Planning with ants: Efficient path planning with rapidly exploring random trees and ant colony optimization

Article *in* International Journal of Advanced Robotic Systems · October 2016

DOI: 10.1177/1729881416664078

Research Article

INTERNATIONAL JOURNAL OF ADVANCED ROBOTIC SYSTEMS

International Journal of Advanced Robotic Systems
September-October 2016: 1–16
© The Author(s) 2016
DOI: 10.1177/1729881416664078
arx.sagepub.com

Planning with ants: Efficient path planning with rapidly exploring random trees and ant colony optimization

Alberto Viseras[1], Rafael Ortiz Losada[1], and Luis Merino[2]

Abstract

Rapidly exploring random trees (RRTs) have been proven to be efficient for planning in environments populated with obstacles. These methods perform a uniform sampling of the state space, which is needed to guarantee the algorithm's completeness but does not necessarily lead to the most efficient solution. In previous works it has been shown that the use of heuristics to modify the sampling strategy could incur an improvement in the algorithm performance. However, these heuristics only apply to solve the shortest path-planning problem. Here we propose a framework that allows us to incorporate arbitrary heuristics to modify the sampling strategy according to the user requirements. This framework is based on 'learning from experience'. Specifically, we introduce a utility function that takes the contribution of the samples to the tree construction into account; sampling at locations of increased utility then becomes more frequent. The idea is realized by introducing an ant colony optimization concept in the RRT/RRT* algorithm and defining a novel utility function that permits trading off exploitation versus exploration of the state space. We also extend the algorithm to allow an anytime implementation. The scheme is validated with three scenarios: one populated with multiple rectangular obstacles, one consisting of a single narrow passage and a maze-like environment. We evaluate its performance in terms of the cost and time to find the first path, and in terms of the evolution of the path quality with the number of iterations. It is shown that the proposed algorithm greatly outperforms state-of-the-art RRT and RRT* algorithms.

Keywords

Mobile robots, autonomous agents, motion and path planning, rapidly exploring random trees, ant colony optimization, bio-inspired robotics

Date received: 13 May 2016; accepted: 22 July 2016

Topic: Special Issue - Manipulators and Mobile Robots
Topic Editor: Michal Kelemen

Introduction

The optimal path-planning problem, which can be formulated as the task of driving a robot from an initial state $\mathbf{x}_A$ to a goal state $\mathbf{x}_B$ with the minimum cost, is one of the most fundamental problems in robotics. Despite a vast literature, it is still a challenging problem in many situations. Furthermore, in safety-of-life applications, such as search-and-rescue missions, or disaster relief, we aim to find the best possible path in a given time. Sampling-based methods, such as rapidly exploring random trees (RRTs),[1] are widely used to solve this problem, since the method offers low computational complexity and is efficient in finding a solution. However, the performance of the method can be increased by modifying the way in which the state space is sampled. Rapidly exploring random trees typically perform a uniform sampling of the

[1] Institute of Communications and Navigation of the German Aerospace Center (DLR), Oberpfaffenhofen, Germany
[2] Universidad Pablo de Olavide, Seville, Spain

Corresponding author:
Alberto Viseras, German Aerospace Center (DLR), Münchener Straße 20, 82334 Oberpfaffenhofen, Germany.
Email: alberto.viserasruiz@dlr.de

state space. The uniform sampling treats all samples equally; instead, new samples can be picked at locations where some utility function is optimized. Thus, we aim to sample at locations with a high utility; this utility function is constructed so as to either optimize a time to find a feasible solution, or to improve an already found solution.

Thus, our goal in this work is to answer the question of how to sample to optimize the selected utility function. Furthermore, the definition of the utility function allows us to formulate a framework that is able to incorporate heuristics that will guide the sampling strategy according to the user requirements. This would enable us to extend the original RRT* algorithm[2] to informative path-planning and exploration applications in unknown environments, where our goal is to sample more often where there is more information, or to employ the user's prior knowledge to perform a more intelligent sampling that optimizes some predefined criteria, for instance, avoiding harsh terrains in search and rescue missions. This could easily be introduced into our utility function. However, in this work, we focus on the shortest path-planning problem and formulate a utility function that optimizes the path cost with respect to distance. We suggest further extensions of the algorithm in the concluding discussion of possibilities for future work.

Inspired by machine-learning techniques, we augment our sampling strategy by taking the already accumulated samples into account. This can be interpreted as continuous learning of the probability density function, which represents the optimal sampling distribution at each moment and sampling according to it. To determine the optimal sampling distribution, we rely on ant colony optimization (ACO) for continuous domains ($ACO_{\mathbb{R}}$). We chose $ACO_{\mathbb{R}}$ because it offers superior performance compared with both Monte-Carlo methods and other swarm optimization techniques.[3] The $ACO_{\mathbb{R}}$ algorithm distributes virtual ants according to a utility function that evaluates the ants' relevance and so determines the sampling distribution (as illustrated in Figure 1). The utility function is constructed to trade exploitation of the state space, that is, optimization of the constructed tree, and exploration of the state space, which favours a growth of the tree in as yet explored regions of the state space. Given the tree we have constructed so far, we analyze: (i) how much the sample exploits the current solution; (ii) how much a sample contributes to exploration of the state space. Based on that, we update our ants, which will modify the sampling distribution and, in consequence, the way we construct our tree to solve the path-planning problem.

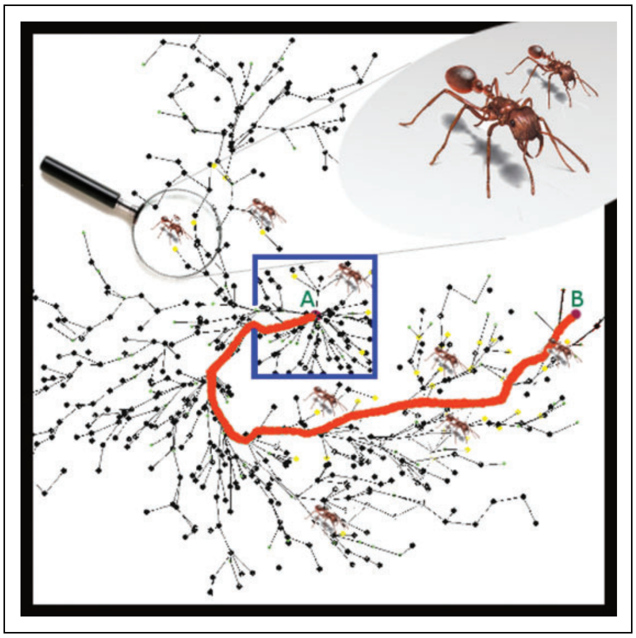

Figure 1. Example of one path generated with the proposed ACO-RRT* algorithm. We build a rapidly exploring random tree based on a modified sampling strategy that learns from previous experience.

Related work

Sampling-based path-planning algorithms are widely used because of their efficiency in providing path-planning solutions in high-dimensional spaces. These methods are well exemplified by probabilistic roadmaps (PRMs)[4] and rapidly exploring random trees (RRTs);[1] a modification of the RRT algorithm, called RRT*, is also known to achieve asymptotic optimality with respect to a given cost function.[2]

In recent years, a great amount of sampling-based path-planning algorithms have been proposed.[5–11] These works have in common that they outperform the RRT* algorithm by modifying and optimizing some of the subroutines that compose the original RRT* algorithm. However, the cited algorithms are specifically designed to solve the optimal shortest path-planning problem under certain restrictions. Here, we aim to go one step further and propose a framework that allows us to introduce some heuristics into the original RRT and RRT* algorithms. These heuristics are incorporated into the algorithm by modifying the sampling distribution, which is learnt online according to those heuristics as we sample the state space. The advantage of defining such heuristics is twofold: (i) it enables the introduction of some additional knowledge to solve the path-planning problem more efficiently; (ii) it could be used in conjunction with any of the aforementioned works to improve their performance. Specifically, in this work we will show that our approach, combined with shortest path heuristics, outperforms the state-of-the-art RRT and RRT* algorithms.

Sampling-based path planners consist of several subroutines that can be optimized individually to improve the algorithm's performance, which also make the methods very attractive. Denny et al. proposed a 'lazy planning' to improve the collision checking by the assumption that only 10% of the collisions checks are positive.[12] Conversely, algorithms like RRT connect[13] and the one proposed by Urmson and Simmons[14] increases the algorithm performance by heuristically biasing the tree growth. This tree growth has also been adapted by Denny et al.,[15] who adapted the branch size according to the space in heterogeneous environments. In contrast to previous works, here we focus on a modification of the sampling strategy. Essentially, if we can identify regions of higher importance, that is, regions in the state space that could help us to improve our current path, then we should sample these regions more often.

It is possible to dichotomize sampling strategies for path planning into importance sampling and adaptive sampling. Importance sampling methods exploit some predefined *a-priori* sampling strategy. Examples include goal-biased sampling,[16] medial-axis sampling,[17] where samples are taken from the medial axis of free space, and the bridge test,[18] which is designed to solve narrow passage problems. These methods are specifically designed to solve concrete problems. Alternatively, in adaptive sampling methods, the samples are drawn from a distribution that is adapted based on the information obtained from previous samples, which makes them more flexible. Siméon et al. propose the visibility PRM algorithm,[19] which just takes samples from the unexplored area within the planner visibility region. Although the constructed roadmaps are significantly smaller, the computation of the visibility region is expensive. Adaptive dynamic domain RRT adapts the previous concept to the RRT algorithm.[20] In this work, we additionally consider the importance of the previous samples, which are not necessarily within the visibility region. This is exploited for PRMs through an utility-guided sampling by Burns and Brock.[21] There, the authors do not aim to learn the sampling distribution, but to perform a Monte-Carlo sampling and select the samples with a higher utility. However, our focus lies in rapidly exploring random trees, owing to their efficiency, since they do not require any pre-computation time, as in PRMs. Adaptive sampling within the RRTs framework has also been exploited in recent works.[22–25] In contrast to our algorithm, these are able neither to incorporate nor to learn arbitrary heuristics.

The work of Rickert et al.[26] inspires the definition of our utility function. In their work, Rickert et al. propose the exploring-exploiting tree algorithm, which balances exploitation and exploration to construct the tree more effectively. Yet this method requires some environment-dependent pre-computing time to grow the tree, which does not make it suitable for online planning. The exploration-exploitation trade-off has also been employed in several works.[27–29] Alterovitz et al.[28] propose the rapidly exploring roadmaps algorithm. This algorithm first finds a solution, as in RRTs, and then refines this solution. Balancing exploration and exploitation is also employed by Persson and Sharf,[29] who generalize the A* algorithm to allow sampling-based motion planning. Also Akgun and Stilman[27] have developed an algorithm that trades off exploration and exploitation to improve the RRT* in high dimensions. This is done by introducing sampling heuristics. Our algorithm is also based on sampling heuristics, which are learnt using machine learning. In contrast to the aforementioned studies,[27–29] our framework allows us to introduce sampling heuristics that are not just specifically designed for the optimal shortest path planning but also for different applications.

Our goal in this paper is not just to define a framework that can incorporate arbitrary heuristics. In addition, we aim to learn the sampling distribution of the planning algorithm that better fits the user-defined heuristics. This is done by introducing machine-learning techniques. Morales et al.[30] divide the planning problem into several subproblems, and then employ machine learning to select a roadmap from a set that is more adequate to solve each of the subproblems. In the last step, the selected roadmaps are fused to obtain a global one. Machine learning is also introduced by Diankov and Kuffner[31] into A* to select the best heuristic from a set in order to improve the algorithm performance. Both of these algorithms[30,31] require a discrete predefined set from which they select the best roadmap or heuristic, respectively. In contrast, we aim to apply machine learning to learn not a discrete set but a continuous distribution.

This paper is strongly influenced by the idea of cross-entropy motion planning outlined by Kobilarov.[32] Here, Kobilarov[32] learns the sampling distribution from previous samples by evaluating its entropy. Its limitation comes from the high computational requirement to calculate the sampling distribution for the environment, which does not make it feasible for real-time applications. We improve this concept by using an ant colony optimization algorithm to learn the sampling distribution.[3] Ant colony optimization has also been used, by Mohamad et al.,[33] in the context of PRMs. The goal of Mohamad et al.[33] was to reduce the number of intermediate configurations from an initial to a goal position. Although it has a different objective, that work serves as an inspiration to incorporate the ACO into a sampling-based path planner. Learning the sampling distribution, together with the definition of a novel utility function, lets us derive a scalable algorithm suitable for real-time path-planning applications.

In the remainder of this paper, we briefly describe the rapidly exploring random trees and ant colony optimization algorithms that serve as the basis of our work. We then introduce the proposed algorithm and extend it to allow an anytime implementation. We evaluate and discuss the algorithm performance and finally draw conclusions and discuss avenues for future work.

Background

Rapidly exploring random trees

The rapidly exploring random trees (RRTs) algorithm is a solution to the path planning problem in complex high-dimensional spaces.[1] The RRT algorithm iteratively constructs a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ (tree) with a set of vertices $\mathcal{V}$ and edges $\mathcal{E}$, with the goal of establishing a path between $\mathbf{x}_A$ and $\mathbf{x}_B$ in the state space – a feasible trajectory $\mathcal{T}_{A,B}(\mathcal{G})$. The key steps of the RRT algorithm are summarized in Algorithm 1.

The algorithm is realized as follows. We draw a sample $\mathbf{x}_{\text{rand}}$ randomly from a uniform distribution defined over

Algorithm 1 RRT-Path Planner$(\mathbf{x}_A, \mathbf{x}_B, n, \mathcal{G}(\mathcal{V}, \mathcal{E}))$

```
1: $\mathcal{V} \leftarrow \{\mathbf{x}_A\}$; $\mathcal{E} \leftarrow \emptyset$; $\mathcal{T}_{A,B} \leftarrow \emptyset$; $c_{\text{path}} \leftarrow \infty$;
2: for $i = 1, \ldots, n$ do
3:     $\mathbf{x}_{\text{rand}} \leftarrow$ SampleFree;
4:     $\mathbf{x}_{nearest} \leftarrow$ Nearest$(\mathbf{x}_{\text{rand}}, \mathcal{V})$;
5:     $\mathbf{x}_{\text{new}} \leftarrow$ Steer$(\mathbf{x}_{nearest}, \mathbf{x}_{\text{rand}})$;
6:     if CollisionFree$(\mathcal{T}(\mathbf{x}_{nearest}, \mathbf{x}_{\text{new}}))$ then
7:         $\mathcal{V} \leftarrow \mathcal{V} \cup \{\mathbf{x}_{\text{new}}\}$;
8:         $\mathcal{E} \leftarrow \mathcal{E} \cup \{\mathcal{T}(\mathbf{x}_{nearest}, \mathbf{x}_{\text{new}})\}$;
9:     $\mathcal{T}_{A,B}(\mathcal{G}) \leftarrow$ FindBestPath$(\mathbf{x}_A, \mathbf{x}_B, \mathcal{G})$;
10: return $\mathcal{T}_{A,B}(\mathcal{G})$;
```

free space using the function `SampleFree`. Then the `Nearest` function finds the nearest neighbour (in terms of the cost-to-reach) of $\mathbf{x}_{\text{rand}}$ from the set of vertices $\mathcal{V}$. We use the function `Steer` to simulate driving the robot from $\mathbf{x}_{\text{nearest}}$ to $\mathbf{x}_{\text{rand}}$ according to our controller. We drive the robot a maximum distance η. This is a user-selected parameter, which sets the maximum branch size. As the output we obtain the state $\mathbf{x}_{\text{new}}$. If the trajectory $\mathcal{T}(\mathbf{x}_{\text{nearest}}, \mathbf{x}_{\text{new}})$ does not collide with any obstacles, we add the vertex $\mathbf{x}_{\text{new}}$ and the edge $\mathcal{T}(\mathbf{x}_{\text{nearest}}, \mathbf{x}_{\text{new}})$ to the tree $\mathcal{G}$. Given the current tree, we search the best path $\mathcal{T}_{A,B}(\mathcal{G})$ from $\mathbf{x}_A$ to $\mathbf{x}_B$ using the function `FindBestPath`. If there is no feasible path, the output would be a void set. We repeat this process during n iterations.

The RRT* algorithm is an evolution of the RRT algorithm, which has been shown to be asymptotically optimal.[2] We describe the RRT* in Algorithm 2. It differs from RRT in two aspects: choosing a parent and rewiring. In contrast to RRT, we choose the parent of $\mathbf{x}_{\text{new}}$ as the node from the set $\mathcal{X}_{\text{near}}$ that allows us to reach $\mathbf{x}_{\text{new}}$ with the minimum cost. $\mathcal{X}_{\text{near}}$ is calculated using the function `Near`, which is defined as

$$\texttt{Near}(\mathbf{x}, \mathcal{V}) := \{\| \mathbf{x} - \mathbf{x}' \| \leq r(\text{card}(\mathcal{V}))\} \tag{1}$$

with

$$r(\text{card}(\mathcal{V})) = \min\{\gamma(\log(\text{card}(\mathcal{V}))/\text{card}(\mathcal{V}))^{1/d}, \eta\} \tag{2}$$

where $\text{card}(\mathcal{V})$ is the number of elements in set $\mathcal{V}$, γ is a constant and d is the number of dimensions of the state space. The RRT* algorithm also incorporates a rewiring process to find an optimal trajectory. This is done by finding minimum cost sub-paths. Here, we define two different costs: $\texttt{Cost}(\mathbf{x}, \mathcal{G})$ is the cost-to-reach sample $\mathbf{x}$ from $\mathbf{x}_A$ following the tree $\mathcal{G}$. $\texttt{Cost}(\mathcal{T}(\mathbf{x}, \mathbf{x}'))$ would be the cost of going directly from $\mathbf{x}$ to $\mathbf{x}'$, regardless of the obstacles. In this work, we define the cost between two samples $\texttt{Cost}(\mathcal{T}(\mathbf{x}, \mathbf{x}'))$ as the Euclidean distance between them. The cost $\texttt{Cost}(\mathbf{x}, \mathcal{G})$ is the sum of these Euclidean distances along the edges towards $\mathbf{x}$.

Ant colony optimization for continuous domains

Ant colony optimization is a nature-inspired algorithm to solve hard combinatorial optimization problems.[34] Its driving principle comes from the behaviour of ants when searching for food. First, they leave the nest walking in random directions. Once they find a food source, they come back to the nest, leaving a pheromone trail on the ground. The pheromone deposited depends on the quality and quantity of the food and guides the other ants to the food source. Based on the same principle, ant colony optimization for continuous domains (ACO$_{\mathbb{R}}$) is proposed to solve continuous optimization problems.[3] This work inspires our sampling strategy, in which the ants, according to their utility, will decide where to sample next.

The ants are stored in a table $\mathbf{T}$, as depicted in Figure 2. Each row contains one of the k ants, where $\mathbf{s}_l = [s_l^{[1]}, s_l^{[2]}, \ldots, s_l^{[d]}]$ is the vector of coordinates describing the lth ant's location and d is the number of dimensions of the state space. The ant's utility is given by u_l, which determines the importance of the lth ant. The utility is defined according to the algorithm's optimization objective.

The algorithm works as follows. First, we take a sample $\mathbf{x}_{\text{rand}} = [x_{\text{rand}}^{[1]}, \ldots, x_{\text{rand}}^{[j]}, \ldots, x_{\text{rand}}^{[d]}]$, where each of the components $x_{\text{rand}}^{[j]}$ is drawn from a Gaussian kernel probability density function

$$G^{[j]}(x) = \sum_{l=1}^{k} w_l g_l^{[j]}(x) = \sum_{l=1}^{k} w_l \frac{1}{\sigma_l^{[j]}\sqrt{2\pi}} e^{-\frac{(x-s_l^{[j]})^2}{2\sigma_l^{[j]2}}} \tag{3}$$

with $j = 1, 2, \ldots, d$, and $\sigma_l^{[j]}$ the lth ant's standard deviation in dimension j. The standard deviation is calculated as the average distance from the lth ant to the rest of the ants stored in $\mathbf{T}$

$$\sigma_l^{[j]} = \xi \sum_{e=1}^{k} \frac{|s_e^{[j]} - s_l^{[j]}|}{k-1} \tag{4}$$

```
Algorithm 2 RRT*-Path Planner($\mathbf{x}_A, \mathbf{x}_B, n, \mathcal{G}(\mathcal{V}, \mathcal{E})$)
1: $\mathcal{V} \leftarrow \{\mathbf{x}_A\}$; $\mathcal{E} \leftarrow \emptyset$; $\mathcal{T}_{A,B} \leftarrow \emptyset$; $c_{\text{path}} \leftarrow \infty$;
2: for $i = 1, \dots, n$ do
3:     $\mathbf{x}_{\text{rand}} \leftarrow$ SampleFree;
4:     $\mathbf{x}_{nearest} \leftarrow$ Nearest($\mathbf{x}_{\text{rand}}, \mathcal{V}$);
5:     $\mathbf{x}_{\text{new}} \leftarrow$ Steer($\mathbf{x}_{nearest}, \mathbf{x}_{\text{rand}}$);
6:     if CollisionFree($\mathcal{T}(\mathbf{x}_{nearest}, \mathbf{x}_{\text{new}})$) then
7:         $\mathbf{x}_{min} \leftarrow \mathbf{x}_{nearest}$;
8:         $minCost \leftarrow$ Cost($\mathbf{x}_{nearest}, \mathcal{G}$) + Cost($\mathcal{T}(\mathbf{x}_{nearest}, \mathbf{x}_{\text{new}})$);
9:         $\mathcal{X}_{near} \leftarrow$ Near($\mathbf{x}_{\text{new}}, \mathcal{V}$);
10:        for $\mathbf{x}_{near} \in X_{near}$ do                                ▷ Choose parent
11:            $c_{\text{new}} \leftarrow$ Cost($\mathbf{x}_{near}, \mathcal{G}$) + Cost($\mathcal{T}(\mathbf{x}_{near}, \mathbf{x}_{\text{new}})$);
12:            if $c_{\text{new}} < minCost$ then
13:                if CollisionFree($\mathcal{T}(\mathbf{x}_{near}, \mathbf{x}_{\text{new}})$) then
14:                    $\mathbf{x}_{min} \leftarrow \mathbf{x}_{near}$; $minCost \leftarrow c_{\text{new}}$;
15:        $\mathcal{V} \leftarrow \mathcal{V} \cup \{\mathbf{x}_{\text{new}}\}$;
16:        $\mathcal{E} \leftarrow \mathcal{E} \cup \{\mathcal{T}(\mathbf{x}_{min}, \mathbf{x}_{\text{new}})\}$;
17:        for $\mathbf{x}_{near} \in \mathcal{X}_{near}$ do                         ▷ Rewire near nodes
18:            $c_{near} \leftarrow$ Cost($\mathbf{x}_{\text{new}}, \mathcal{G}$) + Cost($\mathcal{T}(\mathbf{x}_{\text{new}}, \mathbf{x}_{near})$);
19:            if $c_{near} <$ Cost($\mathbf{x}_{near}, \mathcal{G}$) then
20:                if CollisionFree($\mathcal{T}(\mathbf{x}_{\text{new}}, \mathbf{x}_{near})$) then
21:                    $\mathbf{x}_{parent} \leftarrow$ Parent($\mathbf{x}_{near}$);
22:                    $\mathcal{E} \leftarrow \mathcal{E} \setminus \{\mathcal{T}(\mathbf{x}_{parent}, \mathbf{x}_{near})\}$;
23:                    $\mathcal{E} \leftarrow \mathcal{E} \cup \{\mathcal{T}(\mathbf{x}_{\text{new}}, \mathbf{x}_{near})\}$;
24:    $\mathcal{T}_{A,B}(\mathcal{G}) \leftarrow$ FindBestPath($\mathbf{x}_A, \mathbf{x}_B, \mathcal{G}$);
25: return $\mathcal{T}_{A,B}(\mathcal{G})$;
```

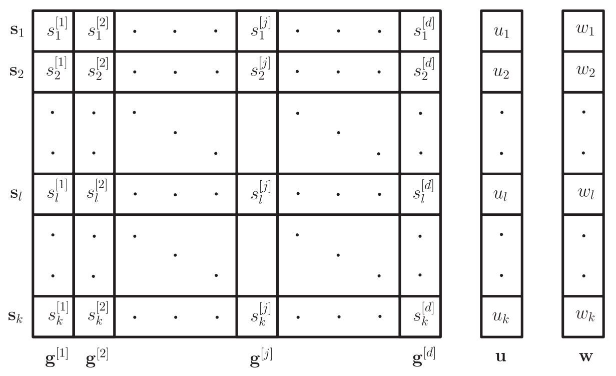
$\mathbf{s}_1$
$s_1^{[1]}$ $s_1^{[2]}$ $s_1^{[j]}$ $s_1^{[d]}$ u_1 w_1
$\mathbf{s}_2$
$s_2^{[1]}$ $s_2^{[2]}$ $s_2^{[j]}$ $s_2^{[d]}$ u_2 w_2
$\mathbf{s}_l$
$s_l^{[1]}$ $s_l^{[2]}$ $s_l^{[j]}$ $s_l^{[d]}$ u_l w_l
$\mathbf{s}_k$
$s_k^{[1]}$ $s_k^{[2]}$ $s_k^{[j]}$ $s_k^{[d]}$ u_k w_k
$\mathbf{g}^{[1]}$ $\mathbf{g}^{[2]}$ $\mathbf{g}^{[j]}$ $\mathbf{g}^{[d]}$ $\mathbf{u}$ $\mathbf{w}$

Figure 2. **T**-table of the ACO$_{\mathbb{R}}$ algorithm. It stores the k ants, sorted according to their utility given by the elements of **u**. Together with the ants' coordinates, the elements of **w** determine the probability density function described by the ants.

where $\xi > 0$ is the pheromone evaporation rate, which avoids the algorithm converging too quickly before approaching the optimal solution. The parameter w_l from **w** is set as

$$w_l = \frac{1}{qk\sqrt{2\pi}} \, \mathrm{e}^{-\frac{(l-1)^2}{2q^2k^2}} \tag{5}$$

where q is a user-defined parameter. When q is small, the best ranked solutions are strongly preferred. The vector of weights **w** is normalized so that the integral of the probability density function $G^{[j]}(x)$ over the entire space is equal to one. The value of w_l is initialized and is not modified during the execution of the algorithm.

Next, we sort the table **T** in descending order according to the utility given by vector **u** and insert the new sample $\mathbf{x}_{\text{rand}}$. The sample $\mathbf{x}_{\text{rand}}$ will now become an ant. In this way, samples with a higher utility will move up the table and will be selected with a higher probability. If the sample's utility is higher than the last ranked solution $\mathbf{s}_k$, this last one will be removed from the table **T** to keep k ants in the algorithm. This loop goes on during n iterations.

ACO-RRT* algorithm

In this work, we propose the ACO-RRT* algorithm, which aims to improve on the RRT and RRT* performance by

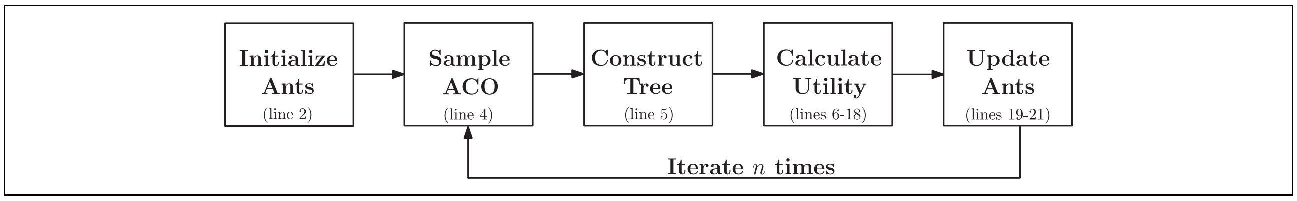

Figure 3. ACO-RRT* algorithm block diagram. Each of the five blocks points to its respective lines from Algorithm 4.

Algorithm 3 InitializeAnts$(k, \mathbf{T})$

```
1: for $l = 1, \ldots, k$ do
2:     $\mathbf{x}_{\text{rand}} \leftarrow$ SampleFree;
3:     for $j = 1, \ldots, d$ do                                   ▷ Iteration $j$-dimension
4:         $\mathbf{T}.s_l^{[j]} \leftarrow x_{\text{rand}}^{[j]}$;
5:     $\mathbf{T}.\mathcal{F}_l.\alpha \leftarrow NULL$; $\mathbf{T}.u_l \leftarrow NULL$; $\mathbf{T}.w_l \leftarrow w_l$;
6:     $\mathbf{T}.\mathcal{F}_l.U_{\text{exploit}} \leftarrow NULL$; $\mathbf{T}.\mathcal{F}_l.U_{\text{explore}} \leftarrow NULL$;
7: return $\mathbf{T}$;
```

modifying the sampling distribution using ant colony optimization for continuous domains. Our motivation lies in learning from the experience. This means that, after sampling, we evaluate how much that sample contributed to improve our current path. This evaluation will influence how we obtain the next sample.

The algorithm consists of five steps (see Figure 3). First, we initialize the ants that will generate future samples. Second, we sample from the distribution described by the current ants. Then we update our tree according to the original RRT/RRT* algorithm. After that, we calculate the utility of that sample based on how much it could improve the current path. This is divided into two factors: (i) exploitation of the current solution and (ii) exploration of the state space to find a new, better solution. Based on that utility, we update the ants and resample according to the new distribution. The algorithm is formulated in Algorithms 3, 4 and 5, and is described in detail in the following subsections.

Initialize ants

The first part of the algorithm consists of filling the table **T** with k ants (see Algorithm 3). We take a sample from a uniform distribution defined over the obstacle-free space (line 2). Then we insert its position coordinates in row l, where $x_{\text{rand}}^{[j]}$ represents the coordinate of $\mathbf{x}_{\text{rand}}$ at the jth dimension (lines 3 and 4). We initialize the utility u_l to zero. To calculate the utility, we require the exploitation and exploration utility as well as the α that trades off the two factors. These three elements ($\mathcal{F}_l.U_{\text{exploit}}, \mathcal{F}_l.U_{\text{explore}}, \mathcal{F}_l.\alpha$) are stored in set $\mathcal{F}_l$ and initialized to zero. (The notation $\mathcal{A}.b$ makes reference to the element b that is part of set $\mathcal{A}$.) The parameter w_l is computed according to equation (5) (lines 5 and 6). This initialization is only performed once, at the beginning of the algorithm.

Sample ACO

Given the table **T**, we sample from the probability density function described by the ants (line 4 in Algorithm 4). The following method is equivalent to sampling directly from the distribution described by equation (3). First we select an ant l with a probability

$$p_l = \frac{w_l}{\sum_{e=1}^{k} w_e} \tag{6}$$

with w_e given by equation (5) and $l = 1, 2, ..., k$. The new sample will be $\mathbf{x}_{\text{rand}} = [x_{\text{rand}}^{[1]}, ..., x_{\text{rand}}^{[j]}, ..., x_{\text{rand}}^{[d]}]$. The position coordinate $x_{\text{rand}}^{[j]}$ is taken from a Gaussian distribution with $x_{\text{rand}}^{[j]} \sim \mathcal{N}(s_l^{[j]}, \sigma_l^{[j]2})$. This function outputs the new sample $\mathbf{x}_{\text{rand}}$, as well as the ant index l that generated it. We use rejection sampling to select a sample that belongs to the free space.

This modification of the sampling strategy implies that the algorithm cannot guarantee the theoretical asymptotic optimality from RRT*. However, simulation results suggest that the proposed algorithm is able to approach the optimal solution. The explanation for such behaviour lies in the fact that the ants are associated with Gaussian probability density functions. Samples extracted from such a function can take values from an infinite domain that results in sampling over the complete state space. Even in the worst case, when all ants could converge to a single point, the variance of the distributions associated with the

```
Algorithm 4 ACO-RRT/RRT*($\mathbf{x}_A, \mathbf{x}_B, n, k, \mathcal{G}(\mathcal{V}, \mathcal{E})$)
1:  $\mathcal{V} \leftarrow \{\mathbf{x}_A\}$; $\mathcal{E} \leftarrow \emptyset$; $\mathbf{T} \leftarrow \emptyset$; $\mathcal{T}_{A,B} \leftarrow \emptyset$; $c_{\text{path}} \leftarrow \infty$;
2:  $\mathbf{T} \leftarrow$ InitializeAnts($\mathbf{T}, k$);
3:  for $i = 1, \ldots, n$ do
4:      $\mathbf{x}_{\text{rand}}, l \leftarrow$ SampleACO($\mathbf{T}$);
5:      $\mathbf{x}_{\text{new}}, \mathcal{G}(\mathcal{V}, \mathcal{E}) \leftarrow$ ConstructTree($\mathbf{x}_{\text{rand}}, \mathcal{G}$);
6:      $U_{\text{explore}} \leftarrow U_{\text{explore}}(\mathbf{x}_{\text{new}}, \mathcal{G})$;
7:      if $c_{\text{path}} \neq \infty$ then                                    ▷ Path found
8:          if Cost($\mathbf{x}_{\text{new}}, \mathcal{G}$) + Cost($\mathcal{T}(\mathbf{x}_{\text{new}}, \mathbf{x}_B)$) $< c_{\text{path}}$ then
9:              $\alpha \leftarrow \check{\alpha}$;
10:             if Cost($\mathbf{x}_B, \mathcal{G}$) $< c_{\text{path}}$ then              ▷ Path improvement
11:                 $U_{\text{exploit}} \leftarrow \dot{\check{U}}_{\text{exploit}}(\mathbf{x}_{\text{new}}, \mathcal{G})$;
12:             else
13:                 $U_{\text{exploit}} \leftarrow \bar{\check{U}}_{\text{exploit}}(\mathbf{x}_{\text{new}}, \mathcal{G})$;
14:         else
15:             $\alpha \leftarrow NULL, U_{\text{exploit}} \leftarrow NULL, U_{\text{explore}} \leftarrow NULL$;
16:     else
17:         $\alpha \leftarrow \hat{\alpha}$;
18:         $U_{\text{exploit}} \leftarrow \hat{U}_{\text{exploit}}(\mathbf{x}_{\text{new}}, \mathcal{G})$;
19:     $\mathcal{T}_{A,B}(\mathcal{G}) \leftarrow$ FindBestPath($\mathbf{x}_A, \mathbf{x}_B, \mathcal{G}$);
20:     $\mathcal{P} \leftarrow \{U_{\text{exploit}}, U_{\text{explore}}, \mathbf{x}_{\text{new}}, \mathbf{x}_B, \alpha, l, \mathcal{T}_{A,B}(\mathcal{G}), c_{\text{path}}\}$;
21:     $\mathbf{T} \leftarrow$ UpdateAnts($\mathbf{T}, \mathcal{G}, \mathcal{P}$);
22:     if $\mathcal{T}_{A,B}(\mathcal{G}) \neq \emptyset$ then
23:         $c_{\text{path}} \leftarrow$ Cost($\mathbf{x}_B, \mathcal{G}$);
24: return $\mathcal{T}_{A,B}(\mathcal{G})$;
```

ants will be always slightly greater than zero. This fact guarantees that the state space will always be fully sampled and, therefore, the algorithm will approach the optimal solution.

Construct tree

The next step is to construct the tree according to the basic rapidly exploring random tree path planner. This step corresponds to lines 4–8 in Algorithm 1 (RRT) and lines 4–23 in Algorithm 2 (RRT*). This function needs the $\mathbf{x}_{\text{rand}}$ sample and the current tree. The output of this function is the new vertex $\mathbf{x}_{\text{new}}$ added to the tree as well as the new constructed tree $\mathcal{G}$. Based on the new sample and the current tree, we calculate the utility function that will modify how the ants sample the states' space.

Calculate utility

The key part of the algorithm is the calculation of the utility of the $\mathbf{x}_{\text{new}}$ sample. It corresponds to lines 6–18 in Algorithm 4. We define the utility function as a trade-off between exploitation and exploration. Exploitation (i) tries to go directly to the goal position using the shortest possible path, if no path has been found, and, (ii) once a path has been found, tries to improve it. Exploration aims to sample at those locations have not yet been sampled. It helps us to: (i) find a first path by exploring the state space and (ii) search for new better paths once we have found a solution. The utility function $U(\mathbf{x}, \mathcal{G})$ of sample $\mathbf{x}$ given the tree $\mathcal{G}$ is defined as

$$U(\mathbf{x}, \mathcal{G}) = \alpha \cdot U_{\text{exploit}}(\mathbf{x}, \mathcal{G}) + (1 - \alpha) \cdot U_{\text{explore}}(\mathbf{x}, \mathcal{G}) \quad (7)$$

where α models the trade-off between exploitation and exploration, $U_{\text{exploit}}(\mathbf{x}, \mathcal{G})$ is the exploitation utility, and $U_{\text{explore}}(\mathbf{x}, \mathcal{G})$ is the exploration utility.

Exploration utility. The exploration utility $U_{\text{explore}}(\mathbf{x}, \mathcal{G})$ represents the density of samples in the tree $\mathcal{G}$ in the vicinity of sample $\mathbf{x}$ and is defined as

$$U_{\text{explore}}(\mathbf{x}, \mathcal{G}) = \frac{1}{\text{card}(\mathcal{X}_{\text{near}})} \left(\frac{R}{\eta}\right)^d \quad (8)$$

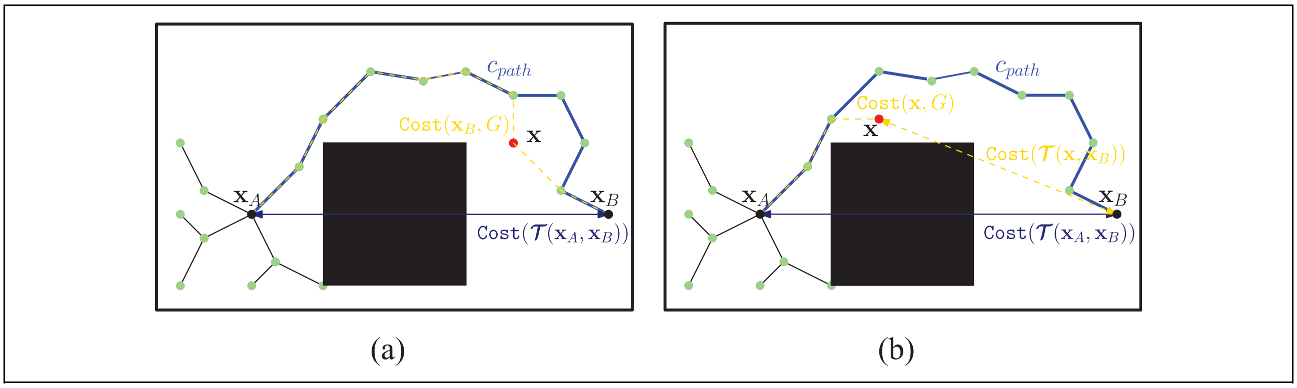

Figure 4. Graphical representation of the exploitation utility in the path found mode: (a) path improvement; (b) no path improvement. The black square represents an obstacle. The red dot corresponds to the sample **x**. The black lines and green dots represent the current tree $\mathcal{G}$. The superposed thick blue line is the best found path before sampling the state **x**. The dashed yellow line is the new best path after sampling **x**. Arrows represent the direct path between one state and the goal $\mathbf{x}_B$.

with $\mathcal{X}_{\text{near}} \leftarrow \texttt{Near}(\mathbf{x}, \mathcal{V})$ the set of neighbours of **x** given by equation (1), card$(\mathcal{X}_{\text{near}})$ the number of elements in the set $\mathcal{X}_{\text{near}}$, η a parameter of the RRT/RRT* path planner and R the connection radius. We define $R = r(\text{card}(\mathcal{V}))$ for RRT* with $r(\cdot)$ given by equation (2), and $R = \eta$ for RRT.

The first term of the product models a decay of the exploration utility as proportional with respect to the number of elements in $\mathcal{X}_{\text{near}}$. That implies that a sample that has a low number of neighbours in the current tree $\mathcal{G}$ will have a high exploration utility. Therefore, the exploration utility function will bias the exploration towards the not-yet-sampled state space. However, the number of neighbours of a sample depends on the connection radius R given by the RRT/RRT* algorithm. The bigger the connection radius, the higher the probability of having a larger number of neighbours. As the tree growths, the connection radius decreases. To make the exploration utility independent of the tree's current state, we introduce a second term $(R/\eta)^d$ to act as a normalization factor.

Exploitation utility. The exploitation utility takes advantage of the acquired knowledge about the state space. Here, we distinguish two modes: no path found and path found, so that we can incorporate the information about the current solution. To add more flexibility to the algorithm, we assign to it the parameter α, which trades off exploitation and exploration in equation (7), one of the two possible values: (a) $\alpha = \hat{\alpha}$ if no path was found; (b) $\alpha = \check{\alpha}$ otherwise.

No path found. Before finding a first path, we bias the sampling to connect the state **x** with the goal as quickly as possible regardless the obstacles.[16] Conversely, the exploration utility will bias the growth to obtain a path free of collisions. In this mode, we define the exploitation utility as

$$\hat{U}_{\text{exploit}}(\mathbf{x}, \mathcal{G}) = 1 - \frac{\texttt{Cost}(\mathcal{T}(\mathbf{x}, \mathbf{x}_{\text{B}}))}{c_{\max}} \tag{9}$$

where the cost to go directly to the goal from **x**, $\texttt{Cost}(\mathcal{T}(\mathbf{x}, \mathbf{x}_B))$, is normalized by the maximum cost $c_{\max}$ to reach the goal from any of the possible states. We can observe that sampling in the goal state will have the maximum utility since it will direct the tree growth towards the goal position.

Path found. Once we have found a path, we can exploit this information to derive a richer exploitation utility function. We consider two possible situations: path improvement (see Figure 4(a)), and no path improvement (see Figure 4(b)).

Consider that sample **x** leads to an improvement on the current path. Then we expect that this region of the state space could help us to improve the solution again in a future iteration. Therefore, we formulate the exploitation utility to quantify this improvement

$$\dot{\check{U}}_{\text{exploit}}(\mathbf{x}, \mathcal{G}) = \frac{c_{\text{path}} - \texttt{Cost}(\mathbf{x}_{\text{B}}, \mathcal{G})}{c_{\text{path}} - \texttt{Cost}(\mathcal{T}(\mathbf{x}_{\text{A}}, \mathbf{x}_{\text{B}}))} \tag{10}$$

with $\texttt{Cost}(\mathbf{x}_{\text{B}}, \mathcal{G})$ the cost of the best path after sampling **x**, and c_{path} the cost of the previous path. The denominator normalizes the function so that it ranges between 0 (no path improvement) and 1 (the path is the best possible one).

In contrast, if sample **x** has not contributed to improve the solution, we define the exploitation utility to shape the path as a straight line connecting the initial and goal positions. This represents the best possible path regardless of the obstacles. Again, the exploration utility will compensate this bias to find the best feasible path considering the obstacles. This utility is given by

$$\bar{\check{U}}_{\text{exploit}}(\mathbf{x}, \mathcal{G}) = \frac{\texttt{Cost}(\mathcal{T}(\mathbf{x}_{\text{A}}, \mathbf{x}_{\text{B}}))}{\texttt{Cost}(\mathbf{x}, \mathcal{G}) + \texttt{Cost}(\mathcal{T}(\mathbf{x}, \mathbf{x}_{\text{B}}))} \tag{11}$$

It is important to note that, once we have found a first path, we only introduce the ant in table **T** if it could improve the current solution (line 8). This is equivalent to setting the exploration and exploitation utilities to zero

Algorithm 5 `UpdateAnts`$(\mathbf{T}, \mathcal{G}(\mathcal{V}, \mathcal{E}), \mathcal{P})$

```
1:  $\{U_{\text{exploit}}, U_{\text{explore}}, \mathbf{x}_{\text{new}}, \mathbf{x}_B, \alpha, l, \mathcal{T}_{A,B}(\mathcal{G}), c_{\text{path}}\} \leftarrow \mathcal{P}$;
2:  if $(\mathcal{T}_{A,B}(\mathcal{G}) \neq \emptyset) \wedge (c_{\text{path}} = \infty)$ then                ▷ First path found
3:      $\mathbf{T}.\mathbf{u} \leftarrow NULL; \mathbf{T}.\mathcal{F} \leftarrow NULL$;
4:  else
5:      $\mathcal{F}_l.U_{\text{explore}} \leftarrow U_{\text{explore}}$;
6:      $\mathbf{s}_l \leftarrow \mathbf{T}.\mathbf{s}_l; \mathcal{F}_l.\alpha \leftarrow \mathbf{T}.\mathcal{F}_l.\alpha; \mathcal{F}_l.U_{\text{exploit}} \leftarrow \mathbf{T}.\mathcal{F}_l.U_{\text{exploit}}$;
7:      if $\mathcal{T}_{A,B}(\mathcal{G}) \neq \emptyset$ then
8:          if $\texttt{Cost}(\mathbf{s}_l, \mathcal{G}) + \texttt{Cost}(\mathcal{T}(\mathbf{s}_l, \mathbf{x}_B)) > \texttt{Cost}(\mathbf{x}_B, \mathcal{G})$ then
9:              $\mathcal{F}_l.U_{\text{exploit}} \leftarrow NULL; \mathcal{F}_l.U_{\text{explore}} \leftarrow NULL$;
10:     $u_l \leftarrow \texttt{CalculateUtility}(\mathcal{F}_l)$;
11:     $\mathbf{T} \leftarrow \texttt{Extract}(l, \mathbf{T})$;
12:     $\mathbf{T} \leftarrow \texttt{Insert}(\mathbf{s}_l, u_l, \mathcal{F}_l, \mathbf{T})$;
13:     $\mathcal{F}_i.\alpha \leftarrow \alpha; \mathcal{F}_i.U_{\text{exploit}} \leftarrow U_{\text{exploit}}; \mathcal{F}_i.U_{\text{explore}} \leftarrow U_{\text{explore}}$;
14:     $u_i \leftarrow \texttt{CalculateUtility}(\mathcal{F}_i)$;
15:     $\mathbf{T} \leftarrow \texttt{Insert}(\mathbf{x}_{\text{new}}, u_i, \mathcal{F}_i, \mathbf{T})$;
16:     $\mathbf{T} \leftarrow \texttt{Extract}(\text{"}last\text{"}, \mathbf{T})$;
17: return $\mathbf{T}$;
```

(line 15). By doing this, we allow the algorithm to sample in the future again in that region, which could incur in a path improvement.

Update ants

The last step is to update the ants in table **T**, according to Algorithm 5. One of the inputs is the minimum cost path $\mathcal{T}_{A,B}(\mathcal{G})$ from the initial to the goal position given by the tree $\mathcal{G}$. The function `FindBestPath` finds it, returning a void set if no path has been found yet (line 23 in Algorithm 4). The first time a path is found we reset the utility values $\mathbf{T}.\mathbf{u}$ and the parameters in $\mathbf{T}.\mathcal{F}$, since the ants that we will store from this point on will have more information based on the current found path (lines 2 and 3).

The sample $\mathbf{x}_{\text{new}}$ was generated from the lth ant. The utility of this ant should be updated to incorporate the current information provided by the tree (lines 5–12). One example could be an ant that had a great exploration utility when it was stored, but several iterations later the area associated with the ant is fully explored. In line 8, we introduce a soft pruning condition that allows the algorithm to shape the sampling distribution according to the most promising areas, given the current knowledge about the state space. We insert the lth ant in table **T** according to the updated utility u_l, calculated using equation (8) from the elements of vector $\mathcal{F}_l$. Then, the ant associated with $\mathbf{x}_{\text{new}}$ is inserted into the table in the position given by its utility u_i that is calculated from $\mathcal{F}_i$ (lines 13–15). Here, we have made a simplification that consists of two heuristics: (i) the utility of the new ant is the same as the utility of the l th ant; (ii) the introduction of a new sample in the table does not incur a modification of the exploration utility of the rest of the ants contained in the table. These two heuristics allow us to reduce the algorithm's computational complexity, since they avoid recalculating the utilities each time we introduce a new ant into the table. Despite this simplification, these heuristics have been shown to work well, since the next time an ant is selected its utility will be recalculated according to the updated information. The last step of this algorithm is to remove the last row of table **T** after a new ant has been added (line 14). This is done to keep k ants in the table.

This complete loop is repeated during n iterations. The output of the algorithm is the trajectory $\mathcal{T}_{A,B}(\mathcal{G})$.

Anytime ACO-RRT*

The main drawback of the ACO-RRT* algorithm is that it needs more time to find a first path than does the basic RRT/RRT*. This is because of the time needed by the ants to converge the first time. However, the solution obtained has a better quality; that is a smaller cost. There are situations, for example in search and rescue missions, where finding a first solution rapidly is crucial. Then, if we had more time, we could improve it to reach our goal faster. Inspired by Ferguson and Stentz,[35] we exploit this concept in our anytime ACO-RRT* algorithm. First, we run the fastest algorithm (RRT) to find a first solution $\mathcal{T}_{A,B}(\mathcal{G})$. Second, we initialize our tree $\mathcal{G}$ as the found path $\mathcal{G}(\mathcal{V}, \mathcal{E}) = \mathcal{T}_{A,B}(\mathcal{G})$. Then we improve the current solution using ACO-RRT*, taking that tree as input. This

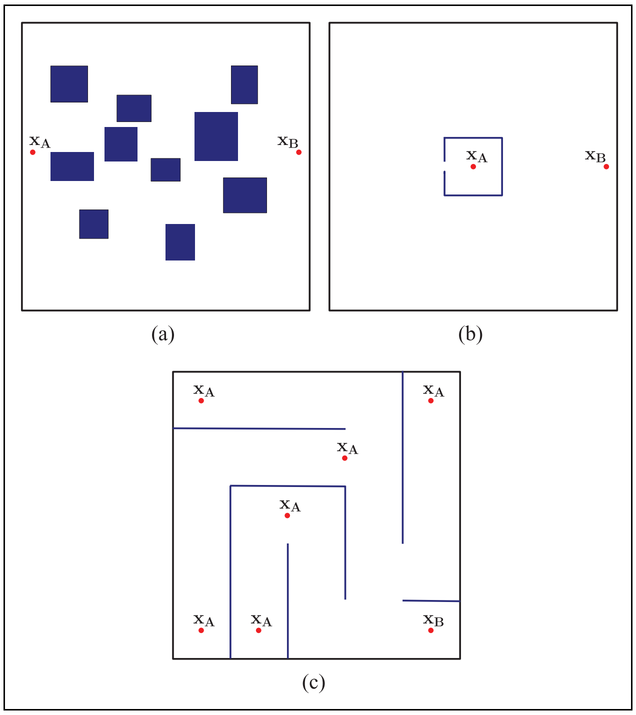

Figure 5. Tested scenarios: (a) Scenario 1; (b) Scenario 2; (c) Scenario 3. We aim to find the optimal path that goes from $\mathbf{x}_A$ to $\mathbf{x}_B$. All scenarios measure 100 m × 100 m.

mechanism allows us to combine the best of both algorithms to increase the algorithm's performance.

Simulations and discussion of results

We tested the ACO-RRT* algorithm performance with a holonomic robot in three simulated scenarios (see Figure 5). We chose a holonomic robot, since it enables us to abstract the algorithm capabilities from the robot's kinodynamic constraints. We assume that the robot corresponds to a single point. However, more complex robot shapes could easily be introduced within this framework. The three scenarios correspond to realistic scenarios that could be encountered while navigating an indoor facility. Moreover, similar scenarios have been considered to evaluate some of the most recent state-of-the art methods.[7,24] Analysis in more complex scenarios and the consideration of kinodynamic constraints is left for future research. All scenarios measure 100 m × 100 m and the goal is to find the optimal path that goes from $\mathbf{x}_A$ to $\mathbf{x}_B$. Since Scenario 3 is more structured, the placement of the initial position plays a crucial role. Therefore, in Scenario 3 we consider different possible starting positions $\mathbf{x}_A$, which are randomly selected in each simulation run. For the evaluation we consider a goal region centred around the goal position, not just a single state. Scenario 1 is composed of 10 rectangles of different sizes and the optimal path measures 88 m. Scenario 2 contains a narrow passage, which is often considered one of the most challenging path-planning problems.

Table 1. Simulation parameters. For each simulation, we ran the algorithm for 220 s.

n (s)	η (m)	k (ants)	q	ξ	$\breve{\alpha}$	$\hat{\alpha}$
220	5	100	50	0.4	0.3	0.1

The optimal path in this scenario is 63 m. Scenario 3 corresponds to a maze-like environment. This last scenario allows us to test the algorithm performance in a more structured scenario.

We carried out the simulations using a Intel Xenon E31225 processor at 3.10 GHz with 8 GB of RAM. We ran each simulation 100 times, according to the parameters shown in Table 1.

We evaluated the following parameters: (i) time to find the first path and its associated cost; (ii) evolution of the cost of the best found path over time; (iii) performance of the anytime implementation; (iv) influence of the different parameters in the algorithm performance.

Time to find first path and associated cost

One of the key figures to evaluate the performance of the path-planning algorithm is the number of iterations needed to find a first path. This is strongly correlated with the cost associated to that path. In Figure 6, we evaluate both indicators for Scenarios 1, 2 and 3. For Scenarios 2 and 3, we represent the time instead of the number of iterations, to demonstrate the algorithm's performance in an actual system. We compared the ACO-RRT* algorithm with the ACO-RRT, RRT* and RRT algorithms. We did not perform a comparison against the cited state-of-the-art works because they follow a different goal, that of approaching the optimal solution to the path planning problem as quickly as possible. By contrast, the objective of our paper is to show that our algorithm is able to learn some user predefined heuristics and then use them to improve the solution of the original RRT/RRT* algorithms.

Figure 6 shows a box plot representation of the obtained results, where the dashed red line is the median of the data, the bottom and top of each box represent the 25th and 75th percentiles, and the two strokes encompass the minimum and maximum values. We observe that the ACO-RRT* algorithm finds a better, but slower, solution when compared with the other algorithms. The ACO-RRT* algorithm is slower because the ACO requires some time to place the ants in the best positions to guide the tree's growth. During the first iterations of the algorithm, the ants are not correctly placed and therefore the planner cannot find a path between the initial and goal position. Conversely, we can conclude that RRT is the fastest algorithm to find a first solution to the path-planning problem, although it has the highest cost. We exploit this capability in our anytime implementation to find rapidly a first solution.

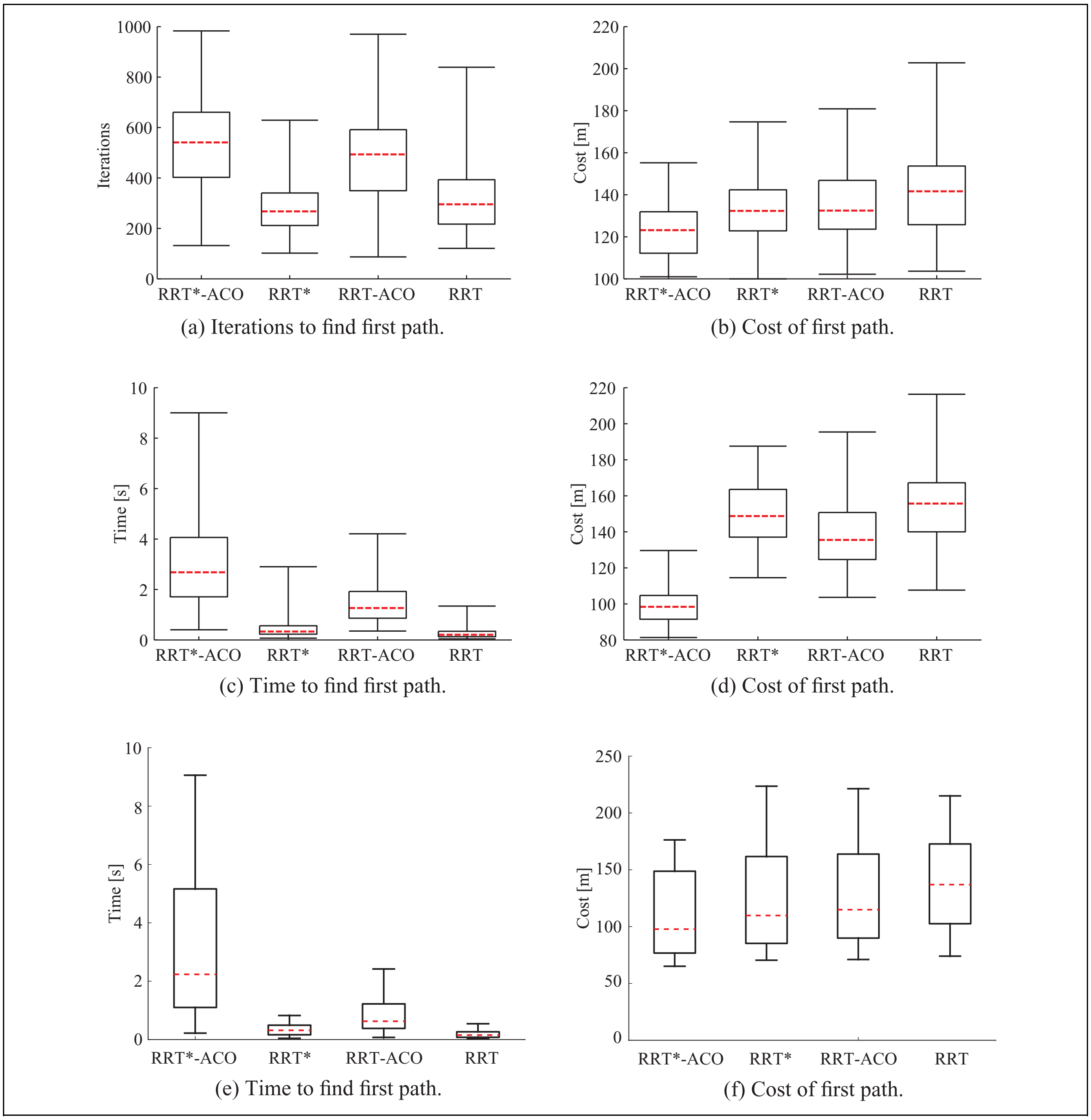

Figure 6. (a, b) Scenario 1. Multiple rectangles. (c, d) Scenario 2. Narrow passage. (e, f) Scenario 3. Maze. Box plot representation of the number of iterations and time to find a first path and its associated cost.

Algorithm performance with time

Once we have found a first path, we aim to improve it to reach the optimal solution. Figure 7 shows the evolution of the best path found over the number of iterations and time. We accompany these figures with a simulation of the algorithm's complexity for the three scenarios. The results of the proposed ACO-RRT* algorithm are compared with ACO-RRT, RRT* and RRT algorithms.

The curves in Figure 7 correspond to the mean value calculated over 100 runs. For each of the curves in Figures 7(a), (c) and **(e)**, we have considered the worst case; that is each of the curves starts when a path was found in all the 100 runs. We observe that the ACO-RRT* algorithm offers a superior performance over time. However, for scenario 3, the performance is similar to the one offered by RRT*. This is because Scenario 3 is more structured and therefore, once the algorithm finds a first solution, it has little room for improvement. These results naturally led us to formulate the anytime ACO-RRT* algorithm. Although the solution offered by the RRT algorithm in the first place is of worse

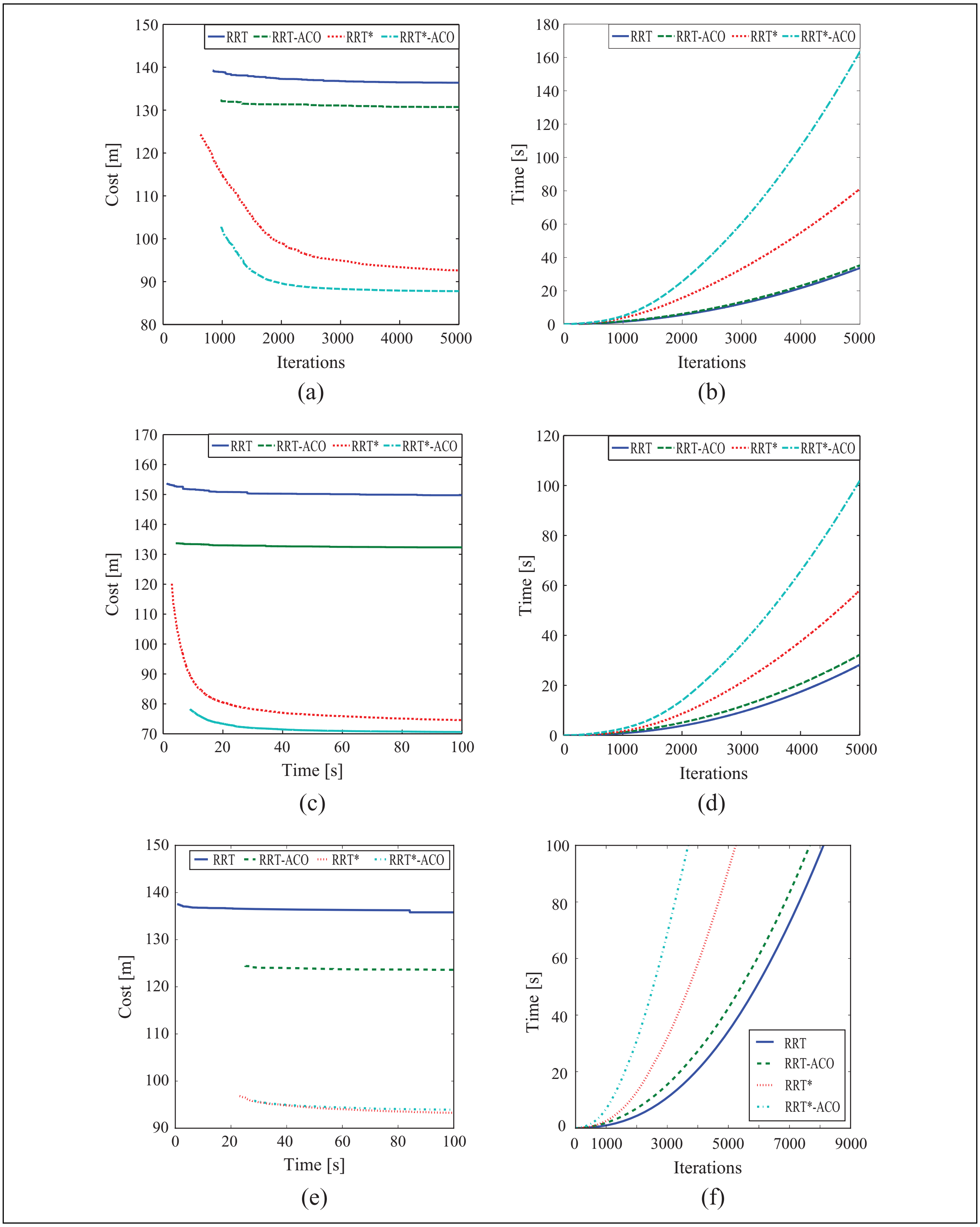

Figure 7. (a, b) Scenario 1: multiple rectangles. (c, d) Scenario 2: narrow passage. (e, f) Scenario 3: maze. Evolution of the best path cost over time and iterations after finding a first solution. The right hand side shows an evaluation of the algorithm time complexity. (a) Path cost versus iterations. (b, d, f) Time versus iterations. (c, e) Path cost versus time.

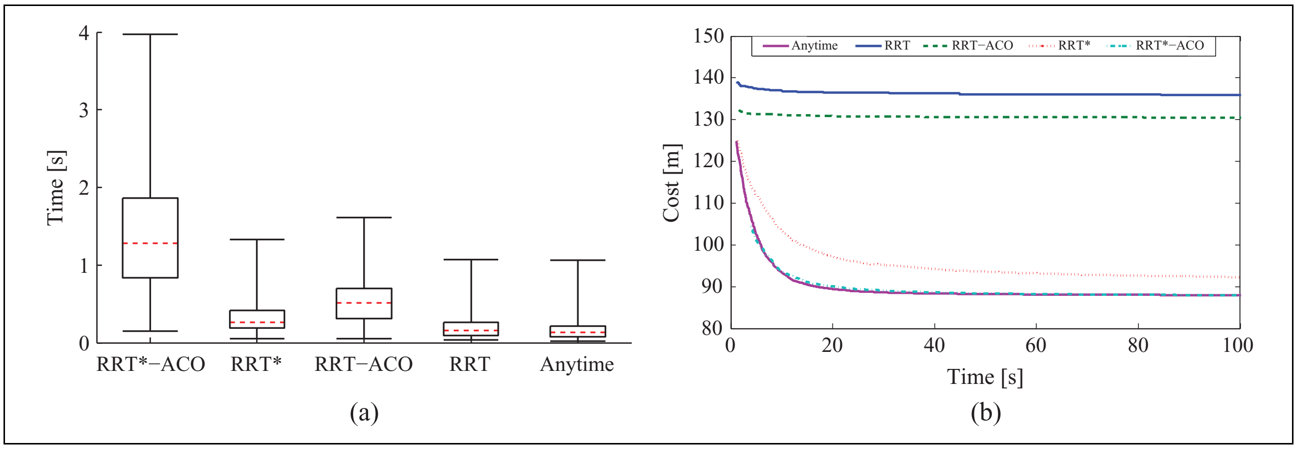

Figure 8. Scenario 1: multiple rectangles, anytime ACO-RRT* performance. (a) Box plot representation of the time to find a first path. (b) Evolution of the best path cost over time once we have found a first solution.

quality, we expect to improve it using ACO-RRT*. We expect this combination to incur an increase in performance over time.

Note that the introduction of the ACO increases the algorithm's computational complexity. This is mainly because of the time needed to compute the ant's utility and update the table that contains the ants. However, this additional complexity is beneficial, since the ACO-RRT/RRT* offers a better performance.

Anytime ACO-RRT performance*

Figure 8 shows the performance of the anytime implementation of the algorithm. We would assume that the RRT and anytime curves should start at the same position, since both of them start running the RRT algorithm. However, here we represent the first moment in which we have found a path for all the 100 algorithm runs. They would then start at the same point if the number of runs approaches infinity. This algorithm is the fastest to find a first path (equal to RRT) and has the same evolution of the performance over time (equal to ACO-RRT*).

Performance with respect to algorithm parameters

For Scenario 1, we evaluated the evolution of the path cost over time with respect to the number of ants, the exploitation–exploration trade-off parameter, and the evaporation rate; while keeping the not-analyzed parameters constant, according to Table 1. We did not simulate the influence of varying q, since this is strongly correlated with k. This allows us to keep q fixed and just modify the number of ants k.

Figure 9 shows the performance with respect to the number of ants. We also performed simulations with a smaller number of ants but the algorithm was not able to converge to any solution in the given planning time. We observe as well that 50 ants corresponds to the best solution. Increasing the number of ants, however, incurs a decrease in performance. The explanation of such behaviour comes from the trade-off that exists between including more ants to better learn the sampling distribution, and the complexity added at the sampling procedure when increasing the number of ants.

To analyze the impact of the α factor in the algorithm performance over time, we keep constant $\hat{\alpha}$ and vary $\check{\alpha}$ between 0 and 1 (see Figure 9). As we could expect, for extreme values of $\check{\alpha}$, the algorithm does not find a solution. For the remaining values, the performance varies only slightly.

In Figure 9, we observe as well that the algorithm does not converge only for the extreme values of the convergence rate ξ. As in the previous case, it is important that performance does not drastically change as we vary this parameter.

Examples of paths planned with the ACO-RRT algorithm*

We have analyzed the different parameters that influence the algorithm's performance. In addition, we include in Figure 10 three snapshots of the paths planned after running our proposed ACO-RRT* algorithm. The figures show the resulting trajectory, the samples that conform the tree, and the ants at the end of the algorithm's execution. We can observe that most of the ants are placed in the region of the state space that contains the optimal trajectory. This results in the presence of more samples in this region, which is the goal of our algorithm. For Scenario 3, it can be seen that the first path found is already very close to the optimal path.

Conclusions and future work

In this work, we have proposed and analyzed a novel path-planning algorithm (ACO-RRT*) based on rapidly exploring random trees (RRTs). We have modified the RRT algorithm sampling strategy so that the current tree

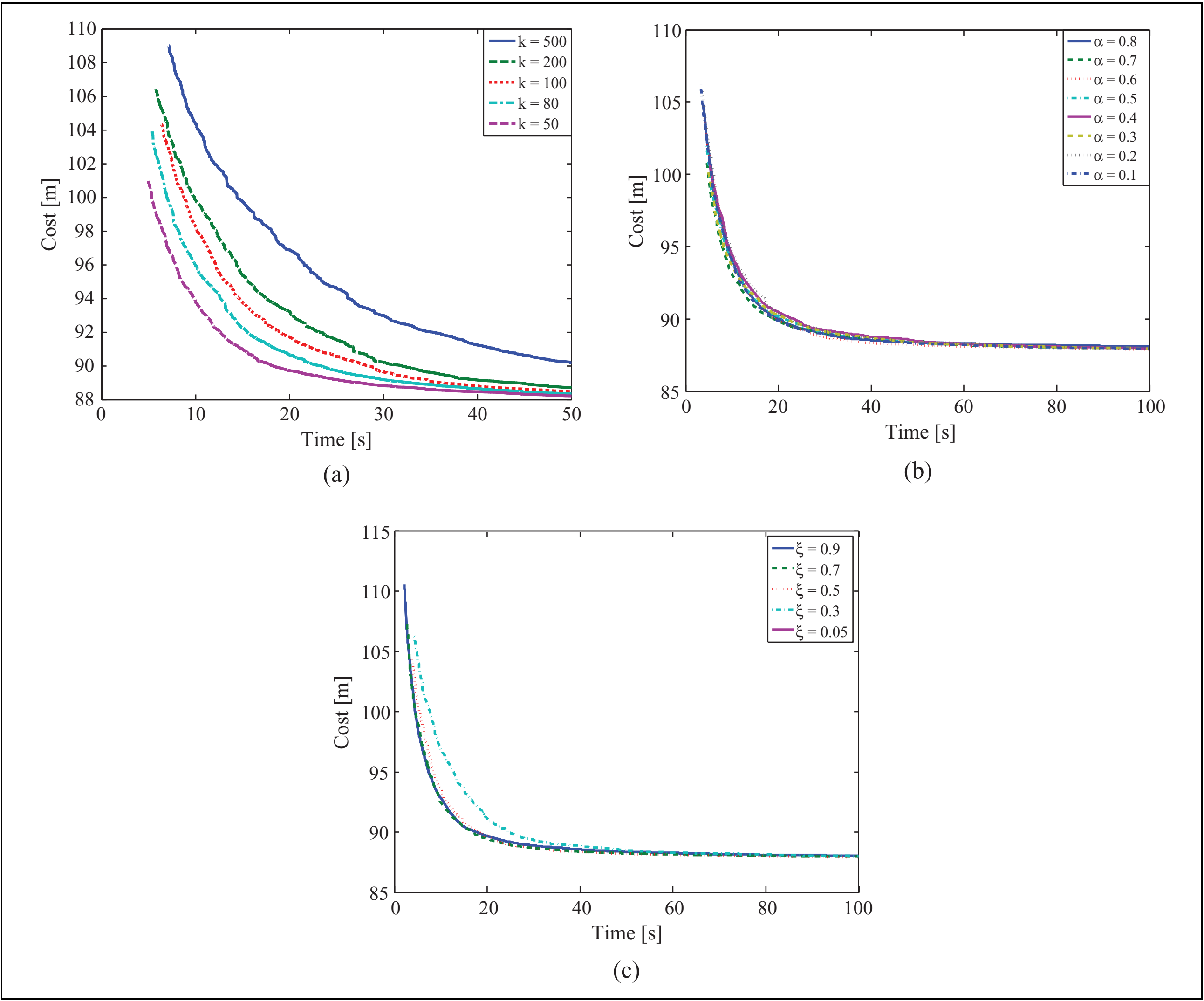

Figure 9. Scenario 1: multiple rectangles; Analysis of the algorithm performance with respect to: (a) number of ants, *k*; (b) the exploitation–exploration trade-off parameter once we have found a first path, $\breve{\alpha}$; (c) evaporation rate, ξ. Each simulation corresponds to the variation of the specific parameter, while leaving the rest constant according to the values of Table 1.

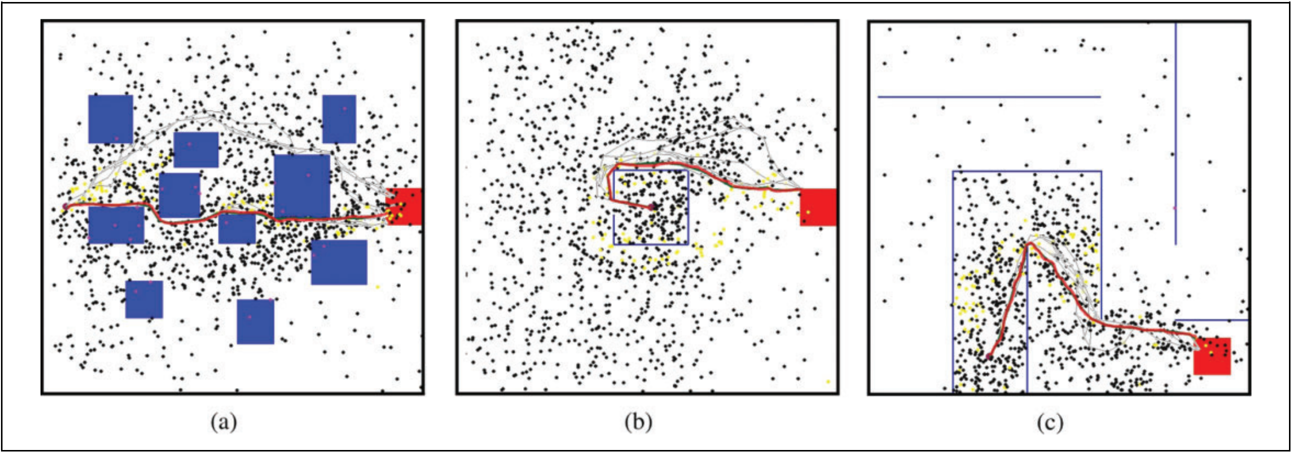

Figure 10. Example of one path planned with the ACO-RRT* algorithm for each of the analyzed scenarios: (a) Scenario 1; (b) Scenario 2; (c) Scenario 3. The large pink dot is the starting position. The red square is the goal region. The final path is coloured red. The black dots are the samples generated by the algorithm. The yellow and pink dots represent the ants' positions at the end of the algorithm's execution. The ants represented with the pink dots are the ones that were placed on top of an obstacle during the algorithm's execution.

influences the sampling. This is done by defining a novel utility function in combination with the ant colony optimization algorithm. The utility function is defined to trade off between (i) exploiting the current solution and (ii) exploring the states' space. We have compared the ACO-RRT* algorithm performance with the RRT and RRT* algorithms in three challenging scenarios. The proposed algorithm is able to find a higher quality first path than the other alternatives (improvement factor between a 1.08 and 1.5). In addition, the results suggest that our algorithm approaches the optimal solution 3.6 faster than the RRT* algorithm. However, it takes more time to find the first path. To reduce this time, we extended the algorithm to an anytime version. Here, the algorithm searches a first path as quickly as possible regardless of the path's cost, and then improves it using the ACO-RRT* algorithm. Simulations results demonstrate that this anytime ACO-RRT* outperforms the state-of-the-art RRT/RRT* algorithms. We also compared the algorithms' performance, by varying the different parameters that conform the algorithms.

Future steps are to encompass the experimental validation of the algorithm with a robot-in-the-loop. We also aim to learn the optimal algorithm's parameters. This could be done by reinforcement learning, where the robot could automatically tune these parameters by analyzing the current solutions as it moves. We are working to extend this framework to handle more complex objective functions; for example autonomous exploration and handling model uncertainty. In addition, a future goal is to perform path planning in an environment populated with obstacles and moving agents that can cooperate. In this situation, we believe we could obtain a great improvement by exchanging ants between agents. Here, we have proposed a framework to treat the rapidly exploring random trees algorithm. We believe that incorporating some of the state-of-the-art methods in this framework, by the definition of proper utility functions, will lead to a greatly superior performance.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship and/or publication of this article: The work of author L. Merino is partially funded by the European Commission FP7 (grant number 611153 TERESA)

References

1. LaValle SM and Kuffner JJ. Randomized kinodynamic planning. *Int J Rob Res* 2001; 20(5): 378–400.
2. Karaman S and Frazzoli E. Sampling-based algorithms for optimal motion planning. *Int J Rob Res* 2011; 30(7): 846–894.
3. Socha K and Dorigo M. Ant colony optimization for continuous domains. *Eur J Oper Res* 2008; 185(3): 1155–1173.
4. Kavraki LE, Svestka P, Latombe JC, et al. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Trans Rob Autom* 1996; 12(4): 566–580.
5. Arslan O and Tsiotras P. Use of relaxation methods in sampling-based algorithms for optimal motion planning. In: *2013 IEEE international conference on robotics and automation (ICRA)*, Karlsruhe, Germany, 6–10 May 2013, pp. 2421–2428. Piscataway, NJ: IEEE.
6. Arslan O and Tsiotras P. Dynamic programming guided exploration for sampling-based motion planning algorithms. In: *2015 IEEE international conference on robotics and automation (ICRA)*, Seattle, WA, 26–30 May 2015, pp.4819–4826. Piscataway, NJ: IEEE.
7. Gammell JD, Srinivasa SS and Barfoot TD. Informed RRT*: optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. *arXiv preprint* 2014; arXiv:14042334.
8. Gammell JD, Srinivasa SS and Barfoot TD. BIT*: batch informed trees for optimal sampling-based planning via dynamic programming on implicit random geometric graphs. Technical report no. TR-2014-JDG006, 2014. Toronto, ON: ASRL University of Toronto.
9. Karaman S, Walter MR, Perez A, et al. Anytime motion planning using the RRT*. In: *2011 IEEE international conference on robotics and automation (ICRA)*, Shanghai, China, 9–13 May 2011, pp.1478–1483. Piscataway, NJ: IEEE.
10. Salzman O and Halperin D. Asymptotically near-optimal RRT for fast, high-quality, motion planning. In: *2014 IEEE international conference on robotics and automation (ICRA)*, Hong Kong, 31 May 2014–7 June 2014, pp.4680–4685. Piscataway, NJ: IEEE.
11. Salzman O and Halperin D. Asymptotically-optimal motion planning using lower bounds on cost. In: *2015 IEEE international conference on robotics and automation (ICRA)*, Seattle, WA, 26–30 May 2015, pp.4167–4172. Piscataway, NJ: IEEE.
12. Denny J, Shi K and Amato NM. Lazy toggle PRM: a single-query approach to motion planning. In: *2013 IEEE international conference on robotics and automation (ICRA)*, Karlsruhe, Germany, 6–10 May 2013, pp.2407–2414. Piscataway, NJ: IEEE.
13. Kuffner JJ and LaValle SM. RRT-connect: an efficient approach to single-query path planning. In: *IEEE international conference on robotics and automation, 2000, proceedings ICRA'00*, San Francisco, CA, 24–28 April 2000, volume 2, pp.995–1001. Piscataway, NJ: IEEE.
14. Urmson C and Simmons RG. Approaches for heuristically biasing RRT growth. In: *Proceedings 2003 IEEE/RSJ international conference on intelligent robots and systems, 2003 (IROS 2003)*, Las Vegas, NV, 27–31 October, 2003, volume 2, pp.1178–1183. Piscataway, NJ: IEEE.
15. Denny J, Morales M, Rodriguez S, et al. Adapting RRT growth for heterogeneous environments. In: *2013 IEEE/RSJ international conference on intelligent robots and systems*

(IROS), Tokyo, Japan, 3–7 November 2013, pp.1772–1778. Piscataway, NJ: IEEE.

16. Amato NM and Song G. Using motion planning to study protein folding pathways. *J Comput Biol* 2002; 9(2): 149–168.
17. Guibas LJ, Holleman C and Kavraki LE. A probabilistic roadmap planner for flexible objects with a workspace medial-axis-based sampling approach. In: *Proceedings 1999 IEEE/RSJ international conference on intelligent robots and systems, 1999. IROS'99*, Kyongju, Republic of Korea, 17–21 October 1999, volume 1, pp.254–259. Piscataway, NJ: IEEE.
18. Hsu D, Jiang T, Reif J, et al. The bridge test for sampling narrow passages with probabilistic roadmap planners. In: *Proceedings ICRA'03 IEEE international conference on robotics and automation, 2003*, Taipei, Taiwan, 14–19 September 2003, vol. 3, pp.4420–4426. Piscataway, NJ: IEEE.
19. Siméon T, Laumond JP and Nissoux C. Visibility-based probabilistic roadmaps for motion planning. *Adv Rob* 2000; 14(6): 477–493.
20. Jaillet L, Yershova A, La Valle SM, et al. Adaptive tuning of the sampling domain for dynamic-domain RRTs. In: *2005 IEEE/RSJ international conference on intelligent robots and systems*, Edmonton, AB, Canada, 2–6 August 2005, pp. 2851–2856. Piscataway, NJ: IEEE.
21. Burns B and Brock O. Toward optimal configuration space sampling. In: *Proceedings of robotics: science and systems*, Cambridge, MA, 8–11 June 2005, pp.105–112.
22. Jaillet L, Cortés J and Siméon T. Sampling-based path planning on configuration-space costmaps. *IEEE Trans Rob* 2010; 26(4): 635–646.
23. Janson L, Schmerling E, Clark A, et al. Fast marching tree: a fast marching sampling-based method for optimal motion planning in many dimensions. *Int J Rob Res* 2015; 34(7): 883–921.
24. Kim D, Lee J and and Yoon Se. Cloud RRT*: sampling cloud based RRT*. In: *2014 IEEE international conference on robotics and automation (ICRA)*, Hong Kong, 31 May–7 June 2014, pp.2519–2526. Piscataway, NJ: IEEE.
25. Nasir J, Islam F, Malik U, et al. RRT*-smart: a rapid convergence implementation of RRT*. *Int J Adv Rob Syst* 2013; 10.
26. Rickert M, Sieverling A and Brock O. Balancing exploration and exploitation in sampling-based motion planning. *IEEE Trans Rob* 2014; 30(6): 1305–1317.
27. Akgun B and Stilman M. Sampling heuristics for optimal motion planning in high dimensions. In: *2011 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, San Francisco, CA, 25–30 September 2011, pp. 2640–2645. Piscataway, NJ: IEEE.
28. Alterovitz R, Patil S and Derbakova A. Rapidly-exploring roadmaps: weighing exploration vs. refinement in optimal motion planning. In: *2011 IEEE international conference on robotics and automation (ICRA)*, Shanghai, China, 9–13 May 2011, pp.3706–3712. Piscataway, NJ: IEEE.
29. Persson SM and Sharf I. Sampling-based A* algorithm for robot path-planning. *Int J Rob Res* 2014; 33(13): 1683–1708.
30. Morales M, Tapia L, Pearce R, et al.A machine learning approach for feature-sensitive motion planning. In: Erdmann M, Hsu D and Overmars M et al. (eds) *Algorithmic foundations of robotics VI*. Berlin: Springer, 2005. pp. 361–376.
31. Diankov R and Kuffner J. Randomized statistical path planning. In: *2007 IEEE/RSJ international conference on intelligent robots and systems*, San Diego, CA, 29 October– 2 November 2007, pp.1–6. Piscataway, NJ: IEEE.
32. Kobilarov M. Cross-entropy motion planning. *Int J Rob Res* 2012; 31(7): 855–871.
33. Mohamad MM, Taylor NK and Dunnigan MW. Articulated robot motion planning using ant colony optimisation. In: *2006 3rd international IEEE conference on intelligent systems*, London, UK, 4–6 September 2006, pp.690–695. Piscataway, NJ: IEEE.
34. Dorigo M, Maniezzo V and Colorni A. Ant system: optimization by a colony of cooperating agents. *IEEE Trans Syst Man Cybern Part B Cybern* 1996; 26(1): 29–41.
35. Ferguson D and Stentz A. Anytime RRTs. In: *2006 IEEE/RSJ international conference on intelligent robots and systems*, Beijing, China, 9–15 October 2006, pp.5369–5375. Piscataway, NJ: IEEE.