Mach Learn (2006) 64:91–119
DOI 10.1007/s10994-006-8258-y

Graph kernels and Gaussian processes for relational reinforcement learning

Kurt Driessens · Jan Ramon · Thomas Gärtner

Received: 7 August 2004 / Revised: 9 March 2006 / Accepted: 11 March 2006 / Published online: 8 May 2006

Abstract RRL is a relational reinforcement learning system based on Q-learning in relational state-action spaces. It aims to enable agents to learn how to act in an environment that has no natural representation as a tuple of constants. For relational reinforcement learning, the learning algorithm used to approximate the mapping between state-action pairs and their so called Q(uality)-value has to be very reliable, and it has to be able to handle the relational representation of state-action pairs. In this paper we investigate the use of Gaussian processes to approximate the Q-values of state-action pairs. In order to employ Gaussian processes in a relational setting we propose graph kernels as a covariance function between state-action pairs. The standard prediction mechanism for Gaussian processes requires a matrix inversion which can become unstable when the kernel matrix has low rank. These instabilities can be avoided by employing QR-factorization. This leads to better and more stable performance of the algorithm and a more efficient incremental update mechanism. Experiments conducted in the blocks world and with the Tetris game show that Gaussian processes with graph kernels can compete with, and often improve on, regression trees and instance based regression as a generalization algorithm for RRL.

Editors: David Page and Akihiro Yamamoto

K. Driessens (✉)
Department of Computer Science, University of Waikato,
Private Bag 3105, Hamilton, New Zealand
e-mail: kurtd@cs.waikato.ac.nz

J. Ramon
Department of Computer Science, Katholieke Universiteit Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail: jan.ramon@cs.kuleuven.be

T. Gärtner
Knowledge Discovery, Fraunhofer Institut Autonome Intelligente Systeme,
Schloss Birlinghoven, 53754 Sankt Augustin, Germany
e-mail: thomas.gaertner@ais.fraunhofer.de

Keywords Reinforcement learning · Relational learning · Graph kernels · Gaussian processes

1. Introduction

Reinforcement learning (Sutton & Barto, 1998), in a nutshell, is about controlling an autonomous agent in an unknown environment—often called the state space. The agent has no prior knowledge about the environment and can only obtain some knowledge by acting in that environment. The only information the agent can get about the environment is the state in which it currently is and whether it received a reward. The aim of reinforcement learning is to act such that this reward is maximized.

Q-learning (Watkins, 1989)—one particular form of reinforcement learning—tries to map every state-action pair to a real number (Q-value) reflecting the quality of that action in that state, based on the experience so far. While in small state-action spaces it is possible to represent this mapping extensionally, in large state-action spaces this is not feasible for two reasons: On the one hand, one can not store the full state-action space; on the other hand the larger the state-action space gets, the smaller becomes the probability of ever visiting the same state again. For this reason, the extensional representation of the quality mapping is often substituted with an intensional mapping found by a learning algorithm that is able to generalize to unseen states. Ideally, an incrementally learnable regression algorithm is used to learn this mapping.

The RRL-algorithm (Relational Reinforcement Learning) is a Q-learning technique that can be applied whenever the state-action space can not easily be represented by tuples of constants but has an inherently relational representation instead (Džeroski et al., 1998). In this case explicitly representing the mapping from state-action-pairs to Q-values is—in general—not feasible. So far first-order distance-based algorithms (Driessens & Ramon, 2003) as well as first-order regression trees (Driessens et al., 2001) have been used as learning algorithms to approximate the mapping between state-action pairs and their Q-value.

Kernel Methods (Schölkopf & Smola, 2002) are among the most successful recent developments within the machine learning community. The computational attractiveness of kernel methods is due to the fact that they can be applied in high dimensional feature spaces without suffering from the cost of explicitly computing the feature map. This is possible by using a positive definite kernel k on any set $\mathcal{X}$. For such $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ it is known that a map $\phi : \mathcal{X} \to \mathcal{H}$ into a Hilbert space $\mathcal{H}$ exists, such that $k(x, x') = \langle \phi(x), \phi(x') \rangle$ for all $x, x' \in \mathcal{X}$. Kernel methods have so far successfully been applied to various tasks in attribute-value learning.

Gaussian processes (MacKay, 1997b) are an incrementally learnable 'Bayesian' regression algorithm. Rather than parameterizing some set of possible target functions and specifying a prior over these parameters, Gaussian processes directly put a Gaussian prior over the function space. A Gaussian process is defined by a mean function and a covariance function, specifying the prior. The choice of covariance functions is thereby only limited to positive definite kernels.

Graph kernels (Gärtner et al., 2003b) have recently been introduced as one way of extending the applicability of kernel methods beyond mere attribute-value representations. The idea behind graph kernels is to base the similarity of two graphs on the number of common label sequences. The computation of these—possibly infinitely long and infinitely many—label sequences becomes feasible by using product graphs and computing the limits of power series of their adjacency matrices.

In this paper we use Gaussian processes to learn the mapping to Q-values. One advantage of using Gaussian processes in RRL is that rather than predicting a single Q-value, they actually return a probability distribution over Q-values. In order to employ Gaussian processes in a relational setting we propose the use of graph kernels as the covariance function between state-action pairs. The model computed by the Gaussian process is basically the inverted covariance matrix. This inversion can, however, become numerically unstable when the matrix is ill-conditioned, which can be expected in the reinforcement learning setting. To avoid this instability, we employ QR-factorization instead of the usual incremental matrix inversion. We show that our theory can be updated efficiently using this method.

Experiments conducted in the blocks world and with the Tetris computer game show that Gaussian processes can compete with, and often improve on, regression trees and instance based regression as a generalization algorithm for relational reinforcement learning.

Section 2 briefly presents the relational reinforcement learning framework and discusses some previous implementations of the RRL-system. Section 3 describes kernel methods in general and discusses a few popular variations. Section 4 proposes graph kernels that are particularly well suited to deal with the structural state-action pairs that are encountered in RRL. Section 5 goes into more depth on Gaussian processes as a regression technique and discusses the use of QR-factorization to improve the computational properties of the combination of Gaussian processes and RRL. The first of the two experimental sections, Section 6.1, shows how states and actions in the blocks world can be represented by graphs and presents some experimental results that compare Gaussian processes with other regression algorithms in RRL. Section 6.2 illustrates the behavior of relational reinforcement learning with Gaussian processes on the Tetris game. Section 7 presents an overview of related work on kernel based approaches to reinforcement learning. Section 8 concludes and discusses some directions for further work.

2. Relational reinforcement learning

RRL (Džeroski et al., 1998)—a relational reinforcement learning approach—is a Q-learning algorithm that allows structural representations for states and actions.

RRL learns through exploration of the state-space in a way that is very similar to normal Q-learning (Sutton & Barto, 1998; Mitchell, 1997; Kaelbling et al., 1996). It starts with running an episode[1] just like table-based Q-learning, but uses the encountered states, chosen actions and the received awards to generate a set of examples that can then be used to build a Q-function generalization. These examples use a structural representation of states and actions.

To build this generalized Q-function, RRL applies incremental relational regression that can exploit the structural representation of the constructed example set. The resulting Q-function is then used to decide which actions to take in the following episodes. Every new episode can be seen as new experience and is thus used to update the model of the Q-function. A more formal description of the RRL-algorithm is given in Algorithm 1.

Previous implementations of the RRL-system have used first order regression trees and relational instance based regression to build a generalized Q-function. In this work, we suggest using Gaussian processes as a generalization algorithm for RRL. Gaussian processes not only provide a prediction for unseen examples but can also determine a probability distribution

[1] An 'episode' is a sequence of states and actions from an initial state to a terminal state. In each state, the current Q-function is used to decide which action to take.

Algorithm 1. The main algorithm of the RRL system.

initialize the Q-function hypothesis $\hat{Q}_0$
$e \leftarrow 0$
repeat {for each episode}
  $Examples \leftarrow \emptyset$
  generate a starting state s_0
  $i \leftarrow 0$
  repeat {for each step of episode}
    choose a_i for s_i using a policy derived from
      the current hypothesis $\hat{Q}_e$
    take action a_i, observe r_i and s_{i+1}
    $i \leftarrow i + 1$
  until s_i is terminal
  for $j = i - 1$ downto 0 **do**
    generate example $x = (s_j, a_j, \hat{q}_j)$
      where $\hat{q}_j \leftarrow r_j + \gamma max_a \hat{Q}_e(s_{j+1}, a)$
    $Examples \leftarrow Examples \cup \{x\}$
  end for
  Update $\hat{Q}_e$ using $Examples$ and a relational
    regression algorithm to produce $\hat{Q}_{e+1}$
  $e \leftarrow e + 1$
until no more episodes

over Q-values. In reinforcement learning, this probability distribution can, for example, very easily be used to determine the exploration strategy. We will compare our new approach with both previous implementations of RRL which will be briefly outlined next.

Tree based regression. RRL-TG (Driessens et al., 2001) uses an incremental first order regression tree algorithm TG to construct the Q-function. Although the TG-algorithm is very fast compared to other approaches, the performance of this algorithm depends greatly on the language definition that is used by the TG-algorithm to construct possible tree refinements. Also, TG has shown itself to be sensitive with respect to the order in which the (*state*, *action*, *qvalue*)-examples are presented and often needs more training episodes to find a competitive policy (Driessens & Džeroski, 2004).

Instance based regression. RRL-RIB (Driessens & Ramon, 2003) uses relational instance based regression for Q-function generalization. The instance based regression offers a robustness to RRL not found in RRL-TG but requires a first order distance to be defined between (state,action)-pairs.

3. Kernel methods

Kernel methods (Schölkopf & Smola, 2002) are a popular class of algorithms within the machine learning and data mining communities. Being on one hand theoretically well founded in statistical learning theory, they have on the other hand shown good empirical results in many applications. One particular aspect of kernel methods such as Gaussian processes and support vector machines is the formation of hypotheses by linear combination of positive-definite

kernel functions 'centered' at individual training examples. By the restriction to positive definite kernel functions, the underlying optimization problem becomes convex and every locally optimal solution is globally optimal.

Kernel methods can be applied to different kinds of (structured) data by using any positive definite kernel function defined on the data.

Definition 3.1. Let $\mathcal{X}$ be a set. A symmetric function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a *positive definite kernel* on $\mathcal{X}$ if, for all $n \in \mathbb{Z}^+$, $x_1, \ldots, x_n \in \mathcal{X}$, and $c_1, \ldots, c_n \in \mathbb{R}$, it follows that

$$\sum_{i,j \in \{1,\ldots,n\}} c_i \; c_j \; k(x_i, x_j) \geq 0 \; .$$

3.1. Kernel machines

The usual supervised learning model (Vapnik, 1995) considers a set $\mathcal{X}$ of individuals and a set $\mathcal{Y}$ of labels, such that the relation between individuals and labels is a fixed but unknown probability measure on the set $\mathcal{X} \times \mathcal{Y}$. The common theme in many different kernel methods such as support vector machines, Gaussian processes, or regularized least squares regression is to find a hypothesis function that minimizes not just the empirical risk (training error) but the *regularized risk*. This gives rise to the optimization problem

$$\min_{f(\cdot) \in \mathcal{H}} \frac{C}{n} \sum_{i=1}^{n} V(y_i, \, f(x_i)) + \|f(\cdot)\|^2_{\mathcal{H}}$$

where $\{(x_i, \, y_i)\}_{i=1}^n$ is a set of individuals with known label (the training set), $\mathcal{H}$ is a set of functions forming a Hilbert space (the hypothesis space), $C(>0)$ is a regularisation parameter and V is a function that takes on small values whenever $f(x_i)$ is a good guess for y_i and large values whenever it is a bad guess (the loss function). The *representer theorem* shows that under rather general conditions on V, solutions of the above optimization problem have the form

$$f(\cdot) = \sum_{i=1}^{n} c_i k(x_i, \cdot) \; . \tag{1}$$

Different kernel methods arise from using different loss functions.

3.1.1. Regularized least squares

Choosing the square loss function, i.e., $V(y_i, \, f(x_i)) = (y_i - f(x_i))^2$, we obtain the optimization problem of the regularized least squares algorithm (Rifkin, 2002; Saunders et al., 1998):

$$\min_{f(\cdot) \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2 + \frac{C}{n} \|f(\cdot)\|^2_{\mathcal{H}} \; .$$

Plugging in our knowledge about the form of solutions and taking the directional derivative with respect to the parameter vector c of the function 1, we can find the analytic solution to

the optimization problem as:

$$c = (K + \mathbf{I}C)^{-1} y \ .$$

A strongly related kernel method is Gaussian processes.

3.1.2. Gaussian processes

Gaussian processes (MacKay, 1997b) is a 'Bayesian' regression technique that is incrementally learnable. Rather than parameterizing some set of possible target functions and specifying a prior over these parameters, Gaussian processes directly put a (Gaussian) prior over the function space. A Gaussian process is defined by a mean function and a covariance function, implicitly specifying the prior. The choice of covariance functions is thereby only limited to positive definite kernels. It can be seen that the mean prediction of a Gaussian process corresponds to the prediction found by a regularized least squares algorithm. This links the regularization parameter C with the variance of the Gaussian noise distribution assumed in Gaussian processes.

3.1.3. Illustration

Figure 1 illustrates the impact of choosing the parameter of a Gaussian kernel function on the regularization of the solution found by a Gaussian process. Training examples are single

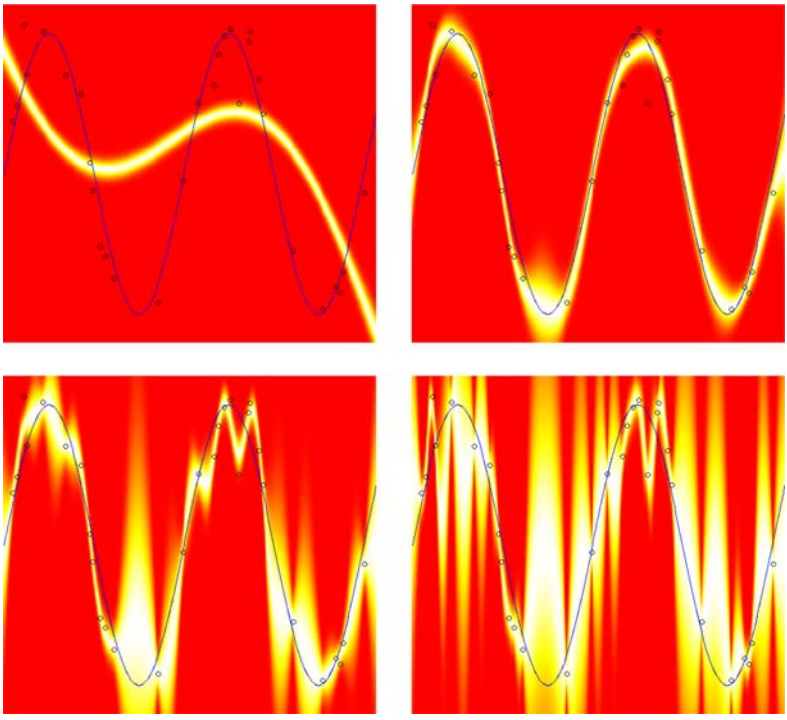

Fig. 1 Impact of the bandwidth of a Gaussian kernel function on the regularization of the solution found by Gaussian processes. The bandwidth is decreasing from left to right, top to bottom

real numbers and the target value is also a real number. The unknown target function is a sinusoid function shown by a thin line in the figure. Training examples perturbed by random noise are depicted by black circles. The lightness of each pixel illustrates the distribution of target values given the example, the most likely value is always white.

A more detailed discussion of Gaussian processes follows in Section 5.

4. Graph kernels

Having in the previous section introduced kernel methods, we will in this section show how they can be applied in state action spaces that have a natural representation as a graph. One of the two testing environments—namely the blocks world—is easily represented as a graph, and thus employing graph kernels as the covariance function of a Gaussian process seems evident. This section gives a brief overview of graphs and graph kernels. For a more in-depth discussion of graphs the reader is referred to Diestel (2000) and Korte & Vygen (2002). For a discussion of different graph kernels (see Gärtner et al. 2003b).

One approach to define kernels on objects that have a natural representation as a graph is to decompose each graph into a set of subgraphs and measure the intersection of two decompositions. With such a graph kernel, one could decide whether a graph has a Hamiltonian path or not (Gärtner et al., 2003b). As this problem is known to be NP-hard, it is strongly believed that this graph kernel can not be computed in polynomial time. This holds even if the decomposition is restricted to paths only.

In literature different approaches have thus been tried to overcome this problem. Graepel (2002) restricted the decomposition to paths up to a given size, and Deshpande et al. (2002) only consider the set of connected graphs that occur frequently as subgraphs in the graph database. The approach taken there to compute the decomposition of each graph is an iterative one (Kuramochi & Karypis, 2001). The algorithm starts with a frequent set of subgraphs with one or two edges only. Then, in each step, from the set of frequent subgraphs of size l, a set of candidate graphs of size $l + 1$ is generated by joining those graphs of size l that have a subgraph of size $l - 1$ in common. Of the candidate graphs only those satisfying a frequency threshold are retained for the next step. The iteration stops when the set of frequent subgraphs of size l is empty. Approaches that overcome the computational problems by restricting the set of graphs that the kernel function can be applied to can be found in Gärtner (2003).

Conceptually, the graph kernels presented in Gärtner (2002), Gärtner et al. (2003b), Kashima and Inokuchi (2002), Kashima et al. (2003) are based on a measure of the walks in two graphs that have some or all labels in common. In Gärtner (2002) walks with equal initial and terminal label are counted, in Kashima and Inokuchi (2002); Kashima et al. (2003) the probability of random walks with equal label sequences is computed, and in Gärtner et al. (2003b) walks with equal label sequences, possibly containing gaps, are counted. In Gärtner et al. (2003b) computation of these—possibly infinite—walks is made possible in polynomial time by using the direct product graph and computing the limit of matrix power series involving its adjacency matrix. The work on rational graph kernels (Cortes et al., 2003) generalizes these graph kernels by applying a transducer to weighted automata instead of forming the direct graph product. However, only walks up to a given length are considered in the kernel computation. More recently, Horvath et al. (2004) suggested that the computational intractability can be overcome in practical applications by observing that 'difficult structures' occur only infrequently in real-world databases. As a consequence of this assertion, Horvath et al. (2004) use a cycle enumeration algorithm to decompose all graphs in a molecule database into all simple cycles occurring.

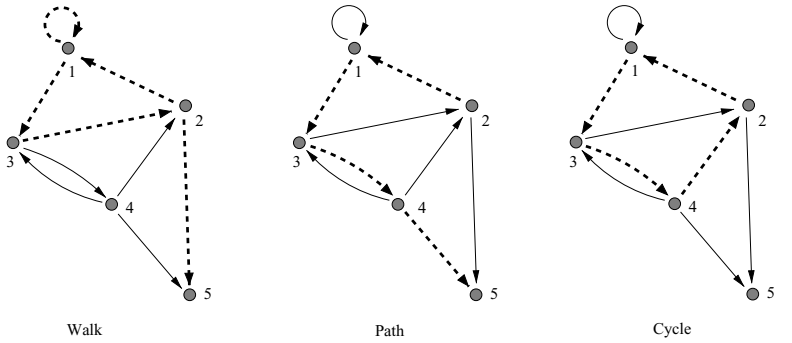

Fig. 2 From left to right, a *walk*, *path* and *cycle* in a directed graph

In the remainder of this section we will describe a modification of walk based graph kernels (Gärtner et al., 2003b) that allows for graphs with parallel edges. For this, we first need to introduce a few basic concepts on graphs.

4.1. Labeled directed graphs

Generally, a *graph* G is described by a finite set of *vertices* $\mathcal{V}$, a finite set of *edges* $\mathcal{E}$, and a function Ψ. For *labeled* graphs there is additionally a set of labels $\mathcal{L}$ along with a function *label* assigning a label to each edge and vertex. We will sometimes assume some enumeration of the vertices and labels in a graph,[2] i.e., $\mathcal{V} = \{v_i\}_{i=1}^{n}$ where $n = |\mathcal{V}|$ and $\mathcal{L} = \{\ell_r\}_{r\in\mathbb{N}}$. For *directed* graphs the function Ψ maps each edge to the tuple consisting of its initial and terminal node $\Psi : \mathcal{E} \to \mathcal{V} \times \mathcal{V}$. Edges e in a directed graph for which $\Psi(e) = (v, v)$ are called *loops*. Two edges e, e' are *parallel* if $\Psi(e) = \Psi(e')$. Frequently, only graphs without parallel edges are considered. In our application, however, it is important to also consider graphs with parallel edges.

To refer to the vertex and edge set of a specific graph we will sometimes use the notation $\mathcal{V}(G), \mathcal{E}(G)$. Wherever we distinguish two graphs by their subscript (G_i) or some other symbol (G', G^*) the same notation will be used to distinguish their vertex and edge sets.

Some special graphs, relevant for the description of graph kernels are walks, paths, and cycles. A *walk*[3] w is a sequence of vertices $v_i \in \mathcal{V}$ and edges $e_i \in \mathcal{E}$ with $w = v_1, e_1, v_2, e_2, \ldots e_n, v_{n+1}$ and $\Psi(e_i) = (v_i, v_{i+1})$. The *length* of the walk is equal to the number of edges in this sequence, i.e., n in the above case. A *path* is a walk in which $v_i \neq v_j \Leftrightarrow i \neq j$ and $e_i \neq e_j \Leftrightarrow i \neq j$. A *cycle* is a path followed by an edge e_{n+1} with $\Psi(e_{n+1}) = (v_{n+1}, v_1)$. Figure 2 gives an illustration of a walk, a path and a cycle.

4.2. Graph degree and adjacency matrix

We also need to define some functions describing the neighborhood of a vertex v in a graph G: $\delta^+(v) = \{e \in \mathcal{E} \mid \Psi(e) = (v, u)\}$ and $\delta^-(v) = \{e \in \mathcal{E} \mid \Psi(e) = (u, v)\}$. Here, $|\delta^+(v)|$ is called the *outdegree* of a vertex and $|\delta^-(v)|$ the *indegree*. Furthermore, the maximal indegree

[2] While ℓ_1 will be used to always denote the same label, l_1 is a variable that can take different values, e.g., $\ell_1, \ell_2, \ldots$. The same holds for vertex v_1 and variable v_1.

[3] What we call 'walk' is sometimes called an 'edge progression'.

and outdegree are denoted by $\Delta^-(G) = max\{|\delta^-(v)| : v \in \mathcal{V}\}$ and $\Delta^+(G) = max\{|\delta^+(v)| : v \in \mathcal{V}\}$, respectively.

For a compact representation of the graph kernel we will later use the adjacency matrix E of a graph. The component $[E]_{ij}$ of this matrix corresponds to the number of edges from vertex v_i to v_j. Replacing the adjacency matrix E by its n-th power ($n \in \mathbb{N}, n \geq 0$), the interpretation is quite similar. Each component $[E^n]_{ij}$ of this matrix gives the number of walks of length n from vertex v_i to v_j.

It is clear that the maximal indegree equals the maximal column sum of the adjacency matrix and that the maximal outdegree equals the maximal row sum of the adjacency matrix. It is important to note that for undirected graphs $\Delta^+(G) = \Delta^-(G)$ is greater than the largest eigenvalue of the adjacency matrix of G.

4.3. Product graph kernels

In this section we briefly review one of the graph kernels defined in Gärtner et al. (2003b). Technically, this kernel is based on the idea of counting the number of walks in product graphs. Note that the definitions given here are more complicated than those given in Gärtner et al. (2003b) as parallel edges have to be considered here.

Product graphs (Imrich & Klavžar, 2000) are a very interesting tool in discrete mathematics. The four most important graph products are the Cartesian, the strong, the direct, and the lexicographic product. While the most fundamental one is the Cartesian product, in our context the direct graph product is the most important one. However, we need to extend its definition to labeled directed graphs. For that we need to define a function $match(l_1, l_2)$ that is 'true' if the labels l_1 and l_2 'match'. In the simplest case $match(l_1, l_2) \Leftrightarrow l_1 = l_2$. Now we can define the direct product of two graphs as follows.

Definition 4.1. We denote the direct product of two graphs $G_1 = (\mathcal{V}_1, \mathcal{E}_1, \Psi_1), G_2 = (\mathcal{V}_2, \mathcal{E}_2, \Psi_2)$ by $G_1 \times G_2$. The vertex set of the direct product is defined as:

$$\mathcal{V}(G_1 \times G_2) = \{(v_1, v_2) \in \mathcal{V}_1 \times \mathcal{V}_2 : match(label(v_1), label(v_2))\} .$$

The edge set is then defined as:

$$\begin{aligned}\mathcal{E}(G_1 \times G_2) = \{(e_1, e_2) \in \mathcal{E}_1 \times \mathcal{E}_2 \ : \exists\, (u_1, u_2), (v_1, v_2) \in \mathcal{V}(G_1 \times G_2) \\ \Psi_1(e_1) = (u_1, v_1) \wedge \Psi_2(e_2) = (u_2, v_2) \\ \wedge\, match(label(e_1), label(e_2))\} .\end{aligned}$$

Given an edge $(e_1, e_2) \in \mathcal{E}(G_1 \times G_2)$ with $\Psi_1(e_1) = (u_1, v_1)$ and $\Psi_2(e_2) = (u_2, v_2)$ the value of $\Psi_{G_1 \times G_2}$ is:

$$\Psi_{G_1 \times G_2}((e_1, e_2)) = ((u_1, u_2), (v_1, v_2)) .$$

The labels of the vertices and edges in graph $G_1 \times G_2$ correspond to the labels in the factors. The graphs G_1, G_2 are called the factors of graph $G_1 \times G_2$.

Figure 3 shows two directed labeled graphs and their direct product. The edges are presumed to be unlabeled.

Having introduced product graphs, we are now able to define the product graph kernel.

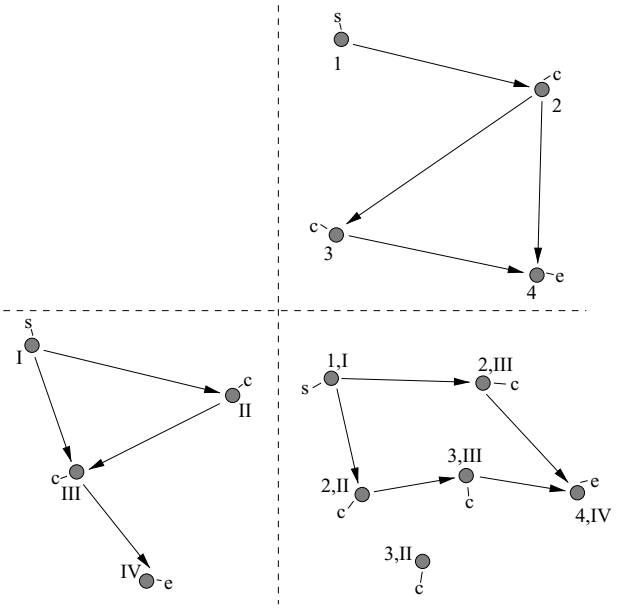

Fig. 3 Two labeled directed graphs and their direct product graph at the bottom right. The numbers 1,2,... and Roman numerals I,II,... and their combinations on the product graph are identifiers for the vertices. (For simplicity reasons, no labels are used on the edges in this example)

Definition 4.2. Let G_1, G_2 be two graphs, let $E_\times$ denote the adjacency matrix of their direct product $G_1 \times G_2$, and let $\mathcal{V}_\times$ denote the vertex set of the direct product $G_1 \times G_2$. With a sequence of weights $\lambda = \lambda_0, \lambda_1, \ldots$ ($\lambda_i \in \mathbb{R}; \lambda_i \geq 0$ for all $i \in \mathbb{N}$) the product graph kernel is defined as

$$k_\times(G_1, G_2) = \sum_{i,j=1}^{|\mathcal{V}_\times|} \left[\sum_{n=0}^{\infty} \lambda_n E_\times^n \right]_{ij}$$

if the limit exists.

For the proof that this kernel is positive definite (see Gärtner et al. 2003b).[4] There it is shown that this product graph kernel corresponds to the inner product in a feature space made up by all possible contiguous label sequences in the graph. Each feature value corresponds to the number of walks with such a label sequence, weighted by $\sqrt{\lambda_n}$ where n is the length of the sequence.

4.4. Computing graph kernels

To compute this graph kernel, it is necessary to compute the limit of the above matrix power series. In this section we show how limits of matrix power series of the form $\lim_{n\to\infty} \sum_{i=0}^{n} \lambda_i E^i$

[4] The proof in there is for graphs without parallel edges. The proof for graphs with parallel edges follows along the same lines.

can be computed for symmetric matrices (i.e., undirected graphs), by means of the limits of eigenvalues of E.

If the matrix E can be diagonalized such that $E = T^{-1}DT$ with a diagonal matrix D and a regular matrix T then arbitrary powers of the matrix E can easily be computed as $E^n = (T^{-1}DT)^n = T^{-1}D^nT$ and for the diagonal matrix D arbitrary powers can be computed component-wise. These results carry over to the limit and

$$\lim_{n\to\infty}\sum_{i=0}^{n}\lambda_i E^i = T^{-1}\left(\lim_{n\to\infty}\sum_{i=0}^{n}\lambda_i D^i\right)T$$

with

$$\left[\lim_{n\to\infty}\sum_{i=0}^{n}\lambda_i D^i\right]_{jj} = \lim_{n\to\infty}\sum_{i=0}^{n}\lambda_i [D_{jj}]^i$$

if E can be diagonalized as above. We note that such a decomposition is always possible if E is symmetric. Matrix diagonalization is then a matrix eigenvalue problem and such methods have roughly cubic time complexity.

Two important special cases are now considered.

Exponential series. Similar to the exponential of a scalar value ($e^b = 1 + b/1! + b^2/2! + b^3/3! + \cdots$) the exponential of the square matrix E is defined as

$$e^{\beta E} = \lim_{n\to\infty}\sum_{i=0}^{n}\frac{(\beta E)^i}{i!}$$

where we use $\frac{\beta^0}{0!} = 1$ and $E^0 = \mathbf{I}$.

Once the matrix is diagonalized, computing the exponential of D can be done in linear time and all that remains is to compute $T^{-1}\exp(D)T$.

Geometric series. The geometric series $\sum_i \gamma^i$ is known to converge if and only if $|\gamma| < 1$. In this case the limit is given by $\lim_{n\to\infty}\sum_{i=0}^{n}\gamma^i = \frac{1}{1-\gamma}$. Similarly, we define the geometric series of a matrix as

$$\lim_{n\to\infty}\sum_{i=0}^{n}\gamma^i E^i \ .$$

If E is symmetric and γ is smaller than one over the largest eigenvalue of E, feasible computation of the limit of a geometric series is possible by inverting the matrix $\mathbf{I} - \gamma E$. To see this, let $(\mathbf{I} - \gamma E)x = 0$, thus $\gamma Ex = x$ and $(\gamma E)^i x = x$. Now, note that $(\gamma E)^i \to 0$ as $i \to \infty$. Therefore $x = 0$ and $\mathbf{I} - \gamma E$ is regular. Then $(\mathbf{I} - \gamma E)(\mathbf{I} + \gamma E + \gamma^2 E^2 + \cdots) = \mathbf{I}$ and $(\mathbf{I} - \gamma E)^{-1} = (\mathbf{I} + \gamma E + \gamma^2 E^2 + \cdots)$ is obvious. Like matrix diagonalization, matrix inversion is roughly of cubic time complexity.

5. Gaussian processes

5.1. Regression using Gaussian processes

Gaussian processes implement a non-parametric Bayesian technique. Instead of assuming a prior over the parameter vectors, a prior is assumed over the target function itself. In this section, we follow the description given in MacKay (1997a).

Assume that a set of data points $\{(x_i, t_i)\}_{i=1}^{N}$ is observed, with x_i the description of the example and t_i the target value. Bayesian approaches aim at modeling the distribution of t_{N+1} given the example description x_{N+1}, i.e.:

$$P(t_{N+1}|(x_1, t_1), \ldots, (x_N, t_N), x_{N+1}) .$$

Gaussian processes (Gibbs, 1997) assume that the observed target values $\mathbf{t}_N = [t_1, \ldots, t_N]^\top$ have a joint Gaussian distribution and are obtained from the true target values by additive Gaussian noise:

$$P(\mathbf{t}_N|x_1, \ldots, x_N, C_N) = \frac{1}{Z}\exp\left(-\frac{1}{2}(\mathbf{t}_N - \mu)^T C_N^{-1}(\mathbf{t}_N - \mu)\right) \tag{2}$$

where μ is the mean vector of the target values, C_N is a covariance matrix ($C_{ij} = C(x_i, x_j)$, $1 \leq i, j \leq N$) and Z is an appropriate normalization constant. The choice of covariance functions is restricted to positive definite kernel functions (see Section 3). For simplicity reasons, we assume the mean vector $\mu = \mathbf{0}$. Although this may seem as a leap of faith, assuming 0 as an a-priori Q-value is standard practice in Q-learning. This assumption was also used in the case of the TG and RIB algorithms, albeit less explicitly.

The distribution of t_{N+1} can be written as the conditional distribution:

$$P(t_{N+1}|x_1, \ldots, x_N, \mathbf{t}_N, x_{N+1}) = \frac{P(\mathbf{t}_{N+1}|x_1, \ldots, x_N, x_{N+1})}{P(\mathbf{t}_N|x_1, \ldots, x_N)}$$

and, using Eq. (2), as the following Gaussian distribution:

$$P(t_{N+1}|x_1, \ldots, x_N, \mathbf{t}_N, x_{N+1}, C_{N+1}) = \frac{Z_N}{Z_{N+1}}\exp\left[-\frac{1}{2}\left(\mathbf{t}_{N+1}^T C_{N+1}^{-1}\mathbf{t}_{N+1} - \mathbf{t}_N^T C_N^{-1}\mathbf{t}_N\right)\right] \tag{3}$$

where Z_N and Z_{N+1} are appropriate normalizing constants and C_N and C_{N+1} are square matrices of dimension N and $N + 1$, respectively, that are related as follows:

$$C_{N+1} = \begin{bmatrix} C_N & \mathbf{k}_{N+1} \\ \mathbf{k}_{N+1}^T & \kappa \end{bmatrix} .$$

The vector $\mathbf{k}_{N+1}$ and scalar κ are defined as:

$$\mathbf{k}_{N+1} = [C(x_1, x_{N+1}), \ldots, C(x_n, x_{N+1})]$$
$$\kappa = C(x_{N+1}, x_{N+1}) .$$

Equation (3) can be rewritten as:

$$P(t_{N+1}|x_1,\ldots,x_N,\mathbf{t}_N,x_{N+1},C_{N+1}) = \frac{1}{Z}\exp\left(-\frac{(t_{N+1}-\hat{t}_{N+1})^2}{2\sigma^2_{\hat{t}_{N+1}}}\right)$$

with

$$\hat{t}_{N+1} = \mathbf{k}^T_{N+1}C_N^{-1}\mathbf{t}_N$$
$$\sigma^2_{\hat{t}_{N+1}} = \kappa - \mathbf{k}^T_{N+1}C_N^{-1}\mathbf{k}_{N+1}$$

and $\mathbf{k}_{N+1}$ and κ as previously defined. This expression maximizes at $\hat{t}_{N+1}$, and $\sigma_{\hat{t}_{N+1}}$ is the standard deviation of the predicted value. Note that, to make predictions, C_N^{-1} is used, so there is no need to invert the new matrix C_{N+1} for each prediction.

Gaussian process regression has some properties that makes it very well suited for reinforcement learning. Firstly, the probability distribution over the target values can be used to guide the exploration of the state-space during the learning process comparable to interval based exploration (Kaelbling et al., 1996). Secondly, the inverse of the covariance matrix can be computed incrementally, using the partitioned inverse equations of Barnett (1979):

$$C^{-1}_{N+1} = \begin{bmatrix} M & \mathbf{m} \\ \mathbf{m}^T & \mu \end{bmatrix}$$

with

$$M = C_N^{-1} + \frac{1}{\mu}\mathbf{k}_{N+1}\mathbf{k}^T_{N+1}$$
$$\mathbf{m} = -\mu C_N^{-1}\mathbf{k}_{N+1}$$
$$\mu = \left(\kappa - \mathbf{k}^T_{N+1}C_N^{-1}\mathbf{k}_{N+1}\right)^{-1}.$$

While matrix inversion is of cubic complexity, computing the inverse of a matrix incrementally after expansion is only of quadratic time complexity. As stated before, no additional inversions need to be performed to make multiple predictions.

5.2. Handling low-dimensional subspaces

5.2.1. Causes for numeric instability

The method discussed in the previous section was used in previous work on Q-learning and uses matrix inversion. However, this assumes that matrix inversion is possible, which is not always guaranteed. In fact, when an example x_l is mapped into feature space onto a linear combination of other examples, i.e.

$$\phi(x_l) = \sum_{i\neq l}\alpha_i\phi(x_i)$$

the row ($C_{l,j}$, $1 \leq j \leq N$) of the covariance matrix will be a linear combination of the other rows, i.e.

$$C_{l,j} = k(x_l, x_j) = k\left(\sum_{i \neq l} \alpha_i \phi(x_i), x_j\right) = \sum_{i \neq l} \alpha_i k(x_i, x_j) = \sum_{i \neq l} \alpha_i C_{i,j} \ .$$

In that case, the covariance matrix will be not of full rank and it will not be possible to invert it. Even if the matrix is of full rank, it might be undesirable to invert it because that may cause numeric stability problems. This kind of problems occurs e.g. when the matrix can be approximated closely by a matrix of lower rank.

Though this problem might be not so important in the usual setting for kernel methods, it occurs naturally in reinforcement learning. One possible cause is that ϕ maps the examples into a low dimensional feature space. More important however is that even when using a ϕ which maps the examples into a high-dimensional feature space (which is where the idea of kernel methods is especially useful), in the application of reinforcement learning, it is very common to revisit the same state multiple times. Revisiting the same state may be necessary in order to reach convergence or it may be a property of the environment (e.g. when each episode starts in the same state or when the goal is to always reach the same state).

A common solution is to observe that the covariance matrix is positive semi-definite (which follows from the fact that its components are values of the kernel k). Hence one can make it full rank by adding a multiple of the identity matrix to it. This means we use $C + \lambda I$ in the computations instead of C. This corresponds to adding a dimension to the feature space for each example and giving that example coordinate $\sqrt{\lambda}$ in this dimension and all other examples coordinate 0. So even if two examples are identical, they will not map to the same point in feature space. This solution was applied in previous applications of the RRL-KBR system, mainly in the blocks world (Gärtner et al., 2003a). Unfortunately, as our experiments show, this is not always sufficient: if λ is large, the matrix $C + \lambda I$ differs too much from the real covariance matrix C, and the resulting predictions will not generalize well. On the other hand, if λ is small, computing $(C + \lambda I)^{-1}$ is numerically still very unstable because the values in the inverted matrix become very high and a small relative error in the computations will cause a large absolute error in the predictions. The fundamental problem is that C might be of too low rank.

So in cases where examples are likely to be not independent in feature space, it is desirable to solve the matrix inversion problem in a more fundamental way. In the next section we will show that a method based on QR-factorization can be used and that this method is in the worst case of the same order of time complexity as the existing method and in the best case an order of magnitude faster. An alternative method which is also well known is to use Cholesky factorization. Although Cholesky factorization can often be made slightly more efficient than QR-factorization, we chose the latter because of its better numeric stability. Other factorizations such as an eigenvalue decomposition might be possible but are unlikely to be computable efficiently.

5.2.2. Avoiding numeric instabilities through QR-factorization

The main idea is to detect when examples are (approximately) linear combinations of other examples in feature space and to handle these examples in such a way that no numerical problems arise.

In a first step we will assign coordinates in the low-dimensional subspace of the feature space spanned by the examples to all examples without explicitly computing ϕ. Formally, we first introduce the space $\mathbb{R}^{\mathcal{X}}$ of functions from the instance space $\mathcal{X}$ to $\mathbb{R}$, which can be seen as the space of all "linear combinations of examples". We can then extend the feature map $\phi : \mathcal{X} \to F$ to the larger space $\mathbb{R}^{\mathcal{X}}$ by defining $\phi' : \mathbb{R}^{\mathcal{X}} \to F$ with

$$\phi'(f) = \sum_{(x,a)\in f} a.\phi(x)$$

and similarly

$$k'(f_1, f_2) = \sum_{(x_1,a_1)\in f_1} \sum_{(x_2,a_2)\in f_2} a_1 a_2 k(x_1, x_2)$$

For any $x \in \mathcal{X}$, let $1_x(x) = 1$ and $1_x(y) = 0, \forall y \neq x$ define the characteristic function 1_x of x. Then, $\phi'(1_x) = \phi(x)$ and $k'(1_x, 1_y) = k(x, y)$. When it is clear from the context, we will write x as argument of ϕ' and k' instead of 1_x. Algorithm 2 computes a set of orthonormal base vectors u_j with $1 \leq j \leq d$. These vectors in $\mathbb{R}^{\mathcal{X}}$ are represented as linear combinations of linearly independent examples x_{b_j} ($1 \leq b_1 \leq \cdots \leq b_d \leq N$):

$$\phi'(u_i) = \sum_{j=1}^{i} u_{i,j} \phi(x_{b_j}) .$$

This allows one to compute $k'(x_i, u_j)$ by computing a linear combination of j kernel values:

$$k'(x_i, u_j) = \sum_{l=1}^{j} u_{j,l} k(x_i, x_{b_l}) .$$

Algorithm 2 also computes the representation of these examples according to these base vectors. Variables $x^*_{i,j}$ that have not been assigned a value are zero.

Let $\mathbf{x}^*_\mathbf{i} = (x^*_{i1} \ldots x^*_{id})^T$ be the representation of example x_i in feature space according to the constructed set of base vectors $\{\phi'(u_1), \ldots, \phi'(u_d)\}$, and let $X_N = (\mathbf{x}^*_\mathbf{1}\, \mathbf{x}^*_\mathbf{2} \ldots \mathbf{x}^*_\mathbf{N})^T$ be a matrix containing the vectors $\mathbf{x}^{*T}_\mathbf{i}$ as rows. We can then write $C_N = XX^T$ and $\mathbf{X_{N+1}} = [X^T \mathbf{x}^*_{\mathbf{N+1}}]^T$ such that Eq. (4) becomes

$$\hat{t}_{N+1} = \mathbf{x}^{*}_{\mathbf{N+1}}{}^T X_N^T \left(X_N X_N^T\right)^{-1} \mathbf{t_N} . \tag{4}$$

Assume now we can write X_N in the form $X_N = Q_N R_N$ where R_N is a square, non-singular upper triangular matrix and Q_N is an orthonormal matrix ($Q_N^T Q_N = I$). Equation (4) then becomes

$$\hat{t}_{N+1} = \mathbf{x}^{*}_{\mathbf{N+1}}{}^T R_N^T Q_N^T \left(Q_N R_N R_N^T Q_N^T\right)^{-1} \mathbf{t_N} .$$

Since $R_N^T Q_N^T (Q_N R_N R_N^T Q_N^T)^{-1} = R^{-1} Q^T$, this reduces to

$$\hat{t}_{N+1} = \mathbf{x}^{*}_{\mathbf{N+1}}{}^T R_N^{-1} Q_N^T \mathbf{t_N} . \tag{5}$$

This equation can be computed in a numerically stable way.

Algorithm 2. Assigning coordinates to the vectors

Require: x_i $(i = 1 \ldots N)$ are training examples
Ensure: $x^*_{i,j}$ are the coordinates assigned to example x_i,
u_i are the new base vectors (linear combinations of x_i)
$d \leftarrow 0$
$B \leftarrow \{\}$
for $i = 1 \ldots N$ **do**
$x^*_{i,j} \leftarrow k'(x_i, u_j)$ for $j = 1 \ldots d$
$x^*_{i,d+1} \leftarrow \left(k(x_i, x_i) - \sum_{j=1}^{d} {x^*_{i,j}}^2\right)^{1/2}$
if $x^*_{i,d+1} \geq \epsilon_d$ **then**
$b_{d+1} \leftarrow i$
$u_{d+1} \leftarrow \phi'^{-1}\left((x_i - \sum_{j=1}^{d} x^*_{i,j}\phi(u_j))/x_{i,d+1}\right)$
$d \leftarrow d + 1$
else
$x^*_{i,d+1} \leftarrow 0$
end if
end for

It is not necessary to recompute the QR-factorization for each new example. Instead, we update the QR-decomposition in a numerically stable way. As explained in detail in Golub & Van Loan (1996), if $X_N = Q_N R_N$ and $X^T_{N+1} = [X^T_N \; \mathbf{x^*_{N+1}}]$, one can determine Q_{N+1} and R_{N+1} such that $X_{N+1} = Q_{N+1}R_{N+1}$ in time $O(Nd)$. This can be seen as follows

$$X_{N+1} = \begin{bmatrix} X_N \\ \mathbf{x^*_{N+1}}^T \end{bmatrix} = \begin{bmatrix} Q_N & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_N & 0 \\ \mathbf{x^*_{N+1}}^T & \end{bmatrix}. \tag{6}$$

To make the extended R-matrix upper-triangular again, (at most) d Givens rotations are required to zero the d first elements of the bottom line of this matrix. The extended Q matrix should be transformed accordingly, i.e. if

$$R_{N+1} = G_1.G_2 \ldots G_d. \begin{bmatrix} R_N & 0 \\ \mathbf{x^*_{N+1}}^T & \end{bmatrix},$$

where G_j are the Givens transformation matrices, then

$$Q_{N+1} = \begin{bmatrix} Q_N & 0 \\ 0 & 1 \end{bmatrix} .G^T_d \ldots . G^T_2 . G^T_1$$

ensures $X_{N+1} = Q_{N+1}R_{N+1}$. Notice that if x^*_{N+1} is not a new base vector, the last column of R_{N+1} will be 0 and since R_{N+1} is upper triangular; so will be its last row, so these together with the last column of Q_{N+1} can be dropped.

While $O(Nd)$ may already be smaller than the $O(N^2)$ time needed to incrementally update the matrix inverse, one can do even better. Indeed, in Eq. (5), we do not need all information

of Q_N, but only $Q_N^T \mathbf{t_N}$. Let $Q_N^T = [\mathbf{q_{N,1}} \ldots \mathbf{q_{N,N}}]$. Then,

$$Q_N^T \mathbf{t_N} = \sum_{i=0}^{N} \mathbf{q_{N,i}} t_i = \mathbf{s_N} .$$

However, summing over these N vectors can be done as soon as possible. Therefore, it suffices to show that

$$\begin{bmatrix} Q_N & 0 \\ 0 & 1 \end{bmatrix} \cdot \prod_{j=d}^{1} G_j^T \mathbf{t_{N+1}} = \begin{bmatrix} \mathbf{s_N} & 0 \\ 0 & 1 \end{bmatrix} \cdot \prod_{j=d}^{1} G_j^T \begin{bmatrix} 1 \\ t_{N+1} \end{bmatrix} .$$

This holds because

$$\begin{aligned} \begin{bmatrix} Q_N & 0 \\ 0 & 1 \end{bmatrix} \cdot \prod_{j=d}^{1} G_j^T \mathbf{t_{N+1}} &= \begin{bmatrix} [\mathbf{q_{N,1}}\ 0] \prod_{j=d}^{1} G_j^T \\ [\mathbf{q_{N,2}}\ 0] \prod_{j=d}^{1} G_j^T \\ \cdots \\ [\mathbf{q_{N,N}}\ 0] \prod_{j=d}^{1} G_j^T \\ [0\ 1] \prod_{j=d}^{1} G_j^T \end{bmatrix} \mathbf{t_{N+1}} \\ &= \begin{bmatrix} \sum_{i=0}^{N} t_i [\mathbf{q_{N,i}}\ 0] \prod_{j=d}^{1} G_j^T \\ [0\ 1] \prod_{j=d}^{1} G_j^T \end{bmatrix} \begin{bmatrix} 1 \\ t_{N+1} \end{bmatrix} \\ &= \begin{bmatrix} \sum_{i=0}^{N} [\mathbf{q_{N,i}} t_i\ 0] \\ [0\ 1] \end{bmatrix} \prod_{j=d}^{1} G_j^T \begin{bmatrix} 1 \\ t_{N+1} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{s_N} & 0 \\ 0 & 1 \end{bmatrix} \cdot \prod_{j=d}^{1} G_j^T \begin{bmatrix} 1 \\ t_{N+1} \end{bmatrix} . \end{aligned}$$

So we only have to store compressed $1 \times d$ matrices Q_N. Notice that since we only need to compute kernel values for new examples with examples that introduced a new dimension (base vector), this reduces the time for the incremental update step to $O(d^2)$.

In summary, we have described a way to update our kernel-based theory avoiding both matrix inversion and changing the data. Note that our method has complexity $O(d^2)$ for each update, which is in worst case of the same order as the previous algorithm but when $d \ll N$, i.e. most examples are (approximately) linearly dependent in feature space, this method will be much faster.

6. Experimental results

6.1. The blocks world

A blocks world consists of a constant number of identical blocks. Each block is put either on the floor or on another block. On top of each block is either another block, or 'no block'. Figure 4 illustrates a (*state*, *action*)-pair in a blocks world with four blocks in two stacks.

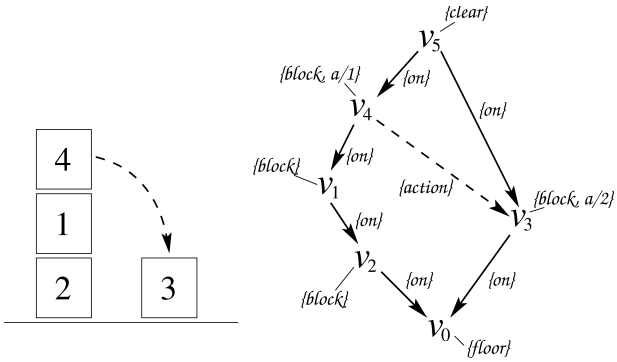

Fig. 4 Simple example of a blocks world state and action (*left*) and its representation as a graph (*right*)

6.1.1. State and action representation

A (*state*, *action*)-pair in the Blocks World can be very naturally represented as a graph. The right side of Fig. 4 shows a graph representation of this blocks world state and action. The vertices of the graph correspond either to a block, the floor, or 'clear'; where 'clear' basically denotes 'no block'. This is reflected in the labels of the vertices. Each edge labeled 'on' (solid arrows) denotes that the block corresponding to its initial vertex is on top of the block corresponding to its terminal vertex. The edge labeled 'action' (dashed arrow) denotes the action of putting the block corresponding to its initial vertex on top of the block corresponding to its terminal vertex; in the example "put block 4 on block 3". The labels 'a/1' and 'a/2' denote the initial and terminal vertex of the action, respectively.

To represent an arbitrary blocks world as a labeled directed graph we proceed as follows. Given the set of blocks numbered $1, \ldots, n$ and the set of stacks $1, \ldots, m$:

1. The vertex set $\mathcal{V}$ of the graph is $\{v_0, \ldots, v_{n+1}\}$.
2. The edge set $\mathcal{E}$ of the graph is $\{e_1, \ldots, e_{n+m+1}\}$.

The node v_0 will be used to represent the floor, v_{n+1} will indicate which blocks are clear. Since each block is on top of something and each stack has one clear block, we need $n + m$ edges to represent the blocks world state. Finally, one extra edge is needed to represent the action. For the representation of a state it remains to define the function Ψ:

3. For $1 \leq i \leq n$, we define $\Psi(e_i) = (v_i, v_0)$ if block i is on the floor, and $\Psi(e_i) = (v_i, v_j)$ if block i is on top of block j.
4. For $n < i \leq n + m$, we define $\Psi(e_i) = (v_{n+1}, v_j)$ if block j is the top block of stack $i - n$.

and the function *label*:

5. We define: $\mathcal{L} = 2^{\{\{\text{'floor'}\},\{\text{'clear'}\},\{\text{'block'}\},\{\text{'on'}\},\{\text{'a/1'}\},\{\text{'a/2'}\}\}}$,

   - $label(v_0) = \{\text{'floor'}\}$,
   - $label(v_{n+1}) = \{\text{'clear'}\}$,
   - $label(v_i) = \{\text{'block'}\}$ $(1 \leq i \leq n)$,
   - $label(e_i) = \{\text{'on'}\}$ $(1 \leq i \leq n + m)$.

All that is left now is to represent the action in the graph

6. We define:

- $\Psi(e_{n+m+1}) = (v_i, v_j)$ if block i is moved to block j.
- $label(v_i) = label(v_i) \cup \{\text{'a/1'}\}$
- $label(v_j) = label(v_j) \cup \{\text{'a/2'}\}$
- $label(e_{n+m+1}) = \{\text{'action'}\}$.

It is clear that this mapping from blocks worlds to graphs is injective.

In some cases the 'goal' of a blocks world problem is to stack blocks in a given configuration (e.g. "put block 3 on top of block 4"). We then need to represent this in the graph. This is handled in the same way as the action representation, i.e. by an extra edge along with extra 'g/1', 'g/2', and 'goal' labels for initial and terminal blocks, and the new edge, respectively. Note that by using more than one 'goal' edge, we can model arbitrary goal configurations, e.g., "put block 3 on top of block 4 and block 2 on top of block 1".

This graph representation will allow the use of the previously defined graph kernel as a covariance function for the Gaussian processes when learning in the Blocks World.

6.1.2. A gaussian kernel version

In finite state-action spaces Q-learning is guaranteed to converge if the mapping between state-action pairs and Q-values is represented explicitly. One advantage of Gaussian processes is that for particular choices of the covariance function, the representation is explicit.

To see this we use the matching kernel k_δ as the covariance function between examples ($k_\delta : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is defined as $k_\delta(x, x') = 1$ if $x = x'$ and $k_\delta(x, x') = 0$ if $x \neq x'$). Let the predicted Q-value be the mean of the distribution over target values, i.e., $\hat{t}_{N+1} = k_{n+1}^\top \mathbf{C}_N^{-1} \mathbf{t}_N$ where the variables are used as defined in Section 5.1. Assume the training examples are distinct and the test example is equal to the j-th training example. It then turns out that $\mathbf{C} = \mathbf{I} = \mathbf{C}^{-1}$ where $\mathbf{I}$ denotes the identity matrix. As furthermore k is then the vector with all components equal to 0 except the j-th which is equal to 1, it is obvious that $\hat{t}_{n+1} = t_j$ and the representation is thus explicit.

A frequently used kernel function for instances that can be represented by vectors is the Gaussian radial basis function kernel (RBF). Given the bandwidth parameter γ the RBF kernel is defined as:

$$k_{\text{rbf}}(x, x') = \exp(-\gamma ||x - x'||^2) .$$

For large enough γ the RBF kernel behaves like the matching kernel. In other words, the parameter γ can be used to regulate the amount of generalization performed in the Gaussian process algorithm: For very large γ all instances are very different and the Q-function is represented explicitly; for small enough γ all examples are considered very similar and the resulting function is very smooth.

In order to have a similar means to regulate the amount of generalization in the blocks world setting, we do not use the graph kernel proposed in Section 4 directly, but 'wrap' it in a Gaussian RBF function. Let k be the graph kernel with exponential weights, then the kernel used in the blocks world is given by

$$k^*(x, x') = \exp[-\gamma(k(x, x) - 2k(x, x') + k(x', x'))] .$$

6.1.3. Experimental results

In this section we describe the tests used to investigate the utility of Gaussian processes and graph kernels as a regression algorithm for RRL.

The RRL-system was trained in an "idealized" training environment, where the number of blocks during experimentation varied between 3 and 5, and RRL was given "guided" traces (Driessens & Džeroski, 2002) in a world with 10 blocks. The Q-function and the related policy were tested at regular intervals on 100 randomly generated starting states in worlds where the number of blocks varied from 3 to 10 blocks.

We evaluated RRL with Gaussian processes on three different goals: stacking all blocks, unstacking all blocks and putting two specific blocks on each other. For each goal we ran five times 1000 episodes with different parameter settings to evaluate their influence on the performance of RRL. After that we chose the best parameter setting and ran another ten times 1000 episodes with different random initializations. For the "stack-goal" only 500 episodes are shown, as nothing interesting happens thereafter. The results obtained by this procedure are then used to compare the algorithm proposed in this paper with previous versions of RRL.

Parameter influence. The used kernel has two parameters that need to be chosen: the exponential weight β (which we shall refer to as *exp* in the graphs) and the radial base function parameter γ (which we shall refer to as *rbf*).

The *exp*-parameter gives an indication of the importance of long walks in the product graph. Higher *exp*-values place means a higher weight for long walks. The *rbf*-parameter gives an indication of the amount of generalization that should be done. Higher *rbf*-values means lower σ-values for the radial base functions and thus less generalization.

We tested the behavior of RRL with Gaussian processes on the "stack-goal" with a range of different values for the two parameters. The experiments were all repeated five times with different random seeds. The results are summarized in Fig. 5. The graph on the left shows that for small *exp*-values RRL can not learn the task of stacking all blocks. This makes sense, since we are trying to create a blocks-world-graph which has the longest walk possible, given a certain amount of blocks. However, for very large values of *exp* we have to use equally small values of *rbf* to avoid numeric overflows in our calculations, which in turn results in non-optimal behavior. The right side of Fig. 5 shows the influence of the *rbf*-parameter. As expected, smaller values result in faster learning, but when choosing too small *rbf*-values, RRL can not learn the correct Q-function and does not learn an optimal strategy.

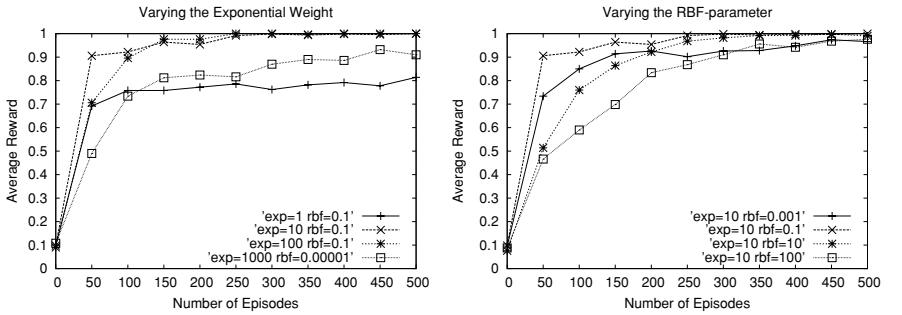

Fig. 5 Comparing parameter influences for the stack goal

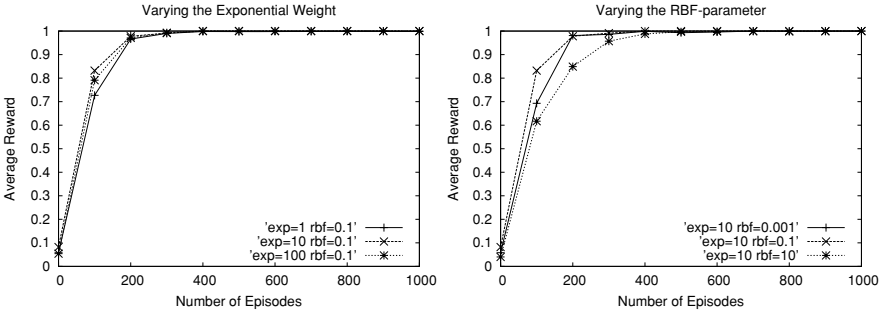

Fig. 6 Comparing parameter influences for the unstack goal

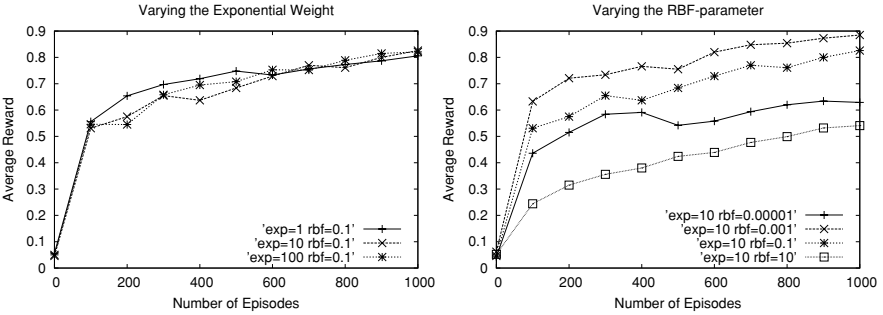

Fig. 7 Comparing parameter influences for the on(A,B) goal

For the "unstack-" and "on(A, B)-goal", the influence of the *exp*-parameter is smaller as shown in the left sides of Figs. 6 and 7 respectively. For the "unstack-goal" there is even little influence from the *rbf*-parameter as shown in the right side of Fig. 6 although it seems that average values work best here as well.

The results for the "on(A,B)-goal" however, show a large influence of the *rbf*-parameter (right side of Fig. 7). In previous work we have always noticed that "on(A,B)" is a hard problem for RRL to solve (Driessens et al., 2001; Driessens & Džeroski, 2002). The results we obtained with RRL-KBR give an indication why. The learning-curves show that the performance of the resulting policy is very sensitive to the amount of generalization that is used. The performance of RRL drops rapidly as a result of over- or under-generalization.

Figure 8 shows the effect of replacing Tikhonov regularization by QR-factorization for matrix inversion. One can see that the effect is positive though not as large as in the Tetris application in the next section. Still, the computation time for the experiment is smaller with QR-factorization (about one fifth).

Comparison with previous RRL-implementations. Figure 9 shows the results of RRL-KBR on the three blocks world problems in relation to the two previous implementations of RRL, i.e. RRL-TG and RRL-RIB. For each goal we chose the best parameter settings from the experiments described above and ran another ten times 1000 episodes. These ten runs were initialized with different random seeds than the experiments used to choose the parameters.

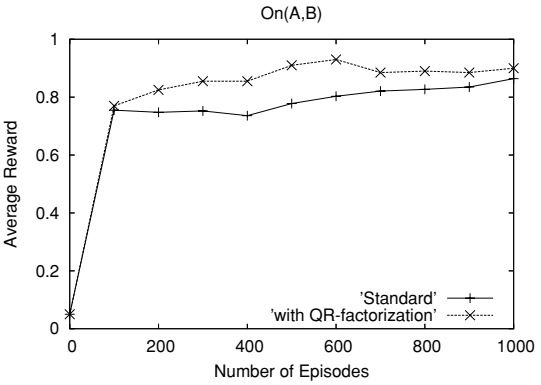

Fig. 8 The effect of using QR-factorization. 'Standard' is the method using the incremental inverse

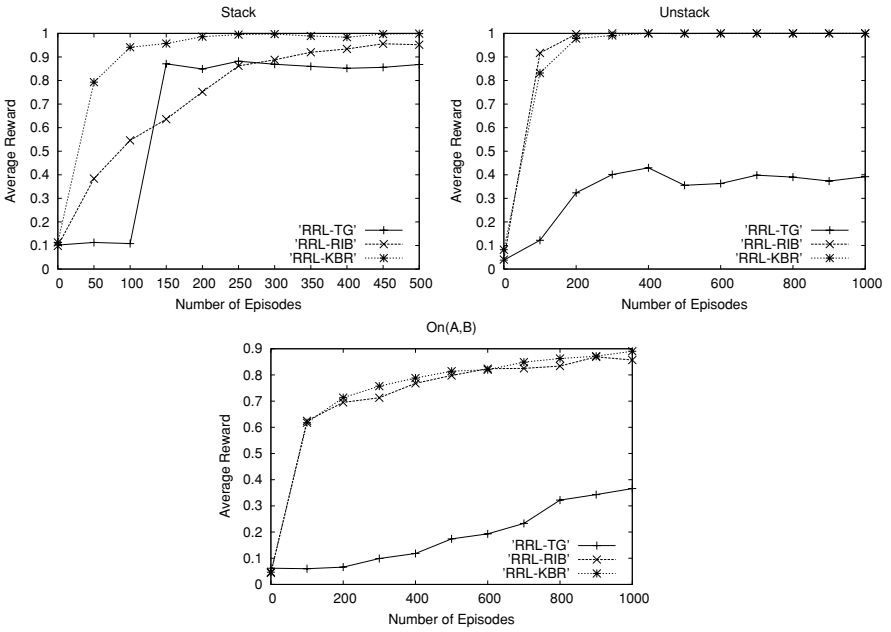

Fig. 9 Comparing kernel based RRL with previous versions

RRL-KBR clearly outperforms RRL-TG with respect to the number of episodes needed to reach a certain level of performance. Note that the comparison as given in Fig. 9 is not entirely fair with RRL-TG. Although RRL-TG does need a lot more episodes to reach a given level of performance, it processes these episodes much faster. This advantage is, however, lost when acting in expensive or slow environments.

RRL-KBR performs better than RRL-RIB on the "stack-goal" and obtains comparable results on the "unstack-goal" and on the "on(A,B)-goal". Our current implementation of RRL-KBR is competitive with RRL-RIB in computation times and performance. However, a big advantage of RRL-KBR is the possibility to achieve further improvements with fairly simple modifications, as we will outline in the next section.

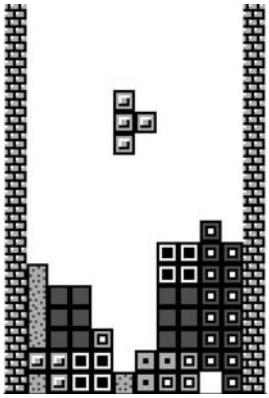

Fig. 10 A snapshot of the Tetris game

6.2. The Tetris game

Tetris[5] is probably one of the most famous computer games around. Designed by Alexey Pazhitnov in 1985 it has been ported to almost any platform available, including most consoles. The Tetris is a puzzle-video game played on a two-dimensional grid as shown in Fig. 10. Different shaped blocks fall from the top of the game field and fill up the grid. The object of the game is to score points while keeping the blocks from piling up to the top of the game field. To do this, one can move the dropping blocks right and left or rotate them as they fall. When one horizontal row is completely filled, that line disappears and the player scores a point. When the blocks pile up to the top of the game field, the game ends. The fallen blocks on the playing field will be referred to as the wall.

A playing strategy for the Tetris game consist of two parts. Given the shape of the dropping block, one has to decide on the optimal orientation and location of the block in the game field. This can be seen as the strategic part of the game and deals with the uncertainty about the shape of the blocks that will follow the present one. The other part consists of using the low level actions—*turn, moveLeft, moveRight, drop*—to reach this optimal placement. This part is completely deterministic and can be viewed as a rather simple planning problem.

We will only consider the task of deciding the placement of the next block. Finding the optimal placement of a given series of falling blocks is an NP-complete problem (Demaine et al., 2002). Although the optimal placement is not required to play a good game of Tetris, the added difficulty of dealing with an unknown series of blocks makes it quite challenging for reinforcement learning, and Q-learning in particular.

There exist very good artificial Tetris players, most of which are hand built. The best of these algorithms score about 500.000 lines on average when they only include information about the falling block and more than 5 million lines when the next block is also considered. The results discussed here will be nowhere near this high and will be even low for human standards. The stochastic nature of the game (i.e. the unknown shape of the next falling block) in combination with the chaotic nature of the Tetris dynamics make it very hard to connect a Q-value to a given state-action pair. It is very hard to predict the future rewards starting from a given Tetris state. The state that can quickly lead to a reward for a given block, can be completely unsuited to deal with other blocks. Also, for a given state, two different actions can lead to completely different resulting states (for example, creating a deep canyon in one

[5] Tetris is owned by The Tetris Company and Blue Planet Software.

case and deleting 3 lines in the other). However, the experiments shown will illustrate the behavior of the Gaussian processes regression and the gain caused by the QR-factorization as discussed in Section 5.2.

One feature that is of specific importance for this work, is that the begin-state for each learning episode is identical (except for maybe the falling block). Still, all states during the first few turns of the Tetris game are very similar in subsequent games. This will cause these states to be visited quite frequently.

6.2.1. Representation and kernel function

For the Tetris application, we do not learn a Q-function that maps state-action pairs to real numbers, but a function that maps the afterstates (states reached after performing the action in the current states but without the next-block info) onto real numbers. We can do that as the afterstates can be computed deterministically from the current state and the action. We expect that this setting is much easier for the learning algorithm. The Q-function can then be computed be adding the reward obtained during the current action to the value of the next state.

We use a kernel which is the inner product of two feature vectors. The features we use are the height of the columns, the maximal height of all columns; the number of holes and the average depth of the holes. This kernel does not use relational features of the Tetris game. Although these can be added, they greatly increase the computational complexity of the learning problem without altering the eventual outcomes of the experiments. To illustrate the effect of repeated visitations to similar states and show the effect of QR-factorization, we decided to use a simpler kernel that would allow us to run more experiments that confirmed the results.

For comparison to two other RRL regression algorithms RRL-RIB and RRL-TG we provided those algorithms with the following Tetris representations:

1. RRL-RIB was given a Euclidian distance using a number of state features, thus basically also working with a propositional representation of the Tetris states. These features were:
   - The maximum, average and minimum height of the wall and the differences between the extremes and the average.
   - The height differences between adjacent columns.
   - The number of holes and canyons of width 1.
2. For RRL-TG the use of afterstates did not result in a better policy, so we used the standard approach of using state-action pairs as learning examples, using the following predicates to build the tree:
   - *blockwidth/2, blockheight/2*: which specify the width and height of the falling block respectively.
   - *topBlock/2*: which returns the height of the wall at a given column.
   - *holeCovered/1*: whether there is a hole in the wall at a given column.
   - *holeDepth/2*: which returns the depth of the topmost hole in the wall at a given column.
   - *fits/2*: whether the falling block fits at a given location with a given orientation.
   - *increasesHeight/2*: whether dropping the falling block at a given location with a given orientation increases the overall height of the wall.
   - *fillsRow/2* and *fillsDouble/2*: whether the falling block completes a line (or two lines) at a given location with a given orientation.

On top of these, RRL was given a number of selectors to select individual rows, columns and different block-shapes.

6.2.2. *Experiments*

In our experiments, the learner gets a reward of 1 point for each line deleted. When the game ends, the learner gets a negative reward of -5 points. This helps the learner in trying to postpone the end of the game in the hope of scoring more points. On the learning graphs however, we show the average number of lines deleted per game, without the penalty at the end of the game.

Experiments using the standard matrix inversion update in the regression algorithm show a small peak in performance in the few first episodes, but the performance of the theory decreases very fast after that.

Figure 11 (top) illustrates this behavior for different values of λ in

$$\hat{t}_{N+1} = \mathbf{k}_{N+1}^T (C_N + \lambda I_N)^{-1} \mathbf{t}_N \ .$$

The bottom graph of Fig. 11 gives the learning curve using the QR-factorization method for updating the theory. All runs were limited to 2000 seconds. One can see that the QR-method is faster (approximately 200 times faster), due to the low dimension of the feature space, and that the performance does not fall back by numerical instability after some time. While the

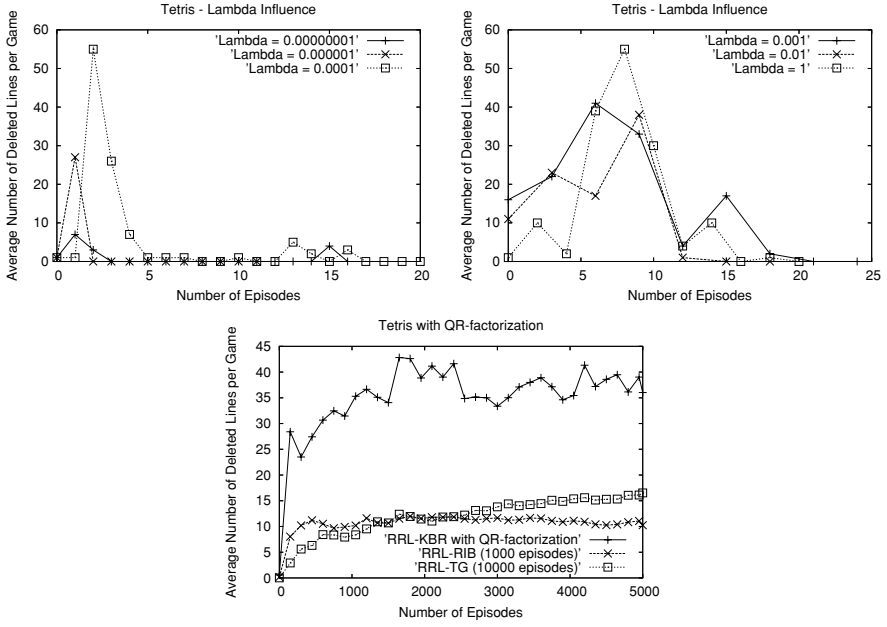

Fig. 11 On top, the learning curves for Tetris for several values of λ using the matrix inversion method. On the bottom, the learning curve using QR-factorization with a comparison to other RRL algorithms. Note that the number of learning episodes shown on the X-axis should be divided by 5 for RRL-RIB and multiplied by 2 for RRL-TG

overall performance of our system on the Tetris domain is not very good, it is better than that of other RRL algorithms which were tried on it in the past. Note that the 2000 seconds runtime was not used for the RRL-TG and RRL-RIB experiments. However, due to the relative computational efficiency of the different techniques, the graph should be read with care as the RRL-RIB experiments only ran to 1000 learning episodes and the RRL-TG experiments to 10 000 learning episodes. To make comparison between the different systems easier, we have mapped all results to the same X-axis width, not the same scale.

7. Related work

Since we presented related work on graph kernels in Section 4 already, we will limit the discussion in this section to related work on kernel based approaches to reinforcement learning. A discussion of other kernels for structured data (less relevant to the work presented here) can be found in Gärtner (2003).

Engel et al. independently developed a method using Gaussian processes for temporal difference learning (Engel et al., 2003) using a regression technique which eliminates almost linearly dependent examples using a technique based on Cholesky factorization. Both Cholesky factorization and QR-factorization have advantages and disadvantages, but have roughly the same time and space complexity. While Cholesky factorization exploits matrix symmetry, QR factorization exploits orthogonality. E.g. when the coordinates according to the orthonormal base vectors of two examples x_1 and x_2 are known, we can efficiently compute the value of $k(x_1, x_2)$. Also, due to the orthogonality of the base, it is easy to control the well-conditionedness of the computations. On the other hand; symmetry can give a factor 2 speedup for part of the computation.

In Rasmussen & Kuss (2004), a method for policy iteration using Gaussian processes is described. In policy iteration, in each iteration the policy is updated (in contrast with Q-learning where the value function or Q function is incrementally updated). The policy is updated by first sampling a value function using the current policy and next adapting the policy the new value function. The authors show that the complexity of each iteration is $O(d^3)$ with d the number of "support points". As the feature space dimension is of the same order as the number of support points, one could say that one complete policy iteration has the same complexity as processing d examples with our Q-learning method ($O(d^2)$ per update). It is however not trivial to apply the same policy iteration method in the relational setting as approximating reward functions by "interpolating" between the value at support points gets problematic as soon as the state space gets more than a few dimensions.

Ormoneit and Sen suggest a kernel based reinforcement learning approach that uses locally weighted averaging to generalize over utilities and use a 'kernel' function as the weight function. However, in Ormoneit & Sen (2002) the term 'kernel' is not used to refer to a positive definite function but to a probability density function.

Dietterich and Wang describe three approaches to combine kernel based methods with reinforcement learning and value function approximation (Dietterich & Wang, 2002). The three approaches consist of support vector machine regression, a formulation based on the Bellman equation and one approach that only tries to represent the fact that good moves have an advantage over bad moves. While all three approaches are model based and are designed to work in batch mode where we update our model incrementally, the SVM regression formulation is related closest to our work.

Dearden et al. describe two techniques that take advantage of the probability distribution supplied by Bayesian regression to guide the exploration in Q-learning (Dearden et al., 1998). Empirical evidence suggests that these exploration strategies perform better than model-free exploration strategies such as Boltzmann exploration (which is currently used in the RRL-system). One approach uses Q-value sampling and the exploration strategy stochastically selects actions according to their probability of being optimal. This type of action selection seems feasible using the probability distributions predicted by the Gaussian processes we use as well.

8. Conclusions and future work

In this paper we proposed Gaussian processes as a new regression algorithm in relational reinforcement learning. Gaussian processes have been chosen as they are able to make probabilistic predictions and can be learned incrementally. To apply Gaussian processes to relational domains, we employed graph kernels as the covariance function. Experiments in the blocks world show comparable and even better performance for RRL using Gaussian processes when compared to previous implementations: decision tree based RRL and instance based RRL.

When examples are heterogeneously spread in feature space, due to constant starting conditions or strong similarities in high performance states, the resulting dependencies between the learning examples cause the standard prediction approach used in Gaussian processes to become numerically unstable. This is due to the problem of inverting an ill-conditioned covariance matrix. This numeric instability causes the RRL system to first learn a good policy but "forget" it again when it collects more learning experience. By using a QR-factorization and thereby representing all the learning examples with respect to an orthonormal base in feature space, we can avoid the numeric instabilities coherent to matrix inversion without making any changes to the learning examples. Paying attention to using a suitable numerical technique improves experimental results significantly and even leads to a computationally more efficient incremental updating mechanism. Experiments with the Tetris game show that better and more stable results can be obtained by using this technique.

As very expressive graph kernels can usually not be computed in polynomial time, we resorted to a variant of walk-based graph kernels (Gärtner et al., 2003b). Clearly, there are other graph kernels that can be computed efficiently and we will explore their applicability to RRL in future work. In this work, we used these graph kernels as the covariance function of a Gaussian Process. We will compare this approach to other kernel-based regression algorithms in future work.

A promising direction for future work is also to exploit the probabilistic predictions made available in RRL by the algorithm suggested in this paper. The obvious use of these probabilities is to exploit them during exploration. Actions or even entire state-space regions with low confidence on their Q-value predictions could be given a higher exploration priority. This approach is similar to interval based exploration techniques (Kaelbling et al., 1996) where the upper bound of an estimation interval is used to guide the exploration into high promising regions of the state-action space. In the case of RRL-KBR these upper bounds could be replaced with the upper bound of a 90% or 95% confidence interval.

Future work will also investigate how reinforcement techniques such as local linear models (Schaal et al., 2000) and the use of convex hulls to make safe predictions (Smart & Kaelbling, 2000) can be applied in RRL.

Many interesting reinforcement learning problems apart from the blocks world also have an inherently structural nature. To apply Gaussian processes and graph kernels to these problems, the state-action pairs just need to be represented by graphs. Future work will explore such applications.

In our empirical evaluation, the algorithm presented in this paper proved competitive or better than the previous implementations of RRL. From our point of view, however, this is not the biggest advantage of using graph kernels and Gaussian processes in RRL. The biggest advantages are the elegance and potential of our approach. The generalization ability can be tuned by a single parameter. Probabilistic predictions can be used to guide exploration of the state-action space.

Acknowledgments Kurt Driessens is a post-doctoral research fellow at the University of Waikato. Jan Ramon is a post-doctoral fellow of the Katholieke Universiteit Leuven. Thomas Gärtner is supported in part by the DFG project (WR 40/2-1) *Hybride Methoden und Systemarchitekturen für heterogene Informationsräume*. The authors thank Peter Flach, Tamás Horváth, Stefan Wrobel and Sašo Džeroski for valuable discussions.

References

Barnett, S. (1979). *Matrix methods for engineers and scientists*. MacGraw-Hill.

Cortes, C., Haffner, P., & Mohri, M. (2003). Positive definite rational kernels. In *Proceedings of the 16th Annual Conference on Computational Learning Theory and the 7th Kernel Workshop*.

Dearden, R., Friedman, N., & Russell, S. (1998). Bayesian Q-learning. In *Proceedings of AAAI-98/IAAI-98*, (pp. 761–768).

Demaine, E., Hohenberger, S., & Liben-Nowell, D. (2002). Tetris is hard, even to approximate. Technical Report MIT-LCS-TR-865, Massachussets Institue of Technology, Boston.

Deshpande, M., Kuramochi, M., & Karypis, G. (2002). Automated approaches for classifying structures. In *Proceedings of the 2nd ACM SIGKDD Workshop on Data Mining in Bioinformatics*.

Diestel, R. (2000). *Graph theory*. Springer-Verlag.

Dietterich, T., & Wang, X. (2002). Batch value function approximation via support vectors. In T. G. Dietterich, S. Becker, & Z. Ghahramani (Eds.), *Advances in Neural Information Processing Systems*, vol. 14, Cambridge, MA, The MIT Press.

Driessens, K., & Džeroski, S. (2002). Integrating experimentation and guidance in relational reinforcement learning. In C. Sammut, & A. Hoffmann (Eds.), *Proceedings of the Nineteenth International Conference on Machine Learning* (pp. 115–122). Morgan Kaufmann Publishers, Inc.

Driessens, K., & Džeroski, S. (2004). Integrating guidance into relational reinforcement learning. *Machine Learning*, *57*, 271–304.

Driessens, K., & Ramon, J. (2003). Relational instance based regression for relational reinforcement learning. In *Proceedings of the Twentieth International Conference on Machine Learning* (pp. 123–130). AAAI Press.

Driessens, K., Ramon, J., & Blockeel, H. (2001). Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In L. De Raedt, & P. Flach (Eds.), *Proceedings of the 13th European Conference on Machine Learning*, vol. 2167 of *Lecture Notes in Artificial Intelligence* (pp. 97–108). Springer-Verlag.

Džeroski, S., De Raedt, L., & Blockeel, H. (1998). Relational reinforcement Learning. In *Proceedings of the 15th International Conference on Machine Learning* (pp. 136–143). Morgan Kaufmann.

Engel, Y., Mannor, S., & Meir, R. (2003). Bayes meets Bellman: The gaussian process approach to temporal difference learning. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML 2003)* (pp. 154–161). Morgan Kaufmann.

Gärtner, T. (2002). Exponential and geometric kernels for graphs. In *NIPS Workshop on Unreal Data: Principles of Modeling Nonvectorial Data*.

Gärtner, T. (2003). A survey of kernels for structured data. *SIGKDD Explorations*, *5*(1), 49–58.

Gärtner, T., Driessens, K., & Ramon, J. (2003a). Graph kernels and Gaussian processes for relational reinforcement learning. In *Inductive Logic Programming, 13th International Conference, ILP 2003, Proceedings*, vol. 2835 of *Lecture Notes in Computer Science* (pp. 146–163). Springer.

Gärtner, T., Flach, P., & Wrobel, S. (2003b). On graph kernels: Hardness results and efficient alternatives. In M. W. B. Schölkopf (Ed.), *Proceedings of the 16th Annual Conference on Computational Learning Theory and the 7th Kernel Workshop* (129–143).

Gibbs, M. (1997). Bayesian Gaussian processes for regression and classification. Ph.D. thesis, University of Cambridge.
Golub, G. H., & Van Loan, C. F. (1996). *Matrix computations*. Johns Hopkins Series in the Mathematical Sciences. The Johns Hopkins University Press.
Graepel, T. (2002). PAC-Bayesian pattern classification with kernels. Ph.D. thesis, TU Berlin.
Horvath, T., Gärtner, T., & Wrobel, S. (2004). Cyclic pattern kernels for predictive graph mining. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*.
Imrich, W., & Klavžar, S. (2000). *Product graphs: Structure and recognition*. John Wiley.
Kaelbling, L., Littman, M., & Moore, A. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.
Kashima, H., & Inokuchi, A. (2002). Kernels for graph classification. In *ICDM Workshop on Active Mining*.
Kashima, H., Tsuda, K., & Inokuchi, A. (2003). Marginalized kernels between labeled graphs. In *Proceedings of the 20th International Conference on Machine Learning*.
Korte, B., & Vygen, J. (2002). *Combinatorial optimization: Theory and algorithms*. Springer-Verlag.
Kuramochi, M., & Karypis, G. (2001). Frequent subgraph discovery. In *Proceedings of the IEEE International Conference on Data Mining*.
MacKay, D. (1997a) Introduction to Gaussian processes. Aavailable at http://wol.ra.phy.cam.ac.uk/mackay.
MacKay, D. J. C. (1997b). Introduction to Gaussian processes. Available at http://wol.ra.phy.cam.ac.uk/mackay.
Mitchell, T. (1997). *Machine learning*. McGraw-Hill.
Ormoneit, D., & Sen, S. (2002). Kernel-based reinforcement learning. *Machine Learning*, *49*, 161–178.
Rasmussen, C. E., & Kuss, M. (2004). Gaussian processes in reinforcement learning. In *Advances in Neural Information Processing Systems*, vol. 16. MIT Press.
Rifkin, R. M. (2002). Everything old is new again: A fresh look at historical approaches to machine learning. Ph.D. thesis, MIT.
Saunders, C., Gammerman, A., & Vovk, v. (1998). Ridge regression learning algorithm in dual variables. In *Proceedings of the Fifteenth International Conference on Machine Learning*. Morgan Kaufmann.
Schaal, S., Atkeson, C. G., & Vijayakumar, S. (2000). Real-time robot learning with locally weighted statistical learning. In *Proceedings of the IEEE International Conference on Robotics and Automation* (pp. 288–293). IEEE Press, Piscataway, N.J.
Schölkopf, B., & Smola, A. J. (2002). *Learning with kernels*. MIT Press.
Smart, W. D., & Kaelbling, L. P. (2000). Practical reinforcement learning in continuous spaces. In *Proceedings of the 17th International Conference on Machine Learning* (pp. 903–910). Morgan Kaufmann.
Sutton, R., & Barto, A. (1998). *Reinforcement learning: An introduction*. Cambridge, MA: The MIT Press.
Vapnik, V. (1995). *The nature of statistical learning theory*. Springer-Verlag.
Watkins, C. (1989). Learning from delayed rewards. Ph.D. thesis, King's College, Cambridge.