2

STATE OF THE ART

Although the major research efforts in this thesis were focused on developing, extending and using path/motion planning methods for autonomous underwater vehicle (AUV) applications, their validation with an AUV in real-world scenarios involved other fields of study. These included but were not limited to: navigation, perception, mapping and control. An extensive presentation of the state-of-the-art background of all these areas would not only prove to be lengthy and diffuse, but would also be out of the scope of this work. Nonetheless, this chapter is dedicated to contextualizing the contribution of this thesis. In order to do so, it firstly provides a survey of the techniques available for solving specifically path/motion planning problems. Secondly, it discusses the most common extensions used when the vehicle motion capabilities have to be taken into consideration or when dealing with online computation constraints. Finally, it presents a review of the algorithms and extensions that have been used with AUVs particularly.

2.1 PLANNING COLLISION-FREE PATHS OVER THE C-SPACE

Even though first works on path/motion planning appeared in the late 60's [103], it was only until the 80's, when Lozano-Perez introduced the concept of the *configuration space* [89–91], that this field of study became active. The configuration space (or configuration space (C-Space)) establishes the set of all possible *configurations* that a robot can adopt when executing tasks in the workspace. While the workspace, $\mathcal{W}$, is typically defined as $\mathbb{R}^n$ (where $n = 2, 3$ for 2-dimensional (2D) and 3-dimensional (3D) motion, respectively), the C-Space, $\mathcal{C}$, depends on the robot motion capabilities. For example, a robot considered to be a point that moves in a plane (i.e., $\mathcal{W} = \mathbb{R}^2$) requires two coordinates in order to specify its *configuration*, q, which is equal to $[x, y]^T$, thus $q \in \mathcal{C} = \mathbb{R}^2$.

However, if the robot is considered to be a rigid body that moves in a plane (i.e., in the same $\mathcal{W}$), it requires three coordinates; these would describe not only its position, but also its orientation, therefore its *configuration* is now $q = [x, y, \psi]^T$, which is commonly expressed as $q \in \mathcal{C} = SE(2) = \mathbb{R}^2 \times SO(2) = \mathbb{R}^2 \times \mathcal{S}$. A similar scenario occurs with a single rigid-body robot that operates in 3D workspaces, in which $q = [x, y, z]$, so that $q \in \mathcal{C} = \mathbb{R}^3$ when only the robot´s position is considered, or $q = [x, y, z, \phi, \theta, \psi]$, so that $q \in \mathcal{C} = SE(3) = \mathbb{R}^3 \times SO(3)$ when considering both position and orientation. Finally, when the robot is an articulated rigid body system, for instance a manipulator arm, a given *configuration* is described by the values of a set of n generalized coordinates $q = q_1, \ldots, q_n$, corresponding to each of the robotic arm degrees of freedoms (DOFs).

The solution to a simple path/motion planning problem, which requires connecting a start and a goal configuration, q_{start} and q_{goal}, is a con-

tinuous path $p : [0,1] \rightarrow \mathcal{C}$, such that $p(0) = q_{start}$ and $p(1) = q_{goal}$. However, a robot generally conducts tasks in environments that contain obstacles, which normally are to be avoided. For this reason, the C-Space is subdivided into *free space* ($\mathcal{C}_{free}$) and the *obstacle region* ($\mathcal{C}_{obs}$), meaning that $\mathcal{C} = \mathcal{C}_{free} \cup \mathcal{C}_{obs}$. Therefore, a collision-free path is defined as a continuous path $p : [0,1] \rightarrow \mathcal{C}_{free}$. This concept is illustrated in Figure 4.

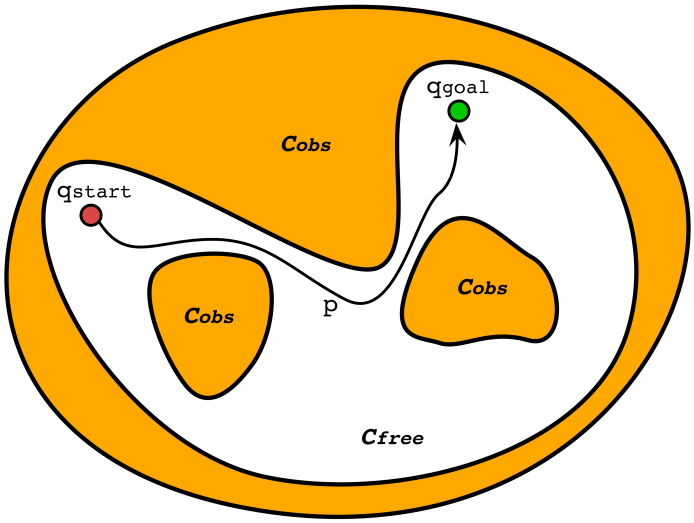


Figure 4: The basic path/motion planning problem seeks to connect a start configuration (q_{start}) and goal configuration (q_{goal}) with a continuous collision-free path $p : [0,1] \rightarrow \mathcal{C}_{free}$.

2.2 BUG-BASED METHODS

Bug-based methods represent one of the earliest reactive and sensor-based path planning approaches, in which a mobile robot moves in the plane towards a global goal, while having a limited (local) knowledge of the environment. The algorithms included in this group are based on two basic behaviors: go straight towards the goal and follow the obstacles' boundary. Lumelsky and Stepanov presented the first of these algorithms, known as Bug1 and Bug2, that mainly rely on tactile or zero range sensors to perceive the obstacles [92]. Later, Kamon et al. introduced the Tangent Bug algorithm, which is an extension that uses non-zero range sensors [61]. All of them are classified as complete algorithms, which means that they find a solution when one is possible or, otherwise, they report when there is no solution. Furthermore, they assume the robot is a point that moves in a 2D workspace, and that it is capable of knowing its position and calculating its distance to the goal.

Bug1 [92] is the most basic algorithm that uses the aforementioned behaviors. With this method, the robot moves from the start position (q_{start}) towards the desired goal position (q_{goal}) until it reaches the goal or until it finds an obstacle (detected with its tactile sensors). When the latter situation occurs, the robot stores its current position and marks it as the *hit point* (H_i). Then it changes its motion mode (behavior) to completely circumnavigate the obstacle. While the vehicle follows the obstacle boundary,

it continuously calculates the distance to the goal in order to determine the position that corresponds to the minimum possible distance. This is marked as the *leave point* (L_i). Once the robot reaches the *hit point* again (i.e., after it has traveled all the obstacle contour), it goes back to the *leave point* and there changes its behavior, once again, in order to move towards the goal. This procedure is repeated until reaching the specified goal. Figure 5 depicts an example of this algorithm solving a start-to-goal query.

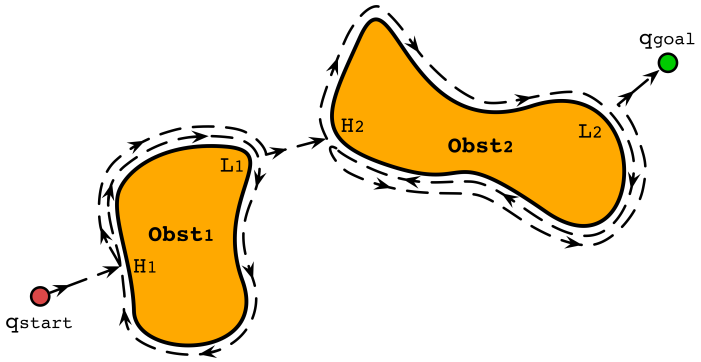
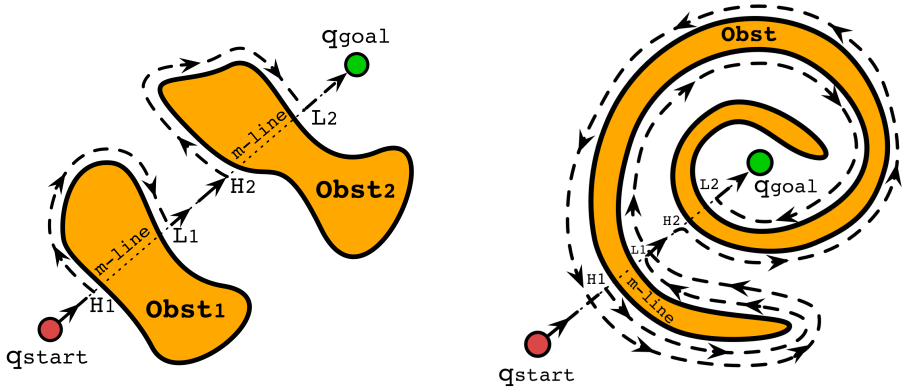


Figure 5: Reactive and sensor-based method Bug1

Bug2 [92] attempts to be an improved version of the previously explained algorithm. It establishes a straight line, sometimes referred as the *m-line* [20], which connects the start and goal positions. With this algorithm, the robot starts moving towards the desired goal by following the *m-line* until it reaches the goal or finds an obstacle. As occurs with Bug1, if the robot is dealing with an obstacle, it first marks that position as the *hit point* (H_i) and then changes its behavior to circumnavigate the obstacle. However, with Bug2 the robot does not necessarily travel the entire obstacle boundary, but instead stops when reaching another point in the *m-line*; if such a point is closer to the goal than the previous *hit point*, the robot marks this position as *leave point* (L_i) and, from there, continues following the *m-line* towards the goal. This procedure is repeated until reaching the specified goal. Even though Bug2 does not require to completely circumnavigate the obstacles, there are environments where the robots do require to travel the entire boundary; in these cases Bug2 may generate longer paths than those calculated by Bug1, and therefore is less efficient. Figure 6 depicts an example of Bug2 solving start-to-goal queries in both simple and complex scenarios.

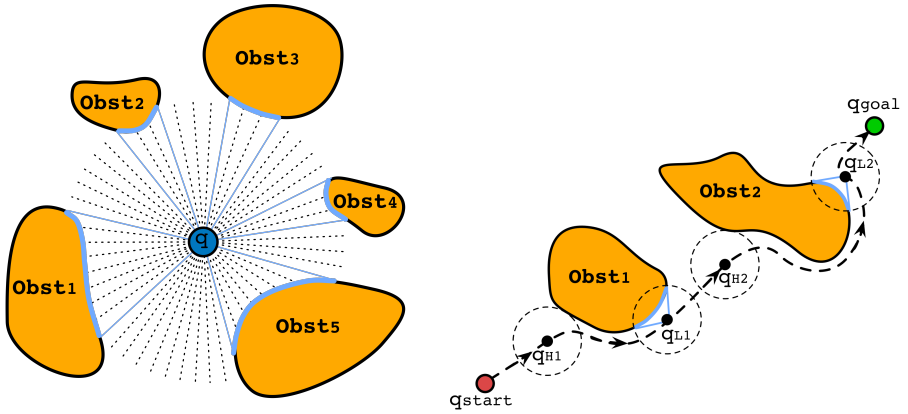
Finally, the Tangent Bug algorithm [61] was proposed as an improved alternative for Bug1 and Bug2. In this version, the robot is assumed to be equipped with a non-zero range sensor, which permits detecting in advance not only the obstacles, but also their continuous boundaries. This way, when the robot is navigating towards the goal and faces an obstacle, it can determine the discontinuities in the boundaries or the limits of the perceived area, which are marked as *endpoints*. Then, in order to avoid the obstacle, the robot checks which of the *endpoints* minimizes the distance to the goal and starts moving towards it. The procedure is repeated until the obstacle is no longer perceived, at which point the robot can continue



(a) Bug2 simple scenario (b) Bug2 complex scenario

Figure 6: Reactive and sensor-based method Bug2

moving towards the goal following a straight line. Figure 7 depicts an example of Tangent Bug solving a start-to-goal query.



(a) Boundaries detection (b) Execution

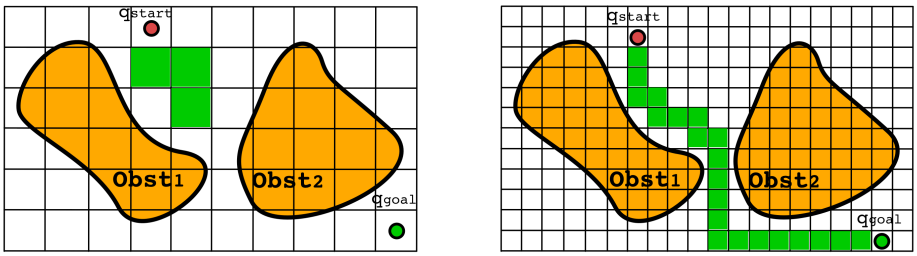
Figure 7: Reactive and sensor-based method Tangent Bug

2.3 SEARCH-BASED (GRID-BASED) PATH PLANNING

Search-based planning is a path planning approach that utilizes graph search methods to find collision-free paths over a discrete version of $\mathcal{C}$. For doing so, these methods overlay a grid on $\mathcal{C}$ and assume that each collision-free configuration corresponds to a point on the grid, which is why they are also called grid-based methods. Over that grid, the robot is allowed to move from one point to any other adjacent point as long as the line between them is proved to be collision-free (i.e., is contained within $\mathcal{C}_{free}$). Hence, using this approach to solve, for instance, a start-to-goal query requires coping with two problems: how to correctly discretize $\mathcal{C}$ to establish the grid, and how to search a path from the start point to the goal point (configuration) over such a grid.

For the first problem, it is important to correctly define the grid resolution, which mainly depends on the environment and the problem's

requirements. Coarser grids, for example, will permit faster searches, but may fail to find paths when dealing with narrow passages in $\mathcal{C}_{free}$ (see Figure 8). Finner grids, on the other hand, will allow solving queries in more complex scenarios, but may be computationally too expensive for online applications. Once the grid is established, there are different methods to calculate an optimal path, some of which are described in this section.



(a) Coarse grid (b) Fine grid

Figure 8: Grid-based method solving a start-to-goal query. (a) Although coarser grids decrease the computing time, they may fail when dealing with narrow passages where finer grids could succeed (b).

2.3.1 *Dijkstra's Algorithm*

Presented in 1959, Dijkstra's algorithm [23] is one of the first widely known methods used to find a path in a graph from one node to another. The algorithm uses a weighted graph $\mathcal{Q}$[1] to determine which path that connects two nodes has of the lowest cost. While this method has been used in different areas and applications such as network routing protocols, in robotics it has been applied to path planning problems. There, the associated edge cost may correspond to different optimization metrics such as distance, energy, time, etc., that can be considered when calculating the optimal path for a start-to-goal query.

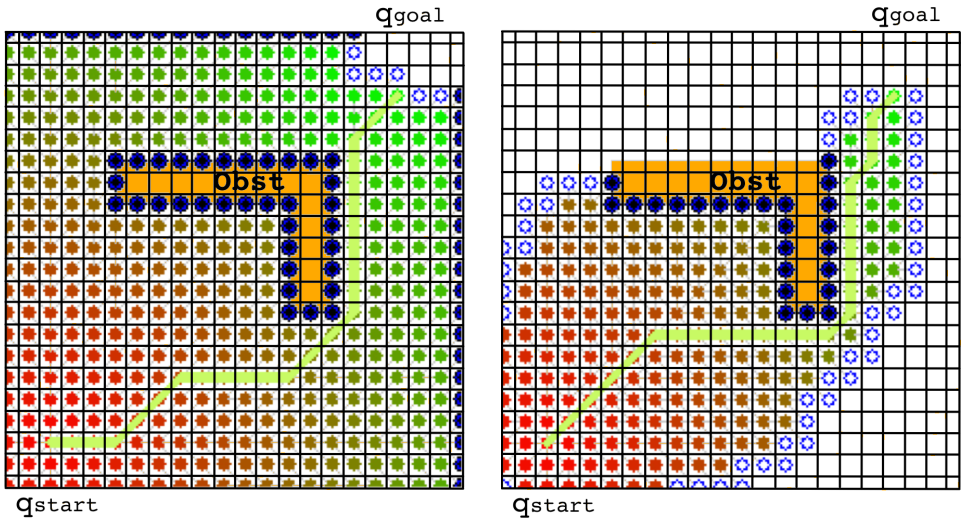
In order to find the optimal path, the algorithm iteratively executes the following steps until reaching the goal (q_{goal}): 1) Setting an initial cost to all nodes, specifically zero to the initial one (q_{start}) and infinity to all others. 2) Establishing q_{start} as the current node and marking the rest as unvisited. 3) Calculating the cost for the current node's unvisited neighbors (equal to the current node's cost plus the edge to the neighbor cost), as well as updating any previously calculated node cost if the new value is lower. For example, if node B cost was x when current node was A, but the new cost is y when current node is C, and $y < x$, then node B cost will be now y. 4) After calculating the cost of all its unvisited neighbors, current node is marked as visited. 5) If the q_{goal} has been marked as visited, the algorithm has finished, otherwise, the new current node will be the unvisited node with the lowest cost, and will be processed as indicated from step 3). Finally, the least cost path can be obtained with backtracking. Figure 9a presents an example of executing this method.

1 A weighted graph is one in which numerical values, or weights, are assigned to its nodes and edges.

2.3.2 *A**

In 1968, Peter Hart et al. described an extension of Dijkstra's algorithm called A* [45], which incorporates a heuristic that permits estimating the cost of paths from any node of the graph to the goal. Because of this characteristic, A* is considered an informed search algorithm; this means that it always attempts to find the path by firstly evaluating those nodes with the minimum estimated cost according to the heuristic. As occurs with Dijkstra's algorithm, A* starts from a weighted graph $\mathcal{Q}$, in which each node represents a different configuration contained in $\mathcal{C}_{free}$ and the edges correspond to collision-free paths between configurations. Then, it builds a search tree, rooted at q_{start}, by expanding different paths, one step at a time, until one of them ends at the desired q_{goal}. The main difference with respect to Dijkstra's algorithm is the order in which each partial path is expanded. In the case of A*, it expands the node q_i that minimizes the total cost $f(q_i) = g(q_i) + h(q_i)$, which combines the cost required to reach the node from the start configuration $g(q_i)$ with the estimated heuristic cost from the node to the goal $h(q_i)$.

While both Dijkstra's algorithm and A* can generate optimal solutions, the latter can result in a more efficient search by reducing the number of nodes required to be visited in order to determine the solution path (see Figure 9b). However, it is important to know that an incorrect heuristic will also provide a valid solution, but it may be suboptimal. For this reason and in order to produce an optimal path, the heuristic has to be optimistic, or admissible, which means that the estimated cost to the goal has to be lower or equal to the real cost [20]. Finally, it is also important to note that in case no heuristic is provided, i.e., $h(q_i) = 0$, A* behaves as Dijkstra's algorithm. Algorithm 1 presents the pseudocode for A* and also provides a general idea for Dijkstra's algorithm explained in the previous section.



(a) Dijkstra algorithm (b) A*

Figure 9: Grid-based methods solving a start-to-goal query. (a) Dijkstra's algorithm requires an exhaustive search to determine the shortest path to the goal. (b) A* requires exploring less cells since it uses a heuristic to guide the search. Image credit: modified from Wikipedia.

Algorithm 1 : A*

Input :
q_{start}: Start configuration.
q_{goal}: Goal configuration.
$\mathcal{Q}$: Graph of configurations q, such that $q \in \mathcal{C}_{free}$

begin
  forall $q \in \mathcal{Q}$ **do**
    $g(q) = \infty$
  $g(q_{start}) = 0$
  UNVISITED = `PriorityQueue()`
  UNVISITED.`insert`$(q_{start}, g(q_{start}) + h(q_{start}, q_{goal}))$
  while $\arg\min_{q \in \text{UNVISITED}} (g(q) + h(q, q_{goal})) \neq q_{goal}$ **do**
    $q \leftarrow$UNVISITED.`popWithMinCost()`
    forall $q' \in \mathcal{Q}$.`Neighbors(`q`)` **do**
      if $g(q') > g(q) + c(q, q')$ **then**
        $g(q') = g(q) + c(q, q')$
        UNVISITED.`insert`$(q', g(q') + h(q', q_{goal}))$

2.3.3 *Dynamic A* (D*)*

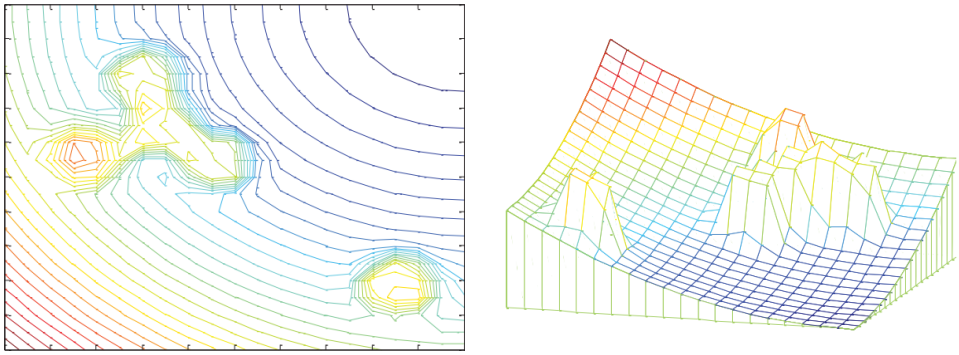
The aforementioned search-based algorithms, at least in their original versions, are intended for planning paths in static environments. Nonetheless, there are situations, especially in mobile robotics applications, in which elements of the environment may change. For those cases, Anthony Stentz proposed the dynamic A* (D*) [118, 119], an incremental search-based algorithm that plans collision-free paths using a similar strategy as A*. The difference is that it also allows to replan according to changes observed in the surroundings while the robot follows the path to the goal. Its most important characteristic is that it locally repairs the path, which is more efficient that invoking multiple times A* to find a new valid path. Contrary to Dijkstra's algorithm and A*, that both search paths from the start to the goal configuration, D* expands nodes by searching backwards from the goal until the node to be expanded coincides with the start configuration, at which time the search is concluded. At the moment, D* and some of its variants are probably the most used search-based methods in mobile robotics. Some of those extensions and applications will be presented in Section 2.7.

2.4 POTENTIAL FIELDS

Even though search-based methods have proved to be successful in 2D and 3D workspaces for some terrestrial, aerial and even aquatic robotic applications, those exhaustive methods suffer from scalability issues in problems involving high-dimensional configuration spaces. Another important drawback is the necessity of establishing a grid over $\mathcal{C}_{free}$, which means discretizing the C-Space, thus limiting the possible and available solutions

according to the chosen resolution. An alternative approach is the use of potential functions.

Originally proposed by Oussama Khatib in 1985 [71, 72], potential functions, also known as *potential fields*, constitute a reactive approach for path planning, which attempts to guide a robot from an initial configuration to a goal configuration while avoiding obstacles. A *potential field* basically defines a real-valued function $U : \mathbb{R}^m \to \mathbb{R}$, which is composed of an attractive component $U_a(q)$ that pulls the robot towards the goal, and a repulsive component $U_r(q)$ that pushes the robot away from the obstacles. This function can be viewed as the total energy $U(q) = U_a(q) + U_r(q)$, which means that the total force applied by the potential field to the robot is defined as the negative of the vector gradient, i.e., $f(q) = -\nabla U(q)$, where $\nabla U(q) = DU(q)^T = \left[\frac{\partial U}{\partial q_1}(q), \ldots, \frac{\partial U}{\partial q_m}(q)\right]^T$. In other words, the gradient $\nabla U(q)$ establishes the force required at any $q \in \mathcal{C}$, in order to guide the robot throughout a collision-free path towards the goal. Figure 10 presents an example of the two components of a potential field and the total potential field.



(a) Top view (b) Perspective view

Figure 10: Example of potential field. The obstacles can be observed as a repulsive component of the whole potential. Image credit: Liu et al. [88].

Once the potential function and the corresponding gradient have been established, the solution path to a start-to-goal query can be incrementally obtained by using the gradient descent, which is a widely-known algorithm in solving optimization problems. Having q_{start} as the initial configuration, the algorithm iteratively calculates a new configuration by moving in the direction opposite to the gradient until the gradient is zero. The pseudocode for this method is presented in Algorithm 2.

However, the gradient descent algorithm using potential functions as explained above does not guarantee finding or converging to a solution for a start-to-goal query. This happens because it may reach a local minimum of $U(q)$ that may not correspond to q_{goal}. To deal with such situations, Barraquand and Latombe proposed the randomized path planner (RPP) [4, 5], which is an algorithm that makes use of the gradient descent together with random walks and backtracking in order to avoid issues related to local minima. However, it is important to note that RPP effectiveness is highly dependent on parameter tuning. Another important aspect to highlight is that RPP was one of the first methods to use a stochastic or random ap-

Algorithm 2 : Gradient Descent

Input :
q_{start}: Start configuration.
$\nabla U(q)$: Gradient of the potential field.

```
1 begin
2     q(0) = q_start
3     i = 0
4     while ∇U(q(i)) ≠ 0 do
5         q(i+1) = q(i) − ∇U(q(i))
6         i = i+1
```

proach for path/motion planning; algorithms with this characteristic will be discussed more in detail in Section 2.6.

2.5 ROADMAPS

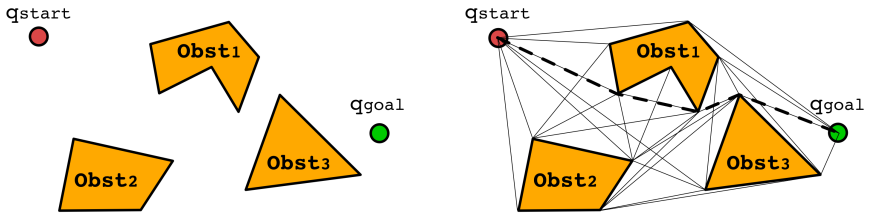
So far, the methods and algorithms presented above attempt to solve a single start-to-goal query, which means they calculate a path that connects a start configuration and a goal configuration. For doing so, they incrementally search a path towards the goal while avoiding, at the same time, collisions with the obstacles. Nonetheless, there are some applications in which the path/motion planner is intended to solve more that one query. For those cases, it makes sense to have a map that contains the information about all feasible routes, and that can also be used more than once to solve multiple start-to-goal queries. There are different alternatives to define a map that can be used for path/motion planning, including topological, geometric, and grid-based representations.

This section reviews methods that use a class of topological maps called *roadmaps* [16, 83]. A roadmap (RM) is defined as a subset of the C-Space that results from the union of curves, in which any q_{start} and q_{goal} contained in $\mathcal{C}_{free}$ can be connected by a path that meets the following properties [20]: 1) **Accessibility** - there is a path from $q_{start} \in \mathcal{C}_{free}$ to some $q'_{start} \in \mathcal{RM}$. 2) **Departability** - there is a path from some $q'_{goal} \in \mathcal{RM}$ to $q_{goal} \in \mathcal{C}_{free}$. 3) **Connectivity** - there is a path in $\mathcal{RM}$ that connects q'_{start} and q'_{goal}.

2.5.1 *Visibility Graphs*

Visibility graphs are one of the alternatives in defining a roadmap. Assuming a 2D C-Space with polygonal obstacles, the set of the visibility graph nodes (v_i) is composed of q_{start}, q_{goal}, and all the vertices of the obstacles. Its edges, e_{ij}, are straight-line segments that can connect any possible combination of two nodes v_i and v_j, without colliding with the obstacles ($e_{ij} \in \mathcal{C}_{free}$). Once the visibility graph is fully defined, the solution path can be obtained by conducting any graph-based search method, such as those explained in Section 2.3. While Figure 11 depicts an example of a

visibility graph and a start-to-goal query solution over it, Algorithm 3 presents the pseudocode to build a visibility graph.



(a) (b)

Figure 11: Example of a visibility graph. (a) A 2D C-Space that contains polygonal obstacles, a start configuration, and a goal configuration. (b) A roadmap is built by connecting any possible combination of two vertices without colliding with the obstacles. The start-to-goal query is solve using a graph search method.

Algorithm 3 : Visibility Graph

Input :
q_{start}: Start configuration.
q_{goal}: Goal configuration.
World: n Obstacles.
Output :
Roadmap ($\mathcal{RM}$): Visibility Graph(VG) = (V, E).

```
1  begin
2      V = {}
3      E = {}
4      for i = 1 : n do
5          V.addNodes(Obst(i).getVertices())
6      for i = 1 :V.getNumNodes() do
7          for j = 1 :V.getNumNodes() and j ≠ i do
8              v_i ← V(i)
9              v_j ← V(j)
10             e_ij ← findStraightLine(v_i, v_j)
11             if isCollisionFree(e_ij) then
12                 E.addEdge(e_ij)
```

2.5.2 *Generalized Voronoi Diagrams*

Another alternative to build a roadmap for path/motion planning is the generalized Voronoi diagram (GVD). The GVD is defined for a set of points called *sites*; given a particular site, the set of points closest to it is called a *Voronoi region*. Finally, the Voronoi diagram is the set of points that are equidistant to at least two sites [3]. In path/motion planning applications, the GVD defines the sets of points equidistant to at least two obstacles. This means that the sites are the center of the obstacles to be avoided, and the

edges correspond to the possible channels that maximize the distance to the obstacles [20]. Figure 12 displays an example of a planar GVD.

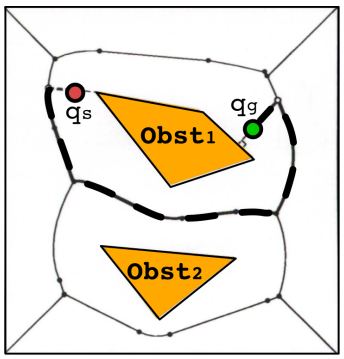


Figure 12: Example of a Voronoi diagram

2.6 SAMPLING-BASED ALGORITHMS

Some of the methods presented in previous sections, such as potential fields, roadmaps, and cell decomposition, require an explicit representation of $\mathcal{C}_{free}$ in order to find a collision-free path from a start configuration to a goal configuration. Nonetheless, there are some situations in which building such a representation is not possible or is computationally intractable, for instance when dealing with high-dimensional configuration spaces. For those cases, sampling-based methods have been demonstrated to be effective.

Firstly developed during the 90's, *sampling-based algorithms* are an alternative approach that aims to create a sampled (discrete) representation of C-Space that captures the connectivity of the regions of $\mathcal{C}_{free}$. These methods exploit the fact that, while explicitly building $\mathcal{C}_{free}$ is expensive, checking a given configuration for collisions can be done quickly. To build such a discrete representation, they firstly generate random samples q_{r_i} from C-Space, which are checked for collision in order to ensure that $q_{r_i} \in \mathcal{C}_{free}$. Secondly, they interconnect the random and collision-free configurations, thus establishing different routes (paths) to solve single or multiple start-to-goal queries. This two-stage (sampling and connecting) approach that uses a collision checking routine to validate samples, also allows generalizing the path/motion planning problem. This is done by separating the algorithm from the specific geometric representation of the environment.

However, there is one important characteristic to be considered when using sampling-based algorithms. Contrary to the methods mentioned previously, sampling-based algorithms weaken the completeness guarantee, which means that they are not capable of notifying if a solution exists. Nonetheless, given that many of these methods are based on random sampling, which is dense with probability one [78], this also implies that with enough points (samples), if a solution exists then the probability of finding it converges to one. In other words, if the algorithm runs for a sufficient amount of time, it will find a solution if there is one. This property is called *probabilistic completeness* [6, 66, 68].

These characteristics have led sampling-based algorithms to be considered the state-of-the-art approach for solving various path/motion planning problems. Albeit there are several methods and variants used nowadays, it is possible to identify those that, at the time, were pioneers and the most relevant ones. This section presents such methods and classifies them according to their capability to solve single or multiple start-to-goal queries.

2.6.1 *Multiple-query Methods*

As it was explained in Section 2.5, roadmaps are data structures that contain all feasible routes that can be used more than once to solve multiple start-to-goal queries. Likewise, there is a sampling-based method called probabilistic roadmap (PRM) that creates a graph attempting to represent the connectivity of $\mathcal{C}_{free}$. PRM was developed simultaneously at Stanford [65, 67] and Utrecht [104], and was jointly presented in 1996 [68].

Nowadays, PRM is one of the most representative sampling-based algorithms. It is mainly composed of a *preprocessing phase* and a *query phase*. During the first phase, the algorithm builds a roadmap, or undirected graph $G = (V, E)$. The set of nodes (V) contains n collision-free samples q_{r_i} (i.e., $q_{r_i} \in \mathcal{C}_{free}$) that are randomly obtained from an uniform distribution[2]. The set of edges (E) corresponds to the collision-free paths from each node q_{r_i} to its k closest nodes q_{r_j}; the connecting paths are calculated by a local planner that checks them for collisions. Algorithm 4 presents the pseudocode for the *preprocessing phase*.

Once the roadmap (graph) has been built, the *query phase* attempts to find a collision-free path between the provided q_{start} and q_{goal}. To do so, PRM tries to connect q_{start} and q_{goal} to their k closest nodes in the graph G. If the connections are successful, a search-based method (e.g., A*, Dijkstra, etc.) attempts to find the shortest path over the graph. If the connections for q_{start} and q_{goal} to the graph are not possible, or if the search-based algorithm fails to find a solution path for the query, it does not necessarily mean that a solution does not exist. As explained above, most sampling-based methods, including PRM, are probabilistic complete, which means that more time may be required to find a solution, if there is one. In this case, it would imply that the number of nodes n have to be increased, as that will generate a more dense probabilistic roadmap.

An important number of variants and extensions have been proposed to deal with different situations in which the originally proposed PRM may fail [20], [78]. A typical example includes a workspace that creates a C-Space with narrow passages. In such a case, a common approach would be oversampling the regions of interest (e.g., narrow passages), thus increasing the probability of finding a solution path. Other extensions that have served as a base for the work developed throughout this thesis, will be discussed in Section 2.7. Finally, Figure 13 depicts PRM solving a start-to-goal query in a 2D scenario for a point-like robot.

2 Uniform distribution is the basic form to obtain the random samples, at least in the basic version of the algorithm. Other distributions have been used in some extensions of the original method.

Algorithm 4 : PRM, preprocessing phase

Input :
n: Number of nodes of the roadmap.
k: Number of closest nodes to attempt connection.
$\mathcal{C}$: C-Space
Output :
Probabilistic roadmap (PRM): $G = (V, E)$.

```
1  begin
2      V = {}
3      E = {}
4      while |V| < n do
5          q_rand = C.generateRandomConf()
6          if C.isCollisionFree(q_rand) then
7              V.addNode(q_rand)
8      for q_ri ∈ V.getNodes() do
9          for q_rj ∈ C.getClosestNodes(q_ri, k) do
10             e_ij ← findPath(q_ri, q_rj)
11             if C.isCollisionFree(e_ij) then
12                 E.addEdge(e_ij)
```

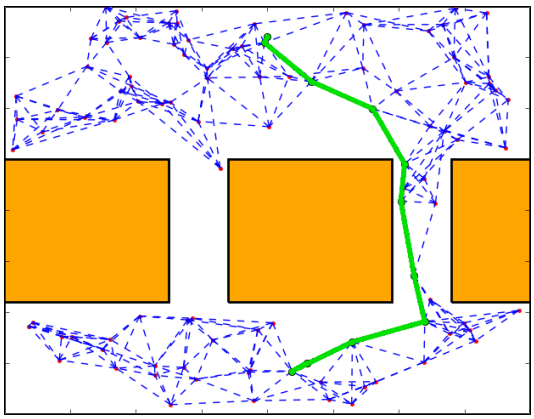


Figure 13: Example of a probabilistic roadmap (PRM) used to solve a start-to-goal query. The PRM was built with 100 random samples, which were attempted to connect to their 6 nearest neighbors.

2.6.2 *Single-query Methods*

There is another group of sampling-based algorithms that is devoted to solving a single start-to-goal query. Such methods conduct an incremental search, which in most cases is achieved by expanding a tree of randomly sampled configurations. The search is generally biased to finding a solution path for one particular start-to-goal query, rather than attempting to completely represent the connectivity of $\mathcal{C}_{free}$. These methods are *unidirectional* when a single tree is expanded from q_{start} until reaching q_{goal} or vice versa. Or, they are *bidirectional* if two trees are expanded, one from q_{start} and other from q_{goal}, until both trees meet at a common point. Choosing q_{start} or q_{goal} as the tree's root depends on the specific problem and scenario, since there can be situations in which expanding from the goal is easier (less constrained) that doing it from the start configuration. There are different sampling-based single-query methods, however, this section presents a brief review of those considered the most relevant.

Randomized Path Planner (RPP)

Section 2.4 presented an approach for guiding a robot from its current position towards the goal position, by following the negative gradient of an artificial potential field. As mentioned there, one of the main disadvantages of this approach is that the vehicle can get trapped in a local minimum that may not correspond to the specified goal. In order to deal with such situations, Barraquand and Latombe proposed in 1990 the randomized path planner (RPP) [4, 5], which uses the strategy of potential fields, but also incorporates random walks to escape local minima. RPP is commonly acknowledged as the first randomized algorithm. Albeit RPP has proved to be successful in many applications, it may fail when coping with narrow passages.

The Ariadne's Clew Algorithm (ACA)

Presented in 1993, Ariadne's clew algorithm (ACA) builds a tree from q_{start} by interleaving the *exploration* of the C-Space and the *search* of a connection between the tree and q_{goal} [10, 97]. During the exploration, the algorithm places a random collision-free configuration as far as possible from the others, therefore guaranteeing resolution completeness. New configurations are selected by using genetic optimization methods. These correspond to those from which a connection to q_{goal} is attempted. The main drawback of this approach is that the exploration of the C-Space is computationally expensive and also requires some parameter tuning.

Expansive-Spaces Tree (EST)

Proposed by David Hsu et al. in 1997, expansive-spaces tree (EST) [55–58] is a single-query method that incrementally builds a tree over the C-Space by interleaving its *construction* and *expansion*. In contrast to what occurs with PRM that computes a roadmap attempting to represent the whole $\mathcal{C}_{free}$, EST tries to sample the region of $\mathcal{C}$ that is relevant in order to solve the specific start-to-goal query. To do so, during the *construction* phase

the algorithm selects the node q to be extended in a way that prioritizes less explored regions, and then it randomly samples a configuration q_{rand} around q. Then, using a local planner, EST calculates the path that connects q and q_{rand}. In case that both q_{rand} and the calculated path are proved collision-free, they will be added to the tree, thus *expanding* it.

Rapidly-exploring Random Tree (RRT)

Within the group of sampling-based single-query algorithms, there is one method that is considered the state-of-the-art, which has been extended, modified and applied to a wide range of applications. This method is known as rapidly-exploring random tree (RRT) and it was firstly presented by Steven LaValle in 1998 [77]. RRT is a tree-based algorithm that has different properties such as rapid exploration of the C-Space, probabilistic completeness, ease of implementation, just to mention some. In 1999, LaValle and Kuffner formally presented the RRT as a path/motion planning method capable of dealing with both geometric and motion constraints. They also proposed a greedy approach to decrease the time required to find a solution by interleaving a random growing of the tree with a biased growing towards the goal [79–81]. They later proposed a bidirectional version called RRT-Connect that extends the basic concept by constructing two RRTs towards each other [75].

Similar to ACA and EST, the basic RRT is mainly composed of two main procedures, *sample* and *extend*. Algorithm 5 presents the first of them, where the tree is incrementally built until a stop condition occurs; such a condition can either be finding a feasible path that reaches the goal close enough, or that a maximum number of iterations has been completed. In each iteration, the RRT attempts to extend the tree towards a randomly sampled configuration q_{rand}. For doing so, the second procedure described in Algorithm 6 finds q_{near}, which is the nearest configuration to q_{rand} in the tree (line 2). Then, a local planner calculates a path of length δ from q_{near} towards q_{rand} (line 3); if the path is proved collision-free, the algorithm generates a new configuration q_{new}, which together with the calculated path are added to the tree (lines 5-6). A typical growth process of an RRT can be clearly observed in Figure 14a, where no goal has been specified and the tree attempts to explore uniformly the C-Space. Figure 14b depicts the RRT solving a start-to-goal query.

2.6.3 *Optimal Planning*

At least in their original formulation, sampling-based algorithms do not guarantee that the calculated path is optimal with respect to a specified cost function. However, more recent contributions have attempted to cope with this situation. In 2010, Jaillet et al. presented the transition-based RRT (T-RRT), which calculates a low-cost path that follows valleys and saddle points in a costmap established over the C-Space [60]. To achieve this, it verifies the quality of the path by using the minimal work criterion. Its key principle is that positive variations of the cost function can be seen as forces acting against motion, and thus producing mechanical work. Contrasted to a standard RRT implementation, an additional transition valida-

Algorithm 5 : sampleRRT

Input :

q_{start} : Start configuration.

q_{goal} : Goal configuration.

$\mathcal{C}$: C-Space.

Output :

Rapidly-exploring Random Tree (RRT): $T = (V, E)$.

```
1 begin
2     V = {}
3     E = {}
4     V.addNode(q_start)
5     while not stopCondition(T, goal) do
6         q_rand = C.generateRandomConf()
7         extendRRT(T, q_rand)
```

Algorithm 6 : extendRRT

Input :

T: an RRT.

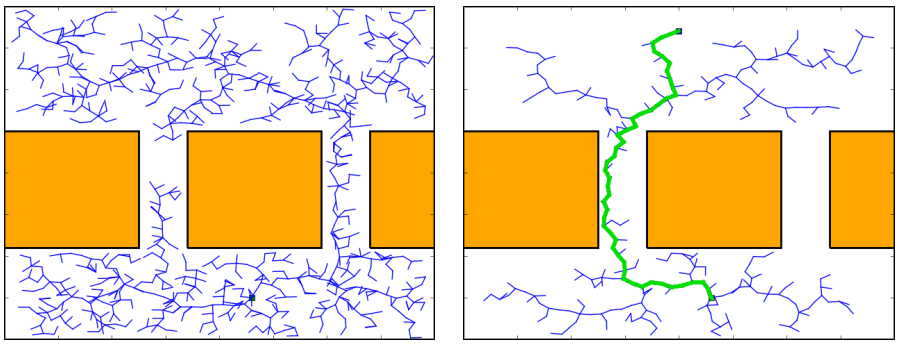
q_{rand}: configuration towards which RRT will be extended.

$\mathcal{C}$: C-Space.

Output :

Result after attempting to extend.

```
1 begin
2     q_near ← T.findNearestNeighbor(q_rand)
3     q_new, collision ←calcNewConf(q_near, q_rand, δ)
4     if collision = FALSE then
5         V.addNewNode(q_new)
6         E.addNewEdge(q_near, q_new)
7         return ADVANCED
8     else
9         return TRAPPED
```



(a) RRT expansion (b) RRT solving a query

Figure 14: Example of a rapidly-exploring random tree (RRT) in a 2D workspace. (a) The uniform and rapid exploration of the C-Space is one of the main characteristics. (b) The expansion of the tree stops once a solution has been found.

tion is performed, which accepts or rejects new potential configurations before adding them into the solution tree.

Nonetheless, the state-of-the-art sampling-based method for calculating optimal paths is the asymptotic optimal RRT (RRT*). In 2010, Karaman and Frazzoli firstly introduced the RRT* and its concept of asymptotic optimality. This property states that the total cost of the solution, measured by a user-defined function, decreases as the number of samples increases [62]. In this approach, new configurations are connected to the closest and best configuration, i.e., the one that guarantees a minimum cost. Furthermore, an additional step of sample reconnection allows improving costs to surrounding configurations (see Algorithm 7). This concept was later extended by the same authors to the PRM in [63], where they formally presented the RRT* and PRM*. Similar extensions have been done to other RRT-based algorithms as T-RRT and its respective T-RRT* version [22]. A typical growth process of an RRT* can be clearly observed in Fig. 15.

Algorithm 7 : extendRRT*

Input :
T: tree of collision-free configurations.
q_{rand}: state towards which the tree will be extended.
$\mathcal{C}$: C-Space.
Output :
Result after attempting to extend.

```
1  begin
2      q_near ← T.findNearestNeighbor(q_rand)
3      q_new, collision ←calcNewConf(q_near, q_rand, δ)
4      if collision = FALSE then
5          addNewNode(T, q_new)
6          Q_near ←findNearestNeighbors(T, q_new)
7          q_min_cost ←findMinCost(T, Q_near, q_new)
8          addNewEdge(T, q_min_cost, q_new)
9          reconnectNearNeighbors(T, Q_near, q_new)
10         return ADVANCED
11     else
12         return TRAPPED
```

2.7 EXTENSIONS AND APPLICATIONS

Most of the aforementioned methods have been widely used in real-world applications not only with manipulator arms, but also with different aerial, terrestrial and aquatic robotic systems. For doing so, some of these methods have been extended and adapted according to the specific applications' requirements. Considering the problem and objectives stated for this thesis in Chapter 1, this section presents a brief review of the most relevant extensions and applications of path/motion planning algorithms, which have permitted conducting autonomous missions under both motion and online computation constraints.

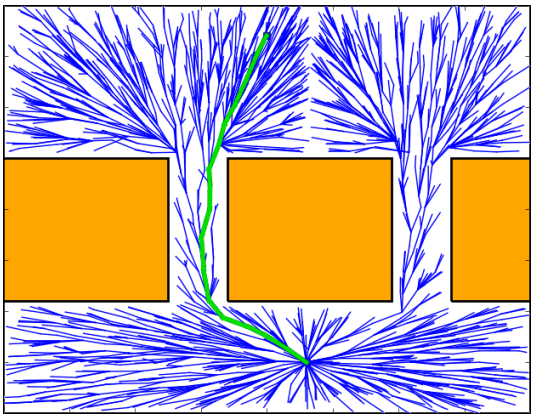


Figure 15: Example of an asymptotic optimal rapidly-exploring random tree (RRT*) in a 2D workspace. The RRT* preserves the uniform and rapid exploration of the C-Space as the standard RRT; however, it can be observed how the additional reconnection step reshapes the tree branches. The expansion of the tree does not stop once a solution has been found. Instead, it keeps improving the solution until either a maximum number of iterations or a maximum time has been reached.

2.7.1 *Path/Motion Planning under Motion Constraints*

As explained in Section 1.2.2.2, planning under motion (differential) constraints has to do with considering the limits of the feasible system's maneuvers. These are described by a set of differential equations, generally expressed as $\dot{q} = f(q, u)$ (where q and $\dot{q}$ are the system state and its first derivative, respectively, and u is the control input). A large body of research has been dedicated to the development and improvement of different approaches attempting to solve planning problems that include this kind of constraints.

When Donald et al. formally introduced the problem of kinodynamic motion planning[3], they proposed the use of dynamic programming to find the shortest path in a directed graph using depth-first search (DFS). In this case, the graph represents a discretization of $\mathcal{C}_{free}$; the edges correspond to trajectory segments obtained after applying an acceleration a (bounded according to the system's capabilities) for a period of time τ (determined by the algorithm) [26].

Other variants of grid-based methods, such as A*, have been also used with similar strategies for discretizing $\mathcal{C}_{free}$. In terrestrial vehicles, for example, the most remarkable contributions were a result of the DARPA Grand Challenge [126][4]. In one of those works, Likhachev and Ferguson used a multi-resolution lattice state space (a $\mathcal{C}_{free}$ discretization), where states represent configurations, and connections between them represent feasible paths (i.e., those that consider kinematic constraints). Then, one A* variant (called AD*) would find paths over the lattice [84]. Similarly, Dolgov et al. presented an approach in which $\mathcal{C}_{free}$ is also discretized and paths are found by running another A* variant (Hybrid-State A*). This is

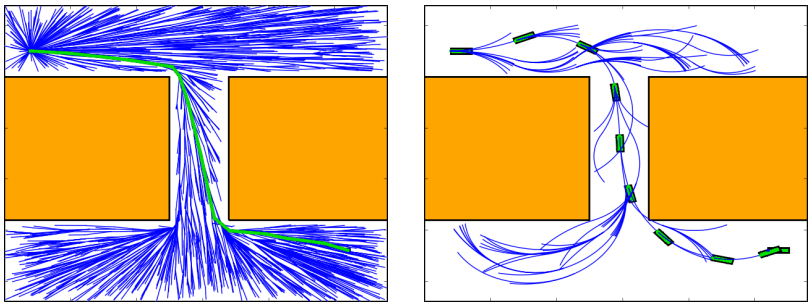
3 Kinodynamic motion planning alternatively refers to motion planning under motion or differential constraints.

4 The DARPA Grand Challenge is a competition of autonomous vehicles, funded by the Defense Advanced Research Projects Agency (DARPA).

guided by two heuristics, one that considers the vehicle's non-holonomic constraints and a second one that computes the Euclidean distance. This approach also connects the states (configurations) by restricting the control inputs according to the feasible (doable) maneuvers of a car-like system, which are expressed by non-holonomic (kinematic) constraints [24, 25].

In all these grid-based approaches, the main drawback is given by the $\mathcal{C}_{free}$ discretization, which establishes a finite set of possible maneuvers, thus limiting the number of possible solution paths. As occurs with geometric path planning approaches, the availability of a solution depends on the grid resolution and, in this case, on the number of maneuvers considered according to the motion constraints.

In what concerns sampling-based methods, EST and RRT were initially conceived to cope with motion constraints [58, 81]. In their original versions, the differential equation of motion is used to generate new nodes (states) during the *expansion* of the tree (e.g., in Algorithm 6, line 3)[5]. Figure 16 depicts two sampling-based algorithms: RRT* and RRT in the process of solving a start-to-goal query in a 2D workspace under both geometric and motion constraints.



(a) Geometric Constraints (b) Differential Constraints

Figure 16: Start-to-goal query solution in a 2D workspace, where obstacles appear in orange, free space in white, and the tree of collision-free configurations in blue. The solution (in green) is calculated by (a) an RRT* under geometric constraints for a point-like system $(\mathcal{C} = \mathbb{R}^2)$, and (b) an RRT under differential constraints for a car-like system $(\mathcal{C} = \mathbb{R}^2 \times \mathcal{S})$.

There are several successful applications based on this approach. In terrestrial vehicles, one of the most relevant application was presented by Kuwata et al. during The DARPA Urban Challenge. They proposed an approach known as closed-loop RRT (CL-RRT), which not only considers the vehicle's motion model, but also includes the controller's dynamic behavior [76]. Another interesting application was done with aerial systems, in which Müller et al. proposed to use an A* for globally finding the collision-free path which, at the same time, guides an RRT expanded under differential constraints. This latter guarantees finding paths that meet the system's motion constraints [100].

5 While tree and graph nodes are generally referred to as configurations in geometric path planning, in kinodynamic motion planning they are commonly called states. However, this latter term has been indistinctly used by some authors to refer to them in either case.

2.7.2 *Online Path Planning*

As discussed in Chapter 1, potential and new applications for AUVs involve navigating unknown or undiscovered environments that require endowing the vehicles with the capability of (re)planning online while, at the same time, they explore the environment. There are different extensions that can contribute to achieving this requirement, however this section focuses on two of them. The first is the *anytime* computation, a characteristic incorporated to different kind of algorithms, including both search-based and sampling-based methods. The second refers to *lazy collision checking*, a strategy specifically used with sampling-based methods. These two characteristics have served as a basis for the proposed approach of this thesis.

Anytime Planning Algorithms

In some situations finding a definite path to solve a start-to-goal query in a finite and deterministic period of time is not possible. It may occur either because of the complexity of the task (i.e., more computation time is required) or because vehicles deal with partially known or dynamic environments. In either case, a common approach is to use an *anytime* algorithm that is capable of providing the best partial solution when the available time is over [141],[21]. The most relevant and well-known planning algorithms have been extended based on this strategy.

Likhachev et al. have studied search-based algorithms, including their extensions for *anytime* computation. They presented the anytime repairing A* (ARA*), a variant that rapidly calculates a suboptimal path to the goal using a loose bound, which is later tightened to progressively improve the path [85]. While this approach obtains a fast solution, it also permits improving it if additional time is available. Nonetheless, this approach results useful when full and accurate information about the environment is available, otherwise it may require recalculating the whole path if the environment changes. For those cases, i.e., when coping with dynamic environments, it is better to use an incremental search-based method as D*, which allows locally repairing (replanning) the path when new information of the environment is provided (see Section 2.3.3). However, in its original version, D* lacks the *anytime* property. In order to improve D*'s characteristics, Likhachev et al. also presented the anytime dynamic A* (AD*). This is a variant that not only replans if required, but also improves simultaneously the available solution path [86]. A detailed discussion of these *anytime* search-based algorithms is provided in [87]. Some applications for vehicles that not only require online/anytime computation, but also navigate under motion constraints are also presented in [84].

In the case of sampling-based methods there are also different extensions for *anytime* computation. Belghith et. al, for example, proposed the flexible anytime dynamic PRM (FADPRM), which is an approach that combines both: a standard PRM for constructing the roadmap and an AD* for finding a solution path. This latter one extends the original PRM by permitting not only *anytime* calculation, but also a progressive improvement of the resulting path. An important characteristic of this approach is the possibility to establish zones with different values of desirability that allow

the sampling strategy to be influenced in order to generate less awkward (inefficient and not smooth) paths [8]. A similar approach presented by van den Berg et al. uses PRM to represent the static portion of C-Space and an AD* to deal with the dynamic elements [9].

On the other hand, and because of its incremental nature, randomized tree-based methods, such as RRT, are more commonly used for applications in partly known or dynamic environments, where online and *anytime* computation is required. Ferguson et al. proposed an anytime RRT-based algorithm that calculates an initial path and its cost using the standard RRT, thus ensuring that a first solution is found in the shortest time possible. Then, a modified RRT iteratively generates a series of new solutions that are guaranteed to have a lower cost. The algorithm executes until a stop condition is reached, e.g., when the best available solution path needs to be provided [34, 35]. Other RRT variants as RRT* were also formulated as *anytime* algorithms [64].

Lazy Collision Checking

Even though PRM was originally intended for multi-query applications, Bohlin and Kavraki presented a modified version known as Lazy PRM, which minimizes the execution time by reducing the quantity of collision checking callbacks when solving a specific (single) query [12, 13]. This variant builds a roadmap just as a standard PRM does, but it assumes that all the nodes (configurations) and edges (paths between configurations) are collision-free, leaving the collision detection to the final stage, i.e., when it must find the shortest path between an initial and a final configuration. If a collision is found, the associated nodes and edges are discarded (eliminated) and a new short path is calculated. The authors also suggested an optional *enhancing roadmap* step when collisions are indeed detected. This consists in including more samples around the discarded nodes.

This strategy of delaying the collision detection is known as *lazy collision checking*, and has been used in different applications, especially those with online computation requirements. Bekris and Kavraki, for instance, proposed and validated a tree-based planning framework for terrestrial vehicles, where the states validity is only checked once a path between the start and the goal configuration has been found [7]. Another example is the one presented by Vahrenkamp et al., where this strategy was used to speed up the motions calculation for humanoid robots (with many degrees of freedom, i.e., high dimensional C-Space) [128].

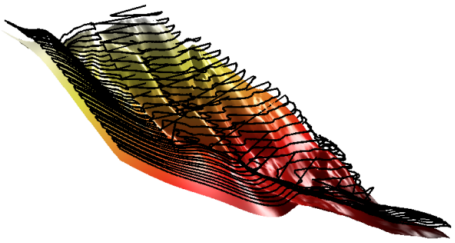
Finally, it is important to note that *lazy collision checking* has served as a base for one of the extensions proposed and used in this thesis, which will be explained in detail in Chapter 5.

2.8 PATH/MOTION PLANNING FOR AUVS

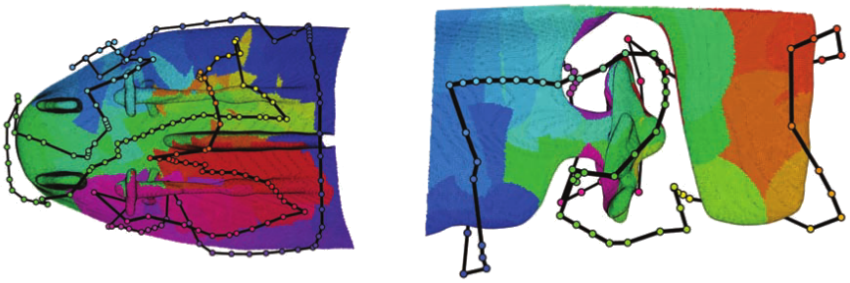
An important aspect to consider when comparing the path/motion planning approaches for AUVs is their application. Based on this, the different contributions can be classified into two main categories. The first group gathers those applications that require coverage path planning (CPP) techniques, which are commonly applied to guide AUVs over survey tasks. The

most common examples within this group include coverage missions used for creating in detail bathymetric maps of the seabed [33, 38, 40, 41], detecting potential targets (such as underwater mines [117, 136]), and inspecting artificial structures (such as in-water ship hulls [29, 30, 49, 54]), as well as natural marine formations [37, 42].

A common characteristic in all these approaches is that the planner is provided with preliminary information of the target area or structure (see Fig. 17). This may include its location and shape. Based on this, the CPP algorithm defines a survey path that, in some cases, is reshaped or refined online according to the data obtained during the mission execution. This characteristic implies that most of the computation is done offline, i.e., before conducting the mission. Most recent work presented by Vidal et al. proposes a novel approach to conduct inspection tasks without preliminary information of the target. Results are, however, still limited to 2D motions (at a constant depth) [130].



(a) Coverage path over of a natural formation. Image credit: Galceran et al. [40]



(b) Coverage path of a ship hull. Image credit: Hover et al. [30, 54]

Figure 17: Coverage path planning (CPP) algorithms are normally used to plan routes to inspect different kind of structures. In most of these applications the planner has preliminary information. This may include a 2D/3D map of the area or structure to be inspected.

The second group of AUV applications, on the other hand, gathers those that are focused on safely and efficiently guiding the vehicle from one initial position to a specified goal. For doing so, different strategies, such as those explained throughout previous sections, have been applied to underwater vehicles. In fact, these start-to-goal methods are commonly used as low-level motion planners required for the aforementioned coverage applications. The work presented throughout this thesis seeks to make contributions to this group of start-to-goal applications, therefore it is important to identify the main characteristics between the different approaches used.

2.8.1 *Start-to-goal Path/Motion Planning for AUVs*

One characteristic to consider in a start-to-goal planner for AUVs, and yet not an obvious one, is the capability of conducting 2D and 3D motions. Although AUVs operate in 3D workspaces, in a significant number of applications the vehicles navigate either at a constant altitude or at a constant depth [98, 106, 108, 115], thus simplifying considerably the motion planning problem. There are, however, some contributions that have presented alternatives in modelling and planning 3D (and therefore 2D) AUV motions.

From the approaches that address 3D motions, it is still necessary to make a distinction between those that have been used for underwater gliders, and those for propeller-driven AUVs (as the ones in this work). In the latter case, the available contributions have made use of different approaches such as potential fields [112, 132], genetic algorithms [1, 50, 122], as well as sensor-based [53, 140], grid-based [18, 73, 112], and sampling-based methods [14, 98, 108]. However, in some of these situations, the vehicle operates in open sea areas, where they do not usually have to deal with obstacles, narrow passages, or high-relief environments. This kind of constraints corresponds to the new and potential AUV applications presented in Chapter 1.

2.8.2 *Online Motion Planning for AUVs through Unexplored Environments*

New AUV applications require a path/motion planner to safely guide the vehicle through unexplored and challenging environments. This implies meeting online computation limitations while, at the same time, considering the vehicle's motion capabilities. Little research on this area, especially for underwater vehicles, has been addressed. In what concerns to generating AUV feasible paths, i.e., those that meet the motion constraints, a first group includes those approaches that use sensor-based methods either to navigate through unknown underwater environments [140], or to follow the terrain shape of a given bathymetric map [53] (see Fig. 18). In both cases, the vehicle is assumed to be equipped with a looking-forward sonar to reactively avoid collisions. Such maneuvers are calculated by a local planner that meets the AUV's kinematics or dynamics. An important characteristic of this kind of approach is the lack of global knowledge of the environment (i.e., a map is not incrementally built), which can cause the vehicle to get trapped in complex scenarios.

A similar reactive approach establishes a set of inequality constraints that describe the obstacles as convex regions contained in the C-Space. Moreover, the initial configuration is treated as the starting point of a nonlinear search, where the goal configuration is assumed to be a unique global minimum of the objective function. The start-to-goal query is then solved as an optimization problem, in which a local planner takes into account the vehicles' constraints. This strategy was one of the first online obstacle avoidance approaches for underwater vehicles; it used a real-world dataset of acoustic images obtained by a remotely operated vehicle (ROV) equipped with a multibeam looking-forward sonar. Its validity was demonstrated by

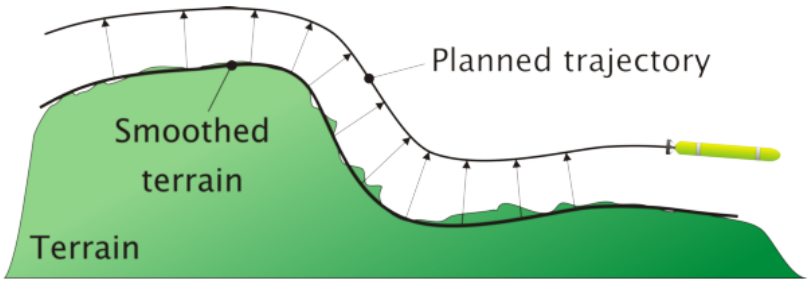
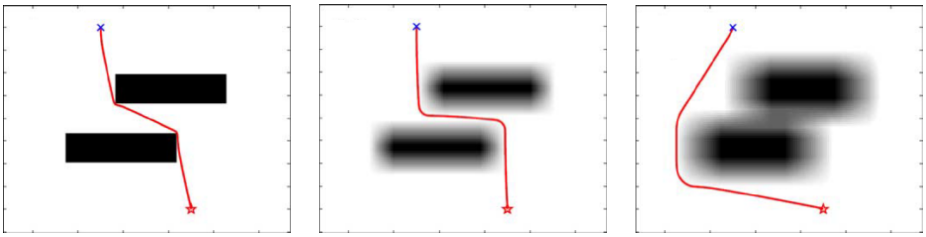


Figure 18: Sensor-based planning for AUVs under motion constraints. The vehicle trajectory can be either preplanned or incrementally calculated to maintain a desired distance from the terrain. The resulting trajectory is fit to satisfy curvature constraints that approximate the AUV motion constraints. Image credit: Houts et al. [52, 53].

guiding a simulated ROV. However, the capability of simultaneous mapping (detection) and planning online was not proven [106].

The formulation that represents obstacles as convex regions has likewise been used either to represent obstacles detected online, which triggers collision avoidance maneuvers [110], or to approximate the terrain shape that must be followed by the vehicle [101]. In both cases, the low-level controller attempts to generate feasible (doable) trajectories by using the AUV kinematic equations and spline-based interpolation techniques, respectively. However, the main drawback of these approaches is the difficulty in creating a convex representation of complex obstacles.

Another common strategy used in some of the aforementioned methods, is mathematically making the solution paths more suitable for a motion-constrained AUV. Pêtrès et al., for instance, proposed a fast marching (FM)-based approach to find collision-free paths, which are smoothed by a cost function that contains kinematic and curvature constraints [107] (see Fig. 19). Likewise, another example based on genetic algorithms (GAs) finds a valid route to the goal by using basis spline (B-spline) curves, thus seeking to generate more feasible trajectories for AUVs [19].

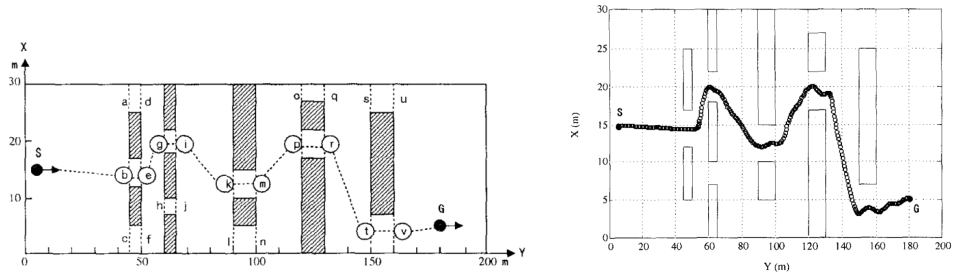


(a) (b) (c)

Figure 19: Start-to-goal query that is solved by a FM-based method. The solution path is smoothed by a cost function to approximate the AUV motion constraints. (a) The initial optimal path. (b) The effect after smoothing the path. (c) The use of the cost function can merge the obstacles, thus discarding possible solutions. Image credit: Pêtrès et al. [107].

There is another group that gathers different grid-based approaches. Sequeira and Ribeiro, for example, presented a two-layer framework that is composed of a high-level planner (HLP) and a low-level planner (LLP). The

HLP creates a visibility graph using the information of the known obstacles, sea currents, and specified waypoints of the mission. Furthermore, the energy required to move between the graph nodes corresponds to the edge weights. The global and optimal geometric route to the goal is then found by Dijkstra's algorithm. Finally, in order to calculate the vehicle's maneuvers between the different solution segments, the LLP uses an artificial potential field (AFP). This way the total artificial force includes a 3D double integrator that takes into account the AUV motion constraints [112]. There are other similar two-layer approaches, where the global path planning problem is tackled with a grid-based method, and the local motion planning deals with the AUV constraints [2] (see Fig. 20). The main disadvantage of this approach is that the grid-based layer generally requires *a priori* information of the environment, e.g., a navigation map.



(a) (b)

Figure 20: (a) Global path planning solved by a grid-based method, (b) which is adjusted by a local planner that takes into consideration the AUV motion constraints. Image credit: Arinaga et al. [2].

A group of recent contributions includes those that use sampling-based planning algorithms. The most common approach builds a tree of collision-free configurations, which are obtained by integrating the differential equation that describes the AUV dynamic behavior [14, 46, 124]. This strategy was briefly introduced in Section 2.7.1, but its use with a torpedo-shaped AUV will be explained in more detail in Chapter 3.

Finally, there is a geometric alternative to generate feasible paths for motion-constrained AUVs. It consists in utilizing the Dubins paths [27], which establishes a set of six maneuvers (RSR, RSL, LSR, LSL, RLR, LRL, where R states for Right, L for Left, and S for straight) to connect two configurations $q_i, q_j \in \mathcal{C} = SE(2)$. It has been typically used with car-like systems. Further discussion of Dubins maneuvers and their use with AUVs will be presented throughout Chapters 3 and 4.

Another characteristic required for the new AUV applications is the capability of mapping and planning safe paths, simultaneously and online, as the environment is incrementally explored. Apart from the already explained work by Petillot et al., which in fact does not prove online mapping and planning capacity [106], Maki et al. proposed an online path planning method that uses landmarks to guide an AUV. Nonetheless, their approach does not permit replanning and, furthermore, results were obtained in a controlled environment (i.e., in a water tank) [93].

This chapter has presented an extensive review of the most common path/motion planning approaches, and those that have been used with

underwater vehicles. The following chapters will present the extension of some of these approaches, and their successful use in some of the intended applications introduced in Chapter 1.

APPENDIX