Auton Robot (2010) 29: 169–200
DOI 10.1007/s10514-010-9191-2

Finding and transferring policies using stored behaviors

Martin Stolle · Christopher Atkeson

Received: 9 November 2008 / Accepted: 20 April 2010 / Published online: 6 May 2010

Abstract We present several algorithms that aim to advance the state-of-the-art in reinforcement learning and planning algorithms. One key idea is to transfer knowledge across problems by representing it using local features. This idea is used to speed up a dynamic programming based generalized policy iteration.

We then present a control approach that uses a library of trajectories to establish a control law or policy. This approach is an alternative to methods for finding policies based on value functions using dynamic programming and also to using plans based on a single desired trajectory. Our method has the advantages of providing reasonable policies much faster than dynamic programming and providing more robust and global policies than following a single desired trajectory.

Finally we show how local features can be used to transfer libraries of trajectories between similar problems. Transfer makes it useful to store special purpose behaviors in the library for solving tricky situations in new environments. By adapting the behaviors in the library, we increase the applicability of the behaviors. Our approach can be viewed as a method that allows planning algorithms to make use of special purpose behaviors/actions which are only applicable in certain situations.

Results are shown for the "Labyrinth" marble maze and the Little Dog quadruped robot. The marble maze is a difficult task which requires both fast control as well as planning ahead. In the Little Dog terrain, a quadruped robot has to navigate quickly across rough terrain.

Keywords Behaviour-based systems · Learning and adaptive systems · Learning from demonstration · Legged robots · Planning

M. Stolle (✉) · C. Atkeson
5000 Forbes Ave, Pittsburgh, PA 15213, USA
e-mail: martin@stolle.name

C. Atkeson
e-mail: cga@cs.cmu.edu

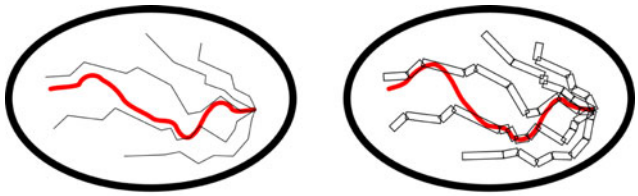
Present address:
M. Stolle
Google Switzerland, Brandschenkestrasse 110, 8002 Zürich, Switzerland

1 Introduction

1.1 Overview

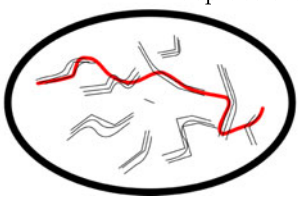
With the work presented here, we aim to advance the state-of-the-art in reinforcement learning and planning algorithms so they can be applied to realistic, high-dimensional problems. Our approach is two-pronged:

One part is to enable the reuse of knowledge across different problems in order to solve new problems faster or better, or enable solving larger problems than are currently possible. The key to attaining these goals is to use multiple descriptions of state in a given domain that enable transfer of knowledge as well as learning and planning on different levels of details. In particular, while most domains have a standard state representation such as Cartesian position and velocity with respect to a fixed origin or joint position and velocities, it is sometimes beneficial to also consider ambiguous descriptions using local features of the agent or task. While only short term predictions of the future are possible due to their local and ambiguous nature, they are powerful tools to generalize knowledge across different parts of the environment or to new problems. In order to avoid the limitations of state aliasing (multiple states mapping to the



(a) A trajectory library populated with elementary actions

(b) A trajectory library populated with closed-loop actions



(c) A library populated with incongruent behaviors

Fig. 1 Different types of libraries. The *red line* illustrates a possible path taken by the agent using the library

same features), it is however important to use local features in conjunction with a global state representation.

The second key idea is to take advantage of libraries of stored behaviors. In particular, when solving stochastic control problems, often great computational resources are spent on computing globally optimal control laws using dynamic programming (DP) or policy search. The reasoning is that given knowledge on how to act from any state, one can still behave well under uncertainty: even after an unexpected state transition, the robot knows what is the best action to pick. In some domains, it is possible to avoid this upfront computation by using path planners to quickly obtain a valid solution. In case the original plan becomes infeasible due to the stochasticity of the environment, replanning is performed to find a new solution. However, this much simpler approach is only possible if computers are fast enough so that delays due to replanning are small enough to be permissible. We aim to close the gap between computing globally optimal policies and replanning by leveraging libraries of trajectories created by path planners. By storing many trajectories in a library, we avoid or reduce replanning while at the same time avoiding the computation required by more traditional methods for finding a globally optimal control law.

Libraries can be implemented in different ways. In the simplest type of library, the planner used to populate the library is reasoning at the level of elementary actions that are directly executed by the agent or sent to motors (Fig. 1(a)). An example of this type of library could be a pendulum swing problem where we know the correct actions for a given angle and angular velocity or a grid world problem where we know in which cardinal direction to move for a given x–y position. For more complex domains, it can quickly become unfeasible to provide such low level actions for a sufficient amount of states. In such complex domains, a library can be populated by planning at the level of more abstract actions (Fig. 1(b)). Low level controllers or heuristics are used to generate the elementary actions when a particular abstract action is executing. An example of this could be an autonomous race vehicle with low level behavior for tracking the optimal race line, passing and obstacle avoidance. Yet another type of library, which is not necessarily created directly from planning, consists of knowledge about behaviors that can be executed in specific parts of the state space (Fig. 1(c)). This third type of library encodes knowledge about possible behaviors, but not all behavior possibilities are necessarily desirable in attaining a specific goal. Hence, a high-level search is necessary to ensure goal-directed behavior. An example of this could be an off-road vehicle with special behaviors for crossing rivers, scaling rocks and tree trunks. Just because there is a rock that the agent knows how to scale, doesn't mean it should do so for achieving its goal.

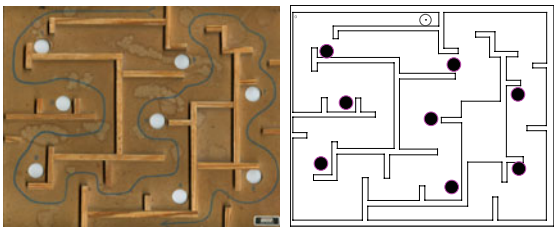
One way the third kind of library is created is when a library of behaviors is transferred to a new environment. Searching through the resulting library of behaviors to find a path can be viewed as augmenting a path planner with external knowledge contained in the library of behaviors. This library of behaviors, describing possibilities for executing special purpose behaviors, enables the path planner to find solutions to hard problems where the models available to the planner are not sufficient to find these behaviors autonomously. Using local features to transfer libraries of behaviors to new problems combines the two key ideas of transfer using local features and libraries of stored behaviors.

This article is organized as follows: In the next section, we introduce and describe the experimental domains we are using to validate the ideas and algorithms. In Sect. 2, we describe an algorithm to speed up the creation of global control laws using dynamic programming by transferring knowledge from previously solved problems. Results are presented for simulations in the marble maze domain. In Sect. 3, we describe a representation for control laws based on trajectory libraries. Results are shown on both simulated and actual marble maze. Finally, in Sect. 4, we propose ways of transferring the libraries presented in Sect. 3.

1.2 Experimental domains

1.2.1 Marble maze

Two domains are used to validate and assess the proposed algorithms: the marble maze domain and the Little Dog domain. The marble maze domain (Fig. 2) is also known as "Labyrinth" and consists of a plane with walls and holes.



(a) original (b) simulation

Fig. 2 A sample marble maze

A ball (marble) is placed on a specified starting position and has to be guided to a specified goal zone by tilting the plane. Falling into holes has to be avoided and the walls both restrict the marble and can help it in avoiding the holes. Both a hardware based, computer controlled setup as well as a software simulator are designed and implemented.

The *simulation* uses a four-dimensional state representation (x, y, dx, dy) where x and y specify the 2D position on the plane and dx, dy specify the 2D velocity. Actions are also two dimensional (fx, fy) and are force vectors to be applied to the marble. This is not identical but similar to tilting the board. The physics are simulated as a sliding block (simplifies friction and inertia). Collisions are simulated by detecting intersection of the simulated path with the wall and computing the velocity at the time of collision. The velocity component perpendicular to the wall is negated and multiplied with a coefficient of restitution of 0.7. The frictional forces are recomputed and the remainder of the time slice is simulated to completion. Some of the experiments use Gaussian noise, scaled by the speed of the marble and added to the applied force in order to provide for a more realistic simulator and to gauge the robustness of the policies. This noise roughly approximates physical imperfections on the marble or the board. Other noise models could be imagined. A higher-dimensional marble maze simulator was used by (Bentivegna 2004). In Bentivegna's simulator the current tilt of the board is also part of the state representation. An even higher fidelity simulator could be created by taking into account the distance of the marble to the rotation axes of the board, because the rotation of the board causes fictitious forces such as centripetal and centrifugal forces on the marble. Additionally, one could improve the simulator using data from executing on a physical maze.

The experiments that were performed on the *physical* maze (Fig. 3) used hobby servos for actuation of the plane tilt. An overhead Firewire 30fps, VGA resolution camera was used for sensing. The ball was painted bright red and the corners of the labyrinth were marked with blue markers. After camera calibration, the positions of the blue markers in the image are used to find a 2D perspective transform for every frame that turns the distorted image of the labyrinth into a rectangle. The position of the red colored ball within this rectangle is used as the position of the ball. Velocity is computed from the difference between the current and the last ball position. Noise in the velocity is quite small compared to the observed velocities so we do not perform filtering. This avoids adding latency to the velocity. As in the simulator, actions are represented internally as forces. These forces are converted into board tilt angles, using the known weight of the ball. Finally, the angles are sent to the servos as angular position.

Fig. 3 The physical maze

Fig. 4 Little Dog environment

1.2.2 Little Dog

Another domain we will use for assessing the effectiveness of the algorithms is the Little Dog domain. Little Dog is a quadruped robot developed by Boston Dynamics for DARPA (Fig. 4). It has four legs, each with three actuated degrees of freedom. Two degrees of freedom are at the hip (inward–outward, forward–backward) and one at the knee

(forward–backward). Torque can be applied to each of the joints. This results in a 12 dimensional action space (three for each of the four legs). The state space is 36 dimensional (24 dimensions for the position and velocity of the leg joints and 12 dimensions for the position, orientation, linear velocity and angular velocity of the center of mass). The task to be solved in this domain is to navigate a small-scale rough terrain.

The robot is controlled by sending desired joint angles to an on-board proportional-derivative (PD) controller for each joint. A PD controller sends desired torques to a motor proportional to the error of the joint angle while subtracting torque proportional to the speed at which the error is decreasing. The desired joint angles can be updated at 100 Hz. The on-board PD controller computes new torque outputs at 500 Hz. The robot is localized using a Vicon motion capture system which uses retro-reflective markers on the robot in conjunction with a set of six infrared cameras. Additional markers are located on the terrain boards. The proprietary Vicon software provides millimeter accuracy location of the robot as well as the terrain boards. We are supplied with accurate 3D laser scans of the terrain boards. As a result, no robot sensor is needed to map the terrain.

The user interface shown in Fig. 5 is used for monitoring the controllers and drawing plans as well as execution traces. Program data is superimposed over a heightmap of the terrain. A more advanced, interactive 3D display is used for playing back and analysing logfiles from previous execution runs (Fig. 6).

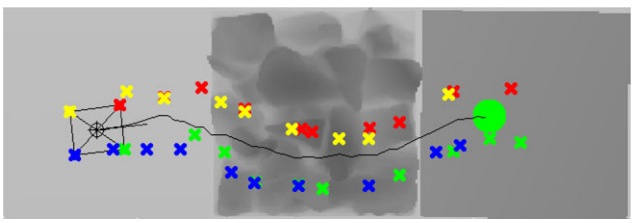


Fig. 5 Little Dog graphical interface with plan. The *black line* shows a hypothetical trajectory for the body while the colored crosses correspond to stance locations of the feet (*red* = front left, *green* = front right; *yellow* = hind left, *blue* = hind right). The dog moves from left to right and the *green circle* marks the goal for the plan

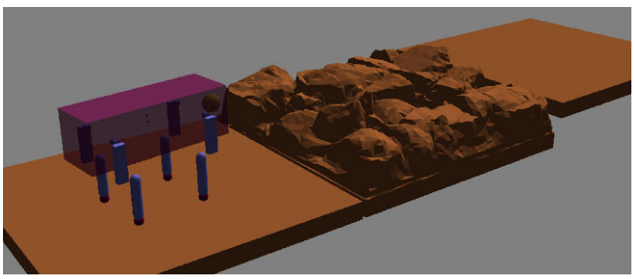


Fig. 6 Little Dog simulator

2 Transfer of policies based on value functions[1]

2.1 Introduction

In this section, we introduce the notion of local features for the purpose of transferring a well-known type of policy, value function based global policies computed using dynamic programming (DP). Policies are functions mapping states to actions, allowing the agent to behave correctly over a large part of the state space. Finding such policies is computationally expensive, especially in continuous domains. The alternative of computing a single path, although computationally much faster, does not suffice in real world domains where sensing is noisy and perturbations from the intended paths are expected.

When solving a new task in the same domain, planning algorithms typically start from scratch. We devise an algorithm which decreases the computation needed to find policies for new tasks based on solutions to previous tasks in the same domain. This is accomplished by initializing a policy for the new task based on policies for previous tasks.

As policies are often expressed using state representations that do not generalize across tasks, policies cannot be copied directly. Instead, we use local features as an intermediate description which generalizes across tasks. By way of these local features, policies can be translated across tasks and used to seed planning algorithms with a good initial policy.

For example, in a navigation domain, a policy is usually defined in terms of (x, y) coordinates. If the terrain or goal changes, the same (x, y) position will often require a different action. For instance, on the left terrain in Fig. 7, the policy of the upper left corner is to go down, whereas in the right terrain the policy of the same position is to go right. However, one can represent the policy in terms of local features that take into account the position of the agent with respect to the goal and obstacles. A new policy is initialized by looking up what the local features are for each state and setting the action of that state to the action that is associated with the local features. By reverting back to the global (x, y)-type state representation, the policy can be refined for the new task without being limited by the local state description.

2.2 Related work

Transfer of knowledge across tasks is an important and recurring aspect of artificial intelligence. Previous work can be classified according to the type of description of the agent's environment as well as the variety of environments

[1] An earlier version was published in (Stolle and Atkeson 2007b).

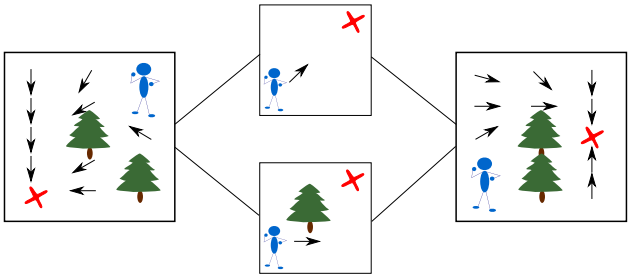


Fig. 7 Example navigation domain, *left*: original terrain, middle: feature-based policy, *right*: new terrain

the knowledge can be transferred across. For symbolic planners and problem solvers, high level relational descriptions of the environment allow for transfer of plans or macro operators across very different tasks, as long as it is still within the same domain. Work on transfer of knowledge in such domains includes STRIPS (Fikes et al. 1972), SOAR (Laird et al. 1986), Maclearn (Iba 1989) and analogical reasoning with PRODIGY (Veloso 1992). More recent relevant work in discrete planning can be found in (Winner and Veloso 2002; Fern et al. 2004).

In controls, research has been performed on modeling actions using local state descriptions (Mahadevan 1992; Chernova and Veloso 2004b). Other work has been done to optimize low-level controllers, such as walking gaits, which can then be used in different tasks (Kohl and Stone 2004; Chernova and Veloso 2004a; Röfer 2005; Weingarten et al. 2004). In contrast, our work focuses on finding policies which take into account features of the specific task.

Some research has been performed in automatically creating macro-actions in reinforcement learning (McGovern 2002; Stolle and Precup 2002; Şimşek and Barto 2004; Mannor et al. 2004), however those macro actions could only transfer knowledge between tasks where only the goal was moved. If the environment was changed, the learned macro actions would no longer apply as they are expressed in global coordinates, a problem we are explicitly addressing using feature-based descriptions. Another method for reusing macro actions in different states by discovering geometric similarities of state regions (homomorphism) can be found in (Ravindran and Barto 2003).

At the intersection of planning and control, work in relational reinforcement learning creates policies that operate on relational domain state descriptions (Fern et al. 2006; Yoon et al. 2008). Applying the policy to the planning domain is expected to either solve the planning query (which explicitly encodes the goal state) or guide a search method. The policy is learned only once for a given domain and is reused for different planning queries. Similar to traditional work in relational planning, the policies derive their ability to generalize from the relational domain description. Work that is more closely related to ours in solving relational Markov Decision Processes (MDP) can be found in (Guestrin et al. 2003). Like in our approach, a domain expert creates an alternative state description. This state description allows for the creation of a factored MDP over classes. Every class has a value function associated with it that depends on the state of an instance of that class. For a particular environment, the value of a state is the sum of the value of each instance of every class. This allows for generalization to new problems, assuming the state of the instances contains the information necessary to generalize.

Finally, a related area of research is multi-task learning (Caruana 1993). The idea behind multi-task learning is that a machine learning algorithm (originally neural networks) can learn faster if it learns multiple related tasks at the same time. There are two ways to look at this: One way is that the input to the machine learning algorithm is very high dimensional and by learning multiple tasks at the same time, the machine learning algorithm can learn which state features are relevant. When learning new tasks, the algorithm can focus learning on those features. Alternatively, one can hope for the machine learning algorithm to compute new relevant features from the given input features. In navigational domains, this would require the whole map to be part of the input state, which would dramatically increase the size of the state space. It is unclear what kind of relationships between original state (such as position) and maps would be learned.

2.3 Case study: marble maze

We used the marble maze domain (Fig. 2) to gauge the effectiveness of our knowledge transfer approach. The model used for dynamic programming is the simulator described in Sect. 1.2.1. The reward structure used for reinforcement learning in this domain is very simple. Reaching the goal results in a large positive reward. Falling into a hole terminates the trial and results in a large negative reward. Additionally, each action incurs a small negative reward. The agent tries to maximize the reward received, resulting in policies that roughly minimize the time to reach the goal while avoiding holes.

Solving the maze from scratch was done using value iteration. In value iteration, dynamic programming sweeps across all states and performs the following update to the value function estimate V for each state s:

$$V^{t+1}(s) = \max_a \{r(s,a) + V^t(s(a))\} \tag{1}$$

where a ranges over all possible actions, $r(s,a)$ is the reward received for executing a in state s and $s(a)$ is the next state reached after a is executed in state s.

The simulator served as the model for value iteration. The state space was uniformly discretized and multi-linear interpolation was used for the value function (Davies 1997).

For the marble maze, we used a 4-dimensional state-space containing position and velocity in the plane ($s = \langle x, y, dx, dy \rangle$). The positional resolution of the state space was 3 mm and the velocity resolution was 12.5 mm/s. The mazes were of size 289 mm by 184 mm and speeds between -50 mm/s to $+50$ mm/s in both dimensions were allowed, resulting in a state space of about 380,000 states. This resolution is the result of balancing memory requirements and accuracy of the policy. At coarser resolution, values in some parts of the state space were inadequately resolved, resulting in bad policies. Variable resolution methods such as (Munos and Moore 2002) could be used to limit high-resolution representation to parts of the space where it is strictly necessary. We used a 2-dimensional action-space representing force in the plane ($a = \langle fx, fy \rangle$). The maximum force on the marble in each dimension was limited to 0.0014751 N and discretized into $-.001475$ N, 0 and $+.001475$ N in each dimension, resulting in 9 possible actions for each state. With a simulated mass of the marble of .0084 kg, maximal acceleration was about 176 mm/s^2 in each dimension. Time was discretized to 1/60th of a second.

2.3.1 Local state description

The local features, chosen from the many possible local features, depicts the world as seen from the point of view of the marble, looking in the direction it is rolling. Vectors pointing towards the closest hole, the closest wall as well as along a path towards the goal (dashed line in Fig. 8) are computed. These vectors are normalized to be at most length 1 by applying the logistic function to them. The path towards the goal is computed using A* on a discretized grid of the configuration space (**position only**). A* is very fast but does not take into account velocities and does not tell us what actions to use. Two examples of this local state description can be seen in Fig. 8. In the circle representing the relative view from the marble, the forward velocity is towards the right. In the first example, the marble is rolling towards a hole, so the hole vector is pointing ahead, slightly to the right of the marble, while the wall is further to the left. The direction to the goal is to the left and slightly aft. This results in a state vector of (.037; $-.25$, $-.97$; .72, $-.38$; .66, .34), where .037 is the scalar speed of the marble (not shown in figure), followed by the relative direction to the goal, relative direction to the closest wall and relative direction to the closest hole. The second example has the closest hole behind the marble, the closest wall to the left and the direction to the goal to the right of the direction of the marble, resulting in a state vector of (.064; .062, .998; $-.087$, $-.47$; $-.70$, .58). As all vectors are relative to the forward velocity, the velocity becomes a scalar speed only. Actions can likewise be *relativized* by projecting them onto the same forward velocity vector. For comparison purposes, we show results for a different local state description in the discussion section (Sect. 2.5). Note: while A* requires global information, we still consider the information it provides as local, as it is not tied to the global state of the marble. The purpose of local features is not to eschew global information but to ensure that information is not tied to a global state.

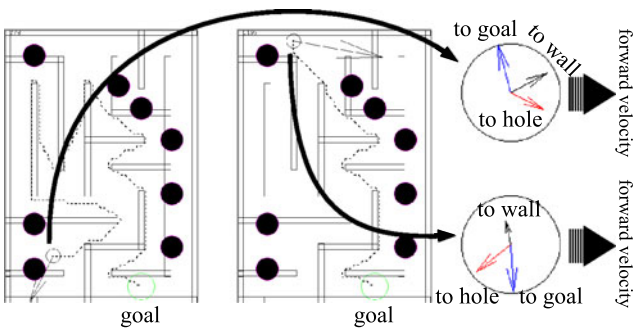


Fig. 8 Local state description

2.3.2 Knowledge transfer

The next step is to transfer knowledge from one maze to the next. For the intermediate policy, expressed using the local state description, we used a nearest neighbor classifier with a kd-tree as the underlying data structure for efficient querying. After a policy has been found for a maze, we iterate over the states and add the local state description with their local actions to the classifier. It is possible to use this intermediate policy directly on a new maze. For any state in the new maze, the local description is computed and the intermediate policy is queried for an action. However, in practice this does not allow the marble to complete the maze because it gets stuck. (It will roll into a corner and the action, deterministically chosen for that state, keeps the marble pushed into the corner.) Furthermore, performance would be expected to be suboptimal as the local description alone does not necessarily determine the optimal action and previous policies might not have encountered states with local features similar to states that now appear on the new task.

Instead, an initial policy based on global coordinates is created using the classifier by iterating over states of the new maze and querying the classifier for the appropriate action based on the local features of that state. This policy is then refined.

2.3.3 Improving the initial policy

Originally, we wanted to use policy evaluation to create a value function from the initial policy which could then be further optimized using value iteration. In policy evaluation, the following update is performed for every state to update the value function estimate:

$$V_\pi^{t+1}(s) = r(s,a) + V_\pi^t(s(a)) \tag{2}$$

where $a = \pi(s)$, the action chosen in state s by the policy π.

Compared to value iteration (1), policy evaluation requires fewer computations per state because only one action is evaluated as opposed to every possible action. We hoped that the initial value function could be computed using little computation and that the subsequent value iterations would terminate after a few iterations.

However, some regions of the state space had a poor initial policy so that values were not properly propagated through these regions. In goal directed tasks such as the marble maze, the propagation of a high value frontier starting from the goal is essential to finding a good policy: At first all states will have a value of zero. When performing a sweep of policy evaluation, all states will lower their value (due to the one step cost and then reaching a state of value zero), except those states where the policy picks an action that reaches the goal. In the next sweep, again most states will lower their value, except those states that reach the goal or which reach a state that itself reaches the goal. This way, every sweep will grow the region of states that have a high values. However, it is possible to have a region of states which could reach the goal but where the policy chooses actions that do not result in a state that reaches the goal (given the policy). If such a region extends from wall to wall in a corridor of the marble maze, the high value region growing from the goal cannot grow past this region.

As a result, the values behind these bad regions will be uninformed and value iteration will not be sped up. Additionally, if a policy improvement step was used to update the policy in these states, the policy of states behind these bad regions would be updated based on an uninformed value function. An intuitive example of this might be the case of a tightrope walk: The optimal policy includes a tightrope walk. However, if in one of the states on the tightrope the policy chooses an action leading to falling off the rope, the value of that state will be low. If we then perform a policy update on the states leading up to the tightrope, we will incorrectly avoid the tightrope walk. We have to ensure to first update the policy of states between the goal and the tightrope walk, including the state with the incorrect action on the tight rope, before we can allow updating the policy of states leading up to the tight rope on the non-goal side.

We overcame these two problems by creating a form of generalized policy iteration (Sutton and Barto 1998). The objective in creating this dynamic programming algorithm was to efficiently use the initial policy to create a value function while selectively improving the policy where the value function estimates are valid. Our algorithm performs sweeps over the state space to update the value of states based on a fixed policy. In a small number of randomly selected states, the policy is updated by checking all actions (a full value iteration update using (1)). As this is done in only a small number of states (on the order of a few percent), the additional computation required is small. The states are selected at random for every sweep.

In order to avoid changing the policy for states using invalid or uninformed values, the randomly selected states are filtered. Only those states are updated where the updated action results in a transition to a state which has been updated with a value coming from the goal. This way we ensure that the change in policy is warranted and a result of information leading to the goal. This method of attempting to update a state and then filtering is necessary, as we do not have an inverse model available which would allow us to propagate values back from the goal more directly. The check we used can easily be implemented by a flag for each state that is propagated back at each update with the values. Note that as a result, we do not compute the value of states that cannot reach the goal.

2.4 Simulation results

In order to gauge the efficiency of the algorithm, a series of simulated experiments was run. First, pools of 30 training mazes and 10 test mazes where created using a random maze generator (mazes available from Stolle 2007). We trained the intermediate classifier with an increasing number of training mazes to gauge the improvement achieved as the initial policy becomes more informed. The base case for the computation required to solve the test mazes was the computation required when using value iteration.

Computational effort was measured by counting the number of times that a value backup was computed before a policy was found that successfully solved the maze. The procedure for measuring the computational effort was to first perform 200 dynamic programming sweeps and then performing a trial in the maze based on the resulting policy. Following that, we alternated between computing 50 more sweeps and trying out the policy until a total of 1000 dynamic programming sweeps were performed.

When performing a trial, the policy was to pick the best action with respect to the expected reward based on the current estimate of the value function. Figure 9 shows the quality of the policy obtained in relation to the number of value backups. The right most curve represents value iteration from scratch and the other curves represent starting with an initial policy based on an increasing number of training mazes. The first data points show a reward of -2100 because policy execution was limited to 2100 time steps. The trials were aborted if the goal was not yet reached.

Clearly, an initial policy based on the intermediate policy reduces the computation required to find a good policy. However, final convergence to the optimal policy is slow because only a small number of states are considered for policy updates. This results in a slightly lower solution quality in our experiments.

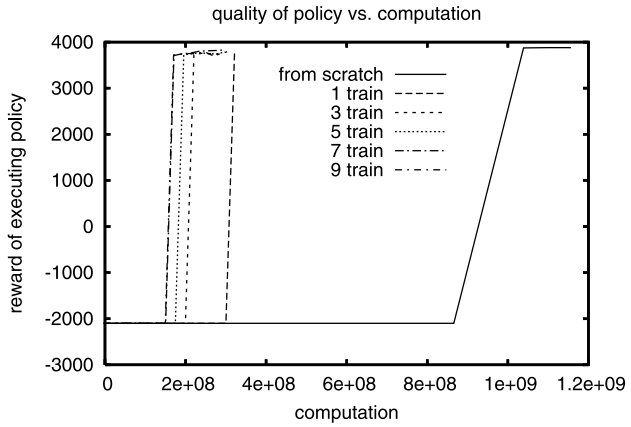


Fig. 9 Results for one test maze. While learning, every couple of iterations the maze is simulated with actions taken from the current policy. Reward for reaching the goal is 5000 and the penalty for every step is 1. A simulation is limited to 2100 steps. An uninformed policy will result in the marble not reaching the goal and a total reward of −2100. Once the policy has improved to the point of reaching the goal, the total reward will be 5000 minus the number of steps to get to the goal

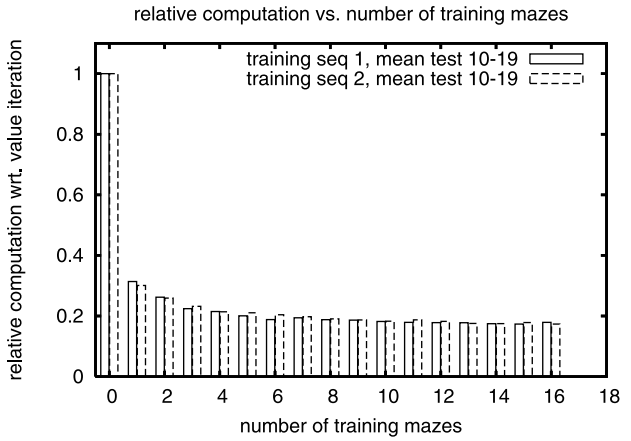


Fig. 10 Relative computation, averaged over 10 test mazes, using two different sequences of training mazes

In order to ensure that the savings are not specific to this test maze, we computed the relative computation required to find a policy that successfully performs the maze for ten different test mazes and plotted the mean in Fig. 10 (solid). Additionally, in order to exclude the peculiarities of the training mazes as a factor in the results, we reran the experiments with other training mazes. The results can be seen in Fig. 10 (dashed). Clearly, the individual training mazes and their ordering do not influence the results very much.

2.5 Discussion

State description: The local features that we are proposing as a solution to this problem are intuitively defined as features of the state space that are in the immediate vicinity

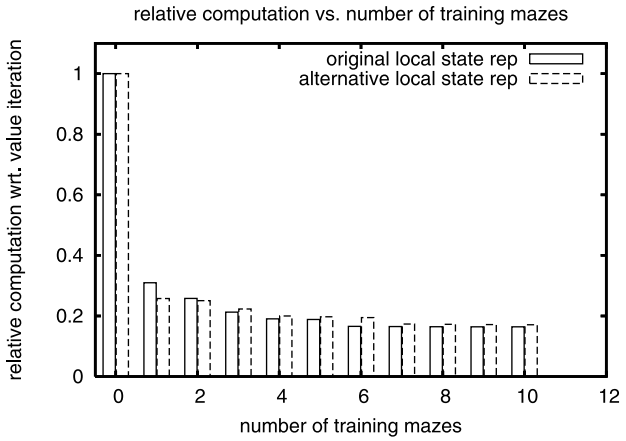


Fig. 11 Relative computation required for one test maze for two different local state descriptions

of the agent. However, often the agent is removed from the actual environment and might even be controlling multiple entities or there may be long-range interactions in the problem. A more accurate characterization of the features we are seeking are that they influence the results of the actions in a consistent manner across multiple tasks and allow, to a varying degree, predictions about the relative value of actions. These new features have to include enough information to predict the outcome of the same action across different environments and should ideally not include unnecessary information that does not affect the outcome of actions. They are similar in spirit to predictive state representation (Littman et al. 2002). These conditions will preclude features such as position on a map, as this alone will not predict the outcome of actions—obstacles and goals are much more important.

In order to gauge the effect of different local state descriptions, we created an alternative state description. In this alternative description, the world is described again as seen from the marble, but aligned with the direction to the goal instead of the direction of the movement. Furthermore, the view is split up into 4 quadrants: covering the 90 degrees towards the path to the goal, 90 degrees to the left, to the right and to the rear. For each quadrant, the distance to the closest hole and closest wall are computed. Holes that are behind walls are not considered. The velocity of the marble is projected onto the path towards the goal. The resulting state description is less precise with respect to direction to the walls or holes than the original local description but takes into account up to four holes and walls, one for each quadrant. As can be seen in Fig. 11, the results are similar for both state descriptions. The new state description performs slightly better with fewer training mazes but loses its advantage with more training mazes.

Computational saving: There are several factors that influence the computational saving one achieves by using an informed initial policy. The computational reduction results

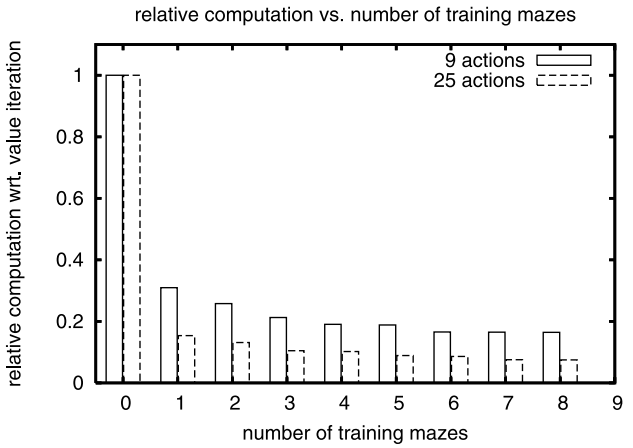


Fig. 12 Relative computation required for one test maze and different number of actions

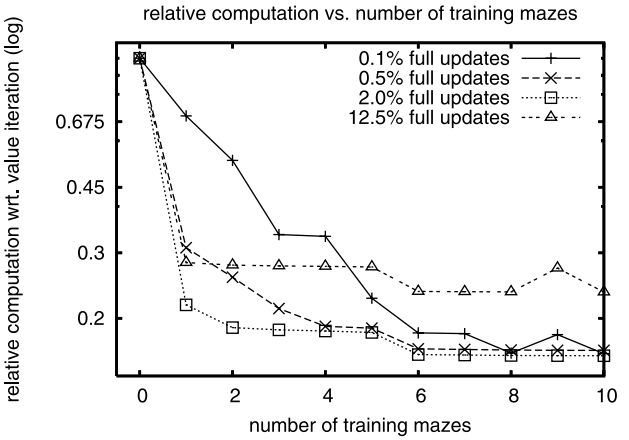


Fig. 13 Relative computation required for one test maze and different percentages of full updates

from the fact that our generalized policy evaluation only computes the value of a single action at each state, whereas value iteration tries out all actions for every state. As a result, if the action space is discretized at high resolution, resulting in many possible actions at each state, the computational savings will be high. If on the other hand there are only two possible actions at each state, the computational saving will be much less. The computation can be reduced at most by a factor equal to the number of actions. However, since in a small number of states in the generalized policy evaluation we also try all possible actions, the actual savings at every sweep will be less. In order to show the effects of changing the number of actions, we reran the experiments for one maze with actions discretized into 25 different actions instead of 9. As seen in Fig. 12, the relative computational saving becomes significantly larger, as was expected.

We also ran experiments to determine the effect of performing policy updates on a varying number of states. If many states are updated at every sweep, fewer sweeps might be necessary, however each sweep will be more expensive. Conversely, updating fewer states can result in more sweeps, as it takes longer to propagate values across bad regions which are now less likely to be updated. The results are presented in Fig. 13. When reducing the percentage of states updated to 0.1%, the computational saving is reduced as it now takes many more sweeps to find a policy that solves the maze, unless the initial policy is very good (based on several mazes). The savings become more pronounced as more states are updated fully and are the greatest when 2.0% of the states are updated, performing better than our test condition of 0.5%. However, increasing the number of states updated further results in reduced savings as now the computational effort at every sweep becomes higher. Comparing the extreme cases shows that when updating few states, the initial policy has to be very good (many training mazes added), as correcting mistakes in the initial policy takes longer. On the other hand, if many states are updated, the quality of the initial policy is less important—many states are updated using the full update anyways.

Intermediate policy representation: Another issue that arose during testing of the knowledge transfer was the representation of the intermediate policy representation. We chose a nearest neighbor approach, as this allows broad generalization early on, without limiting the resolution of the intermediate policy once many training mazes were added to the intermediate policy. However, after adding many mazes, the data structure grew very large (around 350,000 data points per maze, around 5 million for 15 mazes). While the kd-trees performed well, the memory requirements became a problem. Looking at the performance graph, adding more than 5 mazes does not seem to make sense with the current state description. However, if a richer state description was chosen, it might be desirable to add more mazes and then pruning of the kd-tree becomes essential.

The nearest neighbor algorithm itself is modifiable through the use of different distance functions. By running the distances to the closest hole and wall through a logistic function, we have changed the relative weight of different distances already. However, instead one could imagine rescaling distance linearly to range from 0 and 1, where 1 is the longest distance to either hole or wall observed.

Dynamic programming on local state space: As we are using the local state space to express an intermediate policy, it might be interesting to perform dynamic programming in this state space directly. Due to the possible aliasing of different states to the same local state, the problem becomes a partially observable Markov decision process (POMDP). This is aggravated if one keeps the value function across multiple tasks, as now even more states are potentially aliased to the same local state. A policy is less sensitive to this aliasing, as the actions might still be similar while the values could be vastly different. An example can

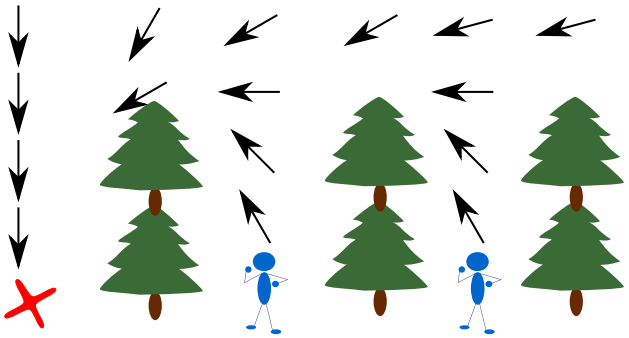


Fig. 14 Aliasing problem: same local features, same policy but different values

be seen in Fig. 14. Both positions with the agent have the same features and the same policy, but the value would be different under most common reward functions which favor short paths to the goal (either with discounting or small constant negative rewards at each time step). We avoid this problem by expanding the intermediate feature-based policy back into a global state-based policy and performing policy iteration in this state space (see also Fig. 7). For similar reasons, it is tricky to transfer the value function directly: the same local features might have different values depending on their distance to the goal. However, adding additional features to the local features such as distance to the goal, might allow function approximation to learn this dependency and allow dynamic programming to be applied directly to local state space.

2.6 Summary

We presented a method for transferring knowledge across multiple tasks in the same domain. Using knowledge of previous solutions, the agent learns to solve new tasks with less computation than would be required without prior knowledge. Key to this knowledge transfer was the creation of a local state description that allows for the representation of knowledge that is independent of the individual task.

3 Policies based on trajectory libraries[2]

3.1 Introduction

In the previous section, we introduced an algorithm to transfer policies based on value functions found via DP. Unfortunately, methods for finding and improving such policies, even when they are initialized from previous policies, are computationally expensive and require large amounts of fast memory. Furthermore, finding a suitable representation for the value function in continuous or very large discrete domains is difficult. Discontinuities in the value function or its derivative are hard to represent and can result in unsatisfactory performance of dynamic programming methods. Finally, storing and computing this value function is impractical for problems with more than a few dimensions.

When applied to robotics problems, dynamic programming methods also become inconvenient as they cannot provide a "rough" initial policy quickly. In goal directed problems, a usable policy can only be obtained when the value function has almost converged. The reward for reaching the goal has to propagate back to the starting state before the policy exhibits goal directed behavior from this state. This may require many sweeps. If only an approximate model of the environment is known, it would be desirable to compute a rough initial policy and then spend more computation after the model has been updated based on experience gathered while following the initial policy.

In some sense, using dynamic programming is both too optimistic and too pessimistic at the same time: it is too optimistic because it assumes the model is accurate and spends a lot of computation on it. At the same time, it is too pessimistic, as it assumes that one needs to know the correct behavior from any possible state, even if it is highly unlikely that the agent enters certain parts of the state space.

To avoid the computational cost of global and provably stable control law design methods such as dynamic programming, often a single desired trajectory is used, with either a fixed, time varying linear or state dependent control law. The desired trajectory can be generated manually, generated by a path planner (LaValle and Kuffner 2001), or generated by trajectory optimization (von Stryk 2001). For systems with nonlinear dynamics, this approach may fail if the actual state diverges sufficiently from the planned trajectory. Another approach to making trajectory planners more robust is to use them in real time at fixed time intervals to compute a new plan from the current state. For complex problems, these plans may have to be truncated (N step lookahead) to obey real time constraints. It may be difficult to take into account longer term outcomes in this case. In general, single trajectory planning methods produce plans that are at best locally optimal.

To summarize, we would like an approach to finding a control law that, on the one hand, is more anytime (Boddy and Dean 1994) than dynamic programming—we would like to find rough policies quickly and expend more computation time only as needed. On the other hand, the approach should be more robust than single trajectory plans.

In order to address these issues, we propose a representation for policies and a method for creating them. This representation is based on libraries of trajectories. Figure 15 shows a simple navigational domain example. The cross marks the goal and the trees represent obstacles. The black

[2]Partially published in (Stolle and Atkeson 2006).

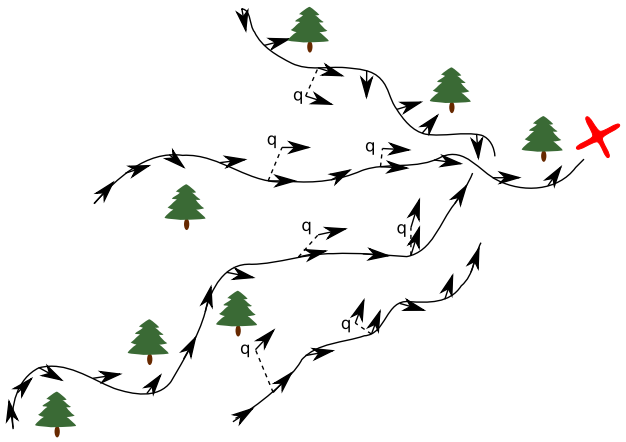


Fig. 15 Illustration of a trajectory library. When queried at any point (e.g. 'q'), the action (indicated by *arrows*) of the closest state on any trajectory is returned

lines are the trajectories which make up the library and the attached arrows are the actions. These trajectories can be created very quickly using forward planners such as A* or Rapidly exploring Random Trees (RRT) (LaValle and Kuffner 2001). The trajectories may be non-optimal or locally optimal depending on the planner used, in contrast to the global optimality of dynamic programming.

Once we have a number of trajectories and we want to use the agent in the environment, we turn the trajectories into a state-space based policy by performing a nearest-neighbor search in the (global) state-space for the closest trajectory fragment and executing the associated action. For example in Fig. 15, when queried in states marked with 'q', the action of the closest state on any trajectory (shown with the dashed lines) is returned.

3.2 Related work

Using libraries of trajectories for generating new action sequences has been discussed in different contexts before. Especially in the context of generating animations, motion capture libraries are used to synthesize new animations that do not exist in that form in the library (Lau and Kuffner 2005; Lee et al. 2002). However, since these systems are mainly concerned with generating animations, they are not concerned with the control of a real world robot and only string together different sequences of configurations, often ignoring physics, disturbances or inaccuracies.

Another related technique in path planning is the creation of Probabilistic Roadmaps (PRMs) (Kavraki et al. 1996). The key idea of PRMs is to speed up multiple planning queries by precomputing a roadmap of plans between stochastically chosen points. Queries are answered by planning to the nearest node in the network, using plans from the network to get to the node closest to the goal and then planning from there to the goal. The method presented here and PRMs have some subtle but important differences. Most importantly, PRMs are a path planning algorithm. Our algorithm, on the other hand, is concerned with turning a library of paths into a control law. Internally, PRMs precompute bidirectional plans that can go from and to a large number of randomly selected points. However, the plans in our library all go to the same goal. As such, the nature of the PRM's "roadmap" is very different than the kind of library we require. Of course, PRMs can be used as a path planning algorithm to supply the paths in our library. Due to the optimization for multiple queries, PRMs might be well suited for this and are complementary to our algorithm.

Libraries of low level controllers have been used to simplify planning for helicopters in Frazzoli's Ph.D. thesis (Frazzoli 2001). The library in this case is not the solution to the goal achievement task, but rather a library of controllers that simplifies the path planning problem. The controllers themselves do not use libraries. Older works exists on using a library of pregenerated control inputs together with the resulting path segments to find a sequence of control inputs that follow a desired path (Grossman et al. 1992; Bailey et al. 1996) or nearby waypoint (Sermanet et al. 2009). These are examples of motion primitives where a library of behaviors is used as possible actions for planning instead of low level actions. These motion primitives encode possible behaviors of the robot and are not used as reactive controllers like the work presented here. Typically, motion primitives assume that their applications will result in the same trajectory *relative to the starting position* no matter which position they are applied from. Bouncing into a wall would be grounds for disqualifying the use of a particular motion primitive. An exception to this is the work by Howard and Kelly (Howard and Kelly 2007), where a library of motion primitives (generated on flat ground) is used to seed an optimizer that takes into account interactions with rough terrain for particular instantiations of a primitive.

Prior versions of a trajectory library approach, using a modified version of Differential Dynamic Programming (DDP) (Jacobson and Mayne 1970) to produce globally optimal trajectories can be found in (Atkeson 1994; Atkeson and Morimoto 2003). This approach reduced the cost of dynamic programming, but was still quite expensive and had relatively dense coverage. The approach of this section uses more robust and cheaper trajectory planners and strives for sparser coverage. Good (but not globally optimal) policies can be produced quickly.

Other related works in planning do not attempt to use libraries of trajectories but exploit geometrical properties of the state space and carefully analyze the model of the environment to create vector fields. These feedback motion plans (Rimon and Koditschek 1992; Connolly and Grupen 1993; Conner et al. 2003) can be hard to compute and it is unclear how to make use of discontinuities in the model, such as

bouncing into a wall in the case of the marble maze. Sampling based methods such as (Yang and LaValle 2004) have been introduced to simplify the construction of feedback motion plans.

3.3 Case study: marble maze

The first domain used for gauging the effectiveness of the new policy representation and generation is the marble maze domain (Fig. 2). The model used for creating trajectories is the simulator described in Sect. 1.2.1. The hardware described in the same section was used for the experiments on the actual maze. The actions in the library are tilts that are directly sent to the maze, which makes this a library of elementary actions (see Fig. 1(a)).

3.3.1 Trajectory libraries

The key idea for creating a global control policy is to use a library of trajectories, which can be created quickly and that together can be used as a robust policy. The trajectories that make up the library are created by established planners such as A* or RRT. Since our algorithm only requires the finished trajectories, the planner used for creating the trajectories is interchangeable. For the experiment presented here, we used an inflated-heuristic (Pearl 1985) A* planner. By overestimating the heuristic cost to reach the goal, we empirically found planning to proceed much faster because it favors expanding nodes that are closer to the goal, even if they were reached sub-optimally. This might not be the case generally (Pearl 1985). We used a constant cost per time step in order to find the quickest path to goal. In order to avoid risky behavior and compensate for inaccuracies and stochasticity, we added a cost inversely proportional to the squared distance to the closest hole on each step. As basis for a heuristic function, we used distance to the goal. This distance is computed by a configuration space (position only) A* planner working on a discretized grid with 2 mm resolution. The final heuristic is computed by dividing the distance to the goal by an estimate of the distance that the marble can travel towards the goal in one time step. As a result, we get a heuristic estimate of the number of time steps required to reach the goal.

The basic A* algorithm is adjusted to continuous domains as described in (LaValle 2006). The key idea is to prune search paths by discretizing the state space and truncating paths that fall in the same discrete "bin" as one of the states of a previously expanded path (see Fig. 16 for an illustration in a simple car domain). This limits the density of search nodes but does not cause a discretization of the actual trajectories. Actions were limited to physically obtainable forces of up to ± 0.007 N in both dimension and discretized to a resolution of 0.0035 N. This resulted in 25 discrete action choices. For the purpose of pruning the search nodes, the state space was discretized to 3 mm spatial resolution and 12.5 mm/s in velocity resolution.

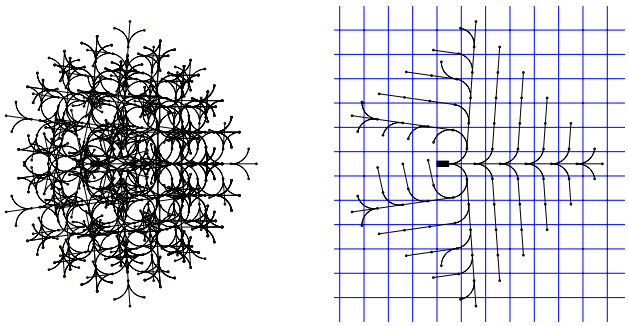


Fig. 16 An example of pruning (LaValle 2006)

The A* algorithm was slightly altered to speed it up. During search, each node in the queue has an associated action multiplier. When expanding the node, each action is executed as many times as dictated by the action multiplier. The new search nodes have an action multiplier that is incremented by one. As a result, the search covers more space at each expansion at the cost of not finding more optimal plans that require more frequent action changes. In order to prevent missed solutions, this multiplier is halved every time none of the successor nodes found a path to the goal, and the node is re-expanded using the new multiplier. This resulted in a speed up in finding trajectories (over $10\times$ faster). The quality of the policies did not change significantly when this modification was applied.

As the policy is synthesized from a set of trajectories, the algorithms for planning the trajectories have a profound impact on the policy quality. If the planned trajectories are poor, the performance of the policy will be poor as well. While in theory A* can give optimal trajectories, using it with an admissible heuristic is often too slow. Furthermore, some performance degradation derives from the discretization of the action choices. RRT often gives "good" trajectories, but it is unknown what kind of quality guarantees can be made for the trajectories created by it. However, the trajectories created by either planning method can be locally optimized by trajectory optimizers such as DDP (Jacobson and Mayne 1970) or DIRCOL (von Stryk 2001).

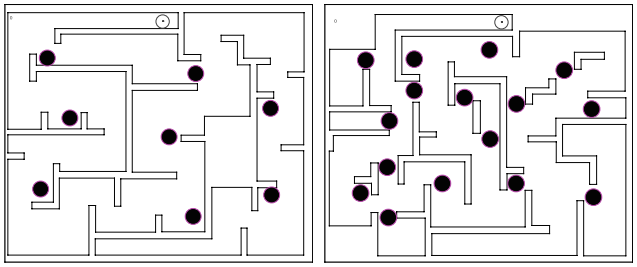
Currently, no smoothness constraints are imposed on the actions of the planners. It is perfectly possible to command a full tilt of the board in one direction and then a full tilt in the opposite direction in the next time step (1/30th second later). Only imposing constraints on the plans would not solve the problem as the policy look up might switch between different trajectories. However, by including the current tilt angles as part of the state description and have the actions be changes in tilt angle, smoother trajectories could be enforced at the expense of adding more dimensions to the state space.

In order to use the trajectory library as a policy, we store a mapping from each state on any trajectory to the planned action of that state. During execution, we perform a nearest-neighbor look up into this mapping using the current state to determine the action to perform. We used a weighted Euclidean distance which tries to normalize the influence of distance (measured in meters) and velocity (measured in meters per second). As typical velocities are around 0.1 m/s and a reasonable neighborhood for positions is about 0.01 m, we multiply position by 100 and velocities by 10, resulting in distances around 1 for reasonably close data points.

We speed up the nearest-neighbor look ups by storing the state-action mappings in a kd-tree (Friedman et al. 1977). Performance of the kd-tree was very fast. After 100 runs, the library contained about 1200 state-action pairs and queries took about 0.01 ms on modest hardware (Pentium IV, 2.4 GHz). Query time is expected to grow logarithmically with the size of the library.

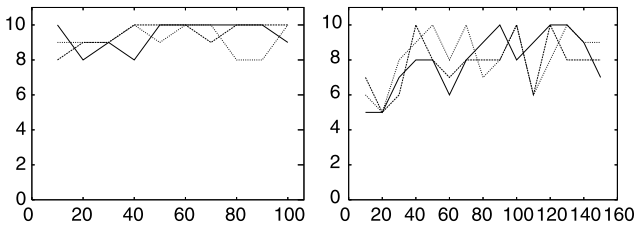
Part of the robustness of the policies derives from the coverage of trajectories in the library. In the experiments on the marble maze, we first created an initial trajectory from the starting position of the marble. We use three methods for adding additional trajectories to the library. First, a number of trajectories are added from random states in the vicinity of the first path. This way, the robot starts out with a more robust policy. Furthermore, during execution it is possible that the marble ceases making progress through the maze, for example if it is pushed into a corner. In this case, an additional path is added from that position. Finally, to improve robustness with experience, at the end of every failed trial a new trajectory is added from the last state before failure. If no plan can be found from that state (for example because failure was inevitable), we backtrack and start plans from increasingly earlier states until a plan can be found. Computation is thus focused on the parts of the state space that were visited but had poor coverage or poor performance. In later experiments, the model is updated during execution of the policy. In this case, the new trajectories use the updated model. The optimal strategy of when to add trajectories, how many to add, and from which starting points is a topic of future research.

Finally, we developed a method for improving an existing library based on the execution of the policy. For this purpose, we added an additional discount parameter to each trajectory segment. If at the end of a trial the agent has failed to achieve its objective, the segments that were selected in the policy leading up to the failure are discounted. This increases the distance of these segments in the nearest-neighbor look up for the policy and as a result these segments have a smaller influence on the policy. This is similar to the mechanism used in learning from practice in Bentivegna's marble maze work (Bentivegna 2004). We also used this mechanism to discount trajectory segments that led up to a situation where the marble is not making progress through the maze.



(a) Maze A (b) Maze B

Fig. 17 The two mazes used for testing



(a) Maze A (b) Maze B

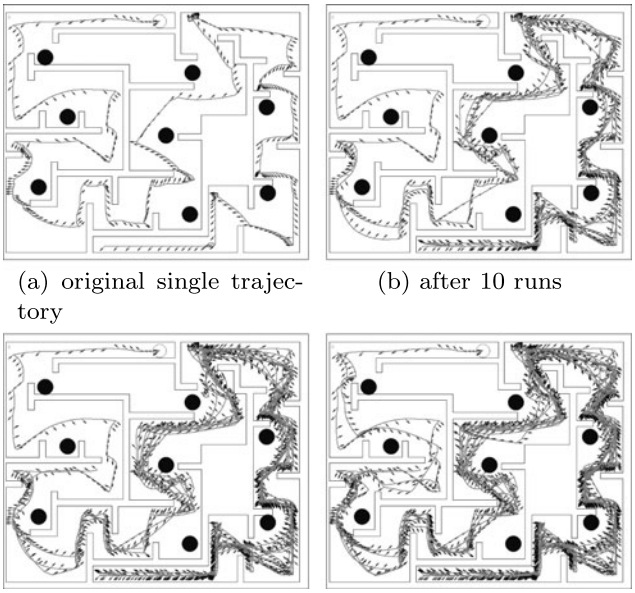
Fig. 18 Learning curves for simulated trials. The x-axis is the number of starts and the y-axis is the number of successes in 10 starts (optimal performance is a flat curve at 10). We restarted with a new (empty) library three times

3.3.2 Experiments

We performed initial simulation experiments on two different marble maze layouts (Fig. 17). The first layout (maze A) is a simple layout, originally designed for beginners. The second layout (maze B) is a harder maze for more skilled players. These layouts were digitized by hand and used with the simulator.

For maze A, we ran 100 consecutive runs to find the performance and judge the learning rate of the algorithm. During these runs, new trajectories were added as described above. After 100 runs, we restarted with an empty library. The results of three sequences of 100 runs each are plotted in Fig. 18(a). Almost immediately, the policy successfully controls the marble through the maze about 9–10 times out of 10. The evolution of the trajectory library for one of the sequences of 100 runs is shown in Fig. 19. Initially, many trajectories are added. Once the marble is guided through the maze successfully most of the times, only few more trajectories are added. Similarly, we performed three sequences of 150 runs each on maze B. The results are plotted in Fig. 18(b). Since maze B is more difficult, performance is initially weak and it takes a few failed runs to learn a good policy. After a sufficient number of trajectories was added, the policy controls the marble through the maze about 8 out of 10 times.

We also used our approach to drive a real world marble maze robot. This problem is much harder than the simula-



(a) original single trajectory (b) after 10 runs (c) after 30 runs (d) after 100 runs

Fig. 19 Evolution of library of trajectories. (The trajectories (*thick lines*) are shown together with their actions (*thin arrows*))

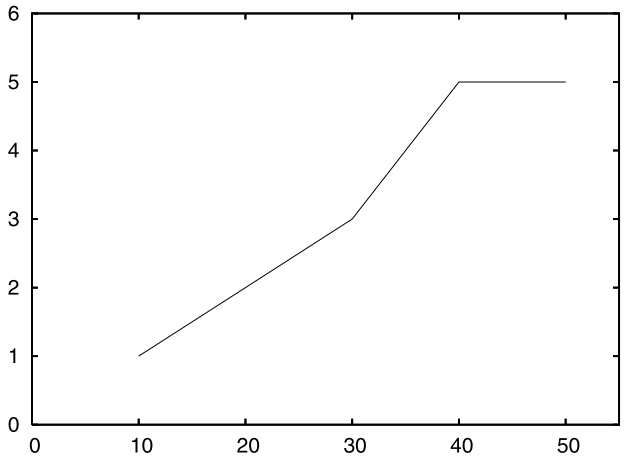
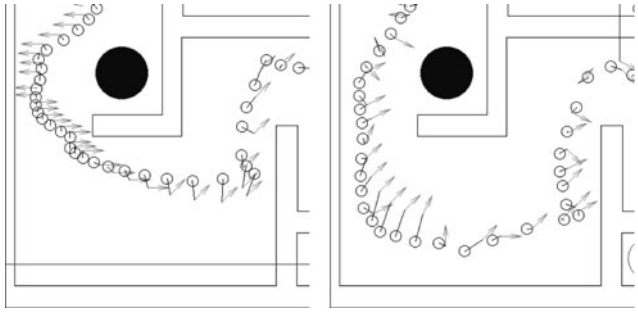


Fig. 20 Learning curve for trials on hardware for maze A. The x-axis is the number of starts and the y-axis is the number of successes in 10 starts

tion, as there might be quite large modeling errors and significant latencies. We used the simulator as the model for the A* planner. In the first experiment, we did not attempt to correct for modeling errors and only the simulator was used for creating the trajectories. The performance of the policy steadily increased until it successfully navigated the marble to the goal in half the runs (Fig. 20).

In Fig. 21 we show the trajectories traveled in simulation and on the real world maze. The position of the marble is plotted with a small round circle at every frame. The arrows, connected to the circle via a black line, indicate the action that was taken at that state and are located in the position for which they were originally planned for. Neither the velocity of the marble nor the velocity for which the action was originally planned for is plotted. Due to artificial noise in the simulator, the marble does not track the original trajectories perfectly, however the distance between the marble and the closest action is usually quite small. The trajectory library that was used to control the marble contained 5 trajectories. On the real world maze, the marble deviates quite a bit more from the intended path and a trajectory library with 31 trajectories was necessary to complete the maze.



(a) simulation with simulated noise (b) real world

Fig. 21 Actual trajectories traveled. The *circles* trace the position of the marble. The *arrows*, connected to the marble positions by a *small line*, are the actions of the closest trajectory segment that was used as the action in the connected state

Close inspection of the actual trajectories of the marble on the board revealed large discrepancies between the real world and the simulator. As a result, the planned trajectories are inaccurate and the resulting policies do not perform very well (only half of the runs finish successfully). In order to improve the planned trajectories, we tried a simple model update technique to improve our model, without much success.

Another factor that impacted the performance of the robot was the continued slipping of the tilting mechanism such that over time, the same position of the control knob corresponded to different tilts of the board. While the robot was calibrated at the beginning of every trial, sometimes significant slippage occurred during a trial, resulting in inaccurate control.

3.4 Case study: Little Dog

Another domain to which we applied the algorithm to is the Little Dog domain (Fig. 4) described in Sect. 1.2.2. Due to the complexity of the domain, we use a hierarchical approach to control the robot. At the high level, actions designate a foot and a new target location for that foot. A low level controller then controls the joints to move the body, lift the foot and place it at the desired location. The trajectory library operates at the high level and is responsible for selecting foot step actions, which makes this a library of abstract

actions (see Fig. 1(b)). Before explaining how the library works, we explain how the hierarchical controller works using a normal planner.

3.4.1 Planning and feedback control

In order to cross the terrain and reach a desired goal location, we use a footstep planner (Chestnutt et al. 2005) that finds a sequence of steps going to some goal location. The planner operates on an abstract state description which only takes into account the global position of the feet during stance. Actions in this planner merely designate a new global position for a particular foot. We use a heuristic method to compute continuous body and foot trajectories in global coordinates that move the dog along the footsteps output by the planner. We also implemented a safety monitor, which evaluates the robot's stance before taking a step and prevents unsafe steps to be taken. Details of the step execution and feedback control are described in the appendix of (Stolle 2008).

In principle, a single plan suffices for the robot to reach the goal location, especially since the low-level step execution always tries to hit the next global target of the swing foot in the plan, regardless of how well the previous step was executed. In practice, if there is significant slip, the next foot hold might no longer be reachable safely. This is detected by the safety monitor and replanning is initiated.

3.4.2 Trajectory libraries

By using a trajectory library, the robot can start with multiple plans and reduce the likelihood of needing to replan. Even with slips, it might be close enough to execute a step of a previous plan safely. Only if the robot detects that the step selected from all the steps in the library is unreachable does it have to stop the execution and replan. The new plan is then merely added to the library and, if necessary in the future, actions from previous plans can still be reused.

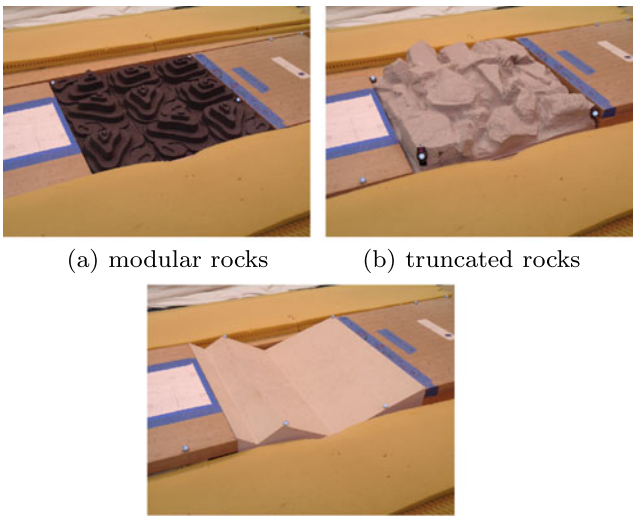
Similar to the marble maze, the robot uses a global state-based lookup into the trajectory library to select its next step. In the Little Dog case, this is the global position of the four feet. When started or after completing a step, the robot computes the position of the four feet in global coordinates. Then, for every step in the library, the sum of the Euclidean distances between the current position of the feet and their respective position at the beginning of the step is computed. The step with the minimum sum of the Euclidean distances is used as the next step to take. For large libraries, a kd-tree can be used to speed up this query. Since a plan in the Little Dog case consists of few, large steps, this was not necessary.

We also implemented a mechanism for improving the library based on experience. For this, we remember the last few steps taken. If one of the last steps taken from the library is selected again, we slow down the execution of this step. This is results in improving the execution of particularly difficult steps so that they are more likely to succeed. Additionally, we keep track of how many times a particular step was selected for a particular trial. If a step is ever taken more than some number of times, we add a penalty to the step which is added to its Euclidean distance when selecting future step. The penalty is chosen such that if the library is queried from the same state again, it will chose the second best step instead of the previously selected step. This also prevents the robot from attempting the same step indefinitely.

3.4.3 Experiments

We performed several experiments to gauge the effectiveness of the library approach. In these experiments, we compared using a single-plan sequential execution with replanning to using the trajectory library approach. In both cases, we use the same safety monitor to check the safety of the next step and replan as necessary. Experiments were run on three different terrains (see Fig. 22). For each terrain board, we started the robot in a number of different starting locations. As planning times of the footstep planner can vary wildly, even with nearly identical starting positions, we only look at the number of planning invocations and not at the time spent planning (total planning times were between 1/6th to 1/4th of the execution times for most boards. Due to peculiarities of the footstep planner, the total planning times for the slope board were between 1.2× to 2× the execution times). As the library approach keeps previous plans, we expect it to require fewer invocations of the planner.

Additionally, we also compare the time it takes for the robot to cross the board. As the library can slow down steps that are executed poorly and even avoid using particular



(a) modular rocks (b) truncated rocks

(c) slopes

Fig. 22 Terrains for gauging the advantage of the trajectory library

Table 1 Results comparing sequential execution and library execution on different terrains

Start	Sequential execution		Library	
	Plan. invocations	Execution time	Plan. invocations	Execution time
(a) modular rocks results				
1	5	30.2 s	2	25.8 s
2	3	27.4 s	1	25.8 s
3	3	25.3 s	2	25.2 s
4	1	25.0 s	1	23.5 s
5	4	23.0 s	1	22.3 s
sum	16	130.9 s	7	122.6 s
(b) truncated rocks				
1	1	28.2 s	1	28.9 s
2	6	30.7 s	5	27.4 s
3	6	31.2 s	1	27.1 s
sum	13	90.1 s	7	83.4 s
(c) slopes				
1	6	24.8 s	6	25.8 s
2	6	24.4 s	0	25.5 s
3	5	24.9 s	4	24.6 s
4	3	24.3 s	0	22.6 s
sum	20	98.4 s	10	98.5 s
(d) modular rocks results, different start states				
1	4	30.8 s	1	25.2 s
2	6	31.2 s	1	24.2 s
3	4	29.7 s	2	26.4 s
4	2	25.4 s	2	23.4 s
5	4	28.3 s	5	20.2 s
sum	20	145.4 s	11	119.4 s

steps, we expect the execution of the library to be slightly better.

As can be seen from the results in Table 1, using a trajectory library reduces the number of planning invocations necessary to cross a terrain by roughly a factor of two. Furthermore, execution speed alone, without considering planning time, is slightly improved. When examining the logging output, we found that the library based executions sometimes skips steps. This is possible because the planner is using a fixed sequence of feet in deciding which foot to move next. However, sometimes it is not possible to move a foot forward so it is stepped in place. In such a case, the trajectory library can skip a step. Also, it is possible that the robot slips forwards and a future step becomes executable early.

Finally, the start states used were chosen to allow for similar paths and hence synergies between plans. For the modular rocks board, we also picked a number of start states deliberately chosen to be far apart. Although the advantage of the library is less pronounced in this case when compared to the original step sequence (see Table 1d vs. Table 1a), there is a still a noticeably positive effect. When looking at the resulting library in Fig. 23, this is a result of the paths going over similar parts of the terrain, even when started from different start positions.

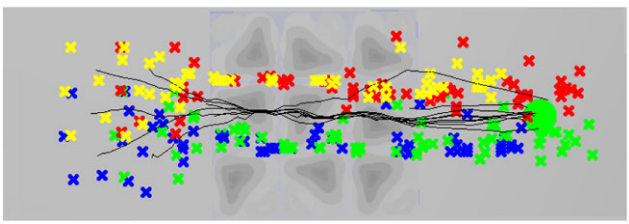


Fig. 23 The library after executing 5 starts (second experiment) for the modular rocks terrain

During these experiments, sometimes Little Dog catastrophically failed in executing a step and fell over. Neither replanning or a trajectory library can recover from this. In our experiments, this happened a small number of times and those runs were excluded. However, it is conceivable that

reflexes can be implemented on Little Dog that interrupt the execution of a step when failure is imminent and attempt to return the robot to a stable pose. Most likely, this would require replanning, or in the case of the library, picking a step that wasn't the original successor to the previous step. Using reflexes to recover from step execution failure will hence benefit from using a library approach.

3.5 Discussion

The main advantage of a trajectory library is a reduction in needed computational resources. We can create an initial policy with as little as one trajectory. In the case of the marble maze, we avoid computing global policies and instead use path planning algorithms for creating robust behavior without the possibility of replanning. In the case of Little Dog, replanning is possible but undesirable and the use of the library reduces the need for replanning. The larger the library is, the less likely replanning is necessary. Due to the setup of the Little Dog domain, the robot is run over a terrain multiple times and often similar paths are found by the path planner, leading to an effective use of previous paths. Similarly, in the marble maze the library becomes more robust the more paths have been added.

By scheduling the creation of new trajectories based on the performance of the robot or in response to updates of the model, policies based on trajectory libraries are easy to update. In particular, since the library can be continually improved by adding more trajectories, the libraries can be used in an anytime algorithm (Boddy and Dean 1994): while there is spare time, one adds new trajectories by invoking a trajectory planner from new start states. Any time a policy is needed, the library of already completely planned trajectories can be used. It is possible to plan multiple paths simultaneously for different contingencies or while the robot is executing the current paths. These paths can be added to the library and will be used as necessary. Planning simultaneously while executing the current plan leads to timing issues in the case of a sequential execution, as the robot will continue executing using the previous plan, while a new plan is created. When the new plan is finished, the robot is likely no longer near the start of the new plan and the problem of splicing the new plan into the current plan can be tricky. Especially in the Little Dog case, in our experience, the foot step planner can find very different paths even with similar starting conditions so it is possible that a plan started from near a state on a previous plan will be very different from the previous plan and it is not possible to start using the new plan by the time the planner has finished planning.

A strength of the library approach is the discrete storage of the policy through state-action pairs. This eases additions of more or less domain-specific learning from experience. For example, in both the marble maze and the Little Dog, we modify the Euclidean distance metric to allow for penalizing bad actions. This approach is likely possible in many domains. Additionally, in Little Dog we slow down steps that need to be repeated. A similar approach could be used in the marble maze by decreasing the force/tilt of an action if it led to failure.

One potential issue that can occur is a scenario in which the state of the system is the same distance from conflicting state-action pairs in the library. In this case, the tie can be broken arbitrarily. It is furthermore conceivable that the system continuously switches between state-action pairs from different trajectories that come from a different topological class (e.g. avoiding an obstacle by going left vs. by going right). In our experience, this has not been a problem. The marble maze is too dynamic of a task as to allow lingering in such a region for long. In the case of Little Dog, an action targets an absolute position as a foot hold, which directly reduces the distances to the next step from that same trajectory. This makes repeated switching between two plans from different topological classes unlikely and we did not observe this behavior in any of our experiments. A more principled approach on how to deal with conflicting data that could be applicable was recently described in (Chernova and Veloso 2008).

Compared to value function based methods for creating policies, trajectory planners have the advantage that they use a time index and do not represent values over states. Thus they can easily deal with discontinuities in the model or cost metric. Errors in representing these discontinuities with a value function can result in the divergence of DP algorithms. Additionally, no discretization is imposed on the trajectories—the state space is only discretized to prune search nodes and for this purpose a high resolution can be used. On the other hand, path planners use weaker reasoning than planning algorithms that explicitly take into account the stochasticity of the problem such as dynamic programming using a stochastic model. For example in the marble maze, a deterministic implementation of A* will result in plans that can go arbitrarily close to holes as long as they don't fall into the hole given the deterministic model that A* uses. In contrast to that, dynamic programming with a stochastic model will take into account the probability of falling into a hole and result in policies that keep some clearance. Unfortunately, dynamic programming using stochastic models is even more computationally intensive than dynamic programming with a deterministic model. Similar limitations apply to Markov Games, where some information becomes available only in future states and might depend on those future state. This information is not available at the time of planning and planners typically do not take into account potential, future information.

Compared to replanning without remembering previous plans, trajectory libraries not only save computational resources, but also have the advantage of being able to use

experience to improve the library. We did this both in the case of the marble maze as well as in the Little Dog case. In the first case, state-action pairs were penalized when they led up to failures. In the case of Little Dog, state-action pairs were penalized when they were invoked multiple times in a row. This results in a limited amount of exploration, as other state-action pairs with potentially different actions get picked in the same state in the future. However, when the planner is invoked, it might re-add the same action in the same state. In order to propagate the information from execution into the planner, one would have to update the model or maybe the cost function. Environments which require complex policies can also result in excessively large policies. For example, one could image arbitrarily complex environments in which small changes to the state require different actions. In this case, replanning might be more applicable as it is not affected by this complexity—it immediately forgets what the previous plan was.

Finally, it is unclear how to assess the quality of a library. Some distance-based metrics can be used to assess the coverage and robustness of the library: given some state, how far away is it from any state-action pair in the library? Given some radius around this state, how many state-action pairs can be found? (This could provide insights into robustness against perturbations.) Alternatively, one could assess the quality of the library by greedily following the reactive policy in simulation, assuming a deterministic model and comparing this to a plan from the exact query state. Some research into assessing the quality of a library of motions in the context of synthesizing animations can be found in (Reitsma and Pollard 2004). Finally, it is possible to imagine interference between two different trajectories which have different solutions (such as two different ways around an obstacle). However, in general picking an action from one trajectory will result in a state that's closer to states on the same trajectory. This is especially true in the Little Dog case where low-level controllers try to hit global foot hold locations. Using optimizers to improve the trajectories in the library will probably result in more congruent trajectories, too.

3.6 Summary

We have investigated a technique for creating policies based on fast trajectory planners. For the marble maze, experiments performed in a simulator with added noise show that this technique can successfully solve complex control problems such as the marble maze. However, taking into account the stochasticity is difficult using A* planners which result in some performance limitations on large mazes. We also applied this technique on a physical version of the marble maze. In this case, the performance was limited by the accuracy of the model.

In the case of Little Dog, we used a library in order to reduce the amount of replanning. Unlike the marble maze, it is possible to stop and replan in the Little Dog domain. However, it is desirable to reduce the time spent on replanning. Furthermore, by remembering steps from previous plans, experience from previous executions can be remembered in the library to improve future executions. This is not possible in the tabula rasa approach of planning from scratch every time.

It is worth noting a similarity between the policy representation introduced here and the intermediate policy used for transfer in the previous section. In both cases, we use a library of state-action pairs to find an action for a given state. However, the intermediate policy representation from the previous section is based on local state and is used to seed a new global-state based policy for a new task. On the other hand, the library from this section is based on global state and is only used directly as a policy and not for transferring to new tasks.

4 Transfer of policies based on trajectory libraries[3]

4.1 Introduction

The previous section introduced policies based on trajectory libraries, which perform lookups into a library of state-action pairs to determine the correct action to take in a given state (see Fig. 15). So far, these policies were created in-place by planners using the problem on hand.

We will now present work that allows transferring an existing library to a new problem, using ideas that we first used to transfer policies based on value function (see Sect. 2). There are several reasons why transferring libraries is useful. For one, as discussed in Sects. 3.3.1 and 3.5, the performance of a trajectory library generally improves the denser it is. Hence, starting with an empty library for a new problem is undesirable and being able to seed a library from previous experience is advantageous.

Another reason for transferring existing libraries to a new problem is that this allows us to consider new sources for state-action pairs in the library. Instead of only relying on planners that can be called on the problem at hand, we can now use domain-specific knowledge to create special purpose behaviors. Using these behaviors, we can solve problems that were previously unsolvable using existing planners.

4.1.1 Direct transfer

We propose to transfer library through the use of local features, the idea we successfully used in Sect. 2 to transfer

[3]Partially published in (Stolle and Atkeson 2007a).

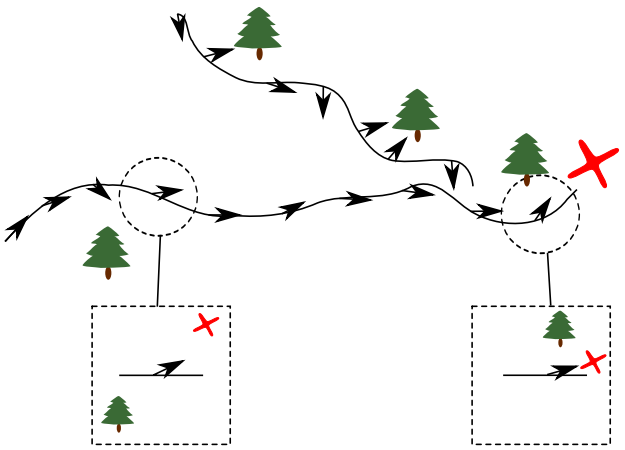


Fig. 24 Illustration of feature space. On the circled states as examples, we show how state-action pairs could be specified in terms of local features such as the relative positions of obstacles and the goal

policies based on value functions. When using features, instead of representing the state of the system using its default global representation, we use properties that describe the state of the system relative to local properties of the environment. For example in a navigational task, instead of using global Cartesian position and velocity of the system, we would use local properties such as location of obstacles relative to the system's position (Fig. 24). This allows us to reuse parts of a library in a new solution if the new problem contains states with similar features. To illustrate the idea of transferring libraries using local features, we present Algorithm 1.

This direct transfer algorithm is a fairly direct combination of the ideas from the previous two sections—using local features to transfer knowledge and using discrete state-action pairs as a control policy. This is expected to work in environments where we can encode enough global information into a local representation so that the agent can behave according to this local representation and succeed in obtaining the global goal. In our previous work on transferring value-based policies (see Sect. 2), we added a direction-to-goal feature to the local features for the marble maze in order to ensure goal directed behavior. Otherwise, goal directed behavior would not have been possible by just looking at nearby walls and holes.

4.1.2 Transfer with planning

In the case of more complex environments, such as Little Dog, it is not always possible to add such a local feature without solving the actual planning problem. While it is conceivable to add a direction-to-goal feature using a simplified planner, without taking into account actual foot placements it is very difficult to produce informative heuristic plans.

Algorithm 1 generic simple transfer algorithm

1: {S is the state space of the system.}
2: {A is the action space of the system.}
3: {F is a local feature space with $\mathrm{lf} : S \to F$ which maps a state $s \in S$ to a local feature vector $f \in F$. This function need not be invertible and may map different states to the same feature vector.}
4: {A_l is a local action space with $\mathrm{la} : S \times A \to A_l$ which maps action $a \in A$ taken in state $s \in S$ to a local action $a_l \in A_l$ and with $\mathrm{la}^{-1} : S \times A_l \to A$ which maps a local action a_l taken in state s back to a global action a.}
5: {L is a library of state-action pairs:}
6: $L := \{\langle s_1, a_1\rangle, \langle s_2, a_2\rangle \cdots \langle s_n, a_n\rangle\}$
7: {transfer library to local state representation:}
8: $L_l \leftarrow \bigcup \langle \mathrm{lf}(s_i), \mathrm{la}(s_i, a_i)\rangle \qquad \forall \langle s_i, a_i\rangle \in L$
9: $L_{new} \leftarrow \emptyset$
10: **for** $t \sim \mathrm{env}_{new}$ {sample target states, $t \in S$} **do**
11: {look up using local representation:}
12: $a_l \leftarrow L_l(\mathrm{lf}(t))$
13: {add action to new library:}
14: $L_{new} \leftarrow L_{new} \cup \langle t, la^{-1}(t, a_l)\rangle$

Note: both L and L_{new} contain state—action pairs $\langle s \in S, a \in A\rangle$, while L_l contains feature—local action pairs $\langle f \in F, a_l \in A_l\rangle$.

Instead of using a library to replace or reduce planning on a new problem, we would like to use it in conjunction with the planner, using the library to encode special-purpose behaviors for situations for which the planner cannot find suitable plans. This is an evolution of the direct transfer algorithm, extending the use of libraries from reactive policies to libraries of behaviors that augment a planner. When transferring such a library to a new problem, it contains *possibilities* for behaviors in different parts of the environment (see Fig. 1(c)). We integrate these into the planning process to find a goal directed sequence of actions, incorporating behaviors from the library only if appropriate (see Fig. 25). In order to plan with these behaviors, the library needs to also store the expected state resulting from executing a behavior from its start state. This transfer algorithm is described in Algorithm 2.

This algorithm has two key differences when compared to Algorithm 1. One key difference is that tuples in the library contain an expected result state. The other key difference is that instead of sampling the environment regardless of the available actions, we now perform a targeted search for states with suitable features in the new environment.

After having found appropriate places for the behaviors in the new environment, we employ a two-level planning algorithm (see Algorithm 3) to find a sequence of actions and behaviors that actually lead to the goal. This uses the library as a library of *possibilities* of behaviors, finds the behaviors useful for achieving a goal and incorporates them in a

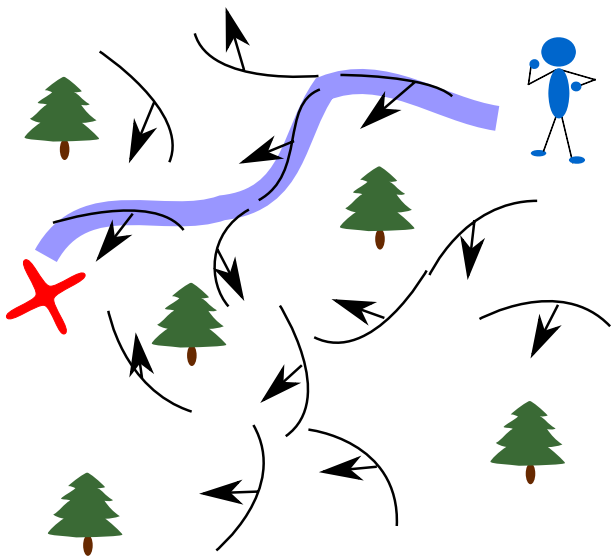


Fig. 25 Illustration of search through a trajectory library. For the given start state, we find a sequence of trajectory segments that lead to the goal

Algorithm 2 generic behavior transfer algorithm

{S, A, F with lf and A_l with la are defined as in Algorithm 1}
{S_l is a local state space with $\text{ls} : S \times S \to S_l$, mapping state $s' \in S$ to local state $s'_l \in S_l$ based on reference state $s \in S$ and with $\text{ls}^{-1} : S \times S_l \to S$, mapping local state $s'_l \in S_l$ back to a global state $s' \in S$ based on a new reference state $s \in S$.}
{L is a library of state-action-state tuples:}
$L := \{\langle s_1, a_1, s'_1\rangle, \langle s_2, a_2, s'_2\rangle \cdots \langle s_n, a_n, s'_n\rangle\}$
{transfer library to local state representation:}
$L_l \leftarrow \bigcup \langle \text{lf}(s_i), \text{la}(s_i, a_i), \text{ls}(s_i, s'_i)\rangle \qquad \forall \langle s_i, a_i, s'_i\rangle \in L$
$L_{new} \leftarrow \emptyset$
for all $\langle f, a_l, s'_l\rangle \in L_l$ **do**
  {find target states with similar features:}
  $T \leftarrow \{t \in \text{env}_{new} \mid \text{lf}(t) \approx f\}$
  {transfer behavior to target state and add to new library}
  $L_{new} \leftarrow L_{new} \cup \bigcup \langle t, \text{la}^{-1}(t, a_l), \text{ls}^{-1}(t, s'_l)\rangle \qquad \forall t \in T$

Note: both L and L_{new} contain state-action-state tuples $\langle s \in S, a \in A, s \in S\rangle$, while L_l contains feature-local action-local state tuples $\langle f \in F, a_l \in A_l, s_l \in S_l\rangle$. For brevity, we will refer to the state-action-state tuples as behaviors b with initial state $s(b)$, action $a(b)$ and expected result state $s'(b)$.

plan. The final plan consists of sequences of actions from the planner ($plan(b_i, b_j)$) and actions or behaviors from the transferred library ($a(b_i)$). After the Related Works section, we show how we implemented two variants of this algorithm

Algorithm 3 planning through transferred library

{create high-level graph:}
$G := \langle V, E\rangle$
{vertices are behaviors from L_{new} and empty behaviors for start $b_s = b(s_s)$ and goal $b_g = b(s_g)$ s.t. $s(b_s) = s'(b_s) = s_s$ and $s(b_g) = s'(b_g) = s_g$}
$V := \{b(s_s), b(s_g)\} \cup \{b \mid b \in L_{new}\}$
{created edges between near behaviors:}
$E := \{\langle b_i, b_j\rangle \mid \|s'(b_i), s(b_j)\| < c\}$
{we invoke the original planner to compute plans and weights for each edge}
for all $\langle b_i, b_j\rangle \in E$ **do**
  $\text{plan}(b_i, b_j) \leftarrow \text{planner}(s'(b_i), s(b_j))$
  $w(b_i, b_j) \leftarrow \text{cost}(\text{plan}(b_i, b_j))$
{perform best-first-search to find a plan through G}
$\text{plan}_G \leftarrow$
  $(\langle b_s, b_2\rangle, \langle b_2, b_3\rangle \cdots \langle b_{n-1}, b_g\rangle) = \text{BFS}(G, b_s, b_g)$
{convert plan into sequence of actions and behaviors}
$\text{plan}_{final} \leftarrow$
  $(\text{plan}(b_s, b_2), a(b_2), \text{plan}(b_2, b_3), a(b_3), \ldots$
  $\ldots, a(b_{n-1}), \text{plan}(b_{n-1}, b_g))$

to Little Dog. Due to research priorities, no case study was done on the marble maze, which would otherwise lend itself for a case study on the "Direct Transfer" algorithm.

4.2 Related work

The work presented in this section, like the work on transferring policies using a generalized policy iteration dynamic programming procedure in Sect. 2, deals with transfer learning. As such, much of the related work in Sect. 2.2 applies here as well. However, since trajectory libraries are explicitly represented as state-action pairs, it is much simpler to express them in a feature space than the value function based policies from our previous work.

In addition to the aspect of transferring knowledge, the aspect to learn from demonstration to solve difficult controls problem is related to (Bentivegna et al. 2006). It explores learning from observation using local features, and learning from practice using global state on the marble maze task. Our approach to learning from demonstration takes a more deliberate approach, since we perform a search after representing the learned knowledge in a local feature space. More recent work on using human input to not only learn, but also effectively improve control knowledge is presented in (Argall et al. 2008).

Related work on Little Dog that uses input from a human teacher to learn value functions for planning based on local features can be found in Kolter et al. (2008). More similar related work which learns policies for the AIBO quadruped robot directly from demonstration using function approxi-

Algorithm 4 Little Dog transfer and planning

Transfer library according to Algorithm 2. Find similar states (line 10) as follows:

- create height profile for every step
- create height profile for a sampling of positions and orientation on the new map
- for each step, find best match; discard if best match is worse than some threshold.

Find appropriate steps to get from start state s_s to the goal state s_g (cf. Algorithm 3)

- add two behaviors, b_s and b_g with $s(b_s) = s'(b_s) = s_s$ and $s(b_g) = s'(b_g) = s_g$
- $\forall b, b'$, find the Euclidean foot location metric between $s'(b)$ and $s(b')$
- Define the successors of b, $succ(b)$ to be the n b' with the smallest distance according to the metric
- Create footstep plans between all $s'(b)$, $s(b')$, s. t. $b' \in succ(b)$
- Perform a Best-First-Search (BFS) through the graph whose vertices are the footsteps b and directional edges are defined from $b \rightarrow b'$ whenever $b' \in succ(b)$
- The final library consists of all steps b on the path determined by the BFS as well as all generated footsteps by the footstep planner on that path.

mation mapping local features to actions was performed in (Grollman and Jenkins 2008).

4.3 Case study: Little Dog

The domain to which we applied the algorithm is the Little Dog domain (Fig. 4) described in Sect. 1.2.2. As described in the previous section, we can use a footstep planner (Chestnutt et al. 2005) that finds a sequence of steps going to some goal location. We then use a heuristic method to compute body and foot trajectories that move the dog along the footsteps output by the planner. On some difficult terrains, we are unable to create good sequences of foot steps that can be executed by the heuristic foot step execution.

In order to increase the capability of the robot on difficult terrain, we use learning from demonstration to navigate the robot across difficult terrain. We use a joystick together with inverse kinematics to manually drive the robot across the terrain and place feet. Sequences of joint angles together with body position and orientation are recorded and annotated with the stance configuration of the robot. The stance configuration describes which legs are on the ground and which leg is in flight. Once the robot has been driven across the terrain, the data can be automatically segmented into individual footsteps according to the stance configuration: a new step is created every time a flight foot returns onto the ground and becomes a stance foot. As a result, every step created in this way starts with a stance phase where all four feet are on the ground. If necessary, we can optionally suppress the automatic segmentation.

Once the steps have been segmented, they can be used immediately on the same terrain without transfer, using the trajectory library approach from the previous section. Steps are added to the trajectory library with the global position of the four feet, as recorded at the beginning of a step, as index. Recorded steps can also be mixed with heuristic steps coming from the planner. By using a trajectory library to pick which step to take, the robot can succeed in traversing a terrain even if it slips or if it is just randomly put down near the start of any step. When a recorded step is picked from the library, we play back the recorded joint angles. Additional details, including information about the feedback controllers used and how smoothness constrains are obeyed can be found in (Stolle 2008).

4.4 Library Transfer 1

4.4.1 Description

Clearly, indexing into the policy based on a global state description such as global position of the feet, limits it to a particular task. If the terrain is moved slightly, the steps will be executed incorrectly. Furthermore, if parts of the terrain have changed, the policy cannot even adapt to such a simple change. In order to solve these problems, we use the algorithm for transferring trajectory libraries to different environments. The transfer algorithm has to recognize appropriate terrain for applying the demonstrated steps successfully and effectively. The source of the transfer are multiple trajectories which were recorded when the robot traversed different terrain boards in varying directions.

Referring back to the algorithm definition in Sect. 4.1.2, the original state space S is the same as used in Sect. 3.4.2: the global 2d-position of the 4 feet (8-dimensional vector). The actions space in this case is the discreet set of recorded behaviors, each consisting of a sequence of recorded joint angles and body poses. The local feature vector f is created from the local height profile (Fig. 27) for each step. The feature-vector is hence a vector of heights: $f = \langle h_{1,1}, h_{1,2}, \ldots, h_{i,j}, \ldots, h_{m,n} \rangle$. The origin of the local frame (Fig. 26) for the profile is the centroid of the global foot positions at the beginning of the step. The x-axis is aligned with a vector pointing from the XY-center of the rear feet towards the XY-center of the front feet. The z-axis is parallel to the global z-axis (aligned with gravity). The height of the terrain is sampled at 464 positions on a regular grid (0.35 m × 0.20 m with .012 m resolution) around this origin to create a length 464 vector. The grid is normalized so that the mean of the 464 entries is zero. The functions ls

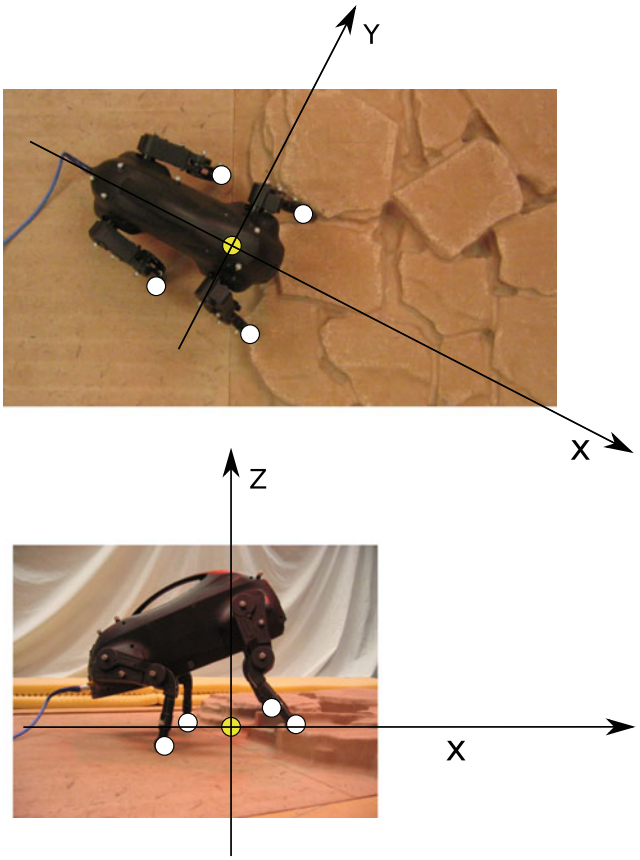
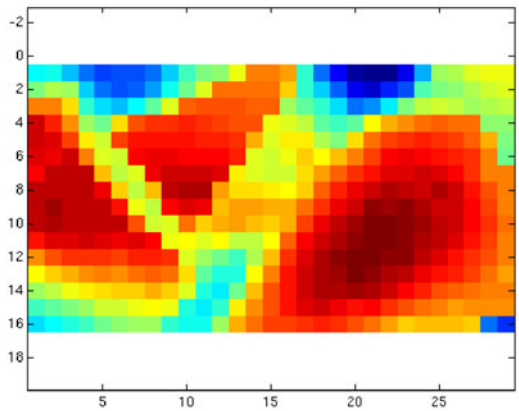


Fig. 26 Local frame

(a) example pose



(b) resulting heightmap

Fig. 27 Local heightmap

and la, used for mapping actions and states into local action- and state-spaces are also defined using this local frame: The joint angles in a are preserved as is, while the body poses are transformed into the coordinate system of the local frame. In the same way, the function ls maps a global pose into the local coordinate frame.

For finding states with similar features in a new environment, we create local terrain descriptions for a sampling of all possible positions and rotations around the z-axis on the new terrain. The rotations around the z-axis are limited to rotations that have the dog pointing roughly to the right ($\theta \in [-\pi/4, \pi/4]$). For every step in the library, we then find the local frame on the new map that produces the smallest difference in the feature vector. If this smallest difference is larger than some threshold, the step is discarded. The threshold is manually tuned to ensure that steps do not match inappropriate terrain.

For performance reasons, after creating the feature vectors for the matching of steps, we used principal component analysis (PCA) to project the vectors into a lower dimensional space. The PCA space was created beforehand by creating feature vectors for one orientation of all the obstacle boards we had. The first 32 eigenvectors, whose eigenvalues summed to 95% of the total sum of eigenvalues, were chosen as the basis for the PCA space (see Fig. 28 for the cumulative energy of the first 100 eigenvectors).

Once all steps have been discarded or translated to new appropriate positions, we perform a search through the library, as described in Sect. 4.1.2. Due to the relocation, there is no guarantee that the steps still form a continuous sequence. Depending on the size and diversity of the source library, the steps of the new library will be scattered around the environment. Even worse, some steps might no longer be goal directed. The steps now represent capabilities of the dog. In places where a step is located, we know we can execute the step. However, it is unclear if we should execute the step at all or in what sequence. We solve this problem by performing a search over sequences of steps. In order to connect disconnected steps, we use a footstep planner (Chestnutt et al. 2005). Given the configuration of the robot at the end of one step and the beginning of another step, the footstep planner can generate a sequence of steps that will go from the first to the latter. The same heuristic walking algorithm as in the previous section was used for controlling the body

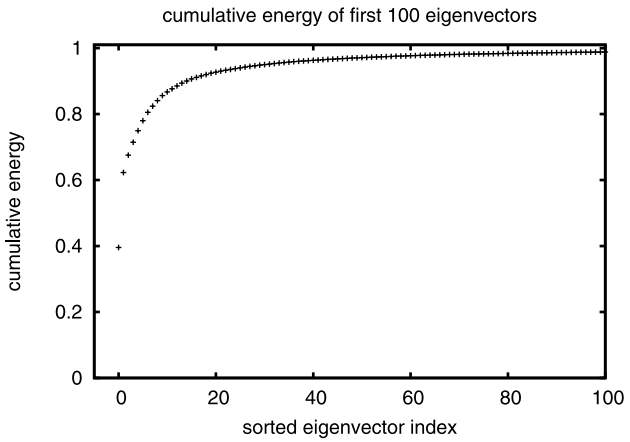


Fig. 28 Cumulative energy of the first 100 eigenvectors

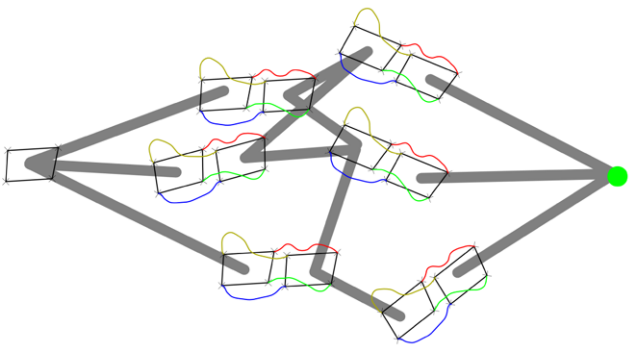


Fig. 29 Illustration of topological graph after transfer of library. Each behavior is a node in the topological graph. The connections (*gray lines*) are made using the foot step planner

and the actual leg trajectories while executing footsteps from the footstep planner.

For the search, we generate a topological graph (see Fig. 29). The nodes of the graph are the start state and the goal state of the robot, as well as every step in the transferred library. Edges represent walking from the end of the pre-recorded step represented by the start node to the beginning of the pre-recorded step represented by the target node. The cost of every edge is roughly the number of additional steps that have to be taken to traverse the edge. If the foot locations at the end of the source pre-recorded step are close to the foot locations at the beginning of the target pre-recorded step of the edge, no additional steps are necessary. In order to know the number of additional steps, the footstep planner is used at this stage to connect the gaps between steps when we generate the topological graph. Since the steps that are output by the planner are considered risky, we assign a higher cost to planned steps. (If the planner created reliable steps, we could just use the planner to plan straight from the start to the goal.) In order to reduce the complexity of the graph, nodes are only connected to the n-nearest steps based on the sum of Euclidean foot location difference metric. We then use a best-first search through this graph to find a sequence of footstep-planner-generated and pre-recorded steps. This sequence is added to the final library.

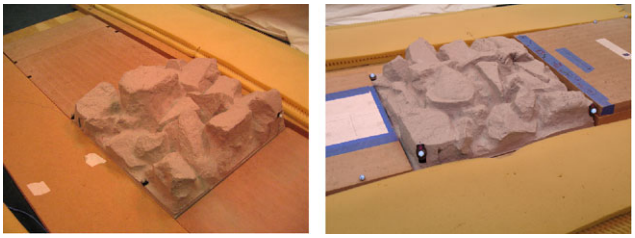


Fig. 30 Terrains used to create trajectory library

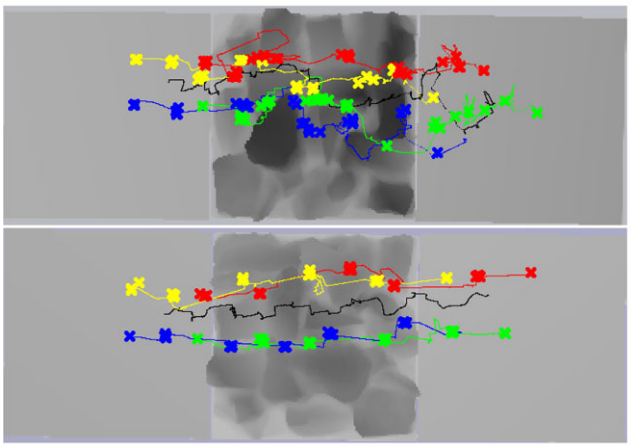


Fig. 31 Excerpts from the trajectory library. *Lines* show the actual trajectories of feet and body (cf. Fig. 5). The dog moves from left to right

4.4.2 Experiments

We performed several experiments to verify the effectiveness of the proposed algorithms. For all experiments we started with 7 libraries that were created from two different terrains. Using a joystick, one terrain was crossed in four different directions and the other terrain was crossed in three different directions (two examples can be seen in Fig. 31). The combined library contained 171 steps.

In order to test transfer using terrain features, we first looked at transferring the steps from these seven libraries to one of the original terrains. In theory, the steps from the library that was created on the same terrain should match perfectly back into their original location. Some spurious steps from the other terrains might also match. This is indeed the case as can be seen in Fig. 32. The spurious matches are a result of some steps walking on flat ground. Flat ground looks similar on all terrains.

When modifying the terrain, we expect the steps to still match over the unchanged parts. However, where the terrain has changed, the steps should no longer match. For this experiment we modified the last part of a terrain to include

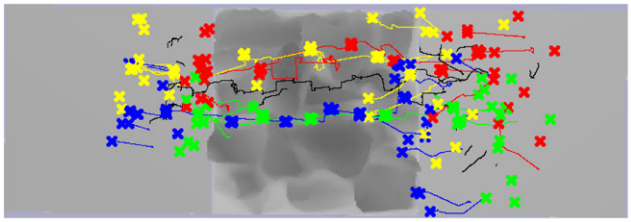


Fig. 32 Library matched against one of the source terrains. Many steps match the flat parts and the steps created from crossing this terrain match back into their original location as seen by black body-trace crossing the terrain. (Some crosses do not have foot-traces extending from them, since they are the starting location for a foot from a step where one of the other three feet was moving)

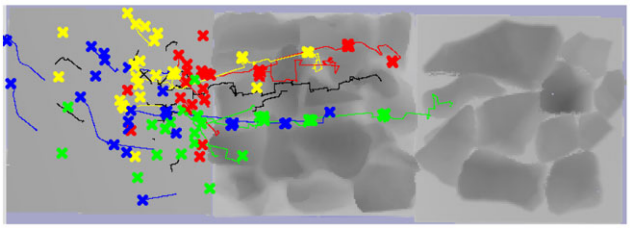


Fig. 33 Library matched against new, modified terrain. No steps match the completely new terrain on the right

new rocks instead of the previously flat part (Fig. 33). The matching algorithm correctly matches the steps that are possible and does not incorrectly match steps on the modified parts.

While the matching correctly identifies where to place steps in the library, the resulting library needs to be improved, as anticipated. There are large gaps between some steps. Moreover, some spuriously matched steps do not make progress towards the goal but can lead the robot away from it, if they happen to be matched greedily. We now use the search algorithm described earlier to postprocess the resulting library. The resulting plan should select the right steps, throwing out the spurious matches. Furthermore, by invoking the footstep planner to connect possible steps together, it will also fill in any gaps. This happens correctly for the modified map (Fig. 34).

Finally, in order to validate the transfer algorithm, we executed the resulting library on the terrain with the modified end board. A plan, from a slightly different start location but otherwise identical to Fig. 34, was executed on the robot and the robot successfully reached the goal, switching between steps from the source library that were created by joystick control and the synthetic steps created by the footstep planner (Fig. 35).

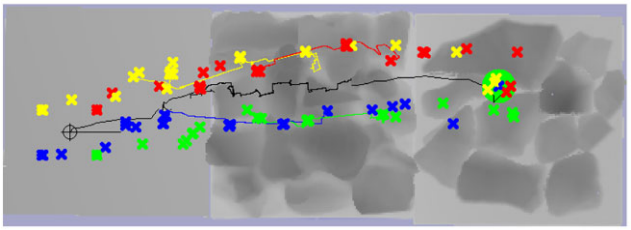
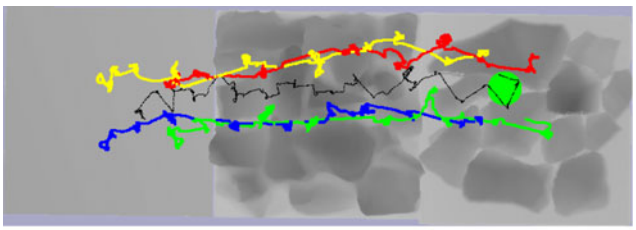


Fig. 34 Result of searching through the library on modified terrain with the green spot as the goal. For a particular start and goal, only the relevant behaviors are used and additional steps created by the footstep planner to the left and right of the familiar rock terrain. (The steps coming from the footstep planner do not show traces from the starting places of the feet (crosses), since the foot trajectories are generated on the fly during execution. The body trajectory for planned steps are only hypothetical trajectories—the on-line controller is used for the actual trajectories during execution)



(a) Trace from the actual execution. *Lines* show the actual trajectories of feet and body (Fig. 6).

(b) Picture from the same execution

Fig. 35 Little Dog executing the plan

4.5 Library Transfer 2

4.5.1 Description

The previous algorithm has a number of limitations that we worked on removing in an alternate transfer algorithm. The improvements were made to specific details of Algorithm 2, in particular the feature generation lf, the search for new states with appropriate features for a behavior (line 10) and the function mapping local actions back to global actions (la^{-1}).

The search for suitable states in the previous algorithm was limited to one match for each state-action pair in the library of behaviors. In the new algorithm, we allow multiple matches of a step in different parts of the terrain. After

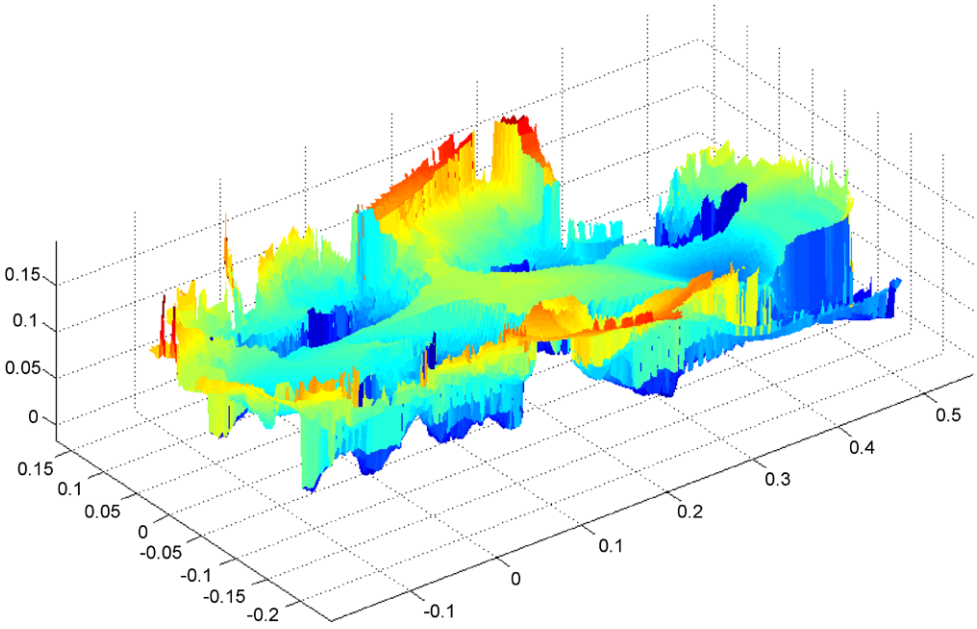


Fig. 36 Example swept volume for crossing a barrier. The robot moves from left to right. This pictures shows for every point the lowest any part of the robot has been at that point while climbing over a barrier. The smooth surfaces in the center are from the bottom of the trunk. The protrusions downwards on both sides are caused by the legs and feet (which touch the ground while in support). Notice that near $x = 0.2$ m, no part of the robot is very low. This is where the barrier was placed

a match has been found, we exclude a region around the match from being matched in the future. We then look for additional matches, excluding regions around each successful match, until the PCA error becomes larger than some threshold.

A more significant change was done in the feature generation and matching algorithm itself. In the first algorithm, matching was done purely based on sum-of-squared error of the PCA feature vectors, which does not take into account the properties of a particular step. In particular for Little Dog, it is important that the terrain supports the stance feet and that neither the body nor the flight foot collide with the terrain. Hence, there are certain variations of the terrain (lower terrain in parts where the stance feet are not supported or higher terrain in parts which are not occupied by any part of the robot) that can be tolerated easily. On the other hand, if the terrain changes under the feet or changes so that parts of the body would collide, the match should no longer be allowed. In order to prevent matches in such cases, the cut-off on the PCA error has to be very low in the first algorithm. This precludes matching even if the terrain is different in tolerable ways.

In this second algorithm, we added additional checks after a PCA match to verify that a particular match is possible and does not result in collisions. This allows the use of a lower dimensional PCA representation with a higher error cut-off, as the PCA matching only has to recognize where the terrain has similar characteristics to the original training terrain. The PCA error is no longer used to judge if the relocation is valid.

Instead, the new algorithm uses a tiered approach to check the validity of a possible location. The first check is based on foot locations and is responsible to ensure that all feet are supported by the terrain. Before transferring a step, we compute the location of touchdown and lift-off of all feet during the step in the coordinates of the local frame of the step (see Fig. 26). We then look up the height of the terrain under the foot and compute the height of the foot over the terrain. When checking a new location, we again compute the height under each foot, placing the local frame at the candidate location, and make sure that the height of the foot over the terrain does not change more than some threshold. This can be computed very quickly, as only a small number of foot locations have to be checked.

The foot check is designed to ensure that the feet have the necessary ground support, however it does not check if the body would come in collision with the terrain. A second, more expensive check is performed to check for collisions if the foot check succeeded. To check for collision, before performing any matches, we compute the swept volume that the body sweeps through space during a step (see Fig. 36). We also compute the clearance (vertical distance) of the swept volume to the terrain. Due to inaccuracies in models and positioning, some parts of the swept volume can have a negative clearance. When checking a possible match, we recompute the clearance in the new location. If no part of the swept volume has a clearance that is worse than the smaller of the original clearance or zero, by some threshold, we allow the match. Otherwise the match is rejected.

As described in the previous section, the PCA matching is performed on a sampling of possible new reference frames for the step. For practical reasons, the resolution of the PCA samples can be too coarse to find a good match. In order to increase the resolution of the possible matches, we perform a local search in the vicinity of the PCA match. In

this local search, we search a range of positions and orientations (constraint to rotations around Z) around the original PCA match. For each possible placement of the reference frame, we first perform the foot check. Then we perform the swept-volume check on the best matched frame based on the foot check error. In case of success, the step is immediately matched.

In case of a failure, it is still possible that a match in the vicinity of this PCA match is possible, but that it was not found because the foot error was not informed enough to find this location. For example, if an obstacle is surrounded by flat areas and the feet are only placed on the flat areas, many possible locations will have a good match based on the foot error, but they might still contain collisions with the obstacle. We look at the variance of the foot errors to determine if the foot errors were informative for finding a possible match. If the foot check has high variance, the match based on the foot check is considered informed and should have found terrain similar to the original terrain and the failure is final. However, if the foot check error had little variance, it is possible that the best match based on the foot error was not well informed. As a result we again search over nearby positions and orientations using the collision check instead of foot checking. If the location with the smallest clearance violation is above the collision threshold, the failure is final again. Otherwise the step is relocated to this location. For a concise description of the second matching algorithm, see Algorithm 5. The addition of foot-height checks and swept-volume checks are essentially additions to the feature space F and appropriate changes to the similarity metric used for finding similar states.

In addition to the changes to the search and feature space which allow matches to locations where the terrain is different in such a way that the unmodified behavior will succeed, we also wanted to increase the power of a library of behaviors by allowing simple modifications to the behaviors. This is essentially a more advanced version of la^{-1}. In particular, we allow each stance foot to move up or down relative to its original location by a limited amount. The amount by which each stance moves is determined by minimizing the foot error metric. However, moving the stance of a foot does not guarantee the elimination of foot stance error: the foot stance error is computed based on the position of the foot relative to the terrain at touch down as well as lift off. Due to roll of the foot and possible slipping, these two locations are not usually the same. However, if a stance is moved, both lift off and touch down are moved together. If the terrain under the foot in the lift off position is lower than in the original location, but the terrain at the lift off location is higher, moving the stance cannot reduce both errors.

Once a delta for every stance throughout a behavior is determined, the behavior is modified as follows: Recall that a behavior is specified by a trajectory of desired body positions and desired joint angles. In order to apply the stance deltas, this information is used to compute desired foot positions in the local reference frame of the step. The stance deltas are interpolated through time by creating 1-d cubic splines of deltas between stances (from lift-off to touch-down). Then the interpolated deltas are applied to the trajectories of the desired foot positions and new joint angles are computed through inverse kinematics.

A major hurdle in implementing the modifications into the transfer algorithm is the collision check. Previously, a particular behavior had one swept volume that could be precomputed. However, now every possible location in the local search can potentially have a difference swept volume, as the step is modified to adapt to the terrain. In order to make collision checking practical, we discretized the deltas for each stance to some resolution. We then compute swept volumes separately for the body and each stance-to-stance segment for every foot. Given a delta configuration, we can quickly assemble a complete swept volume by computing the union of these mini-volumes. By caching swept volumes of stance-to-stance segments, we avoid recomputing of previously computed stance-to-stance swept volumes for the feet. Furthermore, we also cache complete swept vol-

Algorithm 5 Little Dog alternate transfer

Transfer library according to Algorithm 2. Find similar states (line 10) as follows:

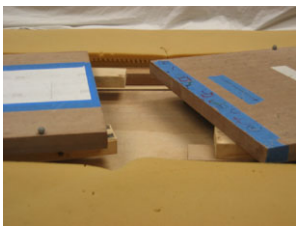
- create height profile for every step, project into PCA space
- compute foot heights for touch-down and lift-offs
- create swept volume and clearance
- create height profile for a sampling of poses (P_{pca}) on the new map

for all steps **do**
  $P_{\text{cur}} \leftarrow P_{\text{pca}}$
  for all $p_{\text{pca}} \in P_{\text{cur}}$ sorted by PCA error **do**
    if pca_error(p_{pca}) > threshold **then**
      break
    $P \leftarrow \text{nearby}(p_{\text{pca}})$
    $p* \leftarrow \arg\min_{p \in P} \text{foot\_error}(p)$
    if foot_error($p*$) < threshold **then**
      if clearance($p*$) > threshold **then**
        relocate_step($p*$)
        $P_{\text{cur}} \leftarrow P_{\text{cur}} - \text{nearby\_large}(p*)$
      else
        if variance(foot_error($p \in P$)) < threshold **then**
          $p* \leftarrow \arg\max_{p \in P} \text{clearance}(p)$
          if clearance($p*$) > threshold **then**
            relocate_step($p*$)
            $P_{\text{cur}} \leftarrow P_{\text{cur}} - \text{nearby\_large}(p*)$

(a) gap (22cm wide) (b) barrier (10cm high)

Fig. 37 Terrains for gauging transfer with modifications

Fig. 38 Simple modification to gap



umes for a full configuration of discretized stance deltas. This cache is implemented as a Trie with the delta configuration as index. This allows essentially constant time access to a previously computed swept volume for a particular discretized delta configuration with a lower overhead than a hash table, since no hashes have to be computed. Due to this aggressive caching, we can compute collision checks for modified behaviors with little penalty over non-modified steps.

4.5.2 Experiments

In order to gauge the effectiveness of the new transfer algorithm, we choose two kinds of terrains that can be easily modified, both in ways that don't require changes to the step and in ways that do require changes to the step: a large gap and a tall barrier (see Fig. 37). Currently, our planning algorithms cannot cross the gap or reliably cross the barrier. In particular, in order to cross the gap, the robot has to fall forward onto a leg—a behavior that the planner cannot currently plan for. Using learning from demonstration, the robot can cross these terrain boards. In order to demonstrate simple transfer that does not require modification of the original behavior, we modify the terrains by rotating the flat boards for the gap terrain (see Fig. 38) and changing the barriers (see Fig. 39).

When running the transfer algorithm on the modified terrains, multiple matches are found, as desired, for both the gap (Fig. 40) as well as the different kinds of barriers (Fig. 41). After performing the high level search through the transferred library, as described in Sect. 4.4, the robot successfully executes a path made from heuristic steps and the behaviors matched on the terrain (Figs. 42, 43).

In a second test, we also modified the terrain in ways that required the behavior to adapt. For this, we raised some of

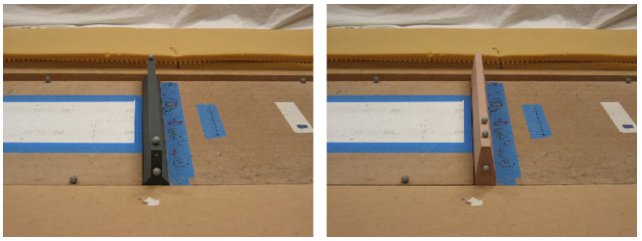
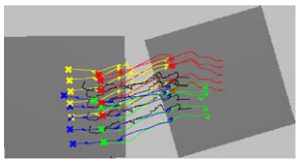


Fig. 39 Simple modifications to jersey barrier (different shapes, widths and heights)

Fig. 40 Matches on simple gap modification



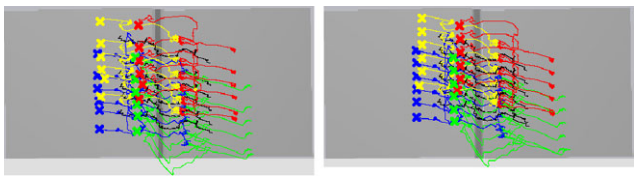


Fig. 41 Matches on simple jersey barrier modification

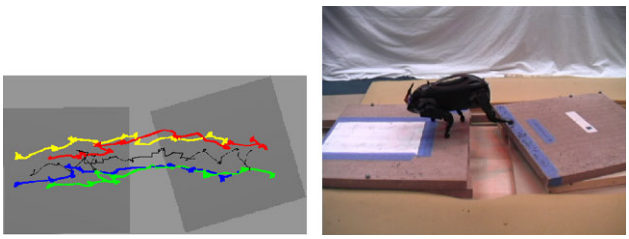


Fig. 42 Little Dog crossing large gap

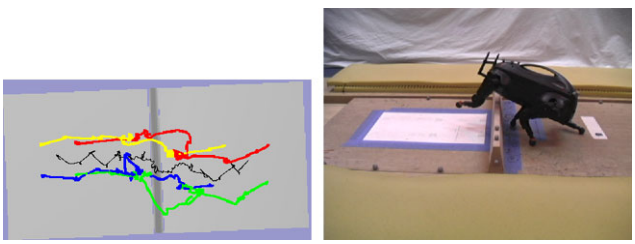
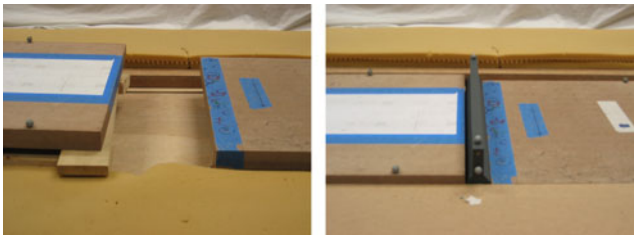


Fig. 43 Little Dog climbing over barrier

the terrain as seen in Fig. 44. The new algorithm which allows modifications to the stored behavior again matches the terrain as expected and the robot executes the modified behavior successfully, both in the case of the large, modified gap (Fig. 45) and the tall barrier (Fig. 46).



(a) modified gap (b) modified jersey barrier

Fig. 44 Modifications that require changes to the behavior

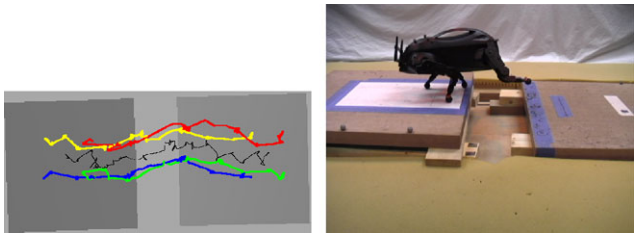


Fig. 45 Little Dog crossing large gap with height difference using modified behavior

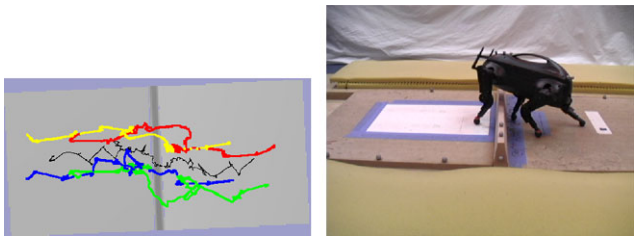


Fig. 46 Little Dog climbing over barrier with height difference using modified behavior

4.6 Discussion

We have devised and implemented two algorithms for transferring stored behaviors to new, modified terrain. The first algorithm showed that it was possible to use terrain features to recognize when a step is applicable and where a step is not applicable. Combined with a high-level search over the matched steps, a sequence of heuristic, planned steps and demonstrated behaviors was used to reach the goal. However, the first algorithm could match steps only once and did not take into account the properties of a particular step. Its transfer potential was limited.

We then introduced a second algorithm that improves on the first algorithm in a number of ways. First of all, it allows a step to be matched multiple times. Because it explicitly checks for the applicability of a behavior based on foot support and collision-freeness, the terrain can be matched more liberally. However, in our experiments, we found it difficult to tune the parameters of the top-level PCA matching. In analyzing the principal components, it became clear that PCA might not be suitable to pick up the kind of terrain features that make a behavior "applicable" in a situation. In particular, in the case of the jersey barrier, the first few principal components with the highest energy resulted, when reprojected into the original space, showed an undulating terrain with no single "barrier" visible. Different low-level representations such as independent component analysis or wavelet decompositions could be explored as alternative representations for picking out "characteristics" that make a particular behavior appropriate.

A limitation of both algorithms is that given a start state, the search through the transferred library only yields a single trajectory. In order to increase the number of trajectories in the final library, one could perform multiple searches from different start states. Alternatively, a backwards search from the goal could be performed and the complete search tree added to the library. Finally, instead of searching once, it is possible to continuously search through the library during execution. Since the search is on a topological graph, this search would be much faster than the search performed by a path planning algorithm in the continuous state space. The gaps between steps are already filled in when creating the topological graph and do not have to be replanned during the continuous search process.

A more radical departure from the current algorithm would be to do away with explicitly finding global states where the features of the state-action pairs from the original library match. Instead, one could greedily match actions from the library based on local features of the start state and its vicinity. After executing the action, this can be repeated. Applying the library greedily based on local features does not allow for searching and might result in dead-ends. Also, it will not allow the robot to cross large gaps in the library if it is not in the vicinity.

Alternatively, one could search for a sequence of steps leading towards the goal, performing a local search at every expansion to find one or more suitable successor steps in the vicinity of the termination of the previous step. However, this will not work if the local searches fail to find matching steps because of gaps—large areas where no steps in the library match. One could extend the local search area until, in the limit, the complete relevant state space is searched at every expansion. This would essentially be the algorithm that is presented here.

Both algorithms have in common that the computationally expensive operations of computing local features, matching and collision checking are performed as an offline preprocessing step. The next step of planning through the transferred library of behaviors is also done before execution, but might need to be redone for multiple executions from different start states. Finally, during execution, one is just following a sequence of steps or, if multiple plans are available, performs state-based look ups in a trajectory library fashion (see Sect. 3). If a nearest-neighbor look up is used, this can be performed quite fast.

While we believe it is possible to transfer behaviors in a large variety of domains, it might not always be advantageous to do so. In particular, when it is not possible to create useful local features, inappropriate transfer can happen where a behavior is transferred to a location where it is not appropriate to execute that behavior. Furthermore, the behaviors that are being transferred need to add additional capabilities to the system. If the behavior can be created from the action choices available to the planner for the domain, it might be more efficient to have the planner come up with them by itself.

4.7 Summary

We introduced a method for transferring libraries of trajectories to new environments. It combines the idea of a library containing discrete state-action pairs from Sect. 3 with the idea of using local features to transfer knowledge from Sect. 2. By enabling libraries of state-action pairs to be transferred, we can sensibly create libraries of special-purpose behaviors that can be applied across problem. We use such libraries to augment planners so that they can solve harder problems than what they could solve on their own.

Two variations of this algorithm were applied to the Little Dog domain using high-dimensional terrain-based local features which were simplified using PCA. The second variant used additional domain-specific knowledge to adapt actions from the library and increase their applicability to a larger variety of situations. This also revealed limitations of PCA in determining good features for determining fitness for transferability of actions.

5 Conclusion

We presented several algorithms that advance the state-of-the-art in reinforcement learning and planning algorithms. The first algorithm uses local features to transfer policies based on value functions. This work is applicable to domains which can be solved using value-function based dynamic programming, which generally limits the algorithm to domains with only a moderate number of dimensions. Furthermore, domains with discontinuous models or reward functions may pose further problems for the representation of the value function. A requirement for transfer, common to all the work presented here, is that states can be described using features that allow for generalizing to new environments.

In order to address the limitations posed by value-function based dynamic programming, we devised a policy representation based on trajectories created by (kinodynamic) programming. Policies based on trajectory libraries naturally focus on relevant parts of the state space and, due to their sparse representation, scale to state-spaces with more dimensions. They are also more adaptive, allowing them to quickly increase the fidelity of the policy in parts of the state space that exhibit poor performance. This makes them more suitable to robotic applications than dynamic programming, which the first transfer algorithm is based on.

Finally, we devise algorithms for transfer with trajectory libraries. We briefly introduce a simple direct-transfer algorithm which directly combines the idea of local features from the dynamic programming based transfer algorithm with the use of trajectory libraries as control policies. We then present a more elaborate transfer algorithm for transferring behaviors in domains with rich environments, where behaviors have to take into account a large number of features from the environment. This is in contrast to many previous works of motion libraries where environments are assumed to be largely uniform, and most non-uniformities are considered obstacles that disqualify affected motion primitives. In contrast to the direct-transfer algorithm, the transfer-with-planning algorithm is geared towards smaller libraries with valuable behaviors that need to be well matched with the states in which they are applied.

6 Future work

There are still some open research questions that future research should address. In particular, alternatives to nearest-neighbor lookup should be explored. It would be interesting to consider other function approximation techniques, especially kernel-based methods such as Support Vector Machines (SVM) or Support Vector Regression (SVR).

Furthermore, the trade-offs between density, coverage and efficiency of the trajectory library could be analyzed more carefully. Especially in domains with many dimensions, one might expect a dramatic increase in the size of the trajectory library necessary to perform well. Fortunately, query complexity into kd-trees used for nearest-neighbor lookup only grows logarithmically with the number of points. Furthermore, methods for improving trajectories based on trajectory libraries focus on relevant parts of the state-space which might be much smaller than the total state-space under consideration. Since trajectories are added in response to poor performance, we hope that the density of the library reflects the underlying difficulty of the task. If the task is easy, we hope that even few trajectories allow for successful policies, regardless of dimensionality of the state-space. Finally, SVMs or SVRs could be explored to represent the policies more compactly.

Another area of future research is the case when lookups into a library are performed from states with features that are not similar to any of the states contained in the library. When used for transfer, we could simply chose not

to transfer knowledge in such states. In the case of transfer for dynamic programming, it would be instructive to see how our generalized policy iteration algorithm performs relative to value iteration in such cases. In the case of transferring trajectory libraries with planning, the problem gracefully degrades to solving the problem directly with the path planner: without transferred behaviors, the high level graph only contains the start-state and goal-state which has a single high-level path connecting the two. In the case of using a trajectory library as a policy, a query from a state far away from any state in the library can either be caught explicitly and cause an invocation of the planner, or will probably result in failure which will also result in the addition of a new plan to the library.

Finally, a significant and important area of future work is formal analysis of the algorithms presented here. It would be informative to derive convergence guarantees for the generalized policy iteration algorithm presented in Sect. 2. More importantly, performance guarantees for policies based on trajectory libraries and convergence guarantees as the density of the libraries increases would be important results.

Acknowledgements This material is based upon work supported in part by the National Science Foundation (NSF) under NSF Grant ECS-0325383 and the Defense Advanced Research Projects Agency Learning Locomotion Program.

References

Argall, B., Browning, B., & Veloso, M. M. (2008). Learning robot motion control with demonstration and advice-operators. In *Proceedings IEEE/RSJ international conference on intelligent robots and systems*.

Atkeson, C. G. (1994). Using local trajectory optimizers to speed up global optimization in dynamic programming. In J. D. Cowan, G. Tesauro, & J. Alspector (Eds.), *Advances in Neural Information Processing Systems* (Vol. 6, pp. 663–670). San Mateo: Morgan Kaufmann. URL ftp://ftp.cc.gatech.edu/pub/people/cga/local.html.

Atkeson, C. G., & Morimoto, J. (2003). Nonparametric representation of policies and value functions: a trajectory-based approach. In S. Becker, S. Thrun, & K. Obermayer (Eds.), *Advances in Neural Information Processing Systems* (Vol. 15, pp. 1611–1618). Cambridge: MIT Press. URL http://www-2.cs.cmu.edu/~cga/publications.html.

Bailey, S., Grossman, R. L., Gu, L., & Hanley, D. (1996). A data intensive computing approach to path planning and mode management for hybrid systems. In R. Alur, T. A. Henzigner, & E. Sontag (Eds.), *Lecture notes in computer science: Vol. 1066. Hybrid systems III, Proceedings of the DIMACS workshop on verification and control of hybrid systems* (pp. 485–495). Berlin: Springer. URL http://www.rgrossman.com/pubs.htm.

Bentivegna, D. C. (2004). *Learning from observation using primitives*. PhD thesis, Georgia Institute of Technology. URL http://etd.gatech.edu/theses/available/etd-06202004-213721/.

Bentivegna, D. C., Atkeson, C. G., & Cheng, G. (2006). Learning similar tasks from observation and practice. In *Proceedings of the 2006 IEEE/RSJ international conference on intelligent robots and systems* (pp. 2677–2683). Beijing, China.

Boddy, M. S., & Dean, T. (1994). Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, *67*(2), 245–285. doi:10.1016/0004-3702(94)90054-X.

Caruana, R. (1993). Multitask learning: a knowledge-based source of inductive bias. In *International conference on machine learning*.

Chernova, S., & Veloso, M. (2004a). An evolutionary approach to gait learning for four-legged robots. In *Proceedings of the international conference on intelligent robots and systems (IROS 2004)*. URL http://www-2.cs.cmu.edu/~coral/publications/b2hd-iros04-chernova.html.

Chernova, S., & Veloso, M. (2004b). Learning and using models of kicking motions for legged robots. In *Proceedings of the international conference on robotics and automation (ICRA 2004)*. URL http://www-2.cs.cmu.edu/~coral/publications/b2hd-icra04-chernova.html.

Chernova, S., & Veloso, M. (2008). Learning equivalent action choices from demonstration. In *International conference on intelligent robots and systems*.

Chestnutt, J., Lau, M., Cheung, K. M., Kuffner, J., Hodgins, J. K., & Kanade, T. (2005). Footstep planning for the Honda ASIMO humanoid. In *Proceedings of the IEEE international conference on robotics and automation*. URL http://www.ri.cmu.edu/pubs/pub_4970.html.

Conner, D. C., Rizzi, A., & Choset, H. (2003). Composition of local potential functions for global robot control and navigation. In *Proceedings of the international conference on intelligent robots and systems (IROS 2003)* (Vol. 4, pp. 3546–3551). New York: IEEE. URL http://www.ri.cmu.edu/pubs/pub_4556.html.

Connolly, C., & Grupen, R. (1993). The application of harmonic potential functions to robotics. *Journal of Robotic Systems*, *10*(7), 931–946.

Davies, S. (1997). Multidimensional interpolation and triangulation for reinforcement learning. In *Advances in neural information processing systems* (Vol. 9). San Mateo: Morgan Kaufmann. URL http://www.autonlab.org/autonweb/showPaper.jsp?ID=davies-multidimensional.

Fern, A., Yoon, S. W., & Givan, R. (2004). Learning domain-specific control knowledge from random walks. In *Proceedings of the international conference on automated planning and scheduling* (ICAPS) (pp. 191–199).

Fern, A., Yoon, S., & Givan, R. (2006). Approximate policy iteration with a policy language bias: solving relational Markov decision processes. *Journal of Artificial Intelligence Research*, *25*, 85–118.

Fikes, R. E., Hart, P. E., & Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, *3*, 251–288.

Frazzoli, E. (2001). Robust hybrid control for autonomous vehicle motion planning. Department of aeronautics and astronautics, Massachusetts Institute of Technology, Cambridge, MA. URL http://rigoletto.seas.ucla.edu/papers/Year/2001.html.

Friedman, J. H., Bentley, J. L., & Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, *3*(3), 209–226. doi:10.1145/355744.355745, URL http://portal.acm.org/citation.cfm?doid=355744.355745.

Grollman, D. H., & Jenkins, O. C. (2008). Sparse incremental learning for interactive robot control policy estimation. In *International conference on robotics and automation* (pp. 3315–3320).

Grossman, R., Mehta, S., & Qin, X. (1992). *Path planning by querying persistent stores of trajectory segments* (Tech. Rep. LAC 93-R3). Laboratory for Advanced Computing University of Illinois at Chicago. URL http://www.rgrossman.com/pubs.htm.

Guestrin, C., Koller, D., Gearhart, C., & Kanodia, N. (2003). Generalizing plans to new environments in relational MDPs. In *Proceedings of the eighteenth international joint conference on artificial intelligence*.

Howard, T., & Kelly, A. (2007). Optimal rough terrain trajectory generation for wheeled mobile robots. *International Journal of Robotics Research*, *26*(2), 141–166. URL http://www.ri.cmu.edu/pubs/pub_5739.html.

Iba, G. A. (1989). A heuristic approach to the discovery of macro-operators. *Machine Learning*, *3*, 285–317.

Jacobson, D. H., & Mayne, D. Q. (1970). *Differential dynamic programming*. Amsterdam: Elsevier.

Kavraki, L., Svestka, P., Latombe, J., & Overmars, M. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, *12*(4), 566–580. doi:10.1109/70.508439, URL http://ai.stanford.edu/~latombe/pub.htm.

Kohl, N., & Stone, P. (2004). Machine learning for fast quadrupedal locomotion. In *Proceedings of the nineteenth national conference on artificial intelligence* (pp. 611–616). URL http://www.cs.utexas.edu/~nate/pubs/b2hd-kohlaaai04.html.

Kolter, J. Z., Abbeel, P., & Ng, A. Y. (2008). Hierarchical apprenticeship learning with application to quadruped locomotion. In *Neural information processing systems 20*.

Laird, J., Rosenbloom, P., & Newell, A. (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, *1*, 11–46.

Lau, M., & Kuffner, J. J. (2005). Behavior planning for character animation. In *SCA '05: proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on computer animation* (pp. 271–280). New York: ACM Press. doi:10.1145/1073368.1073408.

LaValle, S. M. (2006). *Planning algorithms*. Cambridge: Cambridge University Press. URL http://msl.cs.uiuc.edu/planning/, to appear.

LaValle, S. M., & Kuffner, J. J. Jr. (2001). Randomized kinodynamic planning. *The International Journal of Robotics Research*, *20*(5), 378–400. doi:10.1177/02783640122067453, URL http://ijr.sagepub.com/cgi/content/abstract/20/5/378.

Lee, J., Chai, J., Reitsma, P. S. A., Hodgins, J. K., & Pollard, N. S. (2002). Interactive control of avatars animated with human motion data. In *SIGGRAPH '02: Proceedings of the 29th annual conference on computer graphics and interactive techniques* (pp. 491–500). New York: ACM Press. doi:10.1145/566570.566607.

Littman, M. L., Sutton, R. S., & Singh, S. (2002). Predictive representations of state. In *Advances in neural information processing systems* (Vol. 14, pp. 1555–1561). San Mateo: Morgan Kaufmann. URL http://www.eecs.umich.edu/~baveja/PSRmainpage.html.

Mahadevan, S. (1992). Enhancing transfer in reinforcement learning by building stochastic models of robot actions. In *Proceedings of the ninth international conference on machine learning* (pp. 290–299). URL http://www.cs.umass.edu/~mahadeva/organized-pubs-by-year.html.

Mannor, S., Menache, I., Hoze, A., & Klein, U. (2004). Dynamic abstraction in reinforcement learning via clustering. In *Proceedings of the twenty-first international conference on machine learning*.

McGovern, A. (2002). *Autonomous discovery of temporal abstractions from interaction with an environment*. PhD thesis, University of Massachusetts Amherst. URL http://www.cs.ou.edu/~amy/pubs.html.

Munos, R., & Moore, A. (2002). Variable resolution discretization in optimal control. *Machine Learning*, *49*(2/3), 291–323. URL http://www.autonlab.org/autonweb/showPaper.jsp?ID=munos-variable.

Pearl, J. (1985). *Heuristics: intelligent search strategies for computer problem solving*. Reading: Addison-Wesley.

Ravindran, B., & Barto, A. G. (2003). SMDP homomorphisms: an algebraic approach to abstraction in semi Markov decision processes. In *Proceedings of the eighteenth international joint conference on artificial intelligence*. Menlo Park: AAAI Press. URL http://www.cs.iitm.ernet.in/~ravi/.

Reitsma, P. S. A., & Pollard, N. S. (2004). Evaluating motion graphs for character navigation. In *Proceedings of ACM SIGGRAPH/Eurographics 2004 symposium on computer animation*.

Rimon, E., & Koditschek, D. E. (1992). Exact robot navigation using artificial potential fields. *IEEE Transactions on Robotics and Automation*, *8*(5), 501–518.

Röfer, T. (2005). Evolutionary gait-optimization using a fitness function based on proprioception. In *Eighth international workshop on robocup 2004*. URL http://www.informatik.uni-bremen.de/~roefer/public_e.htm.

Sermanet, P., Hadsell, R., Scoffier, M., Grimes, M., Ben, J., Erkan, A., Crudele, C., Muller, U., & LeCun, Y. (2009). A multi-range architecture for collision-free off-road robot navigation. *Journal of Field Robotics*, *26*(1), 58–87. URL http://yann.lecun.com/exdb/publis/index.html.

Şimşek, Ö., & Barto, A. G. (2004). Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the twenty-first international conference on machine learning*. URL http://www.cs.umass.edu/~ozgur/.

Stolle, M. (2007). Images of mazes used. URL http://www.cs.cmu.edu/~mstoll/files/adprl2007-mazes.tar.gz, http://www.cs.cmu.edu/~mstoll/files/adprl2007-mazes.tar.gz.

Stolle, M. (2008). *Finding and transferring policies using stored behaviors*. PhD thesis, Carnegie Mellon University, 5000 Forbes Ave Pittsburgh, PA 15213. URL http://www.cs.cmu.edu/~mstoll/publications.shtml.

Stolle, M., & Atkeson, C. G. (2006). Policies based on trajectory libraries. In *Proceedings of the international conference on robotics and automation (ICRA 2006)*. URL http://www.cs.cmu.edu/~mstoll/publications.shtml.

Stolle, M., & Atkeson, C. (2007a). Transfer of policies based on trajectory libraries. In *Proceedings of the international conference on intelligent robots and systems (IROS 2007)*. URL http://www.cs.cmu.edu/~mstoll/publications.shtml.

Stolle, M., & Atkeson, C. G. (2007b). Knowledge transfer using local features. In *Proceedings of the IEEE symposium on approximate dynamic programming and reinforcement learning (ADPRL 2007)*. URL http://www.cs.cmu.edu/~mstoll/publications.shtml.

Stolle, M., & Precup, D. (2002). Learning options in reinforcement learning. In *Lecture notes in computer science* (Vol. 2371, pp. 212–223). Berlin: Springer. URL http://www.cs.cmu.edu/~mstoll/publications.shtml.

von Stryk, O. (2001). DIRCOL. http://www.sim.informatik.tu-darmstadt.de/sw/dircol.html.en, URL http://www.sim.informatik.tu-darmstadt.de/sw/dircol.html.en.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: an introduction*. Cambridge: MIT Press. URL http://www.cs.ualberta.ca/~sutton/book/the-book.html.

Veloso, M. M. (1992). *Learning by analogical reasoning in general problem solving*. PhD thesis, Carnegie Mellon University.

Weingarten, J. D., Lopes, G. A. D., Buehler, M., Groff, R. E., & Koditschek, D. E. (2004). Automated gait adaptation for legged robots. In *International conference in robotics and automation*. New York: IEEE Press.

Winner, E., & Veloso, M. (2002). Automatically acquiring planning templates from example plans. In *Proceedings of AIPS'02 workshop on exploring real-world planning*. URL http://www-2.cs.cmu.edu/~coral/publications/b2hd-02aipsw-elly.html.

Yang, L., & LaValle, S. M. (2004). The sampling-based neighborhood graph: a framework for planning and executing feedback motion strategies. *IEEE Transactions on Robotics and Automation*, *20*(3), 419–432.

Yoon, S., Fern, A., & Givan, R. (2008). Learning control knowledge for forward search planning. *Journal of Machine Learning Research*, *9*, 683–718.

Martin Stolle is a software engineer at Google in Zurich, Switzerland. He received B.Sc and M.Sc. degrees in computer science from McGill University in 2002 and 2003, and a Ph.D degree in Robotics from the Robotics Institute at Carnegie Mellon University in 2008. His doctoral research focused on developing new policy representations and transfer learning for robotics applications. This work was applied to challenging, dynamic tasks such as the marble maze and Little Dog.

Christopher Atkeson is a Professor in the Robotics Institute and Human-Computer Interaction Institute at Carnegie Mellon University. He received the M.S. degree in Applied Mathematics (computer science) from Harvard University and the Ph.D. degree in Brain and Cognitive Sciences from MIT. He joined the MIT faculty in 1986 and moved to the Georgia Institute of Technology College of Computing in 1994. He has been with Carnegie Mellon University (CMU) since 2000. His research focuses on humanoid robotics and robot learning by using challenging dynamic tasks such as juggling. His specific research interests include nonparametric learning, memory-based learning including approaches based on trajectory libraries, reinforcement learning, and other forms of learning based on optimal control, learning from demonstration, and modeling human behavior. Dr. Atkeson has received a National Science Foundation Presidential Young Investigator Award, a Sloan Research Fellowship, and a Teaching Award from the MIT Graduate Student Council.