Rapidly Exploring Learning Trees

Kyriacos Shiarlis[1], Joao Messias[1], and Shimon Whiteson[2]

***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 the need to repeat costly planning procedures. In this paper, we propose Rapidly Exploring Learning Trees (RLT*), which learns the cost functions of Optimal Rapidly Exploring Random Trees (RRT*) from demonstration, thereby making inverse learning methods applicable to more complex tasks. Our approach extends 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 and real-robot data from a social navigation scenario show that RLT* achieves better performance at lower computational cost than existing methods. We also successfully deploy control policies learned with RLT* on a real telepresence robot.**

I. Introduction

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

In addition, learned cost functions are often useful even when the environment changes. For example, if the friction in a robot's wheels changes due to wear and tear, the optimal policy will change but the cost function will not. Thus, a robot trained via supervised learning would need to learn a new policy, while a robot trained via IRL could simply replan on the new dynamics with its existing cost function. In addition, cost functions are thought to be more succinct representations of the aims of the agent [3]. For example, a robot whose aim is to reach a goal as fast as possible may have a simple cost function but a complex policy.

IRL is an iterative process that, in an inner loop, solves the forward planning problem (i.e., finds an optimal policy under the current cost function) and subsequently, in an outer loop, updates the current cost function. This process repeats until convergence. For continuous domains such as robotics, solving the planning problem exactly is typically intractable. Doing so may result in high-dimensional models to which most planning algorithms scale poorly, necessitating a coarse discretisation that yields poor performance. Consequently, in robotics, planning is often done using sample-based, single-query motion planners such as Rapidly Exploring Random Trees (RRTs) [4] and their variants [5]. Such methods can cope with high-dimensional continuous domains, as well as the presence of obstacles in the environment and motion constraints on the robot.

In this paper, we first propose Approximate Maximum Margin Planning (AMMP), a variant of Maximum Margin Planning (MMP) [6] that does not assume an optimal planner. Doing so allows us to use sample-based planning algorithms, such as RRT* in the inner loop of IRL. Second, we propose a caching scheme that both improves performance and reduces the computational cost when RRT* is used within AMMP. The resulting algorithm, which we call Rapidly Exploring Learning Trees (RLT*), allows computationally efficient IRL in high-dimensional, continuous domains with obstacles and motion constraints.

We evaluate RLT* on real and simulated data from a social navigation scenario. The results demonstrate that, in the absence of obstacles and motion constraints, RLT* performs better and is faster than both MMP and RLT* without caching. Furthermore, we show that RLT*, unlike MMP, can learn cost functions in robotic tasks with obstacles and motion constraints. Finally, we deploy our method on a real telepresence robot using data from human demonstrations.

II. Related Work

Substantial research has applied IRL to robotics [7], [8], [9]. A big challenge in doing so is solving the forward planning problem under the current cost function at every iteration. The planning problem is especially intractable in robotics because the state-action spaces encountered are often continuous and high dimensional.

One approach is to discretise the state-action spaces and solve the resulting Markov decision process (MDP). However, in high-dimensional tasks, fine discretisations render planning intractable and coarse discretisations yield poor performance.

Some methods that use discretisation formulate the problem such that only an open-loop path (rather than a closed-loop policy) is required. Then, given deterministic dynamics, planning can be performed using A* [6], allowing realistic problems to be tackled. We take a similar approach but, by planning using RRT*, we substantially enlarge the class of tasks that can be tackled.

Another approach is to use hybrid planners. Inverse Optimal Heuristic Control [10] models the long-term goals of the agent as a coarsely discretised MDP, to ensure tractability, while using supervised learning to determine the local policy

[1]Informatics Institute, University of Amsterdam, The Netherlands, {k.c.shiarlis,j.messias}@uva.nl

[2]Department of Computer Science, University of Oxford, United Kingdom, shimon.whiteson@cs.ox.ac.uk

of the agent at each state. Graph-Based IOC [11] uses discrete optimal control on a coarse graph and the actual path is executed using local trajectory optimisation techniques such as CHOMP [12], which would otherwise suffer from local minima. However, these methods employ complex and domain-specific planning formulations that are not suitable for all robotics tasks. By contrast, our approach employs a widely used planner, making it versatile and easy to implement.

Another recent method is Adaptive State Graphs [13], which builds a state-action 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 the underlying cost function is different from the one used to execute the robot's behaviour. This can have adverse effects since IRL assumes that the demonstration paths came from the same planner used during learning. Instead of building a controller graph first and then using a different controller to optimise trajectories, RLT* builds a controller tree on the fly and uses the same planner for learning and execution.

All of the above methods require a model of the dynamics of the system. Another paradigm, that of model-free IRL, avoids the need to explicitly plan, employing sampling instead. As a result it tends to scale better to high-dimensional state-action spaces and does not require a model of the system dynamics, [14], [15]. Model free IRL samples trajectories, usually starting from the initial conditions observed in the data. The probability of these trajectories is weighted according to the current cost function, which is then updated to make trajectories closer to the data more likely. A big challenge is to find an appropriate importance distribution at each iteration. [16] considers guiding the importance distribution by simultaneously learning a good policy under the current cost function and using it as an adaptive importance sampler. However, since it uses a modified version of Linear Quadratic Regulator control the derived policies are only locally optimal and can produce highly suboptimal paths or even fail in the presence of obstacles. By contrast, RRTs cope naturally with obstacles, as long as they are static and mapped, and is also asymptotically optimal.

III. Background

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

A. Path Planning

Path planning occurs in a space $\mathcal{S}$ of possible robot configurations. 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.

As there could be several paths to the goal, planners often employ a *cost functional*, $C(\zeta)$, which typically sums the costs between two subsequent configurations $c(s_i, s_j)$:

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

This cost functional is similar to the one used in optimal control and analogous to the return used in reinforcement learning. Given the cost functional, the path planner seeks an optimal path ζ^*, which satisfies,

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

where $Z_{o,g}$ is the set of 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 $\tilde{Z}_{o,g} \subset Z_{o,g}$, whose size depends on the discretisation. However, such methods scale poorly in the size of $\mathcal{S}$. Furthermore, such algorithms ignore motion constraints as they assume all nodes in the graph can be reached by their neighbours in exactly the same way. This assumption does not hold, e.g., for non-holonomic robots, where the ortientation at each node places constraints on how it can be reached. In this constrained space, it is less straightforward to define actions, neighbouring nodes, or an admissible heuristic, which is necessary for optimality.

These drawbacks motivate *sample-based* path planning algorithms such as RRT* [5]. 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 consists of two interleaved steps.

The first step is *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}. In a Euclidean space, steering between two points means simply connecting them with a straight line. However, if orientations and motion constraints are used, then becomes more complex. Finally, the sampling step determines the points, S_{near}, within a given radius of s_{new}.

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

Alternating between these two steps 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 [5]. By contrast, A* is asymptotically optimal in resolution.

B. IRL for Path Planning

In path planning, we are given a cost function and must find a (near) optimal path to the goal. 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 and g_i. 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_i,g_i} 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,

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

Maximum Margin Planning (MMP) [6] introduces a margin function $L_i(\zeta)$ that decreases the cost of the proposed path ζ if it 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. As in Support Vector Machines, requiring the model to fit the data well even in the presence of a margin improves generalisation. Furthermore, the margin helps address the ill-posed nature of IRL, i.e., many cost functions are consistent with the demonstrated behaviour. Formally, MMP solves the following optimisation problem:

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

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

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

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

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

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

also minimises (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. Given that solution, it then updates the weights using (11) in the outer loop. The weights at convergence represent the cost function that is used to plan future behaviour. In [6], 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.

IV. Method

In this section, we propose Rapidly Exploring Learning Trees (RLT*). We first propose a generic extension to MMP 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*.

A. Approximate Maximum Margin Planning

Section III-B shows how the multiple constraints of (5) can be reduced to the single constraint of (6) for each demonstration. However, this reduction requires an optimal planner to perform the minimisation in (5), which is impractical in many robot applications. Suppose instead that, as in RRT*, we have a mechanism for sampling different paths from $Z_{o,g}$ along with their respective costs and that, for a given finite time budget T, this path sampler samples a subset $\tilde{Z}_{o,g} \subset Z_{o,g}$, Then, we can modify (6) to 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 T increases, lower cost paths are sampled, 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). It is similar to MMP 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 $\tilde{Z}_{o_i,g_i}$ changes every time we invoke the sample-based planner. Thus, AMMP can be thought of as sampling the *constraints* that we want our cost function to satisfy. The main advantage of AMMP is that it is not bound by the restrictive assumptions of A*, such as a discrete state-action space and no motion constraints.

However, an ineffective sampler could yield poor gradient estimates and thus poor solutions. In fact, Ratliff et al. [12] argue that, for this reason, sample-based planners like RRT are not suited to learning cost functions from demonstration, as the paths sampled by RRT are heavily biased during the sampling process and thus highly suboptimal. However, asymptotically optimal sample-based planners like RRT* are better able to plan low-cost paths [5] and thus well suited for use within AMMP.

B. Rapidly Exploring Learning Trees

As suggested above, a simple way implement AMMP is to use RRT* as the sample-based planner. RRT* can sample low-cost paths, allowing AMMP to learn a good cost function. Furthermore, RRT* can cope with large and continuous state-action spaces with motion constraints. However, the result is a computationally expensive algorithm that calls the planner $I \times |D|$ times over I iterations, given a dataset of size $|D|$. Furthermore, sampling a separate set of points at every iteration could produce noisy gradients that negatively affect convergence. In this section, we propose Rapidly Exploring Learning Trees (RLT*), which implements AMMP with an RRT* planner using a novel caching scheme to ensure both computational efficiency and more consistent gradients.

A key observation is that, in RRT*, only the rewiring step depends on the cost function. The sampling step is independent of it and, especially when motion constraints are present, contains the most computationally expensive operations. These are: 1) fitting and querying a data structure to find the nearest neighbour and the radius neighbours to a newly sampled point, 2) applying the steer function to all neighbours to and from the newly sampled point, and 3) checking for obstacles in these paths. By contrast, the only potentially expensive operation in the rewiring step is querying the cost of an edge.

Therefore, a key idea behind RLT* is to perform the sampling step just once for each data point and cache it for reuse throughout learning. Then, at each iteration, only the rewiring step needs to be repeated as the cost function changes.

Algorithm 1 describes the caching step, 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, S_{near}, 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 [5]). Next, we determine which points in S_{near} can safely reach s_{new} through the chosen `Steer` function (lines 13-17). These forward paths are stored in $Paths_{fwd}$. We then perform the same procedure but this time checking the paths from s_{new} to S_{near} and store them in the set $Paths_{bwd}$ (lines 18-22). This algorithm turns the sampling process of RRT* into a preprocessing step. Consequently, the expensive `Nearest`, `Near`, `Steer` and `Safe` procedures only need to be repeated $|D|$ times instead of $I \times |D|$ times.

Algorithm 1 `cacheRRT`(n,s_{init},η)

1: $P \leftarrow \emptyset$ {Initialise the point cache}
2: $V \leftarrow s_{init}$
3: **for** $i = 0 \dots n$ **do**
4: $s_{rand} \leftarrow SampleFree_i$
5: $s_{nearest} \leftarrow \texttt{Nearest}(V, s_{rand})$
6: $s_{new} \leftarrow \texttt{Get}(s_{nearest}, s_{rand})$
7: $S_{near} \leftarrow \texttt{Near}(V, s_{new}, \min\{\gamma_{RRT^*}(\frac{\log(|V|)}{|V|})^{\frac{1}{d}}, \eta\})$
8: $Paths_{fwd} \leftarrow \emptyset$
9: $Paths_{bwd} \leftarrow \emptyset$
10: $S_{fwd} \leftarrow \emptyset$
11: $S_{bwd} \leftarrow \emptyset$
12: **for** $s_{near} \dots S_{near}$ **do**
13: $path_{fwd} = \texttt{Steer}(s_{near}, s_{new})$
14: **if** $\texttt{Safe}(path_{fwd})$ **then**
15: $Paths_{fwd} \leftarrow Paths_{fwd} \cup path_{fwd}$
16: $S_{fwd} \leftarrow S_{fwd} \cup s_{near}$
17: **end if**
18: $path_{bwd} = \texttt{Steer}(s_{new}, s_{near})$
19: **if** $\texttt{Safe}(path_{bwd})$ **then**
20: $Paths_{bwd} \leftarrow Paths_{bwd} \cup path_{bwd}$
21: $S_{bwd} \leftarrow S_{bwd} \cup s_{near}$
22: **end if**
23: **end for**
24: $V \leftarrow V \cup s_{new}$
25: $P \leftarrow P \cup \{s_{nearest}, s_{new}, S_{fwd}, S_{bwd}, Paths_{fwd}, Paths_{bwd}\}$
26: **end for**
27: **return** P

The output of Algorithm 1 is input to Algorithm 2, which resembles the rewiring procedure in RRT* [5] and returns a low-cost path to the goal. However, unlike RRT* rewiring, 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.

In addition to saving computation time, the use of caching encourages consistency between the gradients computed in each iteration, as they are estimated from the same sampled points. The effect on the gradients, which resembles that of momentum [17], can improve performance, as our results in the next section show.

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 assessment of the ability of RRT* to sample

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

$E \leftarrow \emptyset$
$V \leftarrow s_{init}$
for $i = 0 \dots |P|$ **do**
  $(s_{nearest}, s_{new}) \leftarrow P_i$
  $(S_{fwd}, S_{bwd}) \leftarrow P_i$
  $(Paths_{bwd}, Paths_{fwd}) \leftarrow P_i$
  $V \leftarrow V \cup s_{new}$
  $s_{min} \leftarrow s_{nearest}$
  $c_{min} \leftarrow \texttt{Cost}(s_{nearest}) + c(path_{s_{nearest}, s_{new}})$
  for $j = 0 \dots |S_{fwd}|$ **do**
    $s_{fwd} = S^j_{fwd}$
    $path_{fwd} = Paths^j_{fwd}$
    $C_{near} \leftarrow \texttt{Cost}(s_{fwd}) + c(path_{fwd})$
    if $C_{near} < c_{new}$ **then**
      $s_{min} \leftarrow s_{fwd}; c_{min} \leftarrow c_{near}$
    end if
  end for
  $E \leftarrow E \cup \{(s_{min}, s_{new})\}$
  for $j = 0 \dots |S_{bwd}|$ **do**
    $s_{bwd} = S^j_{bwd}$
    $path_{bwd} = Paths^j_{bwd}$
    $C_{new} \leftarrow \texttt{Cost}(s_{new}) + c(path_{bwd})$
    if $C_{new} < \texttt{Cost}(s_{near})$ **then**
      $s_{parent} \leftarrow \texttt{Parent}(s_{bwd})$
      $E \leftarrow E \setminus (s_{parent}, s_{bwd}) \cup (s_{new}, s_{bwd})$
    end if
  end for
end for
$\zeta_{min} \leftarrow \texttt{minCostPath}(V, E, c())$
return ζ_{min}

Algorithm 3 $\texttt{RLT}^*(D, p, \eta, \lambda, \delta)$

$\mathbf{w} \leftarrow \texttt{initialiseWeights}$
$\tilde{\mathbf{F}} \leftarrow \emptyset$
$R \leftarrow \emptyset$
for ζ^i in D **do**
  $\tilde{F}_{\zeta^i} \leftarrow \texttt{FeatureSums}(\zeta^i)$
  $\tilde{\mathbf{F}} \leftarrow \tilde{\mathbf{F}} \cup \tilde{F}_{\zeta^i}$
  $r_i \leftarrow \texttt{cacheRRT}(p, s^{\zeta^i}_{init}, \eta)$
  $R \leftarrow R \cup r_i$
end for
repeat
  $\nabla_{\mathbf{w}} \leftarrow 0$
  for ζ^iin D **do**
    $c() \leftarrow \texttt{getCostmap}(\mathbf{w}) + L(\zeta^i)$
    $r_i \leftarrow R\{i\}$; $\tilde{F}_i \leftarrow \tilde{\mathbf{F}}\{i\}$
    $\zeta \leftarrow \texttt{planCachedRRT}^*(r_i, x^i_{init}, c())$
    $F_i \leftarrow \texttt{FeatureSums}(\zeta)$
    $\nabla_{\mathbf{w}} \leftarrow \nabla_{\mathbf{w}} + \tilde{F}_i - F_i$
  end for
  $\nabla_{\mathbf{w}} \leftarrow \mathbf{w} + \frac{\lambda}{|D|} \nabla_{\mathbf{w}}$
  $\mathbf{w} \leftarrow \mathbf{w} - \delta \nabla_{\mathbf{w}}$
until convergence
return $\mathbf{w}$

the right constraints at every iteration of RLT* and hence effectively learn a cost function from demonstration.

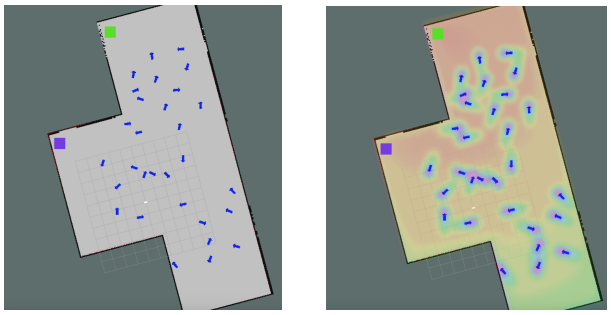
V. Experiments

We evaluate RLT* by comparing it to MMP, implemented using an A* planner, and RLT*-NC, an ablated version of RLT* that does not use caching. We consider three experimental settings: 1) a simulated holonomic robot, 2) a simulated non-holonomic robot, and 3) a real telepresence robot.

Our experiments take place in the context of socially intelligent navigation. IRL has been widely used in this setting [7], [9], [13] 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 [18], museums [19] and care centres [20].

A. Simulated Experiments

We first consider randomly generated social environments in simulation, such as the one shown in Figure 1a. Each 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 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 is to use D to extract a cost function that enables socially intelligent behaviour.



(a) Example setting (b) Cost function

Fig. 1: (a) A randomised instance of the social navigation task. Arrows denote the position and orientation of people in the scene. The robot is represented by the magenta box and the goal location is represented by the green box. (b) The corresponding cost function. Red denotes *low* cost, while purple denotes *high* cost.

The features we use can be divided into three categories. The first category encodes proxemics to the people present in the scene, i.e., the social features. Within this category we consider two variations, for reasons explained in the following section.

- **Social feature set 1 (S1)**: Three isotropic Gaussian functions with different means, centred in front, behind, and on the person.
- **Social feature set 2 (S2)**: Three field-of-view features. The features have a value of 1 if the robot is within a certain distance and angle from the person.

The second category of features encodes the distance from the target location using linear, exponential, and logarithmic functions. The third category 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 modelling the underlying cost function. Sufficient regularisation ensures that that the model does not overfit.

1) Evaluation: To evaluate our algorithms, we generate a dataset D by planning near-optimal paths from initial configurations s_o to goal configurations s_g under a ground-truth cost function $c_{gt}()$ derived from ground-truth weights $\mathbf{w}_{gt}$ and features $\mathbf{F}_{gt}$. A fully optimal path can only be derived asymptotically in terms of either time for RRT*, or resolution for A*. In practice, however, we found that planning for 100 seconds using RRT* achieves a path that is nearly optimal; running longer leads to negligible changes in path cost. The resulting ground truth dataset enables a quantatitive empirical evaluation, which is otherwise problematic in IRL [9], [21]. For each path ζ generated by the learning algorithm, we know its cost under the ground-truth cost function and features is simply $\mathbf{w}_{gt}^T\mathbf{F}_{gt}(\zeta)$. Furthermore, we can compute the *cost difference* between the learned path and the demonstrated path with respect to ground truth:

$$Q(\zeta, \zeta_i, \mathbf{w}_{gt}) = \mathbf{w}_{gt}^T(\mathbf{F}_{gt}(\zeta) - \mathbf{F}_{gt}(\zeta_i)), \qquad (15)$$

which is our primary performance metric. Note that, if the demonstration path ζ_i is optimal under $\mathbf{w}_{gt}$, then $Q(\zeta, \zeta_i, \mathbf{w}_{gt}) \geq 0$. For our holonomic simulated experiments, we consider two learning scenarios.

1) **Unknown weights**: only $\mathbf{w}_{gt}$ is unknown. The demonstrations and the learning algorithm share social feature set S1.
2) **Unknown weights and features**: $\mathbf{w}_{gt}$ and $\mathbf{F}_{gt}$ are unknown. S2 is used to generate the demonstrations and S1 is used for learning.

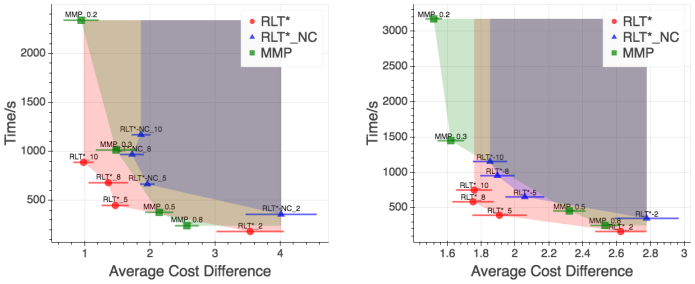
The first scenario evaluates each algorithm's ability to learn good cost functions when provided only with limited demonstrations of the task. The second scenario introduces a feature discrepancy to better simulate real-world settings, since it is unlikely that the features we define will exactly match those implicitly used by the human demonstrator.

We also document the total learning time for K iterations for the algorithms under comparison. 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 a qualitative evaluation by visually comparing the learned cost functions for each algorithm and the paths they generate against ground truth.

2) Holonomic Robot Results: Our dataset D consists of 20 trajectories from random social social situations. We split D into D_{train} and D_{test}, each with 10 trajectories. After training on D_{train}, the cost difference of a cost function is evaluated on D_{test} using (15). The process is repeated 7 times for the same dataset but with different random compositions of D_{train} and D_{test}. All learning algorithms are initialised using the same cost function that only favours shortest paths.

As mentioned earlier, planning time and grid resolution affect the performance of RRT* and A^*, respectively. To make a fair comparison, we vary these two quantities for each algorithm and plot cost difference against learning time at each setting. We can then identify which algorithms at which settings comprise the Pareto front, i.e., are undominated with respect to cost difference and learning time.

Figures 2a and 2b show the results for the two scenarios described earlier. MMP_X (green) refers to MMP with X meters of grid resolution while RLT_X (red) and RLT*-NC_X (blue) refer to RLT and RLT without caching, respectively, with X seconds of planning. The shading represents an interpolation of the performance between settings for each method. In this way an area of single colour illustrates hypothesized domination of that method over another, given the same amount of time. Since lower is better for both cost difference and learning time, the closer a point is to the bottom left corner, the better.



(a) Unknown weights scenario.

(b) Unknown weights and features scenario.

Fig. 2: Learning time vs. cost difference on the holonomic robot for MMP (green), RLT*-NC (blue), and RLT (red) at different planning fidelities. On both axes *lower is better*.

RLT* (red) comprises a large majority of the Pareto front, demonstrating good performance and generalisation in reasonable time. However, the performance of RLT* and RLT*-NC significantly degrades at very low planning times (2 seconds) because AMMP cannot sample good enough paths to compute a useful gradient.

Note that caching not only speeds learning in RLT*, it also modestly reduces cost differences. This suggests that the caching scheme introduces extra robustness and generalisation capabilities within the algorithm. As mentioned in Section IV-B, we hypothesise that the caching scheme improves learning by making the gradients smoother and more consistent, as with momentum. To confirm this, we plot the inner product between successive gradients during learning in Figure 3. The plot confirms that subsequent gradients in RLT* are more similar.

Finally, note that MMP's learning time scales exponentially with the size of the grid. This, however, is not solely due to the graph getting larger but also because A* search scales poorly with the complexity of the cost function itself. Since it relies on an admissible heuristic that for complex cost functions is no longer tight, A* must expand many more states. RLT* is less susceptible to such problems.

3) Non-Holonomic Robot Results: As mentioned earlier, a potential advantage of RLT* is that it can efficiently handle

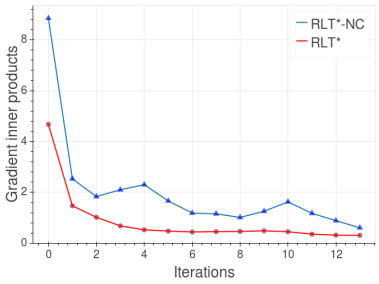


Fig. 3: Inner product of succcessive gradients during learning. The smoother curve for RLT* suggests that fixing the sampled points during learning makes the gradients more consistent.

TABLE I: Non-holonomic robot results.

	RLT*	RLT-NC
Average Cost Difference	0.1	2.31
Learning time (s)	1320	18530

learning in the presence of motion constraints. In this section, we consider a robot planning in a three-dimensional space, (x, y, θ), representing the position and orientation of the robot. The robot's motion is further subject to the following motion constraints:

$$\dot{x} = v \times \cos(\theta), \tag{16}$$

$$\dot{y} = v \times \sin(\theta), \tag{17}$$

$$\dot{\theta} = \omega, \tag{18}$$

where v, ω are the linear and angular velocities respectively. To meet these constraints, we use the POSQ `Steer` function [22]. Since this approach gives a local closed-loop policy between two vertices of the tree, it is more robust to noise and uncertainty in the motion than an open loop trajectory. Furthermore, it has been shown to produce smooth paths between feasible configurations [22]. Evaluation is done in the same way as in the previous section, except that we compare RLT* only to RLT*-NC and not MMP, as the latter cannot handle motion constraints. RLT*-NC was given 100 seconds to plan, in which case about 3000 configurations were sampled, and RLT*'s cache was set to this size.

Figure I shows that RLT* is an order of magnitude faster than RLT*-NC, while achieving a lower cost difference. The kinematic constraints contribute to this speedup since they make the `Steer` and `Safe` procedures, which are cached, more expensive. As in the holonomic case, RLT* resulted in smoother learning, confirming the results of Figure 3.

Figure 4 shows an instance of the types of paths generated by our method when compared to ground truth. This specific example was drawn from the validation set and is thus not a case of overfitting.

B. Real Robot Experiments

In this section, we apply RLT* to real, human demonstrations using a telepresence robot, shown in Figure 5, in a social navigation scenario. Furthermore, we deploy the learned cost function on the actual robot.

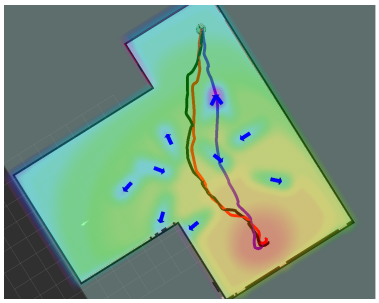


Fig. 4: Qualitative evaluation of simulated results. The colour map denotes the learned cost function. Red denotes *(*low) cost and green *high* cost. The learned path (red) is quite similar to the demonstrated path (black) with a clear improvement over the path before learning (purple).

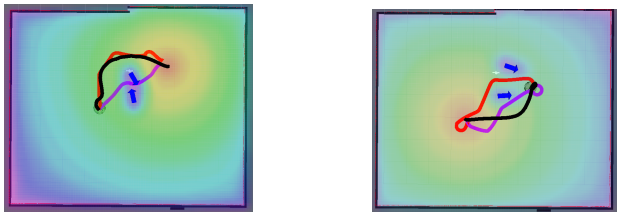
Fig. 5: The telepresence robot used in our experiments.

The experiments take place in a simplified version of the social scenarios we have seen in the previous section. There are two people in the scene at different positions and orientations. A human demonstrator is asked to execute paths for different initial and final conditions across the room. The task is similar to the one used in [13] and the purpose is to find cost functions that account for potential relationships between people depending on their orientation with respect to each other. For example, if people are facing each other, they are likely engaged in conversation or a similar activity (e.g., taking a photograph) that should not be interrupted. By contrast, if they are looking away from each other and there is enough distance between them then, it might be better to pass between them if doing so yields a shorter path.

To collect data for learning and validation, we use an off-the-shelf telepresence system augmented with several sensors that allow localisation and perception [20]. We use an Optitrack motion capture system to accurately collect ground truth data of both people and robot positions. RLT* learns a cost function from this data using the social feature set S1 and the rest of the features described earlier.

Since quantitative evaluation is difficult using real data, as no ground truth is available, we perform a qualitative evaluation instead. Figure 6 shows some representative cases that arose during learning. Figure 6a shows a case where RLT* produced paths (red) that are quite similar to the demonstrated ones (black). Figure 6b shows an instance

instances where the learned paths are reasonable even though they are not similar to the demonstrated paths.



(a) (b)

Fig. 6: Real demonstrated paths (black), learned paths (red), and shortest paths (purple). In (a), the learned path is similar to the demonstrated path; in (b) it is different but reasonable. The paths are laid over the learned cost function.

Finally, we successfully deployed the learned cost function, also shown in Figure 6, on a real telepresence robot. A video demonstrating this deployment can be found in the supplementary material. Even though our planner outputs a full policy in terms of angular and linear velocities, to follow the prescribed path we used an elastic bands local planner in order to deal with dynamic changes in the environment.

VI. Conclusion & 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, it uses a caching scheme that greatly reduces the computational cost and improves performance. Our results in simulated social navigation scenarios show that RLT* achieves better performance at lower computational cost, even when there is a discrepancy between the features used for demonstration and learning. Furthermore, our results show that RLT* can handle more complex configuration spaces with motion constraints. Finally, we used RLT* to learn a cost function using data from real demonstrations on a telepresence robot and successfully deployed that cost function back on the robot.

In future work, we hope to extend RLT* to model how features evolve over time, in order to, e.g., consider people's movement during planning. For complex social path planning applications, periodic replanning has been shown to help [7], [9]. Replanning in RRT* is also possible [23] and could potentially be incorporated into RLT$*$. Another interesting avenue for future work is to attempt learning using other sample based planning methods such as Batch Informed Trees (BIT*) [24] which are faster and thus perhaps more appropriate for tasks such as manipulation.

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

[5] 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.

[6] 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.

[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 *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014, pp. 1341–1346.

[10] 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.

[11] 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.

[12] 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.

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

[14] A. Boularias, J. Kober, and J. Peters, "Relative entropy inverse reinforcement learning." in *AISTATS*, 2011, pp. 182–189.

[15] M. Kalakrishnan, P. Pastor, L. Righetti, and S. Schaal, "Learning objective functions for manipulation," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1331–1336.

[16] C. Finn, S. Levine, and P. Abbeel, "Guided cost learning: Deep inverse optimal control via policy optimization," *arXiv preprint arXiv:1603.00448*, 2016.

[17] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964.

[18] 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.

[19] 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.

[20] 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.

[21] K. Shiarlis, J. Messias, and S. Whiteson, "Inverse reinforcement learning from failure," in *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2016, pp. 1060–1068.

[22] L. Palmieri and K. O. Arras, "A novel rrt extend function for efficient and smooth mobile robot motion planning," in *IROS 2014*. IEEE, 2014, pp. 205–211.

[23] M. Otte and E. Frazzoli, "Rrtx: Asymptotically optimal single-query sampling-based motion planning with quick replanning," *The International Journal of Robotics Research*, p. 0278364915594679, 2015.

[24] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Batch informed trees (bit*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs," in *ICRA 2015*. IEEE, 2015, pp. 3067–3074.