Improving the performance of RRT planners using informed sampling

Borghini Alessia
Sapienza University
Rome, Italy

Corvitto Luca
Sapienza University
Rome, Italy

Di Valerio Federico
Sapienza University
Rome, Italy

{borghini.1834906, corvitto.1835668, divalerio.1835405}@studenti.uniroma1.it

I. Introduction

In this work we present an analysis on the Rapidly-exploring Random Tree star (RRT*) algorithm, along with its informed version, both in euclidean and kinodynamic space.

The euclidean domain is a simple geometric space where the cost of a path is measured by its length. The kinodynamic domain is a more complex space where the cost of a path also depends on the dynamics of the system. For unconstrained robots, straight lines through the state space represent optimal trajectories, but for kinodynamic systems such trajectories are typically not feasible due to the system's differential constraints [5]. In Figure 1 are shown some instances of unconstrained (figures (a) and (b)) and nonholonomic (figure (c)) robots.

RRT [9] is an algorithm designed to efficiently search non-convex, high-dimensional spaces by randomly building a space-filling tree. The tree is constructed incrementally from samples drawn randomly from the search space and is inherently biased to grow towards large unsearched areas of the problem. RRT is probabilistically complete, which means that a solution will be found (if one exists) with a probability approaching 1 if one lets the algorithm run long enough [5]. RRT* [1], instead, is an optimized version of RRT, which achieves asymptotic optimality. This means that if one lets the algorithm run long enough an optimal solution will be found with a probability approaching 1. When the algorithm approaches infinity, the RRT* algorithm will deliver the shortest possible path to the goal. While realistically unfeasible, this statement suggests that the algorithm does work to develop a shortest path.

Informed RRT* [3] is an improvement to the RRT* algorithm that increases the rate at which the found solution converges to the optimum. Developed specifically for problems seeking to minimize path-length, this work uses a method to directly sample the subset of a planning problem that contains all possible improvements to a given solution. In doing so, it continues to consider all relevant paths without requiring any additional parameter.

Kinodynamic RRT* [5], in contrast with RRT*, is applicable not only to systems with simple dynamics but introduces a fixed-final-state-free-final-time controller that exactly and optimally connects any pair of states for any system with controllable linear dynamics in state spaces of arbitrary dimension. This algorithm finds asymptotically optimal trajectories in environments with obstacles and bounds on the state and control input, with respect to a cost function that is expressed as a tunable trade-off between the duration of the trajectory and the expended control effort.

(a) 4WD with omniwheels (b) 4WD with mecanum wheels (c) Car-like robot

Fig. 1: In figures (a) and (b) there are omnidirectional robots, in figure (c) a nonholonomic one.

II. Algorithm Explanation

In this Section we will go into the details of the pseudocodes and practical implementation of the RRTs* algorithms.

A. RRT*

Algorithm 1 RRT* and Informed RRT*

Input: x_{start}, x_{goal}

1: $\mathcal{X}_{soln} \leftarrow \emptyset$
2: $V \leftarrow \{x_{start}\}$
3: **for** iteration = 1...N **do**
4: $c_{best} \leftarrow min_{x_{soln} \in \mathcal{X}_{soln}} \{\texttt{Cost}(x_{soln})\}$
5: $x_{rand} \leftarrow \texttt{Sample}(x_{start}, x_{goal}, c_{best})$
6: $x_{nearest} \leftarrow \texttt{Nearest}(x_{rand})$
7: $x_{new} \leftarrow \texttt{Steer}(x_{rand}, x_{nearest})$
8: **if not** $\texttt{IsCollision}(x_{nearest}, x_{new})$ **then**
9: $V \leftarrow \texttt{ChooseParent}(x_{new}, V)$
10: $V \leftarrow \texttt{Rewire}(x_{new}, V)$
11: **if** $\texttt{InGoalRegion}(x_{new}, r_{goal})$ **then**
12: $\mathcal{X}_{soln} \leftarrow \mathcal{X}_{soln} \cup x_{new}$
13: **end if**
14: $V \leftarrow V \cup x_{new}$
15: **end if**
16: **end for**

RRT* algorithm [1], shown in Algorithm 1, finds partially optimal solutions by incrementally modifying the paths graph. Iteratively, the algorithm generates random samples using `Sample` and `Steer` functions (line 6-8) and then checks its validity using the `IsCollision` function (line 9). If the new node x_{new} is valid, the algorithm selects as its parent (`ChooseParent` function at line 10) the node x that has the lowest overall cost from x_{start} to x_{new} passing by x without colliding with any obstacle. Next, the `Rewire` function reduces the costs of nearby nodes by setting their parent to the newly generated node. In order to guarantee the asymptotic optimality [7] of the RRT* algorithm, we need to use a constant γ such that:

$$\gamma > 2(1 + 1/n)^{1/n} \left(\frac{\mu[\mathcal{X}_{free}]}{\zeta_n} \right)^{1/n}, \tag{1}$$

where n is the dimension of the state space $\mathcal{X} = \mathbb{R}^n$, $\mathcal{X}_{free} \subseteq \mathcal{X}$ is the set of the states in the state space that are collision free and ζ_n is the volume of an n-dimensional sphere. The γ constant is employed in the computation of the radius used as an upper bound of the distance between the node and his parent in the `ChooseParent` and `Rewire` functions:

$$r = \gamma(log[i]/i). \tag{2}$$

When the loop ends and the graph is created, the best path connecting x_{start} and x_{goal} is found.

B. Informed RRT*

The Informed version of RRT* (IRRT*) [2] is equal to the base one until the first solution is found. In fact, the path cost can be dramatically improved by restricting subsequent samples to regions (shown in Figure 2) of the state space that can potentially improve the current solution. These regions compose the informed space, $\mathcal{X}_{inf} \subseteq \mathcal{X}$, which, if it is smaller than $\mathcal{X}$, can reduce the search space resulting in faster convergence. One of the differences between the two versions is that IRRT* takes note of the best cost c_{best} to improve the solution, that is needed to estimate the hyper-ellipsoidal informed space. Another one is in how the random states are sampled. In fact, after the first solution is found, the IRRT* `Sample` function samples a random state x_{rand} from $\mathcal{X}_{inf}$. Then, the nearest node ($x_{nearest}$) to x_{rand} is found and the steering function (line 8) is applied in order to obtain a node that is closer to $x_{nearest}$ than x_{rand} is. If the obtained x_{new} does not collide with any obstacle by going through the path toward $x_{nearest}$, `ChooseParent` and `Rewire` are performed. Additionally, IRRT* checks whether x_{new} is in the goal region (line 12) – i.e., the distance from x_{goal} is shorter than a radius r_{goal} and a straight path to the goal is feasible – if so it is added to the list of the solutions $\mathcal{X}_{soln}$. This version of the algorithm knows at each iteration which is the current best solution, in contrast to the base one that searches for it when the graph is completed.

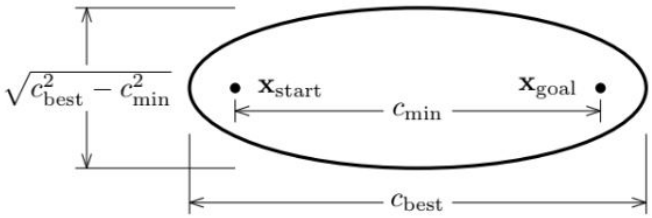


Fig. 2: Informed space approximation in euclidean domain.

C. RRT* Kinodynamics

Algorithm 2 RRT* in Kinodynamic Domain

Input: x_{start}, x_{goal}

$V \leftarrow \{x_{start}\}$
for iteration = 1...N **do**
  $x_{rand} \leftarrow$ `Sample()`
  $x_{new}, x_{nearest}, V \leftarrow$ `ChooseParent`(x_{rand}, V)
  if x_{new} **is not** `None` **then**
    $V \leftarrow$ `Rewire`(x_{new}, V)
    if `isBest`(x_{new}) **then**
      $x_{best} \leftarrow x_{new}$
      $c_{best} \leftarrow$ `Cost`(x_{new})
    end if
    $V \leftarrow V \cup x_{new}$
  end if
end for

In Algorithm 2, the RRT* in the kinodynamic domain [4], [5] is described. The overall structure is similar to the RRT* base algorithm and the main differences are dictated by the kinodynamic domain. In this case: $\mathcal{X} = \mathbb{R}^n$ is the state space and $\mathcal{U} = \mathbb{R}^m$ is the control input space. In our implementation $n = 4$ and $m = 2$ because we consider a double integrator control system with the following dynamics:

$$\dot{\mathbf{x}}(t) = A\mathbf{x}(t) + B\mathbf{u}(t) + \mathbf{c}, \tag{3}$$

where the states are $\mathbf{x}(t)^T = \begin{pmatrix} p_x & p_y & v_x & v_y \end{pmatrix}$, and the control inputs are $\mathbf{u}(t)^T = \begin{pmatrix} u_x & u_y \end{pmatrix}$. Moreover, $A \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{n \times m}$ and $\mathbf{c} \in \mathbb{R}^n$ are constant. In particular:

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \; B = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}, \; \mathbf{c} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \tag{4}$$

Due to the different state space, also the cost differs, since it is no more a geometric distance. In fact, the cost $c[\pi]$ of a trajectory π is defined by the function:

$$c[\pi] = \int_0^\tau (1 + \mathbf{u}[t]^T R \mathbf{u}[t]) dt, \tag{5}$$

which penalizes both the duration of the trajectory and the expended control effort. The $R \in \mathbb{R}^{m \times m}$ matrix found in the formula represents the cost penalty and is equal to $R = \rho I$, with $\rho > 0$. It weights the cost of the control inputs relative to each other and to the duration of the trajectory. The higher the ρ value is, the less frequently the velocity bounds are exceeded.

Looking at the algorithm in detail, we first have the `Sample` function which simply generates a state not exceeding the imposed limits on position and velocity. In the euclidean domain the `Steer` module guides the growth of the tree towards a sampled state, but not necessarily to its entirety. However, as stated by Webb and van den Berg [5], the formulation of the kinodynamic RRT* used in our case makes it complex to determine partial trajectories of a given maximum cost, so the implemented algorithm directly establishes a complete connection to the sampled state and adds the state itself as a node to the tree. These changes do not affect the asymptotic optimality guarantee of the algorithm, thus the generated sample, x_{rand},

can be directly passed to `ChooseParent`.
The pseudocode of `ChooseParent` can be found in Algorithm 3: for each node in V we compute the $cost$ and $time$, needed to go from the considered $node$ to x_{rand}, and the total cost needed from the $start$ to x_{rand}. The optimal time, $time$, is obtained by solving the Equation 6,

$$\tau^* = argmin\{\tau > 0\}\ c[\tau], \tag{6}$$

where the $c[\tau]$ is the following cost function

$$c[\tau] = \tau + (\mathbf{x}_1 - \overline{\mathbf{x}}[\tau])^T G[\tau]^{-1}(\mathbf{x}_1 - \overline{\mathbf{x}}[\tau]). \tag{7}$$

This cost function considers the stability of the system through the *weighted controllability Gramian* term ($G[\tau]$), which is the solution of the following Lyapunov equation for the *fixed final state, fixed final time optimal control problem*:

$$\begin{aligned} \dot{G}[t] &= AG[t] + G[t]A^T + BR^{-1}B^T,\ G[0] = 0, \\ G[t] &= \int_0^t \exp[A(t-t')]BR^{-1}B^T \exp[A^T(t-t')]dt'. \end{aligned} \tag{8}$$

Further, let $\overline{x}[t]$ describe what the state x (starting in x_0 at time 0) would be at time t if no control input were applied:

$$\overline{\mathbf{x}}[t] = exp[At]\mathbf{x}_0 + \int_0^t exp[A(t-t')]\mathbf{c}dt', \tag{9}$$

which is the solution to the differential equation:

$$\dot{\overline{\mathbf{x}}}[t] = A\overline{\mathbf{x}}[t] + \mathbf{c},\ \overline{\mathbf{x}}[0] = \mathbf{x}_0. \tag{10}$$

Then, the optimal control policy for the *fixed final state, fixed final time optimal control problem* is given by:

$$\mathbf{u}[t] = R^{-1}B^T \exp[A^T(\tau - t)]G[\tau]^{-1}(\mathbf{x}_1 - \overline{\mathbf{x}}[\tau]), \tag{11}$$

which is an open-loop control policy. To find an optimal trajectory between two states, we solve the fixed final state, *free* final time optimal control problem, in which we can choose the arrival time τ freely to minimize the cost function. The derivative of $c[\tau]$, Equation 12, has been computed and set to zero, then the roots of this polynomial have been extracted in order to obtain the optimal arrival time τ^* (called $time$ in the implementation) and the corresponding optimal cost $c[\tau^*]$ (called $cost$).

$$\begin{aligned} \dot{c}[\tau] &= 1 - 2(A\mathbf{x}_1 + \mathbf{c})^T \mathbf{d}[\tau] - \mathbf{d}[\tau]^T BR^{-1}B^T \mathbf{d}[\tau], \\ \mathbf{d}[\tau] &= G[\tau]^{-1}(\mathbf{x}_1 - \overline{\mathbf{x}}[\tau]). \end{aligned} \tag{12}$$

The immediate **if** statement checks that the cost to go from $node$ to x_{rand} is lesser than a radius r and that the total cost is lesser than the current minimum cost. The radius has to be computed starting from $v[r]$, the squared volume of the largest ellipsoid that constitutes the set of states that can be reached from x_i with cost less than r (forward reachable set):

$$v[r] = max\{0 < \tau < r\}\ \zeta_n^2 \det[G[\tau](r - \tau)], \tag{13}$$

where ζ_n is the volume of an n-dimensional sphere. The neighbor radius r that is selected in the i-th iteration of the algorithm is then computed by solving the following polynomial equation for r,

$$v[r] = (\gamma log[i]/i)^2, \tag{14}$$

where $\gamma > 2^n(1 + 1/n)\mu[\mathcal{X}_{free}]$ guarantees the asymptotic optimality of the algorithm [8]. Then, if the cost is lower than r, we compute the states and inputs, respectively s and u, and we check that these are actually valid: do not end up in a collision or exceed the imposed limits (states and input limits). The states s are obtained by taking the inputs u, computed through 11, and $\mathbf{x}[t]$ from the following Equation:

$$\begin{aligned} \begin{bmatrix} \mathbf{x}[t] \\ \mathbf{y}[t] \end{bmatrix} = \exp\left[\begin{bmatrix} A & BR^{-1}B^T \\ 0 & -A^T \end{bmatrix} (t - \tau^*) \right] \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{d}[\tau^*] \end{bmatrix} + \\ \int_{\tau^*}^t exp \begin{bmatrix} A & BR^{-1}B^T \\ 0 & -A^T \end{bmatrix} (t - t') \begin{bmatrix} \mathbf{c} \\ \mathbf{0} \end{bmatrix} dt'. \end{aligned} \tag{15}$$

If they are valid then we update the minimum cost, minimum time and the best node x_{min}. Exited from the **For** loop we update the graph, adding x_{rand} which has been given a new parent x_{min}. This node x_{min} in Algorithm 2 is called $x_{nearest}$ and x_{new} is the updated x_{rand}. If `ChooseParent` returns succesfully x_{new} won't be `None` and the `Rewire` function will be called.

Algorithm 3 ChooseParent RRT* in Kinodynamic Domain

Input: x_{rand}, V
Output: x_{rand}, x_{min}, V

$x_{min} \leftarrow None$
$c_{min}, t_{min} \leftarrow \infty$
for $node$ **in** V **do**
  $cost \leftarrow \texttt{Cost}(node, x_{rand})$
  $time \leftarrow \texttt{Time}(node, x_{rand})$
  $c \leftarrow \texttt{Cost}(node) + cost$
  if $cost < r$ **and** $c < c\_min$ **then**
    $u \leftarrow \texttt{GetInputs}(node, x_{rand})$
    $s \leftarrow \texttt{GetStates}(node, x_{rand})$
    $t \leftarrow [0, time]$
    if $\texttt{isValid}(s, t)$ **and** $\texttt{isValid}(u, t)$ **then**
      $c_{min} \leftarrow c$
      $t_{min} \leftarrow time + Time(node)$
      $x_{min} \leftarrow node$
    end if
  end if
end for

Algorithm 4 Rewire RRT* in Kinodynamic Domain

Input: x_{new}, V
Output: V

$c_{imp} \leftarrow 0$
$t_{imp} \leftarrow 0$
$stack \leftarrow ((x_{start}, c_{imp}, t_{imp}))$
while $\texttt{notEmpty}(stack)$ **do**
  $node, c_{imp}, t_{imp} \leftarrow stack.pop()$
  $\texttt{ImproveCostChildren}(node, c_{imp}, t_{imp})$
  $c_{partial} \leftarrow Cost(x_{new}, node)$
  $t_{partial} \leftarrow Time(x_{new}, node)$
  $c_{new} \leftarrow c_{partial} + \texttt{Cost}(x_{new})$
  $t_{new} \leftarrow t_{partial} + \texttt{Time}(x_{new})$
  if $c_{partial} < r$ **and** $c_{new} < \texttt{Cost}(node)$ **then**
    $u \leftarrow \texttt{GetInputs}(node, x_{new})$
    $s \leftarrow \texttt{GetStates}(node, x_{new})$
    $t \leftarrow [0, time]$
    if $\texttt{isValid}(s, t)$ **and** $\texttt{isValid}(u, t)$ **then**
      $c_{imp} \leftarrow \texttt{Cost}(node) - c_{new}$
      $t_{imp} \leftarrow \texttt{Time}(node) - t_{new}$
      $V \leftarrow \texttt{UpdateGraph}(x_{new}, node, c_{new}, t_{new})$
    end if
  end if
  for $child$ **in** $Children(node)$ **do**
    $stack \leftarrow stack \cup (child, c_{imp}, t_{imp})$
  end for
end while

The pseudo-code of the `Rewire` function can be found in Algorithm 4. The purpose of this function, as in the euclidean domain, is to update the overall graph with new information obtained by the insertion of x_{new}, improving the path cost. We do so by firstly creating a stack to store the unexplored nodes with their cost and time improvement gained from the change of their parents. Iteratively, a node is extracted from the stack and whether its parent had cost and time improved, the improvement is propagated to it. Then, we change the current parent node with x_{new} if the new path is feasible and beneficial. Finally, we

update c_{imp} and t_{imp} in order to propagate the improvement to the children nodes, which are added to the stack. When the `Rewire` function complete its execution we check whether the path from x_{new} to x_{goal} is better than the one computed in previous iterations of the algorithm, without caring about collisions. If this is the case the information about the best node and best cost is updated and V gets enlarged by one more node x_{new}. The whole process continues until reaching the last iteration.

D. Informed RRT Kinodynamics*

Informed RRT* in kinodynamic domain [3] differs from the RRT* base version in the introduction of $\mathcal{X}_{inf}$ and consequently with the change of the `Sample` function. The algorithm estimates a non-convex function from which extracts samples using Monte Carlo Markov Chain (MCMC) methods. Similarly to IRRT* algorithm in Section II-B, when the first solution is found, the sampler focuses around that solution to improve it. Computing a closed-form solution to sample $\mathcal{X}_{inf}$, as in the geometric domain, is not trivial because of the dependency of the structure on the start and goal states and because of the potential discontinuities. It is possible, instead, to apply an optimization-based approach in order to produce samples within $\mathcal{X}_{inf}$. In order to extract a sample x in $\mathcal{X}_{inf}$ (line 3, Algorithm 2) we use the Monte Carlo Markov Chain sampling method shown in Algorithm 5.

1) Monte Carlo Markov Chain: Starting from one of the nodes in $\mathcal{X}_{inf}$ we perform the Metropolis-Hastings (MH) algorithm to obtain a new node, x_{next}. Then, if the new node is different from the starting one, we check whether the node is inside the informed space. The minimum-cost trajectory from x_{start} to x_{goal}, passing through x_{next} without consider the collisions has cost $c(\gamma*(x))$. When this cost is lower than c_{best}, x_{next} is in the informed space, otherwise, other samples will be generated.

2) Metropolis Hastings: The algorithm 6 is a MCMC method used to generate samples from a proposed probability distribution which approximates a target distribution. The algorithm is used to create a distribution of samples in the state space that is weighted towards regions that are more likely to contain a path to the goal. The algorithm starts by generating a random sample in the state space. This sample is then perturbed according to a proposal distribution (`sample_from_normal`), which is a function that defines how to move from one sample to another, in our case it is a gaussian distribution, as showed in [3]. The resulting perturbed sample is then evaluated to determine whether it is a better sample than the original one. This evaluation is done using a cost function that incorporates both the cost of reaching the sample and the estimated cost of reaching the goal from the sample, without considering obstacle collisions. If the perturbed sample is better than the original one, it is accepted as the new sample and the Markov Chain gets longer. If it is worse, it may still be accepted with a certain probability, determined by a probability distribution known as the acceptance probability. This probability is a function of the cost of the perturbed sample and the cost of the original sample. The Metropolis-Hastings algorithm is run multiple times to generate a distribution of samples that is weighted towards regions that are more likely to contain a path to the goal. This distribution is used to guide the sampling process in the Informed RRT* algorithm.

Algorithm 5 MCMC Sampler

Input: $\mathcal{X}_{inf}$
Output: x_{next}

1: $x_{next} \leftarrow None$
2: **while not** `In_informed`(x_{next}) **do**
3: $x_0 \leftarrow$ `RandomNode`$(\mathcal{X}_{inf})$
4: $x_{next} \leftarrow x_0$
5: **while** $x_{next} = x_0$ **do**
6: $x_{next} \leftarrow$ `Metropolis_Hastings`$(x_0, \mathcal{X}_{inf})$
7: **end while**
8: **end while**

Algorithm 6 Metropolis-Hastings MCMC

Input: x_{i-1}, c_{best}

$x'_i \leftarrow \texttt{sample\_from\_normal}(x_{i-1}, \sigma)$
$\alpha \leftarrow \texttt{LL}(x'_i, \Sigma) - \texttt{LL}(x_{i-1}, \Sigma)$
if $\texttt{sample\_random}(0.0, 0.1) < \alpha$ **then**
    return x'_i
end if
return x_{i-1}

III. Simulations

In this section, we will provide the description of the algorithms implementation along with the details concerning the parameters and the environment we used.

A. Implementation Details

The code[1] is mostly written in Python language, using the Sympy[2] library [6] to handle mathematical symbols. The first version of the kinodynamic algorithms used the base sympy mathematical functions with no particular optimizations, resulting in low speed performance. Then, we embedded lambda functions in order to compute symbolically only once the highest computationally intensive functions and at each iteration only the replacement of the numerical values was performed. This resulted in a relative speed up, however still a speed that did not allow us to extensively make simulations. In the last version, we moved the most intensive parts of the algorithms to MATLAB. To achieve this we used the MATLAB Engine API[3] as bridge from Python to MATLAB. Using SnakeViz[4] we noticed that the most computationally intensive functions, as expected, were `ChooseParent` and `Rewire`. These improvements resulted in a 2-orders of magnitude's speed up.

B. Environments

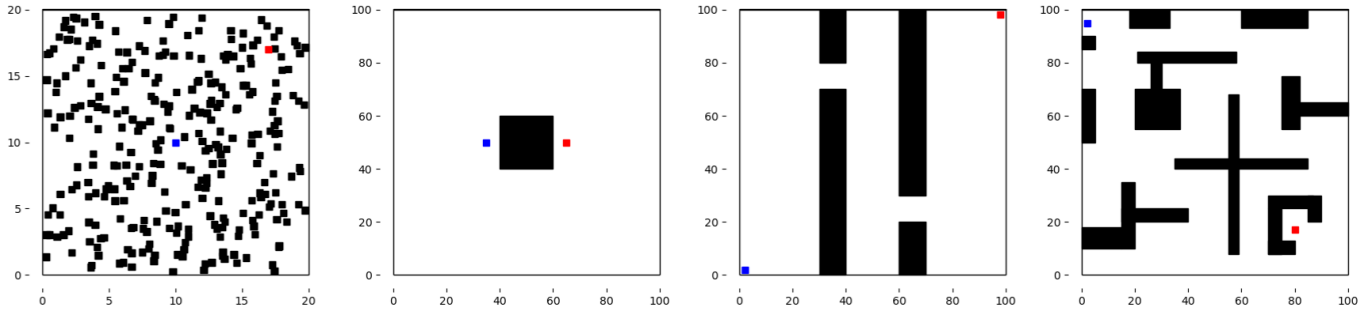
For the simulations we opted for 4 different environments, one random generated and three manually designed. These environments are shown in Figure 3.
The random environment (Figure 3(a)) is a 20×20 square, built with a set number of squared obstacles (after different trials we chose a number ≤ 300) of dimension 0.5×0.5 in the euclidean environment. We took inspiration from the environment built by Gammell et al. [2]. For the sake of reproducibility we chose and set a specific seed (700) for the creation of the environment. The first fixed environment is a 100×100 square with a unique big 20×20 square obstacle set at its centre. This environment is the simplest and the best one to show the differences between the informed and non-informed versions of the algorithms, both in the euclidean and the kinodynamic space. Also this one was inspired by the environments shown by Gammell et al. [2]. The environment for the kinodynamic simulations (Figure 3(c)) is a 100×100 square and was built taking inspiration from Webb and Van Den Berg [5]. This is one of the simplest environment to show the performance of the kinodynamic version of the algorithm. The last one (Figure 3(d)) is a 100×100 maze environment inspired by LaValle [9] and is used in order to get a sense of how the four algorithms differ visually and to compare the informed and non-informed versions respectively in the euclidean and kinodynamic space.

C. Parameters

A fundamental aspect of the simulations is the choice of the parameters. Indeed, we noticed that in the euclidean algorithms the `step_len` and `delta` were related. The `step_len` is the maximum distance between the last generated node and the nearest node of the tree, while the `delta` parameter is the minimum distance between the nodes and the obstacles. If the `delta` is too small it could be possible that, when we randomly generate the obstacles, the start and the goal nodes may be close enough to them, heavily reducing the probability of connecting a new node. That happens also because the `step_len` is too big with respect to the distance towards the obstacles, making the output of the `Steer` function a colliding node.

The informed version of the kinodynamic RRT* uses a probabilistic sampler that generate new nodes within X_{inf}. Given $x_0 \in X_{inf}$, center of the gaussian distribution used in the transition function, the standard deviation σ value changes the shape of the probability distribution from which the samples are extracted to build the Markov Chain. The higher the σ value is, the greater the maximum distance of the extracted nodes from the center of the gaussian. Based on this value, it is also necessary to choose the `covariance` Σ of the proposal distribution used to calculate the log likelihood of the samples, thus

[1]Our implementation: https://github.com/ABorghini/Improving-the-performance-of-RRT-planners-using-informed-sampling
[2]Sympy library: https://www.sympy.org/en/index.html
[3]MATLAB Engine API for Python: https://www.mathworks.com/help/matlab/matlab_external/install-the-matlab-engine-for-python.html
[4]SnakeViz: https://jiffyclub.github.io/snakeviz/



(a) Random Environment (b) Square Environment (c) Kinodynamic Fixed Environment (d) Maze Fixed Environment

Fig. 3: In Figure (a) there is the randomly generated environment containing 300 obstacles, in Figure (b) the environment with a square in the center, in Figure (c) there is the 2-dimensional version of the environment used in [5] for quadrotor helicopter experiments and in Figure (d) the maze-like environment.

influencing the acceptance ratio. In fact, a good the acceptance ratio for the MH algorithm has a value around 25% [10].

During our simulations, we changed the `seed` of the random processes several times because if the same `seed` is used every time, the simulation will produce the same tree, and therefore the same solution, at the same exact iteration, will be found every time the simulation is run. This can lead to a biased analysis and may not accurately represent the true performance of the algorithms. By using different seeds, the simulations explore different paths, providing a more robust and comprehensive understanding of the problem. This can lead to better decision-making and more effective planning. Overall, using different seeds for random processes in simulations is essential to ensure a fair and unbiased representation of the problem at hand.

IV. Results

The Results section presents a comparative analysis of the performance of the RRT* and Informed RRT* algorithms in the euclidean and kinodynamic domains. We evaluate the algorithms on different environments in each domain, using different metrics such as computation time and path cost, taking into account iterations and number of nodes. The section also discusses the advantages and limitations of each algorithm.

A. Comparison between RRT and Informed RRT* in Euclidean domain*

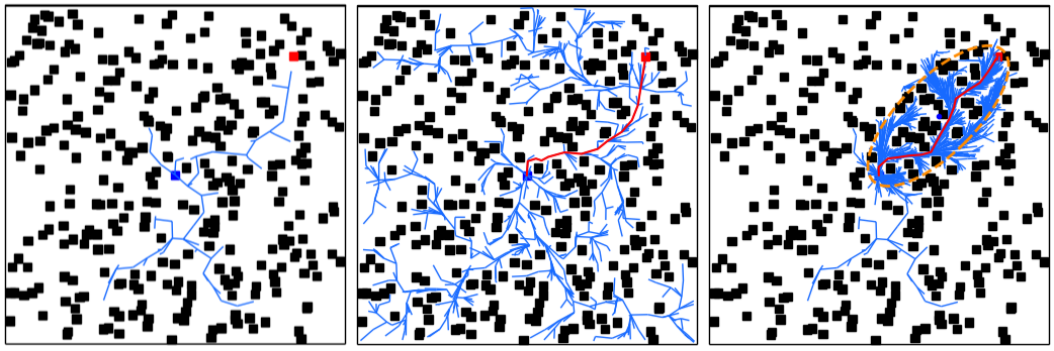
In the euclidean domain we conducted experiments on three environments: they are showed in Figures 3(a), 3(b), 3(d). In the randomly generated environment (Figure 4) we can see that IRRT* finds a better solution compared to RRT*, this is probably related to the fact that the informed version behaves better in difficult environments when the heuristic is appropriate. In fact, the area in which the algorithm searched (delimited by the orange segmented ellipsoid) was way smaller compared to the complete environment in which RRT* instead had to search. In this specific experiment, after the first solution was found (same for both algorithms), IRRT* found a better next solution and it was faster than RRT* in doing so (iteration = 580, cost = 12.09 and iteration = 2165, cost = 12.54 respectively). The same happened for the final solution (iteration = 2860, cost = 10.86 and iteration = 2986, cost = 11.64); while this is a specific case, in our experiments this was found to be a common trend in the randomly generated environments as shown in 10(a).

In Figure 5 instead, we show how the informed region shrinks during the evolution of the path of the IRRT* algorithm. The informed region decreases in size accordingly to [2] (shown in Figure 2): the major radius of the ellipsoid is obtained through the minimum cost c_{best} found until that moment, while the minor one is computed by taking the square root of the difference between c_{best} and c_{min}, where c_{min} is the euclidean distance from start to goal.

In order to provide a better visualization of how the informed algorithm enhance the solution, we show in Figures 6(a) and 6(b) the different paths obtained in a simple environment such as the square one. After the first solution is found, the informed algorithm only considers the informed subset as sample space, so the size of the search space is independent from the dimension of the entire environment. This particular experiment was replicated from [2] and confirms that IRRT* outperforms RRT* in both convergence rate and quality of solution.

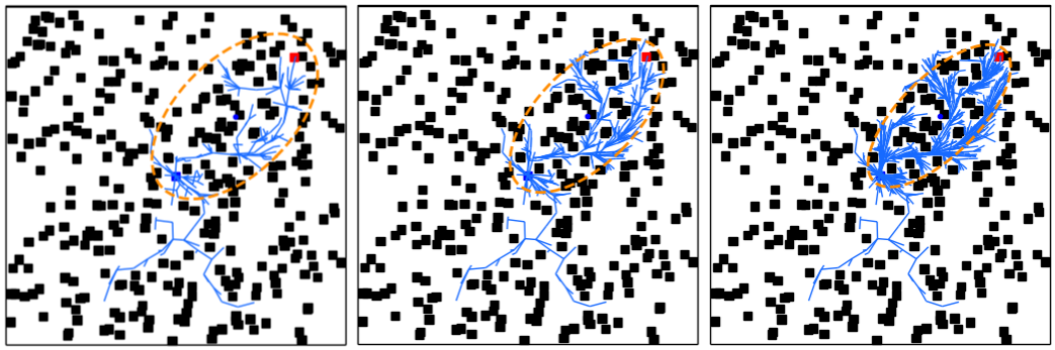
Finally we show in Figures 6(c) and 6(d) respectively the performance of the RRT* and IRRT* algorithm in the maze environment with the start and goal positioned far from each other, obtaining an euclidean distance between them large enough

to almost cover the entire environment. From these we can notice that the effectiveness of the Informed RRT* algorithm depends largely on the characteristics of the environment being mapped. The use of an ellipsoid as informed space in Informed RRT* is only beneficial if the ellipsoid does not completely cover the search space. If the ellipsoid encompasses the entire search space, the algorithm would not receive any benefit from the informed sampling, and it would be equivalent to RRT*. In fact, in the specific case showed the informed space almost covers the entire environment letting the informed RRT* achieving just a slight better cost compared to the non-informed version.



(a) RRT* and IRRT* path, c_{best}=∞ (b) RRT* path, c_{best}=11.64 (c) Informed RRT* path, c_{best}=10.86

Fig. 4: Figure (a) shows the path found by both algorithms at iteration 500, with the same number of nodes = 62 at the exact same position. Figures (b) and (c) compares the path found by both the algorithms at iteration 3000, the plain RRT* found 957 nodes with a resulting best cost equal to 11.64, meanwhile the Informed version found 1366 nodes with a resulting best cost of 10.86.



(a) it = 820, c_{best} = 12.01 (b) it = 1320, c_{best} = 11.44 (c) it = 2220, c_{best} = 10.94

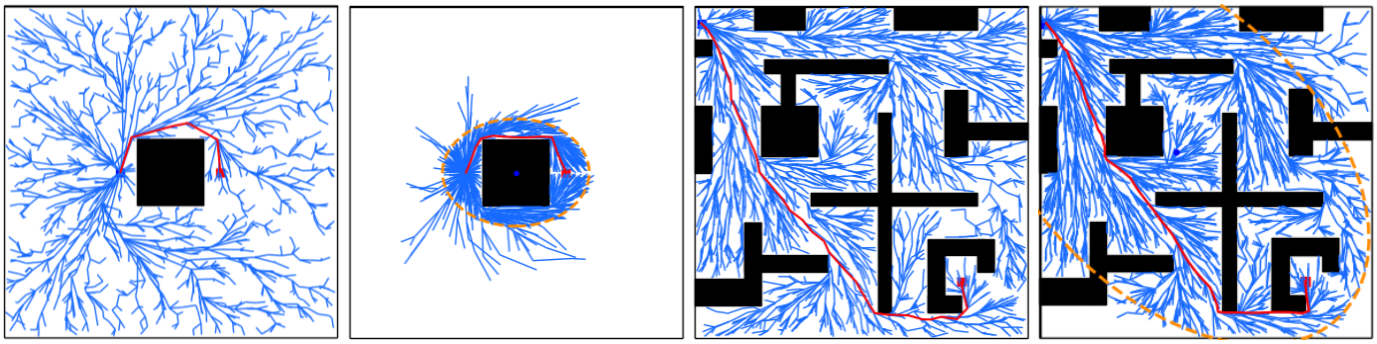
Fig. 5: This figure shows the evolution of the Informed space in euclidean domain in a random environment.

B. Comparison between RRT and Informed RRT* in Kinodynamic domain*

In the kinodynamic domain we experimented on two different fixed environments: they are showed in Figure 3(c) and 3(d). In Figures 10(b) and 10(c) are showed respectively the plots related to the environment used in [5] and the maze environment.

It is quite clear that Informed RRT* in Kinodynamic domain (IRRTK*) algorithm generally discovered better solutions compared to RRT* in Kinodynamic domain (RRTK*) in both settings. However, it should be noted that IRRTK* was comparatively slower than RRTK*. This can be attributed to the particular sampling approach that is utilized in the IRRTK* algorithm. Specifically, IRRTK* and RRTK* execute identically until the initial solution is obtained, whereupon IRRTK* transitions to a different sampling method based on Monte Carlo Markov Chain.

It is noteworthy to observe that RRTK* seems to struggle comparatively more to add nodes in the same number of iterations compared to IRRTK*. In fact, our simulations indicate that RRTK* adds only around 800 nodes in 2000 iterations, whereas its informed counterpart adds approximately 1250 nodes within the same number of iterations (referred to the Maze environment, although the analysis is similar for the other fixed environment as it's showed in Figure 10(b)). We obtain ratios

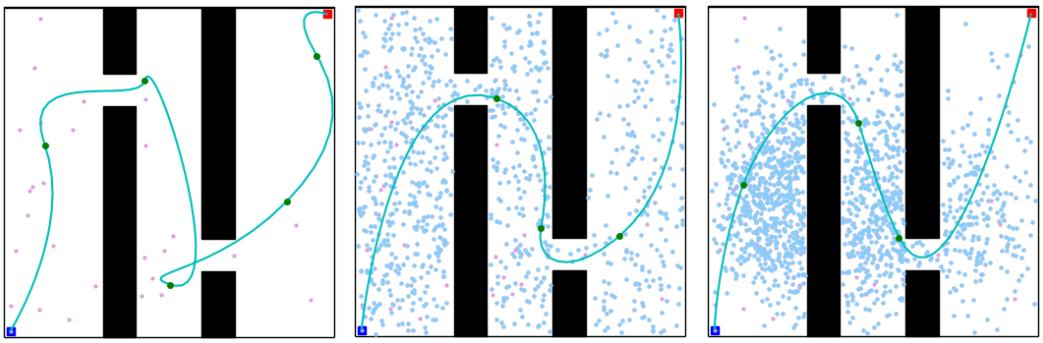


(a) RRT*, c_{best} = 48.43 (b) IRRT*, c_{best} = 44.12 (c) RRT*, c_{best} = 141.97 (d) IRRT*, c_{best} = 140.13

Fig. 6: In figure (a) and (b) the comparison on the square environment; in figure (c) and (d) the maze-like one.

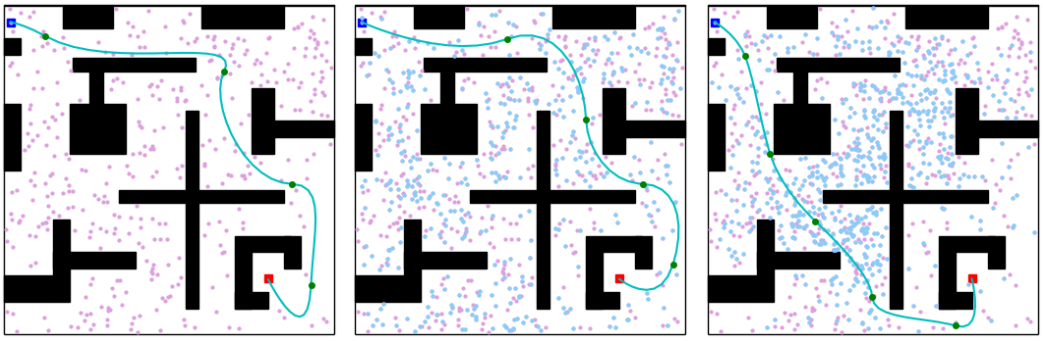
of 0.4005 and 0.5675 of nodes added per generated nodes for the uninformed and informed versions, respectively. However, the number of nodes generated by the informed method is greater than the number of iterations because, as it's showed in Algorithm 5, the MCMC sampler iteratively generates nodes until one is found to be in the informed set for each iteration. The ratio between the two reveals that the difference is less than 6%. Upon examining the cost related to the same number of nodes in the plots, it seems that the nodes added by the informed method are slightly better or comparable in quality to the ones generated by RRTK*, as the obtained cost is marginally lower. Therefore, we can say that the nodes generated by the informed method are slightly nuanced to be of better quality. Nonetheless, it must always be underlined that these results are highly dependent on the distribution given to approximate the informed set and therefore no absolute conclusions can be made in this case. An interesting argument on the MCMC method and specifically about Metropolis-Hastings will be done in the last part of the following section.

In Figures 7 and 8 it is showed IRRTK* outperforming RRTK* in the kinodynamic domain. We can see the visual difference of the sampling adopted by the two algorithms. RRTK* explores the whole environment while IRRTK* focus more on the states which led to the goal, as we can see from the samples generated in gaussian dispositions. In our simulations the informed version improved the solution more often compared to RRTK*, suggesting that in the kinodynamic domain this approach may be better in finding acceptable trajectories between states because they are directly samples from previous accepted states. The new states have their position and velocity generated from a gaussian distribution on the previous accepted samples: this leads to a higher probability of the new states being accepted at the very least given their similarity with previous accepted states. It must be noticed that while they may be acceptable nodes they are not surely going to improve the cost of the path directly. One possible approach to explore, that could increase the quality of MH-generated samples, would be to partially accept and reject samples. Specifically, by discarding only part of the sample, the remaining ones can be generated iteratively until the entire sample is acceptable.



(a) it = 158, c_{best} = 139.60, τ^* = 83.06 (b) it = 1083, c_{best} = 88.28, τ^* = 60.38 (c) it = 1595, c_{best} = 81.77, τ^* = 57.78

Fig. 7: Figure (a) shows the path found by both algorithms at iteration 158 at the exact same position. Figures (b) and (c) compares the path found by both the algorithms after 2000 iterations, the plain RRT* found the best solution with cost equal to 88.28, meanwhile the Informed version found the best solution with cost 81.77.



(a) it = 1086, c_{best} = 87.05, τ^* = 56.02 (b) it = 1784, c_{best} = 76.48, τ^* = 50.98 (c) it = 1318, c_{best} = 62.52, τ^* = 47.37

Fig. 8: Figure (a) shows the path found by both algorithms at iteration 1086 at the exact same position. Figures (b) and (c) compares the path found by both the algorithms after 2000 iterations, the plain RRT* found the best solution with cost equal to 76.48, meanwhile the Informed version found the best solution with cost 62.52.

*C. Differences between Euclidean and Kinodynamic RRTs**

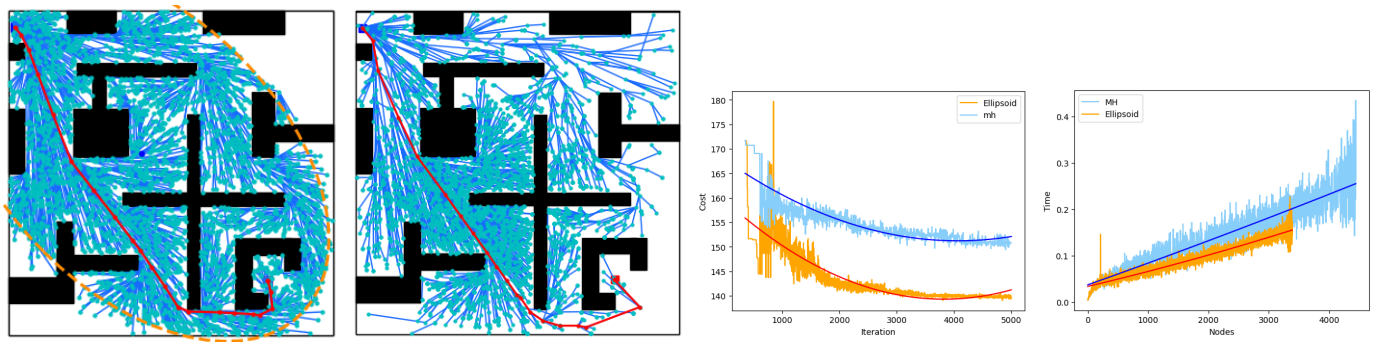
The RRT* algorithm can be used in both euclidean and kinodynamic domains, but there are significant differences in how it operates in each domain. In the euclidean domain, the RRT* algorithm focuses on connecting random points in space to produce a tree-like structure. The algorithm tries to optimize the tree by rewiring the existing branches to produce the shortest path between the start and goal points. On the other hand, in the kinodynamic domain, the RRTK* algorithm needs to take into account the motion constraints of the robot, which makes the optimization process more complex. The algorithm needs to generate valid paths that satisfy the kinodynamic constraints of the robot, such as acceleration and velocity bounds. Therefore, the main comparison between the euclidean and kinodynamic domain for the RRT* algorithm is the level of complexity. The RRTK* algorithm is usually more complex, requiring more computational resources and a better understanding of the kinematic and dynamic constraints of the robot. On the other hand, the euclidean RRT* algorithm is simpler and faster to implement but has limited application in environments where robot motion is highly constrained. In the kinodynamic domain the velocities and accelerations of the system are involved, along with their bounded values.

In Figure 10 we show the performances of the two variations of the algorithm. We can see that the Informed versions generally find a better solution and they do it faster than the non-informed ones, both comparing the number of iterations and the number of nodes (first and third column of the figure respectively). On the other hand, the computational complexity and thus the slowness of the process increases over time and proportionally to the number of nodes. This is not a critical problem, since in practice Informed RRT* generally finds a near-optimal solution before starting to heavily increase the complexity. These analysis applies mostly to the euclidean and slightly to the kinodynamic algorithms.

In Figure 9, instead, we tested the Metropolis-Hastings (MH) sampler (algorithm 5) in the euclidean space with informed RRT* and we compared it to the ellipsoidal heuristic in the maze environment. We can see that the Metropolis-Hastings sampler underperforms in all experiments when compared to the ellipsoid-based sampler. The MH algorithm generates samples that converge to a proposed distribution that, in this particular case, attempts to approximate the subset containing states that can produce higher quality paths. However, if the given distribution does not appropriately approximate this informed set, the generated samples will be of low quality.

In general, MH is most suited for high-dimensional and non-linear problems, where it remains very efficient, although the quality problem of the samples still remains and the tuning of the parameters is needed. In [3] they conduct a deeper analysis on different samplers and they show that in fact MH is the worse sampler when the quality of the generated samples is taken into consideration, while it still remains efficient when the problem is high-dimensional. Unfortunately, their analysis does not comprehend a comparison between IRRTK* using MH and RRTK*. Based on our experiments there seems to generally be a really small difference in ability to find a suitable solution between the two, with IRRTK* taking much more resources: if we focus on Figures 10(b) and 10(c) we can see that the difference in cost when more nodes are added is not really that different between the two algorithms, suggesting that the quality of nodes is mostly similar. Finally, we would argue that for some environments choosing the informed version of RRTK* with MH might not be the obvious solution, given the efforts to be made in order to tune the parameters and the resources used by algorithm. Nonetheless, we stated multiple times that

a more suitable proposed distribution could highly improve the performance of the algorithm: a deeper analysis on this topic could be made in future research.

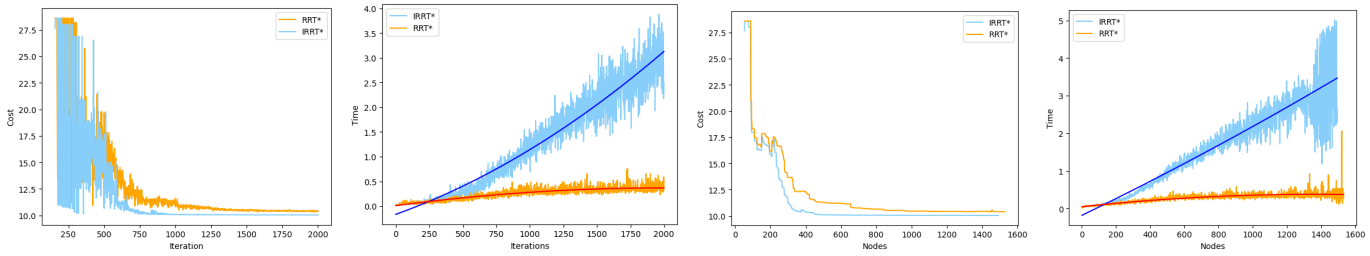


(a) it = 4140, c_{best} = 139.39 (b) it = 4480, c_{best} = 151.12 (c) Cost trend wrt iteration (d) Time trend wrt number of nodes

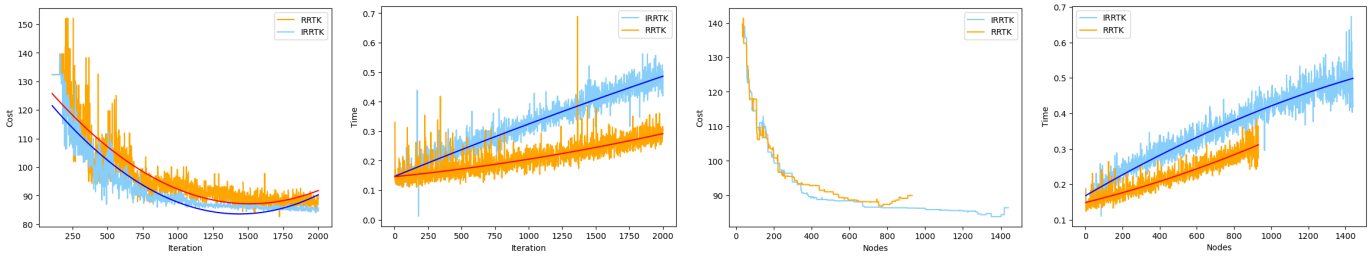
Fig. 9: In the figures Informed RRT* on euclidean domain is performed on the maze and random environment using different heuristics. In Figure (a) the ellipsoidal heuristic has been used, while in Figure (b) the MCMC heuristic has been used. In Figure (c) and (d) the cost-iteration and time-nodes plots are showed, respectively

V. Conclusions

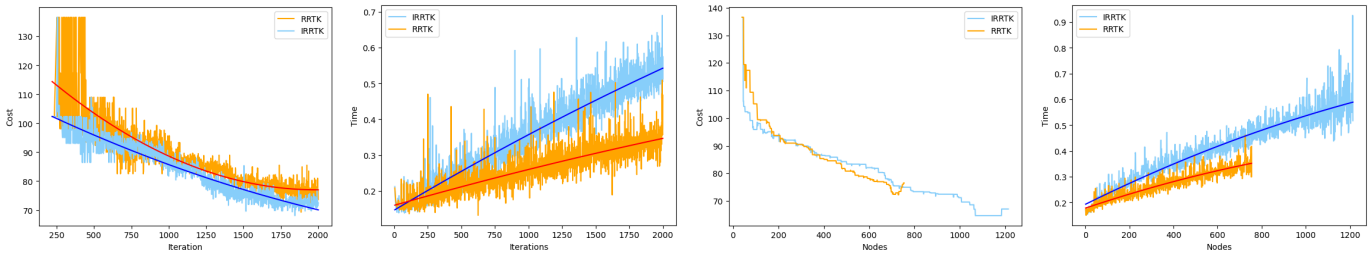
In conclusion, our report compared the performance of RRT* and Informed RRT* algorithms in euclidean and kinodynamic domains. The kinodynamic RRT* algorithm is suited for robotic systems that have dynamic constraints such as acceleration and velocity bounds. It is specifically designed to plan paths that are both feasible and optimal, taking into account the dynamics of the system. These features made the kinodynamic algorithm computationally heavier, and as a result slower than the euclidean counterpart. On the other hand, the euclidean RRT* algorithm can be used for systems that do not have any dynamic constraints and can be modeled as points in space. We found out that, in general, the Informed RRT* outperforms the base version in terms of optimality and convergence speed. We also noticed that the heuristic of the informed algorithms affects the performance as the environment and dimensionality of the problem change. Indeed we tried the MCMC-based heuristic also in euclidean domain. In our experiments this heuristic performed worse than the ellipsoidal heuristic in all the environments. We even tried to tune the parameters specifically for each environment but, even though it achieved slightly better results than the previous ones, it still didn't catch up to the other heuristic. As we stated before this is not enough to confirm an absolute best heuristic but it gives an empirical view on how the two methods behave.



(a) Euclidean domain in Random Environment



(b) Kinodynamic domain in Fixed Environment



(c) Kinodynamic domain in Maze Environment

Fig. 10: First row contains the plots obtained from the simulations of the algorithms in euclidean space using the randomly generated environments. Third and the last row contains the plots of the kinodynamic simulations on the kinodynamic and the maze environment respectively. In the first column the path cost with respect to the iterations is shown, in the second one, instead, the time needed for each iteration is shown. In the third column the cost as the number of nodes vary, and in the last one the trend of the time with respect to the number of nodes is shown. In all the images the blue plots represent the informed version of the algorithms, while the orange plots represent the base ones.

REFERENCES

[1] Karaman, Sertac & Frazzoli, Emilio. (2011). "Sampling-based Algorithms for Optimal Motion Planning." International Journal of Robotic Research - IJRR. 30. 846-894. 10.1177/0278364911406761.

[2] Gammell, Jonathan D.; Srinivasa, Siddhartha S.; Barfoot, Timothy D. "Informed RRT: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic." In: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, 2014. p. 2997-3004.

[3] Yi, Daqing, et al. "Generalizing informed sampling for asymptotically-optimal sampling-based kinodynamic planning via markov chain monte carlo." In: 2018 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2018. p. 7063-7070.

[4] Karaman, Sertac; Frazzoli, Emilio. "Optimal kinodynamic motion planning using incremental sampling-based methods." In: 49th IEEE conference on decision and control (CDC). IEEE, 2010. p. 7681-7687.

[5] D. J. Webb and J. van den Berg, "Kinodynamic RRT*: Asymptotically optimal motion planning for robots with linear dynamics," 2013 IEEE International Conference on Robotics and Automation, Karlsruhe, Germany, 2013, pp. 5054-5061, doi: 10.1109/ICRA.2013.6631299.

[6] Meurer, Aaron, et al. "SymPy: symbolic computing in Python." PeerJ Computer Science 3 (2017): e103.

[7] Karaman, Sertac, and Emilio Frazzoli. "Sampling-based algorithms for optimal motion planning." The international journal of robotics research 30.7 (2011): 846-894.

[8] Karaman, Sertac, and Emilio Frazzoli. "Incremental sampling-based algorithms for optimal motion planning." Robotics Science and Systems VI 104.2 (2010).

[9] LaValle, Steven M. "Rapidly-exploring random trees: A new tool for path planning." (1998): 98-11.

[10] Robert, Christian, et al. "Metropolis–hastings algorithms." Introducing Monte Carlo Methods with R (2010): 167-197.