Reinforcement Learning for Planning in High-Dimensional Domains

Reinforcement Learning für Planungsprobleme in hochdimensionalen Zustandsräumen

Bachelor-Thesis von Dominik Notz aus Frankfurt am Main

September 2013

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Reinforcement Learning for Planning in High-Dimensional Domains
Reinforcement Learning für Planungsprobleme in hochdimensionalen Zustandsräumen

Vorgelegte Bachelor-Thesis von Dominik Notz aus Frankfurt am Main

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: M.Sc. Herke van Hoof

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 20. September 2013

(D. Notz)

Abstract

In this bachelor thesis we address the topic of reinforcement learning for planning in high-dimensional domains. A high-dimensional state space problem, which is modeled as a Markov Decision Process, cannot be solved with classical techniques such as policy or value iteration any more. Reason for this is that the large state space makes tabular representations of the state space and sweeping over it, which is required by exact solution algorithms, practically infeasible. Therefore, we apply several reinforcement learning algorithms to circumvent this issue. The concept of a factored representation of the Markov Decision Process, aiming at exploiting structure in the given problem to reduce the computational costs, can not be profitably applied to the problem due to too many inherent dependencies. However, our least-squares temporal difference learning approach, which learns the weights of features that are capable of modeling the problem correctly, can find optimal solutions to medium scale problems. Hierarchical reinforcement learning lets us define actions that operate over multiple time steps. Thereby, it provides us with the possibility to force structures on the policies, which fastens the process of finding good ones. Finally, this abstraction enables us to solve large scale problems as well and it make knowledge transferable between different problems.

Zusammenfassung

In dieser Bachelor Arbeit befassen wir uns mit Reinforcement Learning für Planungsprobleme in hochdimensionalen Zustandsräumen. Solche Probleme, die als Markow-Entscheidungsprobleme modelliert werden, können nicht mehr mit klassischen Techniken wie Policy- oder Value-Iterationsverfahren gelöst werden. Die Begründung dafür liegt in den großen Zustandsräumen, die sowohl eine tabellarische Repräsentation der Zustände als auch das Iterieren über alle Zustände, das von exakten Lösungsverfahren benötigt wird, in der Praxis unmöglich macht. Um diese Problematik zu umgehen, wenden wir mehrere Reinforcement Learning Algorithmen an. Das Konzept der faktorisierten Markow-Entscheidungsprobleme, das darauf abzielt, dem Problem zugrunde liegende Strukturen auszunutzen, um die Berechnungskosten zu reduzieren, kann aufgrund zu vieler dem Problem inhärenter Abhängigkeiten nicht nutzbringend angewendet werden. Mit unserem Ansatz des Least-Squares Temporal Difference Learning, mit dem wir die Gewichte für Features lernen, die fähig sind, das Problem korrekt zu modellieren, können wir Lösungen für Probleme mittlerer Größe finden. Hierarchisches Reinforcement Learning erlaubt es uns, Aktionen zu definieren, die über mehrere Zeitschritte andauern. Dadurch wird es uns ermöglicht, den zu lernenden Strategien eine Struktur aufzuzwingen, die den Prozess des Findens guter Strategien beschleunigt. Schließlich befähigt uns diese Abstraktion, auch größere Probleme korrekt zu lösen und gesammeltes Wissen auf andere Probleme zu übertragen.

Acknowledgements

During the summer term 2013 at the TU Darmstadt I have written this bachelor thesis, which has been made possible by the support of many people, to whom my sincere thanks is due.

First and foremost, I would like to thank my two supervisors M.Sc. Herke van Hoof and Prof. Jan Peters for having given me the opportunity to work on an interesting research topic and be part of an excellent research group. Herke, as my first supervisor, deserves special recognition for his sympathetic ear, his highly competent remarks and his calm and friendly manner. I want to thank Jan, my second advisor, for the enlightening input I received during the meetings with him and for his creative and targeted advice.

I would also like to give thank to all the other members of the Intelligent Autonomous Systems group for many inspiring conversations about artificial intelligence, magnificent presentations about their own work and their constructive feedback after my intermediate thesis presentation. Moreover, my gratitude goes to Sabine Schnitt and Veronika Weber, the system administrator and secretary of the institute, for the daily smooth operation of the computer system and the imperceptible handling of bureaucracy.

Many friends have constantly supported, encouraged and entertained me. I highly value their friendship and want to thank them for the many pleasant moments we share.

Finally, I am deeply grateful to my parents and my brother for their full support in all respects throughout my studies.

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations and Acronyms

AI	Artificial Intelligence
DBN	Dynamic Bayesian Network
DP	Dynamic Programming
FMDP	Factored Markov Decision Process
HAM	Hierarchical Abstract Machine
HRL	Hierarchical Reinforcement Learning
ICA	Independent Component Analysis
LP	Linear Programming
LSTD	Least-Squares Temporal Difference
LUT	Look Up Table
MC	Monte Carlo
MDP	Markov Decision Process
MSBE	Mean Squared Bellman Error
MSE	Mean Squared Error
MSPBE	Mean Squared Projected Bellman Error
PCA	Principal Component Analysis
RL	Reinforcement Learning
SMDP	Semi-Markov Decision Process
TD	Temporal Difference

1 Introduction

> *"I hold that AI has gone astray by neglecting its essential objective — the turning over of responsibility for the decision-making and organization of the AI system to the AI system itself. It has become an accepted, indeed lauded, form of success in the field to exhibit a complex system that works well primarily because of some insight the designers have had into solving a particular problem. This is part of an anti-theoretic, or "engineering stance", that considers itself open to any way of solving a problem. But whatever the merits of this approach as engineering, it is not really addressing the objective of AI. For AI it is not enough merely to achieve a better system; it matters how the system was made. The reason it matters can ultimately be considered a practical one, one of scaling. An AI system too reliant on manual tuning, for example, will not be able to scale past what can be held in the heads of a few programmers. This, it seems to me, is essentially the situation we are in today in AI. Our AI systems are limited because we have failed to turn over responsibility for them to them."*
>
> Richard Sutton, *What's Wrong with Artificial Intelligence* (from [45])

Free of doubt, one of the most striking features of human beings is their *intelligence*, since they are believed to be more intelligent than any other known species. Biologists and psychologists have tried to understand how the human brain works for a very long time, whereas research in the field of *artificial intelligence (AI)* goes even one step beyond that and aims at imitating the intelligence of humans and building intelligent agents.

One approach to AI is called *acting rationally*: "A rational agent is one that acts so as to achieve the best outcome or, when there is uncertainity, the best expected outcome" ([40]). According to this approach, agents shall operate autonomously, perceive their environments, adapt to changes and pursue goals.

Since the birth of AI in 1956 lots of progress has been made. In 1997 *Deep Blue*, a chess machine developed by IBM, defeated the reigning World Chess Champion Garry Kasparov in a six-game match [8]. In 2011 a computer system called *Watson*, which was developed by IBM as well, won in the *Jeopardy!* quiz show against two former champions being able to answer questions posed in natural language in real time [21]. Furthermore, nowadays, physical robot teams play soccer and compete against each other in the World Cup Robot Soccer (RoboCup) dealing with planning in dynamic environments, multiagent collaboration, real-time reasoning etc. [30].

In the 1980s three - until then - distinct threads, namely, the one concerning the problem of optimal control, the one concerning temporal-difference methods and the one concerning learning by trial and error came together and produced the modern field of *Reinforcement Learning (RL)*, a specific area of machine learning or AI in general [47]. Especially the thread which is centered on the idea of trial and error learning can be seen as very essential for RL. The so called *Law of Effects*, stating that actions followed by a good outcome have the tendency to be reselected more frequently than actions followed by a bad outcome, underlies much behavior. It is both *selectional*, which means that it involves trying different alternatives and selecting them by comparing their outcomes, and *associative*, meaning that the

alternatives are associated with particular situations. Consequently, learning by trial and error combines *search* and *memory* [47].

Compared to supervised learning agents that need to be told which move to take for every single position they may encounter, agents based on RL should be able to learn evaluation functions by interacting with the environment. This selectional and associative RL approach is crucial for a wide range of problems in complex domains, in which it is not feasable to provide a correct evaluation for a huge number of different positions.

RL approaches have been proven to be quite successful. To give two prodigious examples, *TD-Gammon*, which is based on a RL algorithm, learned with zero built-in knowledge just by playing against itself to play backgammon on a level close to the world's best human players [51], and a helicopter was able to learn a controller for autonomous inverted hovering, which is regarded as a very challenging control problem [37].

1.1 Motivation

We can think of many interesting problems with huge amounts of states, which make them practically unsolvable by precisely specifying everything in detail. However, for many of these kind of problems a specific start and a desired end state are known. For example, we can think of a robot putting different dishes into a dish washer (Figure 1.1), where dirty dishes might be on a table, or a robot folding towels (Figure 1.2) that it grasps from a pile. For those planning problems, it would be desirable to have some algorithms which only consume the start and the end configuration and find a close to optimal path between them by learning from experience.

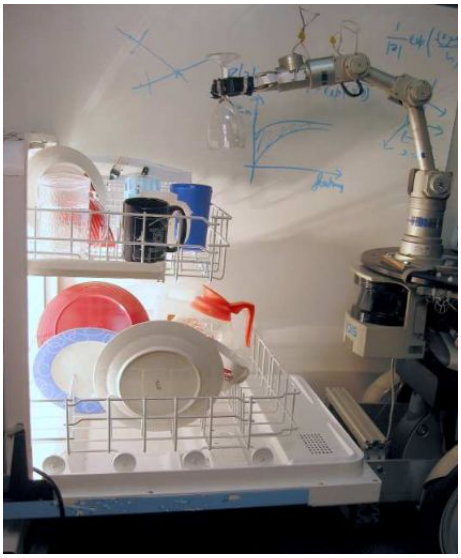


Figure 1.1: Robot Putting Dishes into a Dish Washer (taken from [41]).

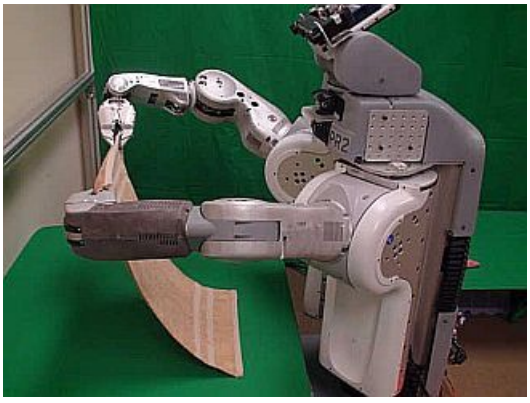


Figure 1.2: Towel Folding Robot (taken from [34]).

Furthermore, for some of these problems, we have to deal with objects that interact with each other. In such cases, robots need to learn affordance models, which consist of objects, actions and effects [35] (Figure 1.3). For such problems, robots would need to learn that when pushing an object which lies flat against an other one, both objects will move. For that reason, it would be admirable to be able to deal with several interacting objects which have to be arranged in a specific way.

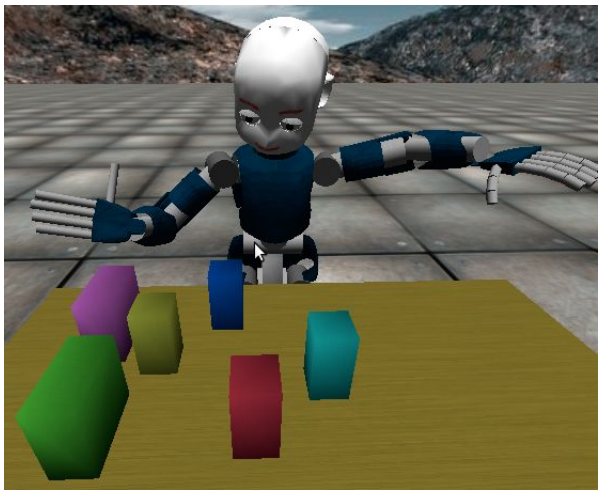


Figure 1.3: Robot Learning Multiple-Object Manipulation (taken from [35]).

Recently, some RL algorithms have been directly applied to raw image pixel data as input states [20]. This application of RL leads us to the idea of a two-dimensional grid configuration representing the pixels of an image. This image is in a particular start state and shall be brought on the shortest possible path in a desired end state. This idea directly leads us to the definition of the example problem from the whole class of planning problems in high-dimensional domains which this thesis deals with.

1.2 Problem Definition

The concrete problem which we try to solve in this work consists of a start and an end configuration of blocks on a two-dimensional grid. Figure 1.4 shows a sample problem instance with three blocks. The goal is to find the *shortest path* between those two states, meaning that we want to minimize the number of actions which have to be chosen to get from the start to the end state. In every state the agent has to choose one of the blocks and an action from the action set $A = \{\text{up}, \text{right}, \text{down}, \text{left}\}$ for this block. The chosen block then will move in the chosen direction under the restriction that a block cannot be moved outside of the game board. In Figure 1.4 pushing block 1 to the right would make it move one field to the right, whereas pushing it to the left would not change anything. Furthermore, when two or more blocks are horizontally or vertically aligned with each other and one block gets pushed against the other one(s), then all blocks will move together in the according direction. For instance pushing block 3 down would make block 2 and block 3 move one field down.

There are some problems in which executing the actions for shortest path yields only single block moves. Such problems can be considered to be easily solvable, since in every step one block gets pushed closer to its goal position. For those problems, there is only one local optimum which therefore cor-

responds to the global one. However, more interesting problems are those in which the shortest path consists of moving several blocks together over the game board. In Figure 1.4 for example it would be beneficial to move block 2 and 3 together to the right. For such problems, there might exist a local opimum which is different from the global one.

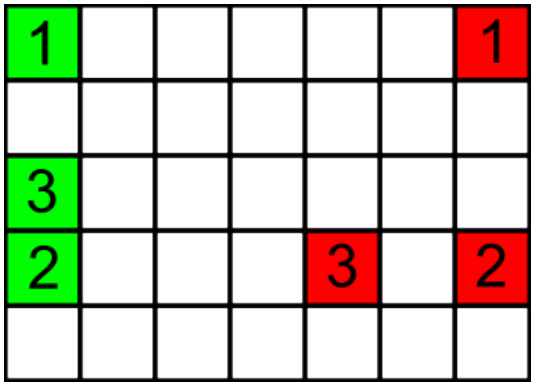


Figure 1.4: The Problem: Finding the shortest path between the start (green) and the end (red) configuration.

1.3 Structure of the Work

In chapter 1 we have named several impressive applications of AI and RL and we have given both a definition of and a motivation for our problem. In chapter 2 we make use of search algorithms to approach this problem, before modeling it as a Markov Decision Process in chapter 3, where we also introduce RL and solve small problem instances with value iteration. Chapter 4 is dedicated to the idea of factorizing the problem to exploit inherent structure. In chapter 5 we derive the general and the least-squares temporal difference learning approach, introduce the concept of value function approximation and select appropriate features for it. Afterwards, we define a set of problems, which we use for evaluating our method, and present several techniques to improve the performance of our algorithm. In chapter 6 we introduce hierarchical reinforcement learning and apply two variants of it, namely, options and hierarchies of abstract machines to our problem and evaluate these approaches, too. Finally, in chapter 7 we summarize our results, discuss advantages and disadvantages as well as opportunities and obstacles of the presented methods and make some suggestions for future research possibilities.

2 Approaching the Problem: Search

The problem of having got a start or initial state and an end state and looking for the shortest path between them can be treated as a graph search problem. Regarding the given problem from this perspective, the initial state, the available actions and the deterministic transition model implicitly define the *state space*, which is the set of all states that are reachable from the initial state by any sequence of actions [40].

Since the state space we are dealing with is very large - that is to say $\frac{\#fields!}{(\#fields-\#blocks)!}$ or respectively $\binom{\#fields}{\#blocks}$ for problems with and without differentiating between blocks - a simple *Dijkstra* graph search algorithm, which runs in $O(|V|^2)$ with $|V|$ being the number of vertices (states), will not be able to find solutions in a reasonable amount of time for rather large problems. For that reason, we will in the following consider both *A* Search* and *Beam Search* implementations instead.

2.1 A* Search

Dijkstra's algorithm finds the shortest path between the start node (state) and every other vertex (state). However, we are not interested in finding the shortest path to every other state, but only in finding the shortest path to the goal state. For this reason A* uses a *heuristic* and expands the node with the lowest expected total cost $f(n)$ with

$$f(n) = g(n) + h(n), \tag{2.1}$$

where $g(n)$ and $h(n)$ correspond to the total path cost to reach the current node n and the estimated (heuristic) cost from node n to the goal state. Because of the fact that A* uses some additional knowledge to be able to do a goal-directed search, we say that it is an *informed best-first* strategy [14].

2.1.1 Finding a Heuristic

Since the heuristic that A* uses is the only difference to Dijkstra's algorithm, it can be regarded as the most essential part of the algorithm. When the heuristic is chosen to be *admissible*, meaning that it is optimistic and never overestimated the cost of reaching the goal state, it can be proven very easily that A* finds the *optimal* solution and expands fewer nodes than any other admissible search algorithm that uses the same heuristics [14].

We now want to design a simple admissible heuristic and then evaluate it practically. For the case in which we treat all blocks as distinguishable from each other and assign to every block a specific goal position, an admissible heuristic would be the sum of the distances between every block's current and goal position divided by the number of blocks. For the case in which we treat the blocks as non-distinguishable from each other, we could compute all permutations of assigning the blocks to goal positions, take the one with the minimal sum of the distances from the current to the goal positions and

divide this sum again by the number of blocks. In the ideal case, in which all blocks would be moved in every step closer to their goal positions, these heuristics would correspond to the true distance function, otherwise they would be too optimistic. Therefore these heuristics are admissible.

2.1.2 Evaluation of A*

To find out how good we can do with A*, we have created a simple evaluation problem, which can be seen in Figure 2.1(a). The fields filled with green and red color represent the current (start) and goal positions of the blocks. A block which has already reached its goal position will be marked as an "X" on the according field.

The evaluation problem game board is of size 4×5, has 4 blocks and therefore $\binom{20}{4} = 4845$ states, since we do not differentiate between the blocks. The heuristic cost from the start to the end state is 2. In contrast to that the shortest path, which is found by A* and displayed in Figure 2.1, is of length 6. This implies that nearly all nodes (states) which can be reached in 4 steps will be expanded before the solution is found. Taking into consideration that this tree of expanded nodes grows exponentially with the path length, A* does not seem suitable for larger problems, at least not with the heuristic designed in section 2.1.1.

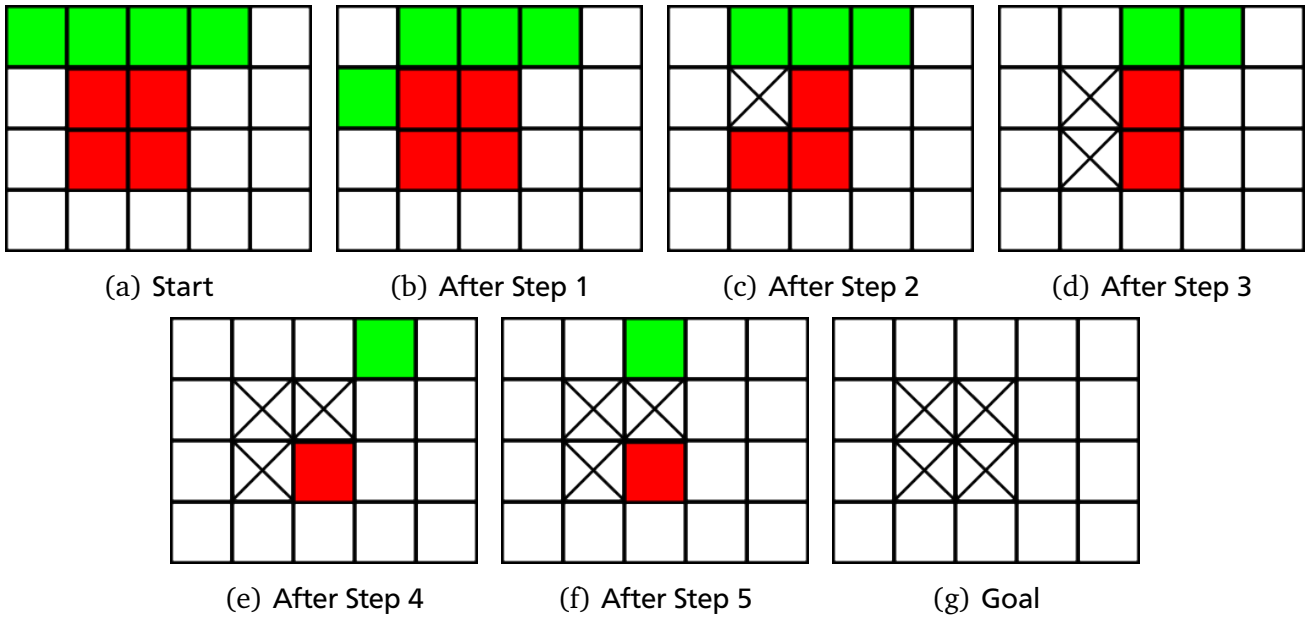


Figure 2.1: A* Evaluation: The shortest path has been found.

On the one hand, to be able to deal with larger problems we are in need of a much better designed heuristic, which is partly capable of figuring out which blocks can be moved together. On the other hand, we prefer not to use too much domain specific knowledge, but rather want our agent to be able to figure out good solutions itself. Therefore, the A* approach does not seem to be suitable for our problem.

2.2 Beam Search

Due to the fact that A* does not scale adequately to large problems, we now take a look at another heuristic search algorithm. Beam search is an optimization of the best-first search algorithm, sacrificing the property of always finding the optimal solution for a - depending on the used heuristic - linear scaling with the problem size. Starting from the start state, beam search builds a search tree, but only stores at every level the k best states according to a heuristic. Hence, both the memory consumption and the time to find a solution are linear to the found path length. k is also called the *beam width*.

For evaluating a state, we have used the same heuristic as for A* in section 2.1, but added a very small random component to deal with the fact that there would otherwise always be many neighboring states with the same heuristic value, because when moving a single block closer to its goal position, the heuristic function will yield the same value, no matter which block has been moved. Furthermore, we set the beam width to be twice as large as the number of blocks. It intuitively makes sense to scale the beam width with the number of blocks, as the number of possible actions grows linearly with the number of blocks on the game board.

2.2.1 Evaluation of Beam Search

At first, we let our beam search implementation run on the evaluation problem from section 2.1.2. The optimal path is found in a very short period of time. As a result, we construct another problem which is illustrated in Figure 2.2, consisting of two blocks. For obtaining the shortest path, the agent would need to make use of synergies, meaning the movement of several blocks at once. In more detail, the agent would have to align both blocks horizontally, first, then move them together to the right hand side of the game board and afterwards up / down to their goal positions. Yet, the agent does not discover this shortest path, since it starts with a non-greedy action in the beginning, namely, aligning the two blocks which increases the heuristic costs and is therefore not one of the $k = 2 \cdot 2 = 4$ best next states.

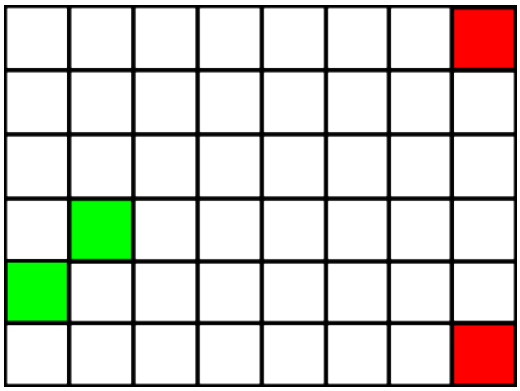


Figure 2.2: Beam Search Evaluation: The shortest path has not been found.

In [53] Zhou and Hansen describe a transformation of beam search into an algorithm, which they call *beam-stack search*, that is guaranteed to find the optimal solution. The idea presented is to inte-

grate systematic backtracking with beam search such that an *anytime algorithm* is created. This anytime algorithm starts with finding a first solution via search and then iteratively improves this solution via backtracking using a memory-efficient beam stack to finally converge to the optimal solution. At first, this approach may sound very promising, since a first solution is found very quickly and depending on the time that we let the anytime algorithm run this solution gets better and better. However, in general, the optimal solution we are looking for often consists of a number of non-greedy moves to align blocks. Consequently, the optimal path will often be quite far away from the initially found solution and will not be discovered by beam-stack search in a reasonable amount of time, either.

Search in general can be regarded as a moderate first approach to the given problem. Nevertheless, it is unlikely to find a good solution to large problems using search algorithms and not giving them lots of handcrafted prior knowledge about the problem. For that reason, we will now follow other approaches and regard the problem as a Markov Decision Process.

3 Markov Decision Processes and Reinforcement Learning

Markov Decision Processes (MDPs) provide a mathematical framework for sequential decision problems in which actions have uncertain effects, inducing stochastic transitions between states [5]. At present, MDPs are used in the context of many different optimization problems and play therefore an important role in the field of AI.

A fundamental idea that underlies nearly all theories of learning and intelligence is learning from interaction. *Reinforcement Learning (RL)* is a computational formalization of this approach that is focused on goal-directed learning from interaction [47], aiming at learning optimal policies. For that reason, RL differs from *supervised learning*, which is based on learning from labeled examples.

3.1 Definition of MDPs

A MDP can be defined as a tuple $\langle S, A, R, T \rangle$ where S represents a finite set of states, A a finite set of actions, R a reward function $R : S \times A \times S' \to \mathbb{R}$ and T a state transition function $T : S \times A \times S \to [0, 1]$, assigning probabilities to all possible transitions between states when an action has been chosen. Transitions are assumed to be *Markovian*, meaning that the probability of ending up in the state s' only depends on the current state s and the chosen action a ($P(s'|s, a)$), but not on the history of earlier states or actions, which makes MDPs sequential. In addition, the environment is assumed to be *fully observable*, which denotes that the agent is able to sense its current state.

Figure 3.1 shows a sample MDP with two states S_0 and S_1. In both states, there are two actions, a_0 and a_1, available. Choosing action a_0 in state S_0 or a_1 in state S_1 respectively, makes the agent stay in the current state and receive a reward of 0, whereas choosing action a_0 in state S_1 changes the state to S_0 with a reward of 1. When the agent is in S_0 and chooses a_1, it will end up in state S_1 and receive a reward of 1 with probability 0.5 and stay in S_0 and receive a reward of -1 with probability 0.5 as well.

The goal of the agent is to maximize its reward. Therefore, the agent searches for a deterministic policy $\pi(s)$ which states for every state s which action to choose when being in s. When assigning a value $V_\pi(S)$ to every state of a MDP which corresponds to the additive, discounted (discount factor $\gamma \in [0, 1]$) future rewards, obtained when acting according to a policy π, we can express the optimal policy, denoted by π^*, which maximizes the expected utility, with

$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s'|s, a) V_{\pi^*}(s'). \tag{3.1}$$

In the next sections, we will present algorithms for finding such an optimal policy.

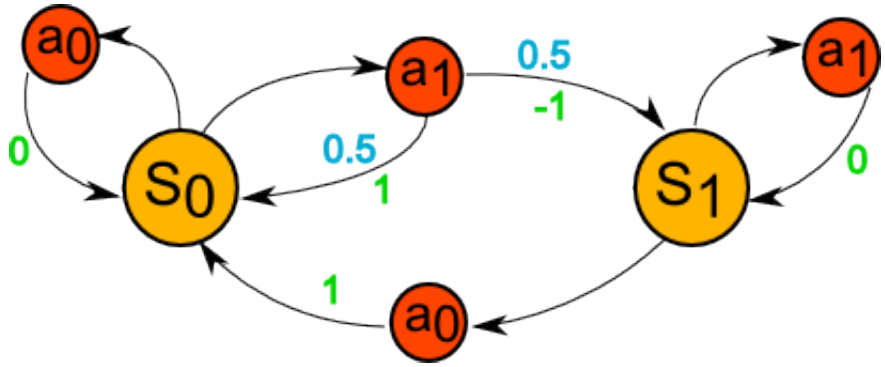


Figure 3.1: Sample MDP with two states S_0 and S_1 and two available actions a_0 and a_1 in both states. The rewards are depicted in green color and the transition probabilities in blue color, whereas transition probabilities of 1 are left out.

3.2 Elements of RL

Beyond the agent and the physical or virtual environment the agent acts in, there are four main sub elements of RL [47]. These are the policy π, specifying the agent's behavior, a reward function R, a value function (either a state-value function V or an action-value function Q) and optionally a model of the environment described through a transition function T. As we can undoubtedly see, these elements directly correspond to those which define MDPs (reward and transition function) and which are used for solving them (value function, policy). That's why RL problems can be well modeled as MDPs [28].

Figure 3.2 shows the agent-environment interaction in RL. The agent interacts with its environment by first executing an action and then obtaining a state, which corresponds to the agent's new state in the environment, and a reward.

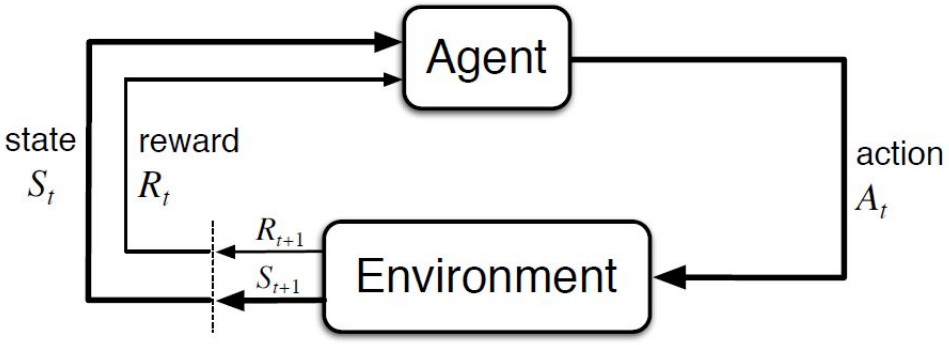


Figure 3.2: Agent-Environment Interaction in RL (taken from [47]).

In RL we can in general differentiate between two approaches or strategies, namely, model-based and model-free. Following the model-based strategy the agent tries to learn a model of the MDP, that is to say the rewards and transition probabilities between states. In contrast to that, an agent following a model-free strategy does not explicitly learn the transition probabilities. In our case, the agent is always

able to compute its next state when executing a specific action. Consequently, we are dealing with a model-based approach and are already given a correct model of the environment so that we do not have to learn such a model.

3.3 Bellman Equation

The optimal policy for a problem cannot be computed directly. Therefore, a value is assigned to every state such that the optimal action to choose would be the one that maximizes the sum of the expected value of the next state and the expected reward received on the transition between the states. Such a policy is called *greedy* with respect to the optimal value function. This implies that the state value function for a given policy π can be expressed recursively, which is stated in the *Bellman equation*:

$$V_\pi(s) = \sum_a \pi(s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_\pi(s')]. \tag{3.2}$$

In our problem domain, particular states and actions always lead to specific next states, or in other words, the environment we are operating in is *deterministic*, implying that our agent does not need to care about *uncertainty*. When we transfer this knowledge to the Bellman equation, we can rewrite and simplify this equation and obtain

$$V_\pi(s) = \max_{a \in A(s)} [R(s,a,s') + \gamma V_\pi(s')]. \tag{3.3}$$

The Bellman equation is actually a system of linear equations. Let N be the number of states. Then there are N equations - one for every state value - and N unknown values to be determined. Solving this system of linear equations with some mathematical method is practically infeasible for a large value of N (our problems have large values for N as stated in chapter 2), because the today's best known algorithm that solves a system of linear equations runs with $O(N^{2.376})$ [9].

3.4 Dynamic Programming

The term *Dynamic Programming (DP)* refers to a collection of algorithms that require a complete and accurate model of the environment, and which then can determine optimal policies. When we want to determine the value function for a given policy π - we call this process *policy evaluation* - we can do this by iteratively evaluating the Bellman equation (equation 3.2) starting from some initial values and obtain

$$V_{k+1}(s) = \sum_a \pi(s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')] \tag{3.4}$$

$$= \sum_a \pi(s)[R(s,a,s') + \gamma V_k(s')], \tag{3.5}$$

where the second equation refers to our deterministic case. This update rule has been proven to converge for $k \to \infty$ to V_π under the condition that either $\gamma < 1$ or the guarantee of eventual termination

from every step [47].

The old value of a state is replaced with a new value that is based on the old values of all successor states. We call this backup, which is illustrated in Figure 3.3, a *full backup*. In this illustration states are represented as white circles and actions as black circles. For the computation of the new value of the state s, we compute for every possible action a the reachable successor states s' and with these for every action a new value and maximize over these values.

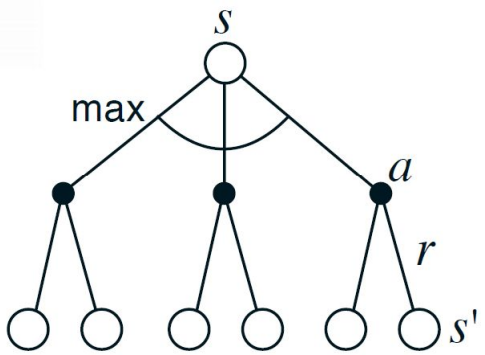


Figure 3.3: Dynamic Programming Backup Diagram (taken from [47]).

As a matter of fact, we are not interested that much in just evaluating a given policy, but rather in computing the optimal policy or improving a policy, respectively. This can be done by linking every state with the action that maximizes its value. The *policy improvement theorem* (see [47]) assures that the new policy will be better than the old one, meaning that the new value of every state will be greater or equal than the old value. This process of evaluating a policy is repeated until convergence.

3.4.1 Policy Iteration

The idea that underlies the *policy iteration* algorithm is pretty simple. We start off by evaluating a policy and then alternately improve and evaluate it until convergence is reached, which is attained quite fast:

- For the current policy $\pi(s)$, compute $V_\pi(s)$.
- For each action a, compute $Q_a = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_\pi(s')$.
- Update the policy $\pi(s) = \arg\max_a Q_a(s)$.

3.4.2 Value Iteration

Value iteration is a variant of policy iteration that stops the policy evaluation already after the first step, which enables us to combine both improvement and evaluation in a single sweep:

$$V_{k+1}(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')] \tag{3.6}$$

$$= \max_a [R(s,a,s') + \gamma V_k(s')], \tag{3.7}$$

where the second equation refers to our deterministic case.

3.5 Solving Small Problems with Value Iteration

We now want to apply the value iteration algorithm to our problem. To do that, we first need to transform the problem into a MDP. The set of states corresponds to all possible block configurations on the game board. We will represent a state as a vector of the row and column positions of the blocks. For the case of not differentiating between blocks, this vector will be sorted after every transition, whereas for the case of differentiating between blocks we can omit this step. The set of actions contains for every block the pushing of it into one of the four directions. Therefore, we can represent an action as a pair of a block and a direction.

Our actual goal, that is to say, finding the shortest path between the start and end configuration, can be transferred using the reward function. On every transition between states we assign a reward of -1 and define our goal state to be a terminal state such that minimizing the path length equals maximizing the agent's reward. Last but not least, the transition function is defined to be deterministic, assigning the transition from the current to the next state, which is reached when a specific action is executed, a probability of 1 and setting all other probabilities to 0. Due to the fact that we can reach the terminal state from every other state with a finite sequence of steps, we neglect the discount factor λ by setting it to 1.

In a next step, we need to compute a *look up table (LUT)* which stores for every state and every action the next state. Once again, this can be done, because we know the model of the environment. We now keep sweeping over all states and apply the value iteration formula given in equation 3.6 until we reach convergence. We formally define that convergence is reached, when the maximum absolute difference between an old state value and its new one falls below some small threshold, implying that the value function does not change much any more.

With this approach, we are able to solve small problems quickly. Compared to the search approach in chapter 2, the value iteration algorithm returns a solution which is independent of the start state. An example solution for the problem given in Figure 2.1 is shown for an arbitrarily chosen start state in Figure 3.4.

Nonetheless, this approach is regarded as computationally expensive [28] and does not scale for large problems, which are usually more interesting. A large problem may consist of a game board of the size 100×100 and may have 10 blocks and therefore has $\binom{100\times100}{10} = \binom{10000}{10} \approx 2.74 \times 10^{33}$ states when not differentiating between blocks. It is obvious that we cannot even create a LUT or explicitly represent this huge amount of states. However, the framework of MDPs and RL provides us with the possibility to explore and try out some more advanced approaches. These aim at requiring less computational effort by avoiding exhaustive sweeps over the state space through restricting the computation to interesting states, by simplifying the backup and by a more compact representation of value functions and policies [3]. In the next chapters we will take a closer look at several of these approaches.

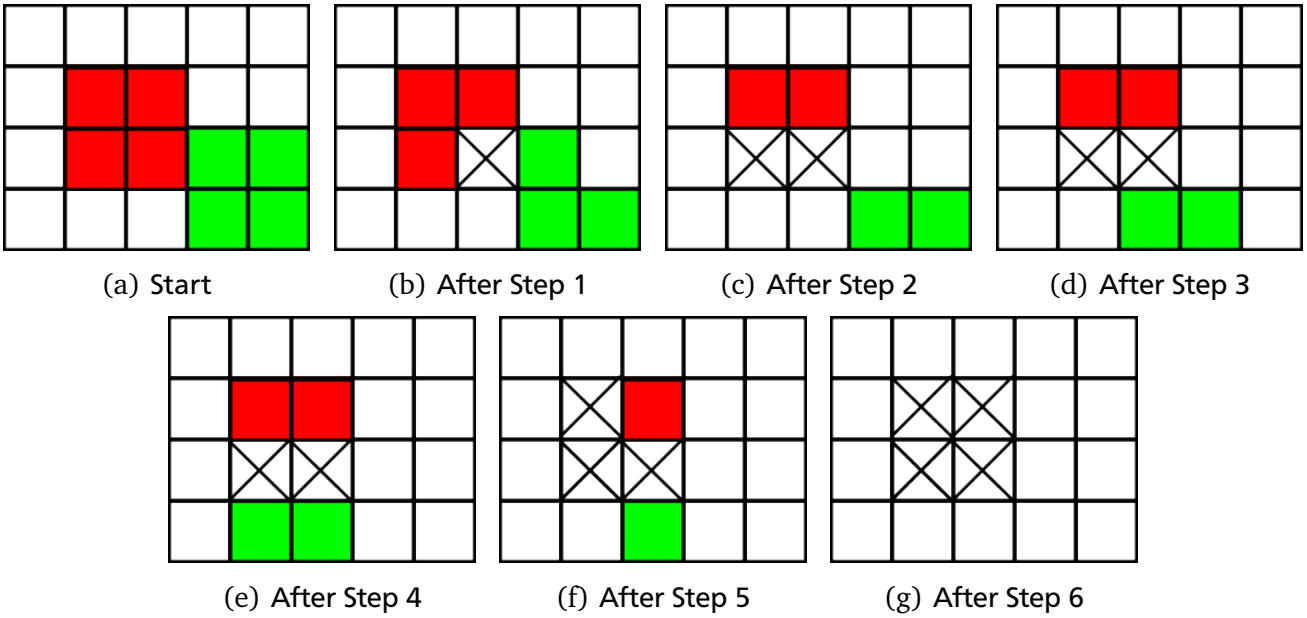


Figure 3.4: The Value Iteration Algorithm Evaluation: Small problems can be solved quickly. An example of the shortest path from an arbitrarily chosen start state is displayed.

4 Factored MDPs

A first concept that we would like to apply to our problem is the one that deals with a factorized representation of the MDP and is therefore referred to as *Factored Markov Decision Process (FMDP)*. The idea to represent large MDPs through a factored model to exploit structure was first proposed by Boutelier et al in 1995 [4]. The theory behind this concept is to make use of *additive* structures of the large system, that is to decompose the system into only locally interacting smaller components, which can be solved almost independently from each other and then can be added in the end again. This implicit dependency information in the factored representation often enables an exponential reduction in the representation size of the MDP and much more efficient solution algorithms [22, 12].

Since the basic algorithm had been published, many improvements and extensions have been developed. In 2002 Guestrin et. al have proposed an approximate *linear programming (LP)* approach for model-based RL and FMDPs [23]. One year later, the approach was further developed to exploit even more structure [22]. Then SDYNA, a framework for FMDPs has been proposed, which does not need any initial knowledge of the structure, but learns a model through the combination of incremental planning and supervised learning [15, 44, 16]. In addition, research on the distributional execution of factored policies by multiple agents with only little communication has been done [39].

We want to start by taking a closer look at factored representations of MDPs, then discuss how an approximate DP approach can solve them efficiently and afterwards apply it to our given problem.

4.1 Factored Representations of MDPs

We will use the representation of FMDPs that follows the one of Koller and Parr [32, 31]. For FMDPs we need a factored representation of states, an additive reward function and a value function approximation, for which usually a linear one is chosen. We will now discuss these issues in more detail.

4.1.1 Factored State Representations

A set of states is described via a set of random variables $\boldsymbol{X} = \{X_1, \ldots, X_n\}$. Each X_i takes a value from some finite domain $\mathrm{Dom}(X_i)$. Hence, the set of all N states, which is described by $\mathrm{Dom}(\boldsymbol{X})$ is exponential in the size of variables. Due to this exponential size, the transition model cannot be represent as a matrix any more, but will be represented with a *dynamic Bayesian network (DBN)* instead [13]. The Markovian state transition model τ defines the probability distribution over the next state, which is described by an assignment of values to X_i', given the current state, which is denoted by the variables X_i. The *transition graph* G_τ of the DBN is then a two-layer directed acyclic graph whose nodes are $\{X_1, \ldots, X_n, X_1', \ldots, X_n'\}$. Figure 4.1 shows an example DBN with four state variables. For instance, the next value of the variable X_1 only depends on the current values of X_1 and X_2.

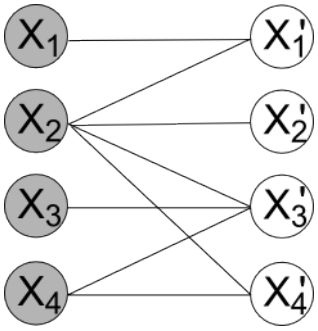


Figure 4.1: Simple DBN (after [32]).

For every action of the MDP a DBN is constructed and with each node of the DBN a conditional probability distribution $P_\tau(X_i'|Parents_\tau(X_i'))$ is associated, where $Parents_\tau(X_i')$ denotes the parents of X_i' in the transition graph G_τ. The transition probability $P_\tau(\boldsymbol{x}'|\boldsymbol{x})$ is then $\prod_{i=1}^{n}(x_i'|\boldsymbol{u}_i)$ with $\boldsymbol{u}_i$ being the values in $\boldsymbol{x}$ of the variables in $Parents_\tau(X_i')$. The transition dynamics of a MDP can then be defined by a set of DBN models $\tau_a = \langle G_a, P_a \rangle$, one for each action a.

4.1.2 Additive Reward Functions

We do also need to provide a compact representation of the reward function. This is done under the assumption that the reward function can be easily split up into a set of r localized reward functions R_i. These local reward functions are only allowed to depend on the current state, but not on the previous state and the chosen action any more (compare it to the reward function definition in section 3.1). Each of these local reward functions should only depend on a small subset of variables $\boldsymbol{W}_i \subset \{X_1, \ldots, X_n\}$. The reward the agent receives when it is in state $\boldsymbol{x}$ is then defined to be $\sum_{i=1}^{r} R_i(\boldsymbol{x})$.

4.1.3 Linear Value Functions

Finally, we need an approximation for the value function and choose a linear one. Later, in section 5.3, we will discuss the topic of value function approximation in more detail. For now, it is sufficient to know that we approximate the value function V by the weighted sum of k basis functions $H = \{h_1, \ldots, h_k\}$ which all dependent only on a subset of states: $V = \sum_{i=1}^{k} w_i h_i$, where $\boldsymbol{w} = (w_1, \ldots, w_2)$ are the weight coefficients. The assigned values to all N states can then be written as a matrix-vector-multiplication $A\boldsymbol{w}$ where A is a $N \times k$ matrix whose rows correspond to states and contain the respective values of the basis functions.

4.2 Solving FMDPs

After having explained how FMDPs can be represented, we now want to discuss how to solve them. Since this is an arduous task, we will not talk about everything in full detail, but rather put the emphasis on a general understanding of the underlying concepts. Through the use of a linear value function approximation, the only variables that need to be determined are the weight coefficients $\boldsymbol{w}$. Hence, we

will first derive a closed form solution for these weights and afterwards try to gain an understanding of how we can compute this solution efficiently by exploiting the factored representation.

4.2.1 Closed Form Solution For Weight Coefficients

When we apply the fact that our reward function only depends on the current state to the Bellman equation, which was stated in equation 3.2, and expand the equation to include all states at once, we obtain the fixed point equation

$$V_\pi = R + \gamma P_\pi V_\pi. \tag{4.1}$$

Here, V_π and R are vectors of length N and P_π is an $N \times N$ matrix. We can use this equation and solve it via policy iteration as discussed in section 3.4.1. When we replace V_π through the approximated values $A\boldsymbol{w}$ and try to minimize the Bellman error via a least-squares approximation [32], we receive

$$\boldsymbol{w} \approx (A^T \Lambda A)^{-1}[\gamma A^T \Lambda P_\pi A\boldsymbol{w} + A^T \Lambda R], \tag{4.2}$$

where Λ is a diagonal weight matrix, whose entries correspond to the state visit frequencies. Let $B = \gamma(A^T \Lambda A)^{-1} A^T \Lambda P_\pi A$, then this equation can be transformed into

$$(I - B)\boldsymbol{w} \approx (A^T \Lambda A)^{-1} A^T \Lambda R. \tag{4.3}$$

If $(I - B)$ is invertible, which is the case for all but finitely many λ, we can provide a solution to this equation [32].

4.2.2 Efficient Computation of the Solution

Until now, we have not won anything through the factorization, since the solution of equation 4.3 still requires for example the multiplication of A^T and A (for the case $\Lambda = I$), which are both of size $N \times N$, and N is very large. However, we are able to exploit the structure of the FMDP.

When taking a closer look at $A^T A$, we realize that this is nothing else than the computation of all dot products $< \boldsymbol{h}_i, \boldsymbol{h}_j > \forall i, j \in \mathbb{N} \wedge i, j \leq k$ with $\boldsymbol{h}_i$ being a vector containing the h_i values for all N states. As we have already stated earlier, the basis functions h_i only depend on a small subset of all states. When we restrict $\boldsymbol{h}_i$ to $\boldsymbol{Y}$ and $\boldsymbol{h}_j$ to $\boldsymbol{Z}$ and let $\boldsymbol{W} = \boldsymbol{Y} \cup \boldsymbol{Z}$, then we only need to compute

$$< \boldsymbol{h}_i, \boldsymbol{h}_j >= \frac{|\mathrm{Dom}(\boldsymbol{X})|}{|\mathrm{Dom}(\boldsymbol{W})|} \sum_{\boldsymbol{w} \in \boldsymbol{W}} h_i(\boldsymbol{w}) \cdot h_j(\boldsymbol{w}). \tag{4.4}$$

In other words, we only need to consider all the cases in the domain $\boldsymbol{W}$ and then can multiply the result by the total number of occurrences of these cases. Under the assumption that $\boldsymbol{W}$ is substantially smaller than $\boldsymbol{X}$, the computational cost will be exponentially lower than the exhaustive enumeration [32].

Not only $A^T A$ can be computed efficiently, but also $P_\pi A$. We can split this matrix-matrix multiplication up into k matrix-vector multiplications $P_\pi \boldsymbol{h}_i$. For these computations we make use of the DBN. Assuming that $\boldsymbol{h}_i$ is restricted to $\boldsymbol{Y}$, then for the computation of $P_\pi A$, we only need to iterate over the union of the set of parents of $\boldsymbol{Y}'$, which is called the *backprojection* of $\boldsymbol{Y}$ through τ. Furthermore, for computing $A^T P_\pi A$ we need to be able to efficiently solve $(\boldsymbol{h}_i)^T P_\pi \boldsymbol{h}_j$. As Koller and Parr show in [32] $P_\pi \boldsymbol{h}_j$ contains enough structure that this computation can be done with much less effort than the straightforward way would require, too.

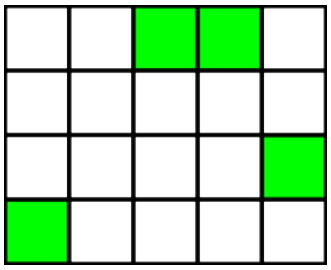
With this knowledge about exploitation of the structure of FMDPs, we can compute $\boldsymbol{w}$ with equation 4.3 efficiently. The next step for the policy iteration is then the computation of the Q values which can be done using again the methods described above. Afterwards, the policy needs to be redefined and for every state the action that maximizes its value has to be chosen. The key to obtain a compactly expressible policy is that most of the components in the weighted combination will be identical for $P_a A$ and $P_d A$, where d is defined to be the *standard action* [32]. It can easily be shown that the differences $Q_a(s) - Q_d(s)$ have a restricted domain, which limits the number of differences that have to be computed. These differences are then sorted in an decreasing order and for a specific state we only need to go through these values starting from the top until we find an assignment of values to the variables in the restricted domain that is consistent with the state's assignment.

At this point, all the steps of the policy iteration algorithm can be executed efficiently. For that reason, we now want to apply the FMDP approach to our problem.

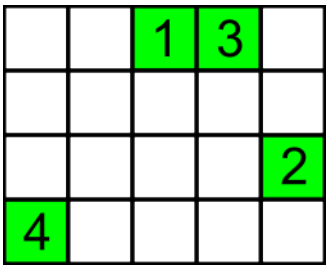
4.3 Application to our Problem

As a first step, we need to find a factored state representation that allows a meaningful representation of the value function by several basis functions that only depend on small subsets of the state variables. We can think of two main state representations. The first one represents the state as a vector of the row and column positions of the blocks. For this representation we have to distinguish two cases. If we want to differentiate between blocks, that is we assign numbers to them, then we have to keep the order of the blocks in the vector the same. In this case, two state representations will be considered to refer to the same state if and only if the two vectors contain the same elements in the same order. If we do not want to differentiate between blocks, that is we do not assign numbers to them, then the situation is the following one. We can say that two state representations refer to the same state if and only if the vectors contain the same elements, which can be in different orders. Figure 4.2 shows one example for each case and Figure 4.3 shows the associated representations of the states.

The second state representation is a bit matrix with the same size as the game board. Ones indicate the presence and zeros indicate the absence of blocks on the positions. With this representation we can only model the case in which we do not differentiate between blocks, since we cannot assign any numbers to the ones. Figure 4.4 shows the bit matrix representation of the state given in Figure 4.2(a).



(a) **An Example State without Differentiating between Blocks.**



(b) **An Example State with Differentiating between Blocks.**

Figure 4.2: Example States with / without Differentiating between Blocks.

1	3
1	4
3	5
4	1

(a) **Vector State Representation for the example from Figure 4.2(a).**

1	3
3	5
1	4
4	1

(b) **Vector State Representation for the example from Figure 4.2(b).**

Figure 4.3: Vector State Representations for the examples from 4.2.

0	0	1	1	0
0	0	0	0	0
0	0	0	0	1
1	0	0	0	0

Figure 4.4: Bit Matrix State Representation for the example from Figure 4.2(a).

After having defined the state representations, we have to create DBNs from them. For the vector state representation the next state of a state variable X_i depends on all state variables, because we can always think of a situation in which two blocks are aligned and pushing one of them will make the second one move as well. The corresponding DBN is shown in Figure 4.5.

Unfortunately, due to these dependencies we cannot save anything for the computation of $P_\pi A$, since the backprojection results in the set of all states $\mathrm{Dom}(\boldsymbol{X})$ as discussed in section 4.2.2. We are aware of the fact that the situations of aligned blocks are compared to the situations of not aligned blocks quite rare, but although we have tried to fix the problem by searching for some structure to exploit, we could

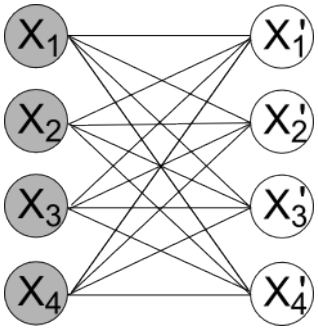


Figure 4.5: DBN for the Vector State Representation.

not find any that was useful. For that reason, we cannot compute the weight coefficients $\boldsymbol{w}$ efficiently.

In the DBN for the bit matrix representation the value of a new state variable depends on the state variables which are in the same row or in the same column. We can even further limit the dependence by taking the number of blocks, respectively the number of ones, into account. For that reason, given an action, the next value of a state variable depends only on #blocks state variables. Now, we need to define the basis functions that are capable of modeling the problem. For being able to do the computations efficiently, these basis functions are only allowed to depend on a small subset of the state variables and exactly this requirement is the problem with the bit matrix representation. We cannot successfully define these basis functions. Since the distance of a block to its designated goal position is the relevant factor, a value function that is able to model this, needs to depend on all state variables, because it has to "know", where the block currently is. Ergo, with the bit matrix representation of states we cannot benefit from the concept of FMDPs, either.

What we actually need is a combination of both models, one for modeling the dependencies between the state variables and one for the computation of the value function. However, the concept of the FMDPs can currently not handle this.

Another approach we have considered was the reduction of the dimensionality of the problem in order to solve a lower-dimensional one and transform its solution back afterwards. We have tried to apply both *Principial Component Analysis (PCA)* [33] and *Independent Component Analysis (ICA)* [26] to instances of our problem. However, we could not find any inherent structure that would have allowed us to put our idea into practice. For that reason, we will try out another approach in the next chapter.

5 Temporal Difference Learning

Another approach aiming at solving the given reinforcement learning problem is *temporal difference learning (TD learning)*. First, we would like to derive the general formula of TD learning. For that reason, we briefly introduce *Monte Carlo (MC) methods* and then combine these with DP to derive TD learning. Afterwards, we talk about linear value function approximation and apply this concept to TD learning. Then, we go one step further and introduce *least-squares temporal difference learning (LSTD learning)*, which can be viewed as an improvement of TD learning. After that, we apply LSTD learning to the given problem and search for appropriate features, which are capable of modeling the given problem. Finally, we evaluate the results and implement several improvements to boost the performance.

5.1 Monte Carlo Methods

Monte Carlo methods are a class of computational algorithms which are used for solving statistical problems. They play an important role in "machine learning, physics, statistics, econometrics and decision analysis" [1]. In the context of RL MC methods refer to the construction sequences of states, actions and rewards through interacting with an environment that does not need to be known. This means that in comparison to DP methods the complete probability distribution of all possible transitions is not needed, but only a model which generates sample transitions [47].

The idea behind MC methods is that the average of the returns received, after having visited a specific state, converges to the expected value. The explanation therefore is that under the assumption of independent, identically distributed estimates of the value of the state the average is an unbiased estimator and the standard deviation of its error falls by $\frac{1}{\sqrt{n}}$ (*Law of Large Numbers*) [43].

As we can see from Figure 5.1, the MC backup diagram goes to the end of an episode, whereas we know from Figure 3.3 that DP backup diagram only consists of one-step transitions. This implies that the estimates for each step are independent, or in an other way, MC methods do not *bootstrap*. Advantage of this is that we are able to deal with a subset of interesting states only, since the computational cost of estimating a state's value is independent of the number of states. In the context of our specific problem this fact provides us with the possibility to only consider an interesting subset of the huge state space.

A simple rule to update the value of a state S_t is

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)], \tag{5.1}$$

with G_t being the accumulated return after time t and α being a step size parameter. This method is called *constant-α MC*.

Figure 5.1: Monte Carlo Backup Diagram (taken from [47]). The white big circles represent states. The black small circles represent chosen actions. The grey square represents the terminal state.

5.2 TD Learning

TD learning as a central idea in RL can be viewed as a combination of MC and DP. After TD learning had been introduced by Sutton in 1988 [46], lots of research has been done on this topic. The high interest in these methods is mainly due to their high data efficiencies. From the done research many new and advanced approaches have evolved.

5.2.1 From DP and MC to TD Learning

Via TD learning an agent learns from raw experience (MC) and updates its estimates based on other estimates (bootstrapping, DP). Therefore, TD methods can make state value updates after each executed step with an observed reward R_{t+1} and a next state $V(S_{t+1})$. The update rule of the *TD(0)* method is

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]. \tag{5.2}$$

Consequently, TD methods unify the advantages of both MC and DP, namely, not requiring a model of the environment (MC) and learning in an incremental online way (DP) [47, 50]. Convergence for TD(0) has been proven with probability 1, if the step-size parameter α satisfies the following conditions [47]:

$$\sum_{k=1}^{\infty} \alpha_k = \infty \qquad \text{and} \qquad \sum_{k=1}^{\infty} \alpha_k^2 < \infty. \tag{5.3}$$

The corresponding backup diagram, which is shown in Figure 5.2, illustrates once more that TD is a combination of MC and DP. The update of the state value is based on a single (DP) sample (MC). Figure 5.3 shows the backups of all so far discussed methods, namely, TD, DP, MC and exhaustive search. Obviously, except for very small state spaces, the exhaustive search is practically infeasible.

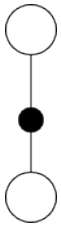


Figure 5.2: TD(0) Backup Diagram (after [47]). The white big circles represent states. The black small circles represent chosen actions.

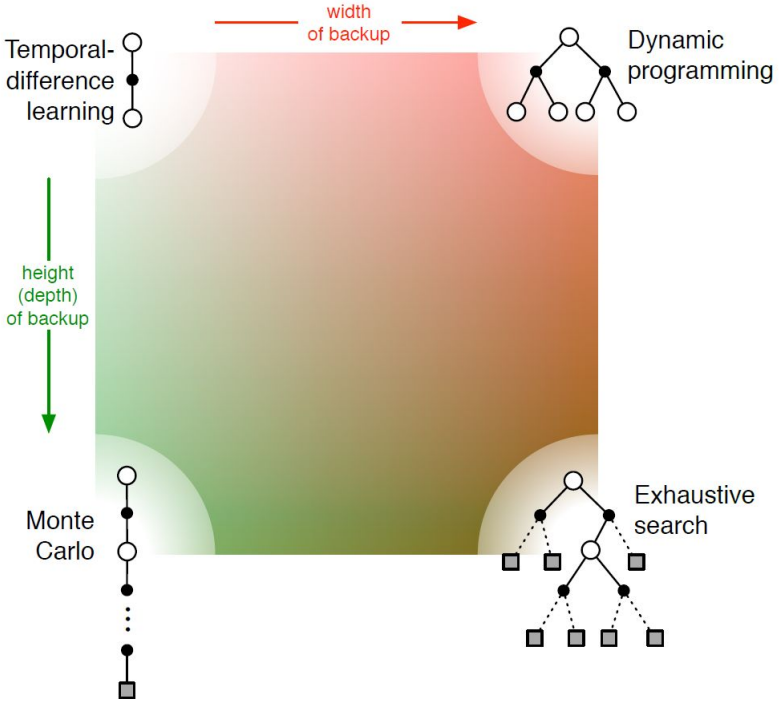


Figure 5.3: Comparison of the Backup Diagrams (taken from [47]). The TD(0) update is based on a single sample. The DP update is based on a full one-step transition. The MC update is based on a sampled episode of transitions. The exhaustive search is based on a full episode of transitions.

5.2.2 Eligibility Traces

The *0* in *TD(0)* refers to another parameter λ ($\lambda = 0$) and the so-called eligibility traces, which are able to bridge the gap between TD and MC methods [47]. In more detail, this means that through the choice of the parameter value λ we can adjust the depth of the backups, which is one for TD methods and equals the sequence length for MC methods.

A formula for an n-step target is given by

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}). \tag{5.4}$$

In the *backward* or *mechanical view*[1] of eligibility traces, which is used to implement the concept of eligibility traces, an additional memory variable $Z_t(S)$ is associated with each state, accumulating how often it has been visited:

$$Z_t(S) = \begin{cases} \gamma\lambda Z_{t-1}(S) & \text{if } S \neq S_t \\ \gamma\lambda Z_{t-1}(S) + 1 & \text{if } S = S_t \end{cases} \tag{5.5}$$

Here, γ is the discount rate and λ acts as a trace-decay parameter. The TD(λ) algorithm that implements this concept will be discussed in section 5.3.2. Empirical research on the value for λ shows that intermediate values work best [6].

5.2.3 Trajectory Sampling and ϵ-Greedy

The main reason for our interest in TD learning is that it allows us to focus only on a small set of states, whereas the DP approach sweeps over the whole state space. As we have shown earlier, the number of states, which our problem consists of, reaches orders of magnitudes of 10^{33} and more. The subset of interesting states are those that lie on short paths from the start to the end configuration. For that reason, we would like to sample from the whole state space according to some unknown distribution that we would like to approximate by always starting from our start state and following the value function, which gets updated in an online manner.

Exploration shall be incorporated via an *ϵ-greedy* action selection policy that chooses the best action according to the value function with probability $1 - \epsilon$ and a random available action with probability ϵ [47], enabling us to find shorter paths. High values for ϵ correspond to many random actions and much exploration, whereas small values for ϵ correspond to few random actions and much exploitation of the best so far found solution.

In Figure 5.4 we compare the performance of a greedy method ($\epsilon = 0$) with two ϵ-greedy methods ($\epsilon = 0.01$, $\epsilon = 0.05$) for a specific problem of ours. As we can see from the diagram, the average run length of the greedy method is always the same, namely, 10. In contrast to that the average run lengths of the ϵ-greedy methods vary, whereas the variance is higher for the higher value of ϵ. However, the ϵ-greedy method with $\epsilon = 0.05$ is able to find a shorter path quickly and outperforms the greedy method after less than 1000 simulated sequences. The ϵ-greedy method with $\epsilon = 0.01$ needs much longer to find the shorter path and starts improving its performance after about 4500 simulated sequences. Details about the algorithm and the implementation will be discussed in the next sections.

5.2.4 Heuristic Search

The term *heuristic search* refers to state space planning methods. The concept behind it is that we would like to make improved action selections by building a tree listing all states that can be reached from the

[1] There also exists a *forward view* which however cannot directly be implemented, since it is acausal, which means that it demands future knowledge about the rewards.

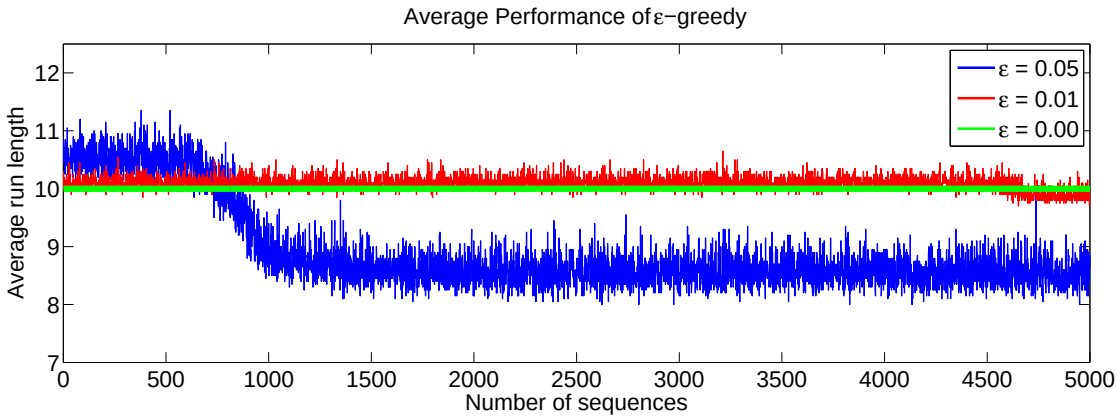


Figure 5.4: ϵ-greedy Performance Comparison for Different Values of ϵ.

current state in k steps. The state values of the leaf nodes are then backed up to the root and instead of choosing the action that maximizes the value of the next state, we choose an action that maximizes the value of the k^{th} next state and call this approach *k-step greedy lookahead* ($k > 1$). Consequence is that we can handle little inaccuracies of our value function better. Deeper search will usually yield better policies due to the focus of backups that immediately follow from the current state [47]. The drawback is a higher computational expense.

5.3 Value Function Approximation

Another consequence from the large number of states we have to deal with is that we cannot have LUTs that store the values of every state. Ergo, we have to use an approximation for the value function. The value for a state will be determined by a function $\hat{V}(S, \theta)$, which is dependent on a weight vector θ.

5.3.1 Gradient-Descent Methods and Linear Value Function Approximation

Gradient-descent methods are widely used and update the weight vector θ in every step by reducing the error between the real and the approximated value of the currently sampled state [47, 2] according to

$$\theta_{t+1} = \theta_t + \alpha \nabla_{\theta_t} [V_\pi(S_t) - \hat{V}(S_t, \theta)]^2, \tag{5.6}$$

where α is the step size parameter which needs to satisfy the equations in 5.3. $V_\pi(S_t)$ is unknown and needs to be approximated by an unbiased estimator, e.g. the expected value of the return. Under these assumptions the weight vector will converge to a local optimum [47].

A special case of gradient-descent methods are *linear* methods, which are viewed as the simplest and probably most commonly used approach. A *linear value function approximation* has the form $\hat{V}(S, \theta) = \theta^T \phi(S)$, with $\phi(S)$ being a feature vector for the state S. The advantage of the use of a linear value function approximation is that a linear value function has only a single local optimum which therefore corresponds to the global one we are looking for. Furthermore, handcrafted non-linear function approximation requires lots of domain knowledge and the agent could get stuck in a local but not global

optimum [11]. The downturn is that problems exist which cannot be modeled well with a linear value function approximation due to the limited representativeness. With this approximation the number of parameters the agent needs to learn is reduced from the number of states to the number of features that we select.

5.3.2 The $TD(\lambda)$ Algorithm

When we apply the linear value function approximation and the concept of eligibility traces to the TD learning equation (equation 5.2), we can create the TD learning algorithm (Algorithm 1), which updates the weights for the features after every simulated sequence. As already stated, we always begin in the start state and then continue choosing next states and receiving rewards based on our ϵ-greedy policy. We accumulate the updates for θ in a variable δ and perform the update after a sequence has been completed.

for $n = 1, 2, \ldots k$ **do**
    $x_t = \text{getStartState}();$
    $\delta = 0;$
    $z_t = \phi(x_t);$
    while $x_t \neq END$ **do**
        //Simulate one step: Receive R_t and x_{t+1}
        $\delta = \delta + z_t[R_t + [\phi(x_{t+1}) - \phi(x_t)]^T \cdot \theta];$
        $z_{t+1} = \lambda z_t + \phi(x_{t+1});$
        $t = t + 1;$
    end
    $\theta = \theta + \alpha \cdot \delta$
end

Algorithm 1: Algorithm for $TD(\lambda)$ Learning (after [6]).

In Figure 5.5 we compare the performance of the TD(λ) algorithm for different values of λ. In comparison to the general statement about the performance differences for different values of λ made in section 5.2.2, this diagram shows that we receive the best performance for the maximum value of λ ($\lambda = 1.0$) and not for an intermediate value (here: $\lambda = 0.5$). This result makes intuitively sense. Our given problem is highly sequential, meaning that we need to align blocks first, to be able to benefit from moving blocks together afterwards. For that reason, we need to backpropagate these benefits to the begin of the sequence and this can be done with a high value for λ ($\lambda = 1.0$).

5.4 Least-Squares Temporal Difference Learning

The actual goal we pursue is approaching as closely as possible the true value function V_π. How well we do is measured by the *Mean Squared Error (MSE)* that is given by

$$MSE(\theta) = \|V_\theta - V_\pi\|_D^2 = [V_\theta - V_\pi]^T D[V_\theta - V_\pi], \tag{5.7}$$

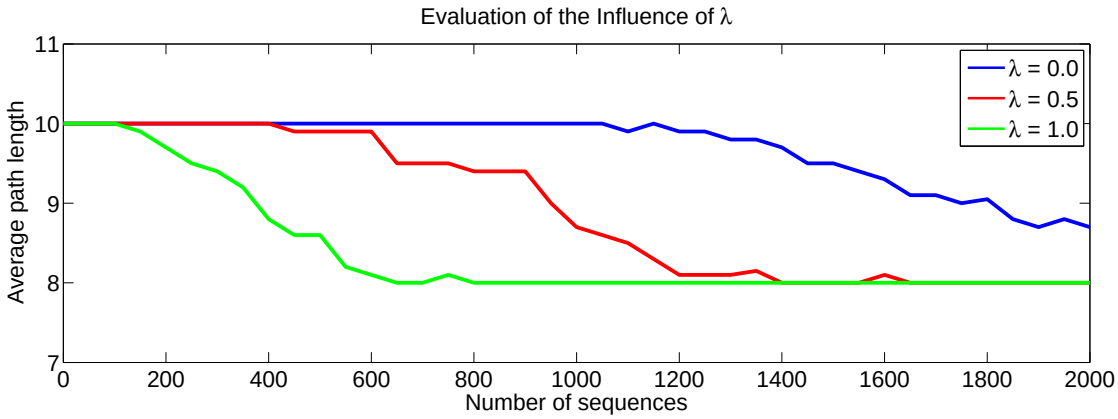


Figure 5.5: TD(λ) Performance Comparison for Different Values of λ.

with D being a diagonal matrix representing the probability distribution, being proportional to the frequency of state visits. Since the true value function is not known, we could measure the difference between the two sides of the Bellman equation (equation 3.2) with the *Mean Squared Bellman Error (MSBE)* given by

$$MSBE(\theta) = \|V_\theta - TV_\theta\|_D^2 = \|V_\theta - \sum_a \pi_\theta(s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_\theta(s')]\|_D^2, \tag{5.8}$$

with T being the so-called *Bellman operator*.

Due to the use of a liner value function approximation, the true value function might not be representable with the chosen features. That's the reason for the introduction of the *Mean Squared Projected Bellman Error (MSPBE)*:

$$MSPBE(\theta) = \|V_\theta - \Pi TV_\theta\|_D^2, \tag{5.9}$$

where Π is a projection operator which projects value functions onto the space of representable functions [11]. The difference between MSBE and MSPBE is illustrated in Figure 5.6.

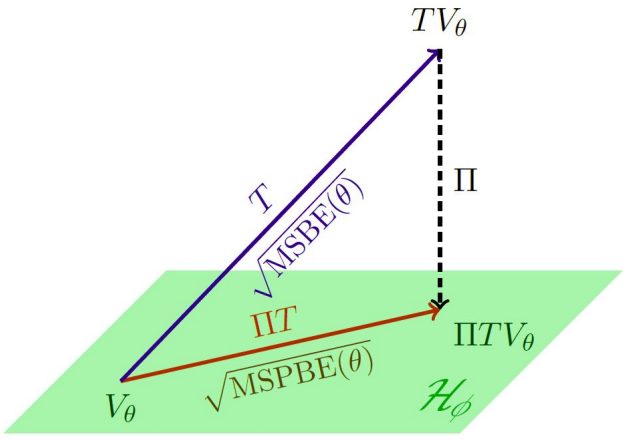


Figure 5.6: Comparison of MSBE and MSPBE (taken from [11]).

The feature weight update rule of the TD learning algorithm presented in Algorithm 1 can be obtained by minimizing the MSE where TV_{θ_t} is used as an approximation for V_π by performing a stochastic gradient descent, which yields the same convergence point as the minimization of the MSPBE [11].

However, when directly minimizing the MSPBE by setting the gradient of the objective function to 0, we obtain the least squares solution

$$\theta = (\underbrace{\Phi^T D(\Phi - \gamma P\Phi)}_{A})^{-1} \underbrace{\Phi^T DR}_{b}, \tag{5.10}$$

with R being a vector containing the expected intermediate reward and $P\Phi$ being a matrix containing the expected features of the successor states. Estimates for A and b can be computed iteratively according to

$$A_{t+1} = A_t + \phi_t[\phi_t - \gamma\phi_{t+1}]^T \text{ and} \tag{5.11}$$

$$b_{t+1} = b_t + \phi_t R_t, \tag{5.12}$$

with convergence for $t \to \infty$ [36]. The use of these estimates for the computation of the feature weights yields the *Least-Squares Temporal Difference (LSTD)* algorithm.

In [6] Boyan presents another way to derive the LSTD algorithm: After having observed a trajectory, the update of the feature weight vector has the form $\theta = \theta + \alpha_n(d + C\theta + \omega)$ where

$$d = E\left[\sum_{i=0}^{L} z_i R_i\right] \text{ and} \tag{5.13}$$

$$C = E\left[\sum_{i=0}^{L} z_i(\phi(X_{i+1}) - \phi(X_i))^T\right]. \tag{5.14}$$

ω is a zero-mean noise with a sufficiently small variance such that θ converges to a fixed point θ_λ, which satisfies $d + C\theta_\lambda = 0$.

The TD(λ) algorithm updates θ only with the most recent trajectory and afterwards forgets the trajectory. On the contrary, LSTD(λ) builds explicit estimates of C and d and, after n independent, observed trajectories, solves for θ by computing $A^{-1}b$, with b being the cumulated d vectors and A being the negative cumulated C matrices. A can be inverted via *Singular Value Decomposition*.

The entire LSTD(λ) algorithm is shown in Algorithm 2. As stated in [6] the LSTD algorithm has several advantages over the TD algorithm. First, as we have seen from the second derivation, LSTD extracts more information from each additional observation [7], since it does not forget earlier observed trajectories. Therefore it needs less training until convergence. Second, LSTD does not require some manually chosen step size parameter α_n. Third, LSTD is not affected by the initial estimate for θ. And fourth,

for $n = 1, 2, \ldots k$ **do**
  $x_t = \text{getStartState}();$
  $z_t = \Phi(x_t);$
  while $x_t \neq END$ **do**
    `//Simulate one step: Receive` R_t `and` x_{t+1}
    $A = A + z_t[\phi(x_t) - \phi(x_{t+1})]^T;$
    $b = b + z_t R_t\ ;$
    $z_{t+1} = \lambda z_t + \phi(x_{t+1});$
    $t = t + 1;$
  end
  `//Whenever updates are desired:` $\theta = A^{-1}b$
end

Algorithm 2: Algorithm for $LSTD(\lambda)$ Learning (after [6]).

LSTD is not sensitive to the ranges of the individual features.

In Figure 5.7 we compare the average performances of the TD(λ) and LSTD(λ) algorithms. As we can see the LSTD(λ) is able to find the optimal solution much faster than the TD(λ) algorithm. Nonetheless, we have to state that TD(λ) might perform better than shown here, since we do not know the optimal values for the step size parameter α_n and the initial feature weights θ. For this evaluation, we have chosen α_n to be $\frac{1}{0.2n+1500}$ and set θ to equal the values that the TD and LSTD agents have initially used for acting in the environment.

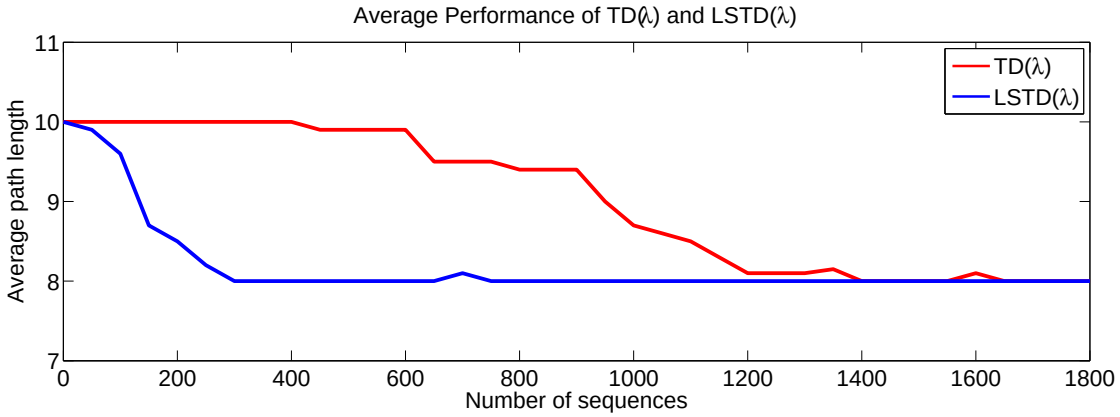


Figure 5.7: Performance Comparison of TD(λ) and LSTD(λ).

5.5 Feature Selection

After having derived the algorithm that we want to use, we now need to select features for the value function approximation. Feature selection is a very crucial step, since the chosen features need to be capable of modeling the problems correctly such that the optimal solutions to them can be found. Besides, the choice of features gives us the opportunity to incorporate prior domain knowledge.

There has been done some research on automatic basis function creation. For example, Keller proposed an algorithm in [29] that uses *neighborhood component analysis* to map the high-dimensional state space based on the Bellman error onto a low-dimensional one, in which states are aggregated to iteratively define additional basis functions. If we followed this approach, we would have to iteratively train a function approximator with LSTD which is very computationally expensive. Furthermore, we have already tried to reduce the dimensionality, as briefly mentioned at the end of chapter 4, but could not find any inherent underlying structure. For these reasons, we will manually define the features.

With the (LS)TD learning approach we will only be able to consider the case of our problem in which we differentiate between blocks. The reason for not being able to deal with the case in which we do not differentiate between blocks is that we cannot find appropriate features for this case. However, we could try to circumvent this problem by assigning every block to a goal position and computing for every permutation the number of expected steps. The problem with this approach is that the number of permutations is #blocks! and we would need to run the algorithm for every permutation, which is computationally too expensive.

5.5.1 Selection of Appropriate Features

It makes sense to choose features that correspond to the natural features of the task [47]. Hence, we will choose for every block both its row distance and its column distance to the goal position as features. In addition it might be important to model the row and column distances between blocks as well to have a feature which is capable of modeling possible together movements of blocks, too. Let's try out these feature values on the problem given in Figure 5.8. On the optimal path both blocks would be pushed down together. The optimal path length is 8. The trivial path length (every block is pushed down separately) is 10.

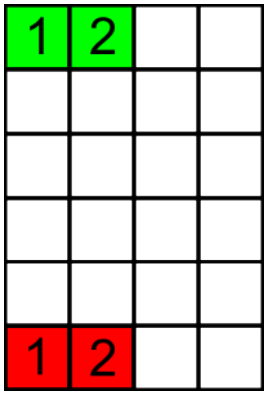


Figure 5.8: Sample Problem for Feature Selection.

We now apply the TD algorithm to the problem and set $R_t = -1$, $k = 1$, $\gamma = 1$, $\lambda = 1$, $\alpha_n = \frac{1}{0.2n+1000}$ and $\epsilon = 0.05$. Moreover, we start with setting all the feature weights for row and column distances to the goal positions to -1 and all other feature weights to 0. These weights would make a greedy agent pursue the trivial path. The averaged feature weights from 10 executions of the experiment with $30,000$

simulated sequences each are shown in Table 5.1.

Feature	$\Delta\text{row}(1)$	$\Delta\text{row}(2)$	$\Delta\text{col}(1)$	$\Delta\text{col}(2)$	$\Delta\text{row}(1,2)$	$\Delta\text{col}(1,2)$
Feature Weight	-1.073	-1.067	-0.951	-0.941	-0.071	-0.862

Table 5.1: Averaged Weights of the Simple Features for the Problem given in Figure 5.8.

The highest negative values are assigned to differences between the two block's current rows and goal rows, followed by the difference between the two block's current columns and goal columns, followed by the difference between the two columns of the blocks. Therefore, a greedy policy simply moves the alternatingly block 1 and block 2 downwards. The obtained path is the trivial one with length 10.

Clearly our selected features are not good enough. While in the beginning we actually want the column distance between both blocks to be 0, which implies that they are vertically aligned, the column distance needs to be 1 in the end again, when both blocks are on their assigned goal positions. Consequently, what we actually want are locally weighted features that model the row / column distances between blocks. However, instead of performing an explicit locally weighted LSTD learning as in [25], we can model the weighting implicitly. We can do so by multiplying the row and column distances between the two blocks with the distances between the blocks' current positions and their assigned goal positions. This will create two features out of one of those features. When we repeat the experiment from above with the new features, we obtain the feature results shown in Table 5.2.

Feature	$\Delta\text{row}(1)$	$\Delta\text{row}(2)$	$\Delta\text{col}(1)$	$\Delta\text{col}(2)$	$\text{dist}(1)\cdot\Delta\text{row}(1,2)$	$\text{dist}(1)\cdot\Delta\text{col}(1,2)$	$\text{dist}(2)\cdot\Delta\text{row}(1,2)$	$\text{dist}(2)\cdot\Delta\text{col}(1,2)$
Feature Weight	-0.572	-0.588	-0.927	-0.922	0.013	-0.238	0.010	-0.217

Table 5.2: Averaged Weights of the Improved Features for the Problem given in Figure 5.8.

These feature weights show that a column difference in the beginning is quite harmful, since the column distance will get multiplied with a value close to -0.2 and the sum of the distances of both blocks to their goal positions. Therefore, a greedy policy will start by vertically aligning both blocks. Then, it will move them downwards together and finally split them up again. The obtained path is the optimal one with length 8.

Figure 5.9 shows the evaluation of the problem given in Figure 5.8 for both feature sets and proves that the latter proposed features are the better ones. In the diagram the blue and the red line correspond to the average path length of 20 independently conducted experiments for the two evaluated feature sets. The semitransparent red stripes around the red line represent the standard deviation for the sample values from the 20 runs. We now want to take a closer look at how our feature weights are updated.

5.5.2 Interpretation of the Weight Coefficients Updates

To obtain a better understanding about the updates of the feature weights, we consider a problem which involves only two blocks. At the beginning of the algorithm, we set the feature weights for the row and

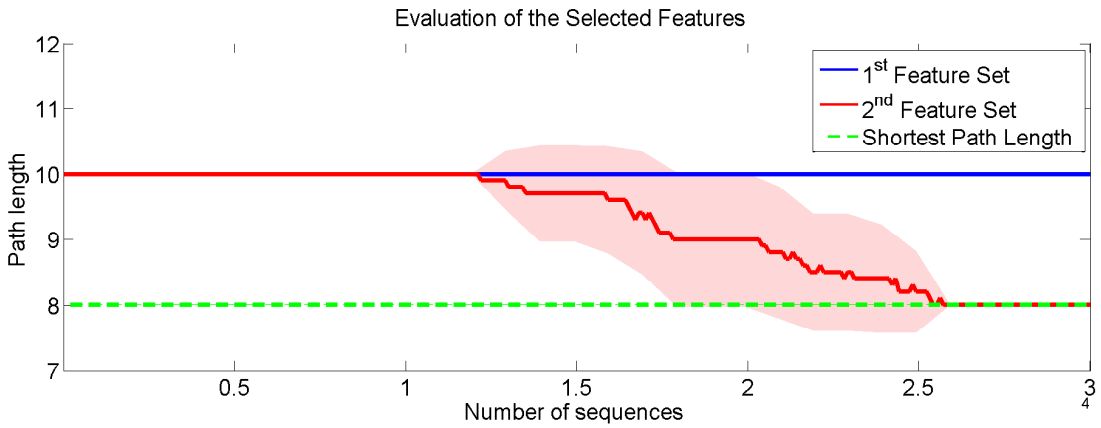


Figure 5.9: Comparison of the Features from Table 5.1 and Table 5.2.

column distances between the block's current and goal positions to -1 and all other feature weights to 0. This would make a greedy agent pursue a trivial path.

Since we have analyzed in section 5.4 that both TD and LSTD learning converge to the same feature weights, we will make use of the TD update rule because of its higher intuitiveness and set $\lambda = 0$ for the same reason. Then, after having completed a sequence, an update of the feature weights is performed according to $\theta = \theta + \alpha \cdot \delta$ where δ contains the accumulated values of all observed state transitions: $\phi(x_t) \cdot [-1 + [\phi(x_{t+1}) - \phi(x_t)]^T \cdot \theta]$. Let us now consider four different cases.

1. Both blocks are moved together towards their goal positions:
   $\phi(x_t) \cdot [-1 + [\phi(x_{t+1}) - \phi(x_t)]^T \cdot \theta] = \phi(x_t) \cdot [-1 + (-1) \cdot (-1) + (-1) \cdot (-1)] = \phi(x_t)$

2. A single block is moved towards its goal position:
   $\phi(x_t) \cdot [-1 + [\phi(x_{t+1}) - \phi(x_t)]^T \cdot \theta] = \phi(x_t) \cdot [-1 + (-1) \cdot (-1)] = 0$

3. No block changes its position due to the border:
   $\phi(x_t) \cdot [-1 + [\phi(x_{t+1}) - \phi(x_t)]^T \cdot \theta] = \phi(x_t) \cdot (-1 + 0) = -\phi(x_t)$

4. A single block is moved away from its goal position:
   $\phi(x_t) \cdot [-1 + [\phi(x_{t+1}) - \phi(x_t)]^T \cdot \theta] = \phi(x_t) \cdot [-1 + 1 \cdot (-1)] = -2 \cdot \phi(x_t)$

It becomes clear that the feature weights will only be updated, if case 1, 3 or 4 takes place. These situations usually occur when exploratory steps are chosen. Most of the time - at least in the beginning - those exploratory steps will lead to either situation 3 or 4. When these situations occur, the weights of the feature vector will become more negative according to the features of the current state. Only when two blocks are aligned and moved together, the weights of the feature vector will become more positive subject to the features of the current state. Consequently, especially in the beginning, exploration will lead to too pessimistic state values. We will come back to the topic of the feature weight updates in the next section when we evaluate our method.

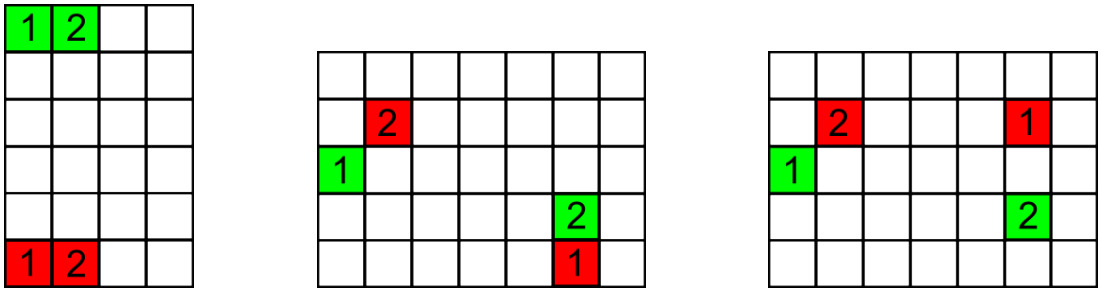
5.6 LSTD Learning Evaluation

We now want to evaluate the LSTD learning approach and the chosen features. For that reason, we will first define a set of six different test problems and then try to solve them. As we will see, we will not be able to solve all of these problems adequately. Therefore, we will propose and evaluate new features in a second step.

5.6.1 Evaluation Problem Set

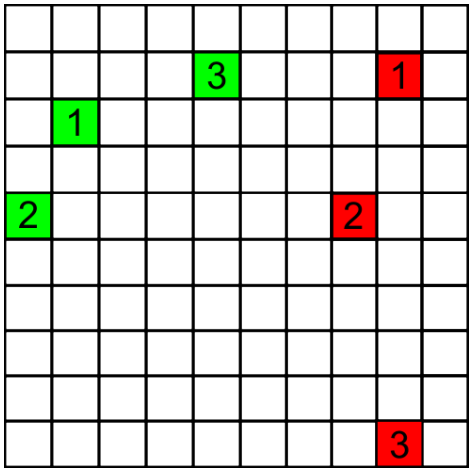
The problem set which we will use to evaluate the performance of our LSTD learning approach consists of six problems which are shown in Figure 5.10. The first problem (Figure 5.10(a)) is the same problem that we have already used in the process of finding appropriate features. In this problem the agent needs to learn that is beneficial to move both blocks together down along the vertical axis. The second problem (Figure 5.10(b)) is a trivial one. Both blocks have to be moved in different directions and should not interact with each other. Therefore the shortest path length equals the trivial path length. In the next problem (Figure 5.10(c)) the trivial path is one step longer than the shortest path. However, for finding the shortest path no *detour move* (a move that an agent following a trivial path would not make) is needed. Consequently, the agent only needs to learn which block movement order is the most beneficial one. In the fourth problem (Figure 5.10(d)) we increase the number of blocks from two to three. Furthermore, for finding the shortest path all three blocks need to interact with each other and there are also detour moves needed. For these reasons this problem is already quite difficult. In the successor problem (Figure 5.10(e)) the number of blocks is increased once more to five. Furthermore not all blocks interact with each other, but only the two and three blocks forming separated groups should be moved together. In addition the size of the game board has increased a lot. The last problem (Figure 5.10(f)) consists again of only two blocks on a big game board. However, for finding the optimal path many detour moves are needed.

In the following evaluation we will always apply the LSTD learning algorithm and set its variables as follows: $R_t = -1$, $k = 2$, $\gamma = 1$, $\lambda = 1$ and $\epsilon = 0.05$. The reason for setting k to 2 is that we want to be able to cope with situations in which for example two blocks need to switch their order. With k being 1 the agent would not be able to solve those situations easily, since it would move one block back and forth all the time.

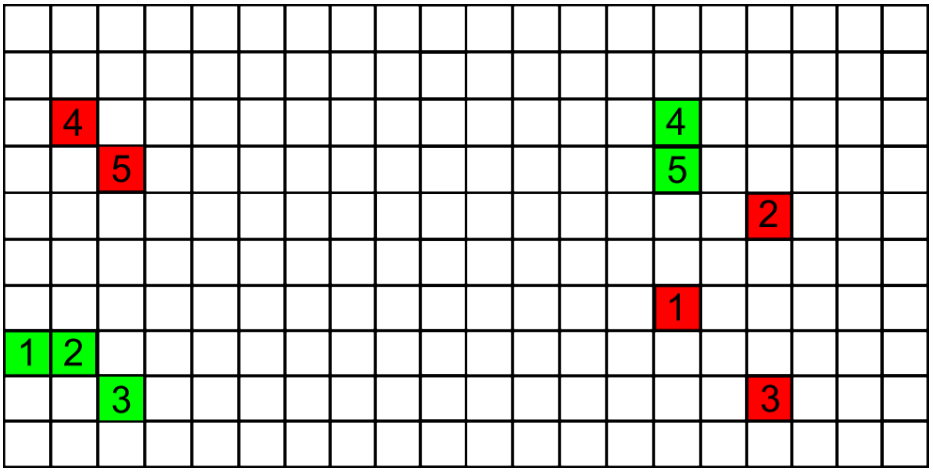


(a) Problem 1 (b) Problem 2 (c) Problem 3

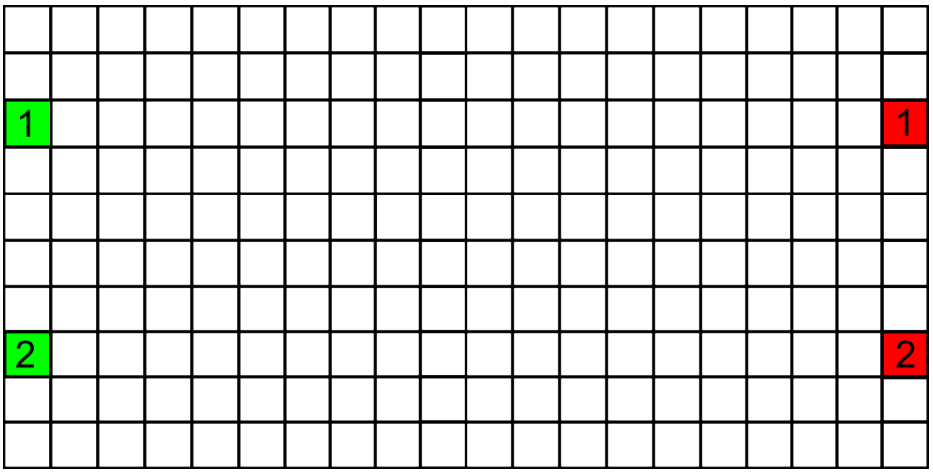
Figure 5.10: Evaluation Problem Set: Six problems.



(d) Problem 4



(e) Problem 5



(f) Problem 6

Figure 5.10: Evaluation Problem Set: Six Problems (cont.).

5.6.2 Evaluation Results

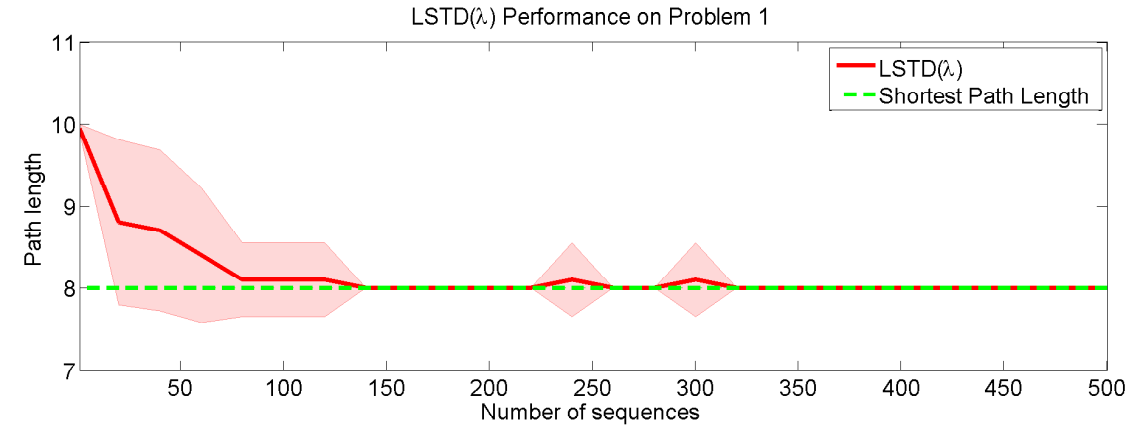
The features which we will use in the LSTD learning evaluation process are the following ones:

- $\left|\Delta\text{row}(x_{\text{current}}, x_{\text{goal}})\right|$
- $\left|\Delta\text{col}(x_{\text{current}}, x_{\text{goal}})\right|$
- $\left|\Delta\text{row}(x_{\text{current}}, y_{\text{current}})\right| \cdot \left[\left|\Delta\text{row}(x_{\text{current}}, x_{\text{goal}})\right| + \left|\Delta\text{col}(x_{\text{current}}, x_{\text{goal}})\right|\right]$
- $\left|\Delta\text{row}(x_{\text{current}}, y_{\text{current}})\right| \cdot \left[\left|\Delta\text{row}(y_{\text{current}}, y_{\text{goal}})\right| + \left|\Delta\text{col}(y_{\text{current}}, y_{\text{goal}})\right|\right]$

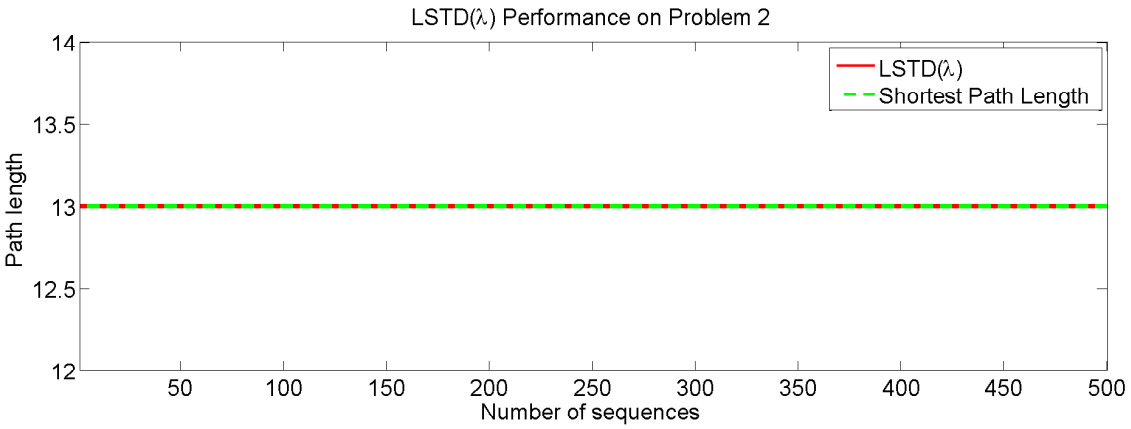
x and y are variables that each refer to a block from the set of all blocks and contain both the current and the assigned goal position of the block. The feature weights which we start with are again -1 for the column / row distances for single blocks and 0 for the features referring to distances between two blocks.

The evaluation results can be seen in Figure 5.11. With these features we are able to solve the problems 1, 2 and 3 quickly. However, we fail to solve problems 4, 5 and 6. For problem 6 we just end up with a trivial path. For the problems 4 and 5 we do not end up with the trivial path, but with feature weights that make a 2-step greedy lookahead agent not find the goal state at all. Why this divergence happens becomes clearer when considering the updates of the features discussed in section 5.5.2. When an agent benefits from moving several blocks together, the feature weights will increase in accordance with the features of the traversed states. When for this together movement of blocks one block for example has some column distance to its destined column, the feature weight referring to its column distance will increase and might become positive. Consequence of a positive weight for the column distance is that the agent tries to maximize the distance between the block's current and destined column and therefore never reaches its goal position. We demonstrate that this case might happen with an example problem given in Figure 5.12. We have trained the agent exclusively with an optimal path sequence of states, in which block 2 moves together with block 1 in the first row. The result we have obtained yields a weight vector with a positive feature weight for block 2's row distance to its destined row.

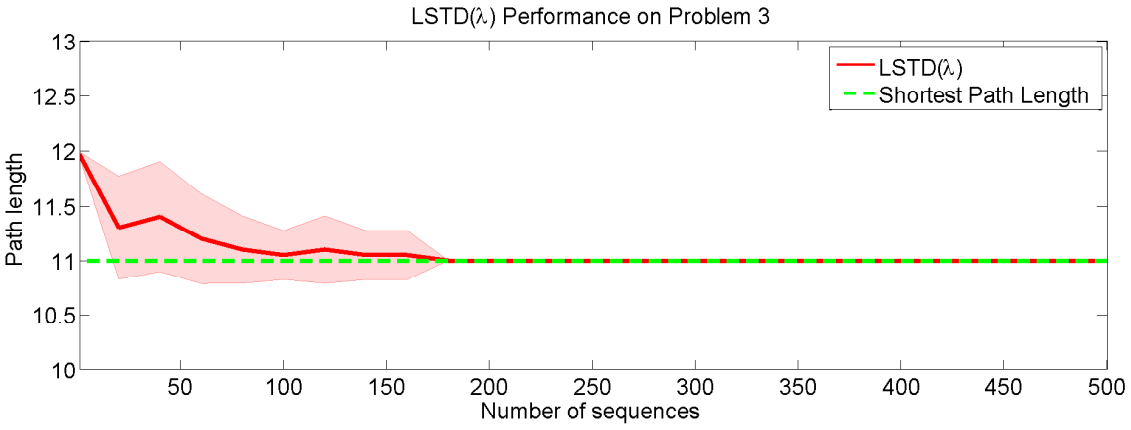
From the problem presented in the last paragraph we conclude that we do not only need some weighting for the distances between blocks, but also for the distances between a block's current and its assigned goal position. However, we can not simply predefine a weighting, because we do not have any information about the length of a possible together movement with other blocks. We have tried several simple weighting functions, e.g. linear weighting, but they only performed well for some problems. From this result we deduce that we need to incorporate some more domain knowledge in the features. In the next section, we will propose several options to improve the general performance of our LSTD learning implementation (beyond the selection of appropriate features). Furthermore, we will discuss which information useful features should capture in order to choose new features and evaluate them in the section thereafter.



(a) Evaluation Result for Problem 1



(b) Evaluation Result for Problem 2



(c) Evaluation Result for Problem 3

Figure 5.11: Evaluation Results for the Problem Set.

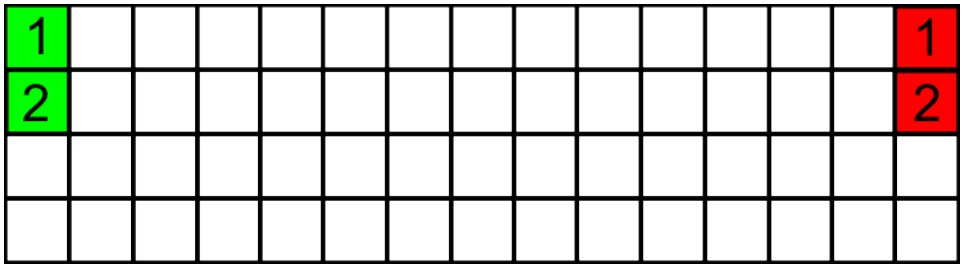


Figure 5.12: Sample Problem Not Solvable with the Selected Features.

5.7 Performance Improvement Techniques

In this section, we will briefly discuss several possibilities to improve the agent's performance. In more detail, we will talk about replacing traces, the discarding of bad runs and a more efficient implementation of the LSTD learning algorithm.

5.7.1 Replacing Traces

Eligibility traces can assign high Z_t values to states that are visited more than one time during a single run, which puts a high weight on those states. This might contradict the idea behind this concept. In [43] Singh has introduced a new concept that he calls *replacing traces*, which resets the Z_t value of a state to 1 when it is revisited:

$$Z_t(S) = \begin{cases} \gamma\lambda Z_{t-1}(S) & \text{if } S \neq S_t; \\ 1 & \text{if } S = S_t. \end{cases} \tag{5.15}$$

The difference between replacing traces and eligibility traces is shown graphically in Figure 5.13.

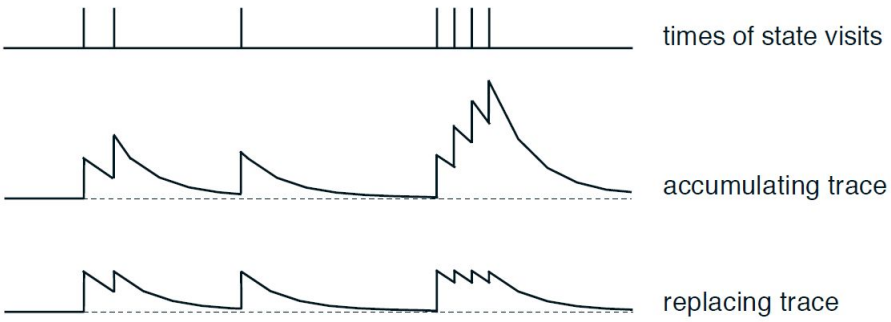


Figure 5.13: Comparison of Eligibility Traces and Replacing Traces (taken from [47]).

Singh also shows that the use of replacing traces eliminates the bias which is created through the use of eligibility traces and that the mean squared error is always lower in the long run. Therefore, he concludes that replacing traces lead to a significant performance improvement, which is affirmed by Sutton in [47].

In an experiment we have compared eligibility and replacing traces against each other. The results are shown in Figure 5.14. It seems as if replacing traces would indeed outperform eligibility traces. Nonetheless, the average performance difference and the number of runs (20) over which we have averaged are too small to allow us to call this difference in performance significant. This result gets emphasized by the fact that the agent - except in the case of exploratory moves - should never visit the same state twice during one sequence. Ergo, the difference between eligibility and replacing traces is negligible.

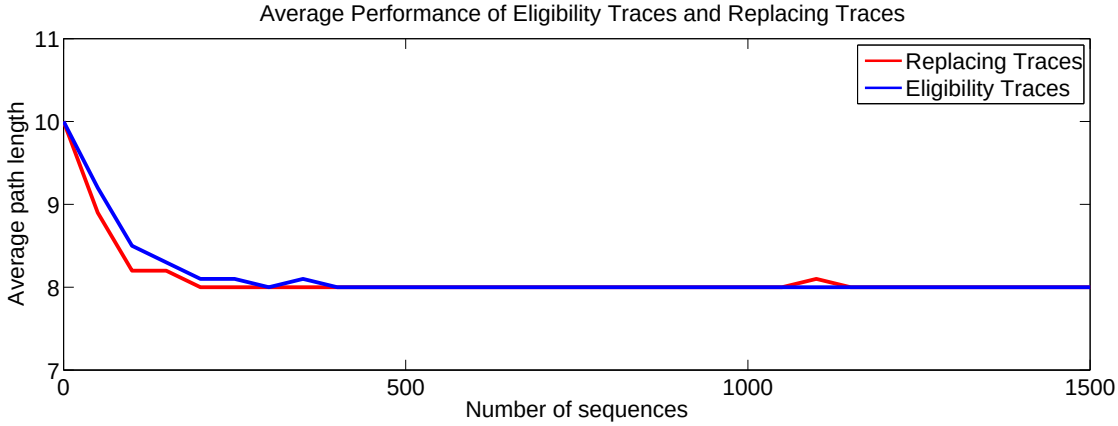


Figure 5.14: LSTD(λ) Performance Comparison of Eligibility Traces and Replacing Traces.

5.7.2 Discarding Bad Runs

A problem which occurs from time to time is that the agent visits some positions far away from the currently optimal path due to the ϵ-greedy policy and sometimes becomes "stuck" there, as the current value function approximation assigns quite wrong values to these unseen states. The consequence is that the time the agent needs to visit the goal state might become impractically long and the accumulated values for A and b might get "distorted" by the occurrences of many for the problem unusual state transitions, which leads to even worse feature weights θ. Furthermore, such "exploration errors" are especially fatal in the beginning, because - in contrast to TD learning - LSTD learning does not rely on an initial, reliable guess of θ. Consequently, when the first update of θ is computed with only a small number of so far simulated sequences, a bad run has a high influence and can lead to totally wrong feature weights. These might make the agent not find its goal state any more after the first feature weight update has been performed.

To prevent this harm, we will discard such *bad* runs. To put it more concretely, we will cancel the current run, if its length exceeds two times the trivial path length and we will not use but discard the made updates on matrix A and vector b.

5.7.3 More Efficient Implementation of LSTD Learning

For every computation of the feature weights for a given A and b, we need to invert a $k \times k$ matrix, where k is the number of features, with computational costs in $O(k^3)$. These costs can be reduced to $O(k^2)$

by directly computing approximated values of A^{-1}. A direct update formula of A^{-1} has been derived by Nedic and Bertsekas in [36]. The implementation of this learning improvement strategy only reduces the computational time of experiments, but does not change the results themselves.

5.7.4 The Real Problem

So far we have discussed several interesting aspects, which all increase the performance of our algorithm. However, the actual problem remains. We are still not able to find correct solutions to some of the presented evaluation problems.

For problem 5 (Figure 5.10(e)) the issue is the large number of blocks, which leads to many features that need to be learned, since the number of features is in $O(n^2)$, where n is the number of blocks. Nonetheless, not all five blocks, but only two and three do interact with each other.

Let's also consider once more the evaluation problem 6 (Figure 5.10(f)). The two blocks, which are quite far away from each other, would benefit from an horizontal alignment, because the detour costs are $2 \cdot 5 = 10$, but the benefit is 18. However, we need lots of exploratory steps in a row that contradict the current policy, which would push each block separately along the horizontal axis, to find the optimal path. Theoretically, the algorithm may converge to this global optimum after having simulated many sequences, but practically it does not. Let's imagine the same problem on an even larger scale. Finding the optimal solution is now even much more unlikely. Nonetheless, for human beings both problems are equally difficult to solve, because we would just need to count the number of extra steps for aligning and disaligning the two blocks and compare it to the number of beneficial together movements of the blocks.

We want to address these problems by redefining the features we use. We will use again for every block its row and column distance to its goal position as features. Furthermore, we will use for every pair of blocks its row and column *bonus* and *cost*. With the term *bonus* we refer to the number of overlapping rows / columns that these two blocks need to traverse to reach their goal positions. With the term *cost* we refer to the number of extra steps that have to be taken to move both blocks to the same row / column. In the case that there is no overlap between two blocks, these features will just be zero. Consequently, we can reduce the number of features with this approach, too. Furthermore, with these features, we hope to address the second issue, that is, finding the optimal path despite many detour moves, as well. The evaluation of the new features will be done in the next section.

Another idea to which these two issues lead us is the one of defining *macros* - sequences of actions that can be invoked by their names and therefore treated similarly as primitive actions. This concept should decrease the difference in difficulty between evaluation problem 6 and a larger version thereof and make both of them more easily solvable. For that reason, we will consider some hierarchical / temporal abstraction approach in chapter 6.

5.8 Evaluation Results for Bonus vs. Cost Features

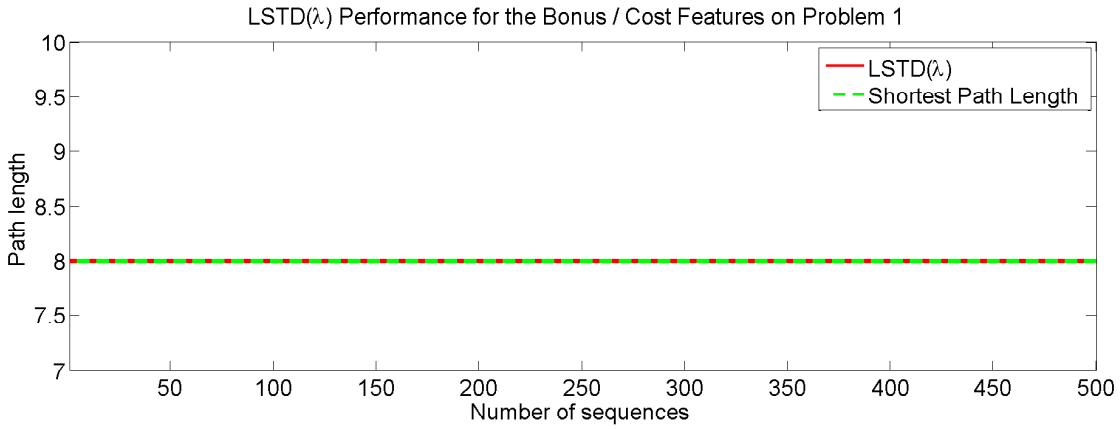
The features that we will use in this LSTD learning evaluation process are the following ones:

- $\left|\Delta\text{row}(x_{\text{current}}, x_{\text{goal}})\right|$
- $\left|\Delta\text{col}(x_{\text{current}}, x_{\text{goal}})\right|$
- $\text{bonus}(\text{row}, x_{\text{current}}, x_{\text{goal}}, y_{\text{current}}, y_{\text{goal}})$
- $\text{bonus}(\text{col}, x_{\text{current}}, x_{\text{goal}}, y_{\text{current}}, y_{\text{goal}})$
- $\text{cost}(\text{row}, x_{\text{current}}, x_{\text{goal}}, y_{\text{current}}, y_{\text{goal}})$
- $\text{cost}(\text{col}, x_{\text{current}}, x_{\text{goal}}, y_{\text{current}}, y_{\text{goal}})$

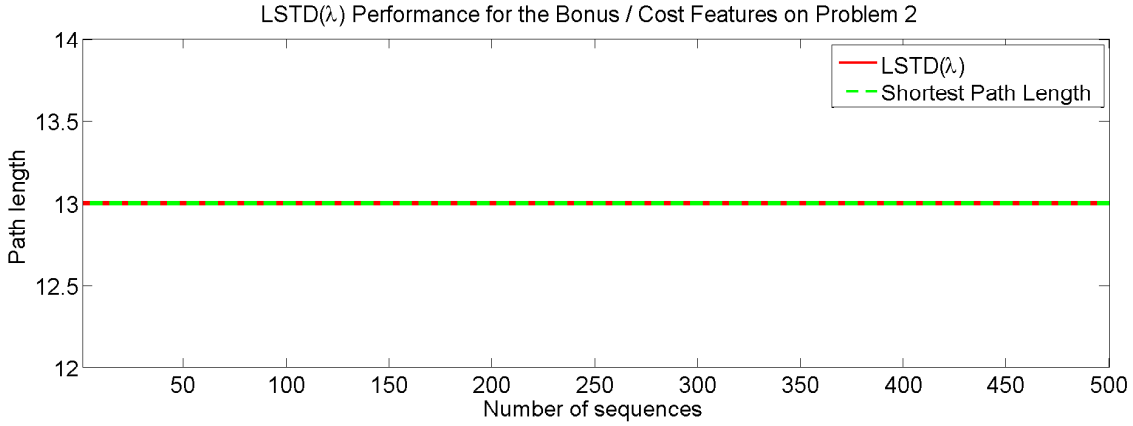
x and y are variables that each refer to a block from the set of all blocks and contain both the current and the assigned goal position of the block. *bonus* is a function that computes and returns the number of overlapping rows / columns for two blocks on the way to their destination positions. *cost* is a function that computes and returns the number of extra steps needed for aligning and disaligning two blocks on the way to their destination positions. The feature weights which we start with are again -1 for the column / row distances for single blocks. To the bonus and cost features we assign values of 0.7 and -0.7 respectively. The reasons why we did not choose values of ± 1 or 0 is that on the one hand, we want to put more emphasis on the trivial than on an alleged optimal path in the beginning to make sure that the agent finds the goal position. On the other hand, we also want to provide the agent with some useful information in the context of discovering a shorter path. In an empirical experiment on the set of evaluation problems values of ± 0.7 turned out to be a good compromise.

The evaluation results can be seen in Figure 5.15. With the selected features we are able to solve all given problems. Nonetheless, it is remarkable that with the chosen start values for the features, the agent sometimes finds the shortest path right away from the beginning. However, we can see from the evaluation of problem 4 (Figure 5.15(d)) that this is not always the case. Yet, the agent is still able to find the shortest path, although it seems to take some time. When taking a closer look at problem 4 again, we notice that it is even for us not that easy and obvious to determine that the shortest path has length 20, since there exist many ways in which the agent could benefit from block interactions. The presented results imply that the finally selected features perform quite well and succeed in modeling the problem adequately.

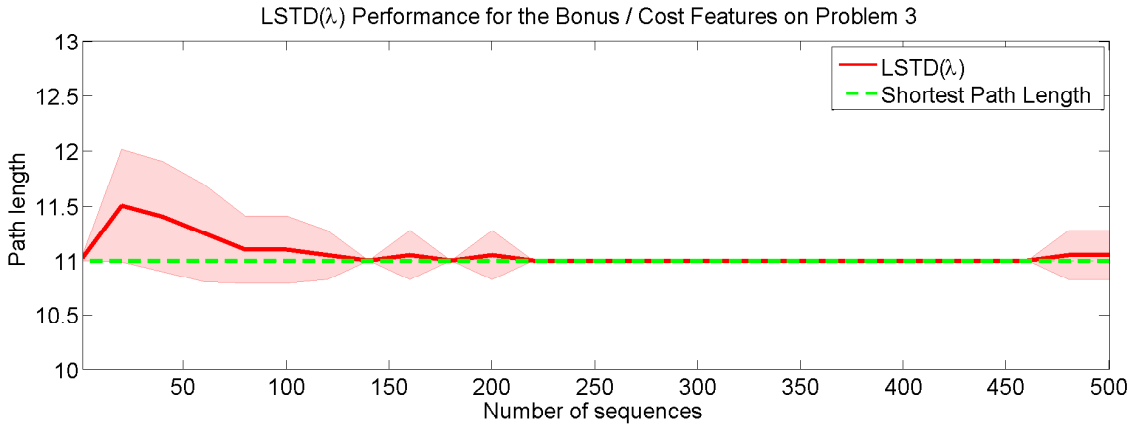
In the next chapter we want to introduce for the last time a new approach to the problem. As already mentioned we will make use of hierarchical / temporal abstraction to be able to focus on the different parts of the problems separately and sequentially. Furthermore, we hope to circumvent the problem of not target-oriented ϵ-greedy exploration with this approach.



(a) Evaluation Result for Problem 1

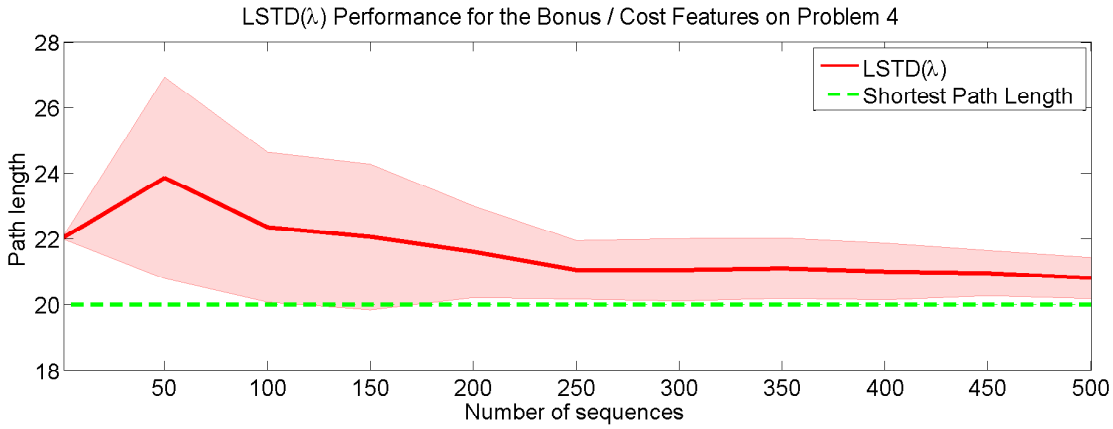


(b) Evaluation Result for Problem 2

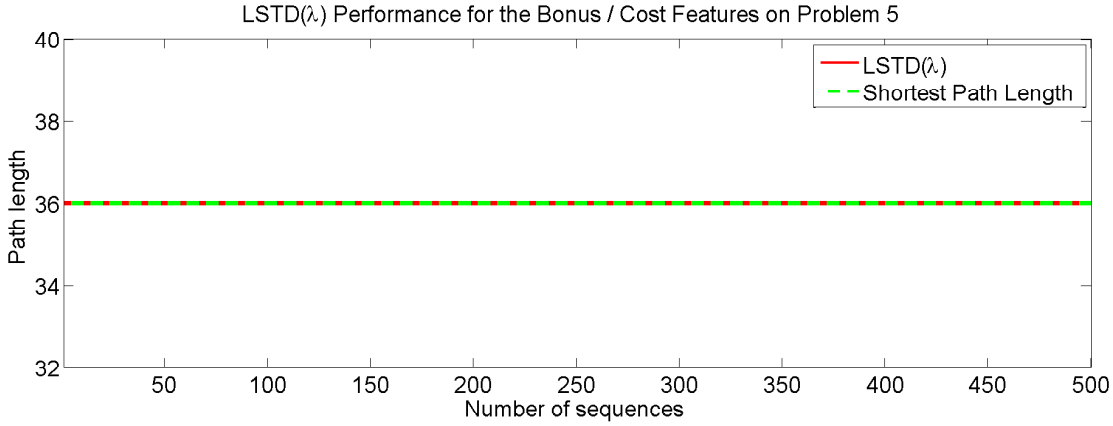


(c) Evaluation Result for Problem 3

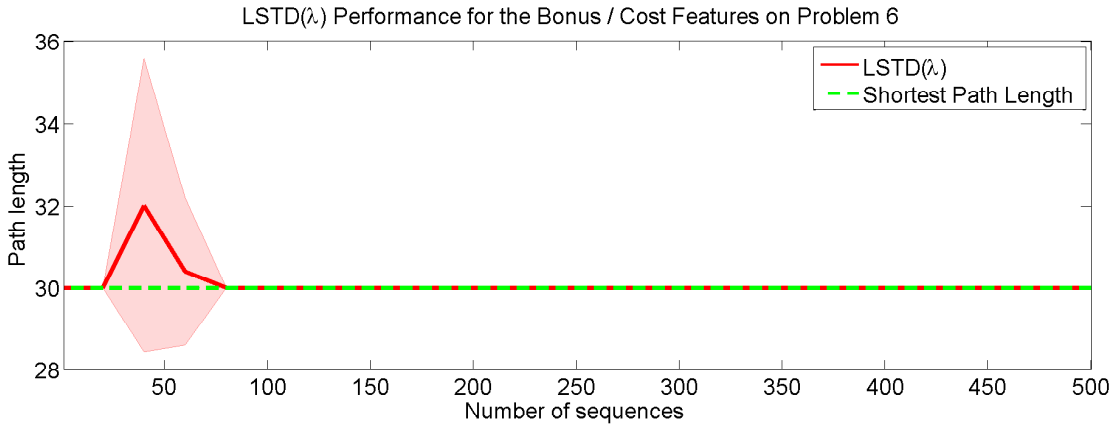
Figure 5.15: Evaluation Results for the Bonus / Cost Features for the Problem Set.



(d) Evaluation Result for Problem 4



(e) Evaluation Result for Problem 5



(f) Evaluation Result for Problem 6

Figure 5.15: Evaluation Results for the Bonus / Cost Features for the Problem Set (cont.).

6 Hierarchical Reinforcement Learning

The idea behind *hierarchical reinforcement learning (HRL)* is similar to the one behind *divide and conquer*. We aim at decomposing a problem into several sub problems, solving them independently from each other and combining their solutions to obtain a solution to the original problem. We expect that the use of hierarchies can help us to overcome the curse of dimensionality and accelerate learning [42].

Recently, HRL has already been successfully applied to some real-world problems dealing with language. For instance, Cuayahuitl et. al have used HRL to learn a well performing spoken dialogue system [10], and Dethlefs and Cuayahuitl have applied HRL to learn a natural language generation policy [17]. However, in both cases the environments, in which the agents learned, consisted of interlinked modules, which lead to a straightforward construction of the hierarchies.

6.1 HRL Basics

The introduction of hierarchies forces a structure on the policies that shall be learned. Imposing constraints on the policy can lead to the effect that the found optimal policy satisfying these constraints is not optimal for the original problem. Such a policy is therefore called *hierarchically optimal* [42]. For that reason, the design of the hierarchical structure is very crucial. There exists an approach that automatically decomposes MDPs into several interlinked MDPs by searching for Markovian subspace regions [24]. Yet, this algorithm is only applicable to problems, whose state variables implicitly define useful subspace regions, which is not the case for our problem.

There are two fundamental ways in which decomposition can be used for HRL. The first one limits the available actions, including hard coded parts of policies and parts of policies which are learned with a limited subset of actions. The second one specifies local goals for specific parts by posing limits on the availability of actions at different times so that the agent does not waste time with exploration of the other, not target-oriented actions [42].

In the literature, there exist three main approaches to HRL [3]: Sutton's *options* [48], the *hierarchies of abstract machines (HAMs)* of Parr and Russell [38] and Dietterich's *MAXQ* value function decomposition [18]. In the following sections, we will briefly discuss the first two of them and apply them to our problem. We will leave out the MAXQ framework, since for several reasons it does not fit very well to our problem. The mentioned approaches have all been developed in 1999 and 2000. Since then, research on HRL has rather focused on combining these approaches with other RL methods. For example Jong and Stone have recently developed an algorithm that combines the MAXQ value function decomposition with the model-based learning algorithm R-MAX. [27].

A fundamental concept in HRL is the one of *temporal abstraction*. Temporal abstraction provides us with the possibility to define actions that operate over multiple time steps such that decisions are not

required at every single step. When we refer to one-step actions, we will use the term *primitive actions*. For actions that last several time steps, we will refer to as *behaviors* or, respectively, use the terms from the original papers (*options*, *machines*).

To integrate the concept of behaviors into MDPs, we need to generalize MDPs to *semi-Markov decision processes (SMDPs)* that deal with continuous time [19]. The amount of time between two decisions is modeled as a random variable τ. In our case this random variable is an integer, since behaviors consist of a multiple of primitive actions. The transition probabilities result in a joint probability distribution: $P(s', \tau|s, a)$ refers to the probability that the agent ends up in state s' after τ time steps when having chosen action a in state s. The rewards $R(s, a, s')$ represent the accumulated discounted rewards received on the path from s to s' [3].

6.2 Options in Theory

The concept of *options* was first introduced by Sutton [48]. An option $\langle \boldsymbol{I}, \pi, \beta \rangle$ is a generalization of primitive actions to temporally extended actions. It consists of a policy $\pi : S \times A \mapsto [0, 1]$, a termination condition $\beta : S \mapsto [0, 1]$ and an input set $\boldsymbol{I} \subseteq S$, referring to the set of states in which the option is available [49]. When an option is executed, then the actions are selected according to the policy π, which assigns to the execution of every action a in state s a probability $\pi(s, a)$. The option terminates stochastically in state s according to the probability $\beta(s)$. The kind of option that we have just defined is also called *Markov option*, since its policy is Markov, meaning that the action probabilities are solely based on the current state of the MDP. To increase the flexibility, one can include *semi-Markov options* as well that allow the setting of action probabilities based on the complete history of states, actions and rewards [3, 48]. The set of available options $O(s)$ for every state s is very similar to the set of available actions. Both sets can be unified by considering primitive actions as special kinds of options that last only one step and are always available, when the primitive actions are available [49].

A policy μ selects in state s option o with probability $\mu(s, o)$. The executed option determines all actions until its termination. In this way a policy defined over options determines a policy over actions, which is called a *flat policy*, $\pi = \text{flat}(\mu)$. Due to the fact that the probability of selecting a primitive action depends not only on the current state, but also on all policies of the options that are in the hierarchical specification currently involved, a flat policy is in general not Markov, even if all options are Markov [3].

The value of a state under a general policy μ is defined as the value of the state under the corresponding flat policy π: $V_\mu(s) = V_{\text{flat}(\mu)}(s)$. Similarly, we can generalize from the action-value function $Q(s, a)$ to the option-value function $Q(s, o)$, which corresponds to the total discounted, expected return when option o is executed in state s. The reward of executing option o, which terminates after k steps, in state s at time t is defined as $R(s, o) = \sum_{i=1}^{k} \gamma^{i-1} R_{t+i}$. Lastly, the state-prediction part of the model of option o is $P(s'|s, o) = \sum_{\tau=1}^{\infty} p(s', \tau)\gamma^\tau$, where $p(s', \tau)$ is the probability that o terminates in s' after τ steps

[3]. When we apply these definitions to Watkins' Q-learning update formula from [52], we receive the *SMDP Q-learning* update formula [49]:

$$Q_{k+1}(s,o) = (1-\alpha_k)Q_k(s,o) + \alpha_k \left[R(s,o) + \gamma^{\tau} \max_{o' \in O(s')} Q_k(s',o') \right], \tag{6.1}$$

where τ is the number of steps that the agent executes between s and s'. These updates are computed after every option termination.

Under the assumption that every option is executed infinitely often in every state, which is visited infinitely often, and that α_k decays according to the formulas given in equation 5.3, then $Q(s,o)$ converges to the optimal option-value function $Q^*(s,o)$. Given Q^*, optimal policies can be determined as greedy policies. Furthermore, if every primitive action is available as a one-step option, then the optimal policy over options will be the same as the optimal policy of the core MDP [3].

6.3 Application of Options

After having described what options are and how they can be used for learning an option-value function, we now want to apply the concept of options to our given problem. In a first step, we will define which options will be available in which states. Then, we will provide a new version of the SMDP Q-learning formula, which fits better to our specific problem. Afterwards, we will evaluate our approach, suggest to limit the number of available options and finally, evaluate the limited options as well.

6.3.1 Option Selection

The selection of available options is almost as important as the selection of features for the LSTD learning algorithm (see section 5.5). To make sure that the policy, which we will find with our options approach, is always able to correspond to the optimal policy, all primitive actions will be made available as options. In addition, we choose the movement of every block in one of the four directions by a specific number of steps, which is limited to k, where k is the number of steps needed to reach the wall, to be available options, too. More formally, we define that the set of available options in state s contains all triples (i,d,j), where i is the number of a block ($i \in [1,N]$), d refers to the moving direction ($d \in [1,4]$) and j corresponds to the number of steps ($j \in [1,k]$, where k is the number of movements of block i in direction d to reach the wall). Figure 6.1 shows an example with two blocks and all destinations they can reach through the execution of one of the available options.

6.3.2 Redefining the Q-Value Update

In section 6.2 we have presented the update formula of the SMDP Q-learning algorithm (equation 6.1). As we have already stated, we are able to find the optimal policy with this update rule. However, we would like to modify it. Since our actual goal is to find the shortest path, it makes sense to not just slightly change the value of $Q(s,o)$ in the direction of $R(s,o) + \gamma^{\tau} \max_{o' \in O(s')} Q_k(s',o')$, but to change it

Figure 6.1: Available Options. The opaque green fields correspond to the current blocks. The transparent green fields correspond to the fields that can be reached with the available options.

completely to $R(s,o) + \gamma^{\tau} \max_{o' \in O(s')} Q_k(s', o')$, if this value is greater than $Q(s,o)$, or otherwise do not change $Q(s,o)$ at all. This modification of the update formula is advisable, when we can guarantee that the Q values never underestimate the costs of moving all blocks to their destination positions. This is the opposite of an *admissible* heuristic (see section 2.1.1). Therefore, the modified update rule goes as follows:

$$Q_{k+1}(s,o) = \max\left[Q_k(s,o), R(s,o) + \gamma^{\tau} \max_{o' \in O(s')} Q_k(s', o')\right] \tag{6.2}$$

The advantage of the use of this modified update rule is a faster convergence to the optimal Q values. Depending on the values of α_k, it can take quite a long time, until the Q values are close to their optimal values. Consider a case in which through the choice of a new option in a specific state, a higher value is received than through any other option. However, the Q value of some other option is still greater than the updated Q value of the newly discovered option. In this case a greedy agent would still choose the worse option and would therefore need to explore the better option some more times to find out that this option is actually the better one.

The entire algorithm for HRL with options, which we will use, is shown in Algorithm 3.

for $n = 1, 2, \ldots k$ **do**
  $x_t = \text{getStartState}()$;
  while $x_t \neq END$ **do**
    $\text{avOptions} = \text{getAvOptions}(x_t)$;
    `// SelectOption(·) eventually creates new entries in the LUTs`
    $\text{opt} = \text{SelectOption}(x_t, \text{avOptions}, \epsilon)$;
    $x_{t+1} = \text{getNextState}(x_t, \text{opt})$;
    $Q(x_t, \text{opt}) = \max\left[Q(x_t, \text{opt}), -\text{getLength}(\text{opt}) + \max_{o' \in \text{getAvOptions}(x_{t+1})} Q_k(x_{t+1}, o')\right]$;
    $t = t + 1$;
  end
end

Algorithm 3: Algorithm for *HRL with Options*.

6.3.3 Option Evaluation

We have implemented the SMDP Q-learning algorithm with the modified update formula. The agent, which chooses which options will be executed, is an ϵ-greedy one. We set $\epsilon = 0.1$ to incorporate more exploration than in the LSTD learning case, since exploration cannot harm by making the agent not find the goal state any more. Every visited state is stored in a LUT and for every state all options with their corresponding Q values are listed in a LUT, too. We set $\gamma = 1$ and the reward of an option to its negative length: $R = -\tau$. The initial Q values correspond to the negative trivial path lengths between the states and the goal position so that we can guarantee to not underestimate the costs.

As evaluation problems we have chosen the same ones as for our LSTD learning algorithm in section 5.6.1 (see Figure 5.10). In addition, we have again executed every experiment 20 times and computed both the mean and the standard deviation of the found path lengths. In Figure 6.2 we compare the performance of the described options approach with a basic Q-learning approach, which we obtain by limiting the options to primitive actions, on problem 3 (Figure 5.10(c)). As we can see, the options approach clearly outperforms the basic Q-learning approach, although this problem is still very small.

The results of all evaluation problems can be seen in Figure 6.3. The problems 1, 2 and 3 are solved quite well. However, for the problems 4, 5 and 6 the agent would still need to simulate more sequences to find the shortest path. Especially in problem 6 1000 sequences seem to be not enough. The main reason for the need of many sequences for learning is the huge number of available options. This makes it difficult for the agent to find the shortest path for problem 6, which consists of several detour moves in the beginning. Therefore, we will now try to limit the number of options, from which the agent can choose.

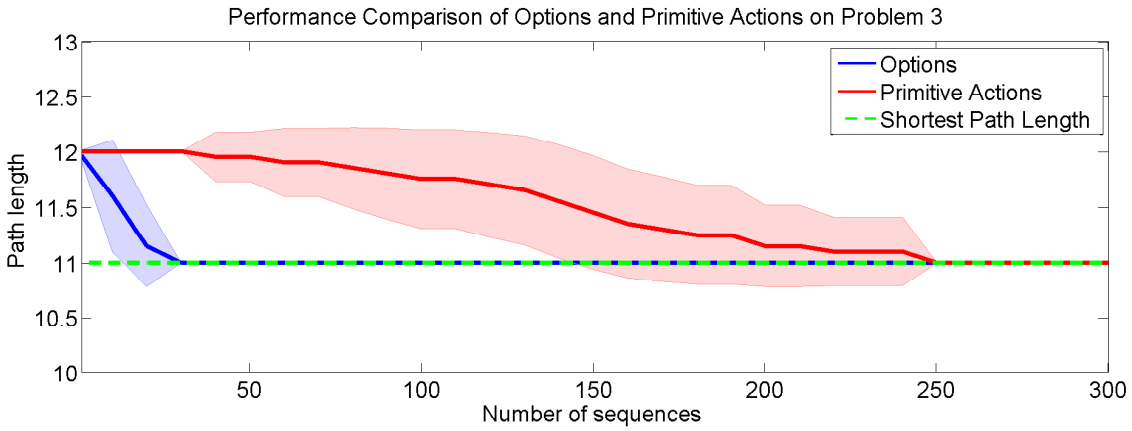
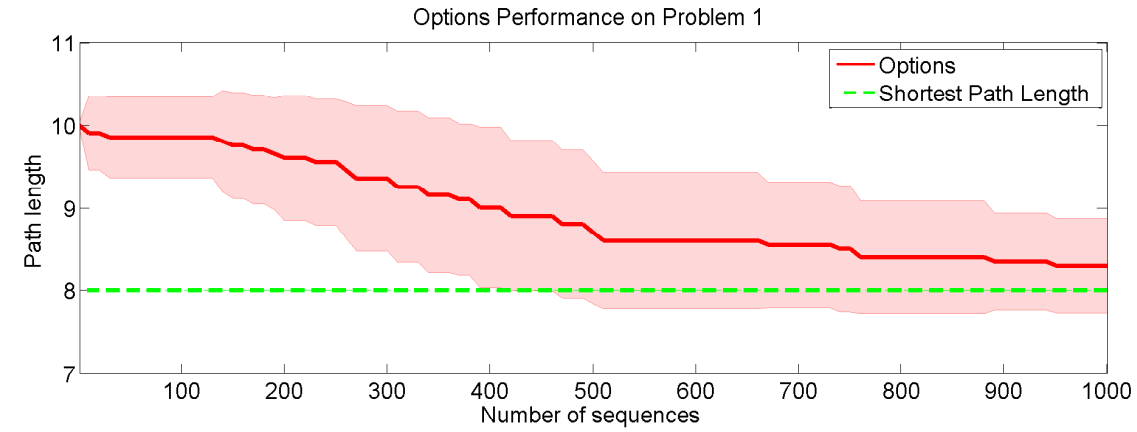
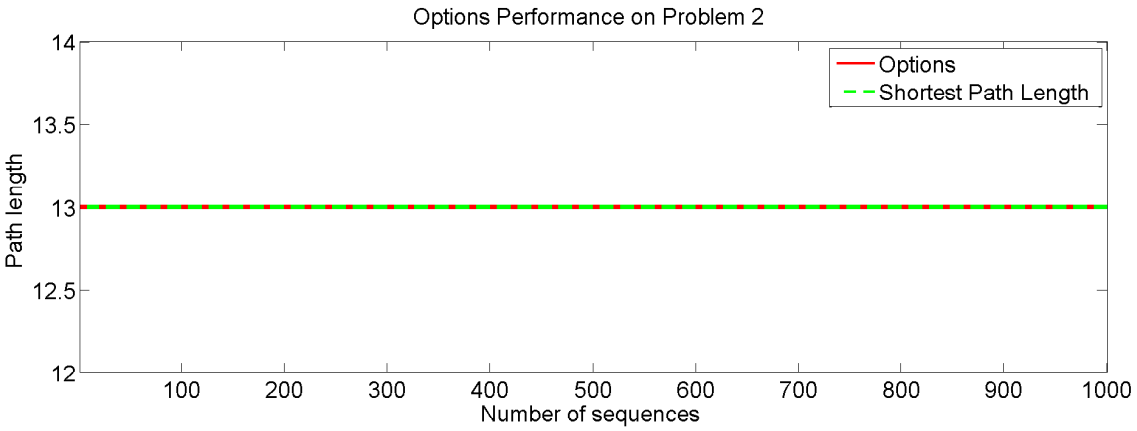


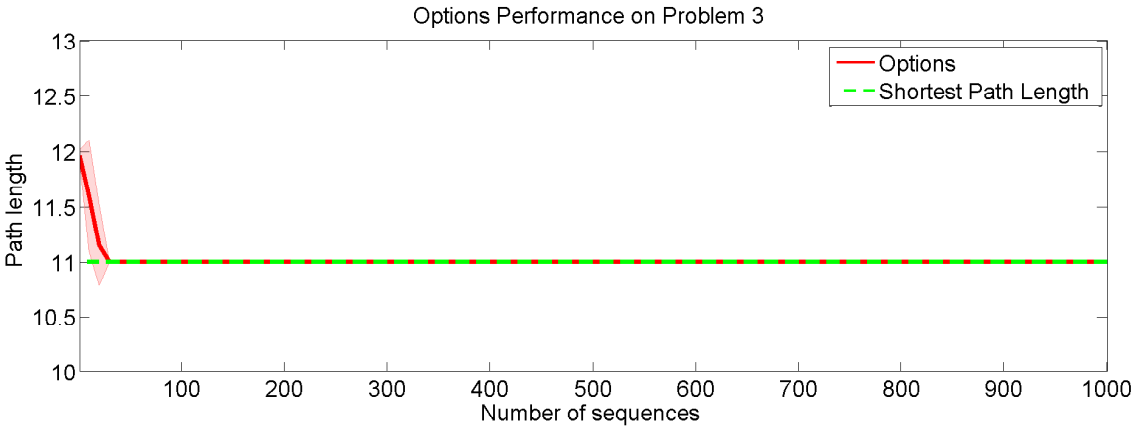
Figure 6.2: Performance Comparison of Options and Primitive Actions.



(a) Evaluation Result for Problem 1

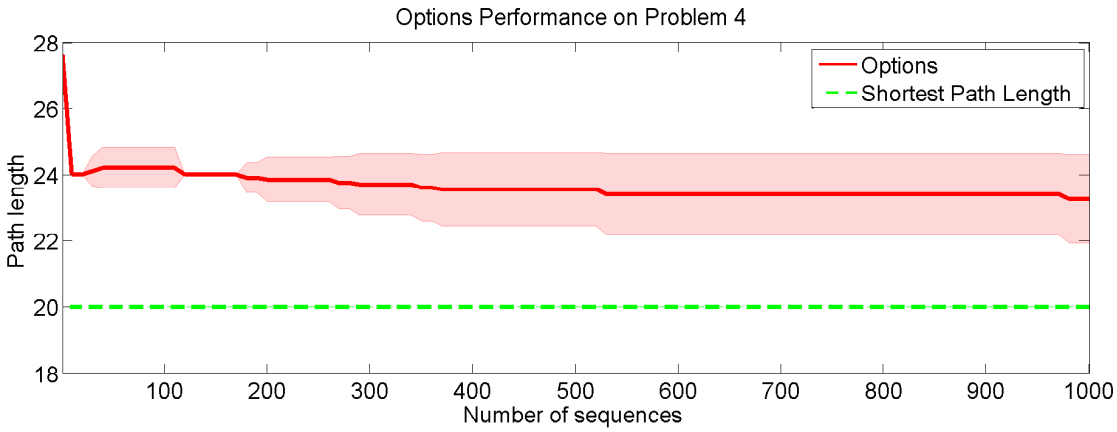


(b) Evaluation Result for Problem 2

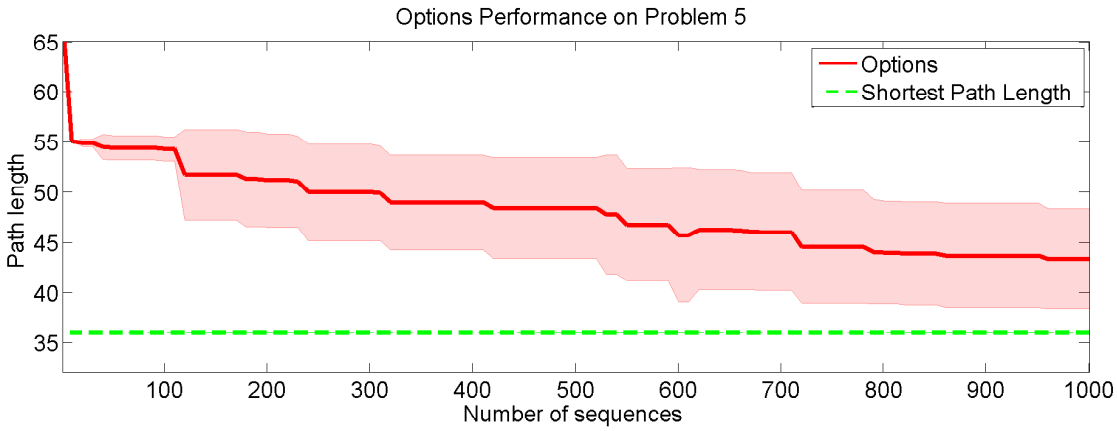


(c) Evaluation Result for Problem 3

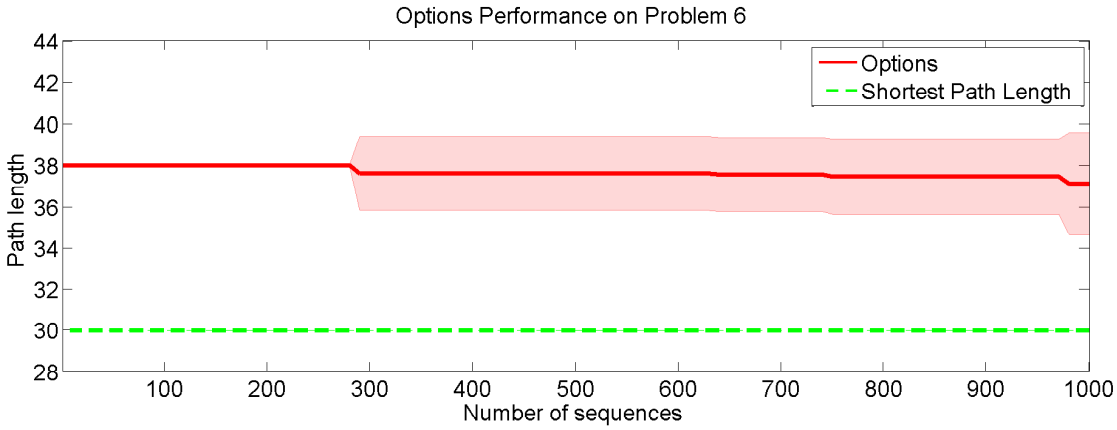
Figure 6.3: Evaluation Results for the Options Approach.



(d) Evaluation Result for Problem 4



(e) Evaluation Result for Problem 5



(f) Evaluation Result for Problem 6

Figure 6.3: Evaluation Results for the Options Approach (cont.).

6.3.4 Limiting Available Options

In each step the agent has for every block many possible options, from which it can choose. The number of options grows linearly with the number of blocks. A large number of possible options, can make the agent need much exploration to find the shortest path. Furthermore, we need much memory, since the LUTs for the Q values will become pretty large. Both is not desirable. For that reason, we would like to limit the number of available options without putting so much restrictions on the policy that we are not able to find the optimal one any more.

In our limited options approach all primitive actions will still be available as options. Furthermore, options that move a block in its destination row or column or that move a block in the row or column of another block (for alignment) will be available. In addition, when several blocks are moved at once, all step lengths are available that move one of the blocks to its assigned row or column. In Figure 6.4 we have added the goal positions of the two blocks to the example from Figure 6.1 and show all destinations the blocks can reach through the execution of one of the limited available options.

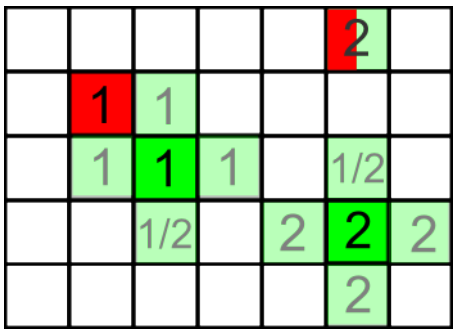
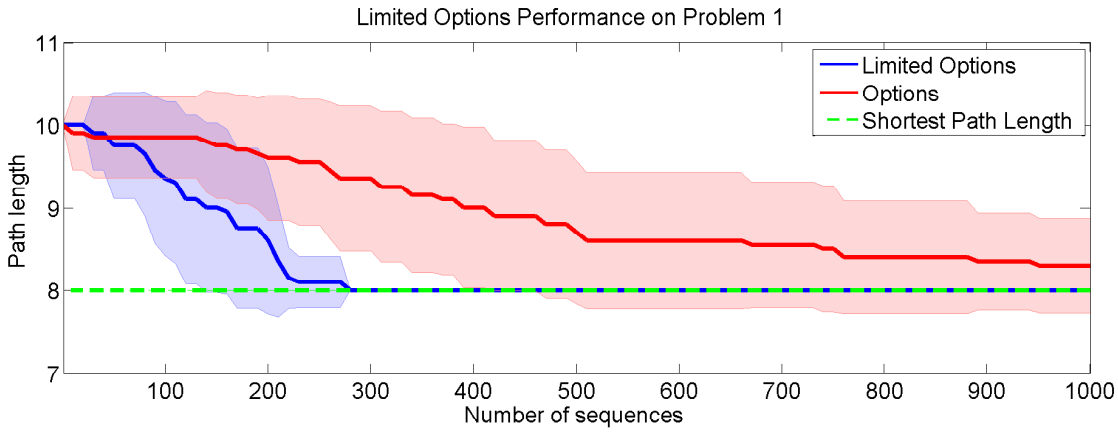
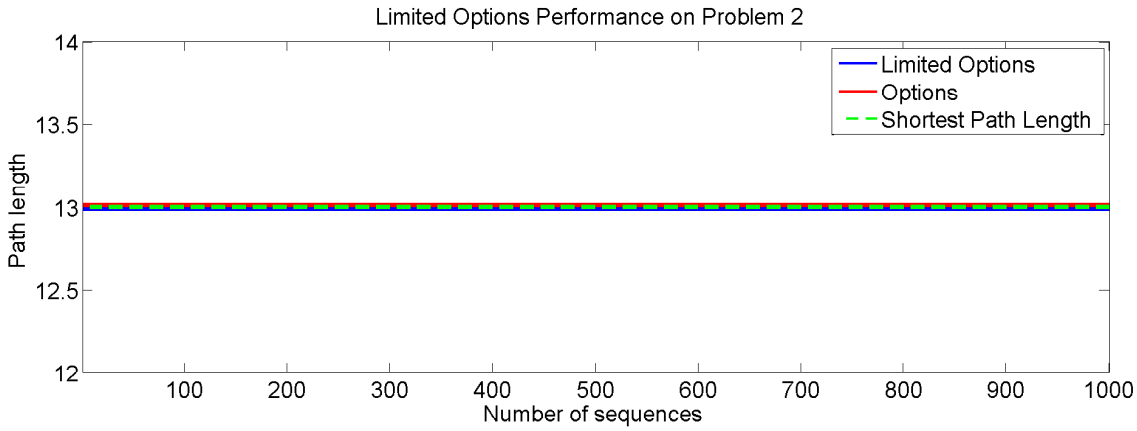


Figure 6.4: Limited Available Options. The opaque green fields correspond to the current blocks. The transparent green fields correspond to the fields that can be reached with the available options.

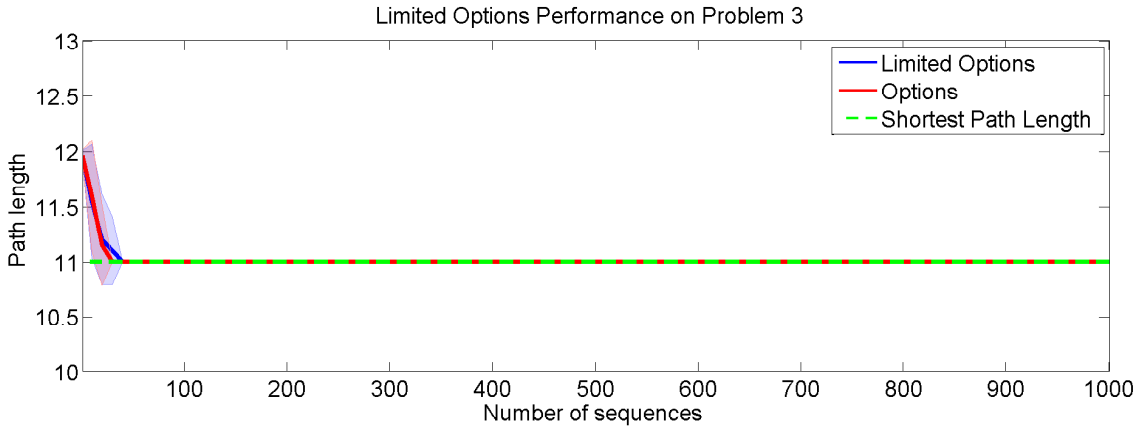
To prove that we are still able to find the shortest path, we repeat the evaluation with the limited options. The results are shown together with the results from the evaluation of the unlimited options in Figure 6.5. As we can clearly see, the limited options approach does not just find the shortest paths, but it also finds them much faster than the options approach without any limits on the availability of options. Due to the decreased number of available options much less exploration is necessary.



(a) Evaluation Result for Problem 1

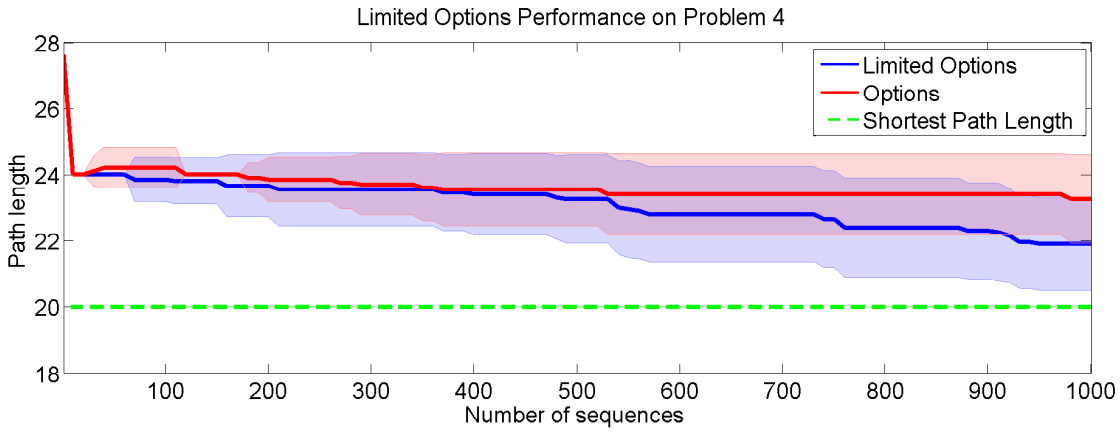


(b) Evaluation Result for Problem 2

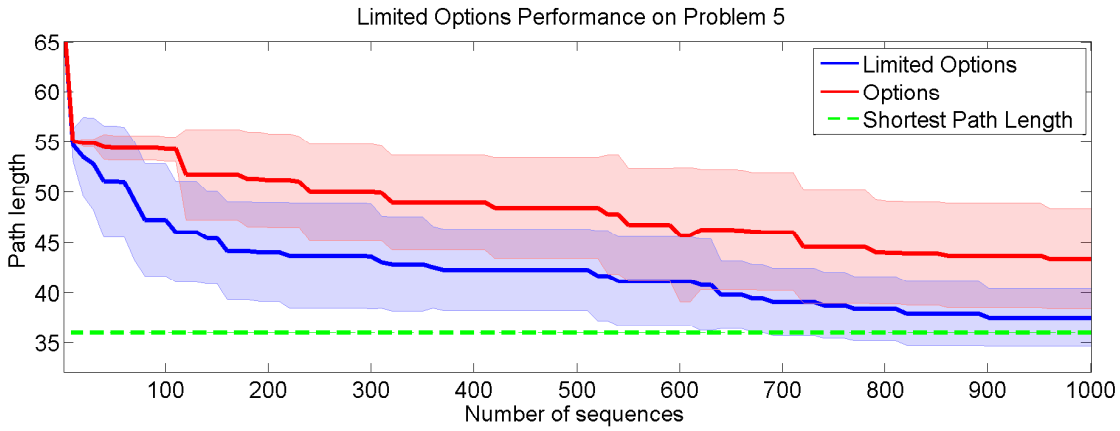


(c) Evaluation Result for Problem 3

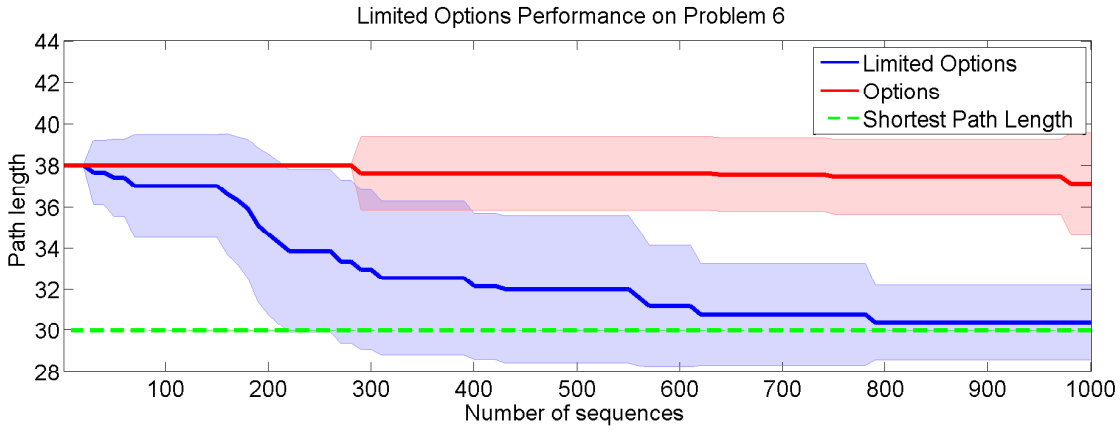
Figure 6.5: Evaluation Results for the Limited Options Approach.



(d) Evaluation Result for Problem 4



(e) Evaluation Result for Problem 5



(f) Evaluation Result for Problem 6

Figure 6.5: Evaluation Results for the Limited Options Approach (cont.).

6.3.5 Options with LSTD Learning

Our HRL approach has performed quite well so far. However, with the learned Q values, we are only able to provide a solution to a problem from states which have been visited. If we moved for example every block one step to the right, it might be that we have not experienced this situation before and do not know, which option to choose, although the problem itself is exactly the same. To be able to deal with such cases, we can combine LSTD learning with the option approach. Instead of learning Q values for all experienced state-option pairs, we can learn feature weights for a linear value function. This means that we only use the presented options for an advanced action selection, but behave otherwise the same as for the LSTD learning.

One of the problems with our LSTD learning approach was that evaluation problem 6 (Figure 5.10(f)) was less difficult for our agent than a larger version of this problem. To show that we have overcome this increase in difficulty, we have evaluated our agent on a stretched version of problem 6, which is twice as high and wide as the original problem and both the distances between the two blocks and the distances from their start to their goal positions have doubled, too. Figure 6.6 shows the evaluation of both problems. As we can undoubtedly see, our algorithm performs equally well on both problems. The almost independence of the game board size allows us to take the transition from discrete to (nearly) continuous states, which can be modeled with a more finely granulated game board. This ability enables applications that were presented in chapter 1. In Figure 1.3 for example the robot would be able to move the objects on the table in steps as little as necessary for its task without an increase in effort.

In the next two sections we will discuss the concepts of HAMs and apply them to our given problem.

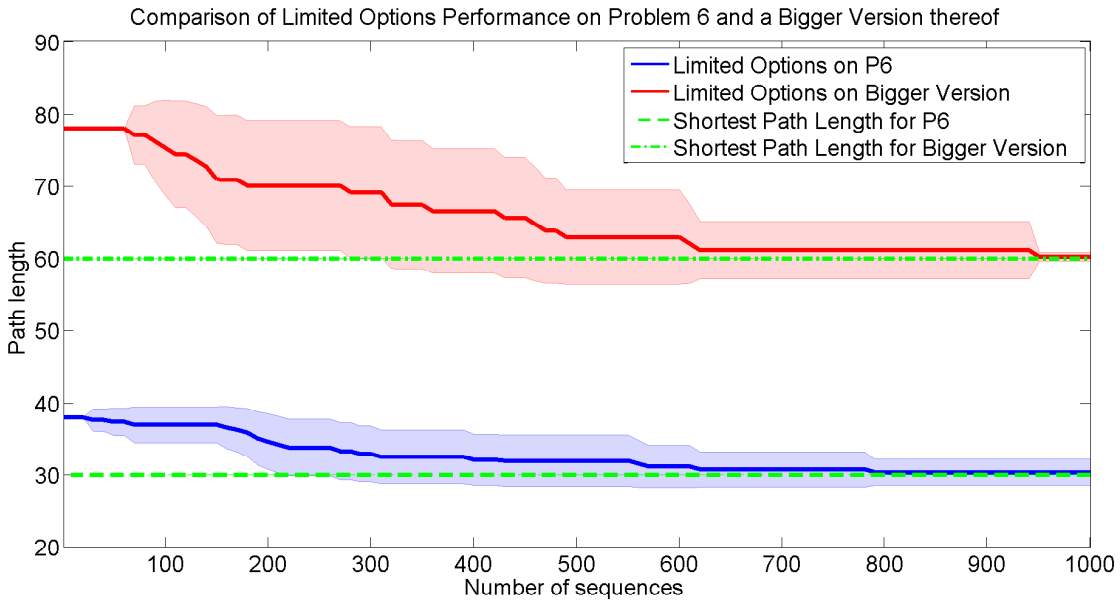


Figure 6.6: Comparison of the Evaluation Results for Problem 6 and a Larger Version thereof.

6.4 Hierarchical Abstract Machines in Theory

Hierarchies of abstract machines have been developed by Parr and Russell and aim at the restriction of realizable policies to simplify complex MDPs [38, 3]. A HAM $\mathcal{H}$ consists of a collection of non-deterministic finite state machines $\mathcal{H}_i$. These are defined by a set of states S_i, a transition function T_i and a start function, which determines the initial state of the machine, $\mathcal{F}_i : S \mapsto S_i$, where S is the set of states of the MDP. There are four different types of machine states [38]:

- *Action states* execute actions of the core MDP.
- *Call states* suspend the execution of the current machine $\mathcal{H}_i$ and start the execution of a machine $\mathcal{H}_j$.
- *Choice states* non-deterministically select the next state of the machine.
- *Stop states* terminate the execution of machine $\mathcal{H}_i$ and return control to its caller machine.

The transition function specifies the next machine state after an action or call state. A HAM $\mathcal{H}$ is defined by the initial machine together with the closure of all machines reachable from the initial machine [38].

The application of a HAM $\mathcal{H}$ to a HAM-consistent MDP $\mathcal{M}$ yields an *induced* MDP $\mathcal{H} \circ \mathcal{M}$. The set of states of $\mathcal{H} \circ \mathcal{M}$ is the cross product of the states of $\mathcal{H}$ and the states of $\mathcal{M}$. For each new state whose machine component is an action state, the model and machine transition functions are combined. For each state whose machine component is a choice state, actions are introduced that change only the machine component of the state. Lastly, the reward is taken from $\mathcal{M}$ for primitive actions and otherwise set to zero. Parr shows that the induced HAM is a MDP and that an optimal policy π for $\mathcal{H} \circ \mathcal{M}$ specifies and optimal policy for $\mathcal{M}$ which satisfies the restrictions from $\mathcal{H}$. Furthermore, $\mathcal{H} \circ \mathcal{M}$ can be reduced to a MDP whose set of states corresponds to the set of choice points in the induced MDP. With the *HAM Q-learning* algorithm an agent can even directly learn in this reduced state space without any model transformations [38].

HAM Q-learning is nothing else than SMDP Q-learning, which we have described in section 6.2, applied to the induced MDP. Let s_c be a state of $\mathcal{M}$ and m_c a choice state of $\mathcal{H}$. When the agent is in $[s_c, m_c]$, takes action a_c and ends up in $[s'_c, m'_c]$ after τ primitive steps with an accumulated, discounted, received reward R_c, then the update rule appears as follows:

$$Q_{k+1}([s_c, m_c], a_c) = (1 - \alpha_k) Q_k([s_c, m_c], a_c) + \alpha_k \left[R_c + \gamma^\tau \max_{a'_c} Q_k([s'_c, m'_c], a'_c) \right] \tag{6.3}$$

The Q value update is applied for each transition to a choice point. Under the assumptions given in section 6.2 HAM Q-learning will converge to the optimal choice for every choice point [38].

6.5 Application of Hierarchical Abstract Machines

We now want to apply the concept of HAMs to our given problem. For that reason, we define our finite state machines such that we force some structure on the problem. Afterwards, we will present a variation

of the actual concept in order to make knowledge transferable between different parts of the problem and between different problems. Finally, we will evaluate this approach on our test problem set.

6.5.1 Defining the Machines

In a first step, we have to define the finite state machines of our HAM. We have chosen a four-level hierarchy. On the first level the *controller machine* (Figure 6.7) chooses a *group machine*, which it calls. When the group machine terminates and returns the control to the controller machine, it is checked, whether the complete problem is solved. If it is not solved, a new group is chosen. If it is solved, the machine stops and we are done. A group consists of several blocks of the given problem, which might interact with each other. Every block belongs to exactly one group and blocks from different group cannot interact with each other. The algorithm that divides the blocks into groups, computes the overlap of rows and columns that two blocks need to traverse, similar to the cost / bonus features from section 5.8. The blocks from two different groups do not share any rows or columns with each other.

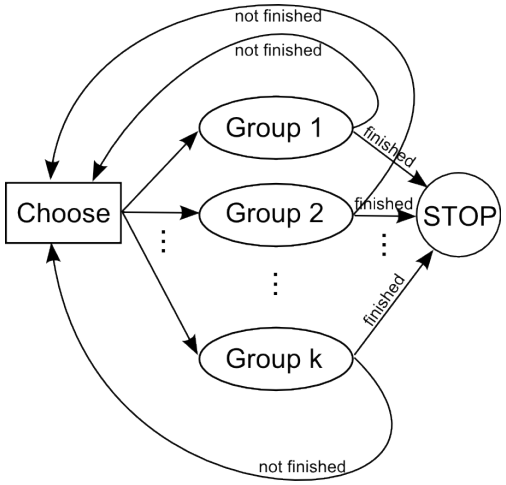


Figure 6.7: Controller Machine. This machine can call all *group machines*.

The group machines (Figure 6.8) are on the second level of the hierarchy. They stop, when all blocks of the group are on their goal positions. The group machines can call *align machines*, which align two given blocks either horizontally or vertically, and *option machines*. Align machines, on the third level of the hierarchy, call option machines and stop, when the two blocks are aligned. Option machines correspond to the limited options from the last section and translate the j-step movement into primitive actions.

6.5.2 Making Knowledge Transferable

The advantage we have gained so far over the limited options approach is that we can explicitly align two blocks. However, the alignment of the blocks 1 and 2 would be learned separately from the alignment of the blocks 3 and 4. What we would actually prefer is that the knowledge we obtain from the alignment of some blocks can be transferred to the alignment of others. For that reason, we will break with the rules of the HAMs and introduce some abstraction for the alignment learning. Instead of considering the

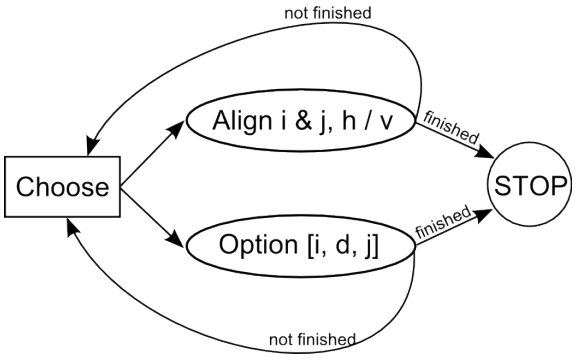


Figure 6.8: Group Machine. This machine can call all *align* and *option machines*.

whole state of the MDP and the current machine ($[s_c, m_c]$), we will abstract from that and only consider the information which is contained in the relative current positions of the two blocks and their goal positions. This leads to the circumstance that for alignment learning neither the specific block numbers, nor the other blocks matter.

Another modification we want to apply to the algorithm refers once again to the update formula. As we have already done for the options approach, we will use again the max operator instead of the step size parameter α_k:

$$Q_{k+1}([s_c, m_c], a_c) = \max\left[Q_k([s_c, m_c], a_c), R_c + +\gamma^{\tau} \max_{a'_c} Q_k([s'_c, m'_c], a'_c)\right] \tag{6.4}$$

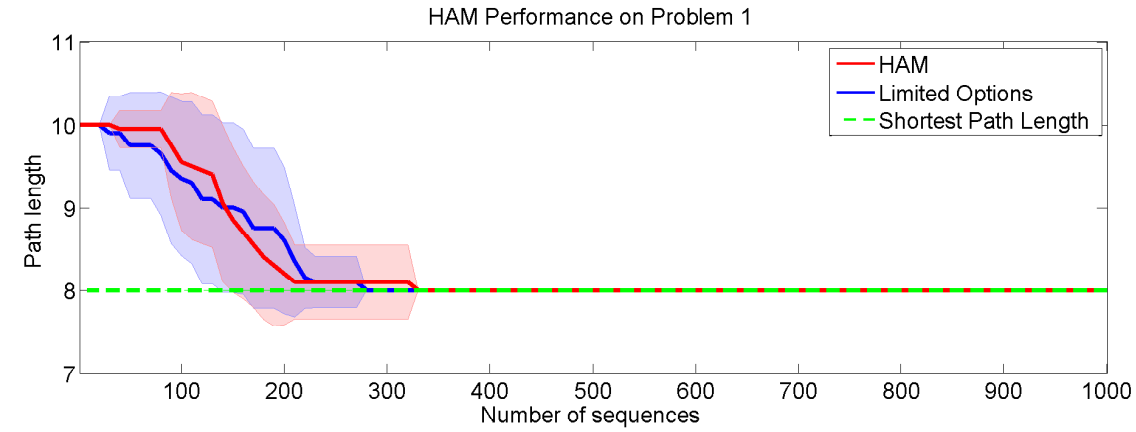
6.5.3 HAM Evaluation

We have implemented the HAM Q-learning with the modified update formula. The agent, which chooses which machine will be executed next, is an ϵ-greedy one. We stay with $\epsilon = 0.1$. Every visited induced state ($[s_c, m_c]$) is stored in a LUT and for every state all actions with the corresponding Q values are listed in a LUT, too. The initial Q values correspond to the negative trivial path lengths between the environment states (s_c) and the final goal state. Furthermore, we have used a pre-trained agent for the align machines. Hence, it knows already from training examples, how to align two blocks with least costs.

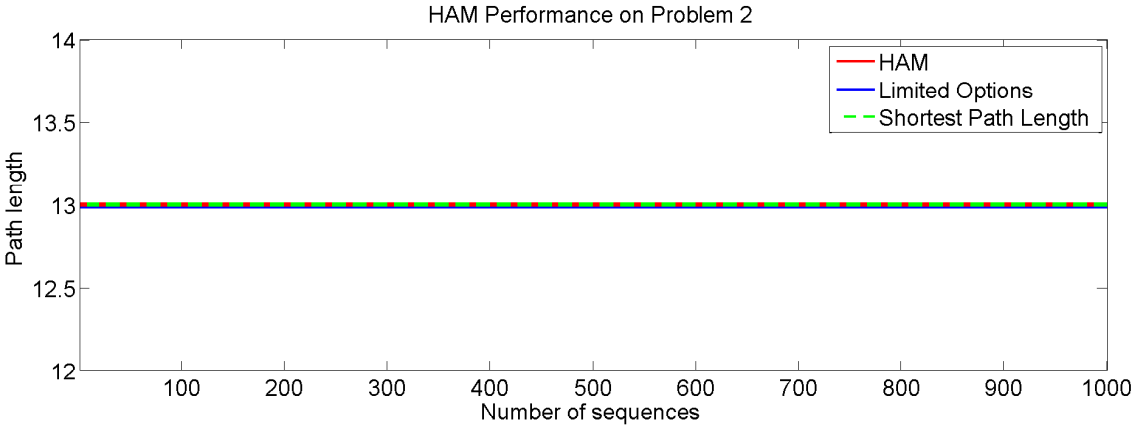
As evaluation problems we have chosen the ones which we have used before (Figure 5.10). In addition, we have executed every experiment 20 times and computed both the mean and the standard deviation of the found path lengths. The results for all evaluation problems together with the results from the evaluation of the limited options approach can be seen in Figure 6.9. For the problems 1, 2 and 4 the HAMs approach does equally well as the limited options approach, whereas the HAM outperforms the limited options for the problems 3, 5 and 6. In problem 3 the HAM agent is able to learn the one together movement of the two blocks faster, since it knows how to align the blocks. For the same reason, it is able to learn the positive effect of the alignment of the blocks in problem 6 faster, too. In problem 5 it additionally benefits from the partitioning of the blocks into two groups, which are then treated

separately.

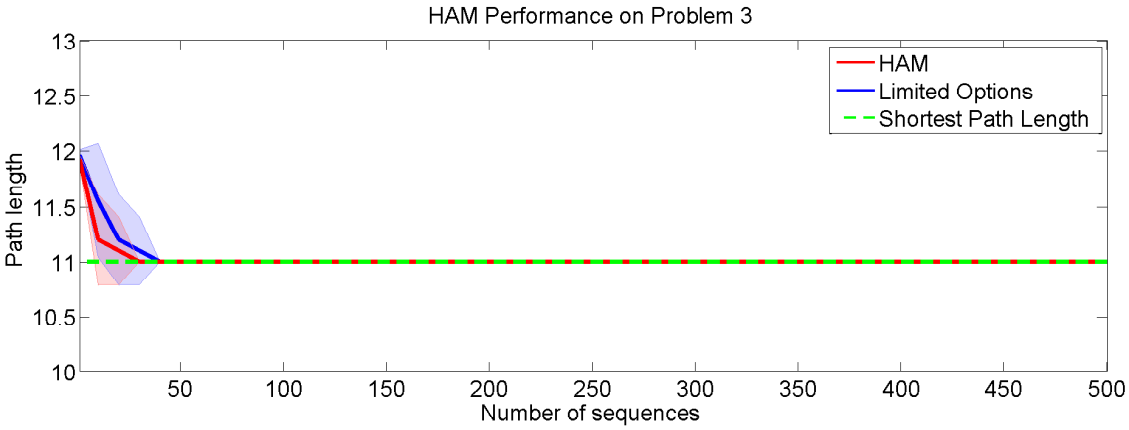
With the HAMs we are able to solve large scale problems and make use of knowledge transferred from other problems, although we still need to explicitly model which knowledge shall be made transferable in what way. In the next chapter we will summarize all our results and briefly discuss advantages and disadvantages as well as opportunities and obstacles of all the methods which we have dealt with in this thesis. Finally, we will make some suggestions for future research topics.



(a) Evaluation Result for Problem 1

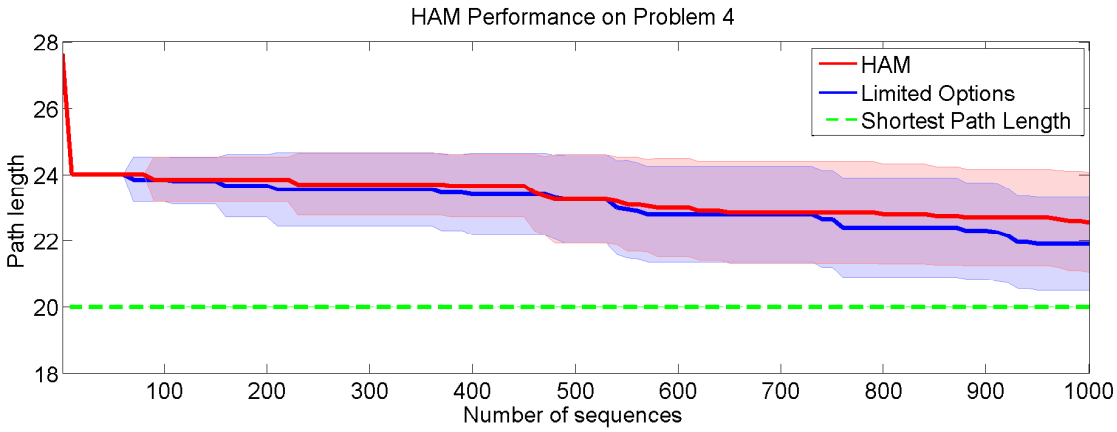


(b) Evaluation Result for Problem 2

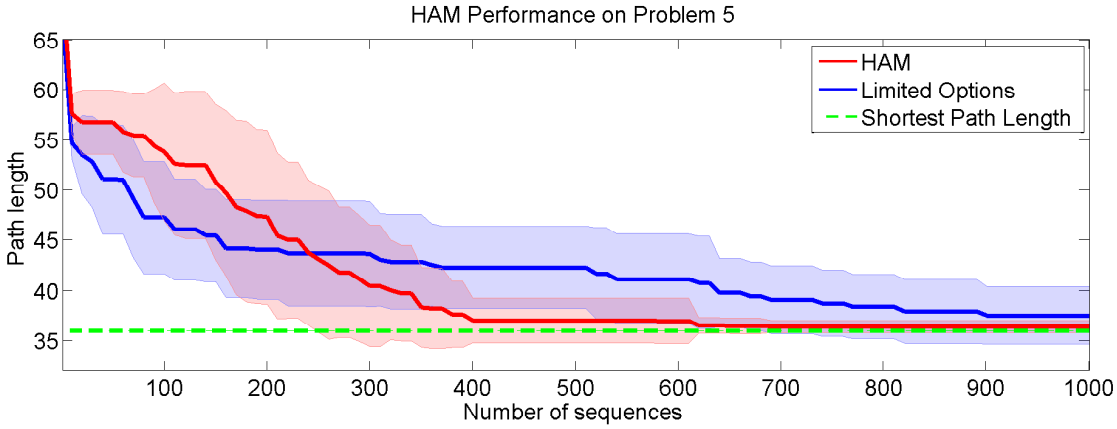


(c) Evaluation Result for Problem 3

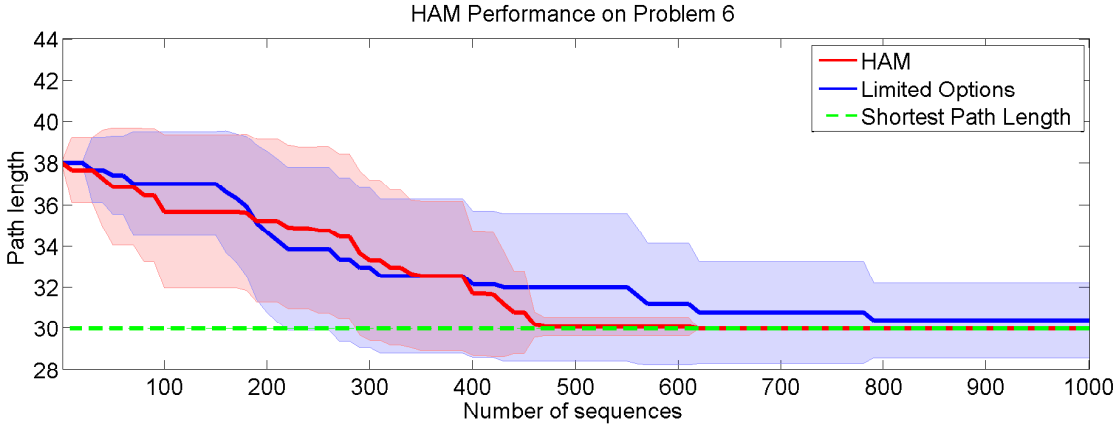
Figure 6.9: Evaluation Results for the HAM Approach.



(d) Evaluation Result for Problem 4



(e) Evaluation Result for Problem 5



(f) Evaluation Result for Problem 6

Figure 6.9: Evaluation Results for the HAM Approach (cont.).

7 Conclusion and Future Work

In this thesis we have presented several approaches to the given problem of reinforcement learning for planning in high-dimensional domains. In the following, we want to summarize the most important results, discuss advantages and disadvantages as well as opportunities and obstacles of the methods we have introduced and suggest some topics for future research.

7.1 Conclusion

Our goal was to identify methods which can be used for solving planning problems in high-dimensional domains. The specific problem we dealt with in this thesis is the one of making an agent find the shortest path between a start and an end configuration of blocks, which can be pushed along the axes of a two-dimensional game board. The number of states of this problem grows nearly exponentially with the size of the game board and the number of blocks. For that reason, neither straightforward search algorithms, e.g. *A** or *beam search*, nor classical techniques for the exact solutions of *MDPs*, such as value or policy iteration, can handle finding optimal solutions to medium or large scale problems. These approaches are very general and independent of the concrete problems which they are applied to. Therefore, they cannot take advantage of domain specifics, as for example the fact that only a small subset of states is interesting or the fact that many different situations are very similar to each other. Hence, we have taken a closer look at *FMDPs*, *LSTD learning* and *hierarchical reinforcement learning*.

The idea behind FMDPs is to find regularities in MDPs, which can then be exploited to reduce the computational costs by an exponential factor. This approach sounded very promising, since it finds the optimal solution for every start state. However, due to the dependencies between blocks - when pushing a block a neighboring block will move as well - no regularities of the factored representation could be exploited in our case. We think that the number of practical problems that can benefit greatly from the concept of FMDPs is very limited. The reason for our belief is that a single factored state representation, in which variables are only allowed to be dependent on a small subset of all variables, has to act as the basis for both a linear value function and an additive reward function, whose basis functions are only allowed to depend on a small subset of variables. Consequently, FMDPs can only be successfully applied to problems that contain sufficient regularities and consist of highly decoupled parts.

A lot of research has already been done on TD learning methods, which allow the approximation of complex value functions by linear ones. Their advantages, from which our LSTD(λ) implementation has benefited greatly, are the opportunities to focus on interesting subsets of states (trajectory sampling) and the fact that eligibility / replacing traces can be used to efficiently backpropagate future benefits to the beginning, where detour moves may have been chosen. Both aspects have accelerated the agent's learning process in our experiments a lot. However, the selection of features was a cumbersome task. We have tried many possibilities that intuitively made sense to us, but failed to solve some of the problems. Nevertheless, in the end, we have found some features that were capable of modeling the problem

correctly. Yet, the number of these features is very large for problems involving many blocks. Two other issues of LSTD learning concern exploration and scaling up to larger problems. To make the agent find the shortest path, a lot of exploration might be needed. Besides, when the size of the game board is increased, the agent needs not only much more time for finishing a sequence, but it also has to do a lot more exploration. These aspects have then led us to the concept of temporal abstraction.

Although the idea of hierarchical RL, aiming at overcoming the curse of dimensionality, has risen about 15 years ago, it seems as if there were still many interesting, open questions. HRL enables us to define actions that operate over multiple time steps. These behaviors help us to cope with larger game board sizes and enable the transition from discrete to nearly continuous states, as we have empirically evaluated. Forcing structures on the policies which we want to learn supports us to focus on interesting subsets of states. An advantage of hierarchical Q-learning over LSTD learning is that the agent is always able to find the goal state and does not get stuck, which happens in the latter case for badly chosen features. In addition, HRL provides us with the possibility to conduct goal-directed learning by allowing us to specify and learn the alignment of blocks, which has significantly fastened the process of finding good policies. Last but not least, HRL provides a framework for making knowledge transferable between problems by means of abstraction. Yet, both the hierarchies and the knowledge transfer still have to be specified manually and require domain knowledge. All in all the HAMs approach with incorporated options on a low-level machine has turned out to perform best for our planning problem.

7.2 Future Work

In the field of RL, there is still lots of interesting research to be done. In this section, we briefly want to describe some approaches related to the subjects we have spent time on that might be worth to be explored.

Concerning FMDPs, one could think of how to still exploit structure and reduce computational costs for a problem, although there are dependencies between all state variables for a relatively small number of states. Furthermore, one could try to combine FMDPs with other solution methods than only DP or LP. For instance, the inherent structure might also be beneficial for TD learning approaches. Another interesting question is, whether it is possible to have two different factored representations of the same problem which are profitably interlinked. One of these representations could then be used for modeling the state transitions, while the other one is favorable for the value function approximation. Possibly, problems could also be transformed in a first step by dimensionality reduction techniques to decrease the dependence of state variables on each other so that the concept of FMDPs can be applied to those problems more easily.

For a value function approximation for (LS)TD learning it would be great to have some algorithm that can extract features that are able to model the problem correctly. As we have already mentioned, there has been proposed a method in [29], dealing with neighborhood component analysis, which, however, only has a limited scope. The automatically extracted features should be chosen in a way such that the learning algorithm yields similar feature weights for different parts of the problems. Another idea might even go beyond the extraction of features by autonomously learning a local weighting of them in

a second step.

In the epigraph at the beginning of chapter 1 we cited Sutton, who said that for AI it was not enough to achieve a better system, but it mattered how the system had been made. In our HRL approach the agent has learned how to solve instances of the given problem. It was also able to extract some of the knowledge and use it for other problems. However, in the designing process we had to handcraft which knowledge should be transferred in what way. This can become very cumbersome when the level of abstraction of primitive actions increases. For that reason, it would be beneficial to make an agent *learn* how to achieve such a transfer of knowledge.

References

[1] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. An introduction to mcmc for machine learning. *Machine learning*, 50(1-2):5–43, 2003.

[2] Leemon Baird and Andrew W Moore. Gradient descent for general reinforcement learning. *Advances in neural information processing systems*, pages 968–974, 1999.

[3] Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379, 2003.

[4] Craig Boutilier, Richard Dearden, and Moises Goldszmidt. Exploiting structure in policy construction. In *IJCAI*, volume 14, pages 1104–1113, 1995.

[5] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1):49–107, 2000.

[6] Justin A Boyan. Technical update: Least-squares temporal difference learning. *Machine Learning*, 49(2-3):233–246, 2002.

[7] Steven J Bradtke and Andrew G Barto. Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22(1-3):33–57, 1996.

[8] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1):57–83, 2002.

[9] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9(3):251–280, 1990.

[10] Heriberto Cuayáhuitl, Steve Renals, Oliver Lemon, and Hiroshi Shimodaira. Evaluation of a hierarchical reinforcement learning spoken dialogue system. *Computer Speech & Language*, 24(2):395–429, 2010.

[11] Christoph Dann, Gerhard Neumann, and Jan Peters. Policy evaluation with temporal differences: A survey and comparison. submitted.

[12] Thomas Dean, Robert Givan, and Sonia Leach. Model reduction techniques for computing approximately optimal solutions for markov decision processes. In *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*, pages 124–131. Morgan Kaufmann Publishers Inc., 1997.

[13] Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. *Computational intelligence*, 5(2):142–150, 1989.

[14] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of a*. *Journal of the ACM (JACM)*, 32(3):505–536, 1985.

[15] Thomas Degris, Olivier Sigaud, and Pierre-Henri Wuillemin. Learning the structure of factored markov decision processes in reinforcement learning problems. In *Proceedings of the 23rd international conference on Machine learning*, pages 257–264. ACM, 2006.

[16] Thomas Degris, Olivier Sigaud, and Pierre-Henri Wuillemin. Exploiting additive structure in factored mdps for reinforcement learning. In *Recent Advances in Reinforcement Learning*, pages 15–26. Springer, 2008.

[17] Nina Dethlefs and Heriberto Cuayáhuitl. Hierarchical reinforcement learning for adaptive text generation. In *Proceedings of the 6th International Natural Language Generation Conference*, pages 37–45. Association for Computational Linguistics, 2010.

[18] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[19] Steven J Duff and Bradtke Michael O. Reinforcement learning methods for continuous-time markov decision problems. *Advances in Neural Information Processing Systems: 7*, 7, 1995.

[20] Damien Ernst, Raphaël Marée, and Louis Wehenkel. Reinforcement learning with raw image pixels as input state. In *Advances in Machine Vision, Image Processing, and Pattern Analysis*, pages 446–454. Springer, 2006.

[21] David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A Kalyanpur, Adam Lally, J William Murdock, Eric Nyberg, John Prager, et al. Building watson: An overview of the deepqa project. *AI magazine*, 31(3):59–79, 2010.

[22] Carlos Guestrin, Daphne Koller, Ronald Parr, and Shobha Venkataraman. Efficient solution algorithms for factored mdps. *J. Artif. Intell. Res.(JAIR)*, 19:399–468, 2003.

[23] Carlos Guestrin, Relu Patrascu, and Dale Schuurmans. Algorithm-directed exploration for model-based reinforcement learning in factored mdps. In *ICML*, pages 235–242, 2002.

[24] Bernhard Hengst. Discovering hierarchy in reinforcement learning with hexq. In *ICML*, volume 2, pages 243–250, 2002.

[25] Matthew Howard, Yoshihiko Nakamura, and Matthew Howard. Locally weighted least squares temporal difference learning, 2013.

[26] Aapo Hyvärinen and Erkki Oja. Independent component analysis: algorithms and applications. *Neural networks*, 13(4):411–430, 2000.

[27] Nicholas K Jong and Peter Stone. Hierarchical model-based reinforcement learning: R-max+ maxq. In *Proceedings of the 25th international conference on Machine learning*, pages 432–439. ACM, 2008.

[28] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *arXiv preprint cs/9605103*, 1996.

[29] Philipp W Keller, Shie Mannor, and Doina Precup. Automatic basis function construction for approximate dynamic programming and reinforcement learning. In *Proceedings of the 23rd international conference on Machine learning*, pages 449–456. ACM, 2006.

[30] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, Eiichi Osawa, and Hitoshi Matsubara. Robocup: A challenge problem for ai. *AI magazine*, 18(1):73, 1997.

[31] Daphne Koller and Ronald Parr. Computing factored value functions for policies in structured mdps. In *IJCAI*, volume 99, pages 1332–1339, 1999.

[32] Daphne Koller and Ronald Parr. Policy iteration for factored mdps. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pages 326–334. Morgan Kaufmann Publishers Inc., 2000.

[33] John John Aldo Lee and Michel Verleysen. *Nonlinear dimensionality reduction*. Springer, 2007.

[34] Jeremy Maitin-Shepard, Marco Cusumano-Towner, Jinna Lei, and Pieter Abbeel. Cloth grasp point detection based on multiple-view geometric cues with application to robotic towel folding. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2308–2315. IEEE, 2010.

[35] Bogdan Moldovan, Plinio Moreno, Martijn van Otterlo, José Santos-Victor, and Luc De Raedt. Learning relational affordance models for robots in multi-object manipulation tasks. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 4373–4378. IEEE, 2012.

[36] A Nedić and Dimitri P Bertsekas. Least squares policy evaluation algorithms with linear function approximation. *Discrete Event Dynamic Systems*, 13(1-2):79–110, 2003.

[37] Andrew Y Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Autonomous inverted helicopter flight via reinforcement learning. In *Experimental Robotics IX*, pages 363–372. Springer, 2006.

[38] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, pages 1043–1049, 1998.

[39] Maayan Roth, Reid Simmons, and Manuela Veloso. Exploiting factored representations for decentralized execution in multiagent teams. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 72. ACM, 2007.

[40] Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson Higher Ed, 2009.

[41] Ashutosh Saxena, Justin Driemeyer, and Andrew Y Ng. Robotic grasping of novel objects using vision. *The International Journal of Robotics Research*, 27(2):157–173, 2008.

[42] Jennie Si, Andrew G Barto, Warren B Powell, Donald C Wunsch, et al. *Handbook of learning and approximate dynamic programming*. IEEE Press Los Alamitos, 2004.

[43] Satinder P Singh and Richard S Sutton. Reinforcement learning with replacing eligibility traces. *Machine learning*, 22(1-3):123–158, 1996.

[44] Alexander L Strehl, Carlos Diuk, and Michael L Littman. Efficient structure learning in factored-state mdps. In *AAAI*, volume 7, pages 645–650, 2007.

[45] Rich Sutton. What's wrong with artificial intelligence. http://webdocs.cs.ualberta.ca/~sutton/IncIdeas/WrongWithAI.html, 2001. [Online; accessed 08/02/2013].

[46] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[47] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. Cambridge Univ Press, 1998.

[48] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999.

[49] Richard S Sutton, Doina Precup, and Satinder P Singh. Intra-option learning about temporally abstract actions. In *ICML*, volume 98, pages 556–564, 1998.

[50] Gerald Tesauro. *Practical issues in temporal difference learning*. Springer, 1992.

[51] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

[52] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[53] Rong Zhou and Eric A Hansen. Beam-stack search: Integrating backtracking with beam search. In *ICAPS*, pages 90–98, 2005.