Q* were in general more stable than those computed by *GNG-Q* which is consistent to the behavior we saw earlier.

Table 10.13 compares the statistics about the differences $\#steps^{\pi}_{diff}(s)$ for policies π computed by *GNG-Q* and *I-GNG-Q* for each viewing range $r_{sheep} \in \{1, 2, 3, 4, 5\}$: As before in Table 10.11 we report the mean difference (with 95% confidence interval radii), the standard deviation, minimal, and maximal values of the differences for each approach and viewing range, as well as the success rates (SR) of the policies computed by *GNG-Q* and *I-GNG-Q*.

The mean differences for both approaches increase while the radii of the 95% confidence intervals decrease with increasing viewing ranges as we have already seen in the analysis of the policies computed by standard Q-Learning. Additionally, the maximal value (which relates to the best solutions found) increases to the same values as in Q-Learning. The minimal values (which concerns the situations that are problematic for the considered policy) are in the same intervals we saw for Q-Learning in Table 10.11. Although the mean differences for viewing ranges $r_{sheep} \in \{1, 2, 3\}$ are in the same intervals as those for the policies computed by Q-Learning, the mean values for both *GNG-Q* and *I-GNG-Q* do not reach the same levels as Q-Learning

Table 10.13: Results of the comparison of *GCC* and our adaptive approximation schemes *GNG-Q* and *I-GNG-Q*: Mean values (with radii of the 95% confidence interval), standard deviation, minimal, and maximal values of the differences $\#steps^{\pi}_{diff}(s)$ as well as success rates (SR) of the policies computed by *GNG-Q* and *I-GNG-Q* for varying viewing ranges r_{sheep} of the sheep.

	r_{sheep}	Mean	Std. Dev.	Min	Max	SR
GNG-Q	1	3.21 ± 0.08	3.62	-14	14	0.9956
	2	6.11 ± 0.07	4.96	-18	24	0.9961
	3	6.69 ± 0.05	6.14	-12	34	0.9932
	4	8.12 ± 0.03	8.03	-32	44	0.9911
	5	8.77 ± 0.03	9.86	-12	54	0.9919
I-GNG-Q	1	2.68 ± 0.15	4.51	-12	14	0.9952
	2	5.26 ± 0.09	5.22	-13	24	0.9931
	3	5.70 ± 0.08	6.83	-19	34	0.9898
	4	7.44 ± 0.06	8.56	-25	44	0.9926
	5	8.18 ± 0.05	10.36	-34	54	0.9958

for $r_{sheep} \in \{4, 5\}$. Comparing *GNG-Q* and *I-GNG-Q* it can be seen that the mean differences for the policies computed by *GNG-Q* are better than that computed by *I-GNG-Q*. Nevertheless, the mean difference is always positive and thus the solutions computed by *GNG-Q* and *I-GNG-Q* are on average better than the baseline computed by *GCC*.

It can be seen that the radii of the 95% confidence intervals for the solutions computed by Q-Learning are smaller than that computed by *GNG-Q* and *I-GNG-Q*. The reason for this is the fact that Q-Learning uses an exhaustive tabular storage that has one entry for every state-action pair which allows Q-Learning to store its knowledge very detailed and which also results in highly similar policies. Thus, the confidence intervals are very close. On the other hand, the resulting approximations for *GNG-Q* and *I-GNG-Q* are less similar.

The success rates of Q-Learning are also slightly higher as those for *GNG-Q* and *I-GNG-Q*. As argued in the analysis of the solutions computed by Q-Learning, some regions of the state space are not sufficiently explored. While in Q-Learning usually only one or at most a small number of state(s) suffer from this, in approximation schemes often slightly larger numbers of states are affected. This explains the difference in the success rates.

Additionally, we also analyzed the performances of *GNG-Q* and *I-GNG-Q* (cf. Table 10.14) when the start states are chosen according to the two settings described on page 186: Once again, the learned strategies were always better than the baseline computed by *GCC*. In general, *GNG-Q* performed better than *I-GNG-Q* and both approaches were clearly stronger than *GCC* in the outer setting. Remarkably, for a viewing range of five, *GNG-Q* had a minimal difference of zero which means that the solutions of *all* policies computed by *GNG-Q* on all start states in this settings were equal to or better than the solutions computed by *GCC*. On the downside, solutions computed by *I-GNG-Q* in the inner setting are only slightly better than those computed by *GCC*.

In order to get insights into the "optimal" policy in Shepherding, we investi-

Table 10.14: Results of the comparison of *GCC* and the approaches developed in this thesis with different settings of the start states: Mean values (with 95% confidence interval radii), standard deviation, minimal, and maximal values of the differences $\#steps^{\pi}_{diff}(s)$ as well as success rates (SR) of the policies computed by *GNG-Q* and *I-GNG-Q* for varying viewing ranges r_{sheep} of the sheep.

		Only inner cells					**Only outer cells**				
	r_{sheep}	Mean	Std. Dev.	Min	Max	SR	Mean	Std. Dev.	Min	Max	SR
GNG-Q	1	1.76 ± 0.18	2.66	-8	4	1.0000	3.42 ± 0.11	3.77	-14	14	0.9950
	2	3.09 ± 0.18	2.99	-13	8	1.0000	6.27 ± 0.07	5.01	-18	24	0.9959
	3	4.79 ± 0.18	3.51	-9	12	1.0000	6.69 ± 0.05	6.16	-12	34	0.9930
	4	6.94 ± 0.12	3.64	-2	16	0.9939	8.14 ± 0.03	8.15	-32	44	0.9911
	5	7.88 ± 0.12	3.84	0	18	0.9995	8.80 ± 0.03	9.88	-12	54	0.9923
I-GNG-Q	1	0.01 ± 0.39	3.52	-10	4	1.0000	3.02 ± 0.16	4.60	-12	14	0.9946
	2	0.52 ± 0.37	3.79	-9	8	0.9917	5.44 ± 0.09	5.22	-13	24	0.9932
	3	1.06 ± 0.35	5.17	-17	12	0.9976	5.73 ± 0.08	6.81	-19	34	0.9894
	4	1.70 ± 0.33	7.47	-23	16	0.9851	7.56 ± 0.06	8.54	-25	44	0.9926
	5	1.86 ± 0.27	8.83	-31	18	0.9953	8.24 ± 0.05	10.32	-34	54	0.9958

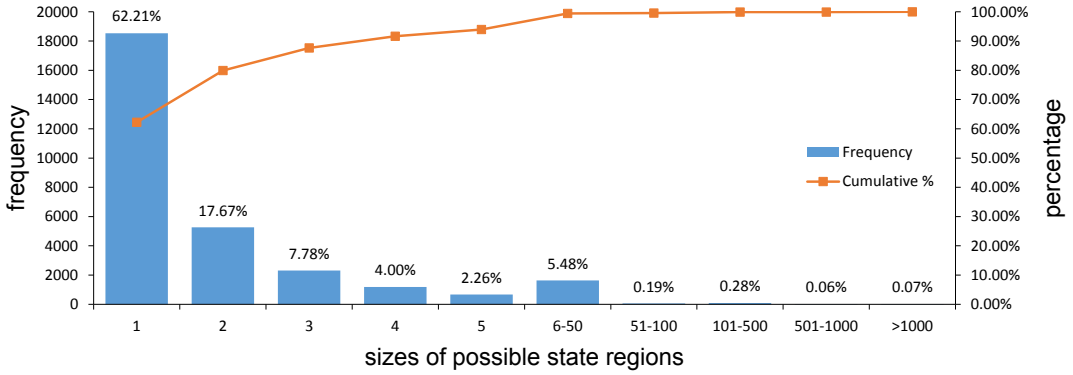


Figure 10.23: Frequencies (left axis) and cumulative percentage (right axix) for the sizes of possible state regions.

gated the best policy derived by Q-Learning in the previous analysis and analyzed the sizes of possible state regions[3]: After 1,000,000 episodes, all groups of states were computed that are neighboring and that share the same optimal action (i.e. states that could possibly become a state region as described in Section 7.4). Figure 10.23 shows the result of this analysis. It can be seen that roughly 62% of all states had no neighboring state with the same optimal action and nearly 18% only had one neighboring state that additionally needed the same behavior. Those states could thus not be usefully approximated by treating them equally. Only about 6% of all states (i.e. groups with more than five states) could really profit from a state-space aggregation as performed by *GNG-Q* since neurons could probably been placed in those areas. Additionally, this result also concerns the performance of *I-GNG-Q*: The rapid changes in the value function that are closely connected to the numerous differences of the optimal action are hard to approximate with a (smooth) value function approximation. This observation was also made by Menache et al. (2005) who noticed that the largest errors in their approximation were in regions where the optimal policy changes frequently in a small area. Similarly, Smart (2004) pointed out that approximation schemes that are based on some sort of similarity or distance may blur discontinuities of the value function.

The reason why *GNG-Q* and *I-GNG-Q* are able to function even in the presence of such discontinuities is the following: Some neighboring states might have different optimal actions in the policy learned by Q-Learning although they may be treated equally without loosing too much performance. These areas are found by the approximations and thus, the number of required states is reduced.

We also investigated where *GNG-Q* and *I-GNG-Q* placed the neurons: It could be seen that both approaches placed most of the neurons in regions where at least some neighboring states with the same optimal action exist. Additionally, the density of neurons is higher in regions closer to the goal as already seen in the analysis of a two-dimensional state space in Section 7.11.3.

Although the success rates of tabular Q-Learning were higher than those of *GNG-Q* and *I-GNG-Q*, both approaches offer a crucial advantage: *GNG-Q* computed

[3] In fact, we did this analysis for all combinations of exploration probability ε and viewing range r_{sheep}. As the results were nearly identical, we here show the result for $r_{sheep} = 1$ and $\varepsilon = 0.6$.

an abstract state space that only needed 4.3% of the number of the total states while *I-GNG-Q* needed 4.2%. Thus, both approaches are able to compute compact policies while only suffering a slight drop of performance.

10.8 Conclusion

We saw that *GNG-Q* and *I-GNG-Q* perform well on the benchmark tasks and that both were able to computed compact policies. In general, the solutions computed by *I-GNG-Q* are more stable and *I-GNG-Q* finds good policies earlier. Both approaches presented in this thesis allow the automatic computation of effective policies that abandon the need to decide the level of approximation beforehand.

The investigation of the approaches' parameter revealed influences on the final performance, the size of the approximation, and the time for finding a good solution (i.e. a solution that is as most 5% worse than the final policy) for the first time.

For *GNG-Q* the largest influence on all three measures is given by the number of episodes between each two insertion of new neurons (insertion delay λ_{insert}). Larger values of λ_{insert} usually result in larger approximations and longer times for finding good policies. Smaller values allow the agent to faster find good policies that are usually larger. The movement strength ϵ_b of *GNG-Q* influences the time of finding a good approximation and of the final solution quality only slightly. Still, too large or too small values influence the stability of the performance. For the maximal connection age age_{max} in *GNG-Q*, smaller values result in less stable performances as well as in slightly longer times for finding a good solution.

In *I-GNG-Q*, the influence of λ_{insert} on the size of the approximation as well as the time of finding good solutions is similar but not as strong as in *GNG-Q*. For the speed of learning, λ_{insert} has an influence of 55% while in *GNG-Q* the influence was 70%. The largest difference can be found in the parameters' influences on the final performance: Here the exponent p of the inverse distance weighting function has the largest influence followed by the number of interpolation bases. Generally, the higher p the better the final performance. If the number of interpolation bases k is too high, the final performance is reduced which can be explained by the fact that then possibly too many prototype Q-vectors are involved.

The Shepherding task confirmed that learning is particularly advantageous if designing a solution strategy beforehand is difficult because the task is only partially known or not known at all. We here saw an example where a learning approach outperformed hand coded strategies: The *GCC* approach is arguably cautious and only approaches the sheep from "safe" positions which results in superfluous steps. The reinforcement learning agent on the other hand tries different strategies and is able to learn efficient ways of handling the sheep (see e.g. the strategy comparison in Figure 10.21). Clearly, a similar behavior could be hard coded into *GCC* but this would require the consideration of many special cases and tremendous amounts of domain knowledge.

Especially for growing viewing ranges of the sheep, the reinforcement learning approach was much more powerful (i.e. the differences in terms of number of steps between the learned behaviors and *GCC* were relatively high). This also strengthens the above point as these high differences are in general due to encircling of the sheep's

viewing range as it is performed in *GCC*.

We saw that the performance of the reinforcement learning approach depends on the exploration probability ε: The higher the value of ε, the better the success rate and the larger the improvement in terms of number of steps over the solutions computed by *GCC*. This observation is true for the comparison on all states as well as for the detailed comparison that separated the states into *inner* and *outer* cells. We also saw that reinforcement learning was in some cases not able to solve the task due to the lack of experience in the affected start states.

The approximation approaches presented in this thesis performed similar to standard Q-Learning. Nevertheless, neither *GNG-Q* nor *I-GNG-Q* were as sensitive to the exploration as Q-Learning. Both approximation approaches computed policies that only needed about 4% of the storage Q-Learning needed and, additionally, the solutions were found faster than with Q-Learning. On the downside, the success rates of the approximations were slightly worse than those of tabular Q-Learning.

In our opinion, learning is clearly advantageous as the policies are in general better than the baseline computed by *GCC*. Additionally, no domain knowledge is needed to derive powerful strategies. The drawbacks of tabular Q-Learning (i.e. the long training time and the large amount of needed storage) can be resolved e.g. by the approximations presented in this thesis.

11 Conclusion and Future Work

This chapter summarizes the conclusion and the results of this thesis. In addition to this, we point out some directions for future work.

11.1 Conclusions

In general, we saw that learning presents a powerful solution for dealing with Shepherding tasks. We also saw that it is possible to adjust adaptive neural methods to serve as approximations for reinforcement learning.

We started in Chapter 4 by introducing the Shepherding task, highlighted the importance of this particular task, and pointed out similar real world tasks that would benefit—either directly or with some adjustments—from shepherding strategies. In addition to this, we showed how the biological background can be modeled as agentsystem and formalized the task of dogs driving sheep to a target area. The resulting system was classified into existing agentsystem taxonomies from literature. For the general task of shepherding agents, we carefully analyzed the state-space complexity and compared the results to the complexities of other tasks. We saw that even the most recent results in solving board games are far away from state space with sizes similar to the task of controlling sheep.

In Chapter 5, we proved close upper and lower bounds on the optimal solution and showed that these upper and lower bounds differ by a term linear in the viewing range of the sheep. We showed that this difference is caused by the possibly necessary circumnavigation of the sheep's viewing range without causing an unintended movement of the sheep. We introduced the Greedy Coordinate Correction (GCC) approached, a greedy algorithm that solves Shepherding$(1, 1)$-instances within these bounds. For this algorithm, we showed that the computational complexity of GCC is linear and, more precisely, that the runtime only depends on the length of the solution. Additionally, the algorithm only needs a constant amount of storage. This result is especially interesting considering the exponential state-space complexity of the Shepherding task as analyzed before.

Supplementary to the *GCC*-approach, we modeled the shepherding task as reinforcement learning task. As we saw in Chapter 10 learning presents an advantageous solution because it often finds very good solutions while not depending on domain knowledge as the algorithmic solution. A drawback of learning may of course be that the agent has to store and to repeatedly improve knowledge for every possible state of the environment—facts that are particularly serious in state spaces as large as of the Shepherding task. Fortunately, the approximation approaches developed in this thesis offer a remedy for this.

Chapter 7 presented *GNG-Q*, a combination of Q-Learning and growing neural gas (GNG) that builds a state-space aggregation for reinforcement learning while the agent interacts with its environment. The core idea of our approach is to use the GNG quantizer to aggregate similar states into regions that can be treated equally. In parallel, we apply Q-Learning on the current approximation and use feedback from learning to adjust the approximation if necessary. The approximation is refined in areas where the learner's estimated policy changes often and thus, similarity in both the state and action space is respected. The Q-function in the *GNG-Q*-approach is defined over neurons and actions and can be learned with tabular Q-Learning using one entry for every neuron-action pair. This approximation leads to a piecewise constant approximation of the Q-function as all states in a region are treated equally.

The advantages of *GNG-Q* include that knowledge achieved during learning is used to refine the approximation which supersedes the need of deciding on the granularity of approximation beforehand. Additionally, *GNG-Q* works online (i.e. at any time during learning, the agent can make use of the knowledge acquired so far) and does not need the model of the reinforcement learning task to compute an efficient discretization. The agent only has to store the positions and the prototype Q-vectors of the neurons which results in a very compact representation. The state aggregation function is covered by the nearest neighbor rule and thus, this approach is well suitable for an implementation on actual robots.

In Chapter 8 we presented a function approximation approach for reinforcement learning that is also based on the growing neural gas. The enhancements in the *I-GNG-Q*-approach lead to a faster stabilization of the performance and an improved regulation of the refinement and adaptation. As *I-GNG-Q* computes Q-values as weighted combinations of several prototype Q-vectors, the approximated value function is no longer piecewise constant but smooth. Additionally, the combination of several prototype Q-vectors helps to stabilize the learning as possible erroneous knowledge can be corrected faster and does not have such high influence.

I-GNG-Q is capable of learning compact approximations in parallel with an (nearly) optimal policy and its performance is well competitive with other approaches from literature without the need of knowing the considered RL task beforehand. Furthermore, we showed how to incorporate eligibility traces to speed up learning and to more efficiently use the agent's experience. In addition to this, we formulated criteria for the decision of when to adjust the approximation. *I-GNG-Q* uses an update function and a feature computation that are designed to prevent an exaggeration of Q-values.

GNG-Q and *I-GNG-Q* share that both approaches compute approximations for reinforcement learning in parallel with the agent's interaction with its environment.

Additionally, the agent's behavior *and* its representation is subjected to learning. We argued that this fact leads to two interleaved learning tasks: The first task is to learn the approximation's parameter (i.e. the positions and the number of the neurons) and, simultaneously, the second task is to use this approximation to learn the agent's behavior. Additionally, the adaption of the approximation used in *GNG-Q* can also be seen as parameter exploration as discussed e.g. by Rückstieß et al. (2010).

Both of our approaches share that the underlying GNG vector quantizer is quite insensitive to the values of its required parameters (Heinke and Hamker, 1998). *I-GNG-Q* needs only two additional parameters: The first, k, controls how much the approximation should generalize its knowledge while the second, p is used to control the emphasis of the nearest neuron's prototype Q-vector. As the computation of exponential functions often employed for RBF-based function approximators is usually very slow (Schraudolph, 1999), the *inverse distance weighting* used here is a performant alternative.

In Chapter 10 we thoroughly analyzed the influences of the parameters for our approaches on the final performance, the size of the approximation, and the speed of finding good solutions. We saw that *GNG-Q* and *I-GNG-Q* perform well on benchmark tasks and that both were able to compute compact policies. In general, the solutions computed by *I-GNG-Q* are more stable and *I-GNG-Q* finds good policies earlier than *GNG-Q*. Nevertheless, both approaches presented in this thesis allow the automatic computation of effective policies that abandon the need to decide the level of approximation beforehand.

Generally, learning is particularly advantageous if designing a solution strategy beforehand is difficult because the task is only partially known or not known at all. For the Shepherding task, the reinforcement learning approach was much more powerful (i.e. the differences in terms of number of steps between the learned behaviors and *GCC* were relatively high) for growing viewing ranges of the sheep. Additionally, both approximation approaches computed policies that only used a fraction of the storage needed by Q-Learning and, additionally, the solutions were found faster than with Q-Learning.

In our opinion, learning in the Shepherding task is clearly advantageous as the policies are generally better than the baseline computed by *GCC*. Additionally, no domain knowledge is needed to derive powerful strategies. The drawbacks of tabular Q-Learning (i.e. the long training time and the large amount of needed storage) can be resolved e.g. by the approximations presented in this thesis.

11.2 Future Work

For the future, several directions can be followed based on the findings in this thesis.

To us, the most interesting point is the extension to multiple agents. In the Shepherding task the number of dogs as well as the number of sheep can be increased. Having a small number of dogs (e.g. two or three) control a large flock of sheep as it is done in the real world makes the task even more interesting. Nevertheless, this also makes the task more complex as we have seen in Section 4.6. In fact, the agents shall still *learn* the needed behavior—requiring e.g. cooperation or coordination—without a central instance and with as little communication as possible.

In order to bring the Shepherding task even closer to its origin in the nature, several possibilities exist. For example, the viewing ranges of the agents may be limited which would force the dogs to decide and to cooperate based on these local perceptions which would lead to an partially observable Markov decision process. Then of course, the Shepherding task may be investigated in continuous state spaces or in the presence of predators.

This thesis investigated one—in our opinion the most suitable—out of numerous learning methods to learn shepherding behavior. Nonetheless, other approaches may be further explored. Especially imitation—i.e. adopting behavior learned by other agents—can be considered as a means to improve the learning performance and the speed of learning.

The transition to multiple agents is also of interest in the field of approximation schemes for reinforcement learning. As those systems suffer even more from the aforementioned curse of dimensionality, state aggregation as well as function approximation should highly improve their performance. To date, most (adaptive) approximation schemes are investigated in single-agent settings as the transition from single-agent to multiagent reinforcement learning itself introduces a moving learning goal. In this context, the question to what extend adaptive approaches (i.e. learning behavior and its representation in parallel) like the neural approximations introduced in this thesis can be applied to multiagent reinforcement learning. There, it could e.g. be investigated, how such approaches can deal with partial observability. Additionally, it could be analyzed to what extend agents can generalize knowledge gained with a small number of agents to a setting with many more agents.

Bibliography

Charles L. Adler and James Tanton. π is the Minimum Value for Pi. *The College Mathematics Journal*, 31(2):102–106, 2000.

Victor L. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, 1994.

Esther M. Arkin, Robert Connelly, and Joseph S. B. Mitchell. On Monotone Paths Among Obstacles with Applications to Planning Assemblies. In *Proceedings of the Fifth Annual Symposium on Computational Geometry (SoCG 1989)*, pages 334–343, 1989.

Samuel Barrett, Matthew E. Taylor, and Peter Stone. Transfer Learning for Reinforcement Learning on a Physical Robot. In *Proceedings of the Ninth International Conference on Autonomous Agents and Multiagent Systems - Adaptive Learning Agents Workshop (ALA 2010)*, 2010.

Maxim A. Batalin and Gaurav S. Sukhatme. Coverage, Exploration, and Deployment by a Mobile Robot and Communication Network. In *Proceedings of the Second International Workshop on Information Processing in Sensor Networks (IPSN 2003)*, volume 2634 of *Lecture Notes in Computer Science*, pages 376–391, Berlin, Heidelberg, Germany, 2003. Springer.

Michael Baumann and Hans Kleine Büning. State Aggregation by Growing Neural Gas for Reinforcement Learning in Continuous State Spaces. In *Proceedings of the Tenth International Conference on Machine Learning and Applications (ICMLA 2011)*, pages 430–435, 2011.

Michael Baumann and Hans Kleine Büning. Adaptive Function Approximation in Reinforcement Learning with an Interpolating Growing Neural Gas. In *Proceedings of the Twelfth International Conference on Hybrid Intelligent Systems (HIS 2012)*, pages 512–517, 2012.

Michael Baumann and Hans Kleine Büning. Learning Shepherding Behavior. In *Advances in Artificial Intelligence—Local Proceedings of the Sixteenth Portuguese Conference on Artificial Intelligence (EPIA 2013)*, pages 166–178, 2013.

Michael Baumann and Hans Kleine Büning. Adaptive Function Approximation in Reinforcement Learning with an Interpolating Growing Neural Gas. *International Journal of Hybrid Intelligent Systems*, 11(1):55–69, 2014.

Michael Baumann, Timo Klerx, and Hans Kleine Büning. Improved State Aggregation with Growing Neural Gas in Multidimensional State Spaces. In *Proceedings of*

the Fifth International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems (ERLARS 2012), pages 27–36, 2012.

Richard E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957.

Dimitri P. Bertsekas and David A. Castañon. Adaptive Aggregation Methods for Infinite Horizon Dynamic Programming. *IEEE Transactions on Automatic Control*, 34(6):589–598, 1989.

Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, USA, 1996.

Justin A. Boyan and Andrew W. Moore. Generalization in Reinforcement Learning: Safely Approximating the Value Function. In *Advances in Neural Information Processing Systems 7: Proceedings of the 1994 NIPS Conference*, pages 369–376, 1994.

Steven J. Bradtke and Andrew G. Barto. Linear Least-Squares Algorithms for Temporal Difference Learning. *Machine Learning*, 22(1-3):33–57, 1996.

Rodney A. Brooks. Intelligence without Representation. *Artificial Intelligence*, 47 (1-3):139–159, 1991.

Guido Bugmann. Normalized Gaussian Radial Basis Function Networks. *Neurocomputing*, 20(1-3):97–110, 1998.

Wolfram Burgard, Armin B. Cremers, Dieter Fox, Dirk Hähnel, Gerhard Lakemeyer, Dirk Schulz, Walter Steiner, and Sebastian Thrun. The Interactive Museum Tour-Guide Robot. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference (AAAI 1998 / IAAI 1998)*, pages 11–18, 1998.

Lucian Buşoniu. *Reinforcement Learning in Continuous State and Action Spaces*. PhD thesis, Delft University of Techology, 2008.

Lucian Buşoniu, Damien Ernst, Bart De Schutter, and Robert Babuška. Fuzzy Partition Optimization for Approximate Fuzzy Q-iteration. In *Proceedings of the Seventeenth IFAC World Congress (IFAC 2008)*, pages 5629–5634, 2008.

Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent Reinforcement Learning: An Overview. In Dipti Srinivasan and Lakhmi C. Jain, editors, *Innovations in Multi-Agent Systems and Applications*, volume 310 of *Studies in Computational Intelligence*, pages 183–221. Springer, Berlin, Heidelberg, Germany, 2010.

Lucian Buşoniu, Damien Ernst, Bart De Schutter, and Robert Babuška. Approximate Reinforcement Learning: An Overview. In *Proceedings of the Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL 2011)*, pages 1–8, 2011a.

Lucian Buşoniu, Damien Ernst, Bart De Schutter, and Robert Babuška. Cross-Entropy Optimization of Control Policies With Adaptive Basis Functions. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 41(1):196–209, 2011b.

Jennifer Casper and Robin R. Murphy. Human-Robot Interactions During the Robot-Assisted Urban Search and Rescue Response at the World Trade Center. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 33(3):367–385, 2003.

Hande Çelikkanat and Erol Sahin. Steering Self-Organized Robot Flocks through Externally Guided Individuals. *Neural Computing and Applications*, 19(6):849–865, 2010.

Soumen Chakrabarti, Earl Cox, Eibe Frank, Ralf Hartmut Gting, Jiawei Han, Xia Jiang, Micheline Kamber, Sam S. Lightstone, Thomas P. Nadeau, Richard E Neapolitan, Dorian Pyle, Mamdouh Refaat, Markus Schneider, Toby J. Teorey, and Ian H. Witten. *Data Mining: Know It All*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

David Chapman and Leslie Pack Kaelbling. Input Generalization in Delayed Reinforcement Learning: An Algorithm and Performance Comparisons. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI 1991)*, pages 726–731, 1991.

Lorna Coppinger and Raymond Coppinger. Dogs for Herding and Guarding Livestock. In Temple Grandin, editor, *Livestock Handling and Transport*, pages 199–214. CABI International, third edition, 2007.

Peter I. Cowling and Christian Gmeinwieser. AI for Herding Sheep. In *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2010)*, pages 2–7, 2010.

André da Motta Salles Barreto and Charles W. Anderson. Restricted Gradient-Descent Algorithm for Value-Function Approximation in Reinforcement Learning. *Artificial Intelligence*, 172:454–482, 2008.

Peter Dayan and Terrence J. Sejnowski. TD(lambda) Converges with Probability 1. *Machine Learning*, 14(1):295–301, 1994.

Mark de Berg. On Rectilinear Link Distance. *Computational Geometry*, 1:13–34, 1991.

Thomas L. Dean and Robert Givan. Model Minimization in Markov Decision Processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference (AAAI 1997 / IAAI 1997)*, pages 106–111, 1997.

Bruce R. Donald, James Jennings, and Daniela Rus. Analyzing Teams of Cooperating Mobile Robots. In *Proceedings of the 1994 International Conference on Robotics and Automation (ICRA 1994)*, pages 1896–1903, 1994.

Kurt Driessens and Saso Dzeroski. Integrating Guidance into Relational Reinforcement Learning. *Machine Learning*, 57(3):271–304, 2004.

Chris Drummond. Preventing Overshoot of Splines with Application to Reinforcement Learning. Technical Report TR-96-05, University of Ottawa, 1996.

John R.G. Dyer, Christos C. Ioannou, Lesley J. Morrell, Darren P. Croft, Iain D. Couzin, Dean A. Waters, and Jens Krause. Consensus Decision Making in Human Crowds. *Animal Behaviour*, 75(2):461–470, 2008.

Stefan Edelkamp and Richard E. Korf. The Branching Factor of Regular Search Spaces. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Conference on Innovative Applications of Artificial Intelligence (AAAI 1998 / IAAI 1998)*, pages 299–304, 1998.

Yaakov Engel, Shie Mannor, and Ron Meir. Bayes Meets Bellman: The Gaussian Process Approach to Temporal Difference Learning. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML 2003)*, pages 154–161, 2003.

Yaakov Engel, Shie Mannor, and Ron Meir. Reinforcement Learning with Gaussian Processes. In *Proceedings of the Twenty-Second International Conference on Machine Learning (ICML 2005)*, pages 201–208, 2005.

Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-Based Batch Mode Reinforcement Learning. *Journal of Machine Learning Research*, 6:503–556, 2005.

Eyal Even-Dar and Yishay Mansour. Learning Rates for Q-Learning. *Journal of Machine Learning Research*, 5:1–25, 2003a.

Eyal Even-Dar and Yishay Mansour. Approximate Equivalence of Markov Decision Processes. In *Proceedings of the Sixteenth Annual Conference on Computational Learning Theory and Seventh Kernel Workshop (COLT 2003 / Kernel 2003)*, pages 581–594, 2003b.

Mark Evered, Peter Burling, and Mark Trotter. An Investigation of Predator Response in Robotic Herding of Sheep. In *International Proceedings of Chemical, Biological and Environmental Engineering*, volume 63, pages 49–54, 2014.

Jacques Ferber. *Multi-agent systems - An Introduction to Distributed Artificial Intelligence*. Addison-Wesley-Longman, Chichester, UK, 1999.

Fernando Fernández and Daniel Borrajo. VQQL. Applying Vector Quantization to Reinforcement Learning. In *RoboCup-99: Robot Soccer World Cup III*, pages 292–303, 2000.

Fernando Fernández and Daniel Borrajo. Two Steps Reinforcement Learning. *International Journal of Intelligent Systems*, 23:213–245, 2008.

Merv Fingas. *The Basics of Oil Spill Cleanup*. Taylor & Francis, Boca Raton, FL, USA, second edition, 2002.

Daniel Fišer, Jan Faigl, and Miroslav Kulich. Growing Neural Gas Efficiently. *Neurocomputing*, 104:72–82, 2013.

Robert M. French. Catastrophic Forgetting in Connectionist Networks. *Trends in Cognitive Sciences*, 3(4):128–135, 1999.

Bernd Fritzke. Fast Learning with Incremental RBF Networks. *Neural Processing Letters*, 1(1):2–5, 1994a.

Bernd Fritzke. A Growing Neural Gas Network Learns Topologies. In *Advances in Neural Information Processing Systems 7: Proceedings of the 1994 NIPS Conference*, pages 625–632. MIT Press, 1994b.

Bernd Fritzke. Growing self-organizing networks – why? In *Proceedings of the European Symposium on Artificial Neural Networks (ESANN 1996)*, pages 61–72, 1996.

Bernd Fritzke. *Handbook of Neural Computation*, chapter Unsupervised ontogenetic networks. IOP Publishing Ltd and Oxford University Press, 1997.

Bernd Fritzke. *Vektorbasierte Neuronale Netze*. Professorial dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, 1998.

Martin Gardner. *The Last Recreations: Hydras, Eggs, and Other Mathematical Mystifications*. Springer, New York, NY, USA, 1997.

Christopher J. Gatti and Mark J. Embrechts. Reinforcement Learning with Neural Networks: Tricks of the Trade. In *Advances in Intelligent Signal Processing and Data Mining*, pages 275–310. Springer, Berlin, Heidelberg, Germany, 2013.

Allen Gersho and Robert M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

Zoubin Ghahramani. Unsupervised Learning. In Olivier Bousquet, Ulrike Luxburg, and Gunnar Rätsch, editors, *Advanced Lectures on Machine Learning*, volume 3176 of *Lecture Notes in Computer Science*, pages 72–112. Springer, Berlin, Heidelberg, Germany, 2004.

Robert Givan, Thomas L. Dean, and Matthew Greig. Equivalence Notions and Model Minimization in Markov Decision Processes. *Artificial Intelligence*, 147(1-2): 163–223, 2003.

Robert L. Goldstone and Lawrence W. Barsalou. Reuniting Perception and Conception. *Cognition*, 65(2-3):231–262, 1998.

Jorge Gomes, Pedro Mariano, and Anders Lyhne Christensen. Cooperative Coevolution of Partially Heterogeneous Multiagent Systems. In *Proceedings of the Fourteenth International Conference on Autonomous Agents & Multiagent Systems (AAMAS 2015)*, pages 297–305, 2015.

Temple Grandin. Farm Animal Welfare during Handling, Transport, and Slaughter. *Journal of the American Veterinary Medical Association*, 204:372–377, 1994.

Robert M. Gray. Vector Quantization. *ASSP Magazine, IEEE*, 1:4–29, 1984.

Jeffrey S. Green and Roger A. Woodruff. *Livestock Guarding Dogs: Protecting Sheep from Predators*. U.S. Department of Agriculture, Animal and Plant Health Inspection Service, Washington D.C., USA, 1993.

William D. Hamilton. Geometry for the Selfish Herd. *Journal of Theoretical Biology*, 31(2):295–311, 1971.

Joseph F. Harrison, Christopher Vo, and Jyh-Ming Lien. Scalable and Robust Shepherding via Deformable Shapes. In *Proceedings of the Third International Conference on Motion in Games (MIG 2010)*, pages 218–229, 2010.

Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, 1998.

Dietmar Heinke and Fred H. Hamker. Comparing Neural Networks: A Benchmark on Growing Neural Gas, Growing Cell Structures, and Fuzzy ARTMAP. *IEEE Transactions on Neural Networks*, 9:1279–1291, 1998.

Peter Hilton and Jean Pedersen. Catalan Numbers, Their Generalization, and Their Uses. *The Mathematical Intelligencer*, 13(2):64–75, 1991.

John E. Hopcroft, Jacob T. Schwartz, and Micha Sharir. On the Complexity of Motion Planning for Multiple Independent Objects; PSPACE- Hardness of the "Warehouseman's Problem". *The International Journal of Robotics Research*, 3(4): 76–88, 1984.

IFR Statistical Department. Service Robot Statistics, 2013. URL `http://www.ifr.org/service-robots/statistics/`. visited July 4, 2015.

Luca Iocchi, Daniele Nardi, and Massimiliano Salerno. Reactivity and Deliberation: A Survey on Multi-Robot Systems. In *Balancing Reactivity and Social Deliberation in Multi-Agent Systems: From RoboCup to Real-World Applications (selected papers from the (ECAI 2000) Workshop and additional contributions)*, pages 9–34, 2001.

Raj Jain. *The Art of Computer System Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. John Wiley, New York, NY, USA, 1991.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence*, 101 (1-2):99–134, 1998.

Sham Machandranath Kakade. *On the Sample Complexity of Reinforcement Learning*. PhD thesis, Gatsby Computational Neuroscience Unit, University College London, 2003.

Philipp W. Keller, Shie Mannor, and Doina Precup. Automatic Basis Function Construction for Approximate Dynamic Programming and Reinforcement Learning. In *Proceedings of the Twenty-Third International Conference on Machine Learning (ICML 2006)*, pages 449–456, 2006.

John M. Kenny, Clark McPhail, Donald. N. Farrer, Dick Odenthal, Sid Heal, Jim Taylor, Steve Ijames, and Peter Waddington. Crowd Behavior, Crowd Control, and the Use of Non-Lethal Weapons. Technical Report A274644, Penn State Applied Research Laboratory, 2001.

Joel A. Kirkland and Anthony A. Maciejewski. A Simulation of Attempts to Influence Crowd Dynamics. In *Proceedings of the International Conference on Systems, Man & Cybernetics (SMC 2003)*, pages 4328–4333, 2003.

Donald E. Knuth. *The Art of Computer Programming, Volume 2 (Third Edition): Seminumerical Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

Teuvo Kohonen. Self-Organized Formation of Topologically Correct Feature Maps. *Biological Cybernetics*, 43:59–69, 1982.

J. Zico Kolter and Andrew Y. Ng. Regularization and Feature Selection in Least-Squares Temporal Difference Learning. In *Proceedings of the Twenty-Sixth International Conference on Machine Learning (ICML 2009)*, pages 521–528, 2009.

George Konidaris, Sarah Osentoski, and Philip S. Thomas. Value Function Approximation in Reinforcement Learning Using the Fourier Basis. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2011)*, pages 380–385, 2011.

Sotiris Kotsiantis and Dimitris Kanellopoulos. Discretization Techniques: A Recent Survey. *GESTS International Transactions on Computer Science and Engineering*, 32(1):47–58, 2006.

Alexander Kron, Günther Schmidt, Bernd Petzold, Michael F. Zaeh, Peter Hinterseer, and Eckehard G. Steinbach. Disposal of Explosive Ordnances by Use of a Bimanual Haptic Telepresence System. In *Proceedings of the 2004 International Conference on Robotics and Automation (ICRA 2004)*, pages 1968–1973, 2004.

Michail G. Lagoudakis and Ronald Parr. Least-Squares Policy Iteration. *Journal of Machine Learning Research*, 4:1107–1149, 2003.

Richard C. Larson and Victor O. K. Li. Finding Minimum Rectilinear Distance Paths in the Presence of Barriers. *Networks*, 11(3):285–304, 1981.

Manfred Lau, Jun Mitani, and Takeo Igarashi. Automatic Learning of Pushing Strategy for Delivery of Irregular-Shaped Objects. In *Proceedings of the 2011 International Conference on Robotics and Automation (ICRA 2011)*, pages 3733–3738, 2011.

Der-Tsai Lee and Franco P. Preparata. Euclidean Shortest Paths in the Presence of Rectilinear Barriers. *Networks*, 14(3):393–410, 1984.

Ivan S. Lee and Henry Y. Lau. Adaptive State Space Partitioning for Reinforcement Learning. *Engineering Applications of Artificial Intelligence*, 17:577–588, 2004.

Theodor Lettmann, Michael Baumann, Markus Eberling, and Thomas Kemmerich. Modeling Agents and Agent Systems. *Transactions on Computational Collective Intelligence*, 5:157–181, 2011.

Jyh-Ming Lien and Emlyn Pratt. Interactive Planning for Shepherd Motion. In *Papers from the 2009 AAAI Spring Symposium: Agents that Learn from Human Teachers*, pages 95–102, 2009.

Jyh-Ming Lien, O. Burçhan Bayazit, Ross T. Sowell, Samuel Rodríguez, and Nancy M. Amato. Shepherding Behaviors. In *Proceedings of the 2004 International Conference on Robotics and Automation (ICRA 2004)*, pages 4159–4164, 2004.

Stephen Lin and Robert Wright. Evolutionary Tile Coding: An Automated State Abstraction Algorithm for Reinforcement Learning. In *Proceedings of the (AAAI 2010) Workshop on Abstraction, Reformulation, and Approximation (WARA 2010)*, pages 42–47, 2010.

Yoseph Linde, Andrés Buzo, and Robert M. Gray. An Algorithm for Vector Quantizer Design. *IEEE Transactions on Communications*, 28(1):84–95, 1980.

Sridhar Mahadevan. Proto-Value Functions: Developmental Reinforcement Learning. In *Proceedings of the Twenty-Second International Conference on Machine Learning (ICML 2005)*, pages 553–560, 2005.

Edgar A. Martiínez-García, Ohya Akihisa, and Shin'ichi Yuta. Crowding and Guiding Groups of Humans by Teams of Mobile Robots. In *Proceedings of the 2005 Workshop on Advanced Robotics and its Social Impacts (ARSO 2005)*, pages 91–96, 2005.

Thomas Martinetz and Klaus Schulten. A "Neural-Gas" Network Learns Topologies. *Artificial Neural Networks*, 1:397–402, 1991.

Maja J. Matarić. Reward Functions for Accelerated Learning. In *Proceedings of the Eleventh International Conference on Machine Learning (ICML 1994)*, pages 181–189, 1994.

Maja J. Matarić, Martin Nilsson, and Kristian T. Simsarin. Cooperative Multi-Robot Box-Pushing. In *Proceedings of the 1995 International Conference on Intelligent Robots and Systems (IROS 1995)*, pages 556–561, 1995.

Andrew McCallum. Instance-Based Utile Distinctions for Reinforcement Learning with Hidden State. In *Proceedings of the Twelfth International Conference on Machine Learning (ICML 1995)*, pages 387–395, 1995.

David McFarland. *A Dictionary of Animal Behaviour*. Oxford University Press, New York, NY, USA, 2006.

Francisco S. Melo and Isabel Ribeiro. Q-Learning with Linear Function Approximation. In *Proceedings of the Twentieth Annual Conference on Learning Theory (COLT 2007)*, pages 308–322, 2007.

Ishai Menache, Shie Mannor, and Nahum Shimkin. Basis Function Adaptation in Temporal Difference Reinforcement Learning. *Annals of Operations Research*, 134 (1):215–238, 2005.

Olivier Michel. Webots: Professional Mobile Robot Simulation. *Journal of Advanced Robotics Systems*, 1(1):39–42, 2004.

Joseph S. B. Mitchell, Günter Rote, and Gerhard J. Woeginger. Minimum-Link Paths Among Obstacles in the Plane. *Algorithmica*, 8(5&6):431–459, 1992.

Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, NY, USA, 1997.

Jörg P. Müller. The Right Agent (Architecture) to do the Right Thing. In *Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages, (ATAL 1998)*, pages 211–225, 1999.

Rémi Munos and Andrew Moore. Variable Resolution Discretization in Optimal Control. *Machine Learning*, 49:291–323, 2002.

Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning (ICML 1999)*, pages 278–287, 1999.

Illah R. Nourbakhsh, Clayton Kunz, and Thomas Willeke. The Mobot Museum Robot Installations: A Five Year Experiment. In *Proceedings of the 2003 International Conference on Intelligent Robots and Systems (IROS 2003)*, pages 3636–3641, 2003.

Dirk Ormoneit and Saunak Sen. Kernel-Based Reinforcement Learning. *Machine Learning*, 49(2-3):161–178, 2002.

Lynne E. Parker. Current Research in Multirobot Systems. *Artificial Life and Robotics*, 7(1-2):1–5, 2003.

Lynne E. Parker and Claude F. Touzet. Multi-Robot Learning in a Cooperative Observation Task. In *Distributed Autonomous Robotic Systems 4, Proceedings of the Fifth International Symposium on Distributed Autonomous Robotic Systems (DARS 2000)*, pages 391–402, 2000.

Marc J. V. Ponsen, Matthew E. Taylor, and Karl Tuyls. Abstraction and Generalization in Reinforcement Learning: A Summary and Framework. In *Revised Selected Papers of the Second Workshop on Adaptive and Learning Agents (ALA 2009)*, pages 1–32, 2009.

Mitchell A. Potter, Lisa Meeden, and Alan C. Schultz. Heterogeneity in the Coevolved Behaviors of Mobile Robots: The Emergence of Specialists. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 1337–1343, 2001.

Martin. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, Hoboken, NJ, USA, 2005.

Larry D. Pyeatt and Adele E. Howe. Decision Tree Function Approximation in Reinforcement Learning. In *Proceedings of the Third International Symposium on Adaptive Systems (ISAS 2001): Evolutionary Computation and Probabilistic Graphical Models*, 1998.

Raghunath, James Cohoon, and Sartaj Sahni. Single Bend Wiring. *Journal of Algorithms*, 7(2):232–257, 1986.

Bohdana Ratitch and Doina Precup. Sparse Distributed Memories for On-Line Value-Based Reinforcement Learning. In *Proceedings of the Fifteenth European Conference on Machine Learning (ECML 2004)*, pages 347–358, 2004.

Balaraman Ravindran and Andrew G. Barto. Model Minimization in Hierarchical Reinforcement Learning. In *Proceedings of the Fifth International Symposium on Abstraction, Reformulation, and Approximation (SARA 2002)*, pages 196–211, 2002.

Sazalinsyah Razali, Qinggang Meng, and Shuang-Hua Yang. Immune-Inspired Cooperative Mechanism with Refined Low-Level Behaviors for Multi-Robot Shepherding. *International Journal of Computational Intelligence and Applications*, 11(1), 2012.

Craig W. Reynolds. Flocks, Herds and Schools: A Distributed Behavioral Model. In *Proceedings of the Fourteenth Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1987)*, pages 25–34. ACM, 1987.

Stuart I. Reynolds. The Stability of General Discounted Reinforcement Learning with Linear Function Approximation. In *Proceedings of the Second UK Workshop on Computational Intelligence (UKCI 2002)*, pages 139–146, 2002.

Thomas Rückstieß, Frank Sehnke, Tom Schaul, Daan Wierstra, Sun Yi, and Jürgen Schmidhuber. Exploring Parameter Space in Reinforcement Learning. *Paladyn Journal of Behavioral Robotics*, 1(1):14–24, 2010.

Gavin A. Rummery and Mahesan Niranjan. On-Line Q-Learning Using Connectionist Systems. Technical report, Cambridge University, Engineering Department, 1994.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (Third Edition)*. Pearson Education, Upper Saddle River, NJ, USA, 2010.

Juan C. Santamária, Richard Sutton, and Ashwin Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, 6(2):163–217, 1997.

Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers Is Solved. *Science*, 317(5844): 1518–1522, 2007.

Nicol N. Schraudolph. A Fast, Compact Approximation of the Exponential Function. *Neural Computation*, 11:853–862, 1999.

Alan Schultz, John J. Grefenstette, and William Adams. Robo-Shepherd: Learning Complex Robotic Behaviors. In *Robotics and Manufacturing: Recent Trends in Research and Applications*, volume 6, pages 763–768, 1996.

Donald Shepard. A Two-Dimensional Interpolation Function for Irregularly-Spaced Data. In *Proceedings of the Twenty-Third ACM National Conference (ACM 1968)*, pages 517–524, 1968.

Alexander A. Sherstov and Peter Stone. Function Approximation via Tile Coding: Automating Parameter Choice. In *Proceedings of the Sixth International Conference on Abstraction, Reformulation and Approximation (SARA 2005)*, 2005.

William P. Shulaw. *Sheep Care Guide*. American Sheep Industry Association, Englewood, CO, USA, 2005.

Olivier Sigaud and Pierre Gérard. Using Classifier Systems as Adaptive Expert Systems for Control. In *Third International Workshop on Advances in Learning Classifier Systems (IWLCS 2000)*, pages 138–157, 2000.

Satinder P. Singh and Richard S. Sutton. Reinforcement Learning with Replacing Eligibility Traces. *Machine Learning*, 22:123–158, 1996.

Satinder P. Singh, Tommi Jaakkola, and Michael I. Jordan. Reinforcement Learning with Soft State Aggregation. In *Advances in Neural Information Processing Systems 7: Proceedings of the 1994 NIPS Conference*, 1994.

William D. Smart. Explicit Manifold Representations for Value-Function Approximation in Reinforcement Learning. In *Proceedings of the Eighth International Symposium on Artificial Intelligence and Mathematics (AI&M 1-2004)*, 2004.

Andrew J. Smith. Applications of the Self-Organising Map to Reinforcement Learning. *Neural Networks*, 15:1107–1124, 2002.

Peter Stone and Manuela M. Veloso. Multiagent Systems: A Survey from a Machine Learning Perspective. *Autonomous Robots*, 8(3):345–383, 2000.

Daniel Strömbom, Richard P. Mann, Alan M. Wilson, Stephen Hailes, A. Jennifer Morton, David J. T. Sumpter, and Andrew J. King. Solving the Shepherding Problem: Heuristics for Herding Autonomous, Interacting Agents. *Journal of The Royal Society Interface*, 11(100), 2014.

Kyra Sundance. *The Dog Rules: 14 Secrets to Developing the Dog YOU Want*. Touchstone, New York, NY, USA, 2009.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, USA, 1998.

Csaba Szepesvári. *Algorithms for Reinforcement Learning*. Morgan and Claypool, San Rafael, CA, USA, 2010.

Csaba Szepesvári and William D. Smart. Interpolation-Based Q-Learning. In *Proceedings of the Twenty-First International Conference on Machine Learning (ICML 2004)*, pages 791–798, 2004.

Pinky Thakkar and Leonard P. Wesley. Autonomous mobile robot assisted herding. In *Proceedings of the Second International Conference on Informatics in Control, Automation, and Robotics (ICINCO 2005)*, pages 73–81, 2005.

Sebastian Thrun. Efficient Exploration In Reinforcement Learning. Technical report, School of Computer Science, Carnegie-Mellon University, 1992.

Sebastian Thrun and Anton Schwartz. Issues in Using Function Approximation for Reinforcement Learning. In *Proceedings of the 1993 Connectionist Models Summer School*, pages 255–263, 1993.

Sebastian Thrun, Michael Beetz, Maren Bennewitz, Wolfram Burgard, Armin B. Cremers, Frank Dellaert, Dieter Fox, Dirk Hähnel, Charles R. Rosenberg, Nicholas Roy, Jamieson Schulte, and Dirk Schulz. Probabilistic Algorithms and the Interactive Museum Tour-Guide Robot Minerva. *Journal of Robotics Research*, 19(11): 972–999, 2000.

John Tromp and Gunnar Farnebäck. Combinatorics of Go. In *Computers and Games*. Springer, Berlin, Heidelberg, Germany, 2007.

United States Coast Guard. On Scene Coordinator Report "Deepwater Horizon Oil Spill", 2011. URL `http://noaa.ntis.gov/view.php?pid=NOAA:ocn760102831`.

William T. B. Uther and Manuela M. Veloso. Tree Based Discretization for Continuous State Space Reinforcement Learning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference (AAAI 1998 / IAAI 1998)*, pages 769–774, 1998.

Martijn van Otterlo. *The Logic of Adaptive Behavior*. IOS Press, Amsterdam, 2009.

Martijn van Otterlo and Marco Wiering. Reinforcement Learning and Markov Decision Processes. In *Reinforcement Learning*, volume 12 of *Adaptation, Learning, and Optimization*, pages 3–42. Springer, Berlin, Heidelberg, Germany, 2012.

Richard Vaughan, Neil Sumpter, Andy Frost, and Stephen Cameron. Robot Sheepdog Project Achieves Automatic Flock Control. In *Proceedings of the Fifth International Conference on the Simulation of Adaptive Behaviour (SAB 1998)*, pages 489–493, 1998.

Richard T. Vaughan, Neil Sumpter, Jane V. Henderson, Andy Frost, and Stephen Cameron. Experiments in Automatic Flock Control. *Robotics and Autonomous Systems*, 31:109–117, 2000.

Christopher Vo, Joseph F. Harrison, and Jyh-Ming Lien. Behavior-Based Motion Planning for Group Control. In *Proceedings of the 2009 International Conference on Intelligent Robots and Systems (IROS 2009)*, pages 3768–3773, 2009.

Branko Šter and Andrej Dobnikar. Adaptive Radial Basis Decomposition by Learning Vector Quantization. *Neural Processing Letters*, 18(1):17–27, 2003.

Gordon L. Walls. *The Vertebrate Eye and Its Adaptive Radiation*. Cranbrook Institute of Science, Bloomfield Hills, MI, USA, 1963.

Ko-Hsin Cindy Wang and Adi Botea. Fast and Memory-Efficient Multi-Agent Pathfinding. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pages 380–387. AAAI, 2008.

Christopher J. C. H. Watkins and Peter Dayan. Q-Learning. *Machine Learning*, 8: 272–292, 1992.

Cristopher J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.

B. P. Welford. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*, 4(3):419–420, 1962.

Shimon Whiteson and Peter Stone. Evolutionary Function Approximation for Reinforcement Learning. *Journal of Machine Learning Research*, 7:877–917, 2006.

Shimon Whiteson, Matthew E. Taylor, and Peter Stone. Adaptive Tile Coding for Value Function Approximation. Technical Report AI-TR-07-339, University of Texas at Austin, 2007.

Marco A. Wiering. *Explorations in Efficient Reinforcement Learning*. PhD thesis, Utrecht University, 1999.

Marco A. Wiering. Convergence and Divergence in Standard and Averaging Reinforcement Learning. In *Proceedings of the Fifteenth European Conference on Machine Learning (ECML 2004)*, pages 477–488, 2004.

Michael Wooldridge. *An Introduction to MultiAgent Systems*. Wiley Publishing, Chichester, UK, second edition, 2009.

Chung-Do Yang, Der-Tsai Lee, and Chak-Kuen Wong. On Bends and Lengths of Rectilinear Paths: A Graph-Theoretic Approach. In *Proceedings of the Second Workshop on Algorithms and Data Structures (WADS 1991)*, pages 320–330, 1991.