Efficient Continuous-Time Reinforcement Learning with Adaptive State Graphs

Gerhard Neumann, Michael Pfeiffer, and Wolfgang Maass

Institute for Theoretical Computer Science, Graz University of Technology
A-8010 Graz, Austria
{neumann, pfeiffer, maass}@igi.tugraz.at

Abstract. We present a new reinforcement learning approach for deterministic continuous control problems in environments with unknown, arbitrary reward functions. The difficulty of finding solution trajectories for such problems can be reduced by incorporating limited prior knowledge of the approximative local system dynamics. The presented algorithm builds an adaptive state graph of sample points within the continuous state space. The nodes of the graph are generated by an efficient principled exploration scheme that directs the agent towards promising regions, while maintaining good online performance. Global solution trajectories are formed as combinations of local controllers that connect nodes of the graph, thereby naturally allowing continuous actions and continuous time steps. We demonstrate our approach on various movement planning tasks in continuous domains.

1 Introduction

Finding near-optimal solutions for continuous control problems is of great importance for many research fields. In the weighted region path-planning problem, for example, one needs to find the shortest path to a goal state through regions of varying movement costs. In robotics specific reward functions can be used to enforce obstacle avoidance or stable and energy-efficient movements. Most existing approaches to these problems require either complete knowledge of the underlying system, or are restricted to simple reward functions. In this paper we address the problem of learning high quality continuous-time policies for tasks with arbitrary reward functions and environments that are initially unknown, except for minimal prior knowledge of the local system dynamics.

Reinforcement learning (RL) [1] is an attractive framework for the addressed problems, because it can learn optimal policies through interaction with an unknown environment. For continuous tasks, typical approaches that use parametric value-function approximation suffer from various problems concerning the learning speed, quality, and robustness of the solutions [2]. Several authors have therefore advocated non-parametric techniques [3, 4], where the value function for the continuous problem is only computed on a finite set of sample states. In this case stronger theoretical convergence and performance guarantees apply [3]. Still, few RL algorithms can cope with continuous actions and time steps.

Sampling-based planning methods [5, 6], on the other hand, can efficiently construct continuous policies as combinations of simple *local controllers*, which navigate between sampled points. Local controllers for small regions of the state space are often easily available, and can be seen as minimal prior information about the task's underlying system dynamics. Local controllers do not assume complete knowledge of the environment (e.g. location of obstacles), and are therefore not sufficient to find globally optimal solutions. Instead, a graph is built, consisting of random sample points that are connected by local controllers. A global solution path to the goal is constructed by combining the paths of several local controllers.

Planning techniques are very efficient, but their application is limited to completely known environments. Guestrin and Ormoneit [6], e.g., have used combinations of local controllers for path planning tasks in stochastic environments. Their graph is built from uniform samples over the whole state space, rejecting those that result in collisions. They also assume that a detailed simulation of the environment is available to obtain the costs and success probabilities of every transition. In this paper we address problems in which the exact reward function is unknown, and the agent has no knowledge of the position of obstacles.

We propose an algorithm for efficiently exploring such unknown continuous environments in order to construct sample-based models. The algorithm builds an *adaptive state graph* of sample points that are connected by given local controllers. Feedback from the environment, like reward signals or unexpected transitions, is incorporated online. Efficiently creating adaptive state graphs can be seen as an optimal exploration problem [7]. The objective is to quickly find good paths from the start to the goal region, not necessarily optimizing the online performance. Initial goal-directed exploration creates a sparse set of nodes, which yields solution trajectories that are later improved by refining the sampling in critical regions. Planning with adaptive models combines the advantages of reinforcement learning and planning. We regard our algorithm more as a RL method, in the spirit of model-based RL [1, 8], since the agent learns both its policy and its world model from actual experience.

The adaptive state graph transforms the continuous control problem into a discrete MDP, which can be exactly solved e.g. by dynamic programming [1]. This results in more accurate policies and reduced running time in comparison to parametric function approximation. The obtained policy still uses continuous actions and continuous time steps, leading to smoother and more natural trajectories than in discretized state spaces. In this paper we address primarily deterministic and episodic tasks with known goal regions, but with small modifications these restrictions can be relaxed. Prior knowledge of the goal position, for example, speeds up the learning process, otherwise the agent will uniformly explore the state space. We demonstrate in comparisons of our algorithm to standard RL and planning techniques that fast convergence and accurate solution trajectories can be achieved at the same time.

In the next section we introduce the basic setup of the problem. We show the structure of the algorithm in Section 3 and present the details of the adaptive

state graph construction in Section 4. In Section 5 we evaluate our algorithm on a continuous path finding task and a planar 3-link arm reaching task, before concluding in Section 6.

2 Graph Based Reinforcement Learning

We consider episodic, deterministic control tasks in continuous space and time. The agent's goal is to move from an arbitrary starting state to a fixed goal region, maximizing a reward function, which evaluates the goodness of every action. In the beginning, the agent only knows the locations of the start state and the goal region, and can use local controllers to navigate to a desired target state in its neighborhood.

Let $\mathcal{X}$ define the state space of all possible inputs $x \in \mathcal{X}$ to a controller. We require $\mathcal{X}$ to be a metric space with given metric $d : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}_0^+$. Control outputs $u \in \mathcal{U}$ change the current state x according to the system dynamics $\dot{x} = f(x, u)$. In this paper we assume that only an approximative local model $\hat{f}(x, u)$ is known, which does not capture possible nonlinearities due to obstacles. The objective is to find a control policy $\mu : \mathcal{X} \rightarrow \mathcal{U}$ for the actual system dynamics $f(x, u)$ that returns for every state x a control output $u = \mu(x)$ such that the agent moves from a starting state $x^S \in \mathcal{X}$ to a goal region $X^G \subset \mathcal{X}$ with maximum reward.

Our algorithm builds an adaptive state graph $\mathcal{G} = \langle \mathcal{V}, E \rangle$, where the nodes in $\mathcal{V} = \{x_1, \ldots, x_N\} \subset \mathcal{X}$ form a finite subset of sample points from $\mathcal{X}$. We start with $\mathcal{V}_0 = \{x^S\}, E_0 = \emptyset$ and let the graph grow in subsequent exploration phases. The edges in $E \subseteq \mathcal{V} \times \mathcal{V}$ correspond to connections between points in $\mathcal{V}$ that can be achieved by a given *local controller*. The local controller $a(e)$ for an edge $e = (x_i, x_j)$ tries to steer the system from x_i to x_j, assuming that the system dynamics along the path corresponds to $\hat{f}(x, u)$. If an edge can be traversed with a local controller, it is inserted into E, and $r(e)$, the total reward obtained on the edge is stored. The combination of multiple edges yields globally valid trajectories.

For a given graph $\mathcal{G}$ the actual task is to find an optimal *task policy* π from the starting state x^S to the goal region X^G. We therefore have to find the optimal sequence of edges $\langle e_i \rangle$ in the graph from x^S to X^G such that the sum of rewards $R^\pi := \sum_{i=0}^{n} r(e_i)$ is maximized. The problem is solved by calculating the optimal value function V^π through dynamic programming. This method is guaranteed to converge to an optimal policy [1], based on the knowledge contained in the adaptive state graph.

The quality of the resulting policy depends on the available edges and nodes of the graph, but also on the quality of the local controllers. We assume here that local controllers can compute near-optimal solutions to connect two states in the absence of unforeseen events. Feedback controllers can compensate small stochastic effects, but the presented algorithm in general assumes deterministic dynamics. We restrict ourselves here to rather simple system dynamics, for which controllers are easily available, e.g. straight-line connections in Euclidean spaces.

While the agent is constructing the graph it is following an *exploration policy* π^{exp}, which can be different from the task policy π. π^{exp} does not always take the best known path to the goal, but also traverses to nodes where the creation of additional nodes and edges may lead to better solutions for the actual task. Virtual *exploration edges* to unvisited regions with heuristic *exploration rewards* are therefore inserted into the graph. This creates incentives for the algorithm to explore new regions. Whenever such virtual edges are chosen by the exploration policy, the graph is expanded to include new nodes and edges.

3 Structure of the Algorithm

The adaptive state graph $\mathcal{G}$ is grown from the start state towards the goal region. We use the approximative model $\hat{f}(x, u)$ to generate new potential successor states from existing nodes, and rank them by a heuristic exploration score. An exploration queue $\mathcal{Q}$ stores the most promising candidates for exploration, and the exploration policy, defined via the value function V^{exp} directs the agent towards one of these targets. Since our goal is finding a good policy from the start, not necessarily maximizing the online performance, the selection of the exploration target involves an exploration-exploitation trade-off inherent to all RL methods. Our method uses the information in the graph to efficiently concentrate on relevant regions of the state space. Whenever a new state is visited, it is added as a node into the graph. We also add all possible edges to and from neighboring nodes that can be achieved by local controllers. Initial optimistic estimates for the reward come from the local controller, but are updated when actual experience becomes available.

Algorithm 1 shows a pseudo-code implementation of the basic algorithm. Details of the subroutines are explained in Section 4. Roughly the algorithm can be structured into 3 parts: the first part in lines 5-11 deals with the generation of new exploration nodes and is described in Sections 4.1-4.3. The second part in lines 12-15 first updates the value functions, and then executes the local controller to move to a different node (see Sections 4.4-4.5). In the remaining part (lines 16-26) we incorporate the feedback received from the environment to update the graph (see Section 4.6).

4 Building the Adaptive State Graph

A key for efficient exploration of the state space is the generation of sample states. Previous approaches for sampling-based planning, e.g. [6, 5], have used uniform random sampling of nodes over the whole state space. This requires a large number of nodes, of which many will lie in irrelevant or even unreachable regions of the state space. On the other hand, a high density of nodes in critical regions is needed for fine-tuning of trajectories. The presented algorithm iteratively builds a graph by adding states that are visited during online exploration. It thereby fulfills two objectives: Firstly, the exploration is directed to search towards a goal

Algorithm 1 Graph-based RL

Input: Start x^S, goal region X^G, local controller a

Initialize $\mathcal{V} = \{x^S\}$, $E = \emptyset$, $\mathcal{G} = \langle \mathcal{V}, E \rangle$, $\mathcal{Q} = \emptyset$
repeat (for each episode):
  Initialize $x = x^S$
  repeat (for each step of the episode):
    for i = 1 **to** N_t **do**
      $\tilde{x}_i =$ `sample_new_node`()
      $[\sigma(\tilde{x}_i), \mathrm{var}_\sigma(\tilde{x}_i)] =$ `exploration_score`$(\tilde{x}_i, V)$
      if $\mathrm{var}_\sigma(\tilde{x}_i) > \theta^{\mathrm{exp}}_{\mathrm{min}}$ **then**
        `insert_exploration_node`$(\tilde{x}_i)$
      end if
    end for
    $[V, V^{\mathrm{exp}}, \mathcal{Q}] =$`replan`$(\mathcal{G})$
    Select next edge $e = (x, x')$ stochastically (e.g. ε-greedy) from V^{exp}
    Execute local controller $a(x, x')$
    Receive actual state $\hat{x}'$ and reward r of transition
    if $d(x', \hat{x}') > \delta$ **then** {*different state than predicted was reached* }
      Delete edge (x, x') from $\mathcal{G}$ and insert edge $(x, \hat{x}')$
      Set $x' = \hat{x}'$
    end if
    if x' was previously unvisited **then**
      `insert_new_node`(x')
      `update_edge`(x, x', r)
      $[V, V^{\mathrm{exp}}, \mathcal{Q}] =$`replan`$(\mathcal{G})$
    else
      `update_edge`(x, x', r)
    end if
  until x is terminal

Output: Task policy π, derived from $\mathcal{G}$ and V

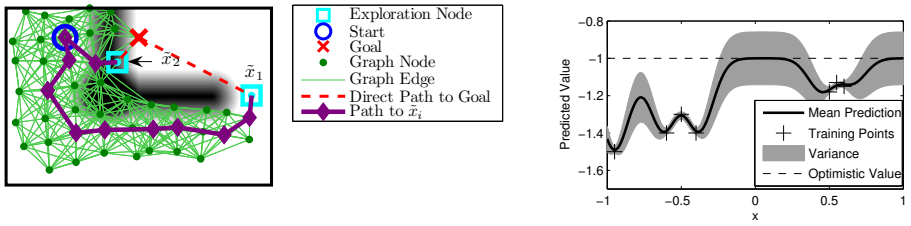
state, and secondly, it optimizes the current policy in regions where the number of nodes is insufficient.

4.1 Generating Samples: `sample_new_node`

Whenever a node x in the graph is visited the algorithm stochastically creates a number of potential *exploration nodes* for that state. New exploration nodes are created uniformly in the neighborhood of the current node. We therefore first uniformly sample an execution time $t_i \in [t_{\min}, t_{\max}]$, and a constant control action u_i in U. Then we simulate the local dynamics $\hat{f}(x, u)$ from x with action u_i for time t_i, and reach a new node $\tilde{x}_i$. For efficiency reasons the number of generated samples N_t should be reduced over time. Similarly the minimum and maximum execution time is reduced over time to create finer sampling and achieve fine-tuning of the policy.

4.2 Evaluating Exploration Nodes: `exploration_score`

Efficient exploration preferentially visits regions where an improvement of the task policy is possible, but avoids creating unnecessary nodes in already densely sampled regions. We estimate the utility of every potential exploration target $\tilde{x}$ by an *exploration score* $\sigma(\tilde{x})$, and direct the agent towards the most promising such nodes. Informed search methods like A* [9] estimate the utility of $\tilde{x}$ as the expected return of a path from the start x^S to the goal region X^G via $\tilde{x}$. This can be decomposed into the path costs $c(x^S, \tilde{x})$ from x^S to $\tilde{x}$ plus the estimated value $\hat{V}(\tilde{x})$, i.e. the estimated rewards-to-goal. Therefore $\sigma(\tilde{x}) = c(x^S, \tilde{x}) + \hat{V}(\tilde{x})$.



(a) (b)

Fig. 1. (a) Illustration of the exploration process. Exploration node $\tilde{x}_1$ is preferred over $\tilde{x}_2$, because the reward to reach $\tilde{x}_2$ is strongly negative. (b) Illustration of value prediction with Gaussian processes on an artificial 1-D dataset. The prediction approaches the optimistic value and has larger variance for points that are farther away from training points.

For calculating the path costs $c(x^S, \tilde{x})$ we use only visited edges of the state graph. Otherwise the optimistic initialization of edge rewards will almost always lead to an underestimation of the path costs, and therefore all exploration nodes will appear similarly attractive. $\hat{V}(\tilde{x})$ must be an optimistic estimate of the value, e.g. the estimated costs of the direct path to the goal in the absence of obstacles. If the goal is not known, a constant value must be used. This prior estimate for the value of a new node $\tilde{x}$ can be improved by considering also the task-policy values $V(x')$ of nearby existing nodes x'. Gaussian process regression [10] is a suitable method to update predictions of a prior function by taking information from a finite set of training examples into account, thereby creating a more-exact posterior. The contributions of individual training examples are weighted by a kernel $k(x, x')$, which measures the similarity between a training point x' and test point x. Typical kernels monotonically decrease with growing distance to a training point. Therefore the prediction for a new node that is far away from existing points approaches the optimistic prior estimation, whereas a point close to existing nodes will receive a prediction similar to the weighted mean of values from neighboring nodes. The range in which training points contribute to predictions can be controlled by a *bandwith* parameter β of the kernel, which is

task dependent and needs to be chosen in advance. In our experiments we use a standard squared exponential kernel $k(x, x') = \exp\left(-\frac{d(x,x')^2}{2\beta^2}\right)$.

Since the prior estimate is an optimistic estimation of the true value, the predictions for an exploration node will usually increase the further the new node is away from existing nodes (see Figure 1(b)). Therefore this approach enforces exploration into unvisited areas. Additionally to the value estimate $\hat{V}(\tilde{x})$ the Gaussian process returns the variance $\mathrm{var}_\sigma(\tilde{x})$ of the prediction. Since the variance increases with distance to training points, we can use $\mathrm{var}_\sigma(\tilde{x})$ as a measure for the sampling density around $\tilde{x}$. To control the number of nodes we reject exploration nodes with variance lower than $\theta^{\mathrm{exp}}_{\mathrm{min}}$. This threshold may be lowered over time, to ensure refinement of the adaptive state graph in later episodes.

4.3 Integrating New Exploration Nodes: `insert_exploration_node`

Newly generated exploration nodes $\tilde{x}_i$ are placed on the *exploration queue* $\mathcal{Q}$, which is a priority queue ranked by the exploration scores $\sigma(\tilde{x}_i)$. The highest scored exploration targets in $\mathcal{Q}$ are the most promising candidates for exploration. If σ_{max} is the best score of a node on the queue, we consider all exploration nodes with a score not worse than $\sigma_{\mathrm{max}} - \theta_\sigma$, with $\theta_\sigma \geq 0$ as targets for the exploration policy π^{exp}. Virtual and terminal *exploration edges* are added to the graph for each such node $\tilde{x}$, originating from the node from which $\tilde{x}$ was created. The rewards of these edges are the estimated rewards-to-goal, given by $\hat{V}$. The exploration policy may then either choose an exploration edge, thereby adding a new node to the adaptive state graph, or move to an already visited node. The latter indicates that exploring from other nodes seems more promising than continuing the exploration at the current node.

The threshold parameter θ_σ has an interesting interpretation in the context of the exploration-exploitation dilemma. If $\theta_\sigma = 0$ then the agent will always choose the most promising exploration target, similar to A* search [9] on a partially unknown graph. This will however yield a bad online performance, because the agent may have to travel all the way through the state space if it discovers that another node promises better solutions. $\theta_\sigma = \infty$ will lead to greedy search, and ultimately to inefficient uniform sampling of the whole state space. By adjusting $\theta_\sigma > 0$ one can balance the trade-off between online performance and finding near-optimal start-to-goal paths as soon as possible.

4.4 Re-planning within the Graph: `replan`

The adaptive state graph yields a complete model of the reduced MDP, which can be solved by dynamic programming methods. In practice we use efficient re-planning techniques like Prioritized Sweeping [8] to minimize the number of updates in every iteration. In most steps this requires only a very small number of iterations on a small set of nodes. Only when important connections are found, and the value of many states changes, we need to compute more iterations.

Re-planning is run twice: once on the graph that includes only exploration targets in $\mathcal{Q}$ with score larger than $\sigma_{\max} - \theta_\sigma$ as terminal states. This yields the value function $V^{\exp}$ for the *exploration policy* $\pi^{\exp}$. We also compute the value function V for the task policy π, using all available targets from $\mathcal{Q}$ as terminal nodes. This policy attempts to reach the goal optimally, without performing exploratory actions. It is therefore used in the computation of exploration scores, because there we are only interested in the optimistic rewards-to-goal.

4.5 Action Selection and Incorporation of Actual Experience

At the current node x the agent selects an outgoing edge $e = (x, x')$ through its exploration policy $\pi^{\exp}$, which is derived stochastically (e.g. ε-greedy) from $V^{\exp}$. The local controller $a(x, x')$ then moves towards x'. If the agent reaches a small neighborhood around x' the controller is deactivated, and the reward of the traversed edge in $\mathcal{G}$ is updated. If the local controller does not reach the vicinity of x' within a given maximum time, the controller stops at a state $\hat{x}'$. We then delete the edge $e = (x, x')$ from the graph $\mathcal{G}$, since it cannot be completed by a local controller, and insert an edge from x to $\hat{x}'$ instead.

4.6 Inserting New Nodes: `insert_new_node`

When a node x' is visited for the first time, it is inserted as a new node into the graph. Local controllers to and from all nodes in a certain neighborhood around x' are simulated to create incoming and outgoing edges. If a connection seems possible we insert the edge into $\mathcal{G}$ and store an optimistic estimate of the reward, e.g. the negative estimated transition time of the local controller in absence of obstacles. Inserting a new node x' also invalidates existing exploration nodes in the neighborhood, if their exploration score variance would fall below the threshold $\theta^{\exp}_{\min}$ (see Section 4.2).

If a newly inserted edge $e = (x', x'')$ with estimated reward $\hat{r}(e)$ reduces the path costs from x^S to x'', the edge becomes an attractive target for exploration. We then insert e as an exploration edge into the queue $\mathcal{Q}$. The exploration score is $\sigma(e) = c(x^S, x') + \hat{r}(e) + V(x'')$, which is the estimated return of a path from x^S to X^G that uses e. For the exploration policy the agent may then equally select exploration nodes or edges as its best exploration targets.

4.7 Practical Implementation Issues

Efficient data structures like *kd*-trees reduce the search time for neighbors during the training phase. The CPU time is still higher than for model-based RL methods with fixed discretizations, e.g. Prioritized Sweeping [8]. The construction of an adaptive state graph is an overhead, but on the other hand, it permits better solutions and faster learning.

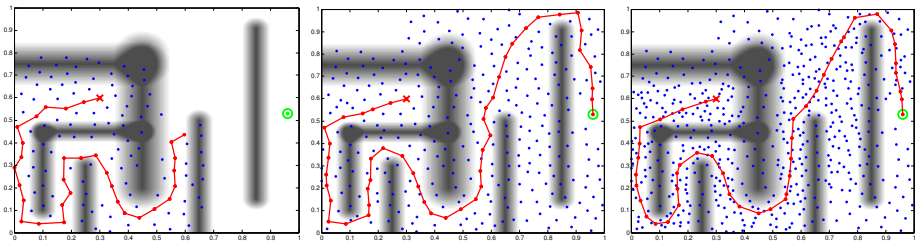
5 Experiments

In this section we show that our algorithm can solve several continuous control problems that are challenging for standard reinforcement learning techniques. We show that the algorithm requires less actual experience than existing methods and finds more accurate trajectories.

5.1 Static Puddle World

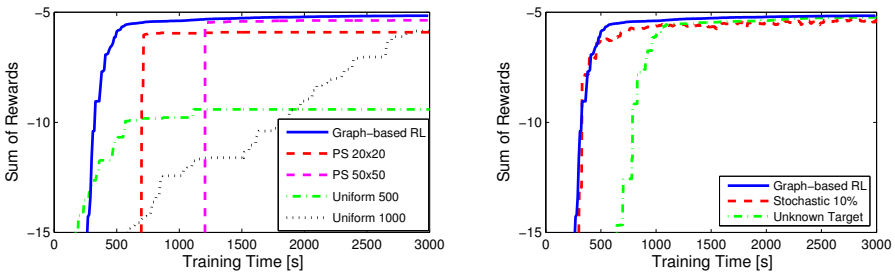
The puddle world task is a well-known benchmark for reinforcement learning algorithms in continuous domains. The objective is to navigate from a given starting state to a goal state in a 2-dimensional environment which contains *puddles*, representing regions of negative reward. Every transition inflicts a reward equal to the negative required time, plus additional penalties for entering a puddle area. The puddles are oval shapes, and the negative reward for entering a puddle is proportional to the distance inside the puddle. The 2-dimensional control action $u = (v_x, v_y)$ corresponds to setting velocities in x and y directions, leading to the simple linear system dynamics $(\dot{x}, \dot{y}) = (v_x, v_y)$. We can safely assume to know this dynamics, but planning a path to the goal state and avoiding the unknown puddles remains a difficult task.

Figure 2 shows various stages of the exploration process in a maze-like puddle world with multiple puddles. As optimistic value estimate $\hat{V}(\tilde{x})$ we use the negative time needed for the direct path to the goal (ignoring any puddles). In Figure 2(a) it can be observed that the agent directs its initial exploration towards the goal, while avoiding paths through regions of negative reward. Less promising regions like the upper left part are avoided. When the agent has reached the goal the first time (Figure 2(b)) the agent knows a coarse path to the goal. With continuing learning time, the agent refines the graph and adds more nodes in relevant regions, which is illustrated in Figure 2(c). The path is almost optimal and avoids all puddles on the way to the goal.



(a) (b) (c)

Fig. 2. Static Puddle World: (a) and (b) shows the graph at the beginning of learning and when the agent has found the goal for the first time. (c) results from further optimization of the graph. The red line indicates the best known policy to the target.



(a) (b)

Fig. 3. Learning performance on static puddle world from Figure 2. (a) Comparison of RL with adaptive state graphs to prioritized sweeping (PS), and greedy search on uniformly sampled nodes (Uniform) with different discretization densities. (b) Influence of stochasticity (10% movement noise) and unknown target states on performance of graph-based RL. (Average over 10 trials.)

Standard TD-learning [1] with CMAC or RBF function approximation needs several thousands of episodes to converge on this task, because a rather fine discretization is required. It is therefore not considered for comparison. Better results were achieved by Prioritized Sweeping [8], a model-based RL algorithm which discretizes the environment and learns the transition and reward model from experience. In Figure 3 we compare the performance of RL with adaptive state graphs to prioritized sweeping with different discretization densities. We also compare the performance of a greedy search method on a graph with 500 and 1000 uniformly sampled nodes, which updates its reward estimates after every step. We evaluate the performance of the agent by measuring the sum of rewards obtained by its greedy policy at different training times. The training time is the total amount of time spent by the agent for actually moving within the state space during the training process. Figure 3(a) shows that the graph-based RL algorithm achieves reasonable performance faster than prioritized sweeping (even with coarse discretization), and the best found policy slightly outperforms all other methods. Our refined graph in the end contains about 730 nodes, which is approximately a fourth of the number of states used by prioritized sweeping on the fine grid. Greedy search on estimated edges initially finds the goal faster, but it either converges to a suboptimal policy, which is due to the uniform sampling, or needs longer to optimize its policy.

In Figure 3(b) we added small Gaussian movement noise (variance is 10% of movement velocity), and used local feedback-controllers. Our algorithm still converges quickly, but due to the stochasticity it cannot reach the same performance as in the deterministic case. We also investigated the (deterministic) problem in which the goal state is unknown. Since the agent has to explore uniformly in the beginning, it needs longer to converge, but ultimately reaches the same performance level.

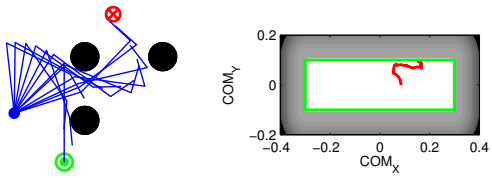


Fig. 4. Arm reaching task with stability constraints. Left: Solution trajectory found by our algorithm. The agent must reach the goal region (red) from the starting position (green), avoiding the obstacles. Right: Trajectory of the CoM of the robot (red) inside the neutral zone (green).

5.2 3-Link Arm Reaching Task

The joints of a simulated planar 3-link robot arm are steered under static stability constraints in an environment with several obstacles (see Figure 4). The objective is to reach a goal area with the tip. The robot consist of a body (point mass with 1 kg), around which the arm - modeled as upper arm (length 0.5 m / weight 0.2 kg), fore arm (0.5 m / 0.1 kg) and hand (0.2 m / 0.05 kg) - can rotate. The center of mass (CoM) of the robot needs to be kept inside a finite support polygon. If the CoM leaves a neutral zone of guaranteed stability ($[-0.2, 0.2]$ in x and $[-0.1, 0.1]$ in y), the agent receives negative reward that grows quadratically as the CoM approaches the boundary of the support polygon. Under these constraints the trivial solution of rotating the arm around the top left obstacle achieves lower reward than the trajectory that maneuvers the arm through the narrow passage between the obstacles.

The 3-dimensional state space consists of the three joint angles, and the control actions correspond to setting the angular velocities. The approximative model $\hat{f}$ is a simple linear model, but the true system dynamics f contains nonlinearities due to obstacles, which are not captured by $\hat{f}$. The optimistic value estimate $\hat{V}(x)$ is the negative time needed by a local controller to reach a target configuration, calculated by simple inverse kinematics. Figure 5 shows that graph-based RL converges much faster to more accurate trajectories than prioritized sweeping with different levels of discretization.

6 Conclusion and Future Work

In this paper we introduced a new efficient combination of reinforcement learning and sampling-based planning for continuous control problems in unknown environments. We use minimal prior knowledge in the form of approximative models and local controllers to increase the learning speed. Our algorithm builds an adaptive state graph through goal-directed exploration. We demonstrated on various movement planning tasks with difficult reward functions that RL with adaptive state graphs requires less actual experience than existing methods to

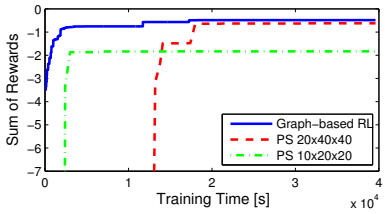


Fig. 5. Learning performance on the 3-link arm reaching task for RL with adaptive state graphs and prioritized sweeping (PS) with different discretization densities. (Average over 10 trials)

obtain high quality solutions. The approach is particularly promising for complicated tasks that can be projected to low dimensional representations, such as balancing humanoid robots using motion primitives [11]. In the future we will extend the approach to non-deterministic and non-episodic, discounted tasks. Extending the approach to non-linear dynamics or even learning the local controllers for more complex dynamical systems is also part of future work.

Acknowledgments This work was supported in part by the Austrian Science Fund FWF under project number P17229 and PASCAL Network of Excellence, IST-2002-506778. This publication only reflects the authors' views.

References

1. Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. MIT Press (1998)
2. Boyan, J.A., Moore, A.W.: Generalization in reinforcement learning: Safely approximating the value function. In: NIPS 7. (1995) 369–376
3. Ormoneit, D., Sen, S.: Kernel-based reinforcement learning. Machine Learning **49**(2-3) (2002) 161–178
4. Jong, N., Stone, P.: Kernel-based models for reinforcement learning. In: ICML Workshop on Kernel Machines and Reinforcement Learning. (2006)
5. Kavraki, L., Svestka, P., Latombe, J., Overmars, M.: Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE T-RA **12**(4) (1996)
6. Guestrin, C.E., Ormoneit, D.: Robust combination of local controllers. In: Proc. UAI. (2001) 178–185
7. Simsek, Ö., Barto, A.: An intrinsic reward mechanism for efficient exploration. In: ICML. (2006) 833–840
8. Moore, A.W., Atkeson, C.G.: Prioritized sweeping: Reinforcement learning with less data and less real time. Machine Learning **13** (1993) 103–130
9. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. SSC **4** (1968) 100–107
10. Rasmussen, C., Williams, C.: Gaussian Processes for Machine Learning. MIT Press (2006)
11. Hauser, H., Ijspeert, A., Maass, W.: Kinematic motion primitives to facilitate control in humanoid robots. In: submitted for publication. (2007)