POLITECNICO DI TORINO
Master degree course in Mechatronic Engineering

Master Degree Thesis

Risk-aware path planning and replanning algorithm for UAVs

Supervisors
prof. Alessandro Rizzo
dott. Stefano Primatesta

Candidates
Luigi Mazzara

Anno accademico 2017-2018

Abstract

During the last decades, the Unmanned Aerial Systems (UAS) are gaining momentum, and they are widely used in various types of operations bringing enormous benefits. Nonetheless allowing such operations requires great awareness but even rising worries, the people safety is in fact one of the main obstacles to be evaluated before any operations could start because of their high population density on working environments. Therefore, the aim of the project conducts to the creation of a structured low-altitude airspace able to achieve autonomous operations of low-cost unmanned vehicles, while maintaining the safety for people on the ground.
In this context, a new research project at TIM Joint Open Lab in Politecnico di Torino began to analyze the potentiality of cloud robotics, in order to build a completely new Traffic Manager able to handle the complexity of this environment, integrate drones efficiently into air traffic and finally ensuring safety for people on ground. So, after studying the state of the art of the involved technologies, we propose and developed a Cloud-Based UTM, called CBUTM, to accomplish the project's goal.
Developing a such kind of manager involves several category of knowledge, so the working group was divided into smaller ones, each one with different aspects to treat. My thesis, especially, will look to the research of the best path planning algorithm capable to be risk-awared and to execute suitable re-planning.

After deep examination of the state of the art, taking to account our precise condition of risk-aware and dynamic map, two variations of the Rapidly-exploring Random Trees (RRT* and RRT^X) have been designated to generate a safe flight mission in order to minimize our specific cost function. Going deeper, the proposed approach dwells on two distinct stages.
At first, an off-line path planning is accomplished; it computes the optimal global path in a static environment without time constraints depending only on the risk-map, in which cells describe a specific location and have been associated with a risk-cost. Afterwards, there is an online path planning, in which, considering a dynamic risk-map, the off-line path is fixed and fitted to the new map, always taking to account to minimize the cost function. Ergo, a quick response constitutes a fundamental design parameter because the system needs to revise the pathway in very

short time. The implementation is based on ROS (Robot Operating System), that played a key role in this project. Simulation results are still exposing, demonstrating both the validity of our approach and some design limitations, especially in the on-line part, while supporting further developments.

Contents

List of Tables

List of Figures

Acronyms

CBUTM	Cloud-Based UAVs Traffic Management
CCS	Cloud Control Station
MM	Map Manager
MPC	Model Predictive Control
NCS	Networked Control System
NFA	National Flight Authorities
OMPL	Open Motion Planning Library
PP	Path Planner
PTP	Predictive Trajectory Planner
RHC	Receding Horizon Control
ROS	Robot Operating System
UAS	Unmanned Aerial System
UAV	Unmanned Aerial Vehicle
UCTF	Unmanned Capture the Flag
UTM	UAS Traffic Management

Chapter 1

Introduction

During last two decades, the usage of Unmanned Aerial Vehicles (also known as Drones) has reached continuous growth in different and unexpected applications. Starting from military field and following the rapid technological progress recorded during the 2000s, thanks to the enormous progress made on remote control, the UAVs have begun to be used also in the civil field, as in surveillance, in aerofotogrammetry, in logistics, in data collection, in the control of power lines and oil pipelines, in retrieving, to cite a few.

Definition 1.1. An Unmanned Aerial Vehicle (UAV) is a powered aerial vehicle that does not carry a human operator, uses aerodynamic forces to provide vehicle lift, can fly autonomously or be piloted remotely, can be expendable or recoverable, and can carry a lethal or nonlethal payload[8].

This kind of innovation, indeed, is certainly not only flanked by technological evolutions. National and international institutions are working on regulation of these systems and their use but, at the same time, their attempts represent a brake on development too.
Fittingly, risks for privacy and safety of people are just two of several issues raised by the development of these kind of systems but the real problem is that the growth of their usage has not been accompanied by a proper adaptation of laws and rules, ending in a lack of coherent legislation and, thus, in imposition of many limitations. Despite of some concessions, the common feeling tells about so many obstacles to overcome in order to fly a drone, with the legislative apparatus being as usual too slow in adapting to this new technology.

Nevertheless, without going far into the analysis of current administrative restrictions, various companies and universities are already involved to manage the condition of risk for safety of people in better way thanks to the growth and the consolidation of internet technology, like the Internet of Things (IoT) and Cloud Robotics.

In particular, TIM, a telecommunication service provider company, has financed a research project in the Joint Open Lab at Politecnico of Turin, aimed to the development of a cloud-based traffic management system for the UAVs.
Taking advantage of the potential offered by cloud computing and next-generation of mobile networks (5G), it is possible to provide an autonomous framework to support commercial fully-autonomous UAS operations in a certain airspace. At the same time, this possibility has double useful advantages: it offers a unified approach to risk management that these missions involve and, consequently, it can guarantee a definite level of safety for people on the ground. A framework, able to access unmanned missions and to be the tool to control them, certainly requires a specified computing power, a big amount of available data and the ability, through Cloud Computing and 5G facilities, to communicate with vehicles, as Beyond Visual Line of Sight (BVLOS). Moreover, taking into account the huge range of their new commercial applications, supplying the framework with a coherent metric for risk assessment brings customers and companies to monitor and systematically settle, or at least effect change on, the execution of these missions.

1.1 Context Definition and Thesis Contributions

As defined above, the safety of people on ground is the main goal of the overall research project. A fully autonomous flight for UAS in a urban environment system must guarantee the safety standards imposed by local flight authorities and minimize, or at least manage, circumstances of crash.
For these reasons, the target of the task has been to design a Cloud-Based UAVs Traffic Management (or CBUTM) framework system in order to handle collisions or internal failures and deal with registration, identification and monitoring issues.

The overall design has been developed in ROS, the Robot Operating System, which is the most popular open source robotics middleware (i.e. collection of software frameworks for robot software development). It basically works through objects, known as nodes, which provide services and exchange messages between each other thanks to interactions called topics. Its structure allows each single node to run specific tasks simultaneously and also separately from other nodes, if it's needed, for achieving a robust and efficient network.

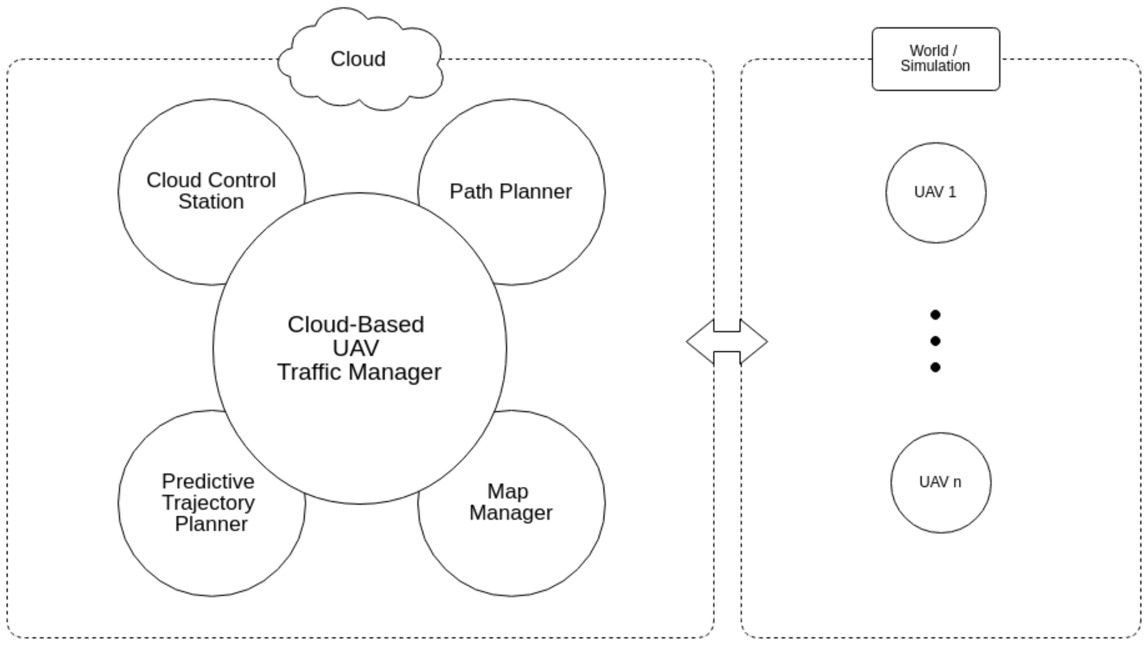


Figure 1.1: Graphical representation at top level of CBUTM's working principle

A general overview of the system is shown in 1.1, the main entities of the network are:

- Cloud Control Station, CCS
- Map Manager, MM
- Path Planner, PP
- Predictive Trajectory Planner, PTP

All these entities, defined as ROS nodes, work as different team in the same environment. First of all, there is a well structured risk assessment, developed in the Map Manager area, realized to generating a standardized metric to evaluate the risk for UAVs' missions. Thanks to this estimation, a maximum acceptable level of failures-per-hour and relative thresholds are settled. Thus, the risk-map is generated, on which path planning can be performed. Combined with the Predictive Trajectory Planner, in the Path Planning area different algorithms are studied to find the best fitting with relative context conditions. The creation of feasible waypoints-based paths rested on informations taken from the risk-map is the aim of this area.
Finally, the CBUTM uses a distributed Cloud-based Control system based on a receding horizon technique, in order to figure out the control of drone's network, it enters in the final Cloud Control Station area. It's the core of network, in which flying drones informations are collected, tracked and made available to handle the

traffic between single nodes.

Just as CBUTM works through supports between its various parts, so, in reality, the overall project has been divided into sub-groups, each one related to an above-mentioned area, acting in parallel. The global design of the architecture has been carried out by the whole team, aware that cooperation would have led us to improve the development of the project, to get better calls and to reach overall knowledge and not only about a single part.
Therefore, the whole work has been conducted by 4 colleagues within the JOL, indeed, several references to their papers have been inserted in my work [33],[48],[41] and my own work is focused on the development and the integration of Path Planning Algorithms.

1.2 State of Art

Informally speaking, given a robot with a description of its dynamics, of the environment, an initial state, and a set of goal states, the motion planning problem aims to define a sequence of control inputs so to drive the robot from its initial state to one of the goal states while obeying the rules of the environment. An algorithm to address this problem is said to be complete if it terminates in finite time, returning a valid solution if one exists, and failure otherwise. In order to generate this movement, during the years, many algorithms, based on different theoretical notions, have been explained.

In this section, we studied pros and cons of contemporary algorithms selecting the best match for our case, to develop and fit it to our case in order to take into account the risk for the population using a proper risk-map based on specific risk-costs. Risk costs in the risk-map are defined considering both static and dynamic factors. As a consequence, the risk-map is a dynamic map.

Planning process for UAS is very complex since there are more paradigms to consider than for a tradition manned aircraft, like, for example, lost link routes, sensors plan, the need to avoid the over flight of populated areas and so on.
Knowing this wide volume and analyzing respective advantages and drawbacks, main relevance is given to the algorithm's choice to find the best one that could get a feasible and optimal solution for each particular case. Classification of path planning algorithms can be made through these 2 principal categories:

Deterministic Algorithms It formally computes a mathematical function that has a unique value for any input in the domain; consequently, the algorithm is a process that produces this particular value as output. It is very simple to implement and the space analyzed does not have a huge size to require a very complex algorithm to calculate the path.

Probabilistic Algorithms On the contrary, it uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the average case over all possible choices of random bits. Probabilistic path planners provide solutions to problems involving vast, high-dimensional configuration spaces that would be intractable using deterministic approaches [47]. The downside of these methods is that they are only probabilistically complete, i.e., the probability of finding a solution to the planning problem when one exists tends to 1 as the execution time tends to infinity. This means that, if no solution exists, the algorithm will run indefinitely.

1.2.1 Deterministic Algorithms

Dijkstra's Algorithm The first algorithm was developed in 1959 by E. Dijkstra [16], it is used for calculating the shortest path from one node to all other nodes in non-negative weights graph and it's considered a graph search algorithm since, systematically, follows the edges of the graph to visit or discover its vertices in order to find the goal node through a minimum cost path. The basic idea is to divide all of the nodes in the weighted graph into two groups: the first one includes the nodes with the shortest path determined, while the second one includes nodes not already explored, then, the shortest path is not determined. Gradually adding the nodes of the second group into first group one by one, according to the order of the increasing length of path, until all the vertices are added to the first group, algorithm is ended.

A* Algorithm Taking advantage of Dijkstra's algorithm, in 1968, Hart et al. [23] evolved the A* algorithm. As the previous algorithm, the cost of each node is computed considering the motion cost of reaching its position, plus the heuristic node, i.e. the estimated cost to reach the goal node. The new expanding node is, then, chosen as the node with the minimum cost, reducing hence the number of processed cells and finding an optimal path as long as the heuristic is admissible. A drawback of A* is that, on a large map, thousands of states might be stored, which can require a lot of memory and cause huge computation time.

D* Algorithm Several modification of A* algorithm have been proposed during this period, one of these is the Dynamic A*, or D* algorithm [49], developed by Stentz in 1994. The algorithm behaves like A* expect that the arc costs can change as the algorithm runs, indeed it's a sensor-based algorithm that changes its edge's weights to form a temporal map, since it is able to work in changing environments and to make re-planning in real time.
In contrast to A*, which follows the path from start to finish, D* begins by

searching backwards from the goal node. Each expanded node has a back-pointer which refers to the next node leading to the target, and each node knows the exact cost to the target. When the start node is the next node to be expanded, the algorithm is done, and the path to the goal can be found by simply following the back-pointers. Thanks to this feature, when a robot observes new map information, such as an unknown obstacles, the algorithm adds the information to its map and, if necessary, re-plans a new shortest path from its current coordinates to the given goal coordinates.

Focussed D and D* Lite Algorithms Stentz, year later, released a modification called Focussed D* algorithm [50]; here waves from a new obstacle propagates only through the cells which are significant and important for the planned path. This is achieved by penalization of heuristic values by the distance of the cell from actual position of the robot. With this consideration, robot's sensors perceive new obstacles only in small distances, adding in this way savings in computational performance and efficiency.
Another version of the previous D* is the D* Lite [38], it's not based on the original D* but implements the same behavior. Depending on cases, it's good as or better than the Focused D*, but in general it's simpler to understand and can be implemented in fewer lines of code.

Theta* Algorithm Coming back to extensions of the A* algorithm, Nash et al. developed the Theta* algorithm [36].The extension resides in the test if the neighborhood cells of actually tested cell have direct visibility to the previously tested cell. If yes, the actually tested cell is ignored in fact, so in this way only these cells are found which the robot has to pass and in which the robots orientation changes.

Phi* and Incremental Phi* Algorithms Evolutions of the previous algorithm can represented by the Phi* algorithm, that can be used only with metric C-space and records also a local predecessor of each evaluated cell, and by the Incremental Phi* algorithm, which is capable to find new path during the new obstacle occurrence without the need of the re-planning on the whole environment saving the computation time, unfortunately, only in cases of small map [17]. Both Theta*, Phi* and Incremental Phi* don't present, hence, for their features, the limitation of diagonal movement of the robot, as for A* or D*; on the contrary, they allow the turns in the path to have any angle, increasing the efficiency of the path by doing so.

RA* Algorithm The last but not the least is an algorithm created by Guglieri et al. [21], called RA* algorithm, which is able to minimize the probability of occurring in catastrophic failures. It presents the classic A* cost function increased by an additional term that depends on the density distribution of the overflown area and takes into account a risk analysis evaluation computed through normative frameworks.

Floyd-Warshall Another deterministic algorithm of importance for path planning is the Floyd-Warshall algorithm, developed in 1962 [19]. It's a shortest path of multi-source algorithm which can solve the shortest path between any two points in weights graph which can be a negative weights graph. The core of Floyd algorithm is to check whether there is a vertex, which makes the distance from u to w and then to v, is shorter than the known path of vertices u and v. If there exist the vertex, then it updates the distance of u and v with the distance from u to w and then to v ; otherwise, the distance of u and v keep its original value. Compared to Dijkstra's algorithm it's much easier to code and all pairs of shortest paths are solved, but, on the other hand, it's a little bit harder to understand and its run time is higher than Dijkstra's [24].

Artificial Potential Field Algorithm Several developments are made on another deterministic algorithm called Artificial potential field (APF), firstly by Khalib [29] and then by Rimon et al. [46]. It's part of reactive algorithms, indeed the concept of attraction and repulsion are employed to approach to the goal and evade obstacles, respectively. More specifically, the method assigns a force field value to each location in the environment, which is a combination of an attractive force that increases towards the goal, and a repulsive force that increases towards the obstacles. Each obstacle is modeled with a built-in repulsive force that prevents the robot from bumping into it. These algorithms are set up to build a cognitive model for path finding from sensory information without prior acquaintance with the environment [22]. It has the characteristic of low computational complexity and consequently of a good speed of execution, but, however, it's not optimal (it's not guaranteed that best solution is found) nor complete (often it generates paths that get trapped on local minima of the potential function). For these reasons there are many studies that try to implement other methods to find out these problems.

Methods	Advantages	Disadvantages
A*	It ensures shortest paths, finding an optimal path, thanks to the heuristic information	Memory and computation time increases exponentially with the complexity of the map
D*	It behaves like A*, but is able to work in changing environments and to make re-planning in real time	Reaction time to obstacle appearance increases exponentially with the complexity of the map
Theta*	Better path efficiency than A*, since it doesn't present the limitation of diagonal movement of the robot	Very slow computational speed increased with the complexity of the map
Incremental Phi*	Better path efficiency thanks to the any-angle movement; it can perform also a fast re-planning of map's portions	Very slow computational speed increased with the complexity of the map
Floyd	Much easier to code than Dijkstra's and all pairs of shortest paths are solved	A little bit harder to understand and its run time is higher than Dijkstra's
APF	Low computational complexity gives, consequently, a good speed of execution	Its solution is not optimal nor complete, so it often generates paths that get trapped on local minima of potential function

Table 1.1: Pros and cons of common Deterministic algorithms

1.2.2 Probabilistic Algorithms

Instead of using an explicit representation of the environment, probabilistic algorithms rely on a collision checking module, providing information about feasibility of candidate trajectories, and connect a set of points sampled from the obstacle-free space in order to build a graph of feasible trajectories. Even though these algorithms are often not complete, they provide probabilistic completeness, i.e., it guarantees that the probability that the planner fails to return a solution, if one exists, decays to zero as the number of samples approaches infinity.

PRM Algorithm One of the main and early algorithms of this category is the Probabilistic Roadmap, also called PRM, created by Kavraki et al. [27] in 1996. In its basic version, it consists of a pre-processing phase, in which a roadmap, which represents a rich set of collision-free trajectories, is constructed by attempting connections among n randomly-sampled points, and then a query phase, in which shortest paths connecting initial and final conditions through the roadmap are sought. The PRM algorithm has been reported to perform well in high-dimensional state spaces. Even though multiple-query methods are valuable in highly structured environments, most on-line planning problems do not require multiple queries, since, for instance, the robot moves from one environment to another, or the environment is not known a priori. Moreover, in some applications, computing a roadmap a priori may be computationally challenging or even infeasible.

PRM* Algorithm In order to address the limitations of sampling-based path planning algorithms available in the literature, new algorithms are proposed and proven to be probabilistically complete, asymptotically optimal, and computationally efficient [26]. Of these, PRM*, a variant of PRM, is a batch variable-radius PRM, applicable to multiple-query problems, in which the radius is scaled with the number of samples in a way that provably ensures both asymptotic optimality and computational efficiency.

RRG Algorithm Rapidly-exploring random graphs, called also RRG, advanced from the RRT algorithm explained later, is an incremental algorithm that builds a connected roadmap, providing similar performance to PRM* in a single-query setting, and in an anytime fashion; i.e., a first solution is provided quickly, and monotonically improved if more computation time is available.
Furthermore, these last two algorithms along with RRT*, which will be shown later, have been modified [25] in order to incorporate kinematic and dynamic constraints to the solution, creating a more feasible one.

Bug and Modified Bug Algorithms Another important probabilistic algorithm is the Bug algorithm, developed completely by Rajko et al. [45] in 2001. The standard algorithm consists in generating a direct path from the start position to the goal while surrounding the obstacles always around the same direction and it is very useful when finding a secure path to the goal is more important than arriving quickly. Unfortunately, this algorithm alone cannot yield optimal shortest paths, thus it requires modifications.
One such modification, called Modified Bug algorithm [22], consists in generating a tangent to the obstacle from the starting position and then continue with the normal bug algorithm. To select the side, which is most likely to be the shortest one, the area to each side of the line that defines the direct path from the starting position to the goal is computed. Although it is possible that in some situations the smallest area does not imply the path is the shortest one, computing the area is faster than calculating the perimeter. Despite of some limitations, it shows a good speed of execution and turns out a suitable method for dynamic path planning applications and should be a good choice for many applications requiring fast path plan updates.

Ant Colony Optimization Algorithm In probabilistic algorithms, it can be considered also a part of the evolutionary algorithms, which use mechanisms inspired by biological evolution [54] and solve the optimization problem through a certain fitness function that determines the quality of the solution. One of this algorithms is called Ant colony optimization algorithm (ACO), developed firstly by Maniezzo in 1992 [34], that searches for optimal path in the graph based on behavior of ants seeking a path between their colony and source of food. At first, the ants wander randomly, then, when an ant finds a source of food, it walks back to the colony leaving "markers" (pheromones) that show the path has food. When other ants come across the markers, they are likely to follow the path with a certain probability. If they do, successively they populate the path with their own markers as they bring the food back. As more ants find the path, it gets stronger and because the ants drop pheromones every time they bring food, shorter paths are more likely to be stronger, hence optimizing the "solution".
As advantages, it has positive feedback accounts for rapid discovery of good solutions and can be used in dynamic applications. On the contrary, as drawbacks, its theoretical analysis is difficult, time to converge is uncertain, even if the convergence is guaranteed, and local minima have reached quite easily.

RRT Algorithm One of the most powerful, or even the most influential with its extensions, is the Rapidly-exploring random trees (or RRT), developed by LaValle [31] in 1998. It's primarily aimed at single-query applications, its trees are constructed incrementally from samples drawn randomly from the

search space and are inherently biased to grow towards large unsearched areas of the problem.
In its basic form, as each sample is drawn, a connection is attempted between it and the nearest state in the tree. If the connection is feasible, that is when it obeys any constraints, the new state is added to the tree. Starting with a graph that includes the initial state as its single vertex and no edges, the iteration is stopped as soon as the tree contains a node in the goal region. Several can be advantages of this approach, also compared to some of other algorithms presented before. It's relative simple and very easy to implement, probabilistically complete under very general conditions, it always remains connected, even though the number of edges is minimal and the entire path planning can be constructed without requiring the ability to steer the system between 2 prescribed states, which greatly broadens the applicability of RRTs. As the PRM, it performs much better and is faster than some deterministic algorithm, like A*, there is still an exponential growth, in the work load when the planning problem become more complex [12].
This approach ensures that the planner does not get stuck in small local minima (as Ant colony or APF), which in turn makes the global planning much easier. Though, there is a downside: this procedure causes different and unfortunate computationally expensive solution. It means that the algorithm presents long tail in computation time distribution, since it has to save the entire tree, which increases with the time of execution, an unknown rate of convergence and, finally, it doesn't compute the optimal solution but only a valid path from start to goal [32].

RRT* Algorithm During last two decades many extensions of the RRT algorithm have been presented in order to reduce its drawbacks. As in the case of the PRM algorithm, at first, a great deal of work has been done on the optimality of the solution and a so called RRT* algorithm has been developed by Karaman et al. [25] in 2010.
RRT* inherits all the properties of RRT and works similar to RRT, but it introduced two promising features called near neighbor search and rewiring tree operations. Near neighbor operations finds the best parent node for the new node before its insertion in tree. Rewiring operation rebuilds the tree within this radius of area k to maintain the tree with minimal cost between tree connections. As the number of iterations increase, RRT* improves its path cost gradually due to its asymptotic quality, whereas RRT does not improves its jaggy and suboptimal path [37].
However, these operations have an efficiency trade-off; it improved path quality, reaching minimum-cost path, and obtained asymptotic optimality at the cost of execution time and slower path convergence rate of asymptotical optimality, especially in large environments.

RRT$^{\#}$ Algorithm After RRT*, working on better convergence rate, a new sampling-based motion planning algorithm, called RRT$^{\#}$, has been proposed [11] in 2015. One of the main reasons of the slow convergence of asymptotically optimal algorithms is the lack of a good exploration strategy. Although every collected sample gives some information about the topology of the search space, it may not necessarily contribute to improving the cost of the solution of a given query. Since the exploration task is one of main computational bottlenecks of sampling-based motion planners, it is preferable to select samples that maximize the improvement of the quality of the solution of a given query. One way to achieve this is to guide exploration to the region of the search space that is relevant to the current query and to adjust the sampling strategy to draw more samples from this relevant region.
RRT$^{\#}$ computes successively tighter approximations of the relevant region of the search space as the number of samples tend to infinity as a by-product of the exploitation step. As shown in [10] the RRT$^{\#}$ algorithm has a faster convergence rate and provides solutions with smaller variance compared to the RRT* algorithm. The RRT$^{\#}$ algorithm also provides a good characterization of the computed information during the search, by identifying the region of the search space, which is highly likely to contain the optimal solution.

RRT^{X} Algorithm Concluded a good resolution about the convergence rate, studies have moved to enhance solutions in dynamic environments, so the RRT^{X} have been created by Otte et al. [39] in 2015. The main differences between RRT^{X} and previous sampling-based motion planning algorithms is that RRT^{X} is a re-planning algorithm that is also asymptotically optimal. It enables real-time kinodynamic navigation in environments with obstacles that unpredictably appear, move and vanish. Whenever obstacle changes are detected, rewiring operations cascade down the affected branches of the tree in order to repair the graph and remodel the shortest-path tree. The expected runtime is achieved, despite rewiring cascades, by using two new graph rewiring strategies: 1) rewiring cascades are aborted once the graph becomes epsilon-consistent, for a predefined epsilon > 0; 2) graph connectivity information is maintained in local neighbor sets stored at each node, and the usual edge symmetry is allowed to be broken. These features significantly decrease reaction time without hindering asymptotic convergence to the optimal solution. Indeed, reaction time is the most important metric for a re-planner in dynamic environments.
From a graph construction/maintenance point-of-view, the most closely related work to our own is RRT$^{\#}$, which is the only other sampling based algorithm that uses a rewiring cascade; RRT$^{\#}$ uses such a cascade after the cost-to-goal of an old node is decreased by the addition of a new node. However, RRT$^{\#}$ cannot handle unforeseen obstacle configuration changes; in particular, obstacle

appearances break the algorithm. In contrast, RRT^X is designed specifically to handle obstacle configuration changes – including those that both increase and decrease the cost at old nodes. Furthermore, although RRT^X is designed for dynamic environments, it is also competitive in static environments where it is asymptotically optimal and has an expected amortized per iteration runtime similar to RRT and RRT* and faster than RRT$^\#$.

Last two evaluated algorithms are both extensions of the RRT* algorithm: the Transition-based RRT* (or T-RRT*) algorithm and the Real-time RRT* (RT-RRT*) algorithm.

T-RRT* Algorithm The first one, developed by Devaurs et al. [15] in 2016, consists of integrating the transition test of T-RRT into RRT*. Thanks to the filtering properties of this transition test, the exploration performed by T-RRT favors low-cost regions of the space. In fact, T-RRT mostly creates new nodes in these favorable areas. Nevertheless, a drawback of T-RRT is that it cannot take a path-quality criterion into account when creating edges and, thus, involves no mechanism to allow for an improvement of the quality of the current solution path. As a consequence, it offers no optimality guarantee.
On the other hand, RRT* is not very efficient because it does not take the configuration-cost function into account when sampling the cost space and creating new nodes. Indeed, RRT* only takes a path-quality criterion into account, when creating or removing edges. As a possible consequence, it has been observed that RRT* may converge slowly toward the optimal solution-path in high-dimensional cost-spaces.
Thanks to this mash up, T-RRT* is able to explore more low-cost regions of the space, meanwhile maintaining the asymptotic optimality property of RRT*. Results presented on several classes of problems show that it converges faster than RRT* toward the optimal path, especially when the topology of the search space is complex and/or when its dimensionality is high.

RT-RRT* Algorithm The second algorithm, advanced by Naderi et al. [35] in 2015, consists in an RRT* variation enable to work in dynamic environment through an on-line rewiring. Actually, there are two modes of rewiring for having shorter paths in large tree with a limited number of the nodes. The first one, rewiring starting from the root, creates a growing circle centered at the agent. In this process every node inside the circle is rewired, and this circle most frequently rewires nodes around the agent and thus the tree root. The second one, rewiring random parts of the tree, is done using both focused and uniform sampling, but in patches instead of just one node. However, RT-RRT*

has its limitations; it requires a large memory capacity because the whole tree is stored at all times and it only works in a bounded environment. The focused sampling inside an ellipsoid works somewhat in an unrestricted environment but the rewiring suffers if the distances are large. Thus, the challenges of unbounded and large distance environments remain to be addressed.

After the analysis of the different algorithms presented in the above section, recalling that our area of interest will consider the dynamic environment and the research of a feasible solution, minimizing the cost function, RRT* and RRT^X algorithms have been chosen and adapted for our path planning problem in order to show which is the best for replanning issues.

Methods	Advantages	Disadvantages
Ant colony	It has positive feedback accounts for rapid discovery of good solutions and can be used in dynamic applications	Time to converge is uncertain, even if the convergence is guaranteed, and local minima have reached quite easily
Modified Bug	It shows a good speed of execution and should be a good choice for applications requiring fast path plan updates	It has to force to increase the radius of the obstacles to avoid the collisions
PRM	Even if it's based on the Dijkstra's algorithm, it performs well in high- dimensional state spaces	It doesn't provide optimal solutions and if the number of iterations is not high enough, a feasible path may not be found
RRT	It's relative simple and very easy to implement, probabilistically complete under very general and complex conditions	Long tail in computation time distribution which increases with the time of execution and an unknown rate of convergence
RRT*	Thanks to its asymptotic quality, as the number of iterations increase, it improves cost and quality of the path	Execution time and slower path convergence rate of asymptotical optimality, especially in large environments.
RRT^X	Able to handle obstacle configuration changes, maintaining the asymptotical optimality	Computational cost of the algorithm could be very high with the extension of the environment

Table 1.2: Pros and cons of common Probabilistic algorithms

1.3 Thesis Structure

First of all, in Chapter 2, an overview on the traffic management system will be introduced through reference of ROS environment, cloud robotics and risk evaluation to explain better the following overall architecture and the high-level operating principle of CBUTM's main components, showing the behavior of the whole system, taking into account also the contribution given by my colleagues. Afterwards, in next Chapter 3, the main purpose of this thesis will be exposed. In Section 3.1, known as Offline Planning, basic principles of both algorithms will be illustrated, focused mainly on the static approach, beeing this section related only to the computation of the global optimal path from initial to target state. On the other hand, in Section 3.2, known as Online Planning, it will be presented the dynamical update of risk-cost and the concerning proper development of the algorithms in order to adapt the off-line path to alterations in the dynamic risk-map. The Chapter 4 will be precisely organized to verify the goodness of the proposed method by means of simulations results and finally, in Chapter 5, comments on tests will be drawn with a look to hopeful future works.

Chapter 2

Basics and Requirements

Performing "dull, dirty or dangerous" [53] missions without involving any kind of risk for the pilot, as we said above, has been one of the great competitive advantages for increasing interest in UAVs in both the public and private sectors. However, having a world, where robots (i.e. drones) and humans can cooperate in a well structured system, implies a growing duty in the handling of these fully autonomous devices during their flights and, consequently, a redefinition of traffic management.
Known as Unmanned Aircraft System Traffic Management (UTM), the goal is to create a system that can integrate drones safely and efficiently into air traffic that is already flying in low-altitude airspace, especially in urban environment. That way, any kind of drones' missions (starting from package delivery or fun flights to saving live) won't interfere with helicopters, nearby airports, and won't produce higher risks for people on ground.
The system will be a bit different than the Air Traffic Management control system (ATM) used by the Federal Aviation Administration (FAA) for today's commercial airplanes. UTM will be based on digital sharing of each user's planned flight details. Each user will have the same situational awareness of airspace, unlike what happens in today's air traffic control. FAA, NASA and other federal partner agencies are collaboratively conducting researches such that UTM development will ultimately identify services, roles/responsibilities, information architecture, data exchange protocols, software functions, infrastructure, and performance requirements for enabling the management of low-altitude uncontrolled UAS operations [9].
According to the International Civil Aviation (ICAO), an UTM framework will include many components, three of which are fundamental:

- Registration System to allow remote identification and tracking of each UAS
- Communications Systems
- Geofencing-like system

Therefore, before presenting how we build a completely new traffic manager (the above-mentioned CBUTM), we will briefly review some features, like the working environment, the cloud and the evaluation of risk cost that the framework requires to be devised.

2.1 ROS

The recent diffusion of robotic applications, as well as allowing an increasingly specific and customized use in numerous fields, has guaranteed a radical drop in production and management of the hardware industry. However, a comparable development of the software part has been required in order to support resources needed to deal with complex and robust systems behavior. And that's where ROS environment comes in.

> Robot Operating System (ROS) is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. [5]

It's similar in some respects to a "robot frameworks", but provides a distributed architecture that involves the state of art algorithms, tools, libraries and conventions implemented and updated by the community. ROS implements several different styles of communication, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a parameter server. It easily lets several applications cooperate and interact both in the same machine and in different ones, even far one from the other.
Taking to account that our project works in very dynamic environment, being a Non-Real-Time operating system can be a limitation. The inability of assure the respect of deadlines may imply waste of fundamental seconds. This consideration must be taken into account during the development process, building light algorithms up, not requiring heavy computational efforts to be performed, and continuously check the connection integrity and data losses.
Collaboration and open-source license sharing represent the real improvement with respect to competitors; these factors have encouraged the robotics software development but even a mechanism in which each member can share its issues but, at the same time, feels responsible to intervene and overcome the others' obstacles through simple contribute of knowledge's exchange, acting as a sincere community without valuing degrees or qualifications.

Software in ROS is organized in packages. A package might contain ROS executables, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module. The goal of these packages aims to provide this useful functionality in an easy-to-consume manner so that software can be easily reused.
Let's now explain how it works, its main elements and concepts: [5],[1]:

Master It's the core of system, provides naming and registration services to the rest of the nodes in the ROS system, tracking publishers and subscribers to topics as well as services and lookup to the rest of the computation graph.
Without the Master, nodes would not be able to find each other, exchange messages, or invoke services; indeed it enables individual ROS nodes to locate one another within the network. Once these nodes have located each other they communicate with each other peer-to-peer.

Nodes Nodes are processes that perform computations, independent executables that act, literally, as nodes of a complex network, able to talk and interact with other processes through topics, services or parameter server. Their isolated feature reduces code complexity and, at the same time, gets system robuster and more organized. Thanks to ROS client libraries, they can be coded both in C++ and Python.

Topics Topics are buses used by nodes to transmit data in unidirectional way, publishing data on it and subscribing to it for reading the message. They have anonymous publish/subscribe semantics, which decouples the production of information from its consumption, in fact, each topic can have multiple publishers and subscribers.
Topics are deeply linked to the ROS type of the message they transmit. If a node publishes a message of a specific type, other nodes can subscribe to it only if they have the same message type. Actually the communication protocols are based on TCP/IP and UDP.

Services Services are used to answer a very common requirement of distributed systems, which is the request/reply interaction. Differently from the publish/-subscribe model, a service deals with situations in which the communication can not be unidirectional, one node requires a function provided by another one, so the first one needs a reply from the second one.
It's defined by a pair of messages: one for the request and one for the reply.
Practically, ROS node offers a service under a string name, a client calls the service by sending the request message and waiting the reply, thus these calls are blocking because a node cannot go on until a reply appears. Like topics, services have an associated service type that is the package resource name of the .srv file.

Parameters Parameters are another mechanism to catch information to nodes, they represent a great tool of customizing, changing or adapting the behavior of a ROS application. Their usage requires a central system, called Parameter Server, which keeps track of a collection of values. Nodes use this server to store and bring back parameters, at runtime, if they are interested in their value. Through a certain parameter service, the value can be changed by the developer or other nodes. They are not designed for high-performances, in fact, this communication method is more suitable for information that will not vary to much over time.

Actions In any large ROS based system, there are cases when someone would like to send a request to a node to perform some task, and also receive a reply to the request. This can currently be achieved via ROS services. In some cases, however, if the service takes a long time to execute, the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing. Actions, and especially the *actionlib* package, provides tools to create servers that execute long-running goals that can be preempted. It also provides a client interface in order to send requests to the server.

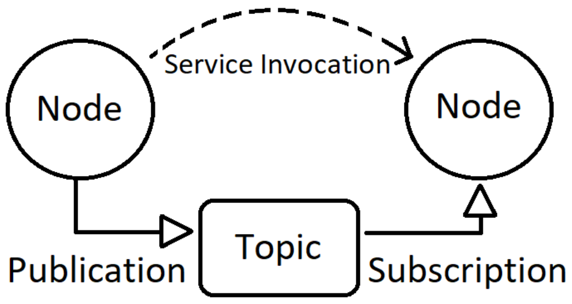


Figure 2.1: Node and topics architecture of a ROS environment

The working principle of the Robot Operating System itself, its architecture is shown in Figure 2.1, running independent nodes and talking thanks to topics and services, seems to be suitable for explaining how the CBUTM system works. It can be seen, in fact, as the cooperation of 4 main nodes, each one belongs to one of four main subjects of the whole project and, consequently, to one of four colleagues within the Joint Open Lab.

2.2 Cloud Robotics

James Kuffner, one of the most brilliant researcher in robotics, coined in the 2010 the term Cloud Robotics. According to him:[20]

> The Cloud Robotics is a new paradigm in which robots and automatic systems exchanges information and execute computations using a common on line network.

The existence of such a cloud, direct consequence of the progresses in IoT (Internet Of Things) and machine learning, brings unlimited improvements, opening to thousands of utilizations.
The main benefits of such a system are:[28]

Big Data Big Data represents a massive amount of data (both structured and non-structured), a huge library of images, maps and data collected and managed to extrapolate knowledge, that every robot connects to.
The concept is clearly linked to the cloud computing. Having such a big amount of data allows to generate a real knowledge of the system one wants to control or interact with. Indeed, system's knowledge, ability to observe its states and future prediction are fundamental factors to improve th efficiency of a controller.

Cloud computing Cloud Computing is a high computational power that, used through Internet, allows to design and to use complex (and demanding) algorithms on multiple robots at the same time.
The idea is to exploit the cooperation of different machines to overcome the computational power of single one. As result, the cloud produces the above-mentioned high computational power while the robot can perform the physical job.

Collective Robot Learning The most efficient way to collect and transfer such an amount of data is putting together in separate sources, making them cooperate and share in a well structured network. Knowledges are instantly and indirectly moved from one element of the network to all others that are connected to it; so that one can learn from the data coming from another and vice versa.

At this point, it's more palpable the choice of a cloud-based framework, as our purpose, to realize our traffic management system and, more generally, to achieve fully autonomous flights. Through machine learning and Iot, at one side, exchanging data, informations and maps allows a faster evaluation of the risk in real time.
On the other hand, the study of many path planning and collision avoidance algorithms has shown the need of a very high computational power, that only a cloud

framework can provide. Each UAV connected to the network must be able to transmit/receive to/from the cloud, updating the common shared information if its sensors perceive something different from what the cloud expected, contributing to more accurate results both for risk analysis and path planning research.
For this reasons, the ideas and results that will be presented in the rest of this work have been developed supposing to use a cloud network.

Please notice that we were forced to make some simplifying hypothesis about the cloud in order to build our framework up. Nowadays, several limitations can show up in this procedure but, mainly, they come from the limited power of on-board computers and the wireless connectivity to the cloud. Although assumptions could be strong, we will analyze the system with no latency or package loss during the drones' flight.
Future work will be precisely dedicated to assimilating new algorithms that can manage and improve on-board UAVs controller performance.

2.3 Risk

After introducing the concept of safety for people on ground as the central concern of our proposal, it's proper time to find out how population security is parameterized and included in our algorithms and calculation, that is by means of the concept of risk.
Especially, my thesis has the precise aim to answer path planning and re-planning problem being risk-aware, in fact, one of the path planning algorithm's input, useful to delineate all the space where algorithms search for solution path, is properly the risk-map. As we said, it's a location-based map and each cell is linked to specific position and specific risk-cost value, where the latter is a scalar that describes people's risk when UAV flies over a given cell.
Therefore, only a brief overview will be described in this work without going into broad details on the precise construction technique of the risk-map.

2.3.1 Unmanned Mission's risk

First of all, looking only at vehicles as UAVs, the term risk is referred to the time frequency in which the drone causes fatal injuries to people on the ground [14]. It is important to highlight that, as damages for people, we mean both physical and social damages. A high frequency of accidents, although not lethal, can affect the common perception of such operations and therefore potentially limit their use.

Depending on accuracy's grade, during last years, several ways to modeling the risk have been introduced but they all are based on four main standard criteria [13]:

- Transparency
- Consistency
- Clarity
- Reasonability

The target of this section is to discover whenever a UAS is able to guarantee a certain level of safety flying over a given geographical zone in a given time window. The *risk management*, according to [13], thus has to follow these four steps to enhance the safety level of the mission:

Mission Definition and Hazard Identification In this step the manager has to produce a explanation of the mission and, as a effect, a definition of safety bounds. The latter, since we are working on civil operations in an urban environment, will be referred to safety levels imposed by the national flight authority. Practically, this means that the Flight Agency has to select a maximum rate of victim per hour, which will be used by the Cloud-Based Traffic Manager as upper-bound for every mission.
Moreover, every possible hazard that could happen during each mission have to be identified.

Risk Assessment According to a specific metric, the system assigns to every hazard a corresponding risk value.

Risk Reduction and Management Compare the imposed bounds with the actual safety level in order to verify matching with the requirements and computation of countermeasures, if they are needed.

Risk Acceptance Once the risk satisfy the requirements, the mission is approved and can starts.

About these steps, there is another description that we have to do regarding different kind of hazards: due to internal failures and due to external causes. It is important to notice that an hazard analysis of internal possible failure of vehicles is out the aims of the whole project, also because there are already many tools to perform it.

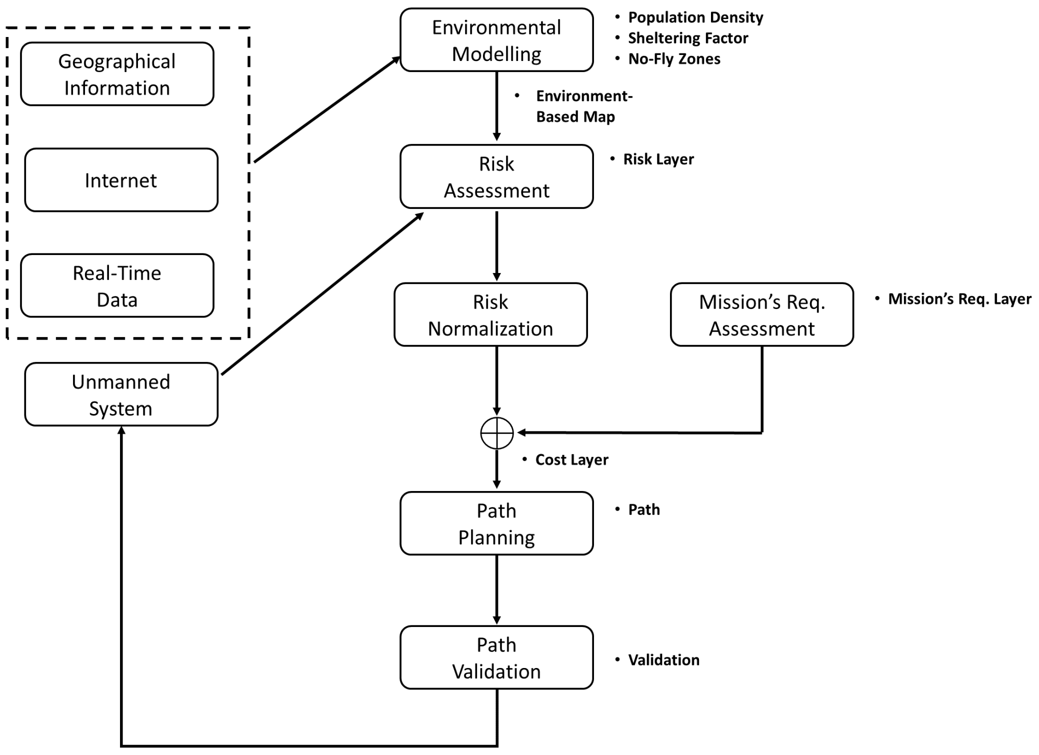


Figure 2.2: Risk-map construction and path planning flow-chart

Accordingly, fundamental hazards, that may affect unmanned missions, can be categorized into [14]:

Drone Involuntary Mobility It's referred to all the accidents that could happen when the drone is not operating, i.e when the vehicle is on the ground and still have to take off.
Applying correctly all the security protocols for the drone managing proves to be better way to avoid fatal injuries in this case.

Mid-Air Collisions It examines accidents due to flight collision, and may concern two or more UAS or fixed obstacles, such as buildings.
Generally the kind of analysis interests line flights and belongs to the "flight's victims" involved in the crash. Although, since we are treating with unmanned vehicles, we can easily keep big airplanes crashing out from the examination, so the analysis of mid-air collision will consider only the damage for people on the ground due to debris or falling vehicles.

Early Flight Termination It concerns all the hazards due to a loss of control of the UAS's flight and the relative anticipation of its landing.

It's interesting to notice that the cloud could control this process through imposing a desired landing zone or imposing a specific velocity in order to reduce fatality odds.

Although there is not yet a commonly accepted definition of risk, we can define it considering a widely held concept in the field of traditional avionics:

Definition 2.1. It is called risk f_F of an Unmanned Aerial Mission the frequency of fatalities, in term of victims per hour of flight, that a given drone, in a certain area will produce.

At this point, ensuring a certain level of safety for a mission it's nothing different from ensuring the risk value always below the upper-bound of victims per hour imposed by the local authorities. Then, it's inherent that, referring to the last definition, risk is linked to the concept of time.
Currently, the best way to assess the value of risk is based on statistical considerations [14]. In order to avoid misunderstanding, we want to remember that our target does not concern a hazard analysis on the used components, but, assuming known the ground impact frequency, we are focused on providing a device able to manage this information for guaranteeing the safety of the missions.

Starting from this, it's possible to increase the typical risk lower-bound, which is $10^{-7}h^{-1}$ for classical manned aviations, to the maximum fatalities rate for a civil operation in an urban context in the range of:

$$f_{F,Max} \in [10^{-6}, 10^{-5}] victims/hour \tag{2.1}$$

that means one victim at most each 10^6 hours of flight. Speaking of maximum, the value of 10^{-6} victims/hour have been selected as the upper bound for the development of the simulations in this thesis.
In order to reach an analytical expression of this fatalities' frequency, we start from analyzing the risk due to a generic uncontrolled landing of the UAV: $f_{F,UFE}$.
It underlines the usage of the term "generic" because the event could be due to both externals and internal failures or both mid-air collision and early flight termination.
According to [14], a proper analytical expression could be:

$$f_{F,UFE} = N_{exp} \times P(fatality|exposure) \times f_{UFE} \tag{2.2}$$

Where:

$f_{F,UFE}$ = Fatalities's frequency due to unexpected flight end - $[\frac{people}{h}]$

N_{exp} = Number of people exposed to the accident - $[people]$

$P(fatality|exposure)$ = Probability that a person involved in the UAS's crash will suffer fatal injuries

f_{UFE} = Frequency of failures that cause an unexpected end of the flight - $[\frac{1}{h}]$

About the Equation 2.2, $P(fatality|exposure)$ takes into account the kinetic energy of the drone, the geographical sheltering factor and the vulnerability of the human body to obtain the probability that the collision between an unmanned vehicle and a person is lethal. For what concerns N_{exp}, it can be evaluated as $N_{exp} = \rho \times A_{exp}$, where ρ is the population density and A_{exp} is the area involved in the impact. Last words on f_{UFE}, which is evaluated thanks to a statistical approach and contains informations about the frequency of the UAS's ground impact: in practice, it introduces the time element inside the equation.
Such expression turns out to be a good compromise between a too detailed analysis and a too abstractive one, it contains all the parameters of interest that can affect an unmanned mission. Through the interpretation of these factors the resulting risk analysis will be considered:

- Coherent with the statistical data actually available
- Not expensive from a computational point of view
- Easy to extend in case some new interesting parameters emerges
- Independent from the dimension of the considered area. The analysis can be performed both with very small and big resolution

Coming back to the fatalities's frequency expression, its proportional dependence on the population density ρ shows why this system should know the number of people in danger due to the UAS's flight, and in general, should be a dynamic framework capable to collect data and provide a real time risk evaluation.
The cloud system can connect to internet (thanks to the TIM network) and extract updated data regarding population density of a certain geographical area of interest. However, in this way, only living people are considered into the count, getting not the real location of people during the day. By exploiting the potential of the cloud system at our disposal, ρ can be expressed as a function of time.
Through internet, it could collect informations about possible presence of events and estimate the influx of people looking at the historical series of the participations. Such informations can then be combined with those obtained by monitoring the number of users connected to a specific cell of the mobile communication network, and that's it.

In the Equation 2.2 the term $P(fatality|exposure)$ represents the probability that a person reports serious injuries after the impact with a falling vehicle. To value this rate, models of vulnerability of the human body have been implemented, taking into consideration age, physical conformation and posture taken at the time of impact, or dependence of the speed vehicle.
Making such simplification, at the end, we can assume $P(fatality|exposure)$ as

a function of the kinetic energy of the falling vehicle and of the sheltering factor P_S of the area in which the crash occurs, and Figure 2.3 shows just how $P(fatality|exposure)$ may vary with factors change.

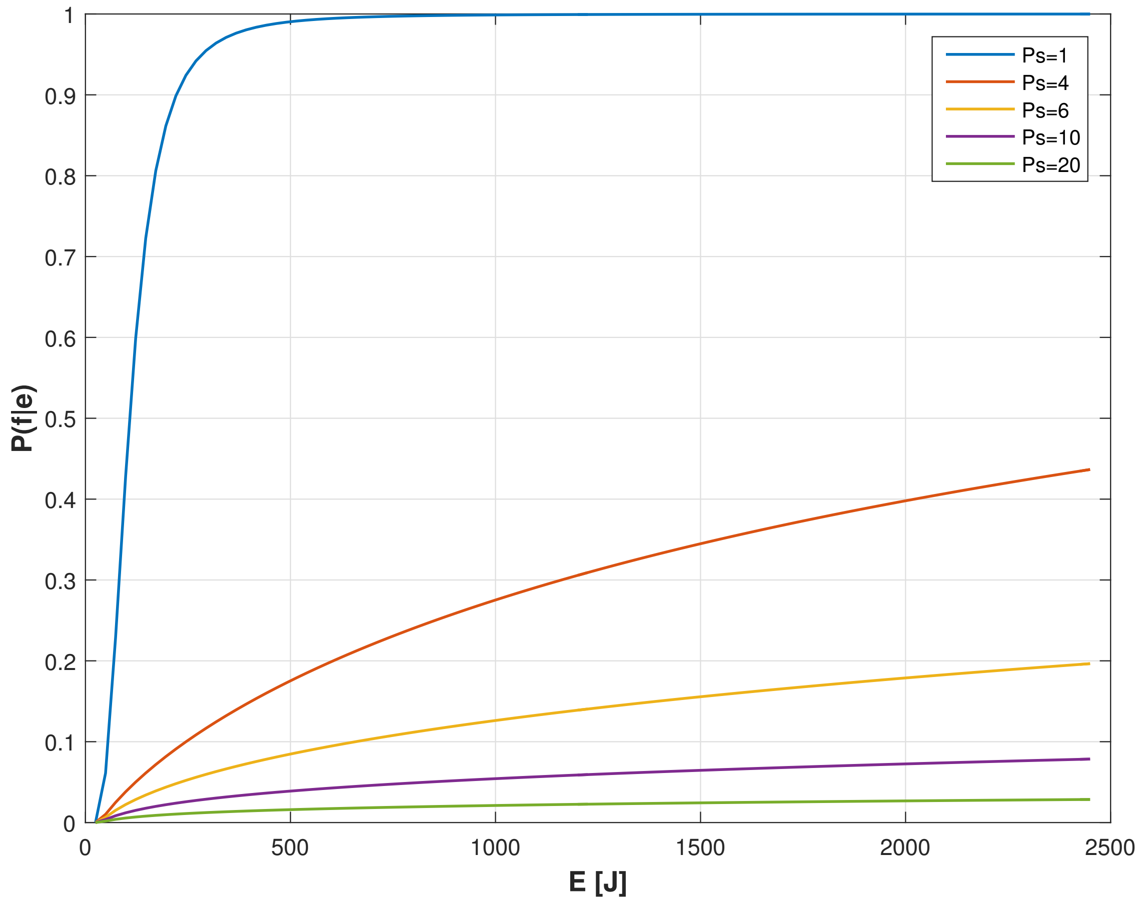


Figure 2.3: $P(fatality|exposure)$ evolution respect to Kinetic Energy and Sheltering

About the sheltering factor, lack of a metric, that coherently describes it, doesn't represent a foolish complication. On the contrary, looking at Figure 2.3, it's clear that $P(fatality|exposure)$ doesn't depend proportionally on P_S and this assumption makes harder to assign a value to the latter factor.

Definition 2.2. The sheltering factor of a geographical area is its capability of protecting people on ground, through artificial or natural structures, from the fall of an Unmanned Aerial System.

After series of empirical tests, colleagues found that a credible upper-bound for sheltering factor is $P_S = 10$. By assigning a value greater than 10 to P_S, 3.4MJ are

required to have a fatal event occurring at 50% probability. Although, due to the small size of drones in urban environment, the maximum kinetic energy released on impact it will be not greater than some KJ.
Going little deeper, P_S is mainly reliant on a crucial factor, the sheltering coefficient C_S of a structure, that measures the amount of drones' kinetic energy reduced by structure itself, once drones hit it.

C_S	Structure's Tipology
0	Free Area
0.25	Shallow and Slightly Leafy Trees
0.5	Tall and Leafy Trees
0.75	Residential Buildings
1	Reinforced Concrete Buildings

Table 2.1: Sheltering Factor definition

Practically, $C_S = 1$ means a building that completely stops the UAV's fall, while $C_S = 0$ will be used for the completely open areas. Once C_S is known, the sheltering factor value can be calculated as:

$$P_S = C_S \times \frac{A_C}{A_{tot}} \times 10 \tag{2.3}$$

Where:

C_S = Sheltering Coefficient

A_C = Covered Area of the zone - $[m^2]$

A_{tot} =Total Area of the zone - $[m^2]$

The multiplicative factor 10 is used to scale P_S, as to make it compatible with previous evaluation of sheltering factor through empirical tests. Moreover, the term $C_S \times \frac{A_C}{A_{tot}}$ evaluates the probability that the drone falls in a covered area times the capacity of the zone itself to reduce its energy.
So, in the continuation of the work, we will be hired $P_S \in [0; 10]$.

2.4 CBUTM

2.4.1 Overview

The proposed Cloud-Based UAVs Traffic Management system, namely CBUTM, is an air traffic management solution exploiting the advantages offered by the Cloud Robotic paradigm. Its final aim is the creation of an organized networked system for managing a well structured low-altitude city airspace, which allows drones operations also in urban environments while preserving a certain level of safety for humans on ground.
The final idea is to provide a unique platform where users, costumers and authorities can meet and cooperate, and where rules, procedures, rights and duties can be clear and well defined.

Basing on what shown in the previous sections, in this one we show the general architecture of CBUTM and the strategies we adopted in order to accomplish our objectives. One key characteristic of CBUTM is that it works on a standardized risk assessment, and so is able to operate assuring the same safety level for each possible mission in the city. The authorities have, at the same time, a unified access to monitor and to intervene on all those parameters that condition the missions' requirements and so the characteristics these missions will have. Metric conservation, despite changing parameters, allows different instances of CBUTM, each one under the control of the respective authorities and, at the same time, a common language (the metric itself) among them. In this way, for example, different cities could use different requirements basing on the characteristics of the local environment, while continuing be inside the parameters' range imposed by the national authorities.
On the other side, a unified portal to access to unmanned missions and to the relative services is offered to the citizenry as well as to private company that would receive huge benefits operating in this field.

As we said, whole structure has been then built up in the ROS environment, managing data streams and connections within the different elements of the network, and setting up some procedures for keeping it organized and operative, able to adapt to actual traffic conditions tracking and managing the active vehicles.
As can be seen from Figure 2.4, our Cloud-based UAVs Traffic Management system can be thought as the cooperation of 4 main entities (so, 4 principal ROS nodes), with specific roles and functionalities, each one main subject of the work of 3 other colleagues within the Joint Open Lab, but in the whole result of the cooperation of each one.

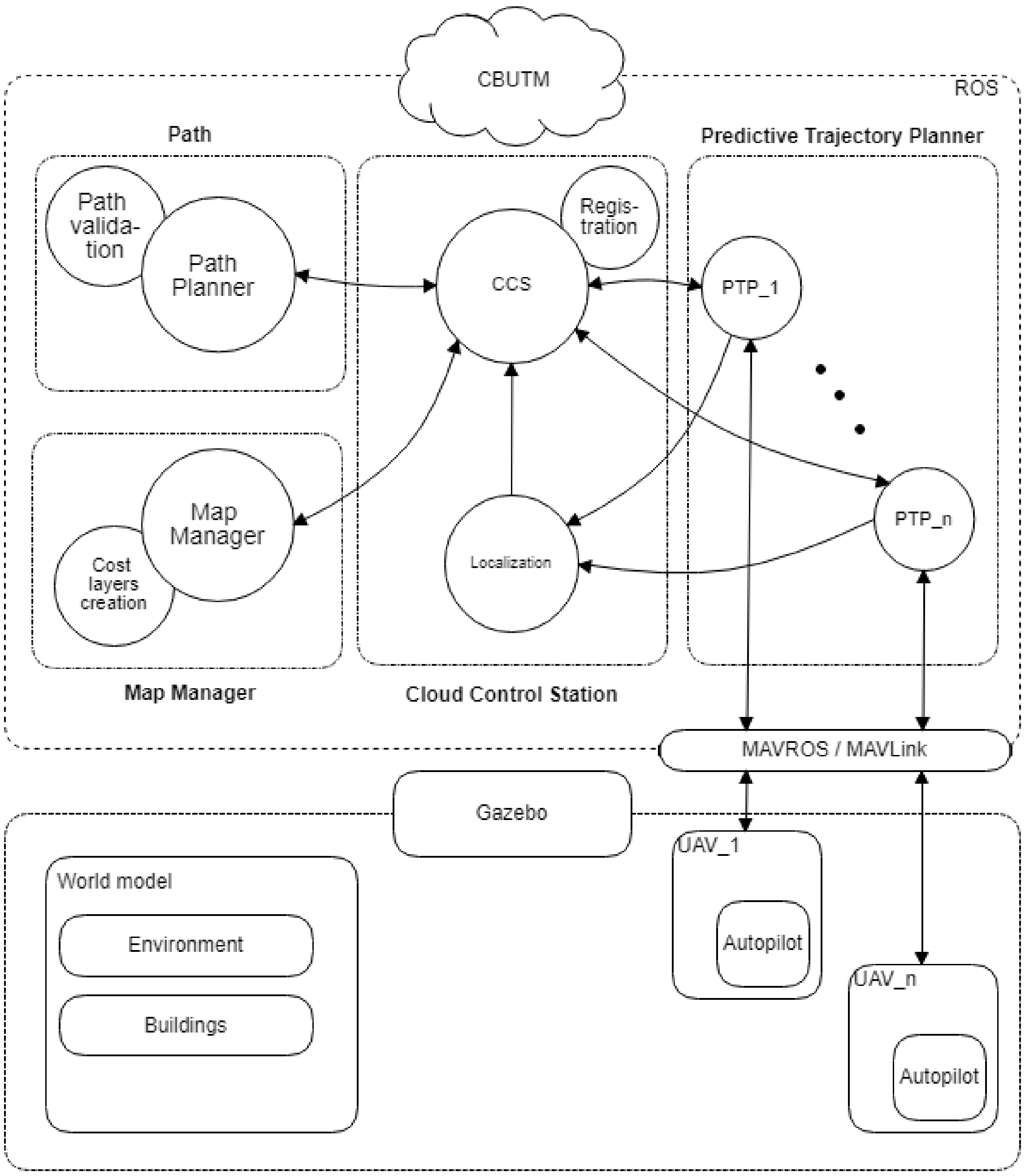


Figure 2.4: Graphical representation of CBUTM working principle

They can be enumerated as:

Map Manager, MM It's aimed at the creation and management of cost layers and risk-map, applying the rules of a standardized risk assessment.

Cloud Control Station, CCS It means to be the equivalent of a Ground Control Station: core of the system, is aimed at catching and collecting informations of all the agents acting within the network, putting in contact different elements and managing data streams.

Path Planner, PP Receiving initial and goal position, it applies a specific path planning algorithm for computing optimal paths on the risk-map generated by the Map Manager.

Predictive Trajectory Planner, PTP A group of similar nodes, images of physical drones flying in the city airspace, aimed at being their in-cloud networked controllers directly linked to the vehicles.

These four entities work in the cloud, interacting with drones flying either in real world or in a Gazebo simulated environment.

Computing optimal paths, minimizing the cost function based on risk cost and dynamic map, remains the critical phase of this thesis but, before entering in details of how path planning algorithms run and how choosing them to accomplish the system's architecture, we have a look to the flow-chart, in Figure 2.5, in order to briefly understand how CBUTM operates. The steps involved in a standard mission process are shown, from the moment of its definition to the goal.
Steps can be summarized as follows [33],[41],[48]:

1. **Registration**: Supposing a user wants to fly his drone in a city's airspace in which CBUTM is working, he should first ask for its drone's cloud image to be created. It means that, for each UAV within the network, a specific ROS node is created within the cloud with a coherent level of priority, according to its needs.
   First step the newly created node will do is calling the registration service, provided by a specific node, the Cloud Control Station. During this process, vehicle's informations (parameters, goals and requirements to fulfill) are collected within specific database structures and, then, analyzed while forwarded to the National Flight Authorities (NFA) in order to verify the compliance between mission's requirements and safety standards. Aim of this service is to set up the structure needed for the drone to be monitored and safely flown within the controlled airspace.
   Remember that this node runs considering different system architecture we could have used in the cloud, so it's directly linked to the real drone by means of the MAVROS/MAVLink bridge we will spoke about in Chapter 4.

2. **Environmental Modeling**: This phase is not triggered by the mission's request but instead happens at a constant rate till the CBUTM stays on-line: it must have, at any time, an update and coherent geographical map of the urban environment where the flights are performed.
   Although not available for now, it is reasonable to suppose that in future municipal offices will share their data on the city structure. Furthermore, this information are going to be merged by the "Environmental Modeling" block with the ones received in real time by the flying drones' sensors: finally, the result is the dynamic map of the city we were looking for. Differently from the rest of the flow chart, this part of the system is independent from the drone that has advanced the request, and remains the same for ever aircraft joining CBUTM.
   Once the map is available, and the registration process has began, we can move to the next step.

3. **Risk Assessment and Map Generation**: Map Manager node provides a special service called by the CCS during the registration service routine, which is performed for every drone.
   Firstly, through the Risk Assessment block, it applies risk modeling techniques to build a point by point map of the risk, that associates at each area of geographical map its corresponding risk value. Then, in the Map Generation block, for each drone, its parameters and characteristics are contained in a specific risk layer and the risk-map is nothing more than the weighted overlap of these (either static or dynamic) layers.
   The output risk-map will result function of geography, drone's parameter and mission standard objectives.

4. **Mission Planner**: Taking as input the risk-map, the Mission Planner has a dual final aim: on one side, it must find the best (lowest cost) path for the drone to follow, on the other hand it must evaluate if this trajectory is compliant with the standard imposed by authorities. This two different souls, called "Path Planner" and "Path Validator", cooperate in this block, since the output of the first is the input of the latter.

   (a) **Path Planner**: Once the drone has been properly inserted in the network, it can proceed asking for its mission. The starting and goal position are then transferred to the Path Planner node, which will use the specific drone risk-map for computing an optimal path, looking for the trade-off between safety and efficiency, with the former being, of course, privileged.

   (b) **Path Validator**: A redundant safety check has been then introduced, aimed to validate the computed path according to the safety requirements imposed by the local NFA. The results of this check can be 2: either the path is accepted as it is, or it is rejected because of unacceptable hazard level. In this case, in turn, the path validator can, if possible, modify some parameters of the risk-map to force the path planner in finding a better solution or it's forced to abort the mission as a whole.

5. **Trajectory Following and Collision Avoidance (TFCA)**: Once the computed path for a mission has been accepted (either at the first attempt or after a re-computation), the drone obtains its authorization to fly, and the mission can start.
   This block is implemented as Predictive Trajectory Planner (PTP) node; it's a system executed alongside every mission and it's responsible to continuously translate a path (the output of the mission planner) into a proper trajectory (a path plus a time law). The generated trajectory has to be equivalent to the received path as long as this doesn't imply that the distance from other vehicles, that clearly depend on time, goes below a safety threshold. In that case, TFCA performs the collision avoidance procedure, producing a trajectory different as much as necessary.

6. **Navigation Management (NM)**: It's the unique system to be on-board and it translates high-level instructions into low-level ones, interacting directly with electromechanical actuators and with other devices on-board, like sensors. TFCA works in parallel with NM, to which it sends instruction and from which it receives a feedback on the actual state.

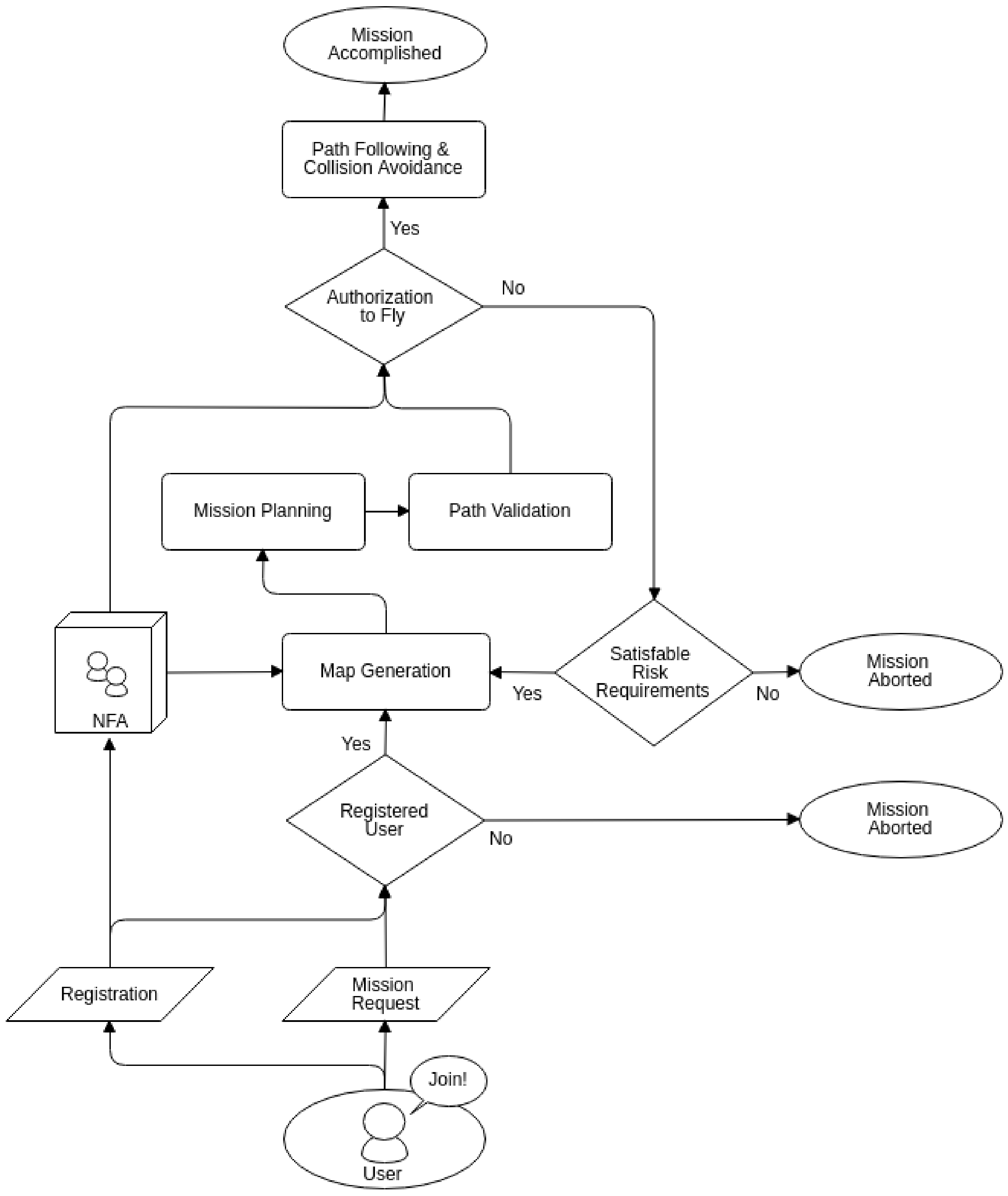


Figure 2.5: CBUTM operational flow chart

Chapter 3

Risk-Aware Path Planning

In this section, the proposed risk-aware path planning method is described. The proposed approach uses a two-phases path planning strategy [42]:

1. **Off-line Path Planning**: Starting from static informations, a optimal global path minimizing the risk to the population on ground.

2. **On-line Path Planning**: Starting from the off-line path, the on-line path planner adapts the path in order to comply with dynamic environment changes, still maintaining the optimal solution.

The architecture of the proposed approach is illustrated in Figure 3.1.

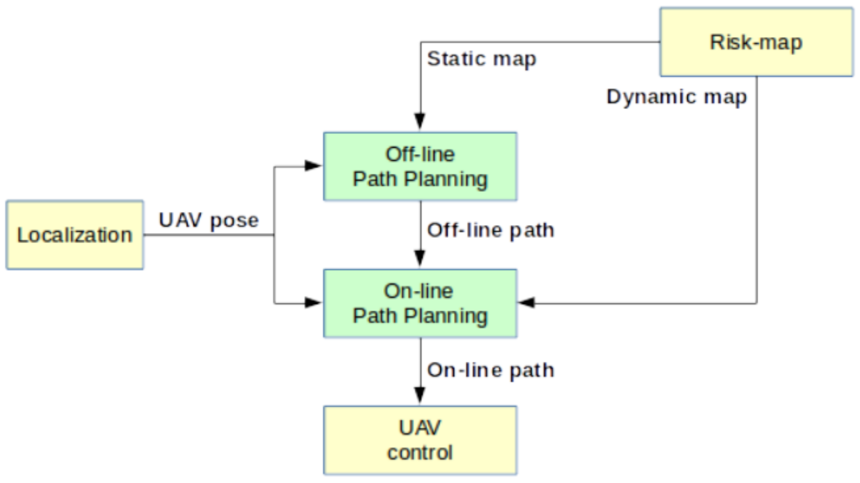


Figure 3.1: The main architecture of the risk-aware path planning method.

The off-line path planning aims to solve an optimal path planning problem. Given a risk-map, a starting and a final position, the off-line path planning looks for an optimal global path avoiding obstacles and no-flight areas, minimizing the risk-cost defined by the risk-map. As its name suggests, the off-line path planning is not time constrained, it has a non-real time behavior because no responses must be guaranteed within any time frame. On the contrary, it's made on purpose to take time, that it needs, because its computation is fundamental for collecting data and reaching the optimal solution. So, it's executed before the mission starts and UAV takes off.
On the other hand, the on-line path planning checks and repairs routine and, consequently, graph in order to fit the old path in real-time, according to changes in the dynamic risk-map. Differently from the previous step, we have a real-time behavior and a strong time dependence. In fact, this path planning is implemented when the UAV executes the mission, i.e. when drone flies, so the reaction to dynamical changing conditions must be immediate in order to avoid critical or fatal situations. After the execution of the path planning procedure, the path is uploaded to the UAV Control System [52], that follows the resulting path.

3.1 Off-line Planning

3.1.1 Problem Formulation

Let $C \subseteq \mathbb{R}^2$ be a continuous search space of a path planning problem, it can be discretized into a discrete space X, on which the risk-map will be constructed considering a certain risk-map resolution. Each state $x \in X$ is a discrete location in the discrete search space, but with a slight abuse of notation, from now on, x behaves as a state of search space, location of risk-map M, or as a node of a search grid graph.
The *obstacle region* $X_{\text{obs}} \subseteq X$, set of locations, or forbidden states, in which flight is outlawed. Thus, re-calling concepts of Section 2.3, regions like obstacles, no-flight or high risk zones have an correspondent cost equal to 1. As a consequence, the set $X_{\text{free}} = X \setminus X_{\text{obs}}$ contains the remaining navigable locations, in which the initial and final states are located, it means $x_{\text{start}}, x_{\text{goal}} \in X_{\text{free}}$, and in particular the final condition is an element of the *goal region* X_{goal}, an open subset of X_{free}.

Let Σ be the set of all paths, where a single path σ is a sequence of connected states x in the search space X.

Definition 3.1. (**Path**) A function $\sigma : [0,1] \to \mathbb{R}^2$ of bounded variation is called a [26]:

- *Path*, if it is continuous;
- *Collision-free path*, if it is a path and $\sigma(s) \in X_{\text{free}}, \forall s \in [0,1]$;
- *Feasible path*, if it is a collision-free path and $\sigma(0) = x_{\text{init}}$ and $\sigma(1) \in X_{\text{goal}}$;

The total variation of a path is essentially its length, i.e., the Euclidean distance traversed by the path in $\mathbb{R}^2$. The feasibility problem of path planning is to find a feasible path, if one exists, and report failure otherwise:

Problem 3.1.1 (**Feasible path planning**). *Given a path planning problem defined by a triplet* $(X_{\text{free}}, x_{\text{start}}, X_{\text{goal}})$*, find a feasible path* $\sigma : [0,1] \to X_{\text{free}}$ *such that* $\sigma(0) = x_{\text{start}}$ *and* $\sigma(1) = x_{\text{goal}} \in X_{\text{goal}}$ *if one exists. If no such path exists, report failure.*

The path planning algorithm searches for an optimal and feasible path σ^* from x_{start} to x_{goal} in X_{free} that minimizes a given cost function:

Problem 3.1.2 (**Optimal path planning**). *Given a path planning problem defined by a triplet* $(X_{\text{free}}, x_{\text{start}}, X_{\text{goal}})$ *and a cost function* $c : \Sigma \to \mathbb{R} \geqslant 0$*, find a feasible path* σ^**, if one exists, such that:*

$$c(\sigma^*) = \arg\min_{\sigma \in \sum} c\big(\sigma(s)\big) \tag{3.1}$$

If no such path exists, report failure.

3.1.2 RRT* Algorithm

Coming from the RRT algorithm, RRT* reveals, concerning general aspects, a similar behavior: the tree is built extracting random samples from the state space, introducing also a bias to explore in the direction of unsearched areas. Every time a sample is drawn, a connection between it and the nearest state of the tree is attempted and, if this link satisfies feasible constraints, the sample becomes part of the tree.

Then, as we have read in the Subsection 1.2.2, two factors, Near neighbor search and Rewiring tree operations, allow to achieve minimal cost between tree connections and, consequently, an optimal solution.

Maintaining a tree structure rather than a graph is not only economical in terms of memory requirements, but may also be advantageous in some applications, due to, for instance, relatively easy extensions to motion planning problems with differential constraints, or to cope with modeling errors. The RRT* algorithm is obtained by modifying RRT in such a way that, through the best parent node searching, formation of cycles is avoided by removing "redundant" edges, i.e., edges that are not part of a shortest path from the root of the tree to a vertex. Since the RRT and RRT* graphs are directed trees with the same root and vertex set, this edges' repression amounts to a "rewiring" of the RRT tree, ensuring that vertices are reached through a minimum-cost path [26].
Thanks to this, RRT* improves asymptotically the quality of its path as the number of samples increases, differently from RRT. In Figure 3.2 the differences between the two approaches are shown: it seems evident that the tree built with RRT* is more ordered than the first one, thanks to the operation described before. Obviously, this features have also a computational trade-off, that can however be overcome with an high computational power cloud framework, as the one we have.

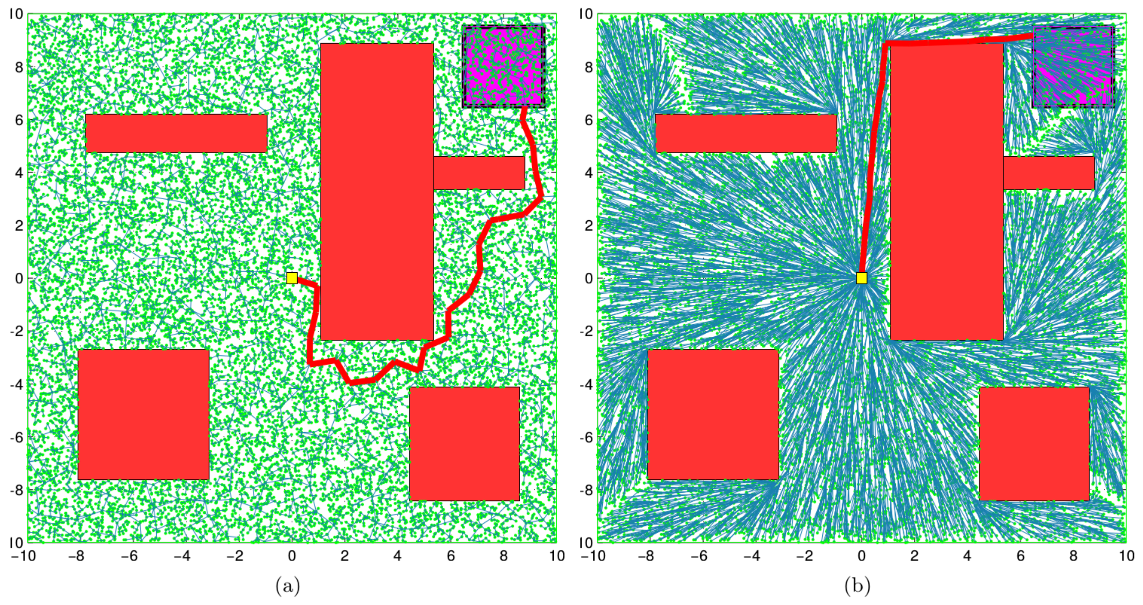


Figure 3.2: A comparison of the RRT (shown in (a)) and RRT* (shown in (b)) algorithms on a simulation example with obstacles [26].

Although, algorithms presented in Figure 3.2 are both implemented to deal only with obstacles, so here is the main step forward of this work, the risk-awareness.
In our approach the failure rate of the UAV is already discussed, from the theoretical point of view, in the Section 2.3. Practically, instead, being risk-aware means that

we have to look for the minimum risk function defined by the risk values in the risk-map while we rewire our graph. Hence, the motion cost of the risk-aware RRT* is defined in the Eq. 3.2, 3.3 :

$$\texttt{Cost}_{\mathrm{m}}(n_{\mathrm{i}}) = \texttt{Cost}_{\mathrm{m}}(n_{\mathrm{i}-1}) + \int_{n_{\mathrm{i}-1}}^{n_{\mathrm{i}}} r(n)dt \tag{3.2}$$

$$\texttt{Cost}_{\mathrm{m}}(n_{\mathrm{i}}) = \texttt{Cost}_{\mathrm{m}}(n_{\mathrm{i}-1}) + \frac{r(n_{\mathrm{i}-1}) + r(n_{\mathrm{i}})}{2}\Delta t(n_{\mathrm{i}-1}, n_{\mathrm{i}}) \tag{3.3}$$

where $c_{\mathrm{m}}(n_{\mathrm{i}-1})$ is the motion cost of the parent node $n_{\mathrm{i}-1}$, $r(n)$ is the risk function defined by the risk values in the risk-map. Practically, because of the discrete search space (the risk-map), the integral is computed with an approximative and incremental method, the Eq. 3.3, where $\Delta t(n_{\mathrm{i}-1}, n_{\mathrm{i}})$ is the flight time expressed in hour needed to cover two adjacent nodes $n_{\mathrm{i}-1}$ and n_{i} [43].

Before discussing the algorithm, looking also at the Figure 3.3, it's necessary to introduce a few explanations:

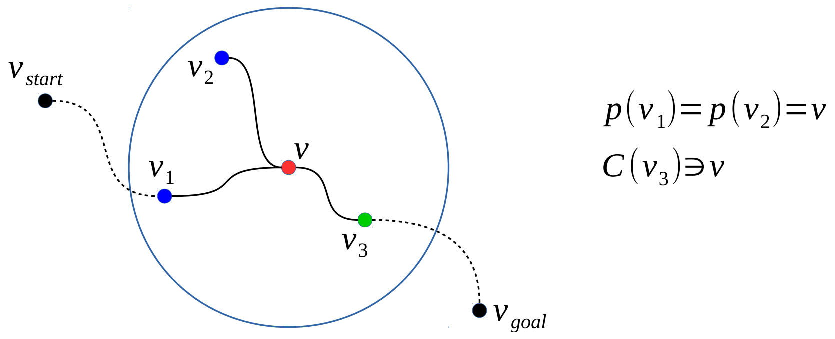


Figure 3.3: Particular of the Graph behavior in RRT*

- Given a graph $G = (V, E)$, depending on the Vertex set V and on the Edge set E, let `Parent` be a function that maps a vertex $v \in V$ to the unique vertex $u \in V$ such that $(u, v) \in E$;
- For simplicity, we assume an addiction cost function, so that:

$$\texttt{Cost}(v) = \texttt{Cost}\big(\texttt{Parent}(v)\big) + c\big(\texttt{Parent}(v), v\big) \tag{3.4}$$

  where the first term of the addition is the risk-cost of its parent's state while the second one is the cost of the straight-line path that goes from the parent's state to the actual state itself;
- Given a graph $G = (V, E)$, P is the Parent set, C is the Children set and, to avoid notation mistakes, from now on, we'll talk about states instead of node/vertex, so the state variable x takes the place of the vertex variable v;

- Given a sampled state, in order to perform rewiring, its neighbors search can be done in 2 different ways and the planner range is a variable that affects both approaches. Through the first one, $r - disc$ calculation, which we use in this work, only neighbors in a certain radius can be examined. Instead, through the second one, $k - nearest$ calculation, we choose the number of neighbors k that we need.

The pseudo-code of the RRT* algorithm is described in Algorithm 1. The inputs are x_{start} and x_{goal} nodes, a grid map M defined taking into account the risk-map and the graph set $G = (V, E)$, which has only x_{start} at initial condition, and the output is the graph set G filled by nodes and their cost information. Thanks to G, using the `getPath` routine, we obtain σ, the vector of adjacent nodes that describes the path between x_{start} and x_{goal}, as shown here, in Eq.3.5:

$$\sigma = \texttt{getPath}(x_{\text{goal}}, x_{\text{start}}, G) \tag{3.5}$$

First, the node x_{start} is added to the vertex set V while the edge set E is empty (line 2) as initialization. Then, it executes the main iterative procedure (line 3 to 28) that continues for a certain number of states n that we define as initial condition of the computation. At each iteration cycle, there are several consecutive operations. The algorithm performs, first, a random sampling (line 4), in which found state is already valid in the map M, and then the nearest state to the random one is searched from the graph G(line 5). If the x_{nearest} is further than planner range value from the x_{rand}, then the x_{nearest} is saturated to a distance equal to the planner range and become x_{new} (line 6-7). On the contrary, if the distance is within the planner range, x_{rand} becomes x_{new} (line 9) . A condition cycle is done (lines 11 to 27) checking if motion between x_{nearest} and x_{new} is allowed. In positive response, the algorithm searches for a neighbors set X_{near} of x_{new} through the above-mentioned $r - disc$ radius (line 12), adds the x_{new} to the vertex set V (line 13) and sets x_{nearest} as x_{new}'s parent (line 14). From now on, there are only two checks. During the first one (lines 16 to 19) the algorithm finds out if exists a neighbor x_{near} that could be a better parent for x_{new} than x_{nearest} and, when it happens, RRT* considers path through x_{near} as associated with minimum cost (line 17), then adds the edge from x_{near} to x_{new} to the edge set E (line 20). During the second one (lines 22 to 26) the algorithm verifies, now, if x_{new} could be a better parent for its neighbors and, when it happens, RRT* deletes the old edge (with the old parent) from the edge set E and adds the edge from x_{near} to x_{new} to the edge set E (line 24). This last iterative check (lines 22 to 26) represents the rewiring factor and, so, the real advantage with respect to RRT. The iterative procedure continues until selected number of sampled states has reached, regardless of solution path achievement, also because RRT* provides only a filled graph set G, it will be the `getPath` sub-routine that should get the best path with the minimum cost in the graph set G.
Note that expressions like that in line 14 or 17 are the same meaning of the Eq.3.2.

Algorithm 1 RRT* algorithm

```
1: procedure RRTSTAR($x_{\text{start}}, x_{\text{goal}}, M, G$)
2:     $V \leftarrow \langle x_{\text{start}} \rangle; E \leftarrow \varnothing;$
3:     for $i = 1, .., n$ do
4:         $x_{\text{rand}} \leftarrow$ SampleFree;
5:         $x_{\text{nearest}} \leftarrow$ Nearest($G, x_{\text{rand}}$);
6:         if dist($x_{\text{nearest}}, x_{\text{rand}}$) $> range$ then
7:             $x_{\text{new}} \leftarrow$ Saturate($x_{\text{rand}}, x_{\text{nearest}}$);
8:         else
9:             $x_{\text{new}} \leftarrow x_{\text{rand}}$;
10:        end if
11:        if CheckMotion($x_{\text{nearest}}, x_{\text{new}}$) then
12:            $X_{\text{near}} \leftarrow$ Near($G, x_{\text{new}}, r$);
13:            $V \leftarrow V \cup \langle x_{\text{new}} \rangle$;
14:            $x_{\text{min}} \leftarrow x_{\text{nearest}}$; $c_{\text{min}} \leftarrow$ Cost($x_{\text{nearest}}$) $+ c(x_{\text{nearest}}, x_{\text{new}})$;
15:            for all $x_{\text{near}} \in X_{\text{near}}$ do
16:                if CheckMotion($x_{\text{near}}, x_{\text{new}}$) $\wedge$ Cost($x_{\text{near}}$) $+ c(x_{\text{near}}, x_{\text{new}}) < c_{\text{min}}$ then
17:                    $x_{\text{min}} \leftarrow x_{\text{near}}$; $c_{\text{min}} \leftarrow$ Cost($x_{\text{near}}$) $+ c(x_{\text{near}}, x_{\text{new}})$;
18:                end if
19:            end for
20:            $E \leftarrow E \cup (x_{\text{min}}, x_{\text{new}})$;
21:            for all $x_{\text{near}} \in X_{\text{near}}$ do
22:                if CheckMotion($x_{\text{new}}, x_{\text{near}}$) $\wedge$ Cost($x_{\text{new}}$) $+ c(x_{\text{new}}, x_{\text{near}}) <$ Cost($x_{\text{near}}$) then
23:                    $x_{\text{parent}} \leftarrow$ Parent($x_{\text{near}}$);
24:                    $E \leftarrow \big(E \setminus (x_{\text{parent}}, x_{\text{near}})\big) \cup (x_{\text{new}}, x_{\text{near}})$;
25:                end if
26:            end for
27:        end if
28:    end for
29:    return $G = (V, E)$;
30: end procedure
```

An example of how the algorithm works and, practically, what kind of expansion it performs can be shown in Figure 3.4, in which we consider 6 steps, 1.2 seconds as solve time (so 0.2 seconds for each step) and a value of planner range equal to 30 meters.

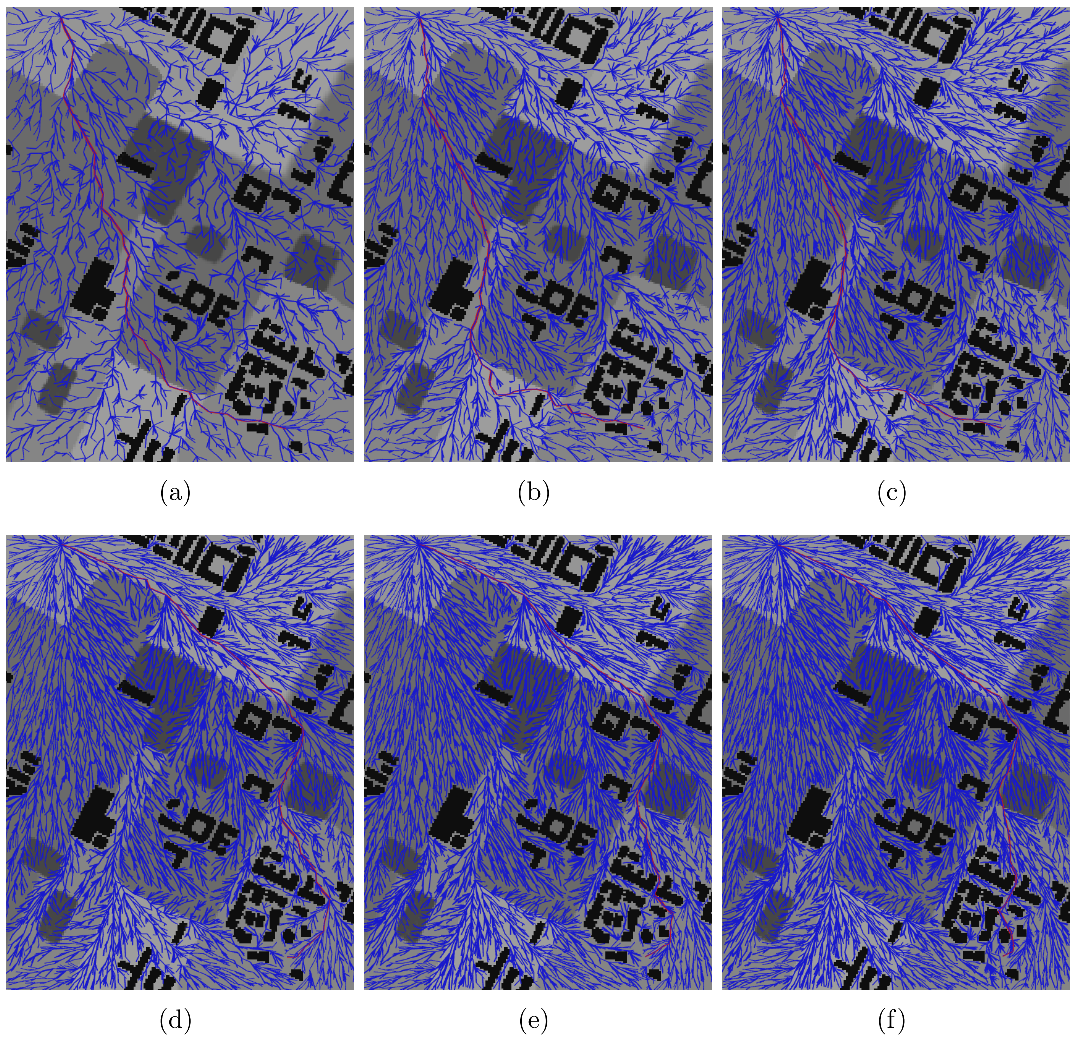


Figure 3.4: Example of the expansion of the RRT* graph

Corresponding data are available in the Table 3.1, in which there are number of steps, number of nodes added to the graph at each step, number of total added nodes after each step and relative solution cost. About the computation of the last variable, it will be explained in Section 4.1

Step	Added nodes for step	Total added node	Solution cost
1	3899	3899	$1{,}424 \cdot 10^{-8}$
2	2336	6235	$1{,}339 \cdot 10^{-8}$
3	1971	8206	$1{,}306 \cdot 10^{-8}$
4	1686	9892	$1{,}288 \cdot 10^{-8}$
5	1623	11515	$1{,}257 \cdot 10^{-8}$
6	1362	12877	$1{,}234 \cdot 10^{-8}$

Table 3.1: Data collection referred to the expansion of the RRT* graph in Figure 3.4

3.2 On-line Planning

3.2.1 Problem Formulation

As we said, the main goal of the on-line path planning is to correct the off-line path planning in order to satisfy the dynamic risk-map requirements and its change. Therefore, considerations made in Subsection 3.1.1 are still working and we have to add only the *check and repair* approach.
This procedure takes into account updates for both path and map involved by changes in the risk-map and exploits environment changes and time-dependence for settings to be performed.

Considering changes happen at each discrete-time step k, every treated variable, once map starts to modify risk-costs of its locations, comes time-dependent. Therefore, it become appropriate speaking of differential variable and, in particular, we can focus on the search space $M(k)$ that changes at each discrete-time step k.
Being the principal input of our procedures, at every time step, we can define a differential search space $M_{\text{diff}}(k)$ as in Eq.3.6:

$$M_{\text{diff}}(k) = M(k) - M(k-1) \tag{3.6}$$

At every time step k, the *check* routine verifies if $x(k) \in M_{\text{diff}}(k) \forall x(k) \in \sigma$, i.e., it checks which states of the path are involved in changes in the risk-map. Thus, the *repair* routine adjusts the path with a fast algorithm, in order to deal with the dynamic risk-map [42].
At time t_{k} the location of the robot is $x_{\text{bot}}(t_{\text{k}})$, where $x_{\text{bot}} : [t_0, t_{\text{curr}}] \to X$ is the traversed path of the robot from the starting time t_0 to the current time t_{cur}, and is undefined for $t_{\text{k}} > t_{\text{cur}}$.
A *static* environment has an obstacle set that changes deterministically with time and x_{bot}, this means that in a generic deterministic environment $\Delta X_{\text{obs}} = f(t_{\text{k}}, x_{\text{bot}})$ and f is known *a priori*. In contrast, a dynamic environment has an unpredictably

changing obstacle set and, even though an obstacle may have the same math expression of the previous case, f is a "black-box" that cannot be known *a priori*. The assumption of incomplete prior knowledge of ΔX_{obs} guarantees myopia and is the defining characteristic of replanning algorithms, in general. While nothing prevents us from estimating ΔX_{obs} based on prior data and/or on-line observations, we cannot guarantee that any such estimate will be correct [39].

Problem 3.2.1 (**Shortest-Path Replanning**). *Given a path planning problem defined by a quadruplet* $(X, X_{\text{obs}}, x_{\text{goal}}, x_{\text{bot}}(0) = x_{\text{start}})$ *and an unknown function* $\Delta X_{\text{obs}} = f(t_{\text{k}}, x_{\text{bot}})$*, find a feasible and optimal path* $\pi^*(x_{\text{bot}}, x_{\text{goal}})$ *and, until* $x_{\text{bot}}(t_{\text{k}}) = x_{\text{goal}}$*, simultaneously update* $x_{\text{bot}}(t_{\text{k}})$ *along* $\pi^*(x_{\text{bot}}, x_{\text{goal}})$ *while recalculating* $\pi^*(x_{\text{bot}}, x_{\text{goal}})$ *whenever* $\Delta X_{\text{obs}} \neq \varnothing$ *where:*

$$\pi^*(x_{\text{bot}}, x_{\text{goal}}) = \underset{\pi(x_{\text{bot}}, x_{\text{goal}}) \in X_{\text{free}}}{\text{argmin}} \; d_\pi(x_{\text{bot}}, x_{\text{goal}}) \tag{3.7}$$

where $\pi^*(x_{\text{bot}}, x_{\text{goal}})$ is an optimal movement trajectory, a curve defined by a continuous mapping $\pi : [0,1] \to X$ such that $0 \to x_{\text{bot}}$ and $1 \to x_{\text{goal}}$. A trajectory is *valid* if both $\pi(x_{\text{bot}}, x_{\text{goal}}) \cap X_{\text{obs}} = \varnothing$ and it is possible for the robot to follow $\pi(x_{\text{bot}}, x_{\text{goal}})$ given its kinodynamic and other constraints $d_\pi(x_{\text{bot}}, x_{\text{goal}})$ is the length of $\pi(x_{\text{bot}}, x_{\text{goal}})$.

In this work, being this section fundamental for thesis' work, two algorithms will be introduced, the Online RRT* and the Online RRT$^{\text{X}}$, in order to figure out which one has better behavior in different cases analyzed in Section 4.

3.2.2 Dynamic RRT* Algorithm

Considerations made in the previous section are the starting point to analyze the on-line behavior of the RRT* algorithm. We're still taking into account improvement factors, that makes RRT* a good chance to get optimal solution, and same interpretation of the risk-cost function and of the way to minimize it.
In addition to some changes about variables in stake, that we'll treat in the pseudocode, the main extension of the off-line algorithm regards updating cost of the graph set G. In fact, for simplicity, the dynamical behavior of the risk-map is represented by the presence of a new map, that we call M_{new}. So the algorithm has to generate an updated graph, called G_{upd}, through a specific routine in order to comply with the environment change and, consequently, with the different risk-cost of each location in the search space. An open set list O is applied to reach our purpose and we'll see, in a while, that the updating will performed by the `updateCost` function.

The pseudo-code of the Dynamic RRT* algorithm is described in Algorithm 2, then the `updateCost` routine is presented in Algorithm 3 to comprehend better the updating part.
About the main Algorithm 2 the inputs are slightly different from the off-line case, x_{start} and x_{goal} are still the original one, but we have to take in consideration a new map, M_{new}, and the old graph set, called G_{old}, achieved by the off-line path planning and filled of nodes and corresponding risk-cost information, that are referred to the old search space risk-map.
The output is a solution path σ reconstructed by exploring the graph set from x_{goal} to x_{start}. It's crucial to note that this graph set has to be referred to the new map, so we called it G_{new} to overcome notation issues. Moreover, different from the previous case, here the computation of the solution path is made inside the algorithm and not with an extern routine.
The code is composed by really few lines: an update is performed (line 2) to bring risk-cost of the G_{old} up to date, such that we can execute the same procedure of the off-line RRT* (line 3) basing on the updated graph set G_{upd}. Finally, from this operation, taking back the new graph set G_{new}, the `getPath` function gets the best solution path with the minimum risk-cost (line 4) and the procedure ends.

Certainly more interesting will be the `updateCost` routine in Algorithm 3. As inputs, there are only the new search space risk-map M_{new} and the old graph set G_{old} referred to the old map. x_{start} and x_{goal} are not inserted as input because they are still present in the old graph set. The output, as we said previously, is the updated graph set G_{upd}.
First, the start state is added to the bottom of the open set list O (line 2) and then the algorithm implements an iterative procedure (lines 4 to 14) that lasts until the open set list O will be empty. In the loop the algorithm performs the update of states, here's how: `pop` operation allows to take the first element x of the open list (line 4), then it starts a search into the graph set G_{old} for the latter element's children (line 5), that we later put in the children set X_{child}. Taking this children, another loop is executed (lines 7 to 12), in which if the motion between x and each x_{child} is allowed (line 7), there will be the risk-cost update of the x_{child} (line 8) and then an addition of this child, x_{child}, to the open set list O (line 9). On the contrary, if motion is not permitted the x_{child} will be erased from the old graph set G_{old} (line 11). The resulting updated graph set will concern only states with allowed and weighted motion, so that new obstacles, new no-flight zones or, simply, new high risk-cost zones can be detected and avoided.
Searching for parent is not needed, as in the off-line case, because the above exploration is overriding, thus each parent will be, consequently, already verified.

Algorithm 2 Online RRT* algorithm

```
1: procedure ONLINERRTSTAR($x_{\text{start}}, x_{\text{goal}}, M_{\text{new}}, G_{\text{old}}$)
2:     $G_{\text{upd}} = \texttt{updateCost}(G_{\text{old}}, M_{\text{new}});$
3:     $G_{\text{new}} = \texttt{RRTstar}(x_{\text{start}}, x_{\text{goal}}, M_{\text{new}}, G_{\text{upd}});$
4:     $\sigma = \texttt{getPath}(x_{\text{goal}}, x_{\text{start}}, G_{\text{new}});$
5:     return $\sigma$;
6: end procedure
```

Algorithm 3 `updateCost` routine

```
1: procedure UPDATECOST($G_{\text{old}}, M_{\text{new}}$)
2:     $O \leftarrow x_{\text{start}};$
3:     repeat
4:         $x \leftarrow \texttt{pop}(0);$
5:         $X_{\text{child}} \leftarrow \texttt{Child}(G_{\text{old}}, x);$
6:         for all $x_{\text{child}} \in X_{\text{child}}$ do
7:             if $\texttt{CheckMotion}(x, x_{\text{child}})$ then
8:                 $\texttt{Cost}(x_{\text{child}}) \leftarrow \texttt{Cost}(x) + c(x, x_{\text{child}});$
9:                 $O \leftarrow O \cup \langle x_{\text{child}} \rangle;$
10:            else
11:                $G_{\text{old}} \leftarrow \big(G_{\text{old}} \setminus (x_{\text{child}})\big);$
12:            end if
13:        end for
14:    until $O \leftarrow \varnothing;$
15:    $G_{\text{upd}} \leftarrow G_{\text{old}};$
16:    return $G_{\text{upd}};$
17: end procedure
```

3.2.3 RRT^X Algorithm

Although they come from the same family, RRT^X has an attitude completely different from RRTs "cousins", it was born to deal with unpredictable environments. Recalling to the Section 1.2.2, it has been presented as the first sampling-based replanning algorithm that is both asymptotically optimal and designed for situation in which a priori off-line computation is unavailable [40]. In particular, RRT^X computes an initial path, then continually refines it toward the optimal solution during navigation, while also repairing it on-the-fly whenever changes to the obstacle set are detected.
One of the main significant divergences with the respect to the other RRT's descendants regards the pruning of the branches. In RRT* procedure, if obstacles appear or change their position, the algorithm cuts off an edge that are no more valid and loose the information about all the branch because it's completely erased from the graph set. In fact RRT* can recognize risk-costs variations, through the update function, but needs to recall the solver in order to rebuild the tree up. On the contrary, RRT^X uses the same graph/tree even when obstacles change and it's able to quickly remodel and repair it, thanks to rewiring operations cascade, instead of pruning disconnected branches away. Both graph and sub-tree (that is the condition of tree no more entire but waiting for complete rewiring) exist in the robot's state space and the tree is rooted at the goal state, allowing it to remain valid despite of robot's state changes during navigation.
Rewiring cascade are performed taking the advantage of a complex maintainance of graph connectivity information, so sometimes the computation could be heavy. Luckily, this drawback is managed by a proper strategy through the concept of ϵ-consistency. Rewiring operation cascades , in fact, are aborted as soon as the graph becomes ϵ-consistent for a certain value of $\epsilon > 0$. It means that the cost-to-goal stored at each node is within ϵ of its look-ahead cost-to-goal, where the latter is the minimum sum of distance-to-neighbor plus neighbor's cost-to-goal. Moreover, this concept is supported by the maintenance of graph set information but in local neighbor sets stored at each node. Being local, the expansion of the neighbor set is smaller, so combining these two aspects the computation won't be too hard and the optimal solution is still computed.

Please notice that both Off-line and On-line RRT^X will be intentionally presented in this section. Although some literatures say something slightly different, the off-line part of this algorithm is not constantly competitive to the off-line part of RRT*. Therefore, for this reason and because RRT^X has been mainly exhibited as a replanning algorithm, we decide to put all together in on-line section, at first the Off-line part with its subroutine and then the On-line one.
Comparisons and comments will be treated in the Chapter 4.

Before discussing the algorithm, looking also at the Figure 3.5, it's necessary to introduce some explanations that are not presented in the RRT$^{\mathrm{X}}$:

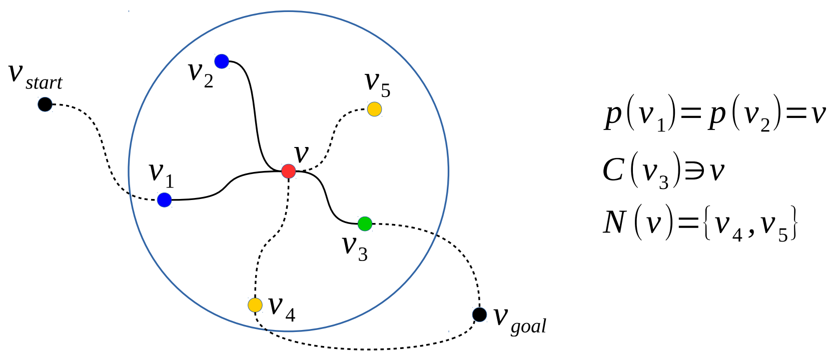


Figure 3.5: Particular of the Graph behavior in RRT$^{\mathrm{X}}$

- Given a graph $G = (V, E)$, depending on the Vertex set V and on the Edge set E, the robot starts at x_{start} and goes to x_{goal}. The shortest path sub-tree of G is $T = (V_{\mathrm{T}}, E_{\mathrm{T}})$, where T is rooted to x_{goal};
- About the ϵ-consistency, $g(x)$ is the (ϵ-consistent) cost-to-goal of reaching x_{goal} from x through T, while $lmc(x)$ is the look-ahead estimate of cost-to-goal. Note that the algorithm stores both $g(x)$ and $lmc(x)$ at each node, and updates $lmc(x)$, as it's shown in Eq. 3.8, when appropriate conditions have been met, while x is (ϵ-consistent) is expressed in Eq.3.9;

$$lmc(x) \leftarrow min_{\mathrm{x_{near} \in N+(x)}} \cdot d_{\pi}(x, x_{\mathrm{near}}) + lmc(x_{\mathrm{near}}) \tag{3.8}$$

$$g(x) + lmc(x) < \epsilon; \tag{3.9}$$

  where then $\sigma^*_{\mathrm{X}}(x, x_{\mathrm{goal}})$ is the optimal path from x to x_{goal} through X and the length of $\sigma^*_{\mathrm{X}}(x, x_{\mathrm{goal}})$ is $g^*(x)$;
- Q is the priority queue that is used to determine the order in which nodes become ϵ-consistent during the rewiring cascades. The key that is used for Q is the ordered pair $(min(g(x), lmc(x)), g(x))$ nodes with smaller keys are popped from Q before nodes with larger keys;
- Finally, just some substitutions are made in order to maintain notation quite similar to the RRT* cases:

  1. $g(x) \rightarrow \texttt{Cost}_{\mathrm{old}}(x)$;

2. $lmc(x) \rightarrow \texttt{Cost}(x)$
3. $d_\pi(x, y) \rightarrow c(x, y)$

The pseudo-code of the RRT$^{\mathrm{X}}$ algorithm is described in Algorithm 4, while its major sub-routines are expressed in Algorithms 5, 6, 7. Starting from the main code, its inputs, as well, are x_{start} and x_{goal} nodes, a search space M defined taking into account the risk-map and a graph set G, while the output is always G filled by nodes and their cost information. As the RRT* case, Thanks to G, using the `getPath` routine, we obtain σ, the vector of adjacent nodes that describes the path between x_{start} and x_{goal}, as shown here, in Eq.3.10:

$$\sigma = \texttt{getPath}(x_{\mathrm{goal}}, x_{\mathrm{start}}, G) \tag{3.10}$$

After the initialization (lines 2-3), the algorithm executes the main iterative procedure (line 5 to 27) that continues for a certain number of states n that we define as initial condition of the computation. At each iteration cycle, there are several consecutive operations.
The algorithm performs, first, a random sampling (line 5), in which found state is already valid in the map M , and then a first check is accomplished (line 6 to 9) in order to verify that these random node are invalid. In this case, the random node is added to the invalid list I (line 7) and through the *continue* command it skips directly to the next cycle of the iteration. The nearest state to the random one is searched from the graph G(line 10). If the x_{nearest} is further than planner range value from the x_{rand} , then the x_{nearest} is saturated to a distance equal to the planner range and become x_{new} (lines 11-12). On the contrary, if the distance is within the planner range, x_{rand} becomes x_{new} (line 14) . A condition cycle is done (lines 16 to 26) checking if the actual x_{new} is a valid state or belongs to an obstacle. In positive response, the algorithm finds the best parent for x from the neighbor set N (line 17) and, then, it finds neighbors within a certain radius value and adds the visited state x as neighbor of these found neighbors (line 18); while in negative reaction x_{new} is added in a invalid list I and its cost is set to infinite (lines 24-25), since the state is in the obstacle region X_{obs}. We'll see in the On-line RRT$^{\mathrm{X}}$ the usage of the invalid list I.
Coming back to the positive iteration of the loop there is, finally, a last condition cycle (lines 20-21), in which better neighbor is rewired to x that becomes its parent (line 20) and then the algorithm calls a function to manage the propagation of cost-to-goal information, maintaining the ϵ-consistency (line 21). Finally, the iterative procedure continues until selected number of sampled states has reached, regardless of solution path achievement, also because RRT$^{\mathrm{X}}$ provides only a filled graph set G, it will be the `getPath` sub-routine that should get the best path with the minimum cost in the graph set G.

Successively, a brief explanation of the sub-routines algorithms is introduced:

- `findParent`(x, V) (Alg. 5) finds the best parent for x from the neighbor set X_{near} (created in line 2), thanks to the iteration cycle, in which check motion verifies available motion and updates risk-costs of selected neighbors.

- `rewireNeighbors`(x) (Alg. 6) rewires x's neighbors x_{near} to use x as their parent, if doing so results in a better cost-to-goal at x_{near} (lines 3-6). This rewiring is similar to RRT*'s rewiring, except that here we verify that ϵ-inconsistent neighbors are in the priority queue (lines 7-8) in order to set off a rewiring cascade during the next call to `reduceInconsistency`(). The `cullNeighbors` function updates x's neighbors to allow only edges that are shorter than r with the exceptions that we don't remove edges that are part of the sub-tree set T.

- `reduceInconsistency`() (Alg.7) manages the rewiring cascade that propagates cost-to-goal information and maintains ϵ- consistency in G (at least up to the level-set of `Cost`$(\cdot)$ containing x_{bot}). An iteration loop is initialized until the Q list is empty (lines 3 to 7). The cascade only continues through x's neighbors if x is ϵ-inconsistent (lines 3-5). The first element of the Q list is picked up (line 3), its parent's cost is updated (line 4) and then its better neighbors are rewired to it, as their parent, if ϵ-inconsistency is verified. This is one reason why RRT$^{\text{X}}$ is faster than RRT$^{\#}$. Finally x's cost becomes the best cost-to-goal, acting always locally 0-consistent (line 6).

Algorithm 4 RRT$^{\text{X}}$ algorithm

```
1: procedure RRTX($x_{\text{start}}, x_{\text{goal}}, M, G$)
2:     $V \leftarrow \langle x_{\text{goal}} \rangle$;
3:     $x_{\text{bot}} \leftarrow x_{\text{start}}$;
4:     for $i = 1, .., n$ do
5:         $x_{\text{rand}} \leftarrow$ Sample;
6:         if then(!Check($x_{\text{rand}}$))
7:             $I \leftarrow x_{\text{rand}}$;
8:             continue
9:         end if
10:        $x_{\text{nearest}} \leftarrow$ Nearest($G, x_{\text{rand}}$);
11:        if dist($x_{\text{nearest}}, x_{\text{rand}}$) $> range$ then
12:            $x_{\text{new}} \leftarrow$ Saturate($x_{\text{rand}}, x_{\text{nearest}}$);
13:        else
14:            $x_{\text{new}} \leftarrow x_{\text{rand}}$
15:        end if
16:        if Check($x_{\text{new}}$) then
17:            findParent($x_{\text{new}}, V$);
18:            getNeighbors($x_{\text{new}}$);
19:            if $x_{\text{new}} \in V$ then
20:                rewireNeighbors($x_{\text{new}}$);
21:                reduceInconsistency();
22:            end if
23:        else
24:            $I \leftarrow \langle x_{\text{new}} \rangle$;
25:            Cost($x_{\text{new}}$) $\leftarrow infiniteCost$
26:        end if
27:    end for
28:    return $G = (V, E)$;
29: end procedure
```

Algorithm 5 findParent routine

```
1: procedure FINDPARENT(($x, V$))
2:     $X_{\text{near}} \leftarrow$ Near($G, x, r$);
3:     for all $x_{\text{near}} \in X_{\text{near}}$ do
4:         if CheckMotion($x_{\text{near}}, x$) $\wedge$ Cost($x_{\text{near}}$) $+ c(x_{\text{near}}, x) < c_{\text{min}}$ then
5:             $x_{\text{min}} \leftarrow x_{\text{near}}$; $c_{\text{min}} \leftarrow$ Cost($x_{\text{near}}$) $+ c(x_{\text{near}}, x)$;
6:         end if
7:     end for
8: end procedure
```

Algorithm 6 `rewireNeighbors` routine

```
 1: procedure REWIRENEIGHBORS($x$)
 2:     if $\texttt{Cost}_{\text{old}}(x) - \texttt{Cost}(x) > \epsilon$ then
 3:         cullNeighbors($x, r$);
 4:         for all $x_{\text{near}} \in X_{\text{near}}$ do
 5:             if $\texttt{Cost}(x_{\text{near}}) > c(x_{\text{near}}, x) + \texttt{Cost}(x)$ then
 6:                 $\texttt{Cost}(x_{\text{near}}) \leftarrow c(x_{\text{near}}, x) + \texttt{Cost}(x)$;
 7:                 $x \leftarrow \texttt{Parent}(x_{\text{near}})$;
 8:                 if $\texttt{Cost}_{\text{old}}(x_{\text{near}}) - \texttt{Cost}(x_{\text{near}}) > \epsilon$ then
 9:                     verrifyQueue($x_{\text{near}}$);
10:                 end if
11:             end if
12:         end for
13:     end if
14: end procedure
```

Algorithm 7 `reduceInconsistency` routine

```
1: procedure REDUCEINCONSISTENCY(())
2:     while $\texttt{size}(Q) > 0$ do
3:         $x \leftarrow \texttt{pop}(Q)$;
4:         updateParent($x$);
5:         rewireNeighbors($x$);
6:         $\texttt{Cost}_{\text{old}}(x) \leftarrow \texttt{Cost}(x)$;
7:     end while
8: end procedure
```

About the On-line version of the RRT^{X}, all the considerations made for the off-line part are still held except for the invalid list I employment. It has made during the off-line procedure and consists of states that, according the old map, have been in the obstacle region X_{obs}, but with the new map they have to be contemplated to certify they are still invalid or not. As in the on-line part of the RRT* the algorithm has to manage environment changes, so a new search space risk-map M_{new} taking into account the old graph set G_{old} that must be updated.
As we'll see, the crucial and innovative factor that represents the real benefit in the robotics avantgarde development is the absence of a solver in the on-line procedure. The updating technique with its complex rewiring operations cascade is able, by itself, to manage dynamic obstacle still remaining the solution to be optimal.

The pseudo-code is presented in Algorithm 8. The inputs are slightly different from the off-line case, x_{start} and x_{goal} are still the original one, but we have to take in consideration a new map, M_{new} , and the old graph set, called G_{old} , achieved by the off-line path planning and filled of nodes and corresponding risk-cost information, that are referred to the old search space risk-map.
The output is a solution path σ reconstructed by exploring the graph set from x_{goal} to x_{start} and performing rewire cascade operations. It's crucial to note that this graph set has to be referred to the new map, so we called it G_{new} to overcome notation issues. Moreover, different from the off-line case, here the computation of the solution path is made inside the algorithm and not with an extern routine.

As for the Dynamic RRT* algorithm, the first step aims to perform an update (line 2) to bring risk-cost of the G_{old} up to date, such that we can execute an iteration procedure (lines 4 to 11) for all the elements belong to the invalid list I. The idea is to check if they are still unvalid even for the M_{new} and, if not, a condition cycle is run (lines 5 to 10). Here, the controlled state is, first, erased from the invalid list I (line 5), then its neighbors let found and it's added to their neighbor set list (line 6). Finally, best parent and best children are searched (line 7 and 8) and the controlled state is added to the node set V (line 9). After that, a new iteration procedure is processed, involves all states contained in the node set V (lines 12 to 15) and aims practically in updating risk-costs of parent and children of each analyzed state. This step is fundamental to implement the rewiring operations cascade. Finally, from this operation, taking back the new graph set G_{new}, at this point already filled of updated informations, the `getPath` function gets the best solution path with the minimum risk-cost (line 16) and the procedure ends.

Algorithm 8 Online RRT$^{\mathrm{X}}$ algorithm

```
1: procedure OnlineRRTx($x_{\mathrm{start}}, x_{\mathrm{goal}}, M_{\mathrm{new}}, G_{\mathrm{old}}$)
2:     $G_{\mathrm{upd}}$ = updateCost($G_{\mathrm{old}}, M_{\mathrm{new}}$);
3:     for all $x_{\mathrm{i}} \in I$ do
4:         if Check($x_{\mathrm{i}}$) then
5:             $I \setminus \langle x_{\mathrm{i}} \rangle$;
6:             getNeighbors($x_{\mathrm{i}}$);
7:             findParent($x_{\mathrm{i}}$);
8:             findChildren($x_{\mathrm{i}}$);
9:             $V \leftarrow V \cup (x_{\mathrm{i}})$;
10:        end if
11:    end for
12:    for all $x \in V$ do
13:        updateParent($x$);
14:        updateChildren($x$);
15:    end for
16:    $\sigma$ = getPath($x_{\mathrm{goal}}, x_{\mathrm{start}}, G_{\mathrm{new}}$);
17:    return $\sigma$;
18: end procedure
```

Chapter 4

Simulations and Results

4.1 Implementations

In order to correctly explain the simulation environment, we get focused on the code and on how we have implemented it in the workstation. Every results presented in this work has been obtained using only open source materials and software, starting from ROS itself. On one side, at least in first developments, poor background with different frameworks, we used, has caused many difficulties and slowdowns, but on the other hand, after a while, we have realized how to handle these tools and to move deeply inside problems to fix and that, probably, makes the system more robust and broader.

First of all, framework and tools have to be introduced. The proposed approach is implemented in C++ as an executable process in ROS (Robot Operating System), the above-mentioned framework presented in Quigley et al. (2009) [44]. The risk-map is generated using the Grid Map library proposed in Fankhauser and Hutter (2016) [18]. Grid Map is a C++ library compatibles with ROS, that is able to generate two-dimensional grid maps with multiple data layers 4.1.

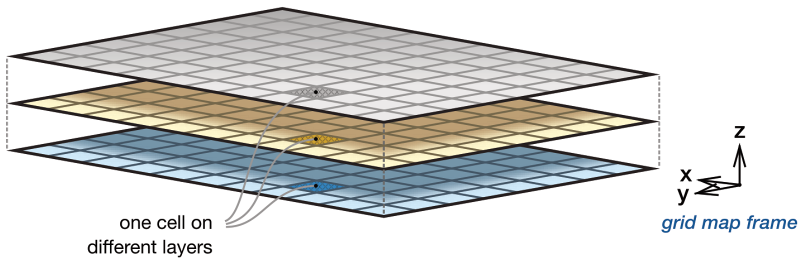


Figure 4.1: Example of Grid Map

Moreover, in order to visualize the grid map, the *grid_map_visualization* package provides a simple tool to convert ROS grid map message in RViz and all the images, that we'll see in Section 4.2 are obtained thanks to specific RViz plugins. RViz (or ROS visualization) is a 3D visualizer for displaying sensor data and state information from ROS. In our case, it has been used to print the risk-map on screen but, above all, to visualize graphs and paths, that are main concepts to be analyzed in this thesis.
Another fundamental used tool is the Open Motion Planning Library (OMPL) in Şucan et al. (2012) [51], it's an open source library, compatible with ROS, specialized in sampling-based motion planning and it consists of many state-of-the-art algorithms.

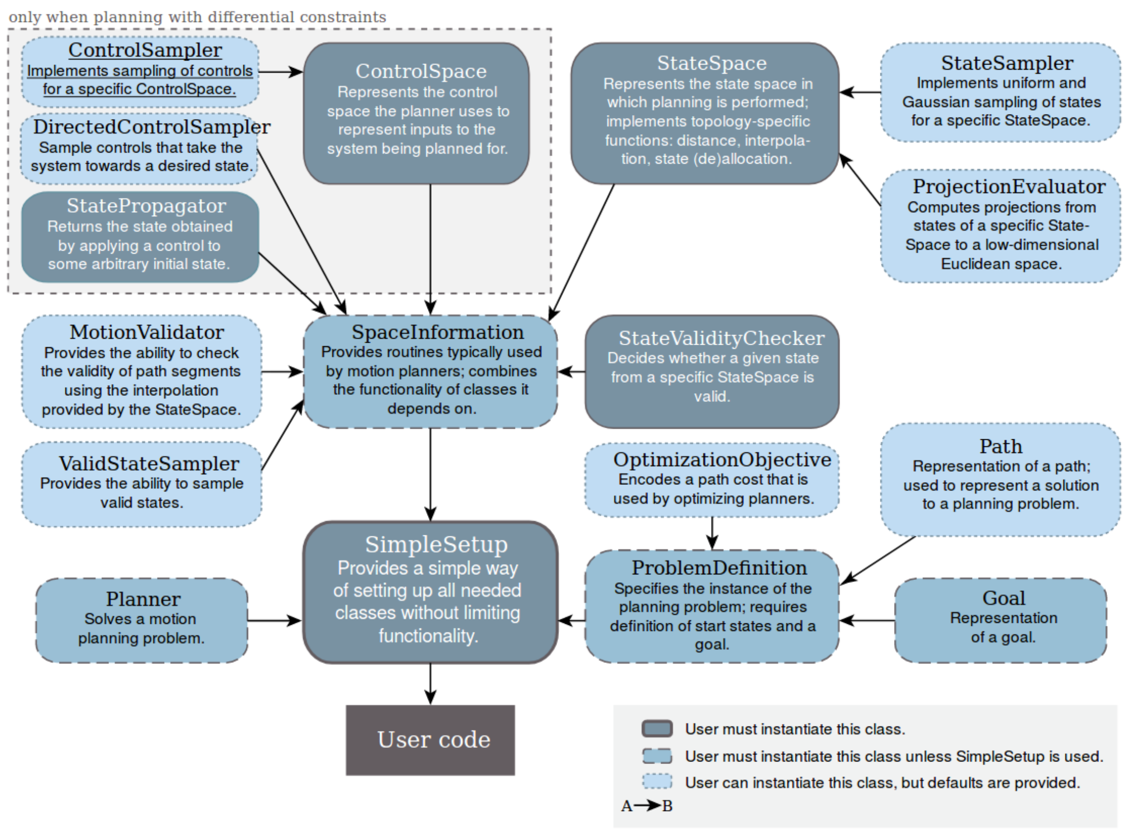

only when planning with differential constraints
ControlSampler
Implements sampling of controls for a specific ControlSpace.
DirectedControlSampler
Sample controls that take the system towards a desired state.
StatePropagator
Returns the state obtained by applying a control to some arbitrary initial state.
ControlSpace
Represents the control space the planner uses to represent inputs to the system being planned for.
StateSpace
Represents the state space in which planning is performed; implements topology-specific functions: distance, interpolation, state (de)allocation.
StateSampler
Implements uniform and Gaussian sampling of states for a specific StateSpace.
ProjectionEvaluator
Computes projections from states of a specific StateSpace to a low-dimensional Euclidean space.
MotionValidator
Provides the ability to check the validity of path segments using the interpolation provided by the StateSpace.
SpaceInformation
Provides routines typically used by motion planners; combines the functionality of classes it depends on.
StateValidityChecker
Decides whether a given state from a specific StateSpace is valid.
ValidStateSampler
Provides the ability to sample valid states.
OptimizationObjective
Encodes a path cost that is used by optimizing planners.
Path
Representation of a path; used to represent a solution to a planning problem.
SimpleSetup
Provides a simple way of setting up all needed classes without limiting functionality.
Planner
Solves a motion planning problem.
ProblemDefinition
Specifies the instance of the planning problem; requires definition of start states and a goal.
Goal
Representation of a goal.
User code
User must instantiate this class.
User must instantiate this class unless SimpleSetup is used.
User can instantiate this class, but defaults are provided.
A→B

Figure 4.2: The hierarchy of the high level components of OMPL.

The library is designed in a way that it allows the user to easily solve a variety of complex motion planning problems with minimal input. OMPL facilitates the addition of new motion planning algorithms, and it can be conveniently interfaced with other software components.

Explained the tools we have worked with, now on, we focus on practical characteristics and parameters that describe code, its operating principle and what we'll see in results after this introduction. During the development of the project, some assumptions and simplifications have been done, so that we are able to write and simulate the code even on modest powered laptop. First of all we only considered quad-copter type aircraft, in particular Iris+ model by 3DRobotics, flying at a cruise speed of 10 m/s. The vehicles are then supposed to have no sensors on-board, relying completely on informations shared within the cloud.
All simulations of the results part are not based and controlled, as often happens, on the computational time but on the number of the states that algorithms have to reach and evaluate. In our specific case, it's set on 10000 states for almost all the tests. The considered motions are performed in a 2D space, so that every drone can just refer to the planar risk-map relative to its flight altitude, which has been set to 30 meters. Speaking of the risk-map, the resolution referred to each pixel is 5x5 meters and the planner range is set to 30 meters. This latter parameter, actually, quite depends on the map's size but, above all, on the obstacles' density. Big amount of obstacles and high value of the planner range entails modest filling of the map by the graph set but, certainly, a very quick solution response. Vice versa, with low values for both, a very high number of states would be reached and, consequently, an increase of the computational time but, at the same time, a better approach to the optimal solution. Therefore, choosing the planner range, actually, means reaching a trade-off between advantages and drawbacks and, in this case, we have selected the above value.

In order to clarify the situation, a summary about parameters is presented in Table 4.1

Parameter	Value
Planner Range	30 $[m]$
Epsilon ϵ	0.1
Flight Altitude	30 $[m]$
Resolution of risk-map	5x5 $[m/pixel]$
Cruise speed	10 $[m/s]$
Number of explored states	10000

Table 4.1: Summary of parameters employed in the Section 4.2

These parameters are given to the node through its own launcher, that is obtained after an application has been completed by the user, where everything is checked according to the criteria imposed by the NFA.
Last but not least, two considerations about the treatment of the data collected in several simulations. First, it's indispensable to notice that we have been worked

with probabilistic algorithms, so it's unthinkable to hope that 2 tests with perfectly same parameters come out same results. Therefore, all resulting variable have been computed as the average of, at least, 5 identical tests. Secondly, the solution cost is precisely the motion cost of the solution and comes out from the Eq. 4.1:

$$\begin{aligned} Motion\_cost &= \sum_{i=start}^{goal-1} \frac{risk(x_{\mathrm{i}}) + risk(x_{\mathrm{i+1}})}{2} \cdot t_{\mathrm{f}}(x_{\mathrm{i}}, x_{\mathrm{i+1}}) = \\ &= \sum_{i=start}^{goal-1} \frac{risk(x_{\mathrm{i}}) + risk(x_{\mathrm{i+1}})}{2} \cdot \frac{d(x_{\mathrm{i}}, x_{\mathrm{i+1}})}{v \cdot 3600} \end{aligned} \tag{4.1}$$

$$Motion\_cost = \frac{\frac{1}{hour} \cdot meter}{\frac{meter}{second} \cdot \frac{second}{hour}} \tag{4.2}$$

where:

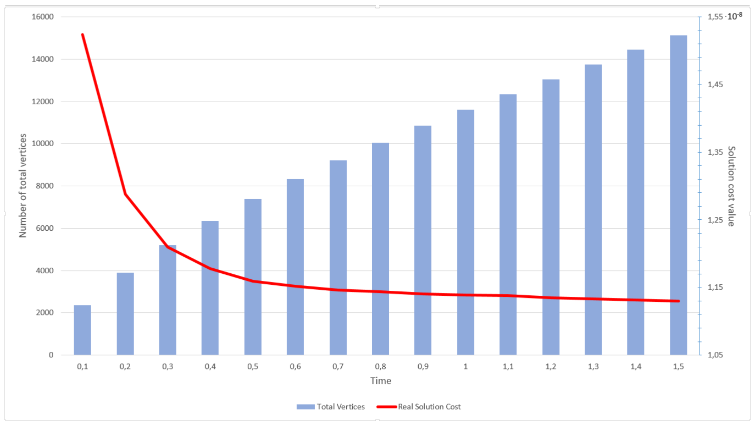
- *risk(x_{i})* is the calculated risk for each node - $[\frac{1}{hour}]$;
- $t_{\mathrm{f}}(x_{\mathrm{i}}, x_{\mathrm{i+1}})$ is the time of flight from one node to the consecutive one - $[second]$;
- v is the cruise speed - $[\frac{meter}{second}]$;
- *d($x_{\mathrm{i}}, x_{\mathrm{i+1}}$)* is the distance between two consecutive nodes in the path, i.e. it's the path length, in this work we consider the Euclidean distance - $[meter]$;

Therefore, adding each contribute we reach the goal and the final solution cost. The Eq. 4.2 shows that, placing 3600 in the denominator to scale hour to second, the equation is correct about the unit of measurement and the solution cost is rightly dimensionless. During the results part, we won't report the calculation of this variable but only the final result, for this reason it seems important to us to specify how it comes out and why it has a very low value.

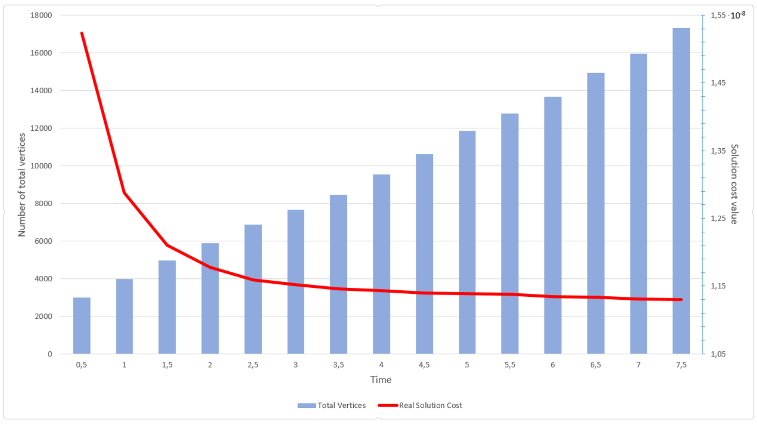
4.2 Results

4.2.1 Off-line Path Planning - RRT* vs RRT^X

About the off-line approach, main interest could be referred to the expansion of the graph and, as a consequence, to the number of explored states related to a certain time and to the achievement of an optimal value of the solution cost. Therefore, in order to recapitulate these features, Figures 4.3 and 4.4 are introduced.
Both tests are based on a sequence of very small solver call in order to better appreciate the trends of the considered variable. In particular, in Figure 4.3, not big differences are deduced about the behavior of the solution function, in fact, in both



(a)

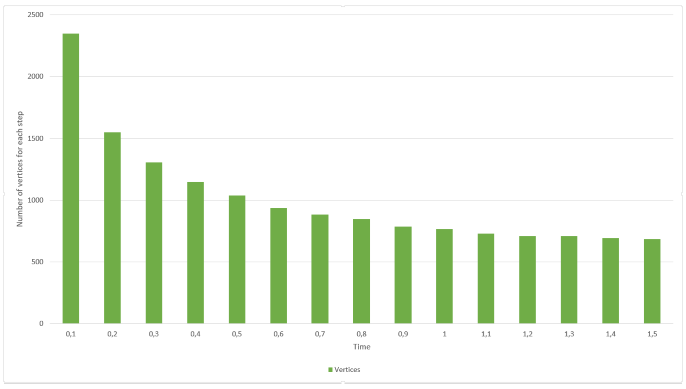


(b)

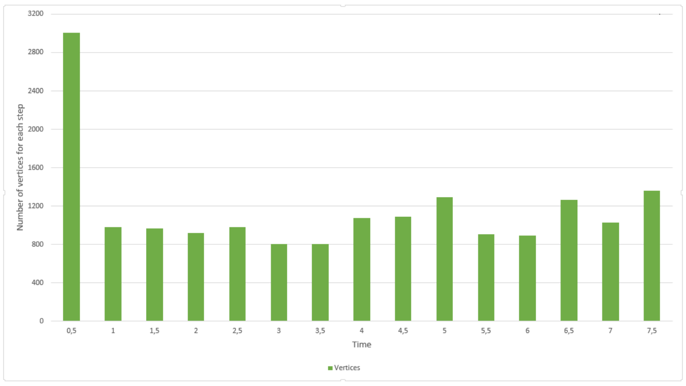
Figure 4.3: Comparison of the RRT* and the RRT^X algorithms about the behavior in off-line environment of total nodes and solution function with respect to the time.

cases, the final value is around $1.13 \cdot 10^{-8}$ and their evolution is included in the 5% variation around this final value already after the fourth step. About evaluated nodes imbalances are minimal. RRT*, during the simulation, slightly reduces the number of added nodes at each step and total explored nodes are around 15200 nodes. While with RRT^X, during the simulation, number of added nodes at each step is various and total explored nodes are around 17300 nodes.
From Figure 4.4, besides, it notices the number of added nodes to the graph at each step. It can be deduced that RRT* has an asymptotic decreasing behavior, as we could see also in Figure 4.3, because of its property of adding much less nodes once it's around the optimal solution region. RRT^X, for its part, after a very high value at first step, it has an irregular behavior in nodes inclusion. This attitude appears because, being the algorithm's search focused on the expansion of the neighbor set

(and not only on the parent/children set as RRT*), more neighbors a single evaluated node has, more nodes, finally, the algorithm finds out. Ergo, being the algorithm probabilistic, the presence of a lot of neighbor is random and, consequently, the behavior of the above function is random.
The most important factor that could be deducted in both the figures is that, contrary to most papers in literature, RRT^X presents a computational time clearly higher that RRT* one. In fact, in order to reach the same solution cost and, approximately, the same number of total evaluated nodes, the former needs 7.5 seconds while the latter only 1.5, it means 5 times higher. This result proves that, in static environment, at least in our conditions, RRT^X can reach an optimal solution but can not compete with RRT* about the computational time.



(a)



(b)

Figure 4.4: Comparison of the RRT* and the RRT^X algorithms about the behavior in off-line environment of the number of explored nodes at each step.

4.2.2 On-line Path Planning - RRT* vs RRT^X

In this subsection we analyze 3 different cases to compare the 2 algorithms behavior in dynamic environments. At first, we start with a map that presents only risk-costs changes, then risk-costs changes and obstacle addition, finally risk-costs changes and obstacle removal. For each case, this structure will be used:

1. A paragraph in which RRT* resulting operations will be introduced, starting from the off-line behavior for both the maps and then moving on to on-line procedure in order to prove that it's functional and useful;
2. A paragraph in which RRT^X resulting operations will be introduced but, contrary to the previous item, since this algorithm doesn't involve a second solver, it doesn't sense to compute the off-line path for the second map. Therefore, the on-line procedure results will be directly shown removing off-line ones;
3. Finally, a paragraph about comments and comparison between behaviors just analyzed in that case;

For all cases, the risk is illustrated in greyscale, in which more the grey is darker more the risk, in that specific area, is higher and likely avoidable. Black areas represent obstacles and no flight zone.

Case 1: Only risk-costs changes

As we said, this case is referred only to risk-costs changes so obstacles still remain at the same position. The maps linked to this test are in Figure 4.5:

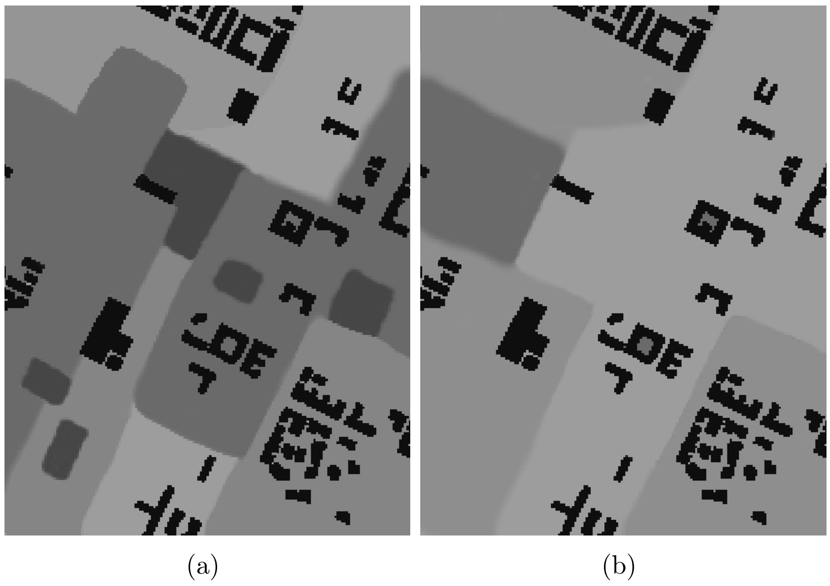


Figure 4.5: Maps used in the case 1. (a) is the first map and (b) is the second one on which re-planning is performed.

<u>RRT*</u> Starting from Figures 4.6, 4.7 and Table 4.2 several considerations can be made: first of all, the on-line algorithm can easily compete with the off-line one because it reaches the same solution cost (so, even the same optimality). Actually, it shows that could be better because the solution cost has been reached faster (0.649 s) than what we see in the off-line case (0.738 s).

Moreover, the expansion of the graph is rightly wider (as we see in Figure 4.7(b)) because, in the on-line case, starting from a graph already quite informed, the second solver is more productive and, in fact, it uses new 4102 explored states to get identical optimality of the off-line case.
Finally, re-planning time is equivalent to 0.787 s, in which 0.65 s comes from the second solver and 0.137 s comes from the risk-costs updating part. Since the total re-planning time is almost equal to the second solver time and the graph is widely more informed, the test can be evaluated as satisfying.

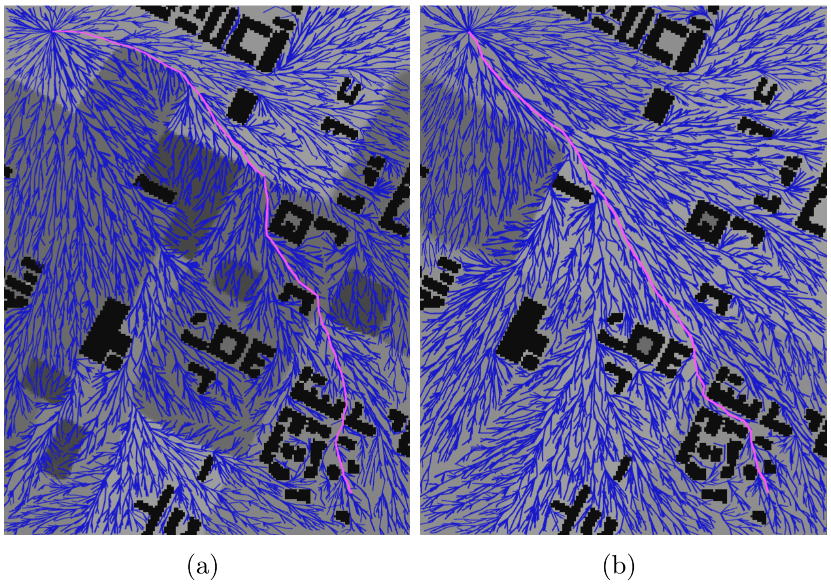


Figure 4.6: In (a) and (b) solution paths for each map is independently computed with the first solver.

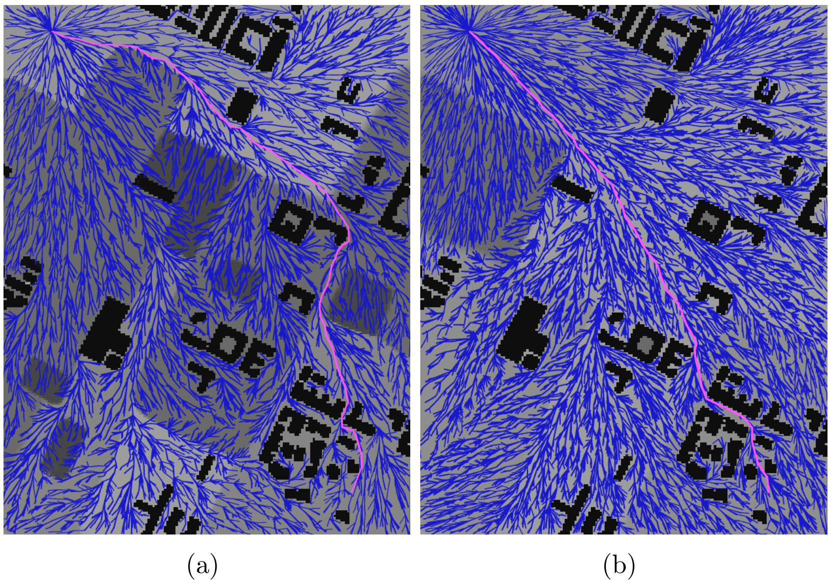


Figure 4.7: The proper on-line procedure is explained in (a), with the first solver, and in (b), update and re-planning with the second solver is computed.

	Explored Nodes	Computational Time [s]	Solution cost
Off-line 1st Solver	10000	0.741	$1.233 \cdot 10^{-8}$
Off-line 2st Solver	10000	0.738	$0.841 \cdot 10^{-8}$
On-line 1st Solver	10000	0.731	$1.245 \cdot 10^{-8}$
On-line 2st Solver	14102	0.649	$0.84 \cdot 10^{-8}$

Table 4.2: Data collection referred to the RRT* procedures shown in Figures 4.6 and 4.7

<u>RRT$^\mathbf{X}$</u> Starting from Figure 4.8 and Table 4.3 several considerations can be made: first of all, as we said in Section 4.2.2, it would be useless to operate with solving also the second map since the algorithm doesn't present a second solver. So we simply register data of our test and compare them to the on-line RRT$^\mathrm{X}$ ones, but we leave comments and comparisons to next paragraph.

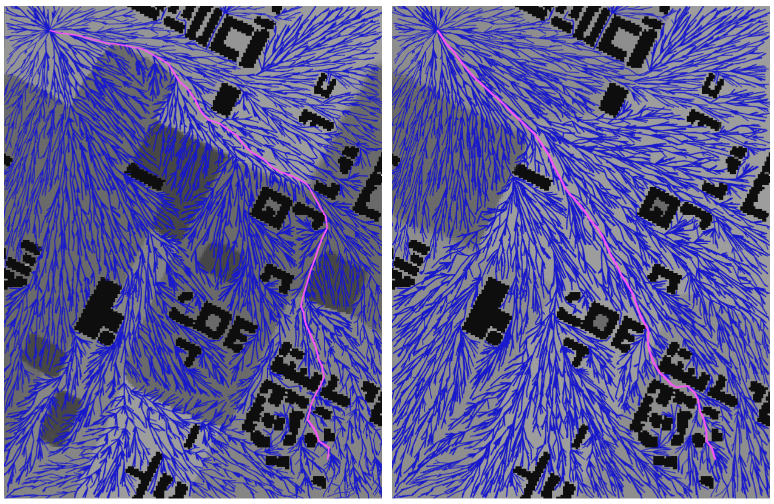

(a) (b)

Figure 4.8: Solution path with the proper on-line procedure is illustrated in (a), thanks to the solver, and (b), thanks to update and re-planning procedure.

The expansion of the graph is wide but even spread on all over the map (as we see in Figure 4.8(b)), while re-planning time is equivalent to 0.482 s, that is a good response of the system since algorithm doesn't search for new nodes and, nonetheless, reaches the optimal solution of the RRT* approach. Finally the re-planning time, this time, is all linked to the updating part, because there is no second solver.

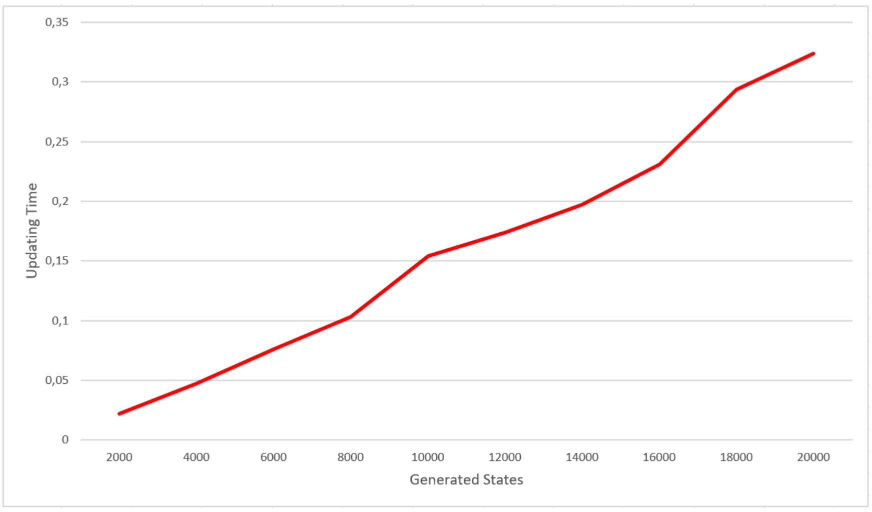
	Explored Nodes	Computational Time [s]	Solution cost
On-line Solver	10000	4.863	$1.255{\cdot}10^{-8}$
On-line Re-planning	10000	0.482	$0.844{\cdot}10^{-8}$

Table 4.3: Data collection referred to the RRT$^{\text{X}}$ procedures shown in Figure 4.8

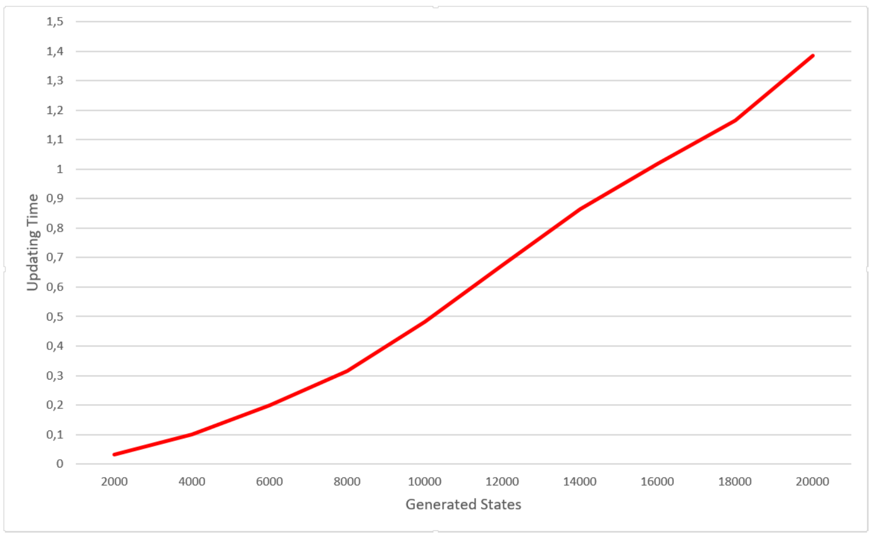
Comments and Comparison Graphically speaking, looking at Figures 4.7(b) and 4.8(b), we notice that, even though RRT$^{\text{X}}$ presents lower number of explored states, it has a more homogenized graph's spread than RRT* actually has. The latter, in fact, has a graph that is weighted to the goal location, i.e. the graph is oriented to the goal and areas quite far away from the goal region, or that are not passing through in order to reach the goal, are much less populated than in the RRT$^{\text{X}}$ case. Actually, also RRT$^{\text{X}}$ has oriented to the goal but not as RRT* because the former has an unlike attitude. Having been made ad hoc for dynamic environment and having a complex rewiring operations due to graph connectivity information of neighbors, the RRT$^{\text{X}}$ algorithm prefers having information around all the valid search space because, in case of unpredictable obstacles appearance, an homogeneous graph set is more convenient since RRT$^{\text{X}}$ doesn't use a second solver. About the re-planning time, it typically depends on solver time and updating part, so another crucial inference regards exactly the re-planning time: about the updating time, RRT* (0.137 s) is faster than RRT$^{\text{X}}$ (0.482 s). However, we are interested on the re-planning time (not to its components) and, in RRT*, it means adding second solver time to updating time and, so, re-planning time becomes higher (0.787 = 0.65 + 0.137) than in RRT$^{\text{X}}$.
Therefore, taking into account only dynamic environment, RRT$^{\text{X}}$ shows that its re-planning is faster than the other one and, above all, that the algorithm reach the same optimal solution.

Finally, in Figure 4.9, a simulation regarding a focus on the updating time is presented. We have chosen to get a test selecting the number of maximum nodes to add to the node set (in our case, 20000 nodes) and dividing the computation in 10 steps in order to achieve the updating time at each step. It could be noticed that behaviors are both increasing and quite similar between each other but the main difference is represented by the updating time value. Total amount of updating time in RRT$^{\text{X}}$, after 10 steps, is a little more than 4 times that one in RRT*.

This consideration proves that RRT$^{\text{X}}$ could get worse than RRT*, when we overcome a certain number of evaluated nodes, because an increasing amount of nodes to update means a long procedure of neighbors update and, consequently, a great increasing of the updating time, and of the computational time. On the other hand, RRT* feels less this above-mentioned dependence on the updating time because it's



(a)



(b)

Figure 4.9: Computation of the updating time with the respect to a certain amount of evaluated states that increases at each step. (a) is referred to RRT* while (b) to RRT^X

still more attached to the value of the second solver and because its updating principle is very easier than the RRT^X one.
Concluding this first case, we can claim that RRT^X works better when only risk-costs changes happen, except for when a big amount of explored nodes is requested. In that case, we have to evaluate pros and cons of both algorithms and then, accept the trade-off.

Case 2: Risk-costs changes and obstacles addition

As we said, this case is referred to risk-costs changes but also to obstacles addition, so, after the update part, it expects that some nodes will be removed from the nodes set. The maps linked to this test are in Figure 4.10:

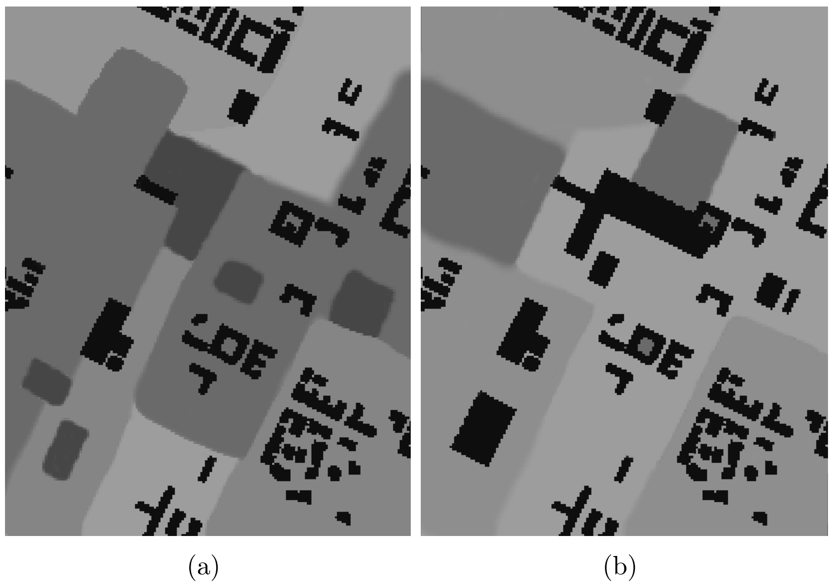


Figure 4.10: Maps used in the case 2. (a) is the first map and (b) is the second one on which re-planning is performed.

RRT*   Starting from Figures 4.11 and 4.12 and Table 4.4 several considerations can be made: first of all, the on-line algorithm can easily compete with the off-line one because it reaches the same solution cost (so, even the same optimality). Actually, it shows that could be better because the solution cost has been reached faster (0.7 s) than what we see in the off-line case (0.81 s).

Moreover, the expansion of the graph is rightly wider thanks to same reasons of the case 1 with the singularity that, as expected, contrary to the above case, obstacles addition in second map causes elimination of nodes that have been localized in that particular region. For this reasoning, the second solver starts with 8145 nodes, instead of 10000, and adds new 5207 explored nodes to the node set to get identical optimality of the off-line part.
About the re-planning time is equivalent to 0.804 s, in which 0.7 s comes from the second solver and 0.104 s comes from the risk-costs updating part. Since the total re-planning time is almost equal to the second solver time and the graph is widely more informed, the test can be evaluated as satisfying.

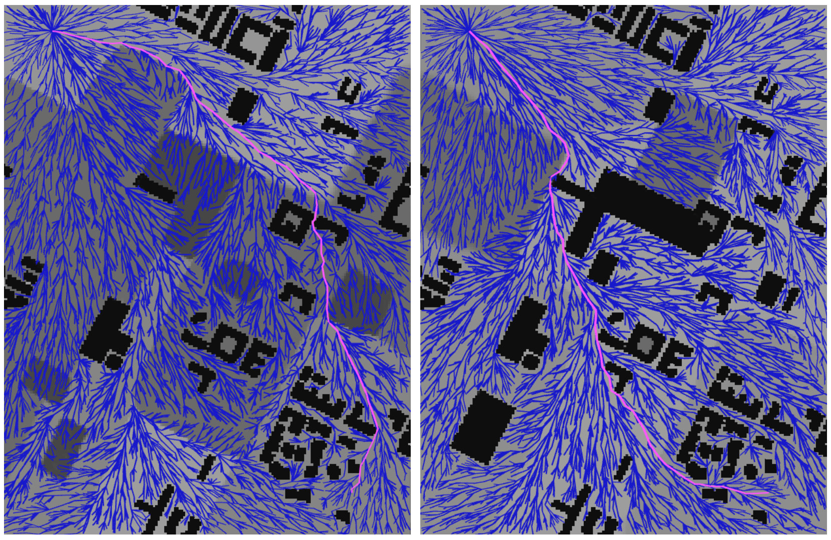

(a) (b)

Figure 4.11: In (a) and (b) solution paths for each map is independently computed with the first solver.

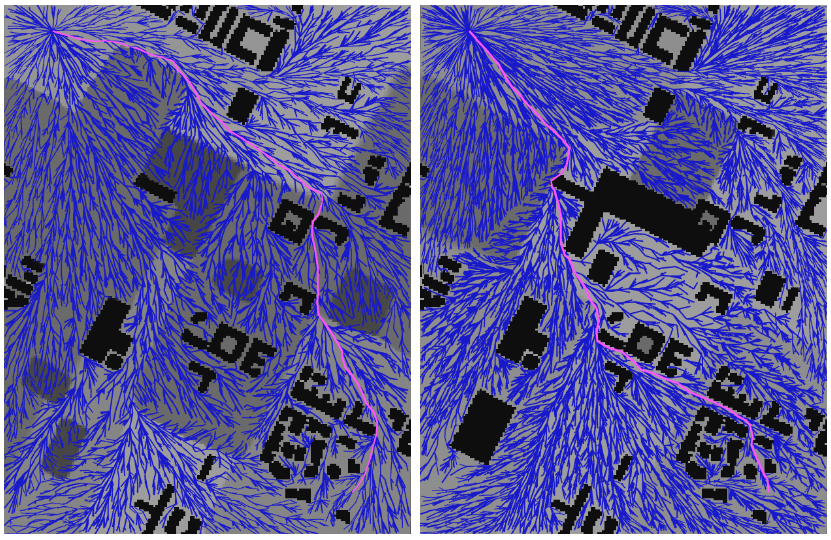

(a) (b)

Figure 4.12: The proper on-line procedure is explained in (a), with the first solver, and in (b), update and re-planning with the second solver is computed.

Furthermore, the obstacles addition influences also the re-planning time, in fact, as we can notice, the updating time (0.104 s) is slightly lower than that one in the case 1 (0.137 s). Indeed, the cancellation of nodes from the new obstacle region reduces the number of explored nodes to update, ergo the updating time and the re-planning time.

	Explored Nodes	Computational Time [s]	Solution cost
Off-line 1st Solver	10000	0.739	$1.237 \cdot 10^{-8}$
Off-line 2st Solver	10000	0.810	$0.926 \cdot 10^{-8}$
On-line 1st Solver	10000	0.738	$1.231 \cdot 10^{-8}$
On-line 2st Solver	13352	0.7	$0.925 \cdot 10^{-8}$

Table 4.4: Data collection referred to the RRT* procedures shown in Figures 4.11 and 4.12

Before going on the RRT$^\text{X}$ it could be particularly appealing showing a detail about the solution cost in the first solve on the first map. From Figure 4.13, reminding that we work with probabilistic algorithms, we can notice how much variable could be the solution cost depending only on probability that a certain branch between two close nodes could be created or not.

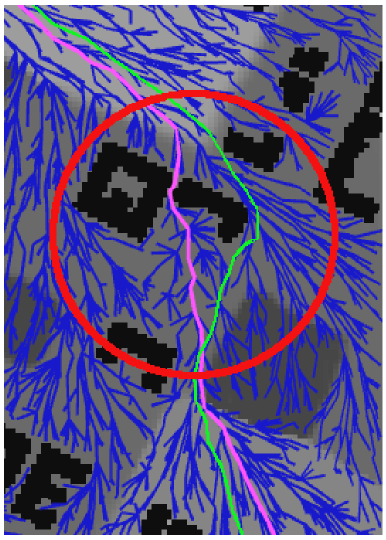


Figure 4.13: Detail of solution path in RRT*, in green the solution cost (that is the average solution cost on 5 identical tests) is $1.237 \cdot 10^{-8}$ while in pink that one is $1.215 \cdot 10^{-8}$

<u>RRT</u>$^{\mathbf{X}}$ Starting from Figure 4.14 and Table 4.5 several considerations can be made: first of all, as we said in Section 4.2.2, it would be useless to operate with solving also the second map since the algorithm doesn't present a second solver. So, as in case 1, we simply register data of our test and compare them to the on-line RRT^{X} ones, but we leave comments and comparisons to next paragraph.

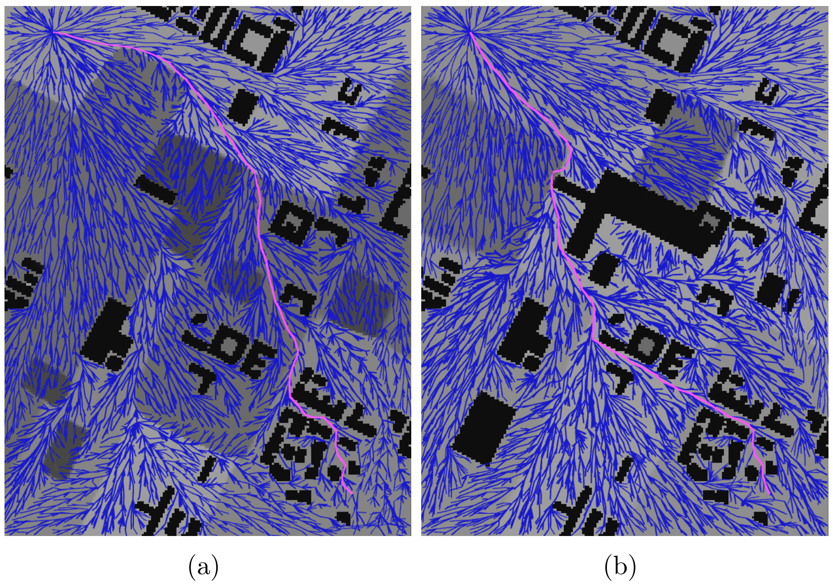


Figure 4.14: Solution path with the proper on-line procedure is illustrated in (a), thanks to the solver, and (b), thanks to update and re-planning procedure.

The expansion of the graph is wide but even spread on all over the map (as we see in Figure 4.14(b)), while re-planning time is equivalent to 0.449 s, that is a good response of the system since algorithm doesn't search for new nodes and, nonetheless, reaches the optimal solution of the RRT* approach.

As in the RRT* part of this case, thanks to the same reasons, the obstacle addition influences both the final number of explored nodes (8327, that are also the number of the evaluated nodes after the solver in the off-line part because RRT^X, having not second solver, can not add nodes to the node set) and the updating time (0.449) slightly lower than that one in the case 1 (0.482).
Finally remember that the re-planning time, this time, is all linked to the updating part, because there is no second solver.

	Explored Nodes	Computational Time [s]	Solution cost
On-line Solver	10000	5.027	$1.258{\cdot}10^{-8}$
On-line Re-planning	8327	0.449	$0.932{\cdot}10^{-8}$

Table 4.5: Data collection referred to the RRT^{X} procedures shown in Figure 4.14

Comments and Comparison Graphically speaking, looking at Figures 4.12(b) and 4.14(b), for same reasons explained in case 1, we notice that, even though RRT^{X} presents lower number of explored states, it has a more homogenized graph's spread than RRT* has. Actually, the difference is rather evident in regions arranged immediately behind, with the respect to the motion from start to goal, the added obstacle. In these regions RRT^{X} has better behavior than RRT*, that has a limited diffusion of its graph just because it doesn't care about neighbors but only about parents and children.
Another crucial inference regards exactly the re-planning time: about the updating time, RRT* (0.104 s) is faster than RRT^{X} (0.449 s). However, we are interested on the re-planning time (not to its components) and, in RRT*, it means adding second solver time to updating time and, so, re-planning time almost doubles ($0.804 = 0.7 + 0.104$) that one in RRT^{X}.

Therefore, especially in this case, RRT^{X} shows that its re-planning is faster than the other one and, above all, that, considering the on-line part, the algorithm reaches the optimal solution path earlier than RRT* could perform.

Case 3: Risk-costs changes and obstacles removal

As we said, this case is referred to risk-costs changes but also to obstacles removal, so, after the update part, it expects that some regions will become valid and they will fill up thanks to the second solver (in RRT* case) or to the updating procedure (in RRT^X). The maps linked to this test are in Figure 4.15:

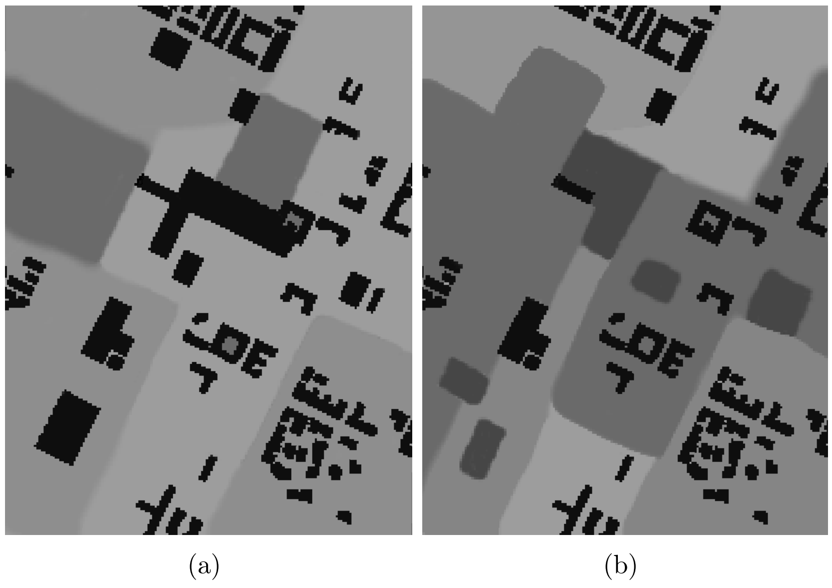


Figure 4.15: Maps used in the case 3. (a) is the first map and (b) is the second one on which re-planning is performed.

Please notice that considered maps are not simply get from the reversal of the case 2 maps. In the first map of this case a new obstacle have been included close to the start node and on the supposed path that algorithms should compute, in order to figure out how algorithms work with nodes that become valid after the off-line part.

RRT* Starting from Figures 4.16 and 4.17 and Table 4.6 several considerations can be made: first of all, the on-line algorithm can easily compete with the off-line one because it reaches the same solution cost (so, even the same optimality). Actually, it shows that could be better because the solution cost has been reached faster (0.7 s) than what we see in the off-line case (0.742 s).
Moreover, the expansion of the graph is rightly wider thanks to same reasons of the case 1 with a singularity different from that one in case 2. The obstacles removal in the second map, as expected, contrary to the previous case, causes an addition of nodes that have been localized in that particular region. For this reasoning, the second solver, starting with 10000 nodes, adds new 4486 explored nodes to the node

set, especially in the new valid region, to get identical optimality of the off-line case.

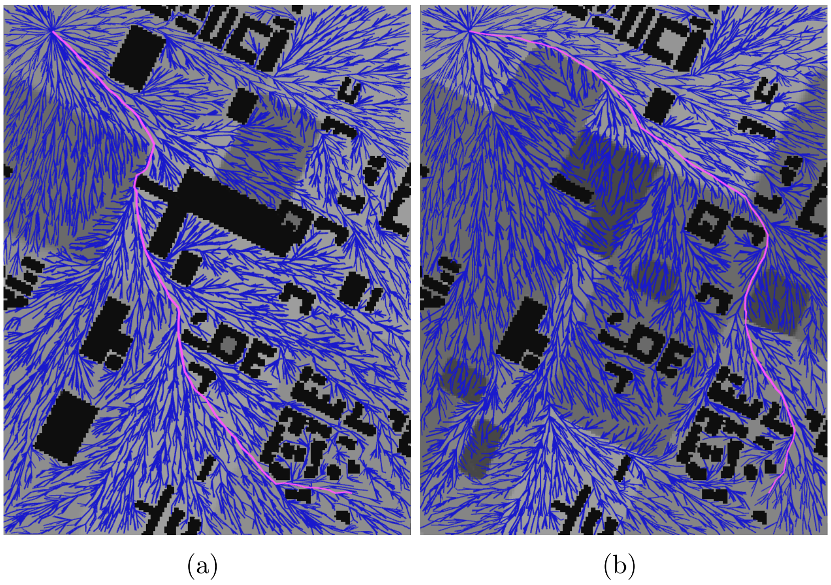


Figure 4.16: In (a) and (b) solution paths for each map is independently computed with the first solver.

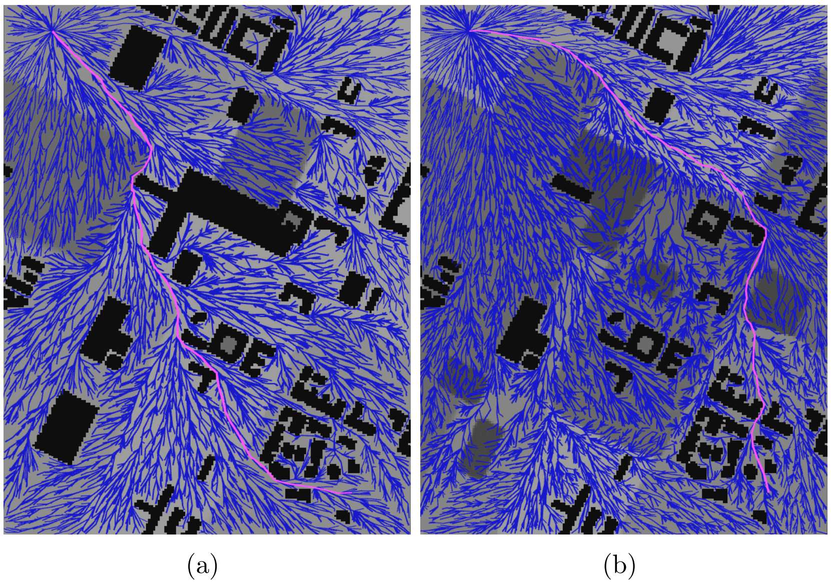


Figure 4.17: The proper on-line procedure is explained in (a), with the first solver, and in (b), update and re-planning with the second solver is computed.

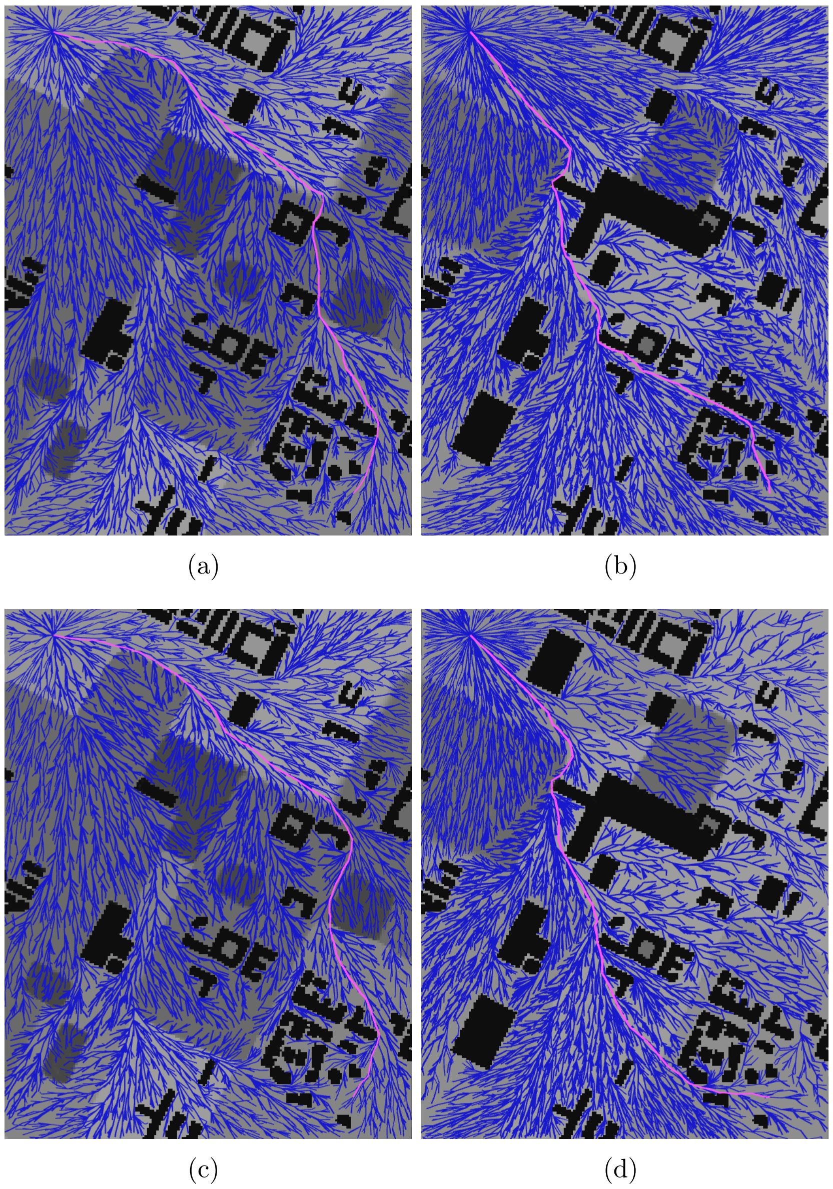
About the re-planning time is equivalent to 0.857 s, in which 0.7 s comes from the second solver and 0.157 s comes from the risk-costs updating part. Since the total re-planning time is almost equal to the second solver time and the graph is widely more informed, the test can be evaluated as satisfying. Furthermore, the obstacles removal influences also the re-planning time, in fact, even if maps of cases 2 and 3 are not perfectly equal, the updating time (0.157 s) is quite higher than that one in the case 2 (0.104 s). Indeed, in case 2, the cancellation of nodes has reduced the number of explored nodes to update.

	Explored Nodes	Computational Time [s]	Solution cost
Off-line 1st Solver	10000	0.760	$0.913{\cdot}10^{-8}$
Off-line 2st Solver	10000	0.742	$1.248{\cdot}10^{-8}$
On-line 1st Solver	10000	0.757	$0.917{\cdot}10^{-8}$
On-line 2st Solver	14486	0.7	$1.252{\cdot}10^{-8}$

Table 4.6: Data collection referred to the RRT* procedures shown in Figures 4.16 and 4.17

Before going on the RRT^{X} it could be particularly appealing showing a detail about the updating time. Supposing maps of the case 3 are inverted and used as map of the case 2 and the on-line procedure gets performed. Our idea is to compare this procedure and that one normally computed in case 2 in order to highlight the effect of obstacles on the updating time for RRT*.
From Figure 4.18, we can detect that, graphically speaking, the obstacle appearance causes, as expected, a pruning of all couple nodes/branches, created in off-line part, that are in the new obstacle region but, as a consequence, also children of these nodes are pruned and so on. The result is that a part of the tree gets erased and thus there are much less nodes to update, as we said in the case 2. The real point is that in Figure 4.18(d) the obstacle is very close to the start point, so a big part of tree have been erased and the remaining branches, that we see in the figure, exist only due to the second solver.
From a computational point of view, this conduct could seem a good thing because the updating time of the Figures 4.18(c)-4.18(d) example is really low (0.058 s) compared with that one come from the Figures 4.18(a)-4.18(b) example (0.104 s), especially because in the Figures 4.18(c)-4.18(d) example the algorithm reaches the optimal solution anyway. However this conduct reveals a very big deal. Even if the updating time will be small, the pruning of pretty big part of the tree represents a drawback for on-line procedure because, if an obstacle forces the path in that specific region in which the tree has been completely erased, the algorithm needs a good value of the second solver time to repopulate that area. And often, this is not

acceptable for real-time behavior. For this reason RRT* doesn't represent always a good approach for the on-line operations and it gets substituted by RRT^X, as we said in Section 3.2.


(a) (b)

(c) (d)

Figure 4.18: Detail of the updating time in RRT*, in (a) and (b) we have maps just used in case 2 and in (c) and (d) we have the maps used in case 3 that, inverted, now become available for the on-line RRT* procedure

RRT^X Starting from Figure 4.19 and Table 4.7 several considerations can be made: first of all, as we said in Section 4.2.2, it would be useless to operate with solving also the second map since the algorithm doesn't present a second solver. So, as in case 1 and 2, we simply register data of our test and compare them to the on-line RRT^X ones, but we leave comments and comparisons to next paragraph.

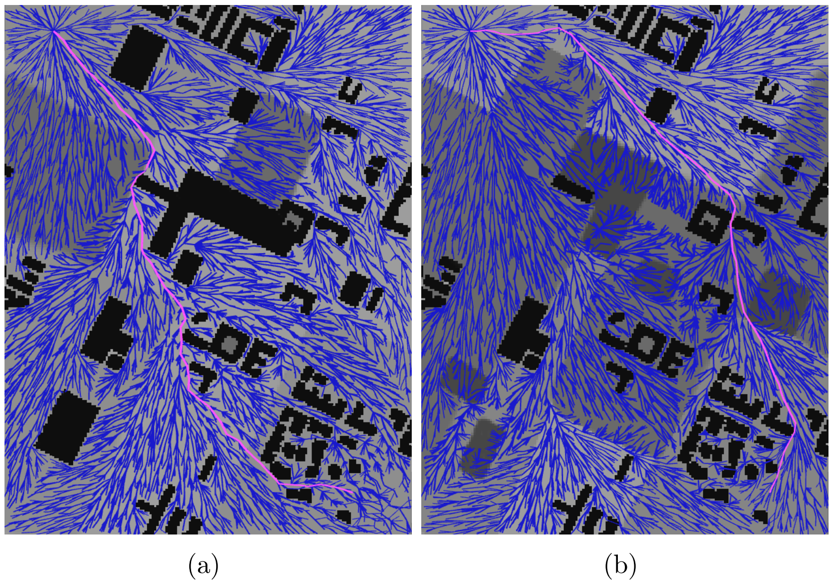


Figure 4.19: Solution path with the proper on-line procedure is illustrated in (a), thanks to the solver, and (b), thanks to update and re-planning procedure.

This is the unique case in which the algorithm through the rewiring operations (and the invalid set I) is able to search for new nodes to explored in regions that become valid. In fact, starting from the usual 10000 nodes after the off-line solver, it adds 930 new nodes to the node set and gets an optimal solution comparable with that one computed with the RRT* with same maps. For this reason, the expansion of the graph is wide and spread on all over the map (as we see in Figure 4.19(b)), here the algorithm doesn't have same problem analyzed in the final part of the previous paragraph.
However, the obstacle removal has a drawback: the algorithm needs a considerable value of computational time to reach the center of the new valid region, so, as we can see in Figure 4.19(b) the solution path is not able to go perfectly through these regions and it's forced to get a trade-off between turning around them and completely crossing them. Most likely, the answer at this issue could depend on how the algorithm uses the planner range. Actually, in reality as in this specific case, this behavior is not a very big deal because obstacles are not ever too huge but identification of this problem could be a good start for future works.

For reasons explained in previous lines, rightly, even the re-planning time is influenced by the obstacle removal. Having to deal with searching of new nodes, in addition to the usual job of costs updating, it has a value of 0.617 s to reach the optimal solution of the RRT* approach. This value is higher than that one referred to case 2 (0.449 s), but, since these conditions are very singular, anyway, it can be evaluated as a sufficiently good response of system.

	Explored Nodes	Computational Time [s]	Solution cost
On-line Solver	10000	5.565	$0.944{\cdot}10^{-8}$
On-line Re-planning	10930	0.617	$1.245{\cdot}10^{-8}$

Table 4.7: Data collection referred to the RRT^{X} procedures shown in Figure 4.19

Comments and Comparison Considering this case as that one with the worst condition for both the algorithm, looking at Figures 4.17(b) and 4.19(b), some comments can be done. Graphically speaking, on one side, for same reasons explained in case 1, we notice that, even though RRT^{X} presents lower number of explored states, it has a more homogenized graph's spread than RRT* has. But, on the other side, having not a real solver, for reasons just explained RRT^{X} is not able to perfectly cross the new valid region as RRT* does.
Another crucial inference regards exactly the re-planning time: about the updating time, RRT* (0.157 s) is faster than RRT^{X} (0.617 s). However, we are interested on the re-planning time (not to its components) and, in RRT*, it means adding second solver time to updating time and, so, re-planning time becomes higher (0.857 = 0.7 + 0.157) than that one in RRT^{X} to get the same optimal solution.

Essentially, on one side, RRT^{X} shows that its re-planning is faster than the other one to reach the optimal solution but has a problem to cross new valid regions when they are pretty big. On the other hand RRT* has issues with branches pruning but tries to solve them with the second solver. Therefore, especially in this case, the difference is very thin and the choice become constrained by working conditions.

4.3 Simulation

4.3.1 Environment Configuration

In this section, a complete simulation of a typical "Autonomous flying of UAV" scenario will be provided: we will show how the processes described in previous chapters really works, focusing mainly on the path planning.
Before starting the simulation, let's have a quick presentation of the main tools we used in it.
In order to perform simulation in which drones have actually flying in the real world, Unmanned Capture The Flag (UCTF) [7] comes to the aid. It is actually nothing more than a game developed in the ROS environment, in which swarms of drones could be flown in an Unmanned Capture The Flag match, that can be played both in the real world and in the simulator. Figure 4.20 shows all the components originally involved and how they are connected with each other. It consists of a simulator (Gazebo), an autopilot software (PX4 in figure) mounted on a real or simulated vehicle, and a mission management interface (GCS), linked together for building up, using a Software-In-The-Loop (SITL) environment.

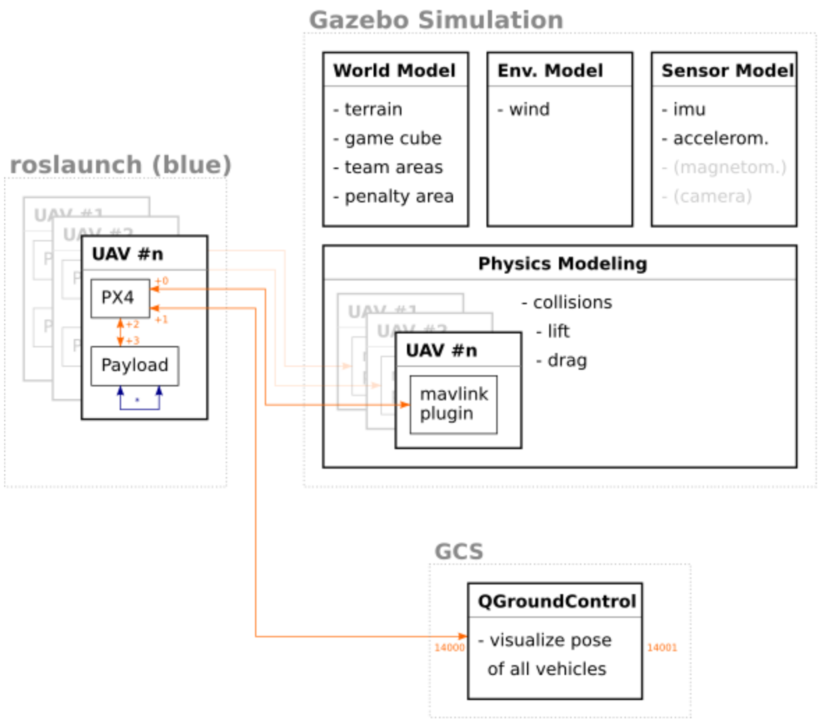


Figure 4.20: Overview of the Unmanned Capture the Flag architecture

Simulation, testing and debugging have been performed in Gazebo [30], a robot simulation software that will be presented more in detail in a while, running on ROS (in Section 2.1) Kinetic distribution. Our needing was to have a simulation environment, in which vehicles as much as some on-board sensors (like the GPS transmitter) were simulated, and could send and receive messages and commands through the MAVLink protocol.

Definition 4.1. A well-designed simulator makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios. Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments [2].

Gazebo is an open source project supported by a great community, that makes it one of the primary tools used by the ROS developers. A specific ROS package has been then developed (Figure 4.21), allowing interaction between the simulation and the ROS environment. It consists of a robust physics engine, high-quality graphics, and convenient programmatic and graphical interface, offering the ability to accurately and efficiently simulate robots in a complex scenario, with everything being customizable according to the needing.

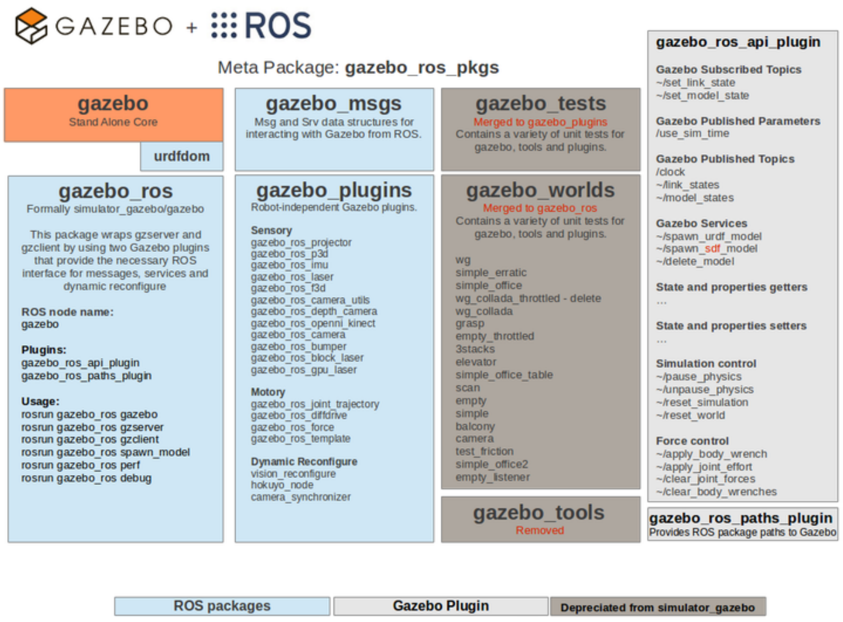
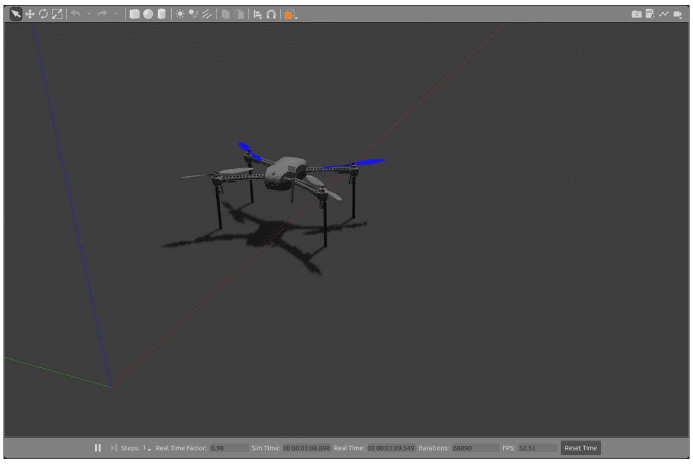
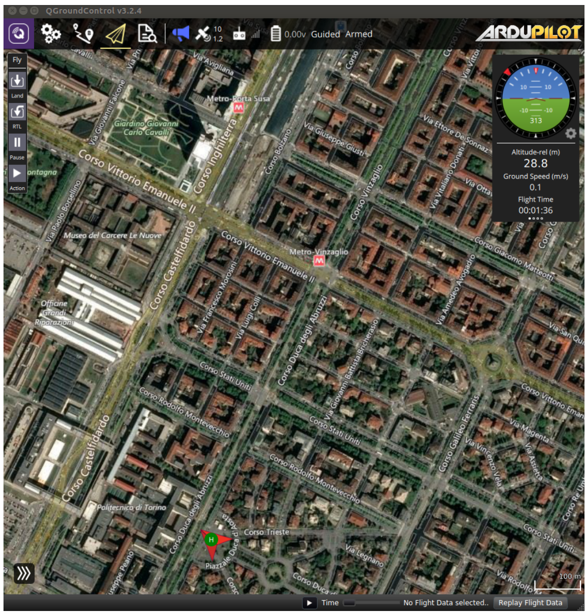


Figure 4.21: Gazebo Application Programming Interface



(a)



(b)

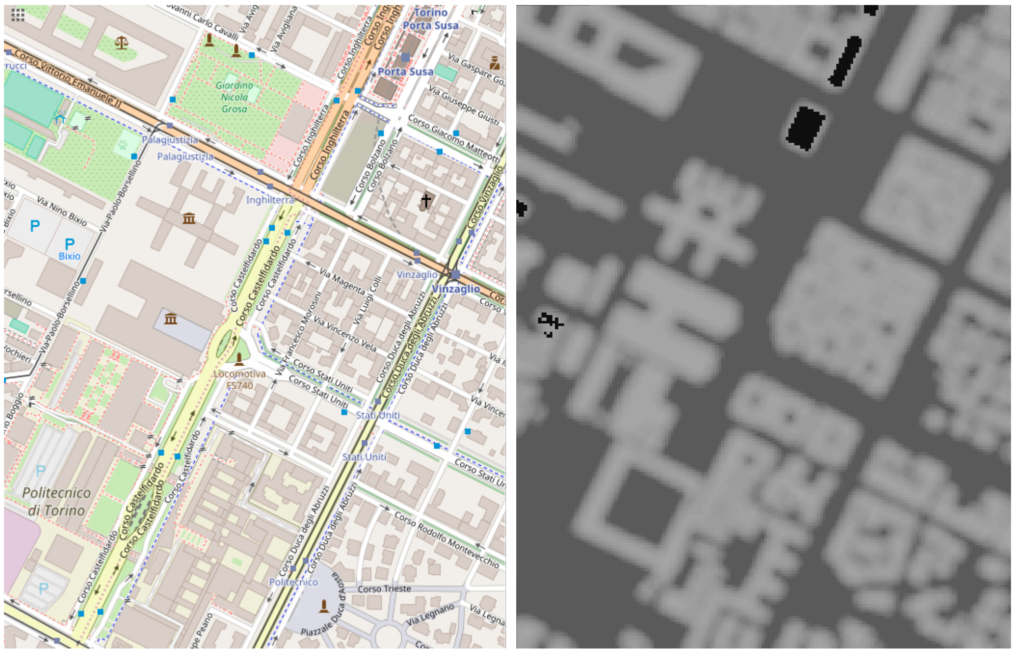
Figure 4.22: In (a) we have the quadcopter located in its reference, thanks to Gazebo, and in (b) its location in the real map, thanks to QGroundControl.

By means of proper world files is, in fact, possible to describe, all the elements making up the scenario, while with model files, is possible to model any kind of robot. This is done in a specific format, called Simulation Description File (SDF), even though it is still possible to find robot model described in the Universal Robot Description File (URDF). SDF is a complete description for everything from the world level down to the robot level. It is described using XML, is scalable, self-descriptive and makes it easy to add and modify elements.

On the other hand, Software-in-the-loop (SITL) allows to run the autopilot software "ArduPilot" directly, without any special hardware. It takes advantage of the fact that ArduPilot is a portable autopilot that can run on a very wide variety of platforms. When running in SITL the sensor data comes from a flight dynamics model in a flight simulator and the wide range of vehicle simulators built in allows ArduPilot to be tested on a very wide variety of vehicle types [6].
A big advantage of ArduPilot on SITL is it gives you access to the full range of development tools available to desktop C++ development, such as interactive debuggers, static analyzers and dynamic analysis tools. Moreover, SITL configuration can be spawned in multiple instances, modeling, if it's needed, different copters to exist at once, allowing us to run simulated drones on a reasonably powered laptop.

Starting from the provided configurations, in our specific case, some modifications have been applied in the JOL, in order to adapt it. Gazebo simulation environment is set up with certain world and environment models. Drone has been inserted by means of its URDF model, describing its physics and dynamics. This is what has been done from UCTF developers, who used a 3DR Iris+ quadcopter as simulated drone (as in Figure 4.22(a)). This vehicle is equipped with an autopilot software, that, managing all the technical aspects of the flight, allows high-level interaction with the drone. It could be chosen between PX4 or ArduCopter, two of the most common autopilot software in commerce: we decided for the latter because of its matching with the SITL. It uses the MAVLink communication protocol, that is used for transmitting commands and informations between vehicle and the control station.
Gazebo node, then, through its plugins, simulates the behaviour of Inertial Measurements Unit (IMU) and GPS sensors, publishing messages about IMU, GPS position and GPS velocity values on specific topics. While, Mavros node represents the ROS/MAVLink bridge, managing the communication between ROS and simulated environments, so the autopilot.
We then used, as interface, the QGroundControl [4] software, a powerful Ground Control Station (GCS) which provides full flight control and mission planning for any MAVLink-enabled PX4- and ArduPilot-powered UAVs (as in Figure 4.22(b)). It has been set according to the Software-In-The-Loop configuration, following the general guidelines, making it read from certain port, where the simulated drone is communicating via MAVLink.
In order to demonstrate the efficiency and the validity of the risk-map and the path planning, a portion of the city of Turin, Italy, will be used to perform the simulation. The model of city, in Figure 4.23(a), is given by OpenStreetMap (OSM) [3], an open source project that distributes geographical data of the world. Thanks to it, we have informations about buildings (position, occupancy and height), so the 3D model can

be extracted and the obstacle layer is defined. And finally, last software involved in the simulation environment set-up is Rviz, that, working on ROS, provides utilities for robotic projects' development and debug phases, allowing to visualize them by reading specific topics and even sending inputs. An example of what Rviz does is in Figure 4.23(b).



(a) (b)

Figure 4.23: In (a) we have the model of the city map, thanks to OpenStreetMap, and in (b) its corresponding map in Rviz environment.

4.3.2 RRT^X simulation

After the environment configuration, we are finally able to perform the simulation. According to our advise, the procedure can be divided in 2 different stages: at first drone executes a certain mission and then, suddenly, it's forced, because of a no flight zone appearance, to compute and perform the re-planning procedure in order to accomplish its initial target.
The computation of the optimal solution path is accomplished by the RRT^X and achieved from the goal to the start, as re-planning theory requests. All messages are sent from the ArduCopter to the Mavros node and vice versa through MAVLink, in order to control the drone's flight.
Graphically speaking, the approach will aim to show in parallel what happens in

Rviz/ROS environment and what the QGroundControl shows in response to its communication with the ArduCopter.
Before comments and discussions, a small overview on the specifications of the aircraft and on the parameters settings is explained in Tables 4.8 and 4.9

Specific	3DR Iris+
Type	Quadrotor
Mass [kg]	1.282
Radius [m]	0.35
Cruise speed [m/s]	10

Table 4.8: Specifications of the aircraft

Parameter	Value
Algorithm	RRT$^{\text{X}}$
Map Resolution [$m \times m$]	5 $\times$ 5
Flight Altitude [m]	30
Explored Nodes	10000

Table 4.9: Parameters setting of the simulation

Starting with the simulations, as we see in Figure 4.24(c) the drone takes off and follows the same path, shown in Figures 4.24(a) and 4.24(b), that has been computed in first step. Suddenly, a no-flight zone appears on the path 4.25(a), so the optimal path is no-longer valid and the re-planning procedure is needed. As we see in Figures 4.25(b) and 4.25(c), clearly the start has changed because drone moves itself and so even the graph changes according to rewiring operations performed by the algorithm. Quickly (and we can see how much quickly in subsection 4.2.2) calculated a new optimal solution path, the autopilot chooses to follow the path so it changes its direction 4.25(d) and, finally, validate our procedure avoiding the obstacle 4.25(e) and reaching the goal.
In order to give some data, the computational time needed to reach 10000 states and get the optimal solution path is equal to 2.411 s, while, the re-planning procedure, that has been performed with 9677 states because of the obstacle addition, is executed in 0.435 s.
Taking into account the huge size of the map, the last results is excellent and certifies the effectiveness of the whole project.

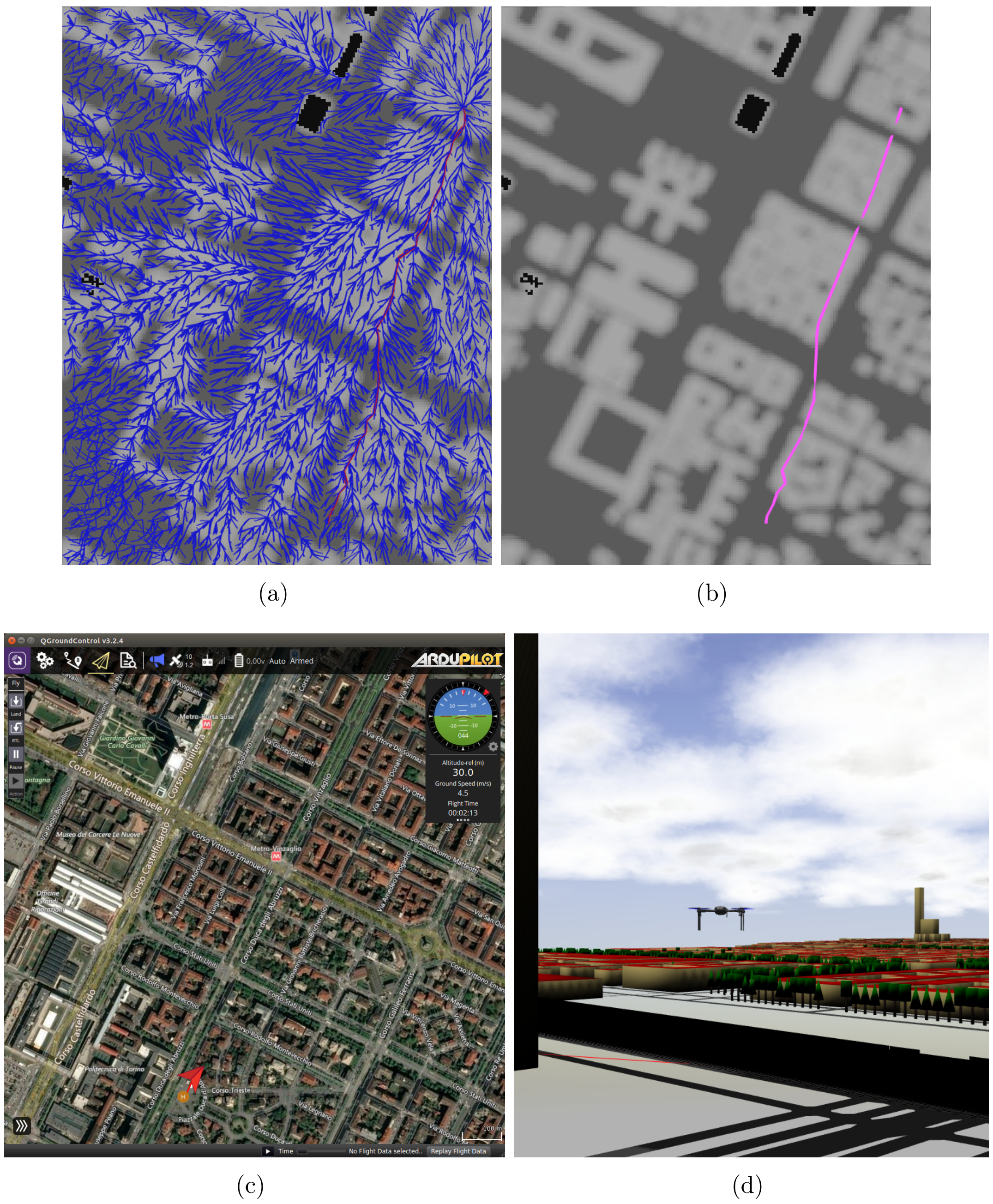
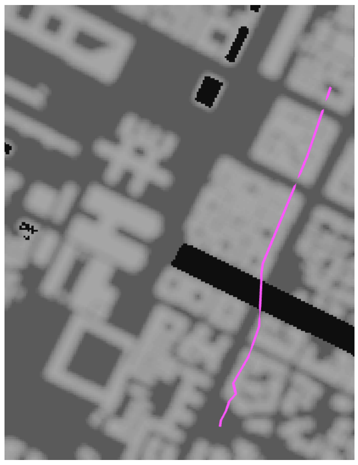
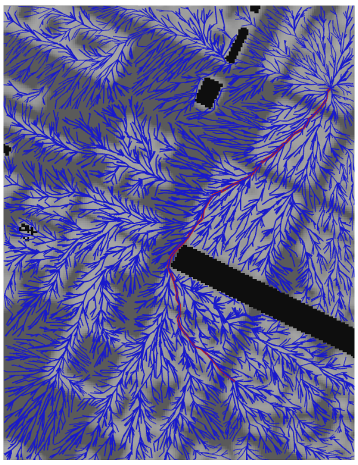


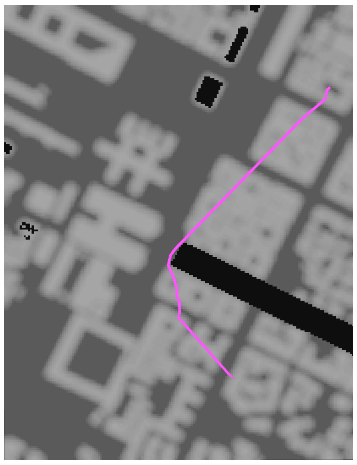
Figure 4.24: We have in (a) the optimal solution path joined with its graph in Rviz, in (b) only the solution path in Rviz, in (c) IRIS+ executes the minimum risk path with SITL simulator in QGroundControl, in (d) IRIS+ in the simulated environment in Gazebo.



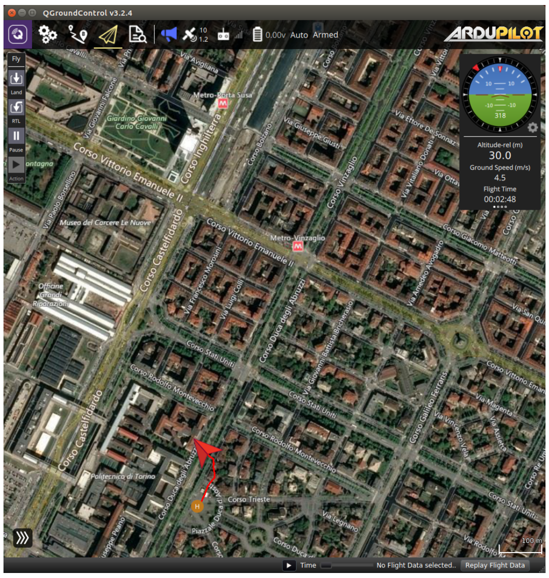
(a)



(b)

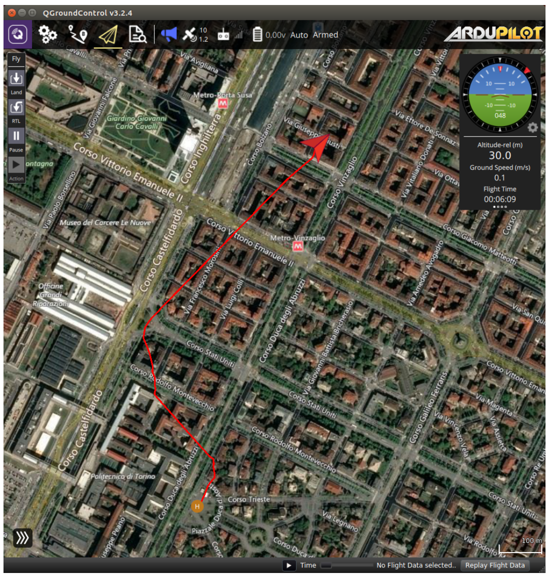


(c)



Altitude-rel (m) 30.0
Ground Speed (m/s) 4.5
Flight Time 00:02:48

(d)



Altitude-rel (m) 30.0
Ground Speed (m/s) 0.1
Flight Time 00:06:09

(e)

Figure 4.25: We have in (a) the obstacle appearance and the no-longer valid optimal path in Rviz, in (b) an (c) the computation of the re-planning optimal path in Rviz and, finally, in (d) and (e) IRIS+ executes the minimum risk path with SITL simulator in QGroundControl, following the instructions obtained by the path planner.

Chapter 5

Conclusions and Future Works

Taking into account the fast and sudden growth of applications field for unmanned aerial vehicles, the main purpose of the whole project has been focused to build a new Traffic Manager able to handle the complexity of a dynamic urban environment, ensuring safety for people on the ground, in order to support commercial fully-autonomous UAS operations in certain airspace.
Starting from the identification of the concepts involved, we have gone on defining problems raised and, then, on researching techniques and methodologies that can be used to treat them, often getting what is available in the literature of the sector. Main preoccupation has been the creation of a well defined structure and set of procedures, so that future developments can be easily integrated within the present architecture. The obtained results have shown the goodness of the proposed solutions, validating the approach that, with further developments, can play a role in the creation of a future UTM system.
Going deeply inside the project, my thesis has been integrated as part of the overall project, conducted in the Joint Open Lab (JOL) with other 3 colleagues, to manage the path planning problem interpreting data coming from the risk-map providing an optimal global path fo dynamic environment, able to avoid obstacles and no-flight areas, an to minimize the risk-cost defined by the risk-map.
The proposed method aspires to compare two probabilistic algorithms, RRT* and RRT^X, through both off-line and on-line analysis, in handling particular and different scenarios, that have been shown in Section 4, to find out which one has best behavior and why.

An essential part was the implementation of the mentioned algorithms, which led us to a better comprehension of the ROS environment. Considering path planner as an independent node improves managing data streams and connections within the different elements of the workstation, however a crucial support have been gave by the open motion planning library (OMPL). It provides an abstract representation for all of the core concepts in motion planning, including the state space, control

space, state validity, sampling, goal representations and planners [51], so that very complex and realistic simulations can be performed.
As we expected, after the initial learning phase in which we invested time to get hold of the tool, the use of this platform allowed and accelerated the entire development. Once the system was implemented, it was possible to test the design results through simulations that have been reported in this thesis in Chapter 4. The simulations conducted finally validated both the proposed algorithms upon the considered working environment. It has been demonstrated that RRT* provides better optimal solutions, especially in off-line settings, but RRT^X can work better in dynamic environment, when time requirements are very burdensome.
Since this is a thesis work, it wasn't possible to cover everything and we really hope someone next will finish it. In particular, some improvements can be done to the specific code composition of the above-mentioned algorithms to take the advantage of all potentialities offered by OMPL. Consideration of kinodynamic constraints of the vehicle, presence of multiple UAVs and adaptation to a tridimensional environment could be other fundamental aspects to be stressed. In this sense, sensors on board and communication link between drones and cloud have to be studied, implementing also a filtering on data.
In any case, more frequent experimentation with real vehicles should be reached to face all those aspects that reality itself involves.

Bibliography

[1] Erle robotics. ros concepts, https://erlerobotics.com/blog/ros-introduction.

[2] Gazebo contributors. http://gazebosim.org/.

[3] Openstreetmap contributors, 2017. planet dump retrieved from https://planet.osm.org.

[4] Qgroundcontrol ground control station for px4 and ardupilot uavs.

[5] Ros contributions. ros wiki, http://wiki.ros.org.

[6] Sitl contributors, 2017. sitl guide retrieved from https://planet.osm.org.

[7] Uctf - unmanned capture the flag; https://github.com/osrf/uctf.

[8] Unmanned aerial vehicle. (n.d.) dictionary of military and associated terms on: https://www.thefreedictionary.com/unmanned+aerial+vehicle.

[9] Unmanned aircraft system traffic management (utm) https://www.faa.gov/uas/research/utm/.

[10] Oktay Arslan and Panagiotis Tsiotras. Use of relaxation methods in sampling-based algorithms for optimal motion planning. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 2421–2428. IEEE, 2013.

[11] Oktay Arslan and Panagiotis Tsiotras. Dynamic programming guided exploration for sampling-based motion planning algorithms. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 4819–4826. IEEE, 2015.

[12] David Brandt. Comparison of a and rrt-connect motion planning techniques for self-reconfiguration planning. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 892–897. IEEE, 2006.

[13] Range Commanders Council. Standard 321-07 "common risk criteria standards for national test ranges: Supplement". *USA Dept. of Defense*, 2007.

[14] Konstantinos Dalamagkidis, Kimon P Valavanis, and Les A Piegl. *On integrating unmanned aircraft systems into the national airspace system: issues, challenges, operational restrictions, certification, and recommendations*, volume 54. springer science & Business Media, 2011.

[15] Didier Devaurs, Thierry Siméon, and Juan Cortés. Optimal path planning in complex cost spaces with sampling-based algorithms. *IEEE Transactions on Automation Science and Engineering*, 13(2):415–424, 2016.

[16] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[17] František Duchoň, Peter Hubinský, Andrej Babinec, Tomáš Fico, and Dominik Huňady. Real-time path planning for the robot in known environment. In *Robotics in Alpe-Adria-Danube Region (RAAD), 2014 23rd International Conference on*, pages 1–8. IEEE, 2014.

[18] Péter Fankhauser and Marco Hutter. A universal grid map library: Implementation and use case for rough terrain navigation. In *Robot Operating System (ROS)*, pages 99–120. Springer, 2016.

[19] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[20] Sara Giammusso. Cloud robotics in real time application.

[21] Giorgio Guglieri, Alessandro Lombardi, and Gianluca Ristorto. Operation oriented path planning strategies for rpas. *AMERICAN JOURNAL OF SCIENCE AND TECHNOLOGY*, 2(6):1–8, 2015.

[22] Felipe Haro and Miguel Torres. A comparison of path planning algorithms for omni-directional robots in dynamic environments. In *Robotics Symposium, 2006. LARS'06. IEEE 3rd Latin American*, pages 18–25. IEEE, 2006.

[23] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[24] ZeFang He and Long Zhao. The comparison of four uav path planning algorithms based on geometry search algorithm. In *Intelligent Human-Machine Systems and Cybernetics (IHMSC), 2017 9th International Conference on*, volume 2, pages 33–36. IEEE, 2017.

[25] Sertac Karaman and Emilio Frazzoli. Optimal kinodynamic motion planning using incremental sampling-based methods. In *Decision and Control (CDC), 2010 49th IEEE Conference on*, pages 7681–7687. IEEE, 2010.

[26] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[27] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580, 1996.

[28] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg. A survey of research on cloud robotics and automation. *IEEE Transactions on Automation Science and Engineering*, 12(2):398–409, April 2015.

[29] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The international journal of robotics research*, 5(1):90–98, 1986.

[30] Nathan P Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IROS*, volume 4, pages 2149–2154.

Citeseer, 2004.

[31] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[32] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[33] Andrea Lorenzini. Cloud-based uavs traffic management system: a risk-aware map manager.

[34] ACMDV Maniezzo. Distributed optimization by ant colonies. In *Toward a practice of autonomous systems: proceedings of the First European Conference on Artificial Life*, page 134. Mit Press, 1992.

[35] 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*, pages 113–118. ACM, 2015.

[36] Alex Nash, Kenny Daniel, Sven Koenig, and Ariel Felner. Theta^*: Any-angle path planning on grids. In *AAAI*, pages 1177–1183, 2007.

[37] Iram Noreen, Amna Khan, and Zulfiqar Habib. A comparison of rrt, rrt* and rrt*-smart path planning algorithms. *International Journal of Computer Science and Network Security (IJCSNS)*, 16(10):20, 2016.

[38] Masoud Nosrati, Ronak Karimi, and Hojat Allah Hasanvand. Investigation of the*(star) search algorithms: Characteristics, methods and approaches. *World Applied Programming*, 2(4):251–256, 2012.

[39] Michael Otte and Emilio Frazzoli. {\mathrm {RRT^{X}}}: Real-time motion planning/replanning for environments with unpredictable obstacles. In *Algorithmic Foundations of Robotics XI*, pages 461–478. Springer, 2015.

[40] Michael Otte and Emilio Frazzoli. Rrtx: Asymptotically optimal single-query sampling-based motion planning with quick replanning. *The International Journal of Robotics Research*, 35(7):797–822, 2016.

[41] Francesco Polia. Cloud-based uass traffic management: Trajectory tracking and collision avoidance.

[42] Stefano Primatesta, Giorgio Guglieri, and Alessandro Rizzo. A risk-aware path planning method for unmanned aerial vehicles. In *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 905–913. IEEE, 2018.

[43] Stefano Primatesta, Luca Spanò Cuomo, Giorgio Guglieri, and Alessandro Rizzo. An innovative algorithm to estimate risk optimum path for unmanned aerial vehicles in urban environments. In *2018 International Conference on Air Transport (INAIR)*. IEEE, 2018.

[44] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[45] Stjepan Rajko and Steven M LaValle. A pursuit-evasion bug algorithm. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International*

Conference on, volume 2, pages 1954–1960. IEEE, 2001.

[46] Elon Rimon and Daniel E Koditschek. Exact robot navigation using artificial potential functions. *IEEE Transactions on robotics and automation*, 8(5):501–518, 1992.

[47] Stefano Scheggi and Sarthak Misra. An experimental comparison of path planning techniques applied to micro-sized magnetic agents. In *Manipulation, Automation and Robotics at Small Scales (MARSS), International Conference on*, pages 1–6. IEEE, 2016.

[48] Enrico Stabile. Cloud-based uass traffic management: Registration, identification and monitoring.

[49] Anthony Stentz. Optimal and efficient path planning for partially-known environments. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 3310–3317. IEEE, 1994.

[50] Anthony Stentz et al. The focussed dˆ* algorithm for real-time replanning. In *IJCAI*, volume 95, pages 1652–1659, 1995.

[51] Ioan A Sucan, Mark Moll, and Lydia E Kavraki. The open motion planning library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, 2012.

[52] PB Sujit, Srikanth Saripalli, and Joao Borges Sousa. Unmanned aerial vehicle path following: A survey and analysis of algorithms for fixed-wing unmanned aerial vehicless. *IEEE Control Systems*, 34(1):42–59, 2014.

[53] Brian P Tice. Unmanned aerial vehicles: The force multiplier of the 1990s. *Airpower Journal*, 5(1):41–55, 1991.

[54] Liang Yang, Juntong Qi, Jizhong Xiao, and Xia Yong. A literature review of uav 3d path planning. In *Intelligent Control and Automation (WCICA), 2014 11th World Congress on*, pages 2376–2381. IEEE, 2014.