Comparing different Rapidly-exploring Random Tree (RRT) versions

Santilli Sofia
id number: 1813509
e-mail: santilli.1813509@studenti.uniroma1.it

Abstract

The sampling based motion planning algorithm known as Rapidly-exploring Random Trees (RRT) was proposed for the first time in 1998 by Steven M. LaValle [13]. It is a rondomized data structure designed for a wide class of path planning problems. It efficiently searches nonconvex, high-dimensional spaces by incrementally building a space-filling tree. The tree is constructed from samples drawn randomly from the search space and is inherently biased to grow towards large unsearched areas of the problem. They easily handle problems with obstacles and differential constraints. Also, they work well in practice, showing computational efficiency and probabilistic completeness. Because of all these properties, the different versions of RRTs have been extensively exploited in autonomous motion planning.
The paper first introduces an overview of the methods applied in the last decades in order to solve path planning problems, explaining in details why RRTs gained the attention of many researchers. Then different versions of the Rapidly-exploring Random Tree algorithms are presented and their functioning is explained also through pseudocode. Their properties are highlighted and comparisons between them are done.

1 Introduction

Path planning is the process used to find a feasible continuous path from a starting point q_i to an end point q_f given a full, partial or dynamic map; this will result in a path $q(s)$, with $s \in [s_i, s_f]$, such that $q(s_i) = q_i$; $q(s_f) = q_f$.
Motion planning is the computational problem to find a sequence of valid configurations that moves the agent on the path planned, from source to destination, while avoiding collisions with any static obstacle or other agents present in the environment. Motion planning has several robotics applications, such as autonomy, automation and robot design in CAD software, as well as other applications in other fields, such as animating digital characters, video games, architectural design, robotic and computer-aided surgery, the study of biological molecules, assembly maintenance, manufacturing and other aspects of daily life.

2 State of the art

Motion planning has been a high-interest-area of research since the late 1970s. In 1983, Schwartz and Sharir introduced the *piano movers' problem* [19], consisting in moving a piano in a cluttered home from a start to a goal without bumping into obstacles. It represented a formalism for robot motion planning, spawning generations of research on graph search, trajectory optimization and randomized algorithms.

2.1 Resolution complete algorithms

The earliest approaches to the problem focused mainly on the development of *complete planners*: algorithms that converge, in a finite time, either to a path solution, if one exists, or return failure otherwise.
But in 1979 was presented the *mover's problem* [16] which concerned the problem of moving a polyhedron through Euclidean space while avoiding polyhedral obstacles. Although it was a basic version of the motion planning problem, it was recognized to be P-space-hard: the first known P-space complete computational geometry problem and one of very few known P-space complete combinatorial problems, non trivial to solve with a polynomial time limit. This proved that *complete motion planning algorithms* suffer from computational complexity in most of the practical motion planning problems. For this reason, this type of algorithms are mainly exploited in low-dimensional problems.
Most motion planning methods that are based on gridding or cell decomposition fall into this category and require fine tuning of resolution parameters. In order to be precise, because of the discretizations of the Configuration space, they are not complete, but *resolution complete algorithms*, that is the property that guarantees the planner to find a path, if one exists, at the level of the discretization chosen. The computational complexity of resolution

complete planners is dependent on the number of points in the underlying grid: $O(1/h^d)$, where h is the resolution (the length of one side of a grid cell) and d is the configuration space dimension.
Once the free space is represented as a graph, a motion planning plan can be found by searching the graph through some methods, such as A*, Dijkstra algorithm, breadth-first search.

2.1.1 Grid-based search

Grid-based approaches [23] overlay a regular grid on the configuration space, such that each grid point identifies a configuration from which the robot is allowed to move to adjacent grid points as long as no obstacles lie on the line between them. In this way, the set of actions is discretized and the space can be searched using standard graph search methods (e.g., A*, which makes the solution path optimal). In order to increase the efficiency, multi-resolution grids can be employed: coarser grids allow faster searches, but fail in finding paths through narrow passages; a dense grid has more probability to find a feasible path, but it takes more time and since the number of points in the grid grows exponentially in the configuration space dimension, a denser grid will be inappropriate for high-dimensional problems.

2.1.2 Interval-based search

These approaches generate a paving covering entirely the configuration space Q and that is decomposed into two subpavings X^-, X^+, made of boxes such that $X^- \subset Q_{free} \subset X^+$. Characterizing Q_{free} amounts to solve a set inversion problem. Interval analysis could thus be used when Q_{free} cannot be described by linear inequalities in order to have a guaranteed enclosure.
The robot is thus allowed to move freely in X^-, and cannot go outside X^+. To both subpavings, a neighbor graph is built and paths can be found using algorithms such as Dijkstra or A*. When a path is feasible in X^-, it is also feasible in Q_{free}. When no path exists in X^+, from one initial configuration to the goal, we have the guarantee that no feasible path exists in Q_{free}. As for the grid-based approach, the interval approach is inappropriate for high-dimensional problems, due to the fact that the number of boxes to be generated grows exponentially with the dimension of the configuration space.

2.1.3 Geometric algorithms

Geometric algorithms [18] compute the shape and connectivity of Q_{free}. A *Visibility graph* [5] is a graph in which each node represents a point location and each edge represents a visible connection between them, not interrupted by any obstacle. The shortest path between two obstacles follows straight line segments except at the vertices of the obstacles, where it may turn. Therefore visibility graphs may be used to find Euclidean shortest paths among a set of polygonal obstacles in the plane: first the visibility graph is built, then a shortest path algorithm (e.g., Dijkstra's algorithm) is applied to the graph.
Cell decomposition (BCD) [12] is another example of geometric algorithm: a method that decomposes the configuration space into cells, marked as blocked if they intersect any obstacle. So, the regions not blocked are such that it is easy to compute a safe path between two configurations in the same cell and between two configurations in adjacent cells. Different types of cell decomposition exist. Once the decomposition has been computed, a sequence of cells has to be found such that q_{ini} is in the first cell and q_{goal} is in the last cell.

2.2 Artificial Potential Fields algorithms

Another approach is to treat the robot's configuration as a point, moving through the potential fields constructed in the configuration space Q [10]. The robot is attracted by the goal and repelled by the obstacle regions. The attractive potential can be represented as paraboloidal or conical, with its minimum in the goal configuration q_g. A convenient solution also consists in combining this two profiles, conical away from q_g and paraboloidal close to q_g. Instead, a repulsive field is defined for each obstacle. The total potential U is the sum of an attractive and a repulsive potential: the robot will move in the most promising local direction of motion, computed as the negative gradient $-\nabla U(q)$. Potential-field algorithms are efficient in high-dimensional spaces, but they suffer from the problem of local minima [11], not finding a path or finding a non-optimal one, and don't perform well in environments with narrow passages.

2.3 Sampling-based methods

A more recent line of research has achieved a great success focusing on the construction of paths connecting randomly-sampled points [3]. Sampling-based methods provide large amounts of computational savings by avoiding explicit construction of the obstacle configuration space, as opposed to most complete motion planning algorithms. Due to the NP-hardness of the Motion Planning problem, these methods have outperformed the classic approaches. Several of these algorithms have been shown to be *probabilistically complete*: as the number of samples approaches infinity, the probability of the planner to find a solution, if one exists, tends to one. The performance of a probabilistically complete planner is measured by the rate of convergence. Unlike Potential fields algorithms, sampling-based ones avoid the problem of local minima. They also return a feasible solution quite quickly, even in high-dimensional spaces. Although, they are unable to determine that no path exists but, as the number of samples increases, their probability of failure decreases around zero. Therefore, sampling-based algorithms have been successfully applied to problems with dozens or even hundreds of dimensions.

The most remarkable sampling-based algorithms are *Probabilistic RoadMaps* (PRM) and *Rapidly-exploring Random Trees* (RRT) [7]. Both are designed with as few heuristics and arbitrary parameters as possible. Also they are relatively simple algorithms, thus facilitating performance analysis. The PRM [9] constructs a graph of feasible paths offline. At each iteration it extracts a sample q from Q with uniform probability distribution; if $q \in Q_{free}$ then q is added to the PRM and, exploiting an arbitrary metric, a configuration $q_{near} \in PRM$ sufficiently near to q is searched. If possible, q and q_{near} are connected through a free local path. Therefore this algorithm generates a certain number of disconnected components (local paths) that, as iterations increase, will tend to join. When the number of disconnected components is below a certain threshold, or the maximum number of iterations was reached, the construction of the map terminates and graph search is used in order to find a path connecting initial and goal positions: a solution will be found only if the two configurations belong to the same component. The main advantage of PRM is the speed. Although, narrow passages are critical and the method tends to be inefficient when obstacle geometry is not known beforehand. On the other hand, RRT present a solution regardless of whether the specific geometry of the obstacles is known beforehand or not. In the last years this algorithm has been extensively explored and different versions have been developed, leading RRT to be currently considered the state-of-the-art for motion planning in high-dimensional configuration spaces.

In this paper Section 3 is dedicated to a formalization of the motion planning problem; in Section 4 the original RRT version is presented, describing the algorithm and the main aspects. It follows descriptions of the other versions of the algorithms and the main differences between them are highlighted.

3 Problem statement

Being $Q \in \mathbb{R}^d$ and $U \in \mathbb{R}^m$ respectively the state space and the input space, we denote the goal region as $Q_{goal} \subset Q$, the obstacle region as $Q_{obs} \subset Q$ and the obstacle-free space as $Q_{free} = Q \backslash Q_{obs} \subset Q$. An explicit representation of the obstacle region is not available: one can only check whether a given configuration of the agent lies in Q_{obs} or not[1]. States in Q_{obs} may correspond to velocity bounds, configurations at which a robot is in collision with an obstacle in the world, or several other interpretations, depending on the application. The motion planning problem consists in determining a control input $\mathbf{u} : [0, T] \rightarrow U$ that yields a feasible and continuous path $\mathbf{q}(t) \in Q_{free}$ for $t \in [0, T]$ from an initial state $\mathbf{q(0)} = \mathbf{q}_{ini}$ to a goal region $\mathbf{q}(T) \in Q_{goal}$ or to a goal state q_{goal}, where $\mathbf{q}_{ini}$, Q_{goal} and q_{goal} belong to Q_{free}. The state space encodes both configuration and velocity. Therefore here it is used in order to exploit a greater generality: in fact standard path planning problems usually consider only the configuration space Q of a rigid body or of a system of bodies in a 2D or a 3D world. Differently, considering also velocities allows to deal with kinodynamic motion planning problems, in which the dynamics of the system must be taken into account in order to generate feasible trajectories. Finally, let $c : \sum_{X_{free}} \rightarrow R_{>0}$ be a cost function that assigns a non-negative cost to all nontrivial collision-free paths.

4 Rapidly-exploring Random Trees

Rapidly-exploring Random Trees have been introduced in the late '90s. A RRT consists in an incremental and sampling-based algorithm designed to efficiently search nonconvex, high-dimensional spaces with obstacles by building a space-filling tree, constructed incrementally from samples drawn randomly from the search space. Therefore all the tree's vertices are states in Q_{free} and each edge will correspond to a path lying entirely in Q_{free}. This method shares many of the beneficial properties of existing randomized planning techniques, first of all the characteristic of working well in high-dimensional configuration spaces Q, since their running time is not exponentially dependent on the dimension of Q. Also, it shares the use of a small numbers of parameters and heuristics, that allow ease of analysis and still a consistent behaviour. At the same time, one of the main advantages with respect to most of the previous techniques, is the ability to handle differential constraints, nonlinear dynamics and non-holonomic constraints, thus naturally extending this algorithm to kinodynamic planning problems, an extremely general and important area in robotics, virtual prototyping and many other applications. These qualities make RRTs a widely used approach for solving a broad class of path planning problems. It is

[1]Often, it is prohibitively difficult to explicitly compute the shape of Q_{free}. However, testing whether a given configuration is in Q_{free} is efficient: first, forward kinematics determine the position of the robot's geometry, then collision detection tests if the robot's geometry collides with the environment's geometry.

also due to the fact that RRT can be considered a path planning module that can be incorporated and adapted into different planning systems.

4.1 Classic RRT

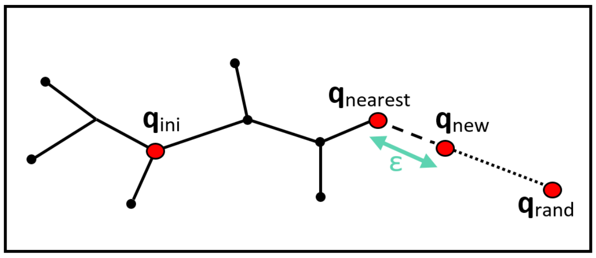
The classical version of RRTs has been proposed by Steven M. LaValle [13] in 1998. In order to explain how the algorithm works, the pseudocode and what each function does are explained. By $\mathcal{T} = (V, E)$ the tree built by RRT is denoted, with V being the set of configurations belonging to Q_{free} and $E \subseteq V \times V$ being the set of connected edges between the elements of V.

Algorithm 1: RRT

```
Data: $\mathbf{q_{ini}}, \mathbf{q_{goal}}$
Result: $\mathcal{T}$
$\mathcal{T} \leftarrow InitializeTree()$;
$\mathcal{T} \leftarrow AddNewNode(\emptyset, \mathbf{q_{ini}}, \mathcal{T})$;
for $i$ = 1 to $i$ = $N$ do
    $\mathbf{q_{rand}} \leftarrow SampleRand()$;
    $\mathbf{q_{nearest}} \leftarrow NearestNeighbour(\mathbf{q_{rand}}, \mathcal{T})$;
    $(\mathbf{x_{new}}, \mathbf{u_{new}}, T_{new}) \leftarrow Steer(\mathbf{q_{nearest}}, \mathbf{q_{rand}})$;
    $\mathbf{q_{new}} \leftarrow LastElement(\mathbf{x_{new}})$;
    if $ObstacleFree(\boldsymbol{x_{new}})$ then
        $\mathcal{T} \leftarrow AddNewNode(\mathbf{q_{nearest}}, \mathbf{q_{new}}, \mathbf{u})$;
    end
end
```

First, the tree $\mathcal{T}$ is initialized.At the beginning, the set of vertices V only contains $q_{ini} \in Q_{free}$, the initial state of the agent and root of the tree, while the set of edges E is empty. At each iteration:

- **SampleRand**: randomly samples, according to a uniform distribution, a state q_{rand} from Q_{free}.
- **NearestNeighbour**: given the newly randomly sampled node, it computes the nearest node in the tree according to an arbitrary metric ρ, that is usually chosen as a distance function (a possibility is the Euclidean distance).
- **Steer**: given two configurations $\mathbf{q_{nearest}}$ and $\mathbf{q_{rand}}$, returns a new node $\mathbf{q_{new}}$ that lies on the line connecting the input nodes, and taken at a distance smaller or equal to an arbitrary ϵ from $\mathbf{q_{nearest}}$. This step is exemplified through Figure 1. It then finds the path $\mathbf{x_{new}} : [0, T] \leftarrow Q$ connecting the two configurations, such that $q(0) = \mathbf{q_{nearest}}$ and $q(T) = \mathbf{q_{new}}$, thus returning $\mathbf{x_{new}}$, the control inputs u_{new} needed to generate $\mathbf{x_{new}}$, and the duration T of the path. The last element of $\mathbf{x_{new}}$ is $\mathbf{q_{new}}$.



$\mathbf{q}_{\text{ini}}$
$\mathbf{q}_{\text{nearest}}$
$\mathbf{q}_{\text{new}}$
ε
$\mathbf{q}_{\text{rand}}$

Figure 1: Exemplification of the computation of $\mathbf{q_{new}}$, during the steering step.

- **AddNewNode**: it is applied in the case the path just found is collision-free. Given the tree $\mathcal{T} = (V, E)$, the node $\mathbf{q_{nearest}} \in V$ and the new node $\mathbf{q_{new}}$, it adds $\mathbf{q_{new}}$ to V and adds the edge between the two nodes in E, meaning that $\mathbf{q_{nearest}}$ is the parent of $\mathbf{q_{new}}$.

The RRT algorithm terminates when the number N of iterations is reached (as implemented in Algorithm 1), when the cardinality[2] of $\mathcal{T}$ reaches an arbitrary number or also when the new node added to the tree is q_{goal}. A simplification can also be implemented, considering a goal region centered in q_{goal} and with a radius d_t (where d_t is an arbitrary and enough small threshold): in fact evaluating when the newly generated node coincides exactly with q_{goal} would take significantly much more iterations. However, even if the algorithm does not terminate when the goal is found, it converges towards a suboptimal solution, since the generated tree does not improve the solution found, but simply keeps adding new nodes and so new paths in the space. Despite the convergence to an optimal solution is not insured, RRT is probabilistically complete: in fact the probability that a solution is found approaches one as the number of vertices added to the tree tends to infinity.
RRTs are not random walkers[3], but are inherently biased to grow towards large unexplored areas of the problem. In fact, the probability of expanding an existing state is proportional to the size of its Voronoi region [4]. As the largest Voronoi regions belong to the states on the frontier of the search, this means that the tree preferentially expands towards large empty spaces. This behaviour is promoted by the nearest neighbour procedure, which will return the nearest node to the (uniformly) sampled point, thus resulting in a greater probability that the sampled point

[2]Cardinality is the number of nodes in the tree, at a certain iteration.

[3]an algorithm that performs random walk over Q_{free}, selecting a vertex at random in the whole space, thus resulting in a strong bias towards places already visited

[4]A Voronoi diagram is a partition of a plane, characterized by different objects, into regions. In the simplest case the objects are just points, called seeds. Each seed is characterised by a corresponding region, called a Voronoi region, consisting of all the points of the plane closer to that seed than to any other.

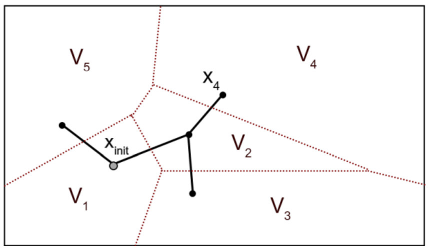


Figure 2: Example of the Voronoi diagram of a tree.

will fall in the largest Voronoi region (V_4 in the example shown in Figure 2) rather than in the other regions. In order to have this *Voronoi bias*, also the fact that the new nodes added to the tree can be at a maximum distance of ϵ help, since they do not allow distant and scattered vertices distributed in the space with no order. Generally the RRT algorithm does not explicitly build the Voronoi diagram of the search space. Although, some implementations do that at every iteration, thus allowing to easily decide which portion of the search space to explore next. Taking into account Figure 3, where a run of the RRT algorithm is shown, it is easy to notice how the RRT expands in a few directions to quickly explore the four corners of the space.

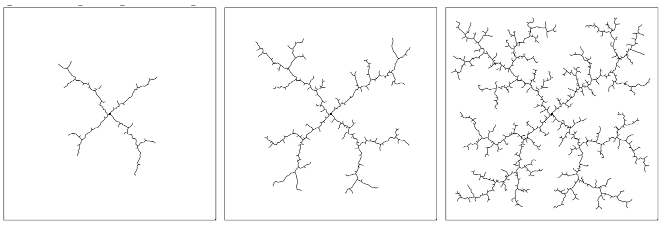


Figure 3: Run of the RRT algorithm.
State space $Q = [0, 100]x[0, 100]$, $\mathbf{q_{init}} = (50, 50)$.

This leads to a distribution of the vertices that converges to the uniform sampling distribution[5]. Uniformity was repeatedly confirmed by the passing of several Chi-square tests, tipically used to evaluate random number generators.
With respect to the basic probabilistic roadmap approach, RRT might be faster and is minimal since it is always able to mantain a connected structure with the fewest edges[6]. Also, single nearest neighbours queries are used by RRTs, while PRMs require more espensive k-nearest neighbour queries. Despite these advantages, an issue in RRTs, as in other randomized path planning methods, consists in the difficulty of defining an ideal metric ρ to obtain the shortest path between two states. The issue just described, together with the increasing cost in order to find the nearest neighbours when the number of nodes in the tree becomes high, are problems that concern all the versions of the RRT algorithm. On the contrary, other disadvantages, such as suboptimality and the difficulty encountered in planning problems such as the ones in Figure 4 have been solved through other versions of the algorithm proposed during the years.

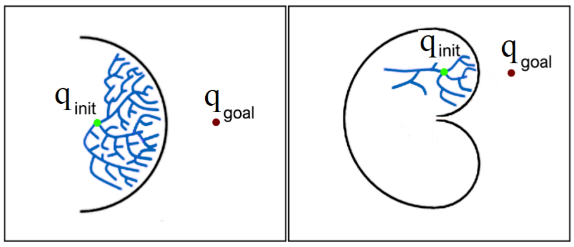


Figure 4: Scenarios in which the classic version of RRT has difficulties in founding a solution path. On the right, the so called "bug trap".

Afterwards, when writing "RRT" we will not be referring to all the family of Rapidly-exploring Random trees algorithm, but to this classical version.

4.2 Bidirectional RRT

The previous versions of the algorithm did not present any bias towards the goal, which is useful to improve the convergence rate towards the goal area. Therefore, in order to do that, one may grow two trees at the same time, T_s and T_g, with root node respectively at the specified start and goal configurations $\mathbf{q_{in}}$ and $\mathbf{q_{goal}}$. At each iteration, one tree will add $\mathbf{q_{new}}$ while the other will try to grow towards $\mathbf{q_{new}}$, thus connecting the two trees whenever no obstacle limits their connection. This method is probabilistically complete, single-query and, with respect to the classical version, bidirectional search showed to be more efficient, especially in cases such as three-dimensional planning problems or the "bag trap". A significant example, in fact, is given by the Alpha 1.0 puzzle[7], cited as one of the most challenging motion planning examples due to the very narrow passages in the 6-dof configuration space. The goal is to separate the two bars from each other, as shown in Figure 5 . In 2001, it was solved by using a balanced bidirectional RRT, taking only few minutes to solve. Until now only two other algorithms have been able to solve this example[8].

[5]The vertices of an RRT will distribute accordingly to the probability density function used for sampling $\mathbf{q_{rand}}$

[6]Since many extra edges can be generated trying to form connected roadmaps, PRM may suffer in performance

[7]constructed by Boris Yamrom, GE Corporate Research Development Center, and posted as a research benchmark by Nancy Amato at Texas AM University.

[8]One is also based on RRTs and developed by Kineo, a French company that develops motion planning software; the other is by Pekka Isto, who was a Ph.D. student from the Helsinki University of

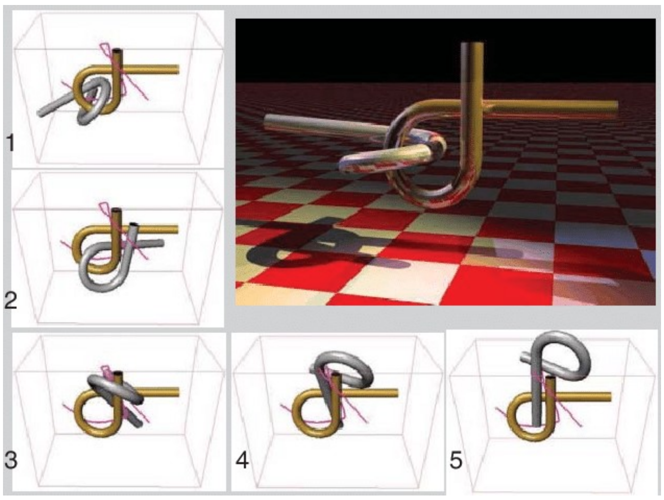


Figure 5: Alpha 1.0 puzzle solution steps

4.3 Other biased RRTs

An alternative to bidirectional RRT was presented by Urmson and Simmons [21] to guide the search towards the planning problem goal through a heuristic quality function incorporating a preference for shorter paths. Another popular approach consists in balancing exploration and exploitation. Exploration is the basic behaviour: $\mathbf{q_{rand}}$ is randomly sampled from Q through the uniform probability distribution. Exploitation is a greedy behaviour, for which $\mathbf{q_{rand}}$ is taken equal to $\mathbf{q_{goal}}$. Different methods can be applied in order to decide, at each iteration, if exploration or exploitation will be performed. The simplest one consists in introducing a constant $\gamma \in [0, 1]$. At each iteration a number $r \in [0, 1]$ is randomly generated: if $r < \gamma$, $\mathbf{q_{rand}}$ is sampled through exploration, otherwise through exploitation. The higher this probability is, the more greedily the tree grows towards the goal. Another alternative is *RRT-Connect*, that biases the tree towards the randomly sampled node, keeping adding nodes to the tree between $\mathbf{q_{nearest}}$ and $\mathbf{q_{rand}}$, until $\mathbf{q_{rand}}$ is reached or an obstacle occurs, as shown in Figure 6.

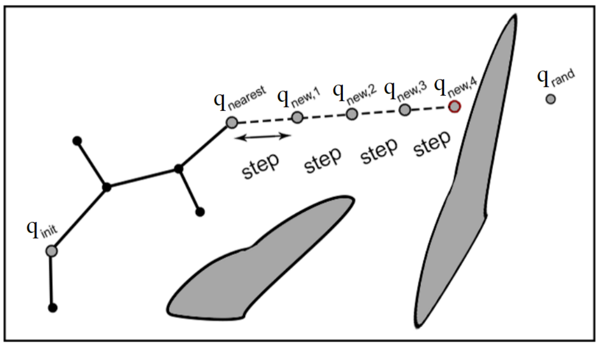


Figure 6: Bias towards $\mathbf{q_{rand}}$.

Technology in Finland.

4.4 RRT$^+$

Since solving high dimensional planning problems in near real-time remains a considerable challenge, Xanthidis et al. [22] recently introduced an enhancement to traditional sampling-based planners, resulting in efficiency increases for high-dimensional holonomic systems such as hyper-redundant manipulators, snake-like robots, and humanoids.

Their novelty consists in the *Sampling* procedure. In fact, instead of searching directly the whole configuration space Q, the algorithm starts exploring lower dimensional volumes of the configuration space. That is, first it optimistically searches for an admissible path in the unique linear 1-dimensional subspace of Q that contains both $\mathbf{q_{init}}$ and $\mathbf{q_{goal}}$ (*(a)* in Figure 7). If this search fails, the planner expands its search to a planar subspace on which $\mathbf{q_{init}}$ and $\mathbf{q_{goal}}$ lies *(b)*, then to a 3D flat *(c)* and so on. This process continues iteratively until the planner finds a path, or until it searches in all the configuration space. In each subsearch, the tree structure created in lower dimensions is kept and expanded in subsequent stages.

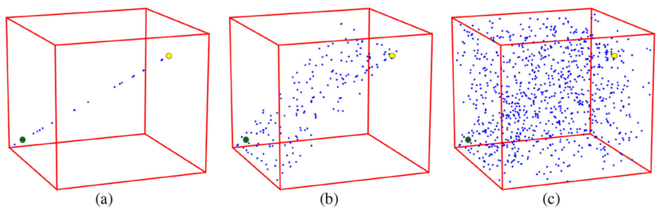


Figure 7: Sampling in one, two and three dimensions of Q. Red lines indicate the boundaries of Q, the yellow dots indicate $\mathbf{q_{init}}$ and the green dots indicate $\mathbf{q_{goal}}$.

4.5 RRT*

Although the variants to RRT presented above increase the convergence rate and improve the growth of the tree, they keep the suboptimal property of RRT. In fact the classical version, biased or not, does not take into account the solution path cost c, defined in Section 3. However in many real-time applications, it is highly desirable to find a feasible solution quickly and improve the quality of the solution in the remaining time. Therefore Karaman and Frazzoli [6] presented RRT*, a slightly modified implementation of the classical RRT algorithm, developed as a variant of the Rapidly-exploring Random Graphs[9] [8]

[9]It is an incremental sampling-based method that does not only generate feasible obstacle-free trajectories towards a goal, but can also handle any specification given in the form of μ-calculus, which includes the widely-used specification language Linear Temporal Logic (LTL) as a strict subclass. The RRG algorithm builds a graph, instead of a tree, since several specifications in LTL require cyclic trajectories, which are not included in trees.

algorithm that inherits the asymptotic optimality of the RRG algorithm while maintaining a tree structure.

Algorithm 2: RRT*

```
Data: $\mathbf{q_{ini}}, \mathbf{q_{goal}}$
Result: $\mathcal{T}$
$\mathcal{T} \leftarrow InitializeTree()$;
$\mathcal{T} \leftarrow AddNewNode(\emptyset, \mathbf{q_{ini}}, \mathcal{T})$;
for $i$ = $1$ to $i$ = $N$ do
    $\mathbf{q_{rand}} \leftarrow SampleRand()$;
    $\mathbf{q_{nearest}} \leftarrow NearestNeighbours(\mathbf{q_{rand}}, \mathcal{T})$;
    $(\mathbf{x_{new}}, \mathbf{u_{new}}, T_{new}) \leftarrow Steer(\mathbf{q_{nearest}}, \mathbf{q_{rand}})$;
    $\mathbf{q_{new}} \leftarrow LastElement(\mathbf{x_{new}})$;
    if $ObstacleFree(\boldsymbol{x_{new}})$ then
        $Q_{near} \leftarrow NearByVertices(\mathbf{q_{near}}, \mathcal{T})$;
        $\mathbf{q_{min}} \leftarrow$
          $ChooseParent(Q_{near}, \mathbf{q_{nearest}}, \mathbf{q_{rand}}, \mathbf{x_{new}})$;

        $\mathcal{T} \leftarrow AddNewNode(\mathbf{q_{min}}, \mathbf{q_{new}}, \mathbf{u_{new}})$;
        $\mathcal{T} \leftarrow Rewire(\mathcal{T}, Q_{near}, \mathbf{q_{min}}, \mathbf{q_{new}})$;
    end
end
```

RRT* generates its tree in a similar way as RRT: at each iteration, $\mathbf{q_{rand}}$, $\mathbf{q_{nearest}}$ and $\mathbf{q_{new}}$ are assigned in the same manner it has been discussed in Section 4.1. The algorithm is different inside the condition with $ObstacleFree(\mathbf{x_{new}})$, checking that the path found from $\mathbf{q_{nearest}}$ to $\mathbf{q_{new}}$ is collision-free.

- **NearByVertices**: given the node $\mathbf{q_{new}}$ that is going to be added to $\mathcal{T}$ and the cardinality n of $\mathcal{T}$, this function returns the nodes in the tree that are the nearest to $\mathbf{q_{new}}$. In particular, in the case of non-Euclidean cost metrics, being $c(e_{\mathbf{q_{rand}},\mathbf{q}})$ the cost for going from $\mathbf{q_{rand}}$ to $\mathbf{q}$, this function returns all the nodes which belong to $Q_{reach} = \{\mathbf{q} \in Q : c(e_{\mathbf{q_{rand}},\mathbf{q}}) \leq l(n) \ \vee \ c(e_{\mathbf{q},\mathbf{q_{rand}}}) \leq l(n)\}$, that are the nodes $\mathbf{q}$ than can reach and can be reached from $\mathbf{q_{rand}}$ with a cost that is lower than $l(n)$, which is a threshold that decreases over iterations as $l(n) = \gamma_l((\log n)/n)$.

- **ChooseParent**: given the newly sampled node $\mathbf{q_{new}}$ and the set of near nodes computed by *NearByVertices*, it returns the parent of $\mathbf{q_{new}}$. Rather than simply choosing the nearest node to $\mathbf{q_{new}}$, it evaluates the cost of picking each node in the set as its parent and chooses the one with the lowest cost from the root.

- **Rewire**: checks each node $\mathbf{q_{near}}$ belonging to the set computed by *NearByVertices*, in order to see whether reaching $\mathbf{q_{near}}$ via $\mathbf{q_{new}}$ would achieve lower cost than via its current parent. When this connection reduces the total cost associated to $\mathbf{q_{near}}$, the algorithm assigns $\mathbf{q_{new}}$ as parent of $\mathbf{q_{near}}$.

Algorithm 3: ChooseParent

```
Data: $Q_{near}, \mathbf{q_{nearest}}, \mathbf{q_{new}}, \mathbf{x_{new}}$
Result: $\mathbf{q_{min}}$
$\mathbf{q_{min}} \leftarrow \mathbf{q_{nearest}}$;
$c_{min} \leftarrow Cost(\mathbf{q_{nearest}}) + c(\mathbf{x_{new}})$;
for $\boldsymbol{q_{near}} \in Q_{near}$ do
    $(\mathbf{x'}, \mathbf{u'}, T') \leftarrow Steer(\mathbf{q_{near}}, \mathbf{q_{new}})$;
    if $ObstacleFree(\boldsymbol{x'})$ and $\boldsymbol{x'}(T') = \boldsymbol{q_{new}}$ then
        $c' = Cost(\mathbf{q_{near}}) + c(x')$;
        if $c' < Cost(\boldsymbol{q_{new}})$ and $c' < c_{min}$ then
            $\mathbf{q_{min}} \leftarrow \mathbf{q_{near}}$;
            $c_{min} \leftarrow c'$;
        end
    end
end
```

Algorithm 4: Rewire

```
Data: $\mathcal{T}, Q_{near}, \mathbf{q_{min}}, \mathbf{q_{new}}$
Result: $\mathcal{T}$
for $\boldsymbol{q_{near}} \in Q_{near} \backslash \{\boldsymbol{q_{min}}\}$ do
    $(\mathbf{x'}, \mathbf{u'}, T') \leftarrow Steer(\mathbf{q_{new}}, \mathbf{q_{near}})$;
    if $ObstacleFree(\boldsymbol{x'})$ and $\boldsymbol{x'}(T') = \boldsymbol{q_{near}}$ and
      $Cost(\boldsymbol{q_{new}}) + c(\boldsymbol{x'}) < Cost(\boldsymbol{q_{near}})$ then
        $\mathcal{T} \leftarrow ReConnect(\mathbf{q_{new}}, \mathbf{q_{near}}, \mathcal{T})$;
    end
end
```

Therefore, the lines of code inside the *if ObstacleFree(x_{new})* constitute the procedure that characterizes the RRT* algorithm and its variants. To better explain it, it is possible to look at *Figure 8*. Once the algorithm has randomly selected a new node q_{new}, it applies the *NearByVertices* in order to find the set Q_{reach} of nearest nodes to q_{new} in the tree (a). In the figure, to simplify, Q_{reach} is represented by the nodes inside the green circle. The nearest node $q_{nearest}$ won't automatically become the parent of q_{new}. Instead, the *ChooseParent* chooses, as parent, the node that allows to reach q_{new} from q_{ini} through the shortest possible path (b). The last step of the procedure consists in *Rewiring* the tree (c): it can be easily seen that both $q_{nearest}$ and q_A can be reached, starting from the root q_{ini}, with a shorter path by passing through q_{new}. Therefore the tree structure is updated: q_B and q_C now don't have children,

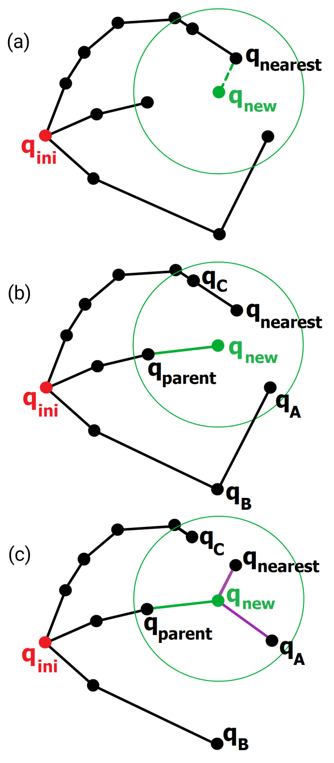
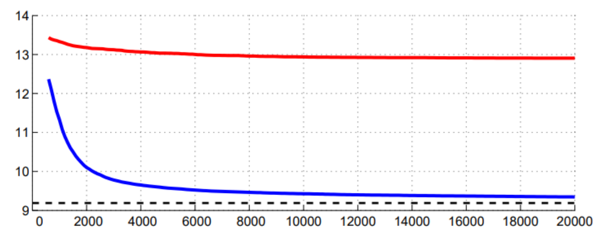


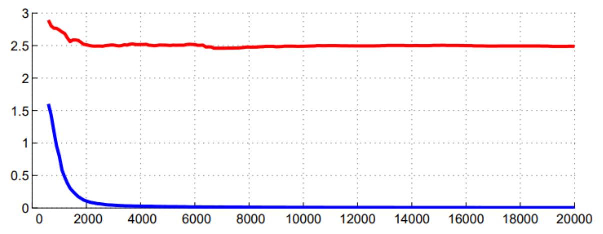
Figure 8: *NearByVertices*, *ChooseParent* and *Rewire* of RRT*

while q_{new} becomes the parent of both q_A and $q_{nearest}$.

As RRT, also in this version the state space is searched incrementally with a bias for open spaces, the distribution of the vertices in the space converges towards the sampling distribution, few parameters and heuristics are required and probabilistic completeness is guaranteed. Also the asymptotic computational complexity is mantained the same.
Contrarywise, unlike RRT, this algorithm can solve an optimal path planning problem, which asks for finding a feasible path while minimizing the cost function c introduced in Section 3. Therefore RRT* is asimptotically optimal, meaning that, when the number of nodes in the tree tends to infinity, the probability of finding an optimal solution, if one exists, will tend to one. This is due to the fact that the algorithm, until the maximum number of iterations is achieved, keeps adding new nodes in the tree and, through the rewiring procedure, updates its structure, thus linking each configuration in the tree to the root configuration with the least cost solution possible.
For demonstration and comparison purposes of what has been said about the optimality of RRT and RRT*, the results of some experiments conducted by Karaman and Frazzoli [6] are presented. Both algorithms were run for 20.000 iterations, 500 times and the cost of the best path in the trees were averaged for each iteration. The results are shown in Figure 9, which shows that RRT has an higher cost, whereas the RRT* converges to the optimal solution. Moreover, the variance over different RRT runs



(a)



(b)

Figure 9: The cost of the best paths in RRT (red) and RRT* (blue); the optimal cost is shown in black(a). The variance of the trials is shown in (b).

approaches 2.5, while that of the RRT* approaches zero. Hence, almost all RRT* runs have the property of convergence to an optimal solution, as expected. In Figure 10 the considered scenario is a squared environment with no obstacles. The cost function is the Euclidean path length. The trees obtained by the two implementations (RRT on the top row and RRT* on the bottom row) are shown at several stages. It is noticeable how the RRT algorithm does not improve the feasible solution to converge to an optimal one. On the other hand, the RRT* further improves the paths in the tree to lower cost ones, as the cardinality of the tree increases. In fact in Figure 11 it is evident how the RRT* first rapidly explores the state space just like the RRT; moreover, as the number of nodes in the tree increases, the cost of the solution found decreases considerably in RRT*.

However RRT* also presents some disadvantages. In fact, it has no bias towards the goal, it is computationally challenging since the *Steering* function is called repeatedly and it needs an infinite amount of time to find an optimal path, thus resulting in a slow convergence rate. Moreover, due to the vast exploration of the whole space, it suffers from memory issues as the cardinality of $\mathcal{T}$ increases. Therefore, the implementation of the RRT* algorithm in embedded systems with limited memory can become problematic, since the number of nodes in the tree grow indefinitely as the solution gets optimized. A chance to alleviate this problem would be terminating the algorithm

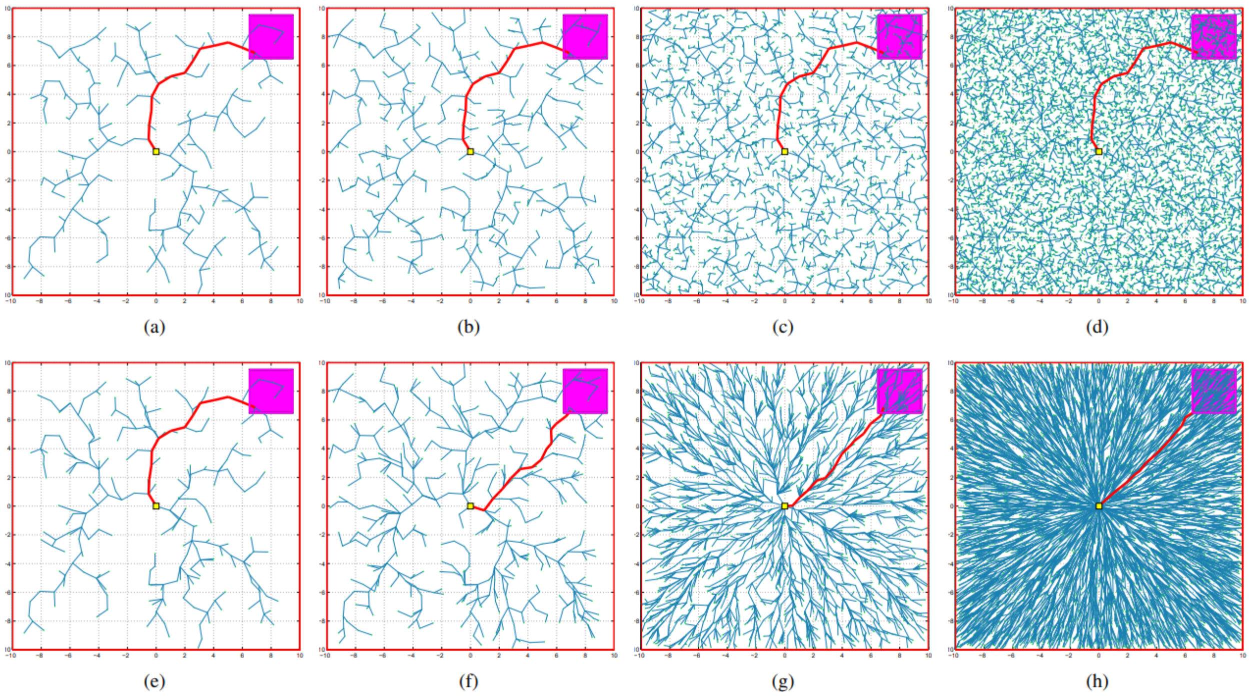


Figure 10: A Comparison of the RRT* and RRT algorithms on a simulation example with no obstacles. Both algorithms were run with the same sample sequence. Consequently, in this case, the vertices of the trees at a given iteration number are the same for both of the algorithms; only the edges differ. The edges formed by the RRT algorithm are shown in (a)-(d), whereas those formed by the RRT* algorithm are shown in (e)-(h). The tree snapshots (a), (e) contain 250 vertices, (b), (f) 500 vertices, (c), (g) 2500 vertices, (d), (h) 10,000 vertices. The goal regions is shown in magenta (in upper right). The best paths that reach the target in all the trees are highlighted with red.

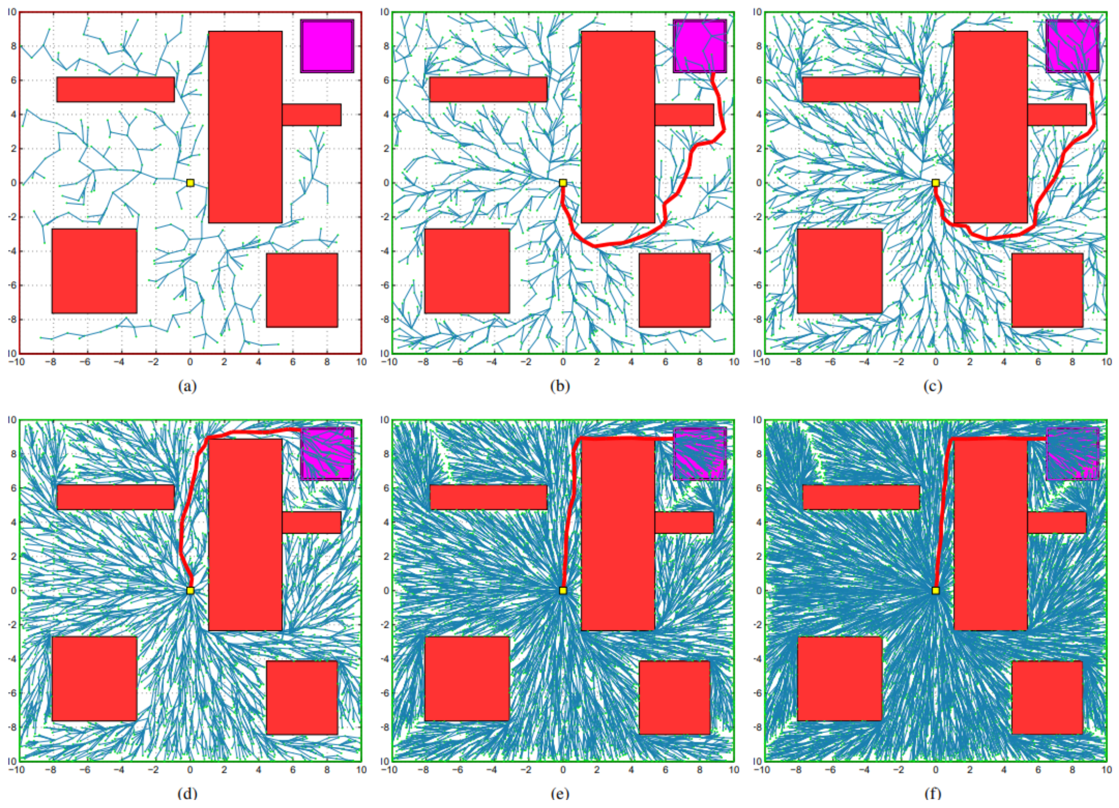


Figure 11: RRT* algorithm shown after 500 (a), 1.500 (b), 2.500 (c), 5.000 (d), 10.000 (e), 15.000 (f) iterations. The red line represents the best path to reach the goal at that iteration.

once a certain number of nodes has been reached. Although this would not guarantee the convergence of the solution to the optimal one. It follows a series of algorithm proposed with the aim of solving the described problems.

4.6 RRT*-Fixed Nodes

This version tries to handle the memory issues of RRT*, by simply imposing a restriction on the maximum cardinality M of the tree, employing a node removal procedure. The standard RRT* algorithm is run until the tree has grown to the predefined number M of nodes. If a feasible path to the goal has not been reached, the algorithm must be restarted. Otherwise, the optimization of the tree continues: while iterating, a higher performance node will be added to $\mathcal{T}$ only if a weaker node is removed. Different heuristics can be applied in order to decide which node to remove. The authors who introduced the memory-limitation idea [1] choose the nodes with one or no child, meaning that this node is not on a path reaching the goal state. If there is more than one node without children, one of them is selected randomly. In this way, RRT*-FN decreases the memory consumption, still outperforming RRT and coming close to RRT* with respect to the optimality of the solution path. This is reached at the cost of a slower convergence rate.

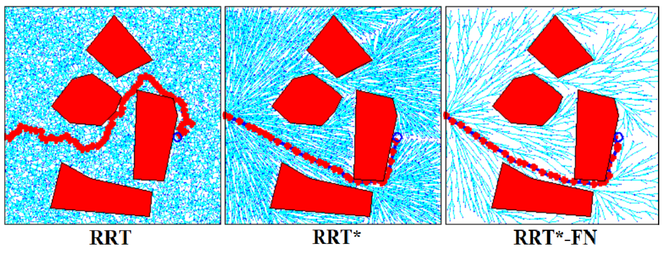


Figure 12: 2D mobile robot navigation task. RRT (left) finds a feasible but not optimal solution. Trees generated by RRT* (middle) and RRT*Fixed Nodes (right) both converge to the optimal solution. RRT*FN tree resembles to a sparse version of the RRT* tree.

4.7 B-RRT*

As RRT, also RRT* does not present any bias towards the goal, making it difficult to find a solution in environments with many obstacles and narrow passages. For this aim, a bidirectional version exists. It builds two trees at the same time, $\mathcal{T}_A$ and $\mathcal{T}_B$ and tries to connect them. The algorithm starts always in the same way, sampling $\mathbf{q_{rand}}$ and calling the functions (presented in Section 4.4) $NearestNeighbour$, $Steer$, $NearByVertices$, $ChooseParent$, $AddNewNode$ and $Rewire$ at each iteration alternately on $\mathcal{T}_A$ or $\mathcal{T}_B$. Therefore, if in this first phase the new node $\mathbf{q_{new}}$ computed starting from $\mathbf{q_{rand}}$ was added to $\mathcal{T}_A$, now the algorithm tries to connect $\mathbf{q_{new}}$ to the vertex in $\mathcal{T}_B$ that provides the least cost path. If the path found is feasible and collision-free, it is added to connect the two trees, establishing a connection path from $\mathbf{q_{ini}}$ to $\mathbf{q_{goal}}$.
A further improvement of the B-RRT* algorithm was presented in [15], who introduced the ***Intelligent-Bidirectional RRT**** algorithm, that showed superior efficiency in complex cluttered environments, by using the bidirectional approach and introducing intelligent sample insertion heuristic for fast convergence to the optimal path solution. Differently from the B-RRT* that at each iteration adds q_{new} alternatively to $\mathcal{T}_A$ or $\mathcal{T}_B$, IB-RRT* computes the nearest neighbour sets of nodes by both trees and adds q_{new} to the nearest vertex of the best selected tree, thus allowing to minimize the cost of the paths.
These two alternative versions of the RRT* algorithm, being simply a bidirectional version of RRT* (IB-RRT* also with intelligent sampling insertion), inherit the probabilistic completeness and asymptotic optimality properties of RRT*. Yet, they allow a faster convergence to the optimal solution, thus also finding easier a feasible path from the initial to the goal configuration: a smaller number of iterations and time is required and the success rate is higher with respect to the standard RRT*. In particular IB-RRT* showed the most significant increase in the fast convergence rate, still keeping the computational complexity almost equal to standard RRT*.

4.8 Primitive-based RRT*

In RRT* multiple calls to the $Steer$ function are necessary at each iteration in order to compute the minimum cost path connecting two configurations (both when searching the nearest neighbour and when rewiring). However, this is computationally challenging, especially when dealing with complex dynamics, such as for nonholonomic vehicles because of the presence of kinodynamic constraints. For this reason, an extension of RRT* has been proposed [17], in which a database of pre-computed motion primitives is used in order to avoid performing these computations online. This version of RRT* is based on a uniform discretization of the state space and on the computation of a finite set of motion primitives with boundary conditions on the points of the gridded space: in fact the algorithm computes offline the set of primitives linking the different points of the gridded space (an example in Figure 13).
In particular, it can be observed that the four trajectories in Figure 14 are characterised by the same cost and can be easily mapped to the "reference trajectory" corresponding

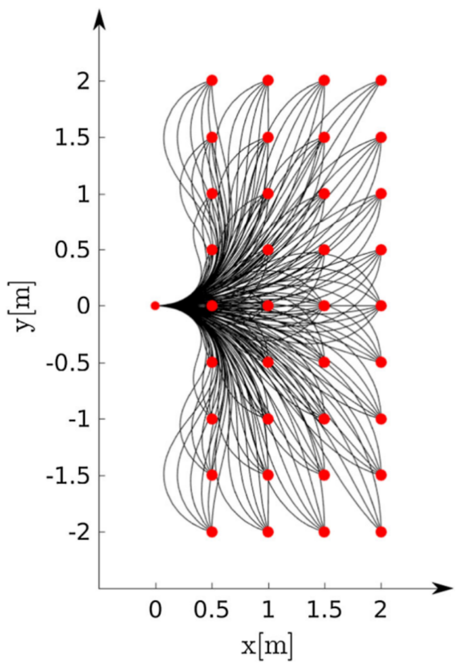


Figure 13: A subset of the motion primitives computed for a 3D search space (x, y, θ). Red dots correspond to the initial and final positions, and black lines represent the resulting trajectories for different final orientations θ.

to the boundary value pair q_0, q_f.

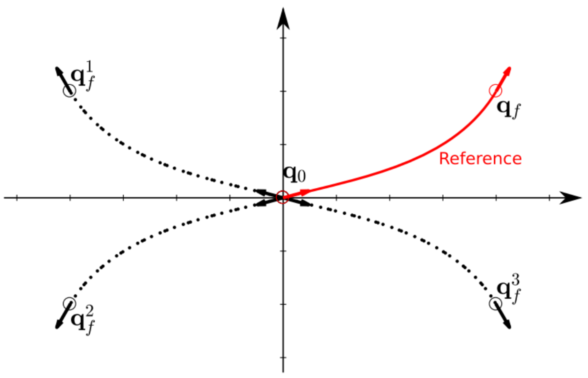


Figure 14: The three dashed trajectories can be traced back to the reference one (red).

Therefore, in order to keep the size of the dataset as small as possible, they computed only the ”reference primitives”. All other cases can be obtained by translating and rotating the trajectories stored in the dictionary. This precomputed catalogue of primitives is saved on a text file, such that, while the algorithm runs, the steering function doesn’t need to compute every time the best path connecting two configurations, but simply searches for the corresponding primitive in the precomputed catalogue. This allows a significant decrease in the computational time of the algorithm. Although the increase in speed comes at the price of a rise in the cost of the solution and a reduction in the success rate since, due to the discretization of the state space, it becomes more difficult to find feasible paths: for example, in scenes in which the points in the narrow passages do not fall on the discrete grid, the primitive-based algorithm would not be able to find a solution, so a finer resolution should be considered. But it cannot be known a priori. Therefore, the choice between the standard and primitive-based versions of RRT* must be done evaluating the needs related to the considered application and also the computational resources. If a quicker and lighter algorithm is needed, the primitive-based version would be better, while if it is necessary to have the solutions with the lowest possible cost, then standard RRT* should be used.

4.9 RRT*-Smart

RRT*-Smart is an informed version that allows faster convergence rate by exploiting a path optimization technique and intelligent sampling. The first phase finds the goal in the same way the standard RRT* does. In the second phase a shorter path between two nodes is found, through some technique, for example triangular inequality where, as shown in Figure 15, after checking that the path connecting diagonally x and y is viable, it updates the path, removing z from it and so smoothing the final path. This method is applied to all the nodes in the path.

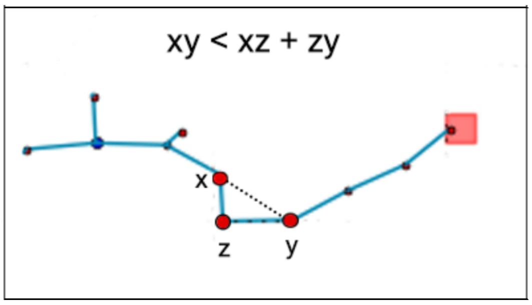


Figure 15: Triangular inequality is exploited in order to smooth the path from the root to the goal.

The third phase exploits the nodes remained in the optimized path as beacons in an intelligent sampling procedure: in particular the beacons are the center of a ball radius within which configurations can be sampled. In this way, the tree is biased to draw more samples along the beacons in order to further smooth the path toward the goal. Through this three-phase procedure a smaller number of iterations is needed, thus guaranteeing a significant improvement in the convergence rate of the algorithm, mantaining the same space and time complexity of RRT*.

4.10 Informed RRT*

This is another algorithm, presented by Gammell et al. [4], that deals with the problem of the slow convergence. It reaches an order-of-magnitude improvement and, if no obstacle inhibit the growth of the tree, allows a linear convergence to the solution. The novelty introduced by this version lies in the *Sampling* procedure, while the rest of the algorithm is the regular RRT*. The classical RRT* extracts the new samples through a uniform distribution from Q_{free}. Informed RRT* extracts samples through the same procedure until the goal has not been found. Then, the algorithm will uniformly take the samples in the region between the start and the goal, bounded in an n-dimensional elongated and symmetrical ellipsoid, that is gradually shrinked during iterations.

The algorithm starts by initializing the tree $\mathcal{T}$ with the initial configuration $\mathbf{q_{ini}}$, the empty set of goal nodes Q_{soln}, and the cost of the current solution $c_{best} = \infty$. $\mathbf{q_{centre}}$ is the centre of the ellipsoid and c_{min} the real distance[10] between $\mathbf{q_{ini}}$ and $\mathbf{q_{goal}}$, focal points of the ellipsoid. It also calculate the matrix $\mathbf{C}$, that performs a rotation needed to compute the ellipsoid: $\mathbf{C} = \mathbf{U}\text{diag}1, ..., 1, det(U)det(V)V^T$, where $\mathbf{U}, \mathbf{V} \in \mathbb{R}^{d \times d}$ are unitary matrices obtained through the Singular Value Decomposition (SVD)[11] of the matrix $M = a_1 1_1^T$, with 1_1^T the first column of the identity matrix.

The algorithm cycles on the number of iterations N, first updating the minimum solution cost c_{min} (in the case Q_{soln} is not empty) and then executing the standard operations of the RRT*. In the case Q_{soln} is not empty, meaning the goal configuration has already been sampled, the new configurations will not be sampled from the whole free configuration space Q_{free}, but by following the passages in the *if* in Algorithm 6. The point q_{rand} is uniformly sampled from the area of the ellipsoid: $q_{rand} = \mathbf{CL}\mathbf{q}_{ball} + \mathbf{q}_{centre}$, with q_{ball} uniformly sampled from a unit n-ball, the $\mathbf{C}$ precomputed above and L=diag$\{\frac{c_{best}}{2}, \frac{\sqrt{(c_{best}^2 - c_{min}^2)}}{2}, ..., \frac{\sqrt{(c_{best}^2 - c_{min}^2)}}{2}\}$ that performs a transformation. As c_{best} approaches to the real distance c_{min}, the square root of the error tends to zero and the ellipsoid shrinkes. In Figure 16 it is shown the ellipsoid computed at a certain iteration of the algorithm, while Figure 17 shows how the tree grows.

In this way, the algorithm increases the probability of sampling from a subspace that can only lead to an improvement of the current path cost to the goal and it also

Algorithm 5: Informed RRT*

```
Data: $\mathbf{q_{ini}}, \mathbf{q_{goal}}$
Result: $\mathcal{T}$
$\mathcal{T} \leftarrow InitializeTree()$;
$\mathcal{T} \leftarrow AddNewNode(\emptyset, \mathbf{q_{ini}}, \mathcal{T})$;
$Q_{soln} \leftarrow \emptyset$;
$c_{best} \leftarrow \infty$;
$\mathbf{q_{centre}} \leftarrow (\mathbf{q_{ini}} + \mathbf{q_{goal}})/2$;
$c_{min} \leftarrow$ Line $(\mathbf{q_{ini}}, \mathbf{q_{centre}})$;
$\mathbf{a}_1 \leftarrow (\mathbf{q_{goal}} - \mathbf{q_{ini}})/\|\mathbf{q_{goal}} - \mathbf{q_{ini}}\|$;
$\mathbf{id}_1 \leftarrow 1_1^T$;
$\mathbf{M} \leftarrow \mathbf{a}_1\mathbf{id}_1$;
$\mathbf{U\Sigma V}^T \leftarrow SVD(M)$;
$\mathbf{C} \leftarrow \mathbf{U}\text{diag}1, ..., 1, det(U)det(V)V^T$;
for $i = 1$ to $i = N$ do
    $c_{min} \leftarrow min_{x_{soln} \in X_{soln}} cost(x_{soln})$;
    $\mathbf{q_{rand}} \leftarrow Sample(c_{best}, c_{min}, \mathbf{q_{best}}, \mathbf{C})$;
    $\mathbf{q_{nearest}} \leftarrow NearestNeighbours(\mathbf{q_{rand}}, \mathcal{T})$;
    $(\mathbf{x_{new}}, \mathbf{u_{new}}, T_{new}) \leftarrow Steer(\mathbf{q_{nearest}}, \mathbf{q_{rand}})$;
    $\mathbf{q_{new}} \leftarrow LastElement(\mathbf{x}_{new})$;
    if $ObstacleFree(\boldsymbol{x_{new}})$ then
        $Q_{near} \leftarrow NearByVertices(\mathbf{q_{near}}, \mathcal{T})$;
        $\mathbf{q_{min}} \leftarrow$
          $ChooseParent(Q_{near}, \mathbf{q_{nearest}}, \mathbf{q_{rand}}, \mathbf{x_{new}})$;

        $\mathcal{T} \leftarrow AddNewNode(\mathbf{q_{min}}, \mathbf{q_{new}}, \mathbf{u_{new}})$;
        $\mathcal{T} \leftarrow Rewire(\mathcal{T}, Q_{near}, \mathbf{q_{min}}, \mathbf{q_{new}})$;
    end
    if $InGoalRegion(\boldsymbol{q_{new}})$ then
        $Q_{soln} \leftarrow Q_{soln} \cup \mathbf{q_{new}}$
    end
end
```

Algorithm 6: Sample

```
Data: $c_{best}, c_{min}, \mathbf{q_{centre}}, \mathbf{C}$
Result: $\mathbf{q_{rand}}$
if $c_{best} < \infty$ then
    $r_1 \leftarrow c_{best}/2$;
    $r_{1 i=2,...,d} \leftarrow \frac{\sqrt{(c_{best}^2 - c_{min}^2)}}{2}$;
    L$\leftarrow$ diag $(r)$;
    $\mathbf{q_{ball}} \leftarrow$ SampleUnitBall;
    $\mathbf{q_{rand}} \leftarrow (\mathbf{CL}\mathbf{q}_{ball} + \mathbf{q}_{centre})$;
    return $\mathbf{q_{rand}}$;
end
return $\mathbf{q_{rand}} \sim$ UniformSample$(Q_{free})$
```

[10] It can be computed through the Euclidean distance.

[11] In linear algebra, the Singular Value Decomposition of a matrix is a factorization of that matrix into three matrices $\mathbf{U}$, Σ and $\mathbf{V}^T$. It has some interesting algebraic properties and conveys important geometrical and theoretical insights about linear transformations.

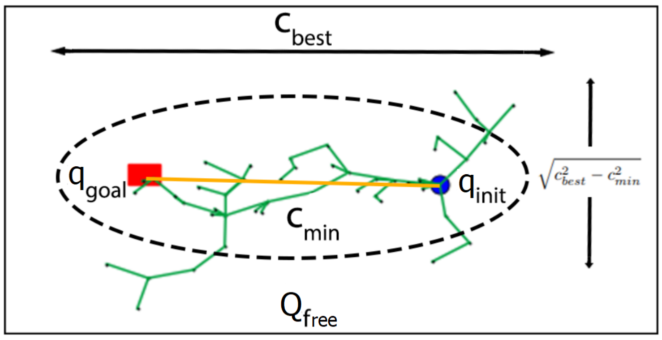


Figure 16: Ellipsoid computed by Informed RRT*. After the goal has been found, all samples are taken from within this ellipsoid area.

limits the solutions only to paths within the ellipsoid. Informed RRT* inherits from RRT* the asimptotic optimality, but converges to an optimal solution in a finite time. Therefore it manages in improving the convergence rate, as was done also by RRT*-Smart. But unlike the latter, which requires fine tuning of the biasing radius, Informed RRT* does not have extra parameters. Also, by taking the new samples from the ellipsoid uniformly, it does not violate the assumption of uniform density.
However with this method the heuristic sampling space becomes too large, so this algorithm has difficulties in scaling well in higher dimensions. It also presents problems when dealing with spaces with narrow passages to the goal.

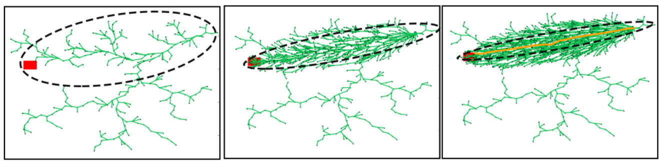


Figure 17: Growth of the tree as iterations increase: on the left, the tree expanded also outside the ellipsoid, since the goal was not found yet. Once the goal is found, the new nodes sampled are only inside the ellipsoid (center). On the right the ellipsoid shrinked since the error on the optimal path decreased.

4.11 Real-Time RRT*

In some application fields, such as many robotic tasks and computer games, we need to solve a real-time path planning problem in a dynamic environment, in which the obstacles cannot be fixed and it is allowed to change the goal point (it is the case of multiple-query tasks, i.e. querying paths to multiple goals). Therefore the formalization of this problem requires to find a path from the agent position $\mathbf{q_0}$ to $\mathbf{q_{goal}}$, where $\mathbf{q_{goal}}$ can be changed on the fly. Also, the path to $\mathbf{q_{goal}}$ should be of minimum length, according to a certain cost function (for example the Euclidean length of the path). Furthermore, the algorithm is real-time, meaning that a real-time response is required: we have a limited amount of time for operations. Naderi et al. [14] proposed the algorithm RT-RRT*, a variant of RRT* and Informed RRT* that uses an online tree rewiring strategy in order to allow the tree root to move with the agent without discarding previously sampled paths, thus obtaining real-time path-planning in a dynamic environment. This type of path planning algorithm is necessary to react to the rapid changes in the environment: they have to find a balance between the goodness of the path and the short search time.

Algorithm 7: Real-Time RRT*

```
Data: $\mathbf{q_a}, Q_{obs}, \mathbf{q_{goal}}$
Result: $\mathbf{q_{rand}}$
$\mathcal{T} \leftarrow InitializeTree()$;
for i = 1 to i = N do
    Update $\mathbf{q_{goal}}, \mathbf{q_a}, Q_{free}, Q_{obs}$;
    while time is left for Expansion-Rewiring do
        Expand and Rewire $\mathcal{T}$;
    end
    Plan ($q_0$, $q_1$, ..., $q_k$) to the goal;
    if $q_a$ is close to $q_0$ then
        $q_0 \leftarrow q_1$;
    end
    Move the agent toward $q_0$ for a limited time
end
```

The algorithm interleaves path planning with tree expansion and rewiring. The tree is initialized with $\mathbf{q_a}$ as its root. At each iteration, the goal, agent and obstacles positions in the environment are updated. Then the tree is expanded and rewired for a limited user-defined time (*while*). Then a path ($\mathbf{q_0}$, $\mathbf{q_1}$, ..., $\mathbf{q_k}$) is planned starting from the current tree root for a limited user-defined amount *k* of steps further. At each iteration we move the agent for a limited time to keep it close to the tree root, $\mathbf{q_0}$. When path planning is done and the agent is at the tree root, we change the tree root to the next immediate node after $\mathbf{q_0}$ in the planned path, $\mathbf{q_1}$. Hence, we enable the agent to move on the planned path towards the goal. By expanding and taking actions in an alternate way, this method does not require to wait the tree to be fully built. The *Expansion and Rewiring* procedure has a structure similar to the methods already seen: a new node $\mathbf{q_{rand}}$ is sampled , the set of nearest nodes is computed, under a certain density condition of the tree the new node is added to the tree and rewiring is performed. As for Informed RRT*, the environment is uniformly sampled until a path to $\mathbf{q_{goal}}$ is found. Then, samples are not only taken from the ellipsis with $\mathbf{q_0}$ and $\mathbf{q_{goal}}$ as focal points: with different probabilities, they can be taken from the line between $\mathbf{q_{goal}}$ and the node of the tree that is closest to $\mathbf{q_{goal}}$, from the environment uniformly or from the ellipsis. Also, in this case the rotation of the ellipsis is updated at every iteration. These differences from Informed RRT* are necessary since, because of the changes in the tree root and $\mathbf{q_{goal}}$, it is we constantly required to rewire random parts of the tree and explore the environment for later queries in multi-query tasks.

The presence of dynamic obstacles is treated by setting to infinity the cost-to-reach value c_i of the configurations in the obstacle region. Thus, rewiring will create another path for nodes with infinite c_i.

When the new node has to be added to the tree, as in RRT*, the algorithm needs to find the parent with the minimum cost-to-reach value c_i in X_{near}. The cost c_i from $\mathbf{q_0}$ to $\mathbf{q_i}$ needs to be recomputed whenever a cost c_j of an intermediate node in a path to node $\mathbf{q_i}$ changes or a new node is added to the path or there are any changes in $\mathbf{q_0}$ or of dynamic obstacles (obstacles that for example can block an ancestor of $\mathbf{q_i}$).

Rewiring is done when a node $\mathbf{q_i}$ gets a lower cost-to-reach value c_i by passing from another node instead of its parent. In this case, rewiring should be done around both the new added node (as in the classical RRT*) and around the already added nodes (differently from RRT*). In fact, because of the changing of the tree root and of dynamic obstacles, it is usually required to rewire large portion of the tree, done by rewiring a random part of the tree and rewiring starting from the tree root.

Finally, paths are planned in two ways: when the tree reaches $\mathbf{q_{goal}}$, the path from $\mathbf{q_0}$ to $\mathbf{q_{goal}}$ is in the tree, so we can just update the path; when the tree has not reached the goal yet, it is necessary to plan a path to get as close as possible to $\mathbf{q_{goal}}$, by using a cost function $f_i = c_i + h_i$, where h_i is an estimator of the remaining path towards the goal. Then, if the planned path leads us to a location closer to $\mathbf{q_{goal}}$, we update the best already found path.

The method presented is a real-time version of RRT* that allows to find shorter paths with a smaller number of iterations to one or multiple goal points with respect to other real time versions. Although, it also has some limitations: it requires a large memory capacity because the whole tree is stored at all times and it is challenging in unbounded and large environments.

5 Conclusions

This paper presented Rapidly-exploring Random tree, a sampling-based method that allows to find a solution to a lot of path planning problems, efficiently searching non-convex, high-dimensional spaces and handling obstacles and differential constraints. They show computational efficiency, probabilistic completeness and they find a lot of applications, from autonomous robotics, to computer-aided surgery, videogames' characters, assembly and manufactoring and a lot of other real-world applications.

Different versions of the RRTs algorithm have been introduced in this paper, each with its own advantages and disadvantages and dealing with some issues encountered when trying to solve a motion planning problem through this class of methods. Still many other versions have been developed, introducing new properties and potentials. For example Choudhury, Scherer and Singh designed RRT*-AR [2], a planning system to generate safe alternate routes in real-time, in order to address the autonomously landing of a helicopter while considering a realistic context: unmodelled obstacles, cluttered and multiple potential landing zones, geographical terrain, sensor limitations, unaccounted disturbances and the pilot's contextual knowledge. Sintov and Shapiro developed Time-Based RRT (TB-RRT) [20], method that enables trajectory planning of a dynamic system that has to reach a goal within a specified time, by adding time parameters to the nodes in the tree such that each node denotes a specific state in a specific time. Other variations deal with an adaptive step-size $\pm\alpha$ to speed up motion in wide open areas (greedy exploration), or address many extensions of the canonical planning problem, for example nonholonomic constraints, manipulation planning and other.

Therefore RRT, in all its versions, is a powerful tool for solving a wide variety of motion planning problems and their requirements. The choice between the different versions will be done by evaluating the needs related to the considered application and also the computational resources available.

References

[1] O. Adiyatov and H. A. Varol. "Rapidly-exploring random tree based memory efficient motion planning". In: *2013 IEEE International Conference on Mechatronics and Automation*. Aug. 2013, pp. 354–359. DOI: 10.1109/ICMA.2013.6617944.

[2] Sanjiban Choudhury, Sebastian Scherer, and Sanjiv Singh. "RRT*-AR: Sampling-based alternate routes planning with applications to autonomous emergency landing of a helicopter". In: *2013 IEEE International Conference on Robotics and Automation*. IEEE. 2013, pp. 3947–3952.

[3] Mohamed Elbanhawi and Milan Simic. "Sampling-Based Robot Motion Planning: A Review". In:

IEEE Access 2 (2014), pp. 56–77. DOI: 10.1109/ACCESS.2014.2302442.

[4] Jonathan D Gammell, Siddhartha S Srinivasa, and Timothy D Barfoot. "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, pp. 2997–3004.

[5] Jason A Janet, Ren C Luo, and Michael G Kay. "The essential visibility graph: An approach to global motion planning for autonomous mobile robots". In: *Proceedings of 1995 IEEE international conference on robotics and automation*. Vol. 2. IEEE. 1995, pp. 1958–1963.

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

[7] Sertac Karaman and Emilio Frazzoli. "Sampling-based Algorithms for Optimal Motion Planning". In: (2011). arXiv: 1105.1186.

[8] Sertac Karaman and Emilio Frazzoli. "Sampling-based motion planning with deterministic μ-calculus specifications". In: *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*. IEEE. 2009, pp. 2222–2229.

[9] Lydia E Kavraki et al. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces". In: *IEEE transactions on Robotics and Automation* 12.4 (1996), pp. 566–580.

[10] O. Khatib. "Real-time obstacle avoidance for manipulators and mobile robots". In: *Proceedings. 1985 IEEE International Conference on Robotics and Automation*. Vol. 2. 1985, pp. 500–505. DOI: 10.1109/ROBOT.1985.1087247.

[11] Yoram Koren, Johann Borenstein, et al. "Potential field methods and their inherent limitations for mobile robot navigation." In: *ICRA*. Vol. 2. 1991, pp. 1398–1404.

[12] Darwin Kuan, James Zamiska, and Rodney Brooks. "Natural decomposition of free space for path planning". In: *Proceedings. 1985 IEEE International Conference on Robotics and Automation*. Vol. 2. IEEE. 1985, pp. 168–173.

[13] Steven M LaValle et al. "Rapidly-exploring random trees: A new tool for path planning". In: (1998).

[14] Kourosh Naderi, Joose Rajamäki, and Perttu Hämäläinen. "RT-RRT* a real-time path planning algorithm based on RRT". In: *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games*. 2015, pp. 113–118.

[15] Ahmed Hussain Qureshi and Yasar Ayaz. "Intelligent bidirectional rapidly-exploring random trees for optimal motion planning in complex cluttered environments". In: *Robotics and Autonomous Systems* 68 (2015), pp. 1–11.

[16] John H Reif. "Complexity of the mover's problem and generalizations". In: *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*. IEEE Computer Society. 1979, pp. 421–427.

[17] Basak Sakcak et al. "Sampling-based optimal kinodynamic planning with motion primitives". In: *Autonomous Robots* 43.7 (2019), pp. 1715–1732.

[18] Jacob T Schwartz and Micha Sharir. "A survey of motion planning and related geometric algorithms". In: *Artificial Intelligence* 37.1-3 (1988), pp. 157–169.

[19] Jacob T Schwartz and Micha Sharir. "On the "piano movers'" problem I. The case of a two-dimensional rigid polygonal body moving amidst polygonal barriers". In: *Communications on pure and applied mathematics* 36.3 (1983), pp. 345–398.

[20] Avishai Sintov and Amir Shapiro. "Time-based RRT algorithm for rendezvous planning of two dynamic systems". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2014, pp. 6745–6750.

[21] Chris Urmson and Reid Simmons. "Approaches for heuristically biasing RRT growth". In: *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453)*. Vol. 2. IEEE. 2003, pp. 1178–1183.

[22] Marios Xanthidis et al. "Motion planning by sampling in subspaces of progressively increasing dimension". In: *Journal of Intelligent & Robotic Systems* 100.3 (2020), pp. 777–789.

[23] Peter Yap. "Grid-based path-finding". In: *Conference of the canadian society for computational studies of intelligence*. Springer. 2002, pp. 44–55.