Robotics and Autonomous Systems 64 (2015) 44–58

Contents lists available at ScienceDirect

Robotics and Autonomous Systems

journal homepage: www.elsevier.com/locate/robot

A comparison of homotopic path planning algorithms for robotic applications

Emili Hernandez *, Marc Carreras, Pere Ridao

Department of Computer Engineering, University of Girona, Girona 17071, Spain

HIGHLIGHTS

- We present three path planners to generate solutions that follow homotopy classes.
- Homotopy classes provide an added value to the path planning problem.
- Our method generates paths with the topology of the optimal solution much faster.
- We show extensive results in synthetic scenarios and on a bathymetric map.

ARTICLE INFO

Article history:
Received 23 March 2014
Received in revised form
26 August 2014
Accepted 28 October 2014
Available online 7 November 2014

Keywords:
Path planning
Homotopy
Marine robotics

ABSTRACT

This paper addresses the path planning problem for robotic applications using homotopy classes. These classes provide a topological description of how paths avoid obstacles, which is an added value to the path planning problem. Homotopy classes are generated and sorted according to a lower bound heuristic estimator using a method we developed. Then, the classes are used to constrain and guide path planning algorithms. Three different path planners are presented and compared: a graph-search algorithm called Homotopic A* (HA*), a probabilistic sample-based algorithm called Homotopic RRT (HRRT), and a bug-based algorithm called Homotopic Bug (HBug). Our method has been tested in simulation and in an underwater bathymetric map to compute the trajectory of an Autonomous Underwater Vehicle (AUV). A comparison with well-known path planning algorithms has also been included. Results show that our homotopic path planners improve the quality of the solutions of their respective non-homotopic versions with similar computation time while keeping the topological constraints.

1. Introduction

Path planning is one of the most studied problems in robotics. The goal of the algorithms is to find a safe path to guide a robot in the Configuration Space (C-Space) [1]. Graph-based search algorithms look for the global optimal path in the C-Space which is obtained with an exhaustive exploration of the search space [2,3]. Most of these algorithms use a heuristic function to speed up the exploration process by first selecting the most promising states according to the heuristics [4]. On the other hand, probabilistic sample-based path planners, most of them based on the Rapidly-exploring Random Tree (RRT) [5], perform the exploration by growing a tree incrementally until the goal is reached. These algorithms do not perform an exhaustive exploration in C-Space, therefore, they provide a solution very quickly at the expense of its quality. Even bug-based algorithms [6], initially developed to perform reactive motion planning, can also be considered for path planning purposes. Their simple navigation strategies alternate "go straight" and "follow boundary" behaviors when an obstacle is detected are suitable to generate paths in a C-Space.

Although the aforementioned strategies generate paths towards a goal state taking into account the non-traversable states due to obstacles, none of them report the topological information of *how* they are avoided. Because of this, we use homotopy classes to provide a topological description of the paths.

A *homotopy class* is the set of all possible trajectories from a start to a goal point that avoids obstacles in the same manner [7,8]. Given two paths, they are said to be homotopic if one can be deformed into the other without encroaching any obstacle. Providing this extra layer of topological information is an added valued to the path planning problem, and knowing the homotopy class before the computation of the path itself has several advantages for robotic applications. First, it is possible to constrain the

* Correspondence to: Autonomous Systems, Digital Productivity and Services (DPAS) Flagship, CSIRO, Pullenvale, QLD 4069, Australia. Tel.: +61 7 33274062.
E-mail addresses: Emili.Hernandez@csiro.au (E. Hernandez), marcc@eia.udg.edu (M. Carreras), pere@eia.udg.edu (P. Ridao).

http://dx.doi.org/10.1016/j.robot.2014.10.021
0921-8890/

path search into those areas of the search space that satisfy the homotopy class speeding up the path computation, which is very useful for low-powered navigation computers commonly used in autonomous robots. Second, if the homotopy class of the optimal path is known, it is possible to generate a good solution very quickly. Third, homotopy classes allow avoidance of obstacles in a specific manner which is of interest in surveillance, coverage and map exploration applications. For instance, homotopy classes can be selected to reinforce surveillance in specific areas, to avoid navigation patterns and to ensure that different areas of interest are explored from a topological point of view. They can also be used to constrain specific parts of the environment to avoid dangerous or unwanted areas in coverage applications. Multi-robot map exploration can also be improved using homotopy classes if each robot explores the environment following a different homotopy class.

The work presented in this paper is focused on, but not constrained to, path planning for Autonomous Underwater Vehicle (AUV) applications. Although the problem is naturally formulated in 3d, for certain scenarios of interest the problem can be simplified to 2d. For instance, a survey and/or search mission where the robot is supposed to fly at a fixed altitude, in bottom-following mode, while acquiring opto-acoustic imagery. Under these conditions, we can consider a 2d map parallel to the seafloor, where any area with a slope greater than a certain threshold behaves as a 2d obstacle. This is the case for applications like benthic habitat mapping, underwater archeology or cable/pipe inspection, being also the target for the system proposed.

In this paper we use homotopy classes to guide path planning algorithms topologically. We generate the homotopy classes that can be followed in any 2d workspace using the method presented in [9] and improved in [10]. Using the topological information, path planners do not have to explore the whole space but the space confined in a homotopy class. Then, using a lower bound heuristic estimator, the homotopy classes that most probably contain the lower cost solutions are known. Thus, the algorithm can generate some good solutions quickly. Three well known algorithms have been adapted to compute paths for each homotopy class generated, expanding on our preliminary work: the Homotopic A* (HA*) [11], the Homotopic RRT (HRRT) [10] and the Homotopic Bug (HBug) [12,13]. The completeness of our method is ensured because in case the goal is not reachable, no homotopy classes will exist and, consequently, no paths will be generated. The homotopy class of the global optimal path is guaranteed to be generated by the algorithm. Constraining the path search to the topology of the optimal solution allows generating good solutions with non-exhaustive search-based methods at a fraction of their computation time. This is an advantage in large environments because these algorithms can be very slow.

The paper also presents an extension of our previous results. The efficiency and scalability of our method has been tested in synthetic scenarios. Furthermore, the path planners have been compared with their non-homotopic versions and with themselves. Finally, the paper extends the results of an experiment under the scope of the TRIDENT European project (EU FP7 ICT-248497), in which we initially explored the generation of homotopic trajectories on a bathymetric map using the HA* [14]. The bathymetric map was generated by means of a Multibeam Profiling Sonar (MPS) and a Differential Global Positioning System (DGPS), which was used to generate a 2d Occupancy Grid Map (OGM) at the depth in which the AUV navigates. Our method has been applied to compute paths that the AUV must follow to reach a goal position avoiding obstacles in different manners. Extended analysis of the HA* execution and new results using the HRRT and the HBug on the bathymetric map are provided, leading to a discussion on the suitability of each path planner for robotic applications.

The paper is structured as follows: Section 2 details the relevant literature about homotopy classes for robotic applications; Section 3 describes the method to generate the homotopy classes from the workspace; Section 4 details the path planners that we propose which follow the homotopy classes previously generated; Section 5 reports the results; Section 6 presents the conclusions and future work.

2. Related work with homotopy classes

From the path planning point of view, the literature about homotopy classes can be classified in three different groups: the computation of the shortest homotopic path solutions that require a homotopy class as an input; those solutions that constrain the search topologically based on previous paths generated in the C-Space; and the automated generation of homotopy classes approaches that do not require generating any prior path in the C-Space.

2.1. The shortest homotopic path problem

Computing the shortest homotopic path using a non optimal path or homotopy class as an input has been studied since the 1980s. The *funnel algorithm* [15] is a well-known reference for triangulated environments. Although it was not explicitly designed to be used with homotopy classes, it generates the shortest path in a simple polygonal environment provided as an input which implicitly constrains it topologically. Other approaches generate the shortest homotopic solution for paths defined by closed curves [16] and adding constraints such as weighted regions [17].

In non-triangulated environments some authors proposed algorithms that efficiently compute the shortest homotopic path for a set of input paths that do not self-intersect [18]. This restriction was overcome in [19] and additional constraints such as path thickness were added in [20]. In the same vein, Grigoriev and Slissenko proposed a method to construct the shortest path for a given homotopy class that does not intersect in a scenario with semi-algebraic obstacles [21].

Despite the set of proposals that perform the computation efficiently for any possible path, the problem becomes intractable when there is no input homotopy class or path. Because of this, these solutions are difficult to apply in robot path planning. On the other side, some of these algorithms are suitable for an optimization process, when a path has been already generated by a path planner.

2.2. Constraining path search topologically

There is also a group of methods that compute the shortest path and then identify its homotopy class. This process can be done during the path search or once the whole path has been computed. Then, the topology of the path is encoded in order to restrict the next path search, which ensures that the new shortest path will have a different topology and hence, belong to another homotopy class. By repeating this process, it is possible to obtain the k-shortest paths of k-homotopy classes. In [22,23], two step methods were proposed: first, for each node of the environment they compute the shortest path that passes through it and then, the number of paths are pruned to generate the shortest path for each homotopy class. Other approaches generate a graph based on a Voronoi diagram computed from the areas to avoid which is then traversed using Depth-First Search (DFS)-based algorithms [24]. Bhattacharya et al. proposed a method to perform path planning with homotopy class constraints using graph-search algorithms. The graph that represents the environment is expanded with Complex Analysis values to characterize homotopy classes while computing the path [25]. This method has recently been extended to work on 3d environments and formulated for nd C-Spaces [26].

In [27], it has also been applied to plan the path and control a flexible cable towed by two robots to separate two types of objects in a planar environment.

Although these methods have the clear advantage of starting with the computation of the global shortest path for robotic applications, if we are interested in the solution of the k path it requires computing all the previous paths.

2.3. Homotopy classes generation approaches

There is a small group of methods that first compute the homotopy classes and then search for a path that follows them. In order to generate the homotopy classes, a data structure that encodes the topology of the environment as a graph is required. Then, the homotopy classes can be systematically generated by exploring the graph with a graph-search algorithm.

Jenkins proposed a method to generate homotopy classes for any 2d workspace with obstacles by turning the workspace into a topological graph, which allows the systematic generation of homotopy classes using graph-search algorithms [28]. Then, the shortest paths for each homotopy class were generated assuming circular obstacles in the workspace [29]. However, some constraints imposed by obstacles in the workspace are not taken into account into the topological graph, which allows generating homotopy classes that cannot be followed back in the workspace.

In [30], homotopy classes characterized fixed routes topologically in a factory environment that allowed unmanned vehicles to avoid collisions among them while keeping the topology and [31] proposed to build a Probabilistic Road Map (PRM) [32] to take into account topological constraints for motion planning.

These methods offer the flexibility of computing a path that does not belong to the homotopy class of the global optimal path without having to compute any previous paths. However, generating homotopy classes systematically can make the problem intractable depending on the number of classes generated. This issue can be overcome by using some restriction criteria during the classes generation such as allowing only homotopy classes in their canonical form and avoiding those that self-intersect or repeat cycles.

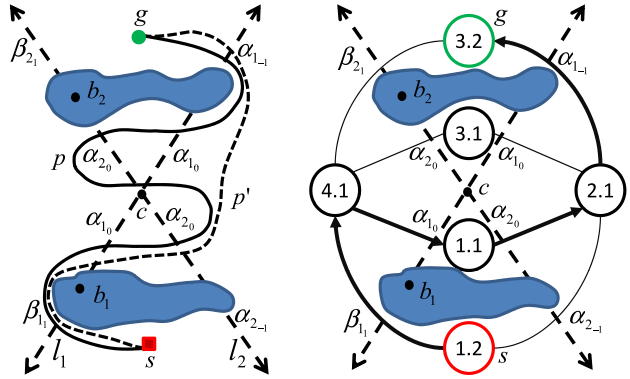
3. Homotopy classes generation

Given a workspace with obstacles, in [9,10] we extended the method proposed by Jenkins [28] to generate the homotopy classes that can be followed in any 2d workspace. The method first builds a reference frame which determines the topological relationships between obstacles in the workspace and is used to name the homotopy classes. The reference frame is then used to build the topological graph which allows the computation of homotopy classes systematically.

3.1. Reference frame

Given a workspace with n obstacles, the reference frame determines, in the metric space, the topological relationships between obstacles and is used to name the homotopy classes. The whole construction process is summarized in three steps:

1. Select a random point inside each obstacle and label it b_k, where $k = 1..n$.
2. Select the central point c of the reference frame. This point cannot be inside an obstacle or on the $n(n-1)/2$ lines determined by the pairwise choices of distinct b_k.
3. Construct n lines l_k joining c with each b_k. Each line is partitioned into $m+1$ segments, where m is the number of obstacles that intersect with l_k in the workspace. The segments from b_k and away from c are labeled with β_{k_s}, and the segments in the opposite direction are labeled α_{k_s}, where $s = 0..u$ with $u \in \mathbb{Z}^+$ for the segments of l_k from c that pass through b_k and $s = 0..v$ with $v \in \mathbb{Z}^-$ for the segments in the opposite direction.



(a) Reference frame. (b) Topological graph.

Fig. 1. Topological path represented in: (a) the reference frame as $p = \beta_{1_1}\alpha_{1_0}\alpha_{2_0}\alpha_{1_0}\alpha_{2_0}\alpha_{2_0}\alpha_{1_0}\alpha_{1_{-1}}$ (solid line) and one possible representation of its canonical sequence $p' = \beta_{1_1}\alpha_{1_0}\alpha_{2_0}\alpha_{1_{-1}}$ (dashed line); (b) representation of its canonical sequence in the topological graph.

Using the reference frame, any path p can be defined topologically by the sequence of labels of the segments crossed in order from the start to the end point. Fig. 1(a) depicts a reference frame for a scenario with two obstacles. The path p is labeled $\beta_{1_1}\alpha_{1_0}\alpha_{2_0}\alpha_{1_0}\alpha_{2_0}\alpha_{2_0}\alpha_{1_0}\alpha_{1_{-1}}$. There are two special cases when defining paths in the reference frame: when p does not cross any rays then $p = \emptyset$; and when p crosses through c meaning that all the α_{k_0}'s are simultaneously crossed, all α_{k_0} are added in subindex order to the sequence.

3.2. Computation of the canonical sequence

It is possible to know whether the paths that do not follow the same crossing-ray order in the reference frame are homotopic through their canonical sequence [33]. The canonical sequence is the simplest representation of a path without changing its topology, and only one canonical sequence exists for each homotopy class, but each canonical sequence can be represented by infinite trajectories in the workspace. With the notation used in the reference frame, it is computed by first sorting the α_{k_0}'s substrings according to the subindex of the path in non-decreasing order. Then, all the elements of the sequence that have the same character by pairs are removed. This process is repeated until no changes are made in the sequence. For instance, once path $\beta_{1_1}\ \alpha_{1_0}\ \alpha_{2_0}\ \alpha_{1_0}\ \alpha_{2_0}\ \alpha_{2_0}\ \alpha_{1_0}\ \alpha_{1_{-1}}$ gets its α_{k_0}'s substring sorted, it becomes $\beta_{1_1}\ \alpha_{1_0}\ \alpha_{1_0}\ \alpha_{1_0}\ \alpha_{2_0}\ \alpha_{2_0}\ \alpha_{2_0}\ \alpha_{1_{-1}}$. At this point the α_{1_0} and α_{2_0} pairs can be canceled $\beta_{1_1}\ \cancel{\alpha_{1_0}\alpha_{1_0}}\ \alpha_{1_0}\ \cancel{\alpha_{2_0}\alpha_{2_0}}\ \alpha_{2_0}\ \alpha_{1_{-1}}$ obtaining $\beta_{1_1}\ \alpha_{1_0}\ \alpha_{2_0}\ \alpha_{1_{-1}}$. Since it cannot be shortened, it represents the canonical sequence of the path. Fig. 1(a) depicts one possible solution in the workspace of this canonical sequence.

3.3. Topological graph

The reference frame is used to compute a topological graph G, providing a model to describe the relationships between regions of the metric space. Its construction can be divided into three steps:

1. The lines in the reference frame divide the metric space into regions or *wedges* and the obstacles that intersect with more than one line at the same time split these wedges into *sub-wedges*. Each sub-wedge represents a node of G.

2. Each node of G is labeled according to the wedge w and sub-wedge sw using the notation $w.sw$; $w \in \mathbb{N}$ is numbered counterclockwise. For each w, its corresponding $sw \in \mathbb{N}$ is numbered sequentially starting by 1 for the one closest to c.
3. Two nodes of G are interconnected according to the number of segments they share in the reference frame. Each edge of G is labeled with the same label of the segment that crosses it in the reference frame.

In the reference frame, a path is defined according to the segments it crosses, whereas in G it turns into traversing the graph from the start node to the end node. Note that the start and end nodes of G are those sub-wedges in the reference frame where the start and end points are located. Fig. 1(b) depicts the canonical sequence $\beta_{1_1}\alpha_{1_0}\alpha_{2_0}\alpha_{1_{-1}}$ in the topological graph.

3.4. Systematic homotopy classes computation

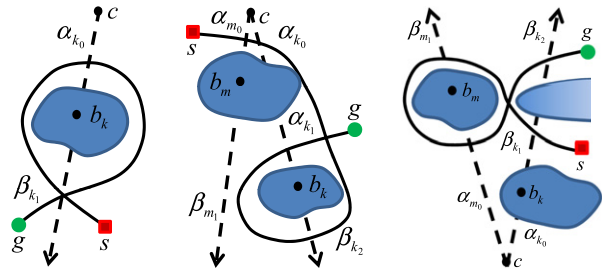
Once the topological graph is constructed, it is traversed using a modified version of the Breadth-First Search (BFS) algorithm. The BFS is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then, for each of those nearest nodes, it explores their unexplored neighbors. The process is repeated until the goal is found. Unlike the standard BFS, which stops when all vertexes have been visited, the modified algorithm continues until there are no more homotopy class candidates to explore or the length of the last homotopy class candidate is larger than a given threshold.

3.4.1. Restriction criteria

During the BFS execution, several restriction criteria are applied to avoid the generation of any homotopy classes which either self-intersect or its canonical sequence is duplicated and has already been considered. All classes that satisfy any of the following restriction criteria are ignored to avoid using them as a root for future homotopy classes:

- *Simple wrap*. Any string that contains a substring of the form $\alpha_{k_s} \dots \chi_{k_t} \dots \alpha_{k_u}$ or $\beta_{k_s} \dots \chi_{k_t} \dots \beta_{k_u}$ where $\chi = (\alpha, \beta)$ with $s = u$ represents a class that wraps around an obstacle and is self-crossing. Fig. 2(a) shows an example of a path that accomplishes the simple wrap criterion.
- *Wrap*. Any string that contains a substring of the form $\chi_{k_s} \dots \chi_{k_t} \dots \chi_{k_u}$ where $\chi = (\alpha, \beta)$ with $s, t, u \geq 0$ and $s > t < u$ or with $s, t, u \leq 0$ and $s < t > u$ represents a class that wraps around an obstacle and is self-crossing. Fig. 2(b) shows an example of a path that accomplishes the wrap criterion.
- *Self-crossing*. Any string that contains a substring of the form $\chi_{k_s} \dots \beta_{m_t} \dots \alpha_{m_u} \dots \chi_{k_v}$ where $\chi = (\alpha, \beta)$ with $s, v \geq 0$ and $s < v$ or with $s, v \leq 0$ and $s > v$ represents a class that self-crosses. The reversed substring $\chi_{k_s} \dots \alpha_{m_t} \dots \beta_{m_u} \dots \chi_{k_v}$ with $s, v \geq 0$ and $s > v$ or with $s, v \leq 0$ and $s < v$ also represents a class that self-crosses. Fig. 2(c) shows an example of a path that accomplishes the self-crossing criterion.
- *Duplicated*. Duplicated strings are not allowed in the list of homotopy class candidates. If a string is not in its canonical form, it can be simplified without modifying its topology. Then, it is ensured that the resultant string has already been computed by the BFS algorithm because it would be shorter than the input string. Finally, the algorithm cannot traverse the same edge on two consecutive occasions. By doing that, a string with a repeated pair would be generated. Consequently, the pair would be simplified and the string discarded for being duplicated.

Table 1 shows the homotopy classes computed for the workspace depicted in Fig. 1 applying all the restriction criteria with their index of generation according to the BFS algorithm.



(a) A simple wrap in path $\beta_{k_1}\alpha_{k_0}\,\beta_{k_1}$.
(b) A wrap in path $\alpha_{m_0}\alpha_{k_0}\beta_{k_2}\,\alpha_{k_1}$.
(c) Self-crossing in path $\beta_{k_1}\beta_{m_1}\alpha_{m_0}\,\beta_{k_2}$.

Fig. 2. Restriction criteria examples during the homotopy class generation process. Any homotopy class that wraps an obstacle or self-crosses is discarded.

Table 1
Homotopy classes with their generation index and their lower bound normalized according to the cost of the A* solution.

Idx	Homotopy class	LB
1	$\beta_{1_1}\beta_{2_1}$	0.94
2	$\alpha_{2_{-1}}\alpha_{1_{-1}}$	0.95
3	$\beta_{1_1}\alpha_{1_0}\alpha_{2_0}\alpha_{1_{-1}}$	1.04
4	$\alpha_{2_{-1}}\alpha_{1_0}\alpha_{2_0}\beta_{2_1}$	1.08

3.5. Lower bound estimator

The number of homotopy classes generated by the BFS algorithm highly depends on the number of the nodes in the topological graph. Therefore, in most scenarios it is not possible to compute all the correspondent paths of the homotopy classes in the workspace in real-time. In order to set up a preference order when choosing the homotopy classes to compute their paths, a modified version of the *funnel algorithm* [15] is used to obtain a quantitative measure for each homotopy class estimating its quality. This algorithm computes the shortest path within a *channel*, which is a polygon formed by the vertexes of the segments in the reference frame that are traversed in the topological graph. The modification consists of accumulating the Euclidean distance between the points while they are being added to the channel's shortest path. Hence, the result of the funnel algorithm is a lower bound of the optimal path in the workspace of the selected homotopy class.

The lower bound estimator is used to set up a preference order to compute the homotopy classes path in the workspace when operating under time restrictions. Note that the segments of the reference frame constrain the region where the paths can go through, but do not take into account the whole shape of the obstacles. For that reason, a homotopy class with a smaller lower bound may have a longer path in the workspace than another homotopy class with a higher lower bound.

Fig. 3 depicts an example where the funnel algorithm computes the lower bound for the homotopy class $\beta_{1_1}\,\alpha_{1_0}\,\alpha_{2_0}\,\alpha_{1_{-1}}$. The solid lines represent the channel and the dashed red line is the path after applying the funnel algorithm. It is worth noting that the modified algorithm takes into account that some subsegments may self-intersect when creating the channel as can be seen with the α_{1_0} and α_{2_0} segments in the figure. Table 1 shows the lower bound for each homotopy class obtained in this example. All lower bounds have been normalized according to the cost of the path obtained with the A*, which belongs to class 2 ($\alpha_{2_{-1}}\alpha_{1_{-1}}$), using an A8 connectivity in a discretized representation of the environment. Some values are lower than 1 since the A* solution takes into account the shape of the obstacles while the computation of the lower bound does not.

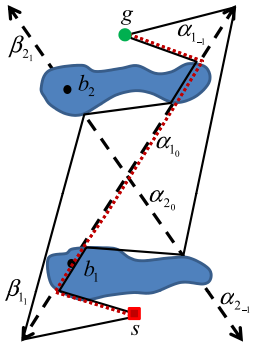


Fig. 3. Lower bound in dashed red and channel in solid black for the homotopy class $\beta_{1_1}\alpha_{1_0}\alpha_{2_0}\alpha_{1_{-1}}$, index 3 in Table 1.

4. Homotopic path planning algorithms

Once the homotopy classes are computed and sorted according to their lower bound, a path planning algorithm has to find a path in the workspace that follows a given homotopy class which essentially implies turning a topological path into a metric path. The only link between the workspace and the topological space is the reference frame. It allows checking whether a metric path in the workspace is following a topological path by following the intersections in order from the initial configuration to the current configuration. For this purpose, three well known algorithms have been adapted to compute paths for a single homotopy class.

4.1. Homotopic A*

The Homotopic A* (HA*) is a graph-based search algorithm based on the A* [4] that instead of exploring the entire search space, it only explores the zones in the workspace that satisfy a given homotopy class by checking the intersections with the reference frame before taking into consideration the node as a candidate to be explored.

The HA* is written in pseudocode in Algorithm 1. The nodes in the algorithm are tuples that contain the configuration of robot q and the topological path from q_{start} to q. These values are accessible through the functions Q and P respectively. Just like the A*, open nodes are processed according to their position in a priority queue *OPEN*. Each node in this queue is ordered according to the sum of its current path cost from the start, $g(n)$, and a heuristic estimation of its path cost to the goal, $h(n, n_{goal})$. The node with the minimum sum is at the top of the priority queue.

The algorithm receives as input the start configuration q_{start}, the goal configuration q_{goal}, a candidate homotopy class to follow H and the reference frame F. The configurations q_{start} and q_{goal} are used to set up the initial node n_{start} and the goal node n_{goal} (line 21).

The function *ComputePath* computes the shortest path that follows H. It starts by adding the n_{start} into the *OPEN* queue. As node n with the minimum $g(n) + h(n, n_{goal})$ is different from n_{goal}, the algorithm pops n to the top of the queue. For all the configurations q' reachable from $Q(n)$, the function *FindIntersections* (line 13) returns the intersections of the segment $[Q(n), q']$ with F sorted by distance. Note that it is possible to intersect more than one segment of the reference frame depending on how close $Q(n)$ and q' are to the c point. Then the *UpdateH* (line 14) generates the new topological path according to the intersections. No intersections with F means that the explored configuration is in the same subwedge of the C-Space and the function returns $P(n)$. If there are intersections and these intersections follow H, the function returns $P(n) \cup I$ in order to create a new node candidate n'. Whether n' is in the *OPEN* queue or not is then checked. If not, it is added to the queue with a priority $g(n')$ plus the heuristic $h(n', q_{goal})$ (line 18). If it is, and the cost $g(n)$ plus the cost of traversing from n to n', $c(n, n')$ is less than its current cost (line 19), $g(n')$ is set to this new lower value. This process is repeated until the n_{goal} is found or *OPEN* has no more nodes to be expanded.

Algorithm 1 Homotopic A*

FindIntersections$([q, q'], F)$
1: $r \leftarrow \emptyset$
2: **for** $i \leftarrow 1$ ***to*** $|F|$ **do**
3: **if** $x \leftarrow Intersection([q, q'], F[i]) \neq null$ **then**
4: $r \leftarrow r \cup \{Edge(i), Distance(q, x)\}$
5: $r \leftarrow SortByDistance(r)$
6: **return** r

ComputePath(n_{start}, n_{goal}, F)
7: $OPEN \leftarrow \emptyset$; $V \leftarrow \emptyset$
8: $OPEN.push(n_{start})$
9: **while** $min_{n \in OPEN}(n \neq n_{goal})$ **do**
10: $n \leftarrow OPEN.top()$
11: $OPEN.pop()$
12: **for all** $q' \in Succ(Q(n))$ **do**
13: $I \leftarrow FindIntersections([Q(n), q'], F)$
14: **if** $H' \leftarrow UpdateH(P(n), I)$ **then**
15: $n' \leftarrow \{q', H'\}$
16: **if** $n' \notin OPEN$ **then**
17: $g(n') \leftarrow g(n) + c(n, n')$
18: $OPEN.push(n')$ with $g(n') + h(n', n_{goal})$
19: **else if** $g(n') > g(n) + c(n, n')$ **then**
20: $g(n') \leftarrow g(n) + c(n, n')$

HA*$(q_{start}, q_{goal}, H, F)$
21: $n_{start} \leftarrow \{q_{start}, \emptyset\}$; $n_{goal} \leftarrow \{q_{goal}, H\}$
22: $ComputePath(n_{start}, n_{goal}, F)$
23: **if** $min_{n \in OPEN}(n = n_{goal})$ **then**
24: publish solution

Fig. 4(a) depicts an example of an HA* execution in a discrete version of the workspace shown during the generation of the homotopy classes for the homotopy class $\beta_{1_1}\alpha_{1_0}\alpha_{2_0}\alpha_{1_{-1}}$ using an A8 connectivity and the Euclidean distance as heuristic estimator. The explored states are depicted in gray.

4.1.1. Complexity analysis

The HA*, shown in Algorithm 1, is based on the A*, which has a time complexity $O(N^d)$ on the dimension d in a grid-based search space [34]. Two additional methods have been implemented for searching a path according to a homotopy class: *FindIntersection* and *UpdateH* in lines 13 and 14 respectively.

FindIntersection time complexity depends on the time to check intersection T_{int}, the number of the reference frame segments $|F|$, which is fixed for a HA* execution, and the sorting process of the intersections found:

$$T_{fi} = |F|\, T_{int} + |F| \approx O(1). \tag{1}$$

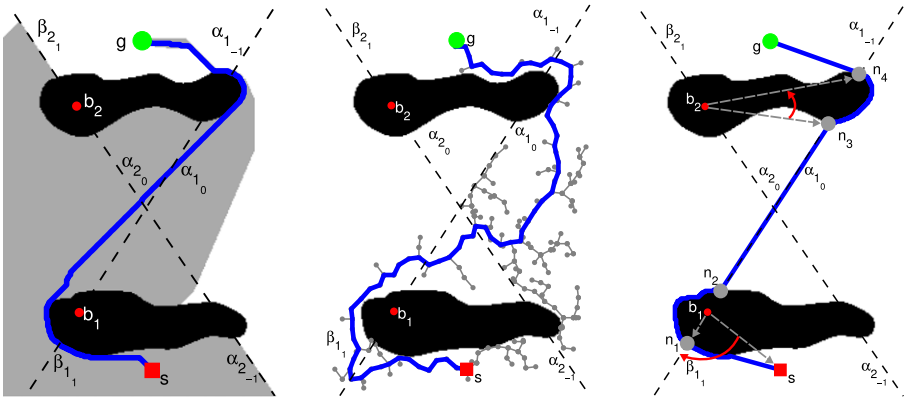
UpdateH time complexity depends on the number of intersections with the reference frame segments returned by *FindIntersection* and the simple operation time to update it T_u. In the worst case scenario the number of intersections could be the number of the segments in the reference frame $|F|$:

$$T_{uh} = |F|\, T_u \approx O(1). \tag{2}$$

Although *FindIntersection* and *UpdateH* are in the inner loop, their order does not affect the dominant complexity of the algorithm. Therefore, the overall time complexity is $O(N^d)$, with $d = 2$ in the context of the paper.

4.1.2. Theoretical properties

The HA* inherits two properties from the A* algorithm:



(a) HA* solution. (b) HRRT solution. (c) HBug solution.

Fig. 4. Execution of the proposed path planners for homotopy class $\beta_{1_1}\alpha_{1_0}\alpha_{2_0}\alpha_{1_{-1}}$: (a) HA* path with the explored states depicted in gray; (b) a solution found by the HRRT with the generated exploration tree in gray; (c) HBug solution with the boundary nodes n_x in gray, the lines used to compute the directions in dashed gray and the direction the path follows at each obstacle in red. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Lemma 1. *The HA* computes the optimal solution from s to g for given homotopy class H.*

Assumption 1. HA* uses an admissible heuristic function.

Proof. The HA* imposes an additional restriction with respect to the A* in the exploration process by expanding only those nodes that satisfy H, which is A* same principle of exploring nodes according to their priority. See proof in [35]. □

Lemma 2. *The HA* is complete.*

Proof. Refer to [35]. □

4.2. Homotopic rapidly-exploring random tree

The Homotopic RRT (HRRT) is based on the goal-biased RRT [5] algorithm which has been shown to be very efficient in time, even in complex workspaces [36,37]. The algorithm allows a constrained growing of the tree only in those directions that satisfy a given homotopy class. Before adding a new node into the tree, the topological path traversed is checked to ensure that it belongs to the homotopy class by computing the intersections of the path with the reference frame.

The HRRT is detailed in Algorithm 2. It receives as input the start configuration q_{start}, the goal configuration q_{goal}, a candidate homotopy class to follow H and the reference frame F. The nodes on tree T are tuples that contain the configuration of the robot q and the topological path from q_{start} to q. These values are accessible through the functions Q and P respectively. Just like the RRT, the function *Extend* (line 20) iteratively extends tree T until the distance between the configuration of n_{new} ($Q(n_{new})$) and n_{goal} ($Q(n_{goal})$) is lower than a *distThreshold*.

The extension of the tree starts by selecting a random configuration q_{rand} from the C-Space with the function *ComputeQRand*. Then, the *NearestNeighbor* function returns the nearest node $n_{nearest}$ regarding a random configuration q_{rand} by looking for the node whose topological path is closer to $P(n_{goal})$ (line 4). If there is more than one candidate, the node selected is the closest to the goal according to the Euclidean distance. After q_{new} is computed using the function *ComputeQNew*, *FindIntersections* (line 11) checks whether the segment $[Q(n_{nearest}), q_{new}]$ intersects with any segment in reference frame F. The function returns the intersected edges sorted by distance from $Q(n_{nearest})$. Then, the function *UpdateH* (line 12) generates the new topological path H' according to the intersections. No intersections with F means that the tree grows in the $n_{nearest}$ sub-wedge and hence, the function returns $P(n_{nearest})$. If there are intersections and these follow the topological path, the function returns $P(n_{nearest}) \cup I$ in order to create a new candidate node n_{new} to be added to the tree; otherwise a *null* path is returned and no node is added to the tree.

Algorithm 2 Homotopic RRT

NearestNeighbor(T, q_{rand})
1: $n \leftarrow T$; $d \leftarrow Distance(Q(n), q_{rand})$
2: **for all** $c \leftarrow T.Children()$ **do**
3: $[n', d'] \leftarrow NearestNeighbor(T, q_{rand})$
4: **if** $(|P(n')| > |P(n)|)$ ***or*** $(|P(n')| = |P(n)|$ ***and*** $d' < d)$ **then**
5: $n \leftarrow n'$; $d \leftarrow d'$
6: **return** $\{n, d\}$

Extend(T, n_{goal}, F)
7: $n_{new} \leftarrow \{\infty, null\}$
8: $q_{rand} \leftarrow ComputeQRand()$
9: $n_{nearest} \leftarrow NearestNeighbor(T, q_{rand})$
10: $q_{new} \leftarrow ComputeQNew(Q(n_{nearest}), q_{rand})$
11: $I \leftarrow FindIntersections([Q(n_{nearest}), q_{new}], F)$
12: $H' \leftarrow UpdateH(P(n_{nearest}), I)$
13: **if** $(H' \neq null)$ **then**
14: $n_{new} \leftarrow \{q_{new}, H'\}$
15: $n_{nearest}.Add(n_{new})$
16: **return** n_{new}

HRRT$(q_{start}, q_{goal}, H, F)$
17: $n_{new} \leftarrow \{q_{start}, \emptyset\}$; $n_{goal} \leftarrow \{q_{goal}, H\}$
18: $T.Add(n_{new})$
19: **while** $Distance(Q(n_{new}), Q(n_{goal})) > distThreshold$ **do**
20: $n_{new} \leftarrow Extend(T, n_{goal}, F)$

Fig. 4(b) depicts an execution example of the HRRT with an exploration tree and the solution obtained with no post-processing. The homotopy class to follow is $\beta_{1_1}\alpha_{1_0}\alpha_{2_0}\alpha_{1_{-1}}$. Note that the algorithm constrains the expansion of the tree only into those zones that accomplish the input homotopy class, thus, in this example, segments $\alpha_{1_{-1}}$ or β_{2_1} in the reference frame do not have to be crossed by the tree.

4.2.1. Complexity analysis

The time complexity analysis of the HRRT, shown in Algorithm 2, depends on the number of times the tree is expanded with *Extend* function. Assuming that N *Extend* calls are executed and leaving the simple assignation operations aside, the time to generate a tree that follows a homotopy class can be calculated as the sum of time

for N iterations of operations in lines 8–12 and 15:

$$T(N) = T_{q_{rand}}(N) + T_{nn}(N) + T_{q_{new}}(N) + T_{fi}(N) + T_{uh}(N) + T_{add}(N) \quad (3)$$

where $T_{q_{rand}}, T_{nn}, T_{q_{new}}, T_{fi}, T_{uh}, T_{add}$ are the time to compute *ComputeQRand*, *NearestNeighbor*, *ComputeQNew*, *FindIntersetions*, *UpdateH* and *Add* a node to the tree respectively.

$T_{q_{rand}}, T_{q_{new}}$ and T_{add} are simple operations that can be done in constant time at each iteration so they are $O(N)$ in the algorithm. Finding the *NearestNeighbour* at each iteration requires checking the distance with all previous configurations added to the tree:

$$T_{nn}(N) = \sum_{i=1}^{N}(i-1)T_{dist} = \frac{N^2 - N}{2}T_{dist} \approx O(N^2 - N) \quad (4)$$

where T_{dist} is the time that takes to calculate the distance. As stated before, *FindIntersection* time complexity depends on the time to check intersection T_{int}, the number of the reference frame segments $|F|$, which is fixed for a HRRT execution, and the sorting process of the intersections found:

$$T_{fi}(N) = N(|F|\,T_{int} + |F|) \approx O(N). \quad (5)$$

UpdateH time complexity depends on the number of intersections with reference frame segments returned by *FindIntersection* and the simple operation time to update it T_u. In the worst case scenario the number of intersections could be the number of the segments in the reference frame $|F|$:

$$T_{uh}(N) = N\,|F|\,T_u \approx O(N). \quad (6)$$

The complexity of the algorithm is the combination of the main operations during the *Extend* process:

$$O(N) + O(N^2 - N) + O(N) + O(N) + O(N) + O(N) \approx O(N^2 - N) \approx O(N^2). \quad (7)$$

4.2.2. Convergence

The HRRT is derived from the RRT algorithm adding a constrained growing of the tree according to a homotopy class.

Lemma 3. *The HRRT is probabilistically complete for a given homotopy class H.*

Assumption 2. H may or may not allow connecting s and g.

Proof. Refer to [38]. □

4.3. Homotopic Bug

The Homotopic Bug (HBug) is the third approach to generate paths according to a homotopy class. Although Bug-based planners are reactive algorithms designed for on-line robot navigation that can be applied to robots with low computational capabilities, it is possible to use them to perform deliberative path planning on a C-Space [39]. The HBug is based on the Bug2 [40]. It is an opportunistic algorithm that takes advantage of the already computed lower bound path, which is similar to the Bug2's m-line but satisfying a homotopy class constraints. Essentially, the HBug tries to follow the lower bound path obtained with the modified funnel algorithm which ensures that the homotopy class is being satisfied. However, as mentioned before, the segments in the reference frame constrain the regions the paths can go through, but do not take into account the shape of the obstacles. For this reason, the lower bound path may intersect with the obstacles. In such cases, the obstacle boundary is followed in a clockwise or counterclockwise direction according to the homotopy class until the lower bound path leaves the obstacle. This process is repeated for all the obstacles intersected by the lower bound path.

The HBug is detailed in Algorithm 3. It receives as input parameters the lower bound path P, a candidate homotopy class to follow H and the reference frame F. Note that the first and last elements of P are the start (s) and goal (g) nodes respectively.

The algorithm is a three step process. First, the function *BoundaryNodes* checks the intersections of P with the obstacles in the C-Space. Every time that P hits or leaves an obstacle, a boundary node is created. Each node contains the contact point c and the obstacle label k, which is the subindex of the point b_k that represents the obstacle in the reference frame. These parameters are accessible through the functions Q and *Obst* respectively. Then, *ObstacleNodes* computes the nodes O based on the boundary nodes N previously computed. Each obstacle node contains the first boundary node that hits obstacle n_h, the last node in its boundary without changing the obstacle n_l, and the direction d to surround the obstacle while following H (line 18). Finally, the function *BuildPath* creates the path P' in the workspace by joining the boundary of each obstacle $o_i \in O$ from n_h to n_l with the direction d.

The direction d to surround an obstacle is set according to the direction of a hit node n_h towards its successor n_{h+1}[1] with respect to point b_k that represents the obstacle in the workspace in the reference frame. Note that n_h and n_{h+1} are ensured to belong to the same obstacle since for any point that hits an obstacle there has to be another that releases it. The perpendicular dot product between vectors $(Q(n_h) - b_k)$ and $(Q(n_{h+1}) - b_k)$ computes the boundary following the direction (line 12). If the result is less than 0, the direction from n_h to n_{h+1} is counterclockwise; if it is greater than 0, the direction is clockwise.

The result of the perpendicular dot product can be 0 if the vectors $(Q(n_h) - b_k)$ and $(Q(n_{h+1}) - b_k)$ are parallel, which means that n_h, n_{h+1} and b_k belong to the same l_k in the reference frame (line 13). In such cases, d is obtained according to two conditions: the initial direction selected to cross l_k from the start point, and the number of times that l_k is crossed until α_k or β_k, denoted by χ_k, of the homotopy class, on which n_{h+1} relies, is reached. The initial direction is obtained with the dot product from the start s to the first χ_k with the same subindex as l_k[2] (line 14). The number of times that l_k is crossed depends on the number of χ_k found in the homotopy class from the beginning to the index i_k, which indicates the position of the χ_k that contains n_{h+1} (line 16).

Fig. 4(c) depicts an example scenario where the HBug is applied. The homotopy class to follow is $\beta_{1_1}\alpha_{1_0}\alpha_{2_0}\alpha_{1_{-1}}$. The dashed line represents its lower bound path, which intersects the first obstacle generating two boundary nodes, n_1 and n_2, both located on line l_1 in the reference frame. The point that represents the obstacle is b_1, also on l_1, which makes the perpendicular dot product between $(Q(n_1) - b_1)$ and $(Q(n_2) - b_1)$ unable to set the direction ($d = 0$). Therefore, using the start point s and a point of the edge β_{1_1}, the initial direction is set clockwise. The last edge involved in this situation is α_{1_0}, located in the second position in the homotopy class. The number of edges with subindex 1 up to this position is 2, thus, the direction is not changed. Then, the lower bound path intersects the second obstacle in n_3 and n_4. Using the base point b_2, the perpendicular dot product sets the direction as counterclockwise. Finally, the path is composed from s to g with the boundaries of obstacle 1 (from n_1 to n_2) and obstacle 2 (from n_3 to n_4) joined by straight lines.

[1] When the lower bound path intersects with an obstacle only once, the n_{h+1} node is also the n_l node.

[2] Note that the start point cannot be in line l_k in the reference frame since the perpendicular dot product would also be 0.

Algorithm 3 Homotopic Bug

BoundaryNodes(P)
1: $N \leftarrow \emptyset$
2: **for** $i \leftarrow 1$***to***$|P|-1, p_i \in P$ **do**
3: $C \leftarrow$ ContourPoints(p_i, p_{i+1})
4: **for all** $j \leftarrow 1$***to***$|C|, c_j \in C$ **do**
5: $k \leftarrow$ Label(c_j)
6: $N \leftarrow N \cup \{c_j, k\}$
7: **return** N

ObstacleNodes(N, H, F)
8: $O \leftarrow \emptyset$; $h \leftarrow 1$
9: **while** $n_h \in N/h < |N| - 1$ **do**
10: $n_l \leftarrow$ last $n_j \in N/j > h$ without changing $Obst(n_h)$
11: $b_k \leftarrow$ point of $Obst(n_h)$ in F
12: $d \leftarrow (Q(n_h) - b_k)^{\perp} \cdot (Q(n_{h+1}) - b_k)$
13: **if** $d = 0$ **then** {parallel}
14: $d \leftarrow (s - b_k)^{\perp} \cdot$ (point of $1^{st}\chi_k \in H - b_k)$
15: $i_k \leftarrow$ index of $\chi_k \in H$ where n_{h+1} relies on
16: **if** $|\chi_k| \in H_{1..i_k}$ is *even* **then**
17: switch d
18: $O \leftarrow O \cup \{n_h, n_l, d\}$
19: $h \leftarrow l + 1$
20: **return** O

BuildPath(O)
21: $P' \leftarrow \emptyset$
22: **for** $i \leftarrow 1$***to***$|O|, o_i \in O$ **do**
23: $P' \leftarrow P' \cup$ Boundary(o_i)
24: **return** P'

HBug(P, H, F)
25: $N \leftarrow$ BoundaryNodes(P)
26: $O \leftarrow$ ObstacleNodes(N, H, F)
27: $P' \leftarrow$ BuildPath(O)

4.3.1. Complexity analysis

The time complexity analysis of the HBug, shown in Algorithm 3, is the combined complexity of the *BoundaryNodes*, *ObstacleNodes* and *BuildPath* operations.

BoundaryNodes computation time T_{bn} depends on the number of nodes N explored along the lower bound path P and on the contour points found during this process. The worst case scenario can be represented by a highly cluttered environment with a single cell obstacles and P intersecting with all of them. Each intersection generates 2 contour points and the maximum number of obstacles is $N/2$:

$$T_{bn}(N) = N + 2\frac{N}{2}T_b = N\,(1 + T_b) \approx O(N) \tag{8}$$

where T_b is the time of the simple operation to create a node.

ObstacleNodes is the highest complexity operation since it looks for the last contour node of a specific obstacle. In the worst case scenario with single cell obstacles its complexity at each iteration is $O(1)$ because the next node leaves the obstacle, but in other environments with fewer but more complex obstacles, an obstacle can be crossed k times making this operation $O(k)$. Therefore, the overall complexity of the method is $O(kN) \approx O(N)$.

BuildPath takes N obstacle nodes found in the previous step and connects them exploring the contour of the obstacle they belong to. The *Boundary* operation, in line 23, is $O(N)$ and the complexity of this method is $O(N^2)$.

Combining the complexity of the methods analyzed returns the complexity of the HBug:

$$O(N) + O(N) + O(N^2) \approx O(N^2). \tag{9}$$

4.3.2. Theoretical properties

The HBug algorithm has been designed to be used together with the automated computation of the homotopy classes. Nevertheless, the algorithm itself has several properties:

Lemma 4. *The HBug is complete.*

Proof. Let us prove it by contradiction assuming that the algorithm is incomplete and there is a path from s to g that follows an homotopy class H. This implies that the solution has a finite length and intersects obstacles a finite number of times. Two cases can arise: no termination or incorrect termination.

Let us assume that the algorithm does not terminate. However, based on [40], each *obstacle node* has a pair of hit and leave nodes (n_h and n_l) which ensure that n_l is closer to the goal than n_h when following H. This implies there exists a finite number of nodes which will be exhausted by the algorithm and terminate.

The algorithm could only finish incorrectly by avoiding obstacles in a manner that H is not satisfied, which requires contradicting the definition of the perpendicular dot product [41] when computing the turning direction. □

Lemma 5. *Given a lower bound path P, that follows homotopy class H, the upper bound UB in the C-Space is*

$$UB \leq LB + \sum_{i=1}^{n} p_i \tag{10}$$

where LB is the length of the P, n is the number of obstacles in the environment that crosses with P and p is the perimeter of the obstacles involved.

Assumption 3. Exists a solution and the position of g and the obstacles are known.

Proof. Based on the Bug2 upper bound proof [40], there is no path that can be shorter than LB and in the worst case scenario each obstacle intersected by LB could be completely circumnavigated. □

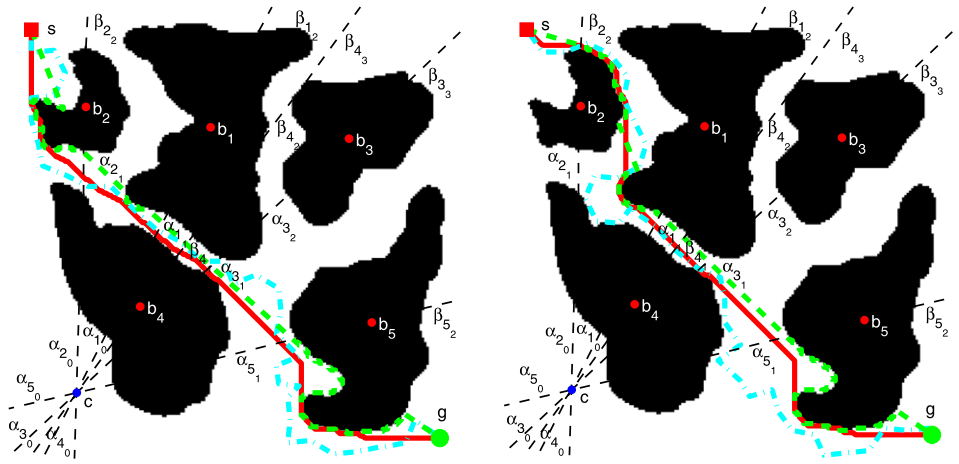
5. Results

The topological path search and the path planning algorithms we propose have been implemented and tested in different scenarios with irregular obstacles, the most commonly found in unstructured underwater environments. In order to identify the obstacles in the scenarios, a modified version of the Component Labeling (CL) algorithm [42] has been applied, which efficiently labels connected cells and their contours in grayscale images at the same time. For the construction of the reference frame, the c point has been set at a fixed position in order to ensure the same topological graph construction and homotopy classes generation through different executions. The homotopy classes have been set at a maximum of 20 character length. In order to show all the possible results, no time restrictions have been taken into consideration.

All the results have been computed using a laptop with a dual core processor at 1.83 GHz and 2 GB of RAM. The quality and performance of the solutions obtained with the HA*, HRRT and HBug have been measured using the path cost and computation time.

5.1. Cluttered scenario

This experiment shows the results obtained with the path planning method in a cluttered environment represented by a 200×200 pixels bitmap with five irregular obstacles (see Fig. 5). The construction of the reference frame, the topological graph and the generation of the homotopy classes with their lower bound computation took 7.9 ms. Table 2 lists the homotopy classes sorted by their lower bound with the path cost for the HA*, the HRRT and the HBug algorithms. To ensure the stability of the results, the path cost for the HRRT is the average of 100 executions using a *step* of



(a) $\alpha_{2_1}\alpha_{1_1}\beta_{4_1}\alpha_{3_1}\alpha_{5_1}$. (b) $\beta_{2_2}\alpha_{1_1}\beta_{4_1}\alpha_{3_1}\alpha_{5_1}$.

Fig. 5. Paths generated with HA* (solid-red), HRRT (dashed-cyan) and HBug (dashed-green) algorithms for the two homotopy classes with the smaller lower bound in the cluttered environment. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 2
Homotopy classes with the cost of the paths of Fig. 5 scenario sorted by their lower bound.

N°	Idx	Homotopy class	LB	HA*	HRRT	HBug
1	2	$\alpha_{2_1}\alpha_{1_1}\beta_{4_1}\alpha_{3_1}\alpha_{5_1}$	0.89	**1**	**1.32**	**1.35**
2	8	$\beta_{2_2}\alpha_{1_1}\beta_{4_1}\alpha_{3_1}\alpha_{5_1}$	0.90	**1.03**	**1.37**	**1.32**
3	3	$\alpha_{2_1}\alpha_{1_1}\beta_{4_1}\alpha_{3_1}\beta_{5_2}$	0.97	**1.28**	**1.64**	**1.62**
4	9	$\beta_{2_2}\alpha_{1_1}\beta_{4_1}\alpha_{3_1}\beta_{5_2}$	0.99	**1.30**	**1.69**	**1.58**
5	1	$\alpha_{5_0}\alpha_{3_0}\alpha_{4_0}\alpha_{1_0}\alpha_{2_0}$	1.05	1.11	**1.42**	**1.31**
6	11	$\beta_{2_2}\beta_{1_2}\beta_{4_2}\alpha_{3_2}\beta_{5_2}$	1.06	1.36	**1.74**	**1.63**
7	10	$\beta_{2_2}\beta_{1_2}\beta_{4_2}\alpha_{3_2}\alpha_{5_1}$	1.08	1.26	**1.61**	**1.55**
8	13	$\beta_{2_2}\beta_{1_2}\beta_{4_3}\beta_{3_3}\beta_{5_2}$	1.12	1.18	**1.45**	**1.36**
9	5	$\alpha_{2_1}\beta_{1_2}\beta_{4_2}\alpha_{3_2}\beta_{5_2}$	1.24	1.71	**2.31**	**2.23**
10	4	$\alpha_{2_1}\beta_{1_2}\beta_{4_2}\alpha_{3_2}\alpha_{5_1}$	1.26	1.61	**2.18**	**2.13**
11	12	$\beta_{2_2}\beta_{1_2}\beta_{4_3}\beta_{3_3}\alpha_{5_1}$	1.27	1.45	**1.93**	**1.81**
12	7	$\alpha_{2_1}\beta_{1_2}\beta_{4_3}\beta_{3_3}\beta_{5_2}$	1.33	1.53	2.03	1.95
13	6	$\alpha_{2_1}\beta_{1_2}\beta_{4_3}\beta_{3_3}\alpha_{5_1}$	1.45	1.80	2.49	2.40

10 cells. Path costs and lower bounds have been normalized with respect to the A* path cost. The HA* generates the optimal path for each homotopy class. Most of the solutions obtained with the HBug have a lower cost than the HRRT solutions, however, the difference is very small.

Fig. 5 depicts the paths of the two homotopy classes with smaller lower bound in Table 2 generated with the HA*, the HRRT and the HBug path planners. Fig. 6 depicts the accumulated computation time for each path, which takes into account the computation of the reference frame, the topological graph, the homotopy classes with their lower bound and the paths generation. The generation of the paths for the whole set of homotopy classes took almost 4.5 s using the HA*. The path of the fifth class (index 1) was the fastest to be generated (94.4 ms) and the path of the eleventh class (index 12) the slowest (603.8 ms). The total computation time with the HRRT was reduced to 0.6 s with the fifth class (index 1) being the fastest one to be generated (12.2 ms) and the ninth class (index 5) the slowest (88.1 ms). Finally, the total computation time using the HBug was only 8.5 ms. The fastest path (class 7, index 10) was generated in 32 μs and the class 13 (index 6) path was the slowest one to be generated with only 76 μs. These low values justify the almost flat line for this path planner when compared with the HA* and the HRRT in Fig. 6.

When operating under time restrictions, it is possible to stop the path search when the lower bound of the next homotopy class to be computed is higher than the minimum cost of the paths already computed. In such cases, it is ensured that the best path has already been computed because it is not possible to obtain a path with a lower cost than its lower bound. For instance, in Table 2 the HA* would stop before computing a path for the fifth homotopy class (index 1) since its lower bound (1.05) is higher than the path length obtained with the first class (index 2, cost 1). The accumulated computation time would be 1.54 s (see Fig. 6). On the other hand, the HRRT and HBug algorithms would stop before computing the path of the two last classes with a lower computation time than the HA* (0.53 s for the HRRT and 8.4 ms for the HBug) at the expense of higher path costs.

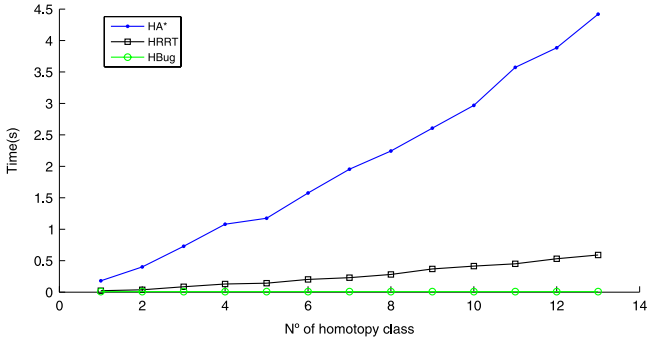


Fig. 6. Accumulated computation time of the paths using HA*, HRRT and HBug algorithms for each homotopy class on Table 2 sorted according their lower bound.

5.2. Large scenario

The scalability of our method has been tested in a 1000 × 1000 pixels bitmap scenario with 15 irregular obstacles (see Fig. 7). The construction of the reference frame, the topological graph and the generation of 112 homotopy classes with their lower bound computation took 0.304 s. Table 3 shows the five homotopy classes with the smaller lower bound. Fig. 8 depicts the normalized cost obtained with the HA*, HRRT and HBug with respect to the optimal path cost for each homotopy class sorted by their normalized lower bound.

Fig. 9 shows the computation time for each path represented in log scale. The paths computed with the HA* took between 7.49 s, for the class of the optimal global path (class 1, index 25), and 53.69 s for the class 75 to be generated with a cost 1.6 times of the global optimal solution. The mean computation time was 30.99 s. The HRRT computed the best solution in 0.069 s (class 1, index 25) with a cost 1.4 times the global optimal solution. However, the fastest path was generated in 0.038 s (class 14, index 93) with a path cost 1.83 times the global optimal solution. On the other

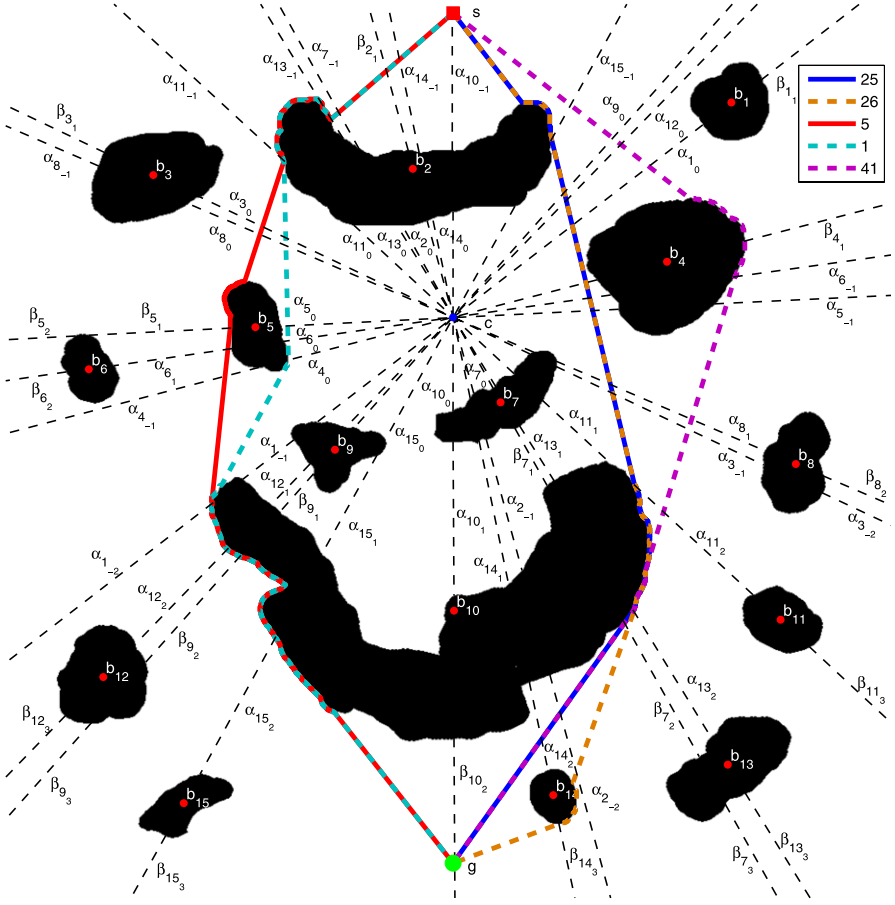


Fig. 7. Paths of the five homotopy classes with the smaller lower bound computed with the HBug. The class associated to the index can be found in Table 3.

Table 3
The homotopy classes of the Fig. 7 environment with the smaller lower bound.

N°	Idx	Homotopy class
1	25	$\alpha_{15_{-1}}\alpha_{9_0}\alpha_{12_0}\alpha_{1_0}\alpha_{4_0}\alpha_{6_0}\alpha_{5_0}\alpha_{8_1}\alpha_{3_{-1}}\alpha_{11_2}\alpha_{13_2}\beta_{7_2}\alpha_{2_{-2}}\alpha_{14_2}\beta_{10_2}$
2	26	$\alpha_{15_{-1}}\alpha_{9_0}\alpha_{12_0}\alpha_{1_0}\alpha_{4_0}\alpha_{6_0}\alpha_{5_0}\alpha_{8_1}\alpha_{3_{-1}}\alpha_{11_2}\alpha_{13_2}\beta_{7_2}\alpha_{2_{-2}}\beta_{14_3}\beta_{10_2}$
3	5	$\alpha_{10_{-1}}\alpha_{14_{-1}}\beta_{2_1}\alpha_{7_{-1}}\alpha_{13_{-1}}\alpha_{11_{-1}}\alpha_{3_0}\alpha_{8_0}\beta_{5_1}\alpha_{6_1}\alpha_{4_{-1}}\alpha_{1_{-2}}\alpha_{12_2}\beta_{9_2}\alpha_{15_2}$
4	1	$\alpha_{10_{-1}}\alpha_{14_{-1}}\beta_{2_1}\alpha_{7_{-1}}\alpha_{13_{-1}}\alpha_{11_{-1}}\alpha_{3_0}\alpha_{8_0}\alpha_{5_0}\alpha_{6_0}\alpha_{4_0}\alpha_{1_{-2}}\alpha_{12_2}\beta_{9_2}\alpha_{15_2}$
5	41	$\alpha_{15_{-1}}\alpha_{9_0}\alpha_{12_0}\alpha_{1_0}\beta_{4_1}\alpha_{6_{-1}}\alpha_{5_{-1}}\alpha_{8_1}\alpha_{3_{-1}}\alpha_{11_2}\alpha_{13_2}\beta_{7_2}\alpha_{2_{-2}}\alpha_{14_2}\beta_{10_2}$

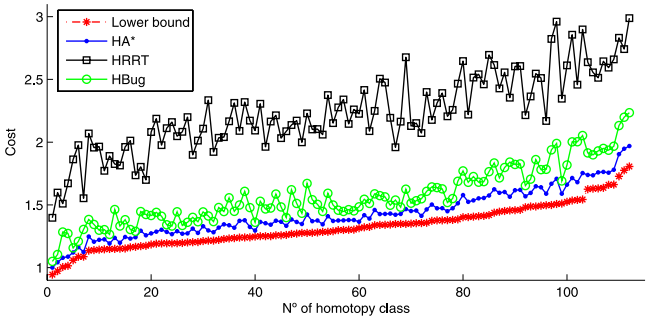


Fig. 8. Normalized cost and normalized lower bound for paths of each homotopy class.

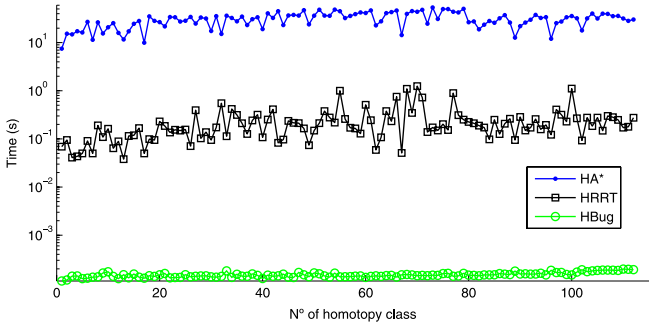


Fig. 9. Computation time for paths of each homotopy class.

hand, the slowest (class 70, index 19) took 1.248 s with a cost of 2.19 times the global optimal solution. The mean computation time for each path was only 0.239 s, almost 130 times faster than the HA* at the expense of computing higher cost solutions. The HBug computation time goes from 1.12×10^{-4} s (class 1, index 25) with a path cost only 1.05 times the optimal solution to 1.98×10^{-4} s with a cost 2.24 times above the global optimal solution (class 110, index 102). The mean computation per path was only 1.50×10^{-4} s, almost 1600 times faster than the HRRT and 2×10^5 times faster than the HA*. In this environment, the paths for the whole set of homotopy classes were computed in 16.8 ms, which is almost negligible when compared with the 304 ms spent in the generation of the reference frame, topological graph and the homotopy classes with their lower bound.

When operating under time restrictions the HA* would stop the path generation process before computing the path of the third homotopy class (index 5 in Table 3) with an accumulated time of 31.25 s. The HRRT would compute until the eightieth homotopy class (index 35) in 19.167 s. Finally, the HBug would generate paths until the fifth homotopy class (index 41) in 304.5 ms taking into account all the steps of the method. Fig. 7 depicts these paths generated with the HBug.

5.2.1. Comparative results

The proposed path planners have been compared against the A* [4], the RRT [5], their respective anytime versions (Anytime Repairing A* (ARA*) [43] and Anytime-RRT (ARRT) [44]) and the Bug2 algorithm [40]. A*, RRT and Bug2 are designed to compute only one path towards the goal; the ARA* and ARRT compute several paths but without taking into account their homotopy. Because of this, the comparison is against the five homotopy classes with the smaller lower bound shown in Table 3. Fig. 10 depicts the comparison. The data obtained with the RRT and the ARRT are the average of 100 executions to ensure the stabilization of the results.

The A* returned the optimal path in 11.90 s. The ARA* generated the first solution in 7.40 s and found the optimal solution after 301 s. The RRT algorithm took 0.012 s to compute a path with a cost 1.48 times the global optimal solution. The ARRT took 3.13 s to obtain the first solution and 78.30 s to compute all of them, ensuring that any new solution generated is closer to the optimal one. The Bug2 algorithm computed the path in 0.044 s with a cost 1.90 times the optimal solution. In order to obtain the best possible path with this algorithm, we have manually chosen the directions to surround the obstacles: the *m*-line, which connects the start with the goal, intersects with the obstacles labeled b_1, b_7 and b_{10}; the directions are clockwise for b_1, counter-clockwise for b_7 and clockwise for b_{10}.

The HA* computed the optimal path (index 25) in 7.79 s and obtained the path for the five selected homotopy classes in 71.61 s. The computation of the optimal path of the best homotopy class with the HA*, which corresponds to the A* solution, was 1.53 times faster than the A*. The computation of the first path took 0.390 s longer than the first solution of the ARA*, but the ARA* needed more iterations to refine the path to obtain the same solution as the HA*. The best HRRT solution (index 25) was computed in 0.373 s with a cost 1.40 times the optimal one, and obtained the path for the five homotopy classes with the smaller lower bound in 0.603 s.

The HRRT computation time was better than the A* and the ARRT. The reduced computation time of the RRT could not be reached due to the computation of the homotopy classes and the extra load of checking the topological restrictions of the HRRT. Despite class with index 25 had a lower cost than the RRT solution, the difference was small. The HBug computed the best solution (index 25) in 0.304 s with a cost 1.05 times the optimal one, and obtained the path for the whole set of homotopy classes in 0.321 s. As stated earlier, the computation of the paths with the HBug for each homotopy class offers a very good performance. Only the RRT and Bug2 had lower computation times at the expense of finding higher cost solutions. Note that most of the time was spent in the computation of the reference frame, the topological graph and the generation of the homotopy classes with their lower bound.

Until now, the quality of all the solutions has been compared with the A* cost, which generates the global optimal path. However, the standard A* is not able to take into account any topology during the path generation. On the other hand, the HA* computes the shortest path for each homotopy class, which is the optimal solution according to the topological constraints. Therefore, a comparison of the solutions generated with the HRRT and the HBug against the HA* cost for each specific homotopy class is depicted

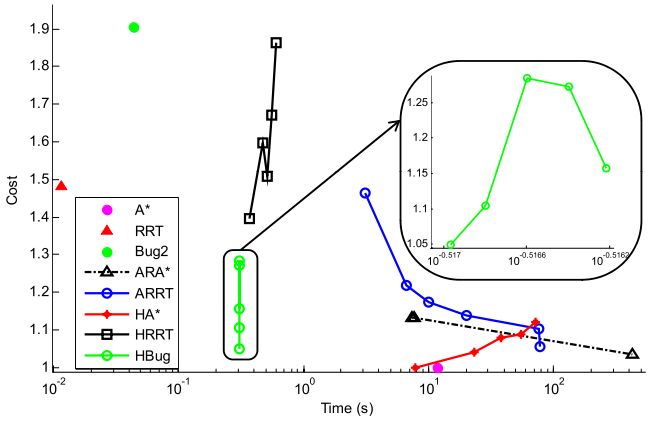


Fig. 10. Comparison of the HA*, HRRT and HBug paths of the five homotopy classes with the smaller lower bound vs. A*, RRT, ARA*, ARRT and Bug2 algorithms.

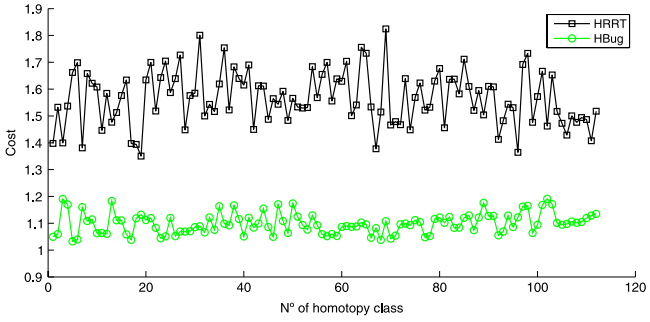


Fig. 11. HRRT and HBug paths cost with respect to the HA* cost for each homotopy class.

in Fig. 11. The HRRT generates solutions between 1.35 (class 15, index 73) and 1.82 (class 83, index 61), with a mean of 1.57 times with respect to the optimal path cost for the specific homotopy class generated with the HA*. The HBug generates solutions between 1.03 (class 5, index 41) and 1.19 (class 100, index 101), with a mean of 1.1 times with respect to the optimal path cost computed with the HA*.

5.3. Experiment in the Formigues Islands

Our path planning method has also been considered as a part of the TRIDENT European project (EU FP7 ICT-248497) which is focused on providing a new methodology for multipurpose underwater intervention tasks with diverse potential applications like underwater archeology, oceanography and offshore industries, going beyond present-day methods typically based on manned and/or purpose built systems. A team of two cooperative heterogeneous robots with complementary skills, an Autonomous Surface Vessel (ASV) and an AUV [45] endowed with a dexterous manipulator will be used to perform underwater manipulation tasks.

The experiments of the project consist of two steps. In the first step, the AUV is deployed from the ASV to perform a path following survey, in which it gathers optical/acoustic data from the seafloor to do an accurate terrain tracking. After the survey, the AUV docks in the ASV and sends the data back to a ground station where a map is set up and a target object is identified by the end user. In the second step, the ASV navigates towards a waypoint near the intervention area where the AUV is launched to search for the object. When the object has been found, the AUV switches to free floating navigation mode to start the manipulation process.

Our proposal is used in the second step of the experiments when the AUV has to compute safe paths for the intervention based on

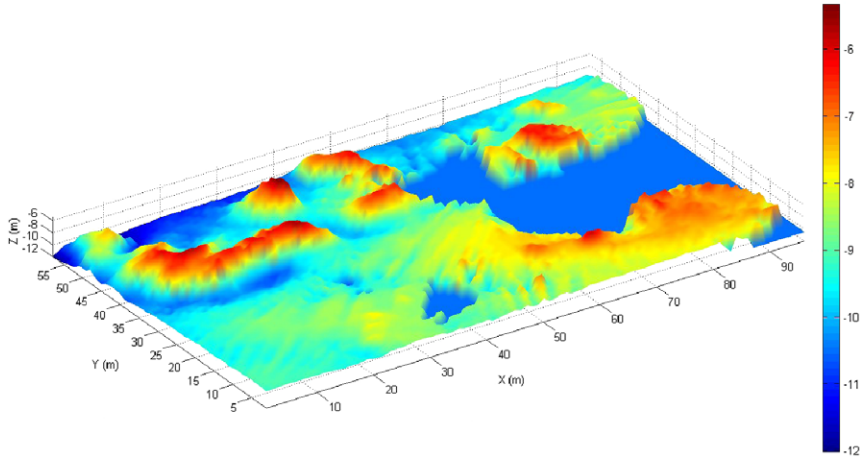


Fig. 12. Bathymetric map obtained in the Formigues Islands.

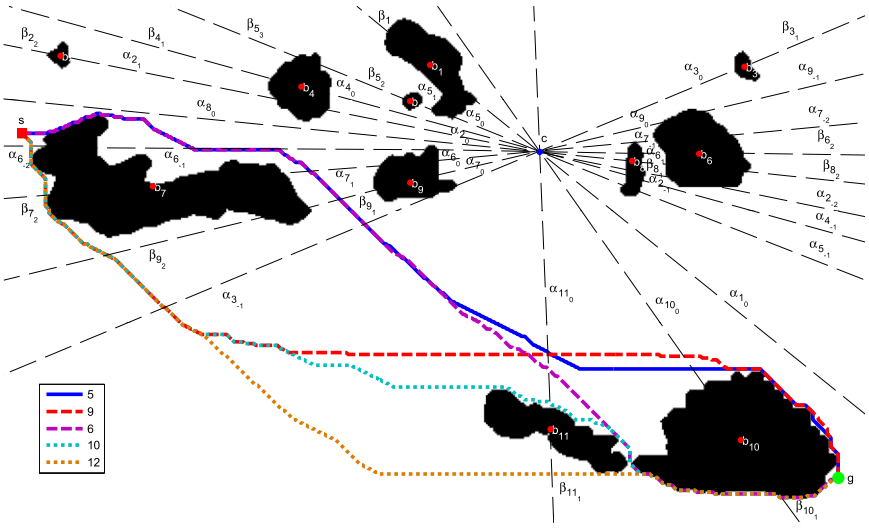


Fig. 13. Paths of the five homotopy classes with the smaller lower bound generated using the HA*. The class associated to the index can be found in Table 4.

the generated map. The method has been applied to a bathymetric map. The path planning method is in charge of generating the different homotopy classes for the environment. After that, the homotopic path planners compute the paths for each class. Since the target for the intervention has to be selected by the user once the vehicle has performed the survey and reached the surface, time is not a constraint. Selecting the trajectory to reach the intervention area prevents the AUV from traversing undesirable areas with strong local currents due to the obstacles configuration or muddy areas not suitable to navigate close by, which can be perceived by the survey sensors but not by the on-line navigation sensors.

The bathymetry was gathered with a Multibeam Profiler Sonar (MPS) Model 837B "Delta T" 1000 from Imagenex. This sensor is a multiple receiver sonar system designed to provide detection over a wide field of view for bottom profiling applications. The MPS has 480 beams spread in a 120° swath overture with a beam rate frequency of 5–10 Hz, depending on the depth of the scanned area. The sensor has a Motion Reference Unit (MRU) sensor to capture roll, pitch and heading.

The experiment took place in the Formigues Islands, on the Catalan coast. A DGPS sensor took georeferenced positions while the MPS was gathering data during a survey mission in an area of 100 × 58 m. The datasets gathered with the MPS and the DGPS were merged with commercial software provided by Imagenex to generate the bathymetric map, depicted in Fig. 12 with a 0.2 m resolution.

Table 4
The five homotopy classes with the smaller lower bound in Fig. 13 scenario.

N°	Idx	Homotopy class
1	5	$\alpha_{6_{-1}}\alpha_{7_1}\beta_{9_1}\alpha_{3_{-1}}\alpha_{11_0}\alpha_{10_0}$
2	9	$\alpha_{6_{-2}}\beta_{7_2}\beta_{9_2}\alpha_{3_{-1}}\alpha_{11_0}\alpha_{10_0}$
3	6	$\alpha_{6_{-1}}\alpha_{7_1}\beta_{9_1}\alpha_{3_{-1}}\alpha_{11_0}\beta_{10_1}$
4	10	$\alpha_{6_{-2}}\beta_{7_2}\beta_{9_2}\alpha_{3_{-1}}\alpha_{11_0}\beta_{10_1}$
5	12	$\alpha_{6_{-2}}\beta_{7_2}\beta_{9_2}\alpha_{3_{-1}}\beta_{11_1}\beta_{10_1}$

In the experiment, it was assumed that the vehicle would navigate at a 7.5 m depth. Therefore, using an Occupancy Grid Map (OGM) technique [46], the cells in the bathymetric map with a lower depth were mapped as occupied and the cells with a higher depth were mapped as free. Fig. 13 depicts the resultant C-Space as a 500 × 290 bitmap. The construction of the reference frame, the topological graph and the generation of 45 homotopy classes with their lower bound computation took 0.273 s. Although in the application only a specific homotopy class has to be selected by the operator, we have executed the proposed path planners for all homotopy classes to provide a detailed example in a real dataset. Fig. 14 depicts the normalized cost with respect to the optimal path cost computed with the A* algorithm and Fig. 15 shows the computation time. In both images the homotopy classes have been sorted by their normalized lower bound. Table 4 shows the five

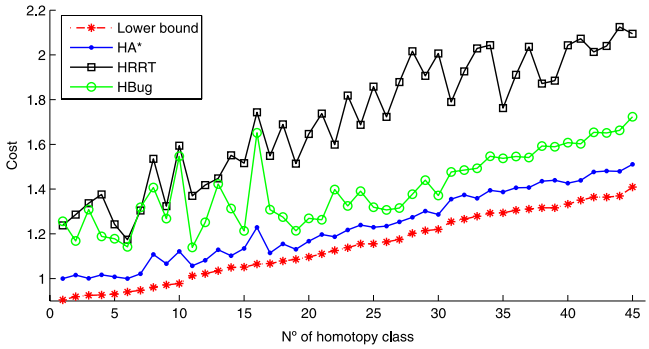


Fig. 14. Normalized cost and normalized lower bound for paths of each homotopy class.

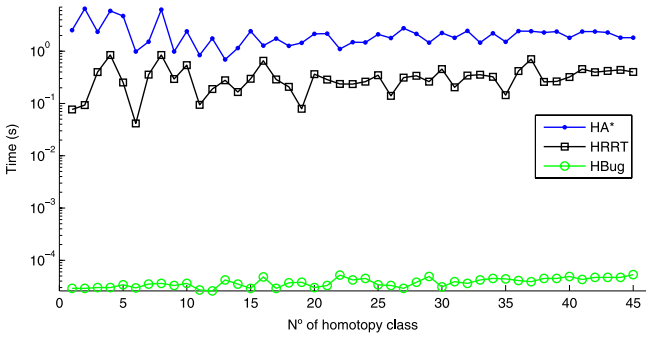


Fig. 15. Computation time for paths of each homotopy class.

best homotopy classes according to their lower bound. Their paths generated with the HA* are depicted in Fig. 13.

Using the HA* the whole process was computed in 99.2 s, with HRRT it took 14.959 s and HBug only needed 0.275 s. When operating under real-time constraints, it is possible to stop before computing the class 11 path when using the HA*. In such cases, the process would take 34.396 s. Using the HRRT the process would stop with the path computation of class 26 (index 17) taking 8.145 s and with the HBug would require computing until path of class 23 (index 26) in 0.274 s. As observed in the synthetic scenarios, the time to compute the paths with the HBug is almost negligible. Note that the computation times take into account all the steps of the process.

5.4. Discussion

The method of generating the homotopy classes was initially combined with the HRRT because RRT-based solutions have been used with success in contexts where high performance is required. The HRRT constrains the growing of a tree to those directions that satisfy a given homotopy class. With respect to the RRT, every time a new node is generated, it is checked whether the branch that connects the tree with the new node intersects any segment of the reference frame. This extra computational load prevents the HRRT from reaching the performance of the RRT, but ensures growth only in those regions that satisfy the homotopy class.

The HA* computes the optimal path cost for an input homotopy class. It is a graph-based search algorithm that exhaustively explores those parts of the search space that satisfy the class. Although it is the slowest of the solutions proposed, it can be used to provide a ground truth.

The HBug is an opportunistic algorithm that takes advantage of the homotopy classes generation process. During the path computation it only considers those nodes of the C-Space that belong to the lower bound path as long as it does not traverse an obstacle. When it does, the algorithm only looks for the free space around the intersected obstacle. Since the number of explored nodes by the HBug is drastically reduced and the computation time is improved from 3 to 5 orders of magnitude when compared with the HRRT and HA* algorithms respectively.

As a conclusion, the HA* should be used when optimal solutions are required at the expense of low performance. On the other hand, the HBug is suitable for applications in which time to perform path planning is highly constrained. Finally, the HRRT should be avoided since the HBug generates, most of the times, lower cost solutions with a fraction of the computation time. Furthermore, our method can be an alternative to traditional path planners in applications with large scenarios that require computing good solutions in a short time. In these situations it can generate a solution that has the same topology than the optimal solution at a fraction of the time when compared to the A* or anytime path planners. Note that the RRT and the Bug2 can generate the path even faster, but they do not guarantee the topology of the optimal solution.

Although the completeness of each stand-alone homotopic path planner depends on the algorithm they are based on, using them together with the homotopy class generation method provides deterministic completeness: if a homotopy class is generated implies there exists a path that avoids obstacles according to it; otherwise, the path does not exist if the homotopy class cannot be generated. In the latter case the path planner execution would not be required.

6. Conclusions and future work

In this paper we have addressed the path planning problem for robotic applications using homotopy classes. The topological information of the homotopy classes provides an important added valued to the path planning problem. Our approach first generates a set of homotopy classes for any 2d workspace provided as an input. Then, they are sorted according to a lower bound heuristic estimator. The homotopy classes are used to guide and to constrain topologically the path search. To take advantage of the homotopy classes we have adapted three well-known path planners from the graph-based search, probabilistic sample-based and bug-based approaches. The path planners have been tested in different simulated scenarios and compared against their respective non-homotopic, anytime versions and with themselves, showing a reduction of the computation time while keeping the topological constraints. The applicability of our proposal has been tested in a 100×58 m bathymetric map gathered with an MPS.

As future work, we will implement the HA* and the HBug into the Girona 500 AUV taking into account the kinodynamic constraints of the vehicle. A path following algorithm will be in charge of guiding the vehicle autonomously according to the generated trajectories. All the system parts will be tested working together in a controlled unknown environment.

Acknowledgments

This research was sponsored by the Spanish Government under the grant DPI2011-27977-C03-02 and the TRIDENT EU FP7-Project under the Grant Agreement No. ICT-248497.

Appendix. Applicability to unknown environments

A robot requires a navigation system to generate collision-free trajectories to carry out a mission. Generally, the navigation system is sub-divided in two groups: the global and the local navigation systems [47]. The former consists of a low resolution map building module and a path planner module that generates a path from the start to a goal position. The latter contains a localization system that computes the position of the robot within the global map,

a higher resolution map of the robot surroundings and a local navigator with reactive obstacle avoidance capabilities that follows the global path.

The homotopic path planning method is intended to be used as part of the global navigation system and the local navigation system has to generate the motion commands that follow the topologically constrained paths. Depending on the mission, the global map can be generated from scratch using data from the on-board navigation sensors or from previous existing data. A common approach is to generate a map based on topographic information, bathymetric data or by performing a two step mission as described in Section 5.3, in which the first run is used to collect data and to generate a map with the on-board sensors of the vehicle. In this case, the localization system estimates the robot position within the global map and updates it, if necessary.

When prior environment information is provided, the current implementation of our method is robust enough to deal with small changes such as updated size or shape of obstacles already detected that do not modify the topology. In [14,13] we showed that the computation of homotopy classes can be applied to an OGM generated with an imaging sonar sensor. The OGM was suitable for an AUV navigation as it took into account basic robot constraints such as size and turning radius for building the map.

In dynamic or unknown environments only those persistent obstacles detected over successive sensor readings that are bigger than a certain area should be considered for the global map, since adding multiple small obstacles can increase the complexity of the topological space and hence, the number of homotopy classes. When obstacles are added/removed, the reference frame and the topological graph change and may invalidate the homotopy classes previously generated. Our current implementation recomputes the homotopy classes from scratch. Although this process is done efficiently adding a minimum overhead to the whole process, a further study on how homotopy classes can be adapted to dynamic environments and how they evolve when new obstacles are added is needed.

References

[1] H. Choset, K.M. Lynch, S. Hutchinson, G.A. Kantor, W. Burgard, L.E. Kavraki, S. Thrun, Principles of Robot Motion: Theory, Algorithms, and Implementations, MIT Press, Cambridge, MA, 2005.

[2] J. Barraquand, B. Langlois, J.-C. Latombe, Numerical potential field techniques for robot path planning, IEEE Trans. Syst. Man Cybern. 22 (2) (1992) 224–241.

[3] D. Ferguson, M. Likhachev, A. Stentz, A guide to heuristic-based path planning, in: Proc. of the Inter. Workshop on Planning under Uncertainty for Autonomous Systems, Int. Conf. on Automated Planning and Scheduling, ICAPS, June, 2005.

[4] P. Hart, N. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybern. 4 (2) (1968) 100–107.

[5] S.M. LaValle, Planning Algorithms, Cambridge University Press, Cambridge, UK, 2006, Available at: http://planning.cs.uiuc.edu/.

[6] J. Ng, T. Braunl, Performance comparison of bug navigation algorithms, J. Intell. Robot. Syst. 50 (2007) 73–84. http://dx.doi.org/10.1007/s10846-007-9157-6.

[7] J. Munkres, Topology; A First Course, Prentice-Hall, 1974.

[8] H. Seifert, W. Threlfall, J. Birman, J. Eisner, Seifert and Threlfall, A Textbook of Topology, in: Pure and Applied Mathematics, Academic Press, 1980.

[9] E. Hernández, M. Carreras, J. Antich, P. Ridao, A. Ortiz, A topologically guided path planner for an AUV using homotopy classes, in: Proc. of the IEEE Int. Conference on Robotics and Automation, ICRA. Shanghai, China, May, 2011, pp. 2337–2343.

[10] E. Hernández, M. Carreras, P. Ridao, A path planning algorithm for an AUV guided with homotopy classes, in: Proc. of the 21st Int. Conf. on Automated Planning and Scheduling, ICAPS. Freiburg, Germany, June, 2011.

[11] E. Hernández, M. Carreras, J. Antich, P. Ridao, A. Ortiz, A search-based path planning algorithm with topological constraints. Application to an AUV, in: Proc. of the 18th IFAC World Congress. Milan, Italy, August, 2011.

[12] E. Hernández, M. Carreras, P. Ridao, A bug-based path planner guided with homotopy classes, in: Proc. of the 9th Int. Conf. on Informatics and Control, Automation and Robotics, ICINCO. Rome, Italy, July, 2012.

[13] E. Hernández, M. Carreras, P. Ridao, A. Mallios, Homotopic path planning for an AUV on maps improved with scan matching, in: Proc. of IFAC Workshop on Navigation, Guidance and Control of Underwater Vehicles, NGCUV, Porto, Portugal, April, 2012.

[14] E. Hernández, M. Carreras, E. Galceran, P. Ridao, Path planning with homotopy class constraints on bathymetric maps, in: OCEANS, 2011 IEEE, Spain, June, 2011, pp. 1–6.

[15] B. Chazelle, A theorem on polygon cutting with applications, in: 23rd Annual Symposium on Foundations of Computer Science, SFCS, November, 1982, pp. 339–349.

[16] J. Hershberger, J. Snoeyink, Computing minimum length paths of a given homotopy class, Comput. Geom., Theory Appl. 4 (1991) 331–342.

[17] S.-W. Cheng, J. Jin, A. Vigneron, Y. Wang, Approximate shortest homotopic paths in weighted regions, in: O. Cheong, K.-Y. Chwa, K. Park (Eds.), Algorithms and Computation, in: Lecture Notes in Computer Science, vol. 6507, Springer, Berlin, Heidelberg, 2010, pp. 109–120.

[18] A. Efrat, S. Kobourov, A. Lubiw, Computing homotopic shortest paths efficiently, in: R. Möhring, R. Raman (Eds.), Algorithms—ESA 2002, in: Lecture Notes in Computer Science, vol. 2461, Springer, Berlin, Heidelberg, 2002, pp. 277–288.

[19] S. Bespamyatnikh, Computing homotopic shortest paths in the plane, J. Algorithms 49 (2003) 284–303.

[20] B. Speckmann, K. Verbeek, Homotopic rectilinear routing with few links and thick edges, in: A. López-Ortiz (Ed.), LATIN 2010: Theoretical Informatics, in: Lecture Notes in Computer Science, vol. 6034, Springer, Berlin, Heidelberg, 2010, pp. 468–479.

[21] D. Grigoriev, A. Slissenko, Polytime algorithm for the shortest path in a homotopy class amidst semi-algebraic obstacles in the plane, in: Proc. of the Int. Symposium on Symbolic and Algebraic Computation (ISSAC), ACM, New York, NY, USA, 1998, pp. 17–24.

[22] Y. Fujita, Y. Nakamura, Z. Shiller, Dual Dijkstra search for paths with different topologies, in: IEEE Int. Conf. on Robotics and Automation, ICRA, vol. 3, September, 2003, pp. 3359–3364.

[23] Z. Shiller, Y. Fujita, D. Ophir, Y. Nakamura, Computing a set of local optimal paths through cluttered environments and over open terrain, in: IEEE Int. Conf. on Robotics and Automation, ICRA, vol. 5, April-1 May, 2004, pp. 4759–4764.

[24] B. Banerjee, B. Chandrasekaran, A framework for planning multiple paths in free space, in: Proc. of 25th Army Science Conference. Orlando, FL, 2006.

[25] S. Bhattacharya, V. Kumar, M. Likhachev, Search-based path planning with homotopy class constraints, in: Proc. of the National Conference on Artificial Intelligence, AAAI, vol. 2. Atlanta, Georgia, USA, July, 2010, pp. 1230–1237.

[26] S. Bhattacharya, M. Likhachev, V. Kumar, Topological constraints in search-based robot path planning, Auton. Robots 33 (3) (2012) 273–290.

[27] S. Kim, S. Bhattacharya, V. Kumar, Path planning for a tethered mobile robot, in: IEEE Int. Conf. on Robotics and Automation, ICRA, vol. 22, 2014.

[28] K.D. Jenkins, The shortest path problem in the plane with obstacles: A graph modeling approach to producing finite search lists of homotopy classes (Master's thesis), Naval Postgraduate School, Monterey, California, 1991, June.

[29] A. Cuerington, The shortest path problem in the plane with obstacles: Bounds on path lenghts and shortest paths within homotopy classes (Master's thesis), Naval Postgraduate School, Monterey, California, 1991, June.

[30] R. Milgram, S. Kaufman, Topological characterization of safe coordinated vehicle motions, in: IEEE Int. Conf. on Robotics and Automation, ICRA, vol. 3, 2000, pp. 2039–2045.

[31] E. Schmitzberger, J. Bouchet, M. Dufaut, D. Wolf, R. Husson, Capture of homotopy classes with probabilistic road map, in: IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, IROS, vol. 3, 2002, pp. 2317–2322.

[32] L. Kavraki, P. Svestka, J.-C. Latombe, M. Overmars, Probabilistic roadmaps for path planning in high-dimensional configuration spaces, IEEE Trans. Robot. Autom. (ICRA) 12 (4) (1996) 566–580.

[33] S. Cabello, Y. Liu, A. Manler, J. Snoeyink, Testing homotopy for paths in the plane, in: Proc. of the Symposium on Computational Geometry, SoCG. Barcelona, Spain, June, 2002, pp. 160–169.

[34] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[35] R. Dechter, J. Pearl, Generalized best-first search strategies and the optimality of A*, J. ACM 32 (3) (1985) 505–536.

[36] S.M. LaValle, J.J. Kuffner, Randomized kinodynamic planning, in: Proc. of the IEEE Int. Conf. on Robotics and Automation, ICRA, vol. 1, September, 1999, pp. 473–479.

[37] J. Kim, J. Ostrowski, Motion planning a aerial robot using rapidly-exploring random trees with dynamic constraints, in: IEEE Int. Conf. on Robotics and Automation, ICRA, vol. 2, September, 2003, pp. 2200–2205.

[38] S.M. LaValle, J.J. Kuffner, Randomized kinodynamic planning, Int. J. Robot. Res. 20 (5) (2001) 378–400.

[39] J. Antich, A. Ortiz, J. Minguez, A bug-inspired algorithm for efficient anytime path planning, in: IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, IROS, October, 2009, pp. 5407–5413.

[40] V. Lumelsky, A. Stepanov, Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape, Algorithmica 2 (1987) 403–430.

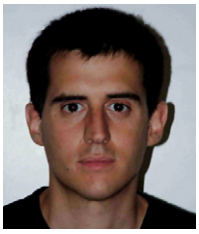
[41] F.S. Hill Jr., The Pleasures of "Perp Dot" Products, Academic Press Professional, Inc., San Diego, CA, USA, 1994, pp. 138–148.

[42] F. Chang, C. -J. Chen, C.-J. Lu, A linear-time component-labeling algorithm using contour tracing technique, Comput. Vis. Image Underst. 93 (2004) 206–220.

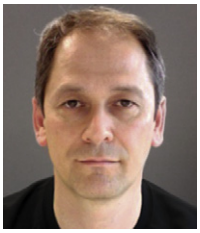
[43] M. Likhachev, G. Gordon, S. Thrun, ARA*: Anytime A* with provable bounds on sub-optimality, in: Proc. of the Advances in Neural Inforamtion Processing Systems. NIPS, vol. 16, MIT Press, 2004.

[44] D. Ferguson, A. Stentz, Anytime RRTs, in: Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, IROS, October, 2006, pp. 5369–5375.
[45] D. Ribas, N. Palomeras, P. Ridao, M. Carreras, A. Mallios, Girona 500 AUV, from survey to intervention, IEEE/ASME Trans. Mechatronics 17 (1) (2012) 46–53.
[46] E. Hernández, P. Ridao, A. Mallios, M. Carreras, Occupancy grid mapping in an underwater structured environment, in: Proc. of the 8th IFAC Int. Conference on Manoeuvring and Control of Marine Craft, MCMC, Guaruja, Sao Paulo, Brazil, September, 2009.
[47] J. Mínguez, L. Montesano, L. Montano, An architecture for sensor-based navigation in realistic dynamic and troublesome scenarios, in: IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, IROS, vol. 3, September, 2004, pp. 2750–2756.

Emili Hernandez received his M.Sc. and Ph.D. degrees in computer engineering from the University of Girona, Girona, Spain in 2005 and 2012 respectively. His research interests include planning and path planning focused on autonomous underwater robots, localization and map building for online navigation purposes. He has been involved in National and European research projects related with underwater robotics. He is currently a postdoctoral researcher at the Autonomous Systems Lab. (ASL) at the Commonwealth Scientific and Industrial Research Organization (CSIRO), Australia.



Marc Carreras received the M.Sc. in Industrial Engineering (1998) and the Ph.D. in Computer Engineering (2003) from the University of Girona, Spain. His research activity is mainly focused on robot learning and intelligent control architectures of autonomous underwater vehicles. He joined the Institute of Informatics and Applications, University of Girona in September 1998. Currently, he is an associate professor with the Department of Computer Engineering of the University of Girona and member of the Research Center in Underwater Robotics (CIRS) of Girona. He is involved in National and European research projects and networks about underwater robotics.



Pere Ridao received the M.Sc. in Computer Science in 1993 from the Technical University of Catalonia, Spain, and the Ph.D. in Computer Engineering in 2001 from the University of Girona, Spain. His research activity is focused on control architectures, navigation and mission control systems for underwater robots. He is an associate professor with the Computer Engineering Department of the University of Girona and the head of the Research Center in Underwater Robotics (CIRS). He is a member of the IFAC's Technical Committee on Marine Systems and the Spanish RAS chapter, and secretary of the Spanish OES chapter.