Finding and transferring policies

Martin Stolle

September 11, 2006

Contents

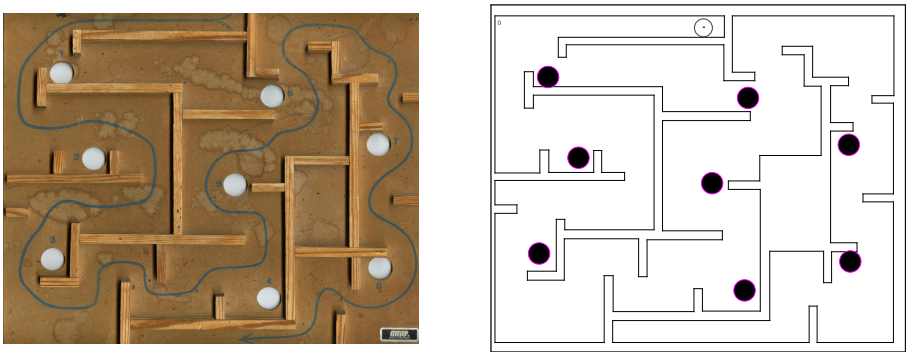
1 Overview

The aim of this thesis is to advance the state-of-the-art in reinforcement learning and planning algorithms so they can be applied to high-dimensional real-world problems. The main focus 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 would be possible without learning from smaller problems first. The key approach for attaining these goals is to use multiple state representations in a single domain that together enable transfer of knowledge as well as learning and planning on different levels of details. In particular, while most domains have an obvious "ground truth" state representation such as absolute position and velocity with respect to a fixed origin or joint position and velocities, it is sometimes beneficial to consider ambiguous state representations in terms of local features of the agent or actuator. While such state representations might only allow for short term predictions of the state evolution and alias multiple states, 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, it is however important to use them in conjunction with more powerful state representations, such as the "ground truth" state representation.

This document is organized as follows: In chapter 2, I introduce and describe the experimental domains I am using to validate the ideas and algorithms. In chapter 3, I describe an algorithm to speed up the creation of global control laws using dynamic programming by transferring knowledge from previously solved problems in the same domain. Results are presented for simulations in the marble maze domain. Several possibilities for future work are described that built upon the presented results. In chapter 4, we describe an alternative representation for control laws based on trajectory libraries. Results are shown on both a simulated marble maze as well as a physical implementation of the domain. Several topics for future work are presented. Finally in chapter 5, we propose ways of transferring the libraries presented in chapter 4.

2 Experimental Domains

2.1 Marble Maze



(a) original (b) simulation

Figure 1: A sample marble maze

Two domains are used to validate and gauge the proposed algorithms: The marble maze domain and the Little Dog domain. The marble maze domain (Figure 1) is also known as "Labyrinth" and consists of a plane with walls and holes. 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* used so far 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. A higher-dimensional

marble maze simulator was used by Bentivegna [3]. In Bentivegna's simulator the current tilt of the board is also part of the state representation.

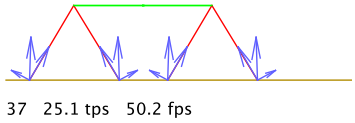
Figure 2: The real world maze

The experiments that were performed on the *real world* maze 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 also 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.

2.2 Little Dog

Another domain we will use for gauging the effectiveness of the algorithms is the Little Dog domain. Little Dog is a quadruped robot developed by Boston Dynamics for DARPA (Figure 3). It has four legs, each with three

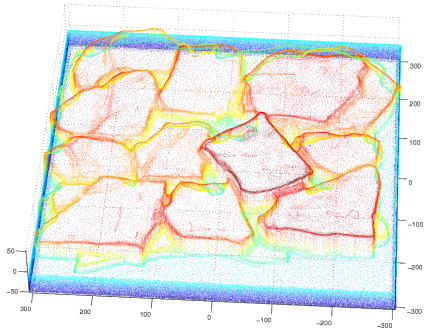
(a) robot



(b) simulator

Figure 3: Little Dog platform

(a) physical board



(b) computer model

Figure 4: Sample terrain board

actuated degrees of freedom. Two degrees of freedom are at the hip (forward-backward, inward-outward) and one at the knee (forward-backward). This results in a twelve dimensional action space (three for each of the four legs). The state space is 36 dimensional (24 dimensions for the legs and twelve for the center of mass). The task to be solved in this domain is to navigate a small-scale rough terrain model with obstacles (figure 4).

For preliminary experiments, we devised a simple *simulator* of a planar quadruped (figure 3). Each leg is simulated as a massless spring-damper with two degrees of freedom: the angle with respect to the body and its length. Actions in the simulator are desired angles and lengths of the legs. If they cannot be obtained because the foot intersects the ground, forces proportional to the difference between achieved length and desired length as well as torques proportional to the difference between achieved angle and

desired angle are applied to the body of the robot. The state space of the system is 14 dimensional: six dimensions for the position and velocity of the center of mass and eight dimensions for the current angle and length of the legs. Since the legs are massless, their velocities are not part of the state. For later experiments, we will use a 3-dimensional model and environment. It will still use massless spring dampers as legs. Digital models of physical obstacles, as seen in figure 4, will be used in the simulator.

To validate the simulators and as a more important testbed for the algorithms, we will use the physical Little Dog. The software interface provided by BDI will be used to communicate with the robot over a wireless or possibly wired Ethernet connection. 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 on the robot as well as the terrain boards. Combined with accurate 3d laser scans of the terrain boards, no on-board sensor is needed to sense the obstacles.

3 Dynamic Programming Policy Transfer

3.1 Introduction

Finding policies, a function mapping states to actions, using dynamic programming (DP) 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 representation which generalizes across tasks. By way of this local state representation, policies can be translated across tasks and used to seed planning algorithms with a good initial policy.

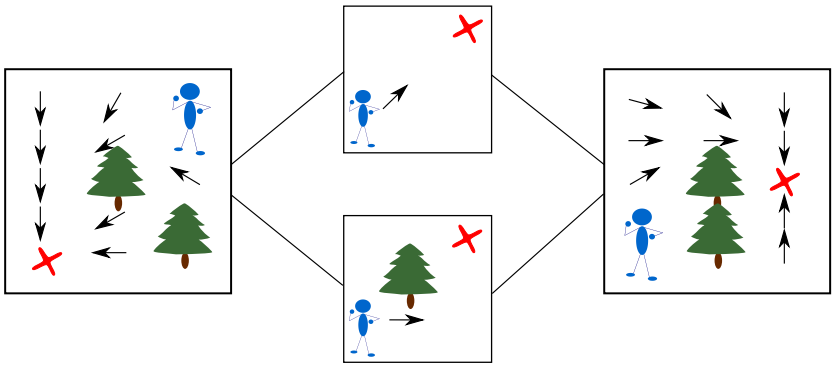


Figure 5: example navigation domain, left: original terrain, middle: feature-based policy, right: new terrain

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 figure 5, 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 representation.

3.2 Related Work

The 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 the knowledge can be transferred across. For symbolic planners and problem solvers, the high level relational description of the environment allows for transfer of plans or macro operators across very different tasks, as long as it is still within the same domain. Work on this goes back to STRIPS [9], SOAR [16], Maclearn [12] and analogical reasoning with PRODIGY [33]. More recent relevant work in planning can be found in [36, 8].

In controls, work has been done on modeling actions using local state representations [22, 6]. Other work has been done to optimize low-level controllers, such as walking gaits, which can then be used in different tasks [15, 5, 28, 35]. Some work has been done in automatically creating macro-actions in reinforcement learning [24, 31, 29, 23], 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 the local state representation. Another method for reusing macro actions in different states using homomorphisms can be found in [27].

3.3 Case Study: Marble Maze

We used the marble maze domain (figure 1) to gauge the effectiveness of our knowledge transfer approach. The model used for dynamic programming is the simulator described in section 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 [7].

The positional resolution of the state space was 3mm and the velocity resolution was 12.5mm/s. The mazes were of size 289mm by 184mm and speeds between -50mm/s to +50mm/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 [25] could be used to limit high-resolution representation to parts of the space where it is strictly necessary. The maximum force on the marble in each dimension was limited to 0.0014751N and discretized into -.001475N, 0 and +.001475N in each dimension, resulting in 9 possible actions for each state. With a simulated mass of the marble of .0084kg, maximal acceleration was about 176mm/s^2 in each dimension. Time was discretized to 1/60th of a second.

3.3.1 Local State Representation

The local state representation, chosen from the many possible local representations, 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 figure 6) 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

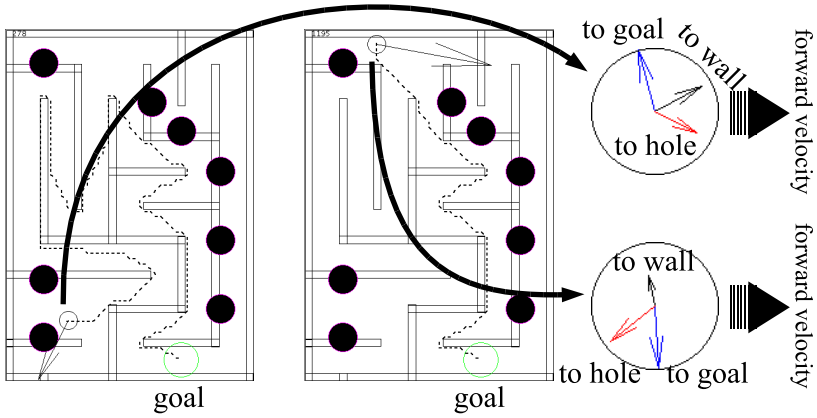


Figure 6: local state representation

tell us what actions to use. Two examples of this local state representation can be seen in figure 6. In the circle representing the relative state of 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 representation in the discussion section (section 3.5).

3.3.2 Knowledge Transfer

The next step is to transfer knowledge from one maze to the next. For the intermediate policy, expressed using the local state representation, 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 representation 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 representation 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. Furthermore, performance would be expected to be suboptimal as the local representation alone does not necessarily determine the optimal action and previous policies might not have encountered certain states that now appear on the new task.

Instead, an initial policy based on global coordinates is created using the classifier by iterating over the 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.

3.3.3 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 (equation 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 as the agent will use high valued states in its policy. If high values cannot propagate back through these bad regions, the values behind these bad regions will be incorrect and value iteration will not be sped up. Similarly, if a policy improvement step was used to correct the policy in these states, the policy of states behind these bad regions would be updated based on an incorrect value function.

We overcame these two problems by creating a form of generalized policy iteration [32]. The objective in creating this dynamic programming algo-

rithm 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 equation 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.

In order to avoid changing the policy for states using invalid 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 can easily be implemented by a flag for each state that is propagated back with the values. Note that as a result, we do not compute the value of states that cannot reach the goal.

3.4 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 [30]). 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 7 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

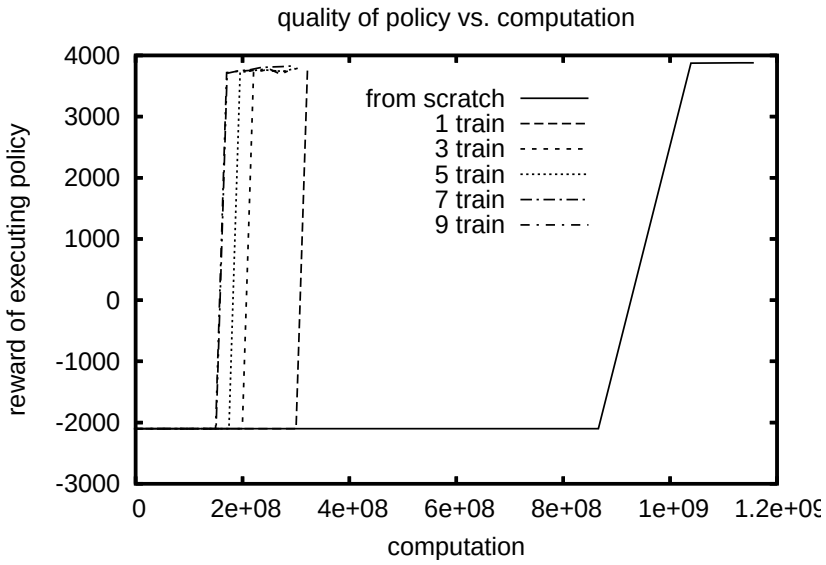


Figure 7: results for one test maze

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.

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 figure 8 (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 figure 8 (dashed). Clearly, the individual training mazes and their ordering do not influence the results very much.

3.5 Discussion

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

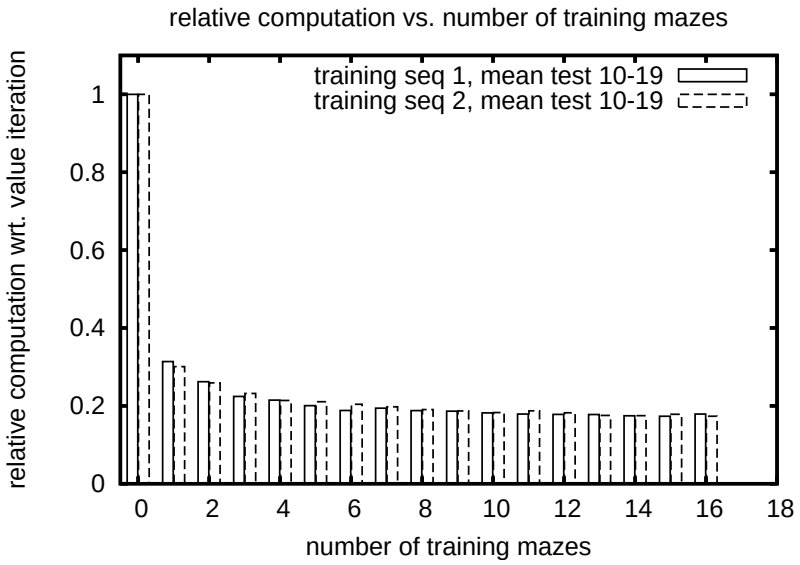


Figure 8: relative computation, averaged over 10 test mazes, using two different sequences of training mazes

that are in the immediate vicinity 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 [21]. 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 representations, we created an alternative state representation. In this alternative representation, the world is represented 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.

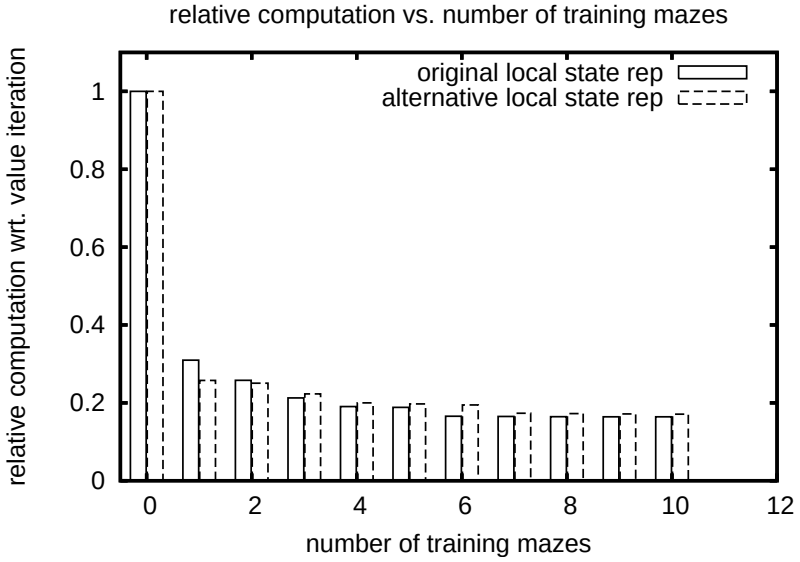


Figure 9: relative computation required for one test maze for two different local state representation

Holes that are behind walls are not considered. The velocity of the marble is projected onto the path towards the goal. The resulting state representation is less precise with respect to direction to the walls or holes than the original local representation but takes into account up to four holes and walls, one for each quadrant. As can be seen in figure 9, the results are similar for both state representations. The new state representation 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 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

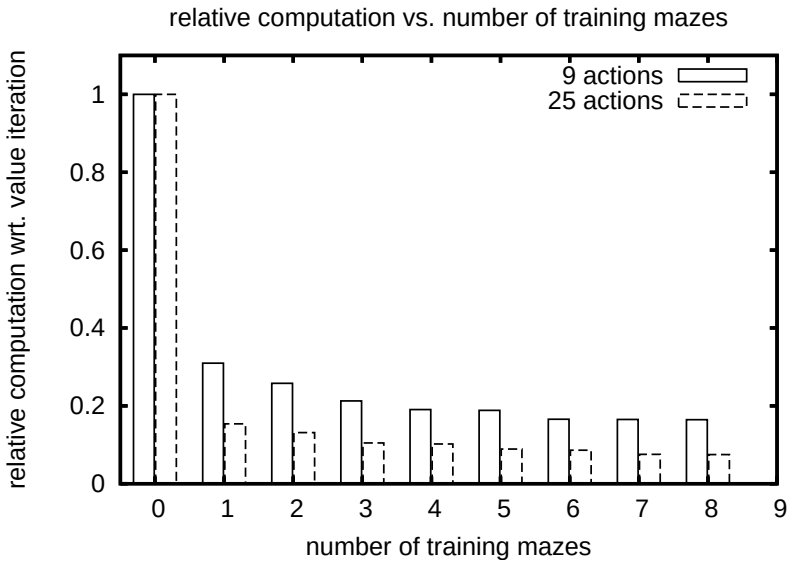


Figure 10: relative computation required for one test maze and different number of actions

the number of actions, we reran the experiments for one maze with actions discretized into 25 different actions instead of 9. As seen in figure 10, 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 figure 11. 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

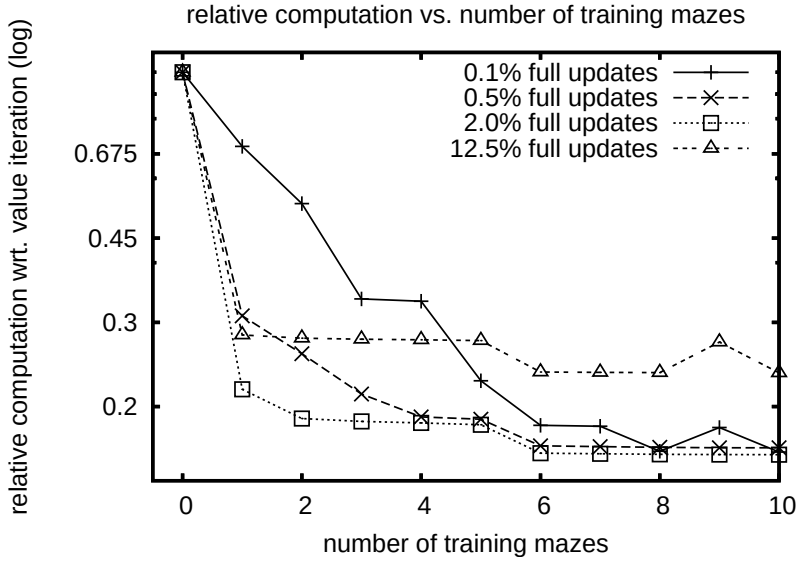


Figure 11: relative computation required for one test maze and different percentages of full updates

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 representation. However, if a richer state representation 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

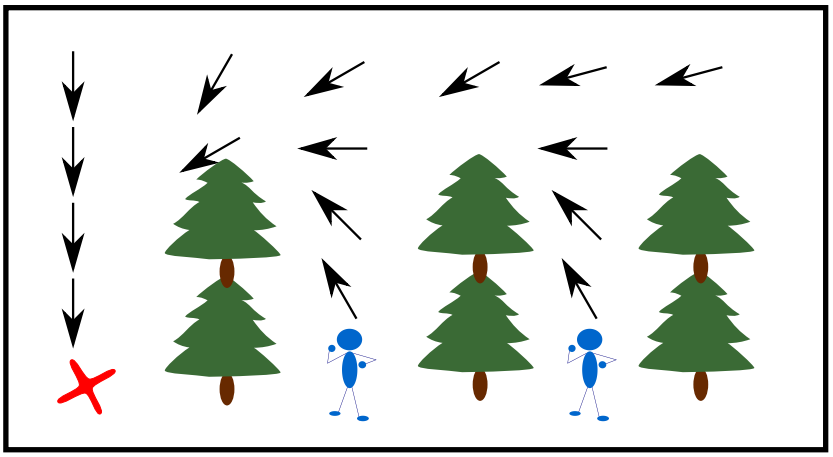


Figure 12: aliasing problem: same local features, same policy but different values

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 be seen in figure 12. 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).

3.6 Conclusion

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 representation that allows for the representation of knowledge that is independent of the individual task.

3.7 Future Work

- Perform local search for best action (instead of discrete actions)
- Verify the claim that doing dynamic programming using this intermediate state representation directly has limited use.
- Apply DP policy to stochastic simulator and physical maze
- Compare 4d state space vs. 6d state space on physical maze
- Use a less brittle metric to compare policies (not just computational effort before successfully solving maze)
- Use reward as metric for basis of comparison

4 Policies based on Trajectory Libraries

4.1 Introduction

Finding a policy, a control law mapping states to actions, is essential in solving many problems with inaccurate or stochastic models. By knowing how to act for all or many states, an agent can cope with unexpected state transitions. Unfortunately, methods for finding policies based on dynamic programming require the computation of a value function over the state space. This is computationally very expensive and requires 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 or time varying linear control law. The desired trajectory can be generated manually, generated by a path planner [19], or generated by trajectory optimization [34]. For systems with nonlinear dynamics, this approach may fail if the actual state diverges sufficiently from

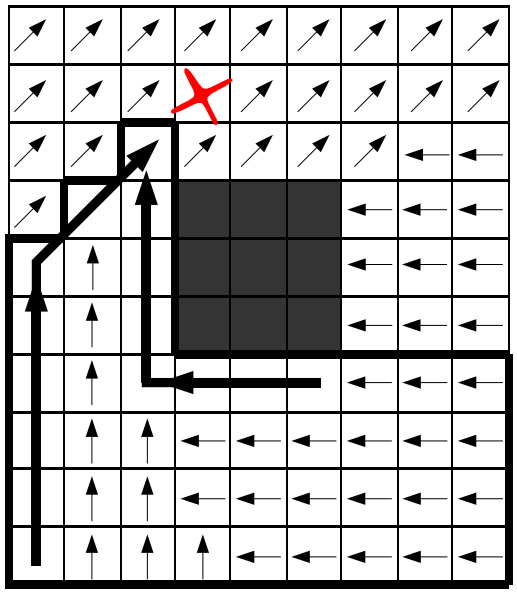


Figure 13: A library of trajectories (thick arrows) is used to create global policy (thin arrows) by nearest-neighbor look up

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[4] 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 13 shows a simple grid world example with eight possible actions (N, E, S, W, NE, SE, SW, NW). The cross marks the goal and the dark 3x3 region is an obstacle. The thick arrows show the two trajectories which make up the library. These paths can be created very quickly using

forward planners such as A* or Rapidly exploring Random Trees (RRT)[19]. These 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 state-space for the closest trajectory fragment and executing the associated action. In the discrete example environment of Figure 13 we have designated the resulting policy for all states using thin arrows. For large parts of the environment, marked by the thick borders, the resulting policy leads to the goal.

4.2 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[17, 20]. 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, ignoring disturbances or inaccuracies.

Another related technique in path planning is the creation of Probabilistic Roadmaps (PRMs)[14]. 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 [10]. The library in this case is not the solution to the goal achievement task, but rather a library of controllers that simplify the path planning problem. Unlike our work presented here, the controllers themselves do not use libraries.

Prior versions of a trajectory library approach, using a modified version of Differential Dynamic Programming (DDP)[13] to produce globally optimal

trajectories can be found in [1, 2]. This approach reduced the cost of dynamic programming, but was still quite expensive and had relatively dense coverage. The approach of this paper uses more robust and cheaper trajectory planners and strives for sparser coverage. Good (but not globally optimal) policies can be produced quickly.

4.3 Case Study: Marble Maze

The domain used for gauging the effectiveness of the new policy representation and generation is the marble maze domain (Figure 1). The model used for creating trajectories is the simulator described in section 2.1. The hardware described in the same section was used for the experiments on the actual maze.

4.4 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[26] 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[26]. 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 2mm 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 [18]. 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

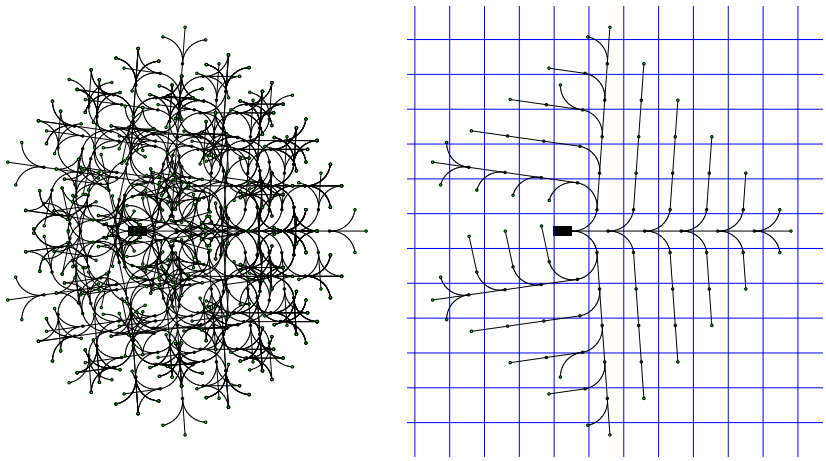


Figure 14: An example of pruning

of a previously expanded path (see figure 14 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 ± 0.007N in both dimension and discretized to a resolution of 0.0035N. This resulted in 25 discrete action choices. For the purpose of pruning the search nodes, the state space was discretized to 3mm spatial resolution and 12.5mm/s in velocity resolution.

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 10x 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[13] or DIRCOL[34].

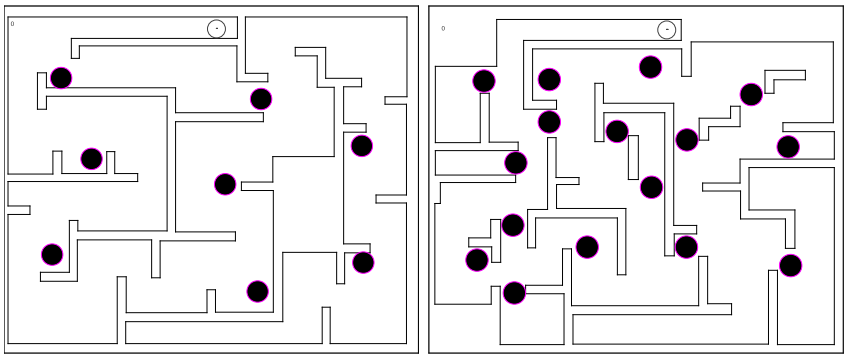
In order to use the trajectory library as a policy, we store a mapping from each state on any trajectory to the planned action of that state. During execution, we perform a nearest-neighbor look up into this mapping using the current state to determine the action to perform. In order to speed up nearest-neighbor look ups, the mapping is stored using a kd-tree[11].

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[3]. 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.

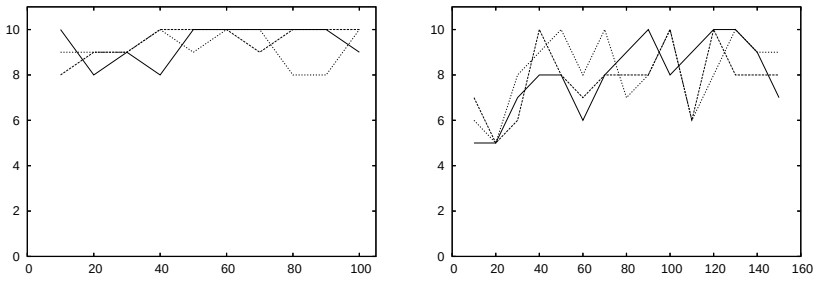
4.5 Experiments



(a) Maze A (b) Maze B

Figure 15: The two mazes used for testing

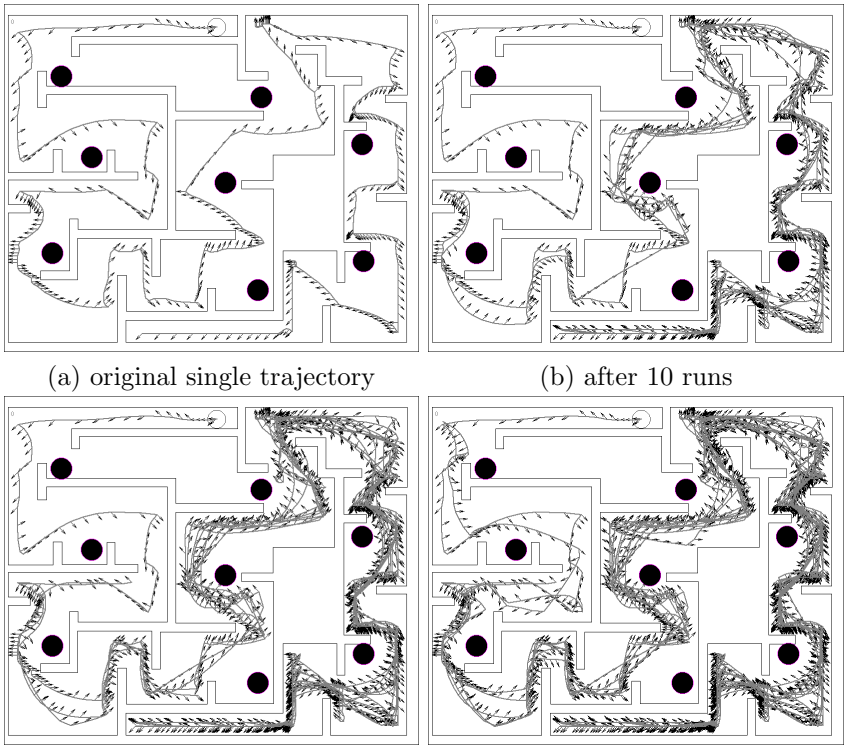
We performed experiments on two different marble maze layouts (Figure 15). The first layout (maze A) is a simple layout, originally designed for beginners. The second layout (maze B) is a harder maze for more skilled players. These layouts were digitized by hand and used with the simulator.



(a) Maze A (b) Maze B

Figure 16: 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.

The first set of experiments was run in simulation. 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



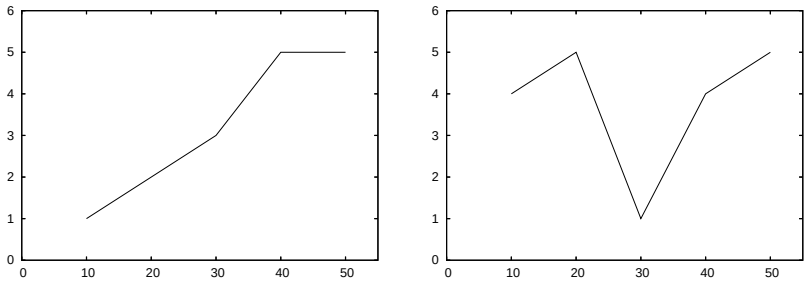
(a) original single trajectory (b) after 10 runs

(c) after 30 runs (d) after 100 runs

Figure 17: Evolution of library of trajectories. (The trajectories (thick lines) are shown together with their actions (thin arrows))

above. After 100 runs, we restarted with an empty library. The results of three sequences of 100 runs each are plotted in Figure 16(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 Figure 17. 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 Figure 16(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.

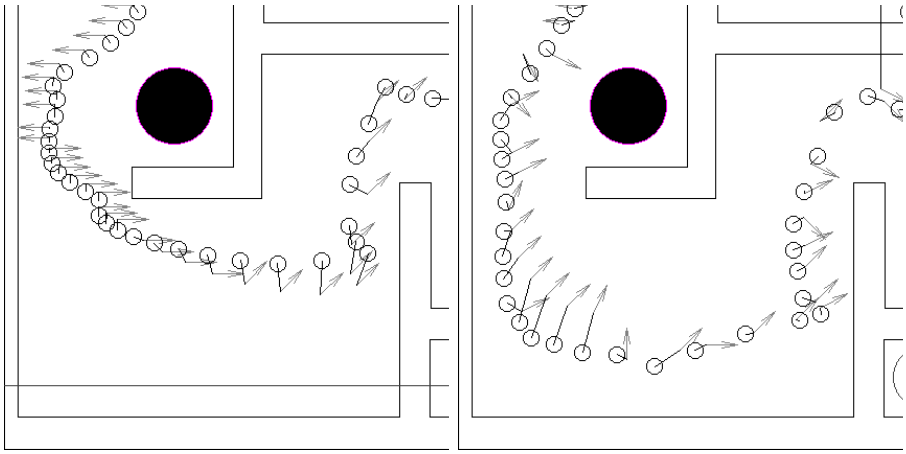
We also used our approach to drive a real world marble maze robot. This problem is much harder than the simulation, as there might be quite



(a) without model updates (b) with model updates

Figure 18: Real world results for maze A. The x axis is the number of starts and the y axis is the number of successes in 10 starts.

large modeling errors and significant latencies. We used the simulator as 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 (Figure 18).



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

Figure 19: 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

In Figure 19 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.

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. The model was updated by storing observed state-action-state change triplets. During planning, a nearest-neighbor look up in state-action space is performed and if a nearby tuplet is found, the stored state change is applied instead of computing the state evolution based on the simulator. While initial results using this method are much better, overall the performance does not improve much over not using the model (Figure 18). Clearly, a better model update mechanism needs to be developed. 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 and even improperly learned models.

4.6 Discussion

We can create an initial policy with as little as one trajectory. Hence, creating this type of policy can be efficient in its use of computation resources. By scheduling the creation of new trajectories based on the performance of the robot or in response to updates of the model, these policies 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[4]: 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.

Furthermore, trajectory planners use a time index and do not require value functions. They can easily deal with discontinuities in the model or cost metric. 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.

Currently, the action selection for the policy is based on a nearest-neighbor look up in the library of trajectories. A poor choice of distance metric in this look up can result in a suboptimal selection of actions. In our experiments 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 0.1m/s and a reasonable neighborhood for positions is about 0.01m, we multiply position by 100 and velocities by 10, resulting in distances around 1 for reasonably close data points. The model learning technique that we used also relies on a nearest-neighbor look up. The same weights were used for position and velocity. Since the model strongly depends on the action, which are quite small (on the order of 0.007N), the weight in the distance metric for the actions is $1x10^6$. The cutoff for switching over to the simulated model was a distance of 1.

Currently, no smoothness constraints are imposed on the actions of the generated policies. 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). Imposing constraints on the trajectories would not solve the problem as the policy uses a nearest-neighbor look up to determine the closest trajectory and 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.

4.7 Conclusion

We have investigated a technique for creating policies based on fast trajectory planners. 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 real world version of the marble maze. In this case, the performance was limited by the accuracy of the model. A

simple model updating technique was used to improve the model with limited success.

4.8 Future Work

- Use 6 dimensional maze model to incorporate current tilt
- Use smoother actions
- Apply to Little Dog
- Incorporate learning from demonstration by using the observed trajectories.
- Use optimizers such as Differential Dynamic Programming (DDP)[13] or DIRCOL[34] to locally optimize the trajectories.
- Use DDP or DIRCOL to update trajectories after a model update.
- Combine trajectories, for example weighted average k-NN.
- Add local control law to trajectories[1].

5 Transfer of Trajectory Libraries

5.1 Introduction

We are considering three alternatives for transferring libraries of trajectories. All approaches hinge on the use of local features to transfer the policy to new problems or new parts of the state space. In principle, all approaches create an intermediate policy from a trajectory library by iterating over every state on all trajectories, computing the local features of that state and creating a policy that maps the given local features to their action. At run time, an action is generated by computing the local features of the current state and doing a nearest neighbor look up to find the most appropriate action.

The three approaches differ in the details that are expected to be part of the feature space and in the kind of post-processing that is required after changing to a new problem. The distinguishing characteristic of the feature space is information about the direction to the goal state of the problem.

In order to clarify the different ways to transfer libraries, I have created illustrations using a simple grid-based environment, where the state of the robot is described by its position (x,y coordinates) and orientation (one of north, east, south or west).

5.2 Explicit Goal Feature

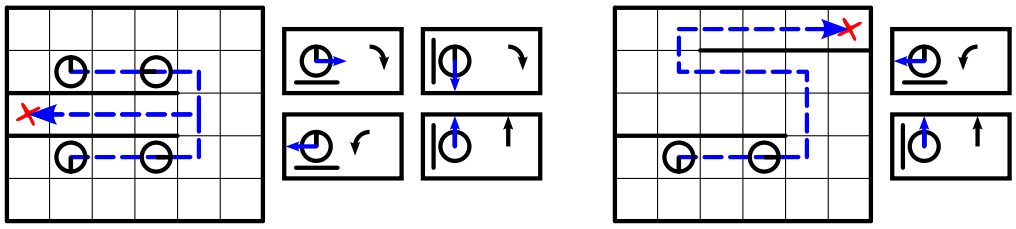


Figure 20: The left hand side shows the environment in which the library was created. The blue, dashed line is the goal feature which for each cell points towards the goal. The library created using a feature space including this goal feature transfers unambiguously to a new environment

In this approach, the local features are assumed to contain information about the goal. For example in the marble maze, the local feature state could contain a vector pointing along a configuration space path to the goal (see figure 6). Similarly, in the illustration (figure 20), we have a feature that

for each cell points to the next cell closer to the goal (this feature does not differentiate between different orientations). Once we have a library of trajectories, we create a local feature based policy by translating the state-action mapping to a feature based state-action mapping. In the simple example, the features that were chosen were the locations of the walls relative to the direction of the robot as well as the direction, relative to the robot, that points to the next cell closer to the goal.

During execution we can perform a nearest neighbor look up both in the library of trajectories based on the absolute state as well as the local feature based policy. Depending on the relative closeness of the nearest neighbor point in either policy, the action from one or the other can be chosen. This way, we can generalize the policy to new parts of the state space. Similarly, if we execute in a new environment, we can start controlling the system without creating trajectories - we just use the local feature based policy. This can be seen on the right side of the illustration: For the two sample states, we immediately know what to do based on the local feature based library. As the local feature space contains information about the goal, we expect this to result in goal directed behavior. This is the easiest way to transfer policies as no post-processing is required.

5.3 Goal-less Features + Search

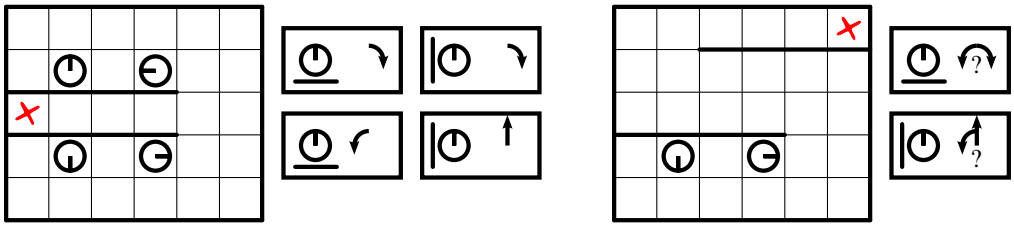


Figure 21: If the goal feature is missing from the feature space used for the library, the library is ambiguous in some states of the new environment. A search can be used to find which action choice is appropriate for reaching the goal

Sometimes, it can be advantageous to not include information about the goal in the local feature space. In this case, we still proceed in the same way and generate a local feature based policy as described above. However, states that are similar in the local feature space are now more likely to map to different actions. For example in the illustration in figure 21, if the local

feature state shows a wall behind the robot, we can either turn left or right. When using this local feature based library in a new environment (right side of figure 21, we now have ambiguity in which action to chose.

This can be viewed as a choice point: if during execution we reach a state where a k-nearest neighbor look up results in different actions, a decision has to be made which action to follow. The decision, which action to take, should ideally depend on where the goal is located and a search is required. In order to avoid a search during on line execution, we propose a preprocessing step when using this kind of intermediate policy.

The preprocessing will use the local feature based policy to create trajectories in the absolute state space quickly. Before executing in a new environment, we have to create trajectories to initialize the library. Instead of planning from scratch from a given start state, we simulate following the feature based policy until we reach a state with multiple possible actions. Now one of the possible actions is picked and the simulation proceeds. Similar to regular search, we later also have to simulate based on the other action and pick the better of the actions. The resulting trajectory is added to the library for this new environment. In the end, this top level search will solve the navigation problem of reaching the goal.

5.4 Extended Action Generation + Search

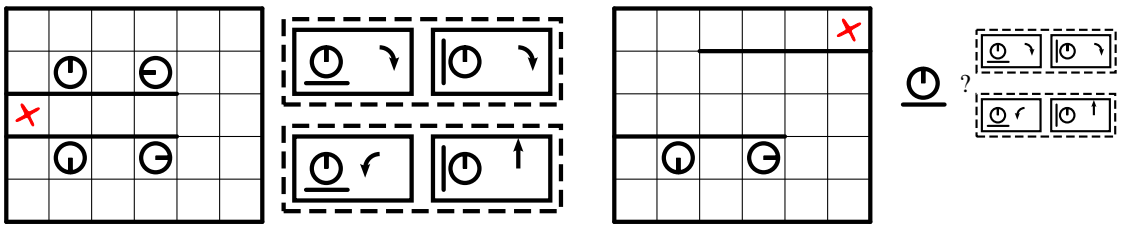


Figure 22: If we created extended actions from coherent parts of the library, we can simplify the search process as now we only need to pick from extended actions and can follow the choice over multiple states.

The previous section described a way to use a planner to select which action to take when the feature based policy contains similar states that map to different actions. Depending on the complexity of the domain, executing an action after the choice point might frequently result in a new state that is ambiguous as well (this would be the case in figure 21, since after the turn,

we would be in a state with only a wall on the left which again is ambiguous). In the end, after every action, multiple actions are again available and search could be as slow as a primitive search over all possible actions. In such cases, it makes sense to cluster the state-action mappings in the intermediate policy into distinct "primitives" or "macro actions". This is illustrated in figure 22, where we have shown two sample extended actions, one for following the wall on the left, the other for following the wall on the right.

Now, once a "primitive" is chosen during the post processing phase, it's policy is followed for a fixed number of time steps or until the region over which the primitive is valid is left. At this point, another primitive can be chosen. A high level search is used to find the best sequence of primitives. This simplifies the high level process as we have to make fewer choices - once an extended action is chosen, we can follow it for multiple actions and despite the fact that the second state in the sample new environment is also ambiguous, we don't have to make a second choice.

In order to find the primitives, we plan on clustering the state-action mappings using unsupervised learning techniques. The clustering will be performed in a combined state-action space. This way, we ensure that each cluster contains state-action mappings with similar actions and states. Each cluster is then converted into a primitive action whose policy is defined as above, with the state-action mappings belonging to the cluster as the data for the feature based policy.

Since the high level planner chooses between a set of discrete primitive actions, the trajectories that it creates will map states to primitive actions. During execution, a nearest neighbor look up is first performed to find which primitive action to pick based on the trajectory library. Once the primitive is decided on, a second nearest neighbor, now in feature space, is performed using the state-action mappings that define the primitive. Since this second look-up is not necessarily performed from a state on the path, the low-level action that is executed might be different from what the planner executed. However, since both low-level actions came from the same primitive, similar behavior is expected.

5.5 Future Work

- Evaluate knowledge transfer using Explicit Goal Feature
- Implement and evaluate knowledge transfer using Goal-less Features

and Search

- Implement and evaluate knowledge transfer using Extended Action Generation + Search
- Do the above for the marble maze and Little Dog

References

[1] Christopher G. Atkeson. Using local trajectory optimizers to speed up global optimization in dynamic programming. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 663–670. Morgan Kaufmann Publishers, Inc., 1994.

[2] Christopher G. Atkeson and Jun Morimoto. Nonparametric representation of policies and value functions: A trajectory-based approach. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems*, volume 15, pages 1611–1618, Cambridge, MA, 2003. MIT Press.

[3] Darrin C. Bentivegna. *Learning from Observation Using Primitives*. PhD thesis, Georgia Institute of Technology, 2004.

[4] Mark S. Boddy and Thomas Dean. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2):245–285, 1994.

[5] Sonia Chernova and Manuela Veloso. An evolutionary approach to gait learning for four-legged robots. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS 2004)*, September 2004.

[6] Sonia Chernova and Manuela Veloso. Learning and using models of kicking motions for legged robots. In *Proceedings of the International Conference on Robotics and Automation (ICRA 2004)*, May 2004.

[7] Scott Davies. Multidimensional interpolation and triangulation for reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 9. Morgan Kaufmann Publishers, Inc., 1997.

[8] Alan Fern, Sung Wook Yoon, and Robert Givan. Learning domain-specific control knowledge from random walks. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 191–199, 2004.

[9] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[10] Emilio Frazzoli. *Robust Hybrid Control for Autonomous Vehicle Motion Planning*. Department of aeronautics and astronautics, Massachusetts Institute of Technology, Cambridge, MA, June 2001.

[11] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[12] Glen A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3:285–317, 1989.

[13] D. H. Jacobson and D. Q. Mayne. *Differential Dynamic Programming*. Elsevier, 1970.

[14] L.E. Kavraki, P. Svestka, J.C. Latombe, and M. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.

[15] Nate Kohl and Peter Stone. Machine learning for fast quadrupedal locomotion. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pages 611–616, July 2004.

[16] J.E. Laird, P.S. Rosenbloom, and A. Newell. Chunking in soar: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[17] Manfred Lau and James J. Kuffner. Behavior planning for character animation. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 271–280, New York, NY, USA, 2005. ACM Press.

[18] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006. to appear.

[19] Steven M. LaValle and James J. Kuffner, Jr. Randomized Kinodynamic Planning. *The International Journal of Robotics Research*, 20(5):378–400, 2001.

[20] Jehee Lee, Jinxiang Chai, Paul S. A. Reitsma, Jessica K. Hodgins, and Nancy S. Pollard. Interactive control of avatars animated with human

motion data. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 491–500, New York, NY, USA, 2002. ACM Press.

[21] Michael L. Littman, Richard S. Sutton, and Satinder Singh. Predictive representations of state. In *Advances in Neural Information Processing Systems*, volume 14, pages 1555–1561. Morgan Kaufmann Publishers, Inc., 2002.

[22] Sridhar Mahadevan. Enhancing transfer in reinforcement learning by building stochastic models of robot actions. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 290–299, 1992.

[23] Shie Mannor, Ishai Menache, Amit Hoze, and Uri Klein. Dynamic abstraction in reinforcement learning via clustering. In *Proceedings of the Twenty-First International Conference on Machine Learning*, 2004.

[24] Amy McGovern. *Autonomous Discovery Of Temporal Abstractions From Interaction With An Environment*. PhD thesis, University of Massachusetts Amherst, 2002.

[25] Remi Munos and Andrew Moore. Variable resolution discretization in optimal control. *Machine Learning*, 49, Numbers 2/3:291–323, November/December 2002.

[26] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1985.

[27] Balaraman Ravindran and Andrew G. Barto. SMDP homomorphisms: An algebraic approach to abstraction in semi markov decision processes. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*. AAAI Press, August 2003.

[28] Thomas Röfer. Evolutionary gait-optimization using a fitness function based on proprioception. In *Eighth International Workshop on Robocup 2004*, 2005.

[29] Özgür Şimşek and Andrew G. Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine Learning*, 2004.

[30] Martin Stolle. Images of mazes used. Internet/WWW, 2005. http://www.cs.cmu.edu/~mstoll/pub/aaai2005-mazes.tar.gz.

[31] Martin Stolle and Doina Precup. Learning options in reinforcement learning. *Lecture Notes in Computer Science*, 2371:212–223, 2002.

[32] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA., 1998.

[33] Manuela M. Veloso. *Learning by Analogical Reasoning in General Problem Solving*. PhD thesis, Carnegie Mellon University, 1992.

[34] Oskar von Stryk. DIRCOL. Internet/WWW, 2001.

[35] Joel D. Weingarten, Gabriel A. D. Lopes, Martin Buehler, Richard E. Groff, and Daniel E. Koditschek. Automated gait adaptation for legged robots. In *International Conference in Robotics and Automation*, New Orleans, USA, 2004. IEEE.

[36] Elly Winner and Manuela Veloso. Automatically acquiring planning templates from example plans. In *Proceedings of AIPS'02 Workshop on Exploring Real-World Planning*, April 2002.