Rapidly Exploring Learning Trees

Kyriacos Shiarlis
University of Amsterdam
k.c.shiarlis@uva.nl

Joao Messias
University of Amsterdam
jmessias@uva.nl

Shimon Whiteson
University of Oxford
shimon.whiteson@cs.ox.ac.uk

Abstract

Inverse Reinforcement Learning (IRL) for path planning enables robots to learn cost functions for difficult tasks from demonstration, instead of hard-coding them. However, IRL methods face practical limitations that stem from their reliance on Markov decision process planners. In this paper, we propose Rapidly Exploring Learning Trees (RLT*), which learns the cost functions of Rapidly Exploring Random Trees (RRT) from demonstration, thereby making inverse learning methods applicable to more complex tasks. Our approach extends the Maximum Margin Planning to work with RRT* cost functions. Furthermore, we propose a caching scheme that greatly reduces the computational cost of this approach. Experimental results on simulated data from a social navigation scenario show that RLT* achieves better performance at lower computational cost than existing methods.

1 Introduction

Learning from demonstration (LfD) [1] is of great interest to roboticists because it can avoid the need for tedious manual programming of complex behaviours. While most LfD methods rely on supervised learning to directly learn policies, certain approaches, namely inverse optimal control (IOC) [2] and inverse reinforcement learning (IRL) [3] instead learn *cost functions* from demonstration. These cost functions are then used to *plan* the robot's behaviour.

Cost functions tend to be more general and robust to changes in the environment such as friction in a manipulator's joints. A robot trained using direct policy estimation (or behavioural cloning), has less chance of adapting than one that has been trained using IRL or IOC because the cost function remains unchanged. In addition, cost functions are thought to be more succinct representations of the aims of the agent [3].

However, inverse methods also have practical limitations. IRL for example requires modeling the task as a Markov decision process (MDP), which is typically impractical in robotics, for several reasons. Firstly, an MDP assumes full observability. Secondly, planning in an MDP, which IRL methods must do repeatedly, is costly, especially in continuous and high-dimensional domains. Finally, the lack of scalability and the assumption of full observability prohibit the accurate modeling of all aspect of the environment that a robot is expected to operate in.

For these reasons, several researchers have proposed replacing the planning step in IRL with more convienient planners, e.g., Maximum Margin Planning (MMP) [4] uses A* search for planning. This avoids the need to do all the planning in advance, as is typical when solving an MDP. It also avoids the need to model all aspects of the environment since the robot can quickly replan its trajectory to account for uncertainty. However, deterministic search methods such as A* require discretisation of the state space, and quickly run into scalability problems as the discretisation becomes finer.

In path planning, Rapidly Exploring Random Trees (RRTs) [5] are popular because they cope well with continuous and high-dimensional domains. The RRT* algorithm [6], which extends RRTs to incorporate a cost function, is especially effective. However, the cost functions used by RRT* are typically simple and hand-coded and, to our knowledge, no methods have been developed to learn RRT* cost functions from demonstrations.

In this paper, we propose Rapidly Exploring Learning Trees (RLT*), which learns RRT* cost functions from demonstration. Specifically we modify Maximum Margin Planning to use RRT* as a planner. Furthermore, we propose a caching scheme that greatly reduces the computational cost of this approach. RLT* requires no additional planner assumptions other than those inherent in RRT*, making it particularly easy to implement.

We evaluate RLT* on real and simulated data from a social navigation scenario. The results demonstrate that RLT* achieves better performance at lower computational cost than methods that learn path planning cost functions for deterministic planners.

2 Related Work

Substantial research has applied IRL to robotics [7; 8; 9]. Because of the need to model the environment as an MDP, researchers usually discretise the state-action space. This introduces several key limitations. First, the size of the state-action space encountered in robotics is often prohibitive for such models. Second, the specific not all physical aspects of the robot can be easily expressed by such representations, introducing non-Markovian dynamics.

To apply IRL to more realistic situations, researchers usually try to replace the MDP model while retaining the main idea behind IRL, i.e., learning the underlying cost function of a planner using data from demonstrations. Maximum Entropy IRL [10] works in domains with linear continuous dynamics with the optimal controller being a Linear-Quadratic Regulator. Since linear dynamics are hard to come by in robotics, [11] considers locally linear aproximations. These bring about locally consistent rewards and achieve good performance in a range of tasks that were previously too hard for MDPs. However, the optimisation of the cost function using such planners is not guaranteed to be optimal. As our method is based on an RRT* planner, it is asymptotically optimal.

Other approaches to the problem use hybrid planners. Inverse Optimal Heuristic Control [12] models the long-term goals of the agent as a coarse state MDP, to ensure tractability, while using supervised learning to determine the local policy of the agent at each state. Graph-Based IOC [13] uses discrete optimal control on a coarse graph and the actual path is executed using local trajectory optimisation techniques such as CHOMP [14], which would otherwise suffer from local minima. The method is effective in robotic manipulation tasks with many degrees of freedom. However, these methods employ complex and domain-specific planning formulations that are not suitable for all robotics tasks. The method presented here employs widely used planners, making it versatile and easy to implement.

Another more recent graph-based concept is that of Adaptive State Graphs [15], which build a controller graph before doing any learning. This controller graph is akin to *options* in semi-Markov decision processes, and allows for a more flexible representation of the state-action space. However, the controller used to learn underlying cost function is not the same as the one used to execute the robot's behaviour. This can have adverse effect since an implicit assumption of IRL is that the demonstration paths came from the same planner that we use during learning. This planner has specific representations, such as discretisation and parameters such as the discount factor. If these change, different policies arise. As a result a path that was optimal under a certain planner ceases to be optimal under another.

Instead of building a controller graph first and then using different controllers to optimise trajectories, RLT* builds a controller tree on the fly. RLT* however uses the exact same planner during learning and execution.

3 Background

We begin with background on path planning and inverse reinforcement learning for path planning.

3.1 Path Planning

Path planning occurs in a space $\mathcal{S}$ of possible configurations of the robot. A configuration $s \in \mathcal{S}$ is usually continuous, and often represents spatial quantities such as position and orientation. A path planner seeks an obstacle-free path $\zeta_{o,g} = (s_1, s_2, s_3 \ldots, s_{l_\zeta})$ of length l_ζ, from an initial configuration $o = s_1$ to a goal configuration $g = s_{l_\zeta}$. When the initial and goal configurations are implied, we refer to a path as ζ.

Because there could be several paths to the goal, path planners typically employ a *cost functional*, $C(\zeta)$ often defined as the sum of the costs between two subsequent configurations in a path $c(s_i, s_j)$. The total cost $C(\zeta)$ is therefore the sum of the individual costs along the path, i.e.,

$$C(\zeta) = \sum_{i=1}^{l_\zeta - 1} c(s_i, s_{i+1}). \tag{1}$$

This cost functional is similar to the one encountered in optimal control as well as to value functions used in reinforcement learning. Given the cost functional, the path planner seeks an optimal path ζ^*, which satisfies,

$$\zeta^*_{o,g} = \operatorname*{argmin}_{\zeta_{o,g} \in Z_{o,g}} C(\zeta), \tag{2}$$

where $Z_{o,g}$ is the set of all possible paths such that $s_1 = o, s_{l_\zeta} = g$.

Many path planning algorithms discretise $\mathcal{S}$ and use graph search algorithms like A* to find the optimal path. Under mild assumptions, these approaches are guaranteed to find the best path on the graph, therefore solving (2) for a subset $\tilde{Z}_{o,g} \in Z_{o,g}$, whose size depends on the graph resolution (discretisation). However, such methods scale poorly in the size of $\mathcal{S}$, as larger and larger graphs need to be searched.

These drawbacks motivate *sample-based* path planning algorithms such as RRT* and PRM*. The application focus of this paper is social robotic navigation, which typically consists of highly non-static environments. Therefore multi-query roadmap methods such as PRM* are not appropriate, since we would need to repeat the learning phase of the algorithm every time as the environment changes. As a result this paper employs RRT* as the underlying planner for the cost function learning process.

Instead of building a graph and then searching it, RRT* builds a tree on the fly and keeps track of the current best path. The algorithm, for which a detailed description and analysis can be found in [6], essentially consists of two interleaved processes.

The first process is that of *sampling*. A random point s_{rand} is sampled from the configuration space. Next, the closest point $s_{closest}$, already in the existing vertex set V is determined and a new point s_{new} is created by *steering* from $s_{closest}$ to s_{rand}. The final step in the sampling process is to determine the radius neighbours, S_{near}, of the point s_{new}.

The second process is that of *rewiring*. During this process, we determine which of the points in S_{near} we should connect to s_{new}, i.e., determining which path to s_{new} results in a lowest cost path. Finally we repeat the process for the parents of S_{near}. In other words, the tree is rewired locally around the new point, such that lower global cost paths arise.

Repeating these two processes interchangably for a given time budget T, solves (2) for a subset $\tilde{Z}_{o,g}$ that is determined by the randomly sampled points. As $T \to \infty$, RRT* minimises over the entire $Z_{o,g}$, i.e., it is asymptotically optimal in time, (while A* is asymptotically optimal in resolution).

3.2 IRL for Path Planning

Path planning involves finding a (near) optimal path to the goal given a cost function. In the inverse problem, we are given example paths and must find the cost function for which these paths are (near) optimal. The example paths comprise a dataset $\mathcal{D} = (\zeta^1_{o_1,g_1}, \zeta^2_{o_2,g_2} ... \zeta^D_{o_D,g_D})$ where $\zeta^i_{o_i,g_i}$ is an example path with initial and final configurations o_i, g_i respectively. We assume the unknown cost function is of the form,

$$c(s_i, s_j) = \mathbf{w}^T \mathbf{f}(s_i, s_j), \tag{3}$$

where $\mathbf{f}(s_i, s_j)$ is a K-dimensional vector of features that encode different aspects of the configuration pair and $\mathbf{w}$ is a vector of unknown weights to be learned. Since $\mathbf{w}$ is independent of the configuration, we can express the total cost of the path in a parametric form:

$$C(\zeta) = \mathbf{w}^T \sum_{i=0}^{l_\zeta - 1} \mathbf{f}(s_i, s_{i+1}) := \mathbf{w}^T \mathbf{F}(\zeta), \tag{4}$$

where $\mathbf{F}(\zeta)$ is the *feature sum* of the path.

While many formulations of the inverse problem exist, the general idea is to find a weight vector that assigns less cost to the example paths than all other possible paths with the same initial and goal configuration. This can be formalised by a set of inequality constraints:

$$C(\zeta^i_{o_i,g_i}) \leq C(\zeta) \quad \forall \zeta \in Z_{o_i,g_i} \quad \forall i. \tag{5}$$

The constraint is an inequality because $Z_{o,g}$ contains only paths available to the planner and thus may not include the example path $\zeta^i_{o_i,g_i}$. Z_{o_i,g_i} can be large but if we have an optimisation procedure that solves (2), it is enough to satisfy,

$$C(\zeta^i_{o_i,g_i}) \leq \min_{\zeta \in Z_{o_i,g_i}} C(\zeta) \quad \forall i, \tag{6}$$

Ratliff et al. [4] propose a maximum margin variant of (6) by introducing a margin function $L_i(\zeta)$ that decreases the cost of the proposed path ζ if is dissimilar to $\zeta^i_{o_i,g_i}$. For example, $L_i(\zeta)$ could be -1 times the number of configurations in the demonstration path not visited by ζ. This margin is very similar to the one encountered in Support Vector Machines. The intuition is that by requiring the model to fit the data well even if we are using a margin against, will result in better generalisation. The full optimisation formulation of Maximum Margin Planning is as follows.

$$\operatorname*{argmin}_{\mathbf{w},\tau} \frac{1}{2}||\mathbf{w}||^2 + \frac{\lambda}{D}\sum_i \tau_i \tag{7}$$

$$\text{s.t.} \quad C(\zeta^i_{o_i,g_i}) - \tau_i \leq \min_{\zeta \in Z_{o_i,g_i}} C(\zeta) + L_i(\zeta) \quad \forall i, \tag{8}$$

where τ_i are slacks that can be used to relax the constraints. Rearranging the inequality in terms of the slacks we get:

$$C(\zeta^i_{o_i,g_i}) - \min_{\zeta \in Z_{o_i,g_i}} C(\zeta) + L_i(\zeta) \leq \tau_i \quad \forall i. \tag{9}$$

Consequently, the $\mathbf{w}$ minimising:

$$\frac{1}{2}||\mathbf{w}||^2 + \frac{\lambda}{D}\sum_i \big(C(\zeta^i_{o_i,g_i}) - \min_{\zeta \in Z_{o_i,g_i}} \big(C(\zeta) + L_i(\zeta)\big)\big) \tag{10}$$

is equivalent to that which minimizes (7), i.e., the slacks are tight. The minimum can be found by computing a subgradient and performing gradient descent on the above objective:

$$\nabla_{\mathbf{w}} = \mathbf{w} + \frac{\lambda}{D}\sum_{i=0}^{D} F(\zeta^i_{o_i,g_i}) - F(\tilde{\zeta}^*_{o_i,g_i}), \tag{11}$$

where,

$$\tilde{\zeta}^*_{o_i,g_i} = \operatorname*{argmin}_{\zeta \in Z_{o_i,g_i}} C(\zeta) + L_i(\zeta). \tag{12}$$

The inverse problem can therefore be seen as an iterative procedure, that first solves (12) in the inner loop while keeping the weights constant. Based on that solution, it updates the weights using (11) in the outer loop. The weights at convergence represent the cost function that is used to plan the future behaviour of the agent. In [4], A* search was used for planning in the inner loop, assuming that the domain contained acyclic positive costs. In this paper, we make the same assumptions but develop methods that use RRT* for planning.

4 Method

In this section, we propose Rapidly Exploring Learning Trees (RLT*). We first propose a generic extension to the maximum margin approach that we call Approximate Maximum Margin Planning. We then show how an implementation of this approach with an RRT* planner and a novel caching scheme yields RLT*.

4.1 Approximate Maximum Margin Planning

In Section 3.2, we showed how the multiple constraints of (5) could be reduced to a single constraint for each demonstration in (6). We now assume access to a mechanism of sampling different paths from $Z_{o,g}$ and their respective costs. For a given finite time budget T, this path sampler therefore samples a subset $\tilde{Z}_{o,g} \in Z_{o,g}$. Thus, we can modify (6), and demand that our cost function satisfies,

$$C(\zeta^i_{o_i,g_i}) \leq \min_{\zeta_{o_i,g_i} \in \tilde{Z}_{o_i,g_i}} C(\zeta) \quad \forall i. \tag{13}$$

As the planning budget T increases, the sample-based planner samples lower cost paths, making this inequality harder to satisfy. Assuming $\tilde{Z}_{o_i,g_i}$ is constant, we can rewrite (10) as:

$$\frac{1}{2}||\mathbf{w}||^2 + \frac{\lambda}{D}\sum_i \big(C(\zeta^i_{o_i,g_i}) - \min_{\zeta \in \tilde{Z}_{o_i,g_i}} \big(C(\zeta) + L_i(\zeta)\big)\big). \tag{14}$$

This gives rise to an approach we call Approximate Maximum Margin Planning (AMMP), which is similar to the one described in Section 3.2, with the crucial difference that the planning step is executed by a sample-based planner and not a deterministic one, like A*. An important consequence is that $\tilde{Z}_{o_i,g_i}$ now changes every time we invoke the sample-based planner. As a result, AMMP can be thought of as sampling *constraints* that we want our cost function to satisfy.

4.2 Rapidly Exploring Learning Trees

A simple way to construct a concrete algorithm from AMMP is to use RRT* as the sample-based planner. The success of RRT* in path planning domains reassures us the the resulting AMMP algorithm will be able to sample low cost paths, allowing us to learn a good cost function. However, this results in a computationally expensive algorithm, which calls the planner $I \times |D|$ times over I iterations given a dataset of size $|D|$. Since planning is typically expensive, it is crucial to find a more efficient approach. In this section, we propose Rapidly Exploring Learning Trees (RLT*), which implements AMMP with an RRT* planner using a caching scheme to achieve computational efficiency.

Two of the most costly operations of RRT* are 1) finding the nearest neighbour to a newly sampled point and 2) finding the radius-neighbours of a newly created vertex in the tree [6]. However, a key observation is that neither of these procedures depend on the cost function used by the planner. Thus, RRT* can be split in two independent steps.

The first step is described in Algorithm 1, which takes as input p, the number of points to randomly sample from free space; s_{init}, the initial point; and η, the steer step size. For each randomly sampled point s_{rand}, we find the nearest neighbour, $s_{nearest}$, from the set of points in the vertex set V. We then create a new configuration point s_{new} by steering from $s_{nearest}$ to s_{rand}. Next, we query the radius neighbours of s_{new} at a radius determined by $\min\{\gamma_{RRT^*}(\frac{\log(|V|)}{|V|})^{\frac{1}{d}}, \eta\}$. Here, d is the dimensionality of S, and γ_{RRT^*} is a constant based on the volume of free space (see [6]). The points s_{new}, $s_{nearest}$ and the set S_{near} are stored together in the map P, which we call the *point cache*, and is returned at the end of the procedure. This algorithm turns the sampling process of RRT* into a preprocessing step. Consequently, the expensive `Nearest` and `Near` procedures only need to be repeated $|D|$ times instead of $I \times |D|$ times.

Algorithm 1 cacheRRT(p,s_{init},η)

```
1: $P \leftarrow \emptyset$                    {Initialise the point cache}
2: $V \leftarrow s_{init}$
3: for $i = 0 \ldots p$ do
4:     $s_{rand} \leftarrow SampleFree_i$
5:     $s_{nearest} \leftarrow \texttt{Nearest}(V, s_{rand})$
6:     $s_{new} \leftarrow \texttt{Steer}(s_{nearest}, s_{rand})$
7:     $S_{near} \leftarrow \texttt{Near}(V, s_{new}, \min\{\gamma_{RRT^*}(\frac{\log(|V|)}{|V|})^{\frac{1}{d}}, \eta\})$
8:     $V \leftarrow V \cup s_{new}$
9:     $P \leftarrow P \cup \{s_{new}, S_{near}\}$
10: end for
11: return $P$
```

The output of Algorithm 1 is input to Algorithm 2, which resembles wiring and re-wiring procedures in RRT* [6], and returns a minimum cost path to the goal. An important difference, however, is that the vertices of the tree and their neighbours at each iteration are already known and contained within the point cache. This speeds computation while keeping consistency between the planners used during learning and final execution. As learning proceeds and the cost function changes, so does the wiring of this tree; however, the points involved do not change.

Algorithm 3 describes Rapidly Exploring Learning Trees (RLT*), which uses Algorithms 1 and 2. First, we initialise the weights, either randomly or using a cost function that simply favours shortest paths. Then, for each datapoint ζ_i, we calculate feature sums and run `cacheRRT`. The main learning loop involves cycling through all data points and finding the best path under a loss-augmented cost function. The feature sums of this path are calculated and subsequently the difference with the demonstrated feature sums is computed. At the end of each iteration, an average gradient is calculated and the cost function is updated. At convergence, the learned weights are returned.

Algorithm 2 planCachedRRT*(P,s_{init},$c()$)

```
1: $E \leftarrow \emptyset$
2: $V \leftarrow s_{init}$
3: for $i = 0 \ldots |P|$ do
4:     $s_{nearest} \leftarrow P\{s^i_{nearest}\}$
5:     $s_{new} \leftarrow P\{s^i_{new}\}$
6:     $S_{near} \leftarrow P\{S^i_{near}\}$
7:     $V \leftarrow V \cup s_{new}$
8:     $s_{min} \leftarrow s_{nearest}$
9:     $c_{min} \leftarrow \texttt{Cost}(s_{nearest}) + c(s_{nearest}, s_{new})$
10:    for $s_{near} \in S_{near}$ do
11:        $c_{near} \leftarrow \texttt{Cost}(s_{near}) + c(s_{near}, s_{new})$
12:        if $\texttt{CollisionFree}(s_{near}, s_{new})$ and $c_{near} < c_{new}$ then
13:            $s_{min} \leftarrow s_{near}; c_{min} \leftarrow c_{near}$
14:        end if
15:    end for
16:    $E \leftarrow E \cup \{(s_{min}, s_{new})\}$
17:    for $s_{near} \in S_{near}$ do
18:        $c_{new} \leftarrow \texttt{Cost}(s_{near}) + c(s_{near}, s_{new})$
19:        if $\texttt{CollisionFree}(s_{near}, s_{new})$ and $c_{new} < \texttt{Cost}(s_{near})$ then
20:            $s_{parent} \leftarrow \texttt{Parent}(s_{near})$
21:            $E \leftarrow E \setminus (s_{parent}, s_{near}) \cup (s_{new}, s_{near})$
22:        end if
23:    end for
24: end for
25: $\zeta_{min} \leftarrow \texttt{minCostPath}(V, E, c())$
26: return $\zeta_{min}$
```

Algorithm 3 RLT*($D, p, \eta, \lambda, \delta$)

```
1: $\mathbf{w} \leftarrow \texttt{initialiseWeights}$
2: $\tilde{\mathbf{F}} \leftarrow \emptyset$
3: $R \leftarrow \emptyset$
4: for $\zeta^i$ in $D$ do
5:     $\tilde{F}_{\zeta^i} \leftarrow \texttt{FeatureSums}(\zeta^i)$
6:     $\tilde{\mathbf{F}} \leftarrow \tilde{\mathbf{F}} \cup \tilde{F}_{\zeta^i}$
7:     $r_i \leftarrow \texttt{cacheRRT}(p, s^{\zeta^i}_{init}, \eta)$
8:     $R \leftarrow R \cup r_i$
9: end for
10: repeat
11:    $\nabla_{\mathbf{w}} \leftarrow 0$
12:    for $\zeta^i$ in $D$ do
13:        $c() \leftarrow \texttt{getCostmap}(\mathbf{w}) + L(\zeta^i)$
14:        $r_i \leftarrow R\{i\}$ ; $\tilde{F}_i \leftarrow \tilde{\mathbf{F}}\{i\}$
15:        $\zeta \leftarrow \texttt{planCachedRRT}^*(r_i, x^i_{init}, c())$
16:        $F_i \leftarrow \texttt{FeatureSums}(\zeta)$
17:        $\nabla_{\mathbf{w}} \leftarrow \nabla_{\mathbf{w}} + \tilde{F}_i - F_i$
18:    end for
19:    $\nabla_{\mathbf{w}} \leftarrow \mathbf{w} + \frac{\lambda}{|D|}\nabla_{\mathbf{w}}$
20:    $\mathbf{w} \leftarrow \mathbf{w} - \delta\nabla_{\mathbf{w}}$
21: until convergence
22: return $V, E$
```

For RRT* the dependance of $\tilde{Z}_{o_i,g_i}$ on the time budget T is hard to quantify since it depends on the size and nature of S as well as the cost function we are using -which also changes with every iteration. For this reason, we resort to an experimental assesment of the ability of RRT* to sample the right constraints at every iteration of RLT* and hence allow the learning of a cost function from demonstration.

5 Experiments

In this section, we experimentally compare RLT* to the original MMP algorithm implemented using an A* planner and an ablated version of RLT* that does not use caching.

Our experiments take place in the context of socially intelligent navigation. IRL has been widely used in this setting [15; 7; 9] because it is usually infeasible to hard-code the cost functions that a planner should use in complex social situations. Having the ability to quickly and effectively learn social navigation cost functions from demonstration would be a major asset for robots that operate in crowded environments such as airports [16], museums [17] and care centres [18].

5.1 Setting

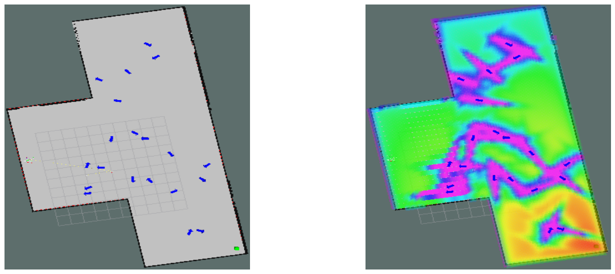
Our experiments take place in a randomly generated social environment, shown in Figure 1a. Every arrow in the figure represents a person's position and orientation. The robot is given the task of navigating from one point in the room to another. While it is aware of the orientation and position of different people, it has no idea on how to trade off reaching the target quickly with avoiding people and obstacles, i.e., the cost function is unknown. Instead, the robot is given a dataset of demonstrations D. Each demonstration ζ_i is a set of configurations $s = (x, y)$ representing positions of the robot in the configuration space and each demonstration takes place for a different random configuration of the social environment, i.e., the people are at different positions and orientations every time. The task of the robot is to, using D, extract a cost function based on different features of the environment, which in turn would allow it to behave socially in future tasks.

The features we use can be divided in three sets. The first set encodes proxemics to the people present in the scene. These are represented by three Gaussian functions of different means and diagonal covariances around each person. The second set of features encodes the distance from the target location using linear, exponential and logarithmic functions. The third set encodes the obstacle cost using a stable function of the reciprocal of the distance from the nearest obstacle. Figure 1b shows an example cost function over the whole configuration space for the configuration in Figure 1a. We use different functions for human and target proximity, to allow for more degrees of freedom when modeling the underlying cost function. Sufficient regularisation ensures that that the model does not overfit.

5.2 Evaluation

To quantitatively assess the quality of our algorithms, we generate a dataset D by planning near-optimal paths from an initial configuration s_o to a goal configuration s_g under a ground-truth cost function $c_{gt}()$ derived from ground-truth weights $\mathbf{w}_{gt}$. A fully optimal path can only be derived only asymptotically in terms of either time for RRT*, or resolution for A*. In practice, however, we found that planning for 60 seconds using RRT* achieves a path that is nearly optimal, as running longer leads to negligible changes in path cost.

The resulting ground truth dataset is extremely useful for evaluation. For each path ζ generated by the learner, we know its cost under the ground-truth cost function is simply



(a) Example setting (b) Cost function

Figure 1: (a) An instance of the randomised social navigation task. Arrows denote the position and orientation of people in the scene. The robot is on the lefthand side of the map and the green box in the bottom right denotes the goal location. (b) The corresponding cost function for the random scenario. Red denotes *low* cost, while purple denotes *high* cost.

$\mathbf{w}_{gt}\mathbf{F}(\zeta)$. Furthermore, we can compute the cost difference between the generated path and the example path:

$$Q(\zeta, \zeta_i, \mathbf{w}) = \mathbf{w}(\mathbf{F}(\zeta) - \mathbf{F}(\zeta_i)), \tag{15}$$

which is our primary performance metric. Note that, if the demonstration path ζ_i is optimal under $\mathbf{w}$, then $Q(\zeta, \zeta_i, \mathbf{w}) >= 0$.

For our experiments, the ground-truth weights $\mathbf{w}_{gt}$ were chosen to induce a cost function that penalises passing in front of people. In addition, small weights were added to the other two Gaussian functions for each person, as well as linear and exponential penalisation of the distance from the goal, so that the cost function, shown in Figure 1b, would not be too trivial.

We also measure the time per iteration of each learning algorithm. All algorithms were implemented in Python, share similar functions, and were not optimised for speed apart from the caching scheme in RLT*. Finally we perform qualitative evaluation by visually comparing the learned cost functions for each algorithm and the paths they generate against their respective ground truth

5.3 Results

Our dataset D consists of 20 trajectories at random social situations within the social environment shown in Figure 1a, using the cost function shown in Figure 1b. Half of these trajectories make up the training dataset D_{train} and the other half the test dataset D_{test}. After being trained on D_{train}, the performance of a cost function is evaluated on D_{test} using (15). The process is repeated four times for the same dataset but with different random compositions of D_{train} and D_{test}. We report both the mean and standard error across the different trajectories in the test and training sets, for each iteration in Figures 2a and 2b.

We report results for RLT*, RLT* without caching, and MMP with A* at grid resolutions of 0.8 and 0.3 metres. For RLT*, we set the number of sampled points $p = 2500$. For RLT* without caching, we cap planning at $12s$, which is about how long RRT* plans for when $p = 2500$. Note that this is much less planning time than the $60s$ used to generate

	$\text{MMP}_{0.8}$	$\text{MMP}_{0.3}$	RLT*-NC	RLT*
Iteration (s)	1.83(0.79)	20.93(12.14)	12(0)	5.87(0.50)
Learning (s)	275.2	3140.6	1808	911.7
$Q(\zeta, \zeta_i, \mathbf{w}_{gt})$	7.28	3.14	0.57	1.57

Table 1: Per iteration and total learning times for our proposed algorithms and the baselines (RLT*-NC is RLT* without caching.)

the demonstrations. All learning algorithms were initialised using a cost function that only favours shortest paths.

The results show that both versions of RLT* always outperform both versions of MMP with A*. This can be attributed to the fact that the underlying RRT* planner is not confined to work on a fixed grid, allowing it to generate paths that are closer to optimal. This is further reinforced by the increased performance as the grid resolution decreases allowing $\text{MMP}_{0.3}$ to approach the performance of RLT*. Standard error rates are similar across methods.

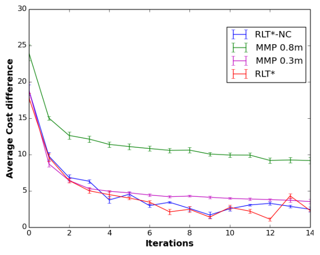
Table 1 shows the average planning time per learning iteration and the total learning time for all algorithms, along with their average cost difference on the test set at convergence. Comparing RLT* with and without caching shows that the cached version is much faster without a significant cost in performance. Furthermore both algorithms, although slower than $\text{MMP}_{0.8}$, are faster than $\text{MMP}_{0.3}$. This means that, even if a higher resolution MMP was able to match the performance of RLT*, it would be much slower. Furthermore, there is large variance in planning time for $\text{MMP}_{0.3}$. At the start of learning, the initial cost function is very simple and and only involves the distance from the goal location. Under this cost function, planning is quick, because a simple heuristic that simply takes into account the distance from the goal is a good approximation of the cost-to-go. As learning proceeds, however, the cost function becomes more complex, and this simple heuristic, although admissible, is no longer tight. This requires the A* planner to expand many more states before an optimal path is found. We can see therefore that MMP with A* scales poorly, not only the size of S, but also in the complexity of the cost function. By contrast, the probabilistic nature of RLT* makes it less susceptible to these pathologies.

For a qualitative comparison, we look to Figures 3 and 4. Figure 3 shows a demonstrated path (black) along with the paths generated by three of the four algorithms. Note the effect of planning on a coarse grid in the case of $\text{MMP}_{0.8}$ (green). We can also see that RLT* (red) more faithfully replicates the example path. Figure 4 compares the ground truth cost function (Figure 4a) against the learned cost functions for $\text{MMP}_{0.3}$ and RLT* (Figures 4b, 4c). This comparison shows that $\text{MMP}_{0.3}$ overestimates the cost related with the distance from the goal, i.e., the cost increases faster as we move away from the goal. This yields paths that reach the goal earlier, possibly accounting for the small difference in performance between the two algorithms.

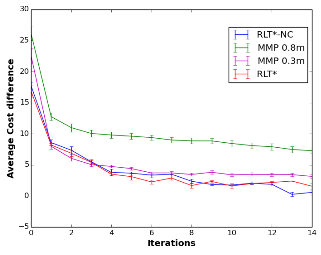
6 Conclusion and Future Work

In this paper, we proposed Rapidly Exploring Learning Trees (RLT*), which learns the cost functions of Rapidly Exploring Random Trees (RRT) from demonstration, thereby making inverse learning methods applicable to more complex tasks. Our approach extends the Maximum Margin Planning to work with RRT* cost functions. Furthermore, we proposed a caching scheme that greatly reduces the computational cost of this approach. Experimental results on real and simulated data from a social navigation scenario showed that RLT* achieves better performance at lower computational cost.

Although our experimental comparison used A* as a baseline, RLT* is also well suited to situations where deterministic planners naturally fail, e.g., a manipulator with many degrees of freedom. In future work we aim to design learning algorithms that explicitly take into account the sampling nature of RRT*. We also plan to evaluate RLT* on data gathered from real robots.



(a) Train



(b) Test

Figure 2: Train and test set average cost difference, for 15 iterations. Error bars represent standard error over four independent runs on shuffled versions of the data. (RLT*-NC is RLT* without caching.)

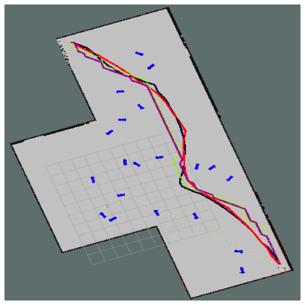
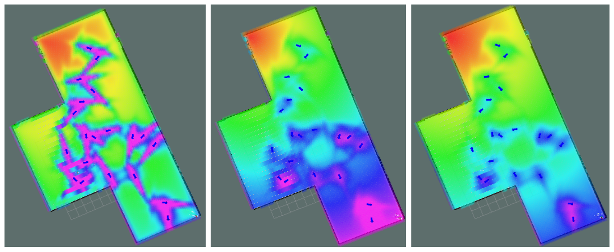


Figure 3: Qualitative comparison of paths. Goal is at the top left, robot begins on the bottom right. Black: demonstration path. Red: RLT*. Green: $\text{MMP}_{0.8}$, Magenta: $\text{MMP}_{0.3}$



(a) Ground truth (b) $\text{MMP}_{0.3}$ (c) RLT*

Figure 4: Ground truth and learned cost functions using $\text{MMP}_{0.3}$ and RLT*.

References

[1] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and autonomous systems*, vol. 57, no. 5, pp. 469–483, 2009.

[2] R. E. Kalman, "When is a linear control system optimal?" *Journal of Basic Engineering*, vol. 86, no. 1, pp. 51–60, 1964.

[3] P. Abbeel and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," in *Proceedings of the twenty-first International Conference on Machine Learning (ICML)*. ACM, 2004, p. 1.

[4] N. D. Ratliff, J. A. Bagnell, and M. A. Zinkevich, "Maximum margin planning," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 729–736.

[5] S. M. LaValle, "Rapidly-exploring random trees a new tool for path planning," 1998.

[6] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[7] P. Henry, C. Vollmer, B. Ferris, and D. Fox, "Learning to navigate through crowded environments," in *Proceedings of the IEEE International Conference on Robotics & Automation (ICRA)*. IEEE, 2010, pp. 981–986.

[8] P. Abbeel, D. Dolgov, A. Y. Ng, and S. Thrun, "Apprenticeship learning for motion planning with application to parking lot navigation," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*. IEEE, 2008, pp. 1083–1090.

[9] D. Vasquez, B. Okal, and K. O. Arras, "Inverse reinforcement learning algorithms and features for robot navigation in crowds: an experimental comparison," in *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. IEEE, 2014, pp. 1341–1346.

[10] B. D. Ziebart, "Modeling purposeful adaptive behavior with the principle of maximum causal entropy," Ph.D. dissertation, Carnegie Mellon University, 2010.

[11] S. Levine and V. Koltun, "Continuous inverse optimal control with locally optimal examples," in *ICML '12: Proceedings of the 29th International Conference on Machine Learning*, 2012.

[12] N. Ratliff, B. Ziebart, K. Peterson, J. A. Bagnell, M. Hebert, A. K. Dey, and S. Srinivasa, "Inverse optimal heuristic control for imitation learning." AISTATS, 2009.

[13] A. Byravan, M. Monfort, B. Ziebart, B. Boots, and D. Fox, "Graph-based inverse optimal control for robot manipulation," in *Proceedings of the International Joint Conference on Artificial Intelligence*, 2015.

[14] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "CHOMP: Gradient optimization techniques for efficient motion planning," in *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*. IEEE, 2009, pp. 489–494.

[15] B. Okal and K. O. Arras, "Learning socially normative robot navigation behaviors with bayesian inverse reinforcement learning."

[16] R. Triebel, K. Arras, R. Alami, L. Beyer, S. Breuers, R. Chatila, M. Chetouani, D. Cremers, V. Evers, M. Fiore, *et al.*, "Spencer: A socially aware service robot for passenger guidance and help in busy airports," 2015.

[17] S. Thrun, M. Bennewitz, W. Burgard, A. B. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schulte, *et al.*, "Minerva: A second-generation museum tour-guide robot," in *Robotics and automation, 1999. Proceedings. 1999 IEEE international conference on*, vol. 3. IEEE, 1999.

[18] K. Shiarlis, J. Messias, M. Someren, S. Whiteson, J. Kim, J. Vroon, G. Englebienne, K. Truong, V. Evers, N. Pérez-Higueras, *et al.*, "TERESA: a socially intelligent semi-autonomous telepresence system," 2015.