JINT manuscript No.
(will be inserted by the editor)

Applying frontier cells based exploration and Lazy Theta* path planning over single grid-based world representation for autonomous inspection of large 3D structures with an UAS*

Margarida Faria, Ivan Maza and Antidio Viguria

Received: date / Accepted: date

Abstract Aerial robots are a promising platform to perform autonomous inspection of infrastructures. For this application, the world is a large and unknown space, requiring light data structures to store its representation while performing autonomous exploration and path planning for obstacle avoidance. In this paper, we combine frontier cells based exploration with the Lazy Theta* path planning algorithm over the same light sparse grid - the octree implementation of octomap. Test-driven development has been adopted for the software implementation and the subsequent automated testing process. These tests provided insight into the amount of iterations needed to generate a path with different voxel configurations. The results for synthetic and real datasets are analyzed having as baseline a regular grid with the same resolution as the maximum resolution of the octree. The number of iterations needed to find frontier cells for exploration was smaller in all cases by, at least, one order of magnitude. For the Lazy Theta* algorithm there was a reduction in the number of iterations needed to find the solution in 75% of the cases. These reductions can be explained both by the existent grouping of regions with the same status and by the ability to confine inspection to the known voxels of the octree.

Keywords Structure Inspection · UAS Applications · Path Planning · Autonomous Exploration

1 Introduction

The increasing need of UAS usage for remote off-shore monitoring activities has raised different challenges. The European Strategy for Marine and Maritime Research states the need

*The first author has been funded by the European Unionś Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 64215 and the other two authors received funding from the MULTIDRONE (H2020-ICT-731667) and AEROARMS (H2020-ICT-644271) European projects

Margarida Faria E-mail: mfaria@catec.aero and Antidio Viguria E-mail: aviguria@catec.aero
Center for Advanced Aerospace Technologies, Calle Wilbur y Orville Wright, 19, 41300 La Rinconada, Sevilla, Spain

Ivan Maza E-mail: imaza@us.es
in the Robotics, Vision and Control Group, University of Seville, Avda. de los Descubrimientos s/n, 41092, Sevilla, Spain

to protect the vulnerable natural environment and marine resources sustainably. The use of UAS provides increased endurance and flexibility while reducing environmental impact, the risk for human operators and the total cost of operations. The work in this paper has been developed in the framework of the MarineUAS [1] initiative, a European Union funded doctoral program which strategically strengthens research training on Unmanned Aerial Systems for Marine and Coastal Monitoring.

Several constraints come with this type of scenario. One such limitation is the large size of the volume to be explored, as it can be seen in Fig. 1. The facilities that need inspection are several orders of magnitude larger than the size of the UAS. In some cases, there is no "a priori" 3D available map usable by the UAS for navigation purposes. Petro-stations and windmill farms are examples of offshore facilities that would benefit from regular systematic and autonomous inspection. Both have several structural features that need to be examined vertically, calling for a full 3D exploration. Another constraint is time. The operation must be done as fast as possible since it may require the activity of the facility to be suspended.

Any operation done offshore is extremely expensive and has inherently rougher conditions. Enabling systematic and autonomous inspection will shift to machines the work at which they excel: repetition without deviation taking into account many factors and parameters. Additionally, by simplifying inspections they will get done more often. Safety increases, as humans need to do less hands-on work and they became less exposed to risky situations.

Fig. 1 Inspecting under this offshore petro-station offers an example of inspection that cannot rely on any global positioning system localization.

As it has been previously mentioned, this work is focused on the autonomous exploration and path planning levels of the UAS. Lower levels such as the trajectory generator and the controller will handle with temporal constraints of the mission and kinematic and dynamic limitations of the vehicle.

The orientation of the multirotor is not considered in either the exploration or the path planning algorithm. The reason why this important parameter can be disregarded is twofold. At motion level, the multirotor can move in all directions. At the sensor level, there are available sensors with a 360 range, as is the case of rotating LIDARs.

The first step to achieve efficient autonomous exploration is to choose the proper data structure for world representation. The characteristics of real-world scenarios need to be taken into account as they will strain the memory capacity of the system. In real-world

[1] http://marineuas.eu

environments, the free space is usually grouped. Particularly in offshore structures, it is also likely to occupy most of the area. Also, the occupied space is often clustered, like the pillars in an offshore platform, the windmill's tower or its blades.

The characteristics of the features used for autonomous navigation should also be taken into account. The driving goal of the UAS is to explore a given volume. Many options are available, but this work is focused on the classical and widely used frontier exploration approach.

This exploration has the advantage of simplicity. By first integrating a simple algorithm we create the possibility of checking its limitations for the particular application experimentally. Another advantage is that allows to keep down the computational load. A frontier cell is a location in the world representation map that is explored and unoccupied but has unexplored space in its vicinity. These places are of particular interest as they yield the highest information gain.

Due both to the focus on low memory requirements and information organization, the data structure selected will be the octree implementation done in the octomap [1] framework.

Techniques from software development will be adopted in the implementation to enable reproducibility. Namely test-driven development, supported by automated unit tests. In this methodology, a unit test is first designed to assert the implementation of a feature. Then the feature is developed until the test is passed, at which point another unit test is designed for another feature. This process continues iteratively until the full desired functionality is achieved. To verify the result of the algorithm under different initial conditions this form of testing will be used whenever possible. Furthermore, both algorithms will be applied to simulated and experimental data.

This paper extends previous work presented in [2]. The novelty of this work relies on the implementation of both the frontier cells exploration algorithm and the Lazy Theta* path planning algorithm over the same data structure for the representation of the 3D environment. This data structure can scale to large scenarios. No local reduction to a regular grid will be used at any point. The frontier cell algorithm was chosen for its simplicity. The group of suitable path planning algorithms is reduced as this is a single query problem. By choosing an Any-angle path planning algorithm the need of post-processing is significantly decreased, and in particular, Lazy Theta* is further optimized to reduce the number of line of sight checks. In addition, this algorithm has been applied successfully in competitions with autonomous multirotors [3].

The paper is organized as follows. Section 2 reviews related work in data structures for environment representation, exploration algorithms and path planning techniques. Section 3 reports relevant aspects for finding frontier cells on the different data structures under analysis. Section 4 describes the challenges and solutions adopted for the implementation, a modification of the baseline algorithm as well as the methods chosen to develop the code. In Sect. 5, the results obtained both with simulated and experimental data with each of the algorithms are detailed. Finally, section 6 closes the paper with the conclusions and future work.

2 Related Work

This section presents the data structures considered for storing the representation of the world, state of the art algorithms for autonomous exploration and path planning techniques with particular attention to the supporting data structures.

still call it grid based after adding visibility graphs and kd trees?

2.1 Efficient data structures for world representation

In this paper, several data structures have been analyzed, with a greater focus on those which are readily available as off the shelf libraries. The amount of data translated into point clouds from most sensors is exceptionally high. Thus, it is crucial to identify data structures that more than just compress data, also arrange information in an useful and efficient manner. One technique to organize the information is to create meshes. A popular algorithm is the Delaunay triangulation. Both PCL and CGAL [4] libraries provide implementations that construct meshes from the points clouds. This technique has the drawback of being dependent on the order the points are analyzed, introducing one additional source of variability. [5] overcomes this issue by creating a mesh for the visibility graph, around found obstacles, showing the computational time advantages of constructing visibility graphs for multiple query cases. Visibility graphs simplify the task of searching for neighbors by quantifying the relationship between samples according to the adopted metric. In [6], a software package for their generation is presented. This package brings into Matlab a tool to compute obstacle avoiding paths in a known world.

Another option is to discretize the world into spaces of the same dimension. Creating a regular grid can be done even without a dedicated library, in less elaborated cases. In these simple cases, random access can have a complexity of $\mathcal{O}(1)$. Any information can be stored per cell, although with more information comes increased memory usage.

To make the search more efficient, trees (with all their multitude of implementations and variations) are another option. The kd-trees are one option for the representation of the world [7]. In this representation, the separation of space is a reflexion of the topology of the existing objects resulting in a tree that precisely matches it. The PCL library offers one implementation integrating the FLANN library, where the kd-tree is used [8].

The octree is a flatter tree compared to the binary kd-tree due to its eight children. The topology is less closely reflected but has the advantage of a smaller hierarchical traversal of nodes when finding nodes in the neighborhood. Two notable implementations of this structure are available: the octomap library [9] and the PCL library. The latter offers several structures; this paper is focused on OctreePointCloudOccupancy [10]. Both offer random access with $\mathcal{O}(1)$ complexity and multiresolution queries. However, they differ in important details. The PCL library has a significant focus on the compression needed for streaming, while the octomap library targets navigation and exploration. In the PCL library, new measurements are added by summing points whereas the octomap library integrates them in a probabilistic manner. The concept of unknown space is also slightly different in each implementation. In the PCL library, space is either occupied or free. In the octomap library, only locations with information are created thus implicitly encoding unknown space. Both PCL and octomap libraries concern themselves with efficiency: the former focuses on read/write efficiency and for this reason, goes so far as to include a double buffered version of the structure. In the latter, the focus lies on memory efficiency with (almost) lossless compression regarding occupancy. For this reason, each node stores only the occupancy probability and one child pointer - forfeiting voxel size, coordinates, and the full children array.

2.2 Exploration

The concept of a frontier and a frontier cell repeatedly appears in the literature with the same basic idea. Frontier locations or cells are points in the world representations that satisfy two conditions: they are in free space and are connected to unexplored space. Due to the focus

on the underlying data structures and world representation, each work will be analyzed with these aspects in mind, keeping in sight how to identify the locations that yield the higher information gain.

Unmanned Ground Vehicles (UGVs) are particularly well suited to reduce the search space to 2D, due to their motion constraints. Many applications use probabilistic occupancy grids to tackle the task of exploring unknown (or partially unknown) spaces in a 2D search space. Reference [11] explains the concept of frontier cells over a regular grid in the context of probabilistic occupancy. This concept is not new, it was presented in [12], but due to its simplicity, it is still in use nowadays. In [13] an UGV is directed to the nearest frontier region, leaving the task of path planning to a lower level of the architecture with purely reactive obstacle avoidance. In [14], an UGV also travels to the nearest unexplored cell but creates roadmaps, i.e., Voronoi diagrams generated from the occupancy grid.

Many approaches [13, 15–17] encode the status of the cells as free, unknown and occupied either explicitly or by probability thresholds. Here each cell encodes a somewhat different approach to status: free, warning, travel and far. In [18] this approach is extended to multiple vehicles. Each frontier cell is scored according to a heuristic combination of occupancy probability and distance traveled. With this combination, the path planning problem is solved with the steepest descent of the heuristic function. In [19] the concept of a frontier is combined with a topological map: these edges are calculated as equidistant points to obstacles, the robot then travels this edges marking them as explored. The process continues for as long as there are unexplored edges. Reference [13] is an example of applying this approach to UAS by setting a safe altitude. The frontier cells found at this altitude are then clustered into labeled regions, disregarding the small and inaccessible frontiers. The remaining ones are considered as goals for the UAS, finding the next goal by applying a vector field histogram.

In 3D space, there are some applications of probabilistic occupancy grids to solve the next best view problem for robotic arms. In [17], to distinguish between unknown and unoccupied cells a ray casting algorithm is used to extrapolate free regions from the sensor location and occupied points. Holes in sensor measurements are extrapolated with Markov Random Fields. The frontier cells are scored integrating all this information into the gain to be later selected as best view. One example where the mission objectives are heavily taken into account is [16]. From the probabilistic grid, a mesh is created as a tool to find void regions. Unknown cells are set at the center of ellipsoids, which expand while maintaining a minimum fitting quality. The ellipsoids are then combined with frontiers and scored according to neighboring voids. The heuristic function maximizes the information gain taking into account the priority each region has for the mission. Knowledge about the area critical for obstacle avoidance has the highest priority, followed by the regions affected by the robot's tools.

Another structure used for 3D space is the octree, being its multi-resolution quality one of its defining characteristics. In [15] the list of frontier cells is compressed as clusters with an union-finding algorithm. The unknown spaces are handled as macro-regions through ellipsoid expansion. Finally, the clusters are combined with the ellipsoids to score frontiers according to unknown region dimension. Another paper using the octree as its underlying structure is [20]. However, the configuration space is searched employing a Rapidly Exploring Random Tree, grown iteratively through safe configurations in the direction of the frontier. In [21] the information of the known space is stored in a map structure similar to an elevation map, although for other purposes the associated point cloud is stored in different data structures. The search for areas with higher information gain is done through sample generation, to address the issue of search space explosion in 3D. From a sample of known

points of the environment, other points are generated and added to the pool. The model dynamics of the expansion of the molecules of a perfect gas is used to generate these points. Then, change rate between particle expansions is evaluated to find frontiers in regions.

2.3 Path Planning

In the field of motion planning, the problem of path planning has been extensively studied. Reference [7] presents different techniques within sampling-based motion planning, updating the motion planning algorithms presented in classical references such as [22,23].

To plan paths in 3D environments is complex and different methods have been proposed for online and offline planning. After considering the resource limitations (both regarding time and of computation) of an aerial robot, two approaches stand out as most frequently used: deterministic and non-deterministic, each one encompassing several methods. When generating paths with the deterministic approach is common to use probability as well as sampling. With the non-deterministic approach, both heuristic and graph-based methods are frequent.

The cost of building a fully connected graph is a good trade-off when solving a multi-query problem. One option when in a single query problem is to update the connectivity of the graphs taking into account the changes in the environment. In [24] this implementation is made dramatically reducing the cost of rebuilding the visibility graph. Another generalization of the visibility graphs from 2D to 3D is found in [25]. In this implementation, the visibility graph is composed of one obstacle graph and two supporting graphs. This approach relies on a world that is previously known.

Sampling-based algorithms like Rapidly-Exploring Random Trees (RRT) [26], [27] and Probabilistic Rad Maps (PRM) [28], are specifically designed to handle non-holonomic constraints (e.g., wheeled robots), high degrees of freedom and large spaces that require being rapidly and uniformly explored. A typical use case is a big manufacturing plant. Several variations of the RRT have been proposed. RRT* [29] produces very optimal paths at the expense of real-time rates. RRT-Connect [30] resolves the time issue, achieving faster solutions but generating longer paths. In [31], different probabilistic methods are used to solve the motion planning problem with UAVs. A continuous-time trajectory optimization method is used for real-time collision avoidance on multirotor UAVs.

The heuristic algorithms are specially designed to obtain the shortest path, exploring directly from the initial state to the target state. Examples are A* [32], Theta* [33] and D* [34]. These restrict the explored areas and get a runtime that is highly configurable and dependent on the number of variables and their resolution. The usual drawback imposed by discrete search techniques is that paths are formed by grid edges, so they are often not the shortest path in the continuous space. Fortunately, this issue was solved by the any-angle path planning Theta* and its Lazy Theta* variation [35], which also optimizes the computational load of the algorithm. One example of the application of a graph algorithm to a 3D search space is [36]. Here a simulated micro-UAS vehicle goes from start to goal using an AD* search algorithm for replanning. The underlying world representation is a 3D occupancy grid that is sampled where the samples are arranged as a multi-dimensional lattice.

Other solutions to generate 3D paths for aerial vehicles have been presented taking completely different approaches [37]. There are some examples of bio-inspired algorithms using neural networks [38], evolutionary algorithms [39] [40]. Others combine several algorithms in one architecture to benefit from the strengths of each one [41], [42], [43].

This work analysis a solution that needs a minimum amount of world representations and processing power compatible with online planning on-board a multirotor. The adopted world representation must have a memory footprint small enough to store the representation of large structures. For exploration, frontier cells will be used, whereas for path planning Lazy Theta* is applied as it can be implemented directly over octrees and has a smaller need for post-processing. The octomap implements octrees with little memory footprint while organizing location information with states suitable for exploration.

3 Exploration Algorithm based on Frontier Cells

In this section, the focus will be on the implementation details of the exploration algorithm. Its purpose is to identify points in the search space that will enable the collection of information. The frontier cell algorithm is used and relies heavily on knowing the neighbors of each cell. It will be implemented over two different data structures: a regular grid and a sparse grid.

A framework was created to assess the impact of the search space explosion in 3D in the different combinations of data structures under the same exact conditions. Each data structure needs to provide a function that returns its neighbors (getNeighbors) and a set of functions to implement iteration (initIteration and endIteration). In Algorithm 1 we can see when these generic functions are called to abstract from the world representation. The algorithm searches for frontier cells from the initial iteration condition until the end condition. The evaluation made for each cell selects locations that meet the following requirements: are in known space, are unoccupied and have at least one neighbor that is unexplored. The dimension flexibility (2D or 3D) is given by the bounding box set at the beginning of the iteration and by adjusting the directions considered for the neighbors. In all cases, the neighbors are in adjacent cells. The diagonal neighbors were disregarded after some preliminary tests since all the frontier regions were identified with and without them.

The algorithms have been implemented in C++ using the Robot Operating System (ROS) [44] as middleware. The open source implementation is freely available in the form of a self-contained Robot Operating System (ROS) unit tests. It was released under the MIT-license and can be obtained from the project dataStructureAnalysis[2]. More data structures can be integrated straightforwardly, being the only requirement to have the four generic functions: neighbor generation, iteration initialization, identification of the end condition and provide the next cell. They will be then called in the manner shown in Algorithm 1.

3.1 Regular Grid

The regular grid makes a discretization of the continuous space into cells that always have the same dimensions. This classical approach is frequently still used due to its simplicity. The full analysis of such a grid will always require a number of iterations given by multiplying the length, width and height of the 3D space considered. This implementation is based on a simple regular increment of the coordinates. The neighbors are always at the same distance.

[2] https://github.com/margaridaCF/dataStructureAnalysis

Algorithm 1 Generic procedure to find frontier cells, highlighting how to make the abstraction among data structures with functions initIteration, isExplored, isOccupied, getNeighbors, and getNextCell. For each cell first, it is determined if the cell is explored and in free space. When these requirements are met, its neighbors are evaluated. If there is at least one neighbor that is in unknown space, the cell will be classified as a frontier.

Input: *dimensions*, *variation_spec*
Output: *frontier_cells*

```
 1: cell = variation_spec.initIteration(min, max)
 2: while cell != variation_spec.endIteration() do
 3:     if isExplored(cell) && !isOccupied(cell) then
 4:         frontier = false
 5:         neighbors = variation_spec.getNeighbors(dimensions)
 6:         for all neighbors : n do
 7:             if !isExplored(n) then
 8:                 frontier = isExplored(n) || frontier
 9:             end if
10:         end for
11:         if frontier then
12:             frontier_cells.add(cell)
13:         end if
14:     end if
15:     cell = variation_spec.getNextCell()
16: end while
17: return frontier_cells
```

3.2 Sparse Grid

The sparse grid extends the concept of the regular grid by grouping same value regions. Its tree-like approach divides the space in different sizes, creating a high-resolution cell only to accommodate known points extracted from the point cloud.

In the octomap implementation, information is added not only for detected occupied locations but also for the free space. The free space is extrapolated from the locations of the sensor and of the obstacle. Another useful feature is the ability to transverse the grid passing only through known leafs, skipping all unexplored cells. This is quite convenient as by definition any unknown cell can never be a frontier cell.

The state of each location is stored in log-odds notation to enable probabilistic fusion of each new point cloud gathered by the on-board sensor. The compression is nearly lossless and there is only a trimming of the maximum and minimum values. Additionally, macro-regions with the same state will be analyzed only once. This will also reduce the number of cells that need to be examined to find frontiers, as the maximum size of each cell is always lower than the sensor limit. It is the regular grid equivalent of analyzing several cells at the same time.

The algorithm to advance to the next leaf location is detailed in [9]. To calculate the neighbors, a similar algorithm to regular grids is applied, with one main difference: the coordinates of each cell refer to the center of the location.

4 Lazy Theta* Implementation Details

The world representation will be constructed as the exploration develops. For this reason, the path planning problem is considered as a single query one. Within the several methods

identified as best performing in single query problems, the Lazy Theta* algorithm was chosen both because of the reduced need for post-processing smoothing and the reduced amount of line of sight checks needed. In particular, a variation of the Lazy Theta* algorithm has been implemented and tested with multirotors at the EUROC [45] competition showing its usability under real-time conditions and realistic constraints. The algorithm is described in [3] as a weighted heuristics to optimize the search space with an asymmetric volume, due to sensor restrictions. One relevant aspect of the implementation is the different approaches in the global planner and the local planner. The first applies the concept of a regular grid to access neighbors over an octree. The second also employs the idea of a regular grid but over a point cloud. Here the incoming point cloud was refined by removing the outliers and integrating the sensor error only after multiple confirmations. The local map was further updated by eliminating occupied points if no information of them was continuously received. Additionally, replanning is used throughout the mission for optimization purposes. However one of the identified bottlenecks is replanning to overcome large obstacles. This work acts as a follow-up to the promising results presented in [3] for the real-time generation of paths with obstacle avoidance.

Our implementation of the Lazy Theta* algorithm takes a different approach by directly implementing the algorithm over a 3D sparse grid representation of the world. Again, the data structure used is the octree implementation of the octomap framework. Aiming for an important reduction of the memory requirements, this will allow sharing the same world representation for exploration and path planning.

In this work, one of the challenges is to abandon the regular grid mindset entirely to take full advantage of the spacial clustering with sparse grids. The algorithm has been implemented in C++ under the Robot Operating System (ROS) [44] as middleware. The open source implementation is freely available including the unit tests. It is released under the MIT-license and can be obtained from the github project FlyingOctomap[3]. In the following the implementation methodology adopted and the adaptation of the algorithm to use Octomap are described.

4.1 Software Implementation Details

One recurring issue in robotics research is the difficulty of gathering the same conditions over time to reproduce the same experiment. The setup is not straightforward, software versions change and the hardware becomes unavailable. Some of these issues have been dealt with in the area of software development and can be mitigated with tools from that field. The adoption of software development methodologies, practices and techniques is starting to spread among the robotics field.

One example is the European project RobMoSys. Some of its key focus is on simplifying the setup and configuration software, predictable and traceable properties, certifiable systems, and better comparability through appropriate metrics (benchmarking) [46]. For the flexible general-purpose modeling of systems, the Unified Modeling Language was taken as reference [47].

Another example is the use of a tool that appeared in the last years as a response to the continuous integration of software developments, especially in the web development area. This tool is docker. It enables a precise account of all the software packages necessary

[3] https://github.com/margaridaCF/FlyingOctomap

to recreate the setup and it is being adopted in robotics: in the ROS build farms, robotics companies [48] and even the ROS-Industrial Consortia [49].

An additional example of a technique borrowed from software development is automated testing. To verify that the implementation achieves the intended results under all conditions is a non-trivial task. Nowadays frameworks exist to assist in this examination and are commonly used in software development. It is such a pivotal aspect of development that one methodology consists of creating tests that show a case where the implementation fails and then proceeding to improve the implementation until that test no longer fails. This methodology is called test-driven development. Due to the great number of such tests needed to achieve a complex program they can be automatedly run. Using automated verification, verifying the impact of new development on existing functions becomes trivial. In this context, an edge case is a set of extreme inputs that make a test fail and a scenario is the set of values that compose the pre and post conditions.

The instruments used in our implementation are described in the following.

4.1.1 Variables that compose a scenario

The Lazy Theta* algorithm has as input variables the starting position, the goal position, the world representation, and the number of maximum iterations before declaring a path unsolvable. The output will be the sequence of waypoints selected to arrive from starting to goal positions.

Some restrictions are enforced over the pre and post conditions. As a precondition to applying Lazy Theta*, all considered positions must be within the space contemplated in the world representation. A requirement for the postcondition is that all waypoints must be in free known space.

The starting and goal positions are straightforward to recreate as they are sets of x, y and z coordinates. However, the world representation poses some problems. The variability involved in generating an octree is tremendous. Each composition of voxel size and quantity has the potential to be a surprising edge case. Another issue is that it is challenging to set up a well-identified edge case. It would entail manipulating precisely the right sequence of sensed point clouds to produce the desired octree. Variability was removed by generating the octree once and then working over a saved, fixed representation. As an additional benefit, it also reduced the time needed to execute a test.

4.1.2 Automate the discovery of edge cases

Another aspect to consider is the verification of edge cases for the different modules of the code.

To enable the analysis of significant amounts of scenarios, the verification of input and output conditions must be automated. Leaping from accessing the specific values for illustrative scenarios to accessing the whole range of values in a scenario type.

One such type of scenarios is finding a path between two points with a line of sight, which are both in the known space and for which the interval space is also known. The range of the input values can be logically asserted as well as the characteristics of the resulting path, as it can be seen in 4.1.2.

Equipped with these conditions, the whole octree can be searched to find eligible scenarios. Any case where the preconditions did not generate the expected postconditions is considered an error. In the developing phase, this provided the means to search for edge

cases. After their discovery and in accordance with the test-driven development methodology, they were documented into individual tests. Always in the list of tests used to verify correctness. Unfortunately, a similar set of pre and post conditions could not be identified to assess the quality of the generated path while avoiding obstacles.

The ROS integration of Google's C++ unit testing framework gtest [50] is the tool used for automated unit testing.

Algorithm 2 Pre and post conditions for a solution to be considered correct for the simple use case.

Start is the starting position.
Goal is the final position.
World is set of all voxels contained in the octree.
v represents an analyzed voxel.
$\| \overline{StartGoal} \|$ is fixed for each test set.
resultingPath set of all waypoints that compose the solution

Require:
$Start \in World_{Free}$
$Goal \in World_{Free}$
$\overline{StartGoal} \in World_{FreeAndKnown}$
$\forall v, v_{neighbors} \in World_{Known}$

Ensure:
$resultingPath$.size == 2
$\| \overline{resultingPath[2]toVoxelCenter(goal)} \| < resultingPath[2]$.size
$\| \overline{resultingPath[1]toVoxelCenter(start)} \| < resultingPath[1]$.size

4.1.3 Automated assertion of correctness

The amount of test cases and operations involved in the algorithm rapidly generated a huge amount of tests. This is the main reason that motivated the adoption of test-driven programming as the development process of the implementation, hence automating the tests.

For each foreseeable use case, there is a suite of unit tests. Each test considers start conditions in a homogenous way supported by the well-defined input elements that define a scenario.

While testing the code in simulation, inputting commands to the UAS quickly proved inefficient. With scripted starting conditions and a manual input of the commands, there was too much variability. An option was to use scripted commands however it still required too much time to complete the tests. Instead, all the components of the scenario are numerically defined for each test case.

4.1.4 Direct pseudo-code to code translation

Frugality is a recognized challenge as it can be seen in the implementation related in [3] where both locally implemented regular grids and the raw point cloud as supplied in the PCL library were needed.

The code was developed as a direct, unmodified (except in heuristics as seen in the section 4.2.4) implementation of Lazy Theta* as found on [35]. To efficiently verify this claim, each line of the pseudocode is included as a comment above its implementation. The implementation is done to encapsulate technical details and highlight the logic. The precise

correlation between code and pseudo code proved crucial in the later stages of implementation to verify its correctness.

4.1.5 Statistical analysis of results

The same automation used to look for edge cases can be used later to analyze the performance of the implementation statistically. This feature was identified in [3] as bringing substantial insight.

As a follow-up of an implementation that relies on regular grids, it is interesting to understand how the sparse grid correlates to a regular grid regarding number of iterations. The results are detailed in section 5.2.3.

4.2 Adapting Lazy Theta* to Octomap

As it was identified in [7], the primitives on which the algorithms are based have a determinant role in the performance of the final solution. This primitives are the direct connection between the algorithm and the environment representation. Critical primitives in Lazy Theta* related in [35] are: uniquely identifying a voxel, finding a voxel's size and finding a voxel's neighbors.

Even though one of the goals pursued in this research is to lighten the memory footprint, one challenge to apply this algorithm to the octree was precisely its lightness. Each node has very little information: only a pointer to an array of higher resolution voxels that form the same volume and an identifier for within a tree level. In the octomap implementation, a node has the highest resolution for its location when its pointer to children is *NULL*.

The tree depth of an octree is fixed from its creation. As the authors of the octomap note at [9], a maximum depth of 16 is sufficient to cover a cube with a volume of 655.36 m^3 at one centimeter resolution. For this implementation, a resolution of 20 cm was considered, making a maximum depth of 16 sufficient for a volume of 1310720 m^3.

4.2.1 Voxel Identification

Coordinates keys cannot uniquely identify each voxel of the octree since coordinates overlap. Each voxel is contained by a bigger one, except for the root node. The identifier with a level is not globally unique either. Apart from the node state, there is no other information associated with each node.

The solution was to use the coordinates of the center of the voxel associated with the voxel size. This creates a composite key[4] that is unique accross the tree.

4.2.2 Voxel Size

Both associated to the voxel identification and finding neighbors, voxel size is crucial to deduct the size of the highest resolution voxel containing a given random point.

Given a coordinate point, finding the size of the associated leaf node has a complexity of $\mathcal{O}(tree\_depth)$. Starting from the root of the tree (which is the node of minimum resolution

[4] A concept borrowed from relational databases. A composite key is a combination of two or more characteristics of an instance that can be used to uniquely identify it. However, when they are taken individually do not guarantee uniqueness.

/ maximum volume), each tree level must be transversed analyzing the subsequently smaller size voxels that contain the target point until a leaf node is found.

4.2.3 Voxel Neighbours

The sparseness of the tree makes finding a voxel's neighbors nontrivial. In [3] the approach was to create a local regular grid and discard the size of the analyzed voxel. In practice, it as if a local potential field was created with the goal as an attractor and the voxel dimensions as boundaries. Only while analyzing positions within the same voxel, the result obtained was the same as by finding the neighboring voxel of the octomap through steepest descent (of the distance to the goal).

To take advantage of the organization in the octree and jump directly to the next neighbors, all possible combinations between the voxels' size and its neighbor's size must be contemplated. Voxels with large neighbors will have a smaller set of neighbors. Voxels surrounded by voxels of maximum resolution will have more neighbors.

As in frontier cells exploration, diagonal neighbors are not contemplated. A voxel of maximum resolution has six neighbors. While a voxel twice the maximum resolution has a maximum of twenty-four neighbors.

To compile the list of neighbors of a random point the following steps are followed:

1. Find unique identifier within tree level
2. Find node level in octree
3. Find voxel size
4. Calculate the coordinates of the center of each neighbor assuming all neighbors have the maximum resolution. The center coordinates of a neighbor will only be added once. This verification has complexity of $\mathcal{O}((node\_size/resolution)^2)$.

4.2.4 Heuristic adaptation to the sparse grid

In [35], the voxel's vertices are the intended candidates for waypoints. However, for the reasons mentioned before, the voxel center's are the coordinates being used as waypoint candidates in this implementation.

The heuristics to manage the list of candidate waypoints (or the open list) is ordered by $g(s)+h(s)$ with the following definitions:

> For every vertex s the g-value $g(s)$ is the length of the shortest path from s_{start} to s found so far. [35, p. 3]

> We use the straight line distances $c(s, s_{goal})$ as h-values in our experiments. [35, p. 5]

On one hand, the original assumption of a regular grid in [35] no longer holds. On the other hand, instead of having many points as waypoint candidates for each voxel (each vertex), now there is only one (the center). From here follows that the distance between the goal point and its voxel center can be significant. This offset will impact the efficiency of the heuristics if left unmodified.

In order to illustrate this case the scenario depicted in Fig. 2 will be considered [5]. The start point is to the left and the goal is to the right, both at the same level. The path should be

[5] octree offShoreOil_1m.bt (available at https://github.com/margaridaCF/dataStructureAnalysis), with start point (-11.3m, -4.9m, 0.5m) and goal point (-1.3m, -4.9m, 0.5m)

	Room	Corridor	Z shape	2 pillars	4 pillars	Pillars
Width (m)	24	24	22	77.8	180.4	13
Length (m)	8	14	13.8	77.2	90	10

Table 1 Size of the scenarios in Fig. 4 used in the simulations.

along the x-axis. This edge case can arise in situations where the goal position is contained by a voxel that has large voxels in its neighborhood. Observing the distance between the goal (highlighted in a) and the center of the voxel that will be the last neighbor explored before reaching the goal (highlighted in b), it is clear that distance between these two points is larger than the resolution of the octree. This throws off the heuristic because many smaller neighbors will be found that have a smaller distance to the goal but are not in the direction of the goal. What happens in this run is the following. At iteration 220, the final voxel is found. However, as it is a large voxel, it will be left near the bottom of the priority queue of nodes to evaluate (at position 173) and it will take many iterations before the algorithm studies this particular option. This is as noticeable as the offset between the goal point and its voxel center.

In order to address this issue, the heuristic function was changed from the continuous function exposed before to a piecewise function with 0 as the result for the voxel that contains the goal point. While this might lead to slightly suboptimal paths, it was considered a fair trade-off.

5 Results and analysis

The results and their analysis for frontier cells exploration and path planning with Lazy Theta* are described in the following.

5.1 Frontier Cells Exploration

The exploration algorithm of frontier cells was first tested in a simulated environment and later with octrees constructed by an UAS in an experimental setting.

5.1.1 Simulation Environment

Five datasets were used to run the tests in simulation. They were all generated using ROS nodes on a Gazebo simulator. The sensor used emulates a VLP-16 LIDAR: it is omnidirectional, dividing the 360 degrees into 1500 samples and stacking it 16 times. The sensor was mounted on the front of the 3D model of a quadrotor (see Fig. 3).

Five simulated worlds have been considered. They are shown in Fig. 4, where the images represent a 2D cut of the world by fixing the z-axis value. Three of them are the same as the ones used in [51]: an enclosed space with only one opening[6] (room), a circular corridor with one opening to the inner area and a Z shaped corridor with openings at both ends. In these scenarios the UAS fully explores the enclosed area, never crossing an opening. Each of the openings is 0.8 meters long. The dimensions of the simulated worlds are detailed in Table 5.1.1. For a scale closer to the targeted application, there is a scenario that emulates

[6] Opening is used in the sense of a break in a continuous wall.

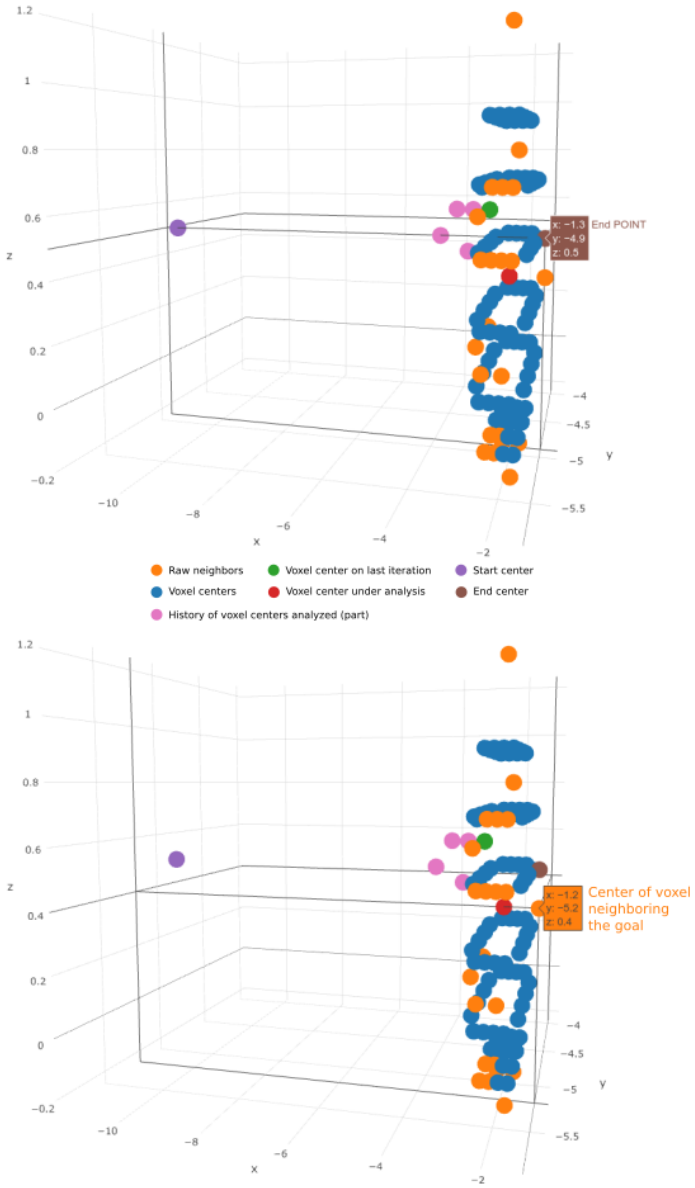


Fig. 2 Illustration of finding a path along the *x*-axis. All axis are in meters. The start point is to the left and the goal is to the right, both at the same level. The path should be along the *x*-axis. Observing the distance between the goal (highlighted in a) and the center of the voxel that will be the last neighbor explored before reaching the goal (highlighted in b) it is clear that this distance is more considerable than the resolution of the octree. The figure also depicts the centers of the voxels involved in evaluating each neighbor. And again it is clear the different amount of neighbors in a neighborhood with large voxels and with maximum resolution cells. (a) Highlighting the goal point. (b) Highlighting the voxel center of the voxel that includes the goal point.

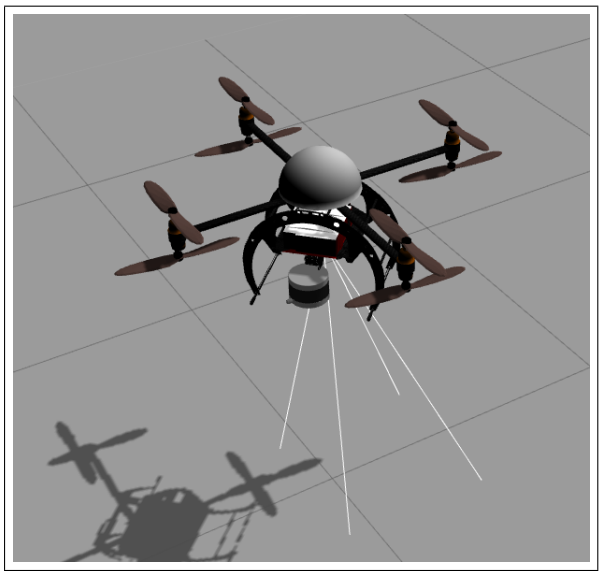


Fig. 3 The UAS model used in simulation with its VLP-16 LIDAR in front.

the lower part of an offshore petro-station. In this scenario, the ground is representing the sea level and the floor of the station is above the UAS - too far to be sensed. From these worlds two datasets were extracted, one where the UAS inspects two pillars and the second one where the UAS inspects three pillars. The comparison of two information states in the same world will bring insight into the effect of unknown space in the identification of frontier cells.

5.1.2 Analysis of the Results in the Simulation Environment

In all the scenarios, both in 2D and 3D, the frontier regions are roughly the same. This result is illustrated in Fig. 5. In zoom A, with the regular grid, all the voxels qualifying for frontier voxels are identifies in black, generating lines around the free space. In zoom B this line becomes dotted. The same positions are classified as frontier voxels, but they are now of variable size. When the image is analyzed taking into consideration both the voxel center and its corresponding volume it is possible to see that the identified regions are the same.

The same regions are detected, however the number of cells needed to represent them is smaller. This difference tends to be small using a LIDAR as the edge of the sensor is often times irregular due to the increased angular interval between rays. However, when in large scenarios (as two pillars or three pillars), it starts to gain more relevance. In addition, the geometrical disposition of the voxels and how neighbors are computed explains this result. When a larger explored voxel is adjacent to the same unknown voxel, its whole volume is considered a frontier. However, in a regular grid, the added volume is always the size of the grid resolution. Computed results are shown in Fig. 2.

For each scenario four combinations are run: a regular grid in 2D, a regular grid in 3D, a sparse grid in 2D and a sparse grid in 3D. Only the portion of the world above the ground was considered, more accurately between zero and one meter altitude. This interval was chosen for study to make the comparisons with the 2D space within the same magnitude.

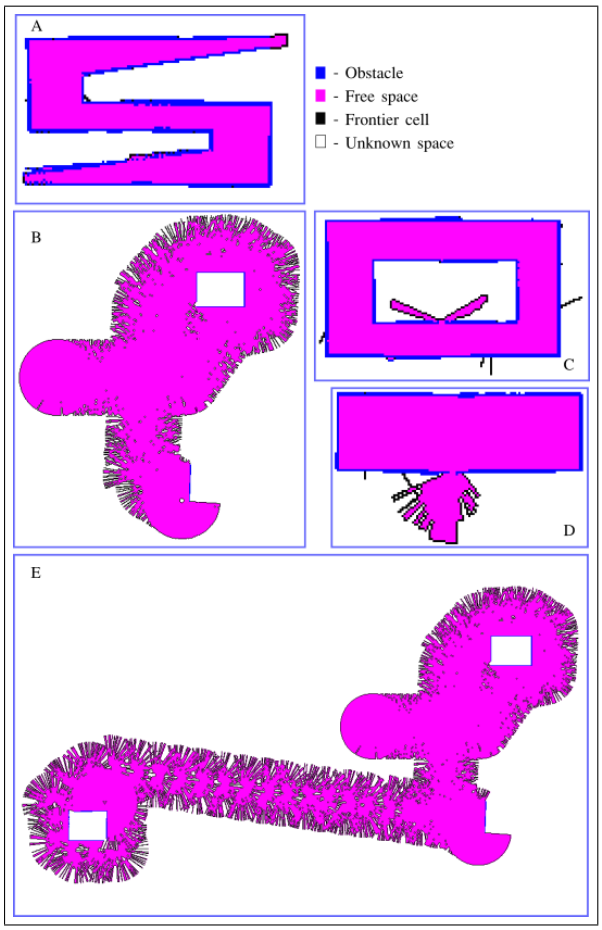
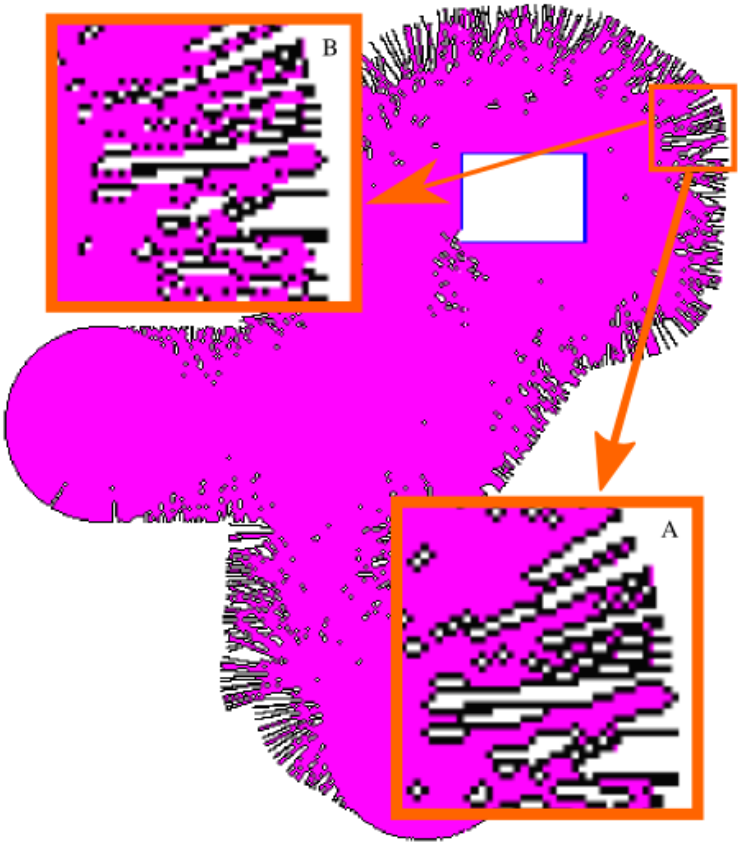


Fig. 4 The different used scenarios in a 2D projection of the octree. The images present a 2D cut of the world by fixing the z-axis value. The projection is generated by extracting the state of the voxel at a constant altitude. It is shown the complete area that the octree could map (in the x and y axes). The worlds (and respective abbreviations) are the following: (A) Z corridor, Z. (B) Offshore petro-oil structure, two pillars sensed. 2 (C) Circular corridor, C. (D) Room, R. (E) Same structure one additional pillar sensed, 3.

Table 2 Frontier representation each simulation dataset: room, R; circular corridor, C; Z corridor, Z; offshore petro-oil structure, two pillars sensed, 2; same structure one additional pillar sensed, 3. The units of space are m^2 and m^3 for two and three dimensions respectively.

		2D			3D		
	Structure	Regular	Sparse	Difference	Regular	Sparse	Difference
R	Frontier cells	236.0	221.0	15.0	50,903.0	47,460.0	3,443.0
	Space	9.4	9.7	-0.2	407.1	407.0	0.1
C	Frontier cells	137.0	110.0	27.0	5,729.0	1,951.0	3,778.0
	Space	5.5	5.8	-0.4	45.8	22.8	23.1
Z	Frontier cells	67.0	61.0	6.0	24,210.0	20,873.0	3,337.0
	Space	2.7	2.9	-0.2	193.7	185.1	8.5
2	Frontier cells	9,000.0	6,809.0	2,191.0	845,091.0	790,015.0	55,076.0
	Space	360.0	386.2	-26.2	6,718.9	7,028.2	-309.3
3	Frontier cells	28,627.0	21,724.0	6,903.0	1,273,737.0	1,156,968.0	116,769.0
	Space	1,145.0	1,165.6	-20.7	10,067.7	11,047.3	-979.6



(a)

Fig. 5 The frontier cells found in the two pillar scenario using the sparse grid data structure. (a) Shows the results using the regular grid. (b) Shows the results using the sparse grid.

By comparing the execution time of each run depicted in Fig. 6, it is clear that the sparse grid needs less time to find the frontier cells.

To analyze the computational load for compiling the list of frontiers positions let us look into the iteration count and the amount of space analyzed by the frontier algorithm. In an online architecture, these calculations would be performed every time significant alterations are done to the world representation. The number of iterations needed to process the whole structure is always lower in a sparse grid. More specifically, the number of iterations is one order of magnitude higher for regular grids in the 2D case and two orders of magnitude higher in the 3D case, as it can be seen in Fig. 7. The linear growth on a logarithmic scale shows the exponential progression of the iterations amount. The granularity afforded by the hierarchical nature of the structure indeed predisposes this sequence since the worst case scenario of the sparse grid is a regular grid. However, the disparity of the results can not be explained only by its hierarchical nature.

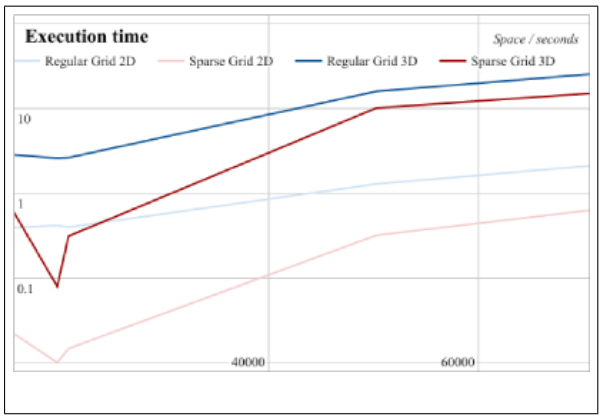


Fig. 6 Analysis of the influence of the search space explosion on the execution time. Each scenario appears first as 2D and then as 3D, using a logarithmic scale for both axis. The units of space are m^2 and m^3 for two and three dimensions respectively.

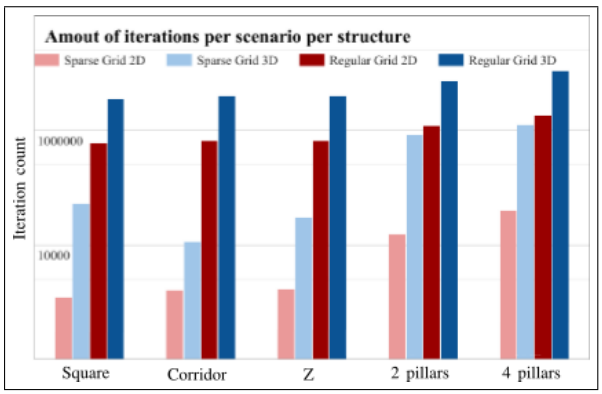


Fig. 7 A comparison of the iterations amount needed to transverse the world representation for each of the data structures, both in two dimensions and three dimensions using a logarithmic scale for the space axis. The state of the grid representing the world is drawn from the datasets used for testing.

Another factor to be taken into account is the number of unknown cells. As it can be seen in Fig. 8, the unknown space is what composes the vast majority of the surveyed area at that moment, opening the possibility of restricting the analyzed cells to the known cells.

The size of the space represented affects the number of iterations and therefore the execution time. The results of the scenarios with two and four pillars reflect it. Here the world is exactly the same - only the amount of information changes. However, in the scenarios Room, Corridor and Z, the increase in space does not correlate to the iterations amount or execution time. As the world configuration in each scenario changes drastically, the distribution of the sizes of the voxels generation for its representation changes with it.

5.1.3 Experimental Data

The algorithms have been applied to a dataset captured in a flight done by the UAS shown in Fig. 9, although in a more confined space. The used autopillot was a Pixhawk (first version).

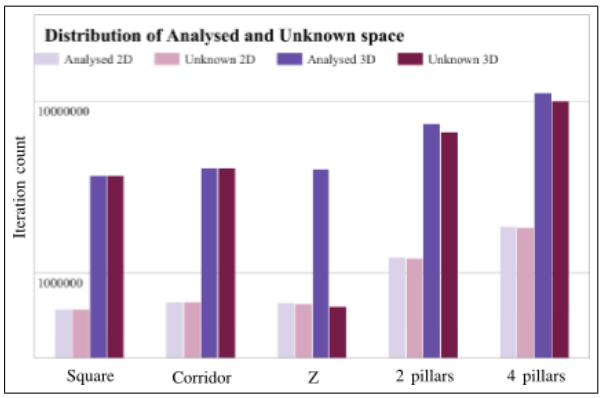


Fig. 8 The amount of space between unknown space during exploration is very large, in any scenario, in any dimension. This graph is using a logarithmic scale for the space axis. The state of the grid representing the world is drawn from the datasets used for testing. The units of space are m^2 and m^3 for two and three dimensions respectively.

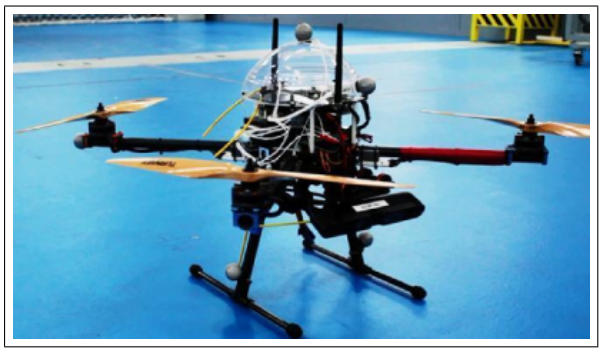


Fig. 9 UAS platform used for data acquisition. It was equipped with two RGB-D cameras Asus Xtion Pro Live, one facing forwards and another facing backwards.

Lazy Theta* was run on an Intel NUC with ROS indigo installed. Positioning was supplied by a Vicon system installed in the testbed. It was equipped with two RGB-D cameras Asus Xtion Pro Live, one facing forwards and another facing backwards. The images captured by the camera were then combined and analyzed to generate the point cloud. This data was recorded in an indoor testbed with dimensions 15x15x5 meters at the Center for Advanced Aerospace Technologies (CATEC) located in Seville (Spain). Fig. 10 shows the point cloud captured in the testbed.

5.1.4 Analysis of the Results with Experimental Data

The dataset captured in the real scenario was analyzed using the same source code (both 2D and 3D in each data structure). Again only the portion of the world above the ground between zero and one meter altitude was considered. The results are consistent with the ones observed with datasets captured in simulation. Concerning the execution time, the octree keeps performing significantly better as it can be seen in Fig. 11. Another measure of computational load brought in by frontier search is the number of voxels inspected at each

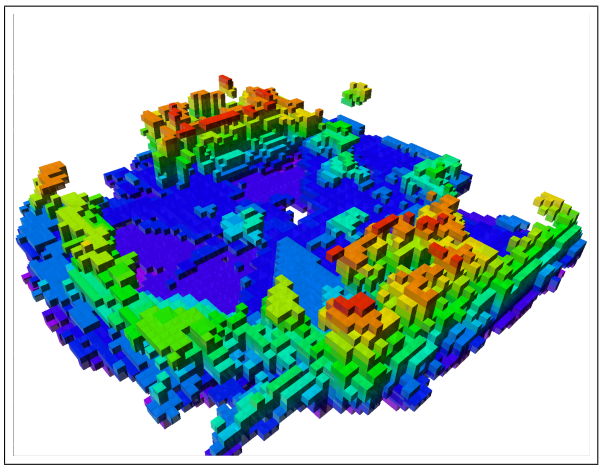


Fig. 10 Point cloud captured in an indoor testbed with the UAS shown in Fig. 9 at the Center for Advanced Aerospace Technologies (CATEC) located in Seville (Spain).

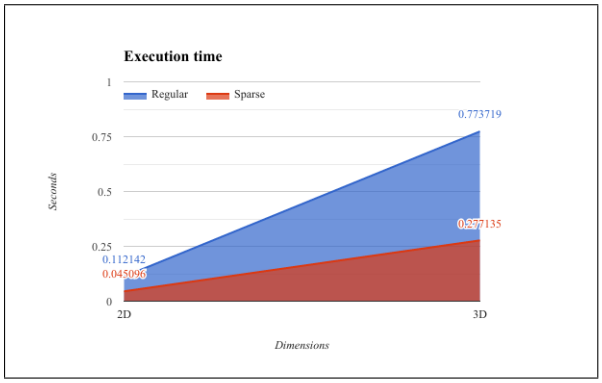


Fig. 11 Visual comparison of execution time between data structures and dimensions for the experimental dataset. This graphical portray of the data illustrates the magnitude of difference in execution times between the regular grid and the sparse grid. In the graph a logarithmic scale is used for the time axis.

sweep of the world representation. In Fig. 12 again it can be seen the high proportion of unknown cells which corroborates the explanation of the efficiency gain by the amount of skipped cells.

Another interesting result refers to the frontier space and the amount of cells needed to represent it. In Table 3, although the frontier space represented by the octree deviates by around 1.3 (square and cubic meters respectively), it is reduced the number of cells that represent it.

It is hypothesized that this is not a variation that will greatly impact the further processing of the frontier cells for selecting a goal, but it is a reduction nonetheless.

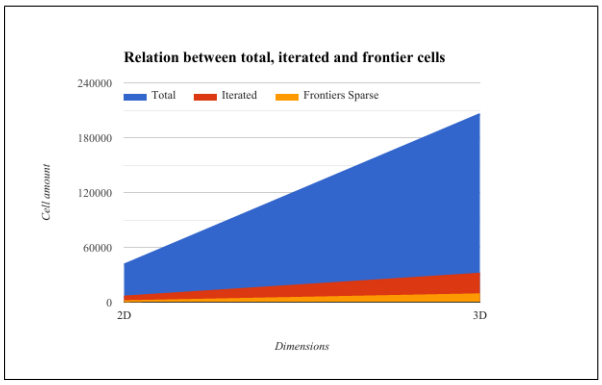


Fig. 12 Relation between the total amount of cells, the cells iterated through with the sparse data structure and a number of frontier cells found. The number of unknown cells can be extrapolated as the blue area since the values are not stacked. In the graph a logarithmic scale is used for the vertical axis.

Table 3 Frontier representation for the experimental dataset. The units of space are m^2 and m^3 for two and three dimensions respectively.

	2D			3D		
	Regular	Sparse	Change	Regular	Sparse	Change
Cells	1341	1067	274	10394	8652	1742
Space	13.4	12.0	1.4	10.4	11.7	-1.3

5.2 Lazy Theta* Results

This section describes the datasets used for the tests, the common points between both algorithms, a statistical data from the simple use case and the results for obstacle avoidance path planning.

5.2.1 Datasets

In the interest of standardization, the datasets used for testing this implementation of Lazy Theta* were collected from the same worlds as the ones used for testing the frontier cells algorithm. Two datasets are here analyzed: one collected from a simulated world and one collected in an experiment.

In fact, the dataset generated from simulation is one of the five used with the frontier cells algorithms. For a greater volume of explored space, the inspection of 3 pillars in a world that emulates the lower part of an offshore petro-station was selected. For more details, refer to subsection 5.1.1 where the dimensions of the world are included in Table 5.1.1.

Regarding the experimental data, a new dataset is used. It was generated in the same testbed as the previous experimental dataset and with the same platform and sensors (see Fig. 9). There is one difference however, the dataset was collected after a longer flight and a larger amount of known space. This flight was one of the three in the evaluation of phase two of the EUROC competition [45].

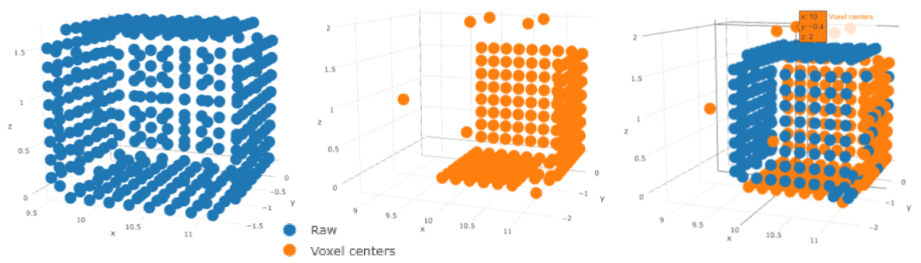


Fig. 13 Spacial representation of the center of the neighboring voxels generated at an iteration of the Lazy Theta*. This amount varies according to the size of the voxel analyzed. (a) Preliminary generation assumes all neighbors have the minimum size. (b) This is then updated to the actual size of the neighbor. (c) Both sets of neighbors juxtaposed.

5.2.2 Neighbors

In Lazy Theta*, for each analyzed point, its neighbors are needed twice: first to generate waypoint candidates and then for the final validity update before adding to the group of inspected points. The process of generating these neighbors is identical to the one used in the frontier cells algorithm. The diagonal neighbors are also discarded and now the voxel grouping is exploited.

Fig. 13 shows the reduction of neighbors by comparing the naive generation of a regular grid and having just one point per voxel. In this case, the number of neighbors is reduced. This is especially obvious on the left and front sides of the voxel. In each case, the number of neighbors is reduced from 64 to 1. This reduction is naturally dependent on the composition of the octree: the upper side changes from 64 to 4 and the bottom from 64 to 61.

An intuition emerges: the type of the terrain present in the scenario will dictate how much this situation will occur. If solid large obstacles are present (like a wall for example) it will happen more. In settings with small, detailed objects or confined spaces it will happen less.

5.2.3 Statistical data from free paths

The simple use case of finding the path between two points in free known space is used to gather information about the number of iterations needed to find a path. It is possible to do this in an automated way using the pre and post conditions described in Algorithm 4.1.2. After finding and correcting all the cases that did not pass this test, it can be used for performance analysis. It is possible to assess the performance change between the regular grid and the octree by comparing how many iterations are used to find a solution with the octree and with the regular grid.

The baseline is defined as the number of iterations needed using a regular grid with perfect heuristics. In other words, the number of cells evaluated if all the world required maximum resolution voxels and the heuristic always surfaces the next waypoint of the correct solution.

First the results are presented in Fig. 14. For all the runs, each amount of iterations used in a run is signaled below as a dash. The bars above show the probability of each one happening. This view presents both the clustering among the result events and the repetition of each result. In both cases the most frequent amount is a little over one-fifth of the baseline.

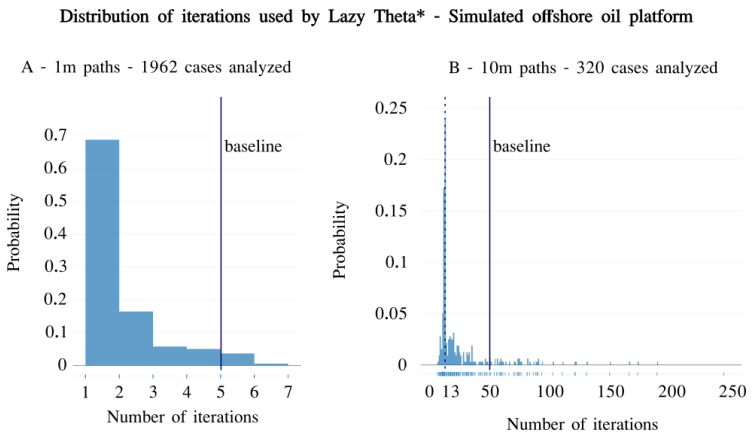


Fig. 14 The probability that a particular amount of iterations is needed to find a path in free space, for each of the amounts. These results were collected while searching the same octree, for two different path lengths. The whole space is scanned for suitable points. Each start and end point tuple is given to Lazy Theta* to find the straight line path between the two points. Each case consists of a variation of start and/or goal. A - For one-meter lines. B - For ten-meter lines.

However, there is a much higher probability of that being exactly the number of iterations in paths 1 meter long.

A more practical insight is the probability of the new implementation outperforming the baseline, with any number of iterations. This observation is what is portrayed in Fig. 15. For a more detailed overview, the runs were grouped into the following groups:

- Use half the iterations of the baseline or less;
- Use between half the iterations of the baseline, up to the same amount;
- Use exactly the same amount of iterations as the baseline;
- Use over same iterations as the baseline up to double that amount;
- Use more than double the number of iterations as the baseline.

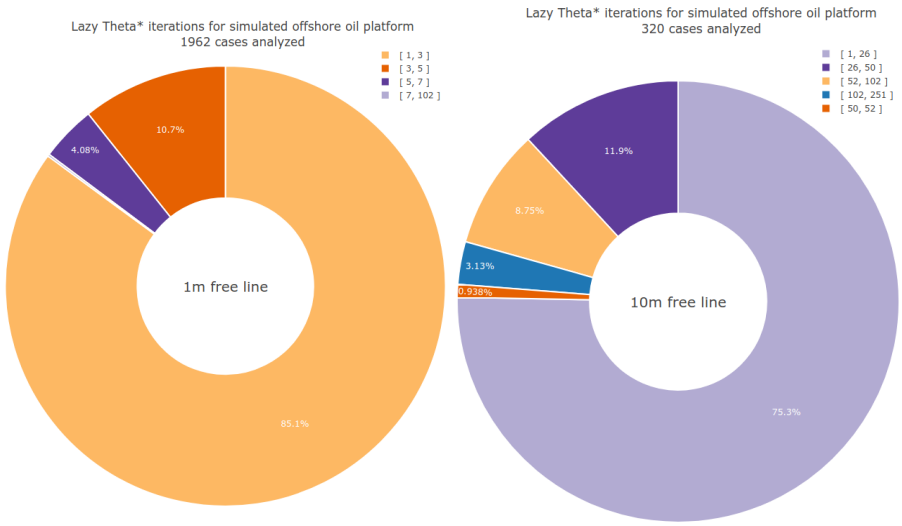
Although for a path of 10 meters the probability of using precisely 13 iterations (the most frequent amount) is as low as 24%, by combining close occurrences it gets much closer to the values found for the 1-meter paths. In both situations, in around 75% of the cases, less than half the iterations are needed.

5.2.4 Obstacle avoidance

To test the ability to generate paths that avoid obstacles, the experimental dataset used is described in Sect. 5.2.1.

Fig. 16 shows an example of a path calculated by the implementation of the Lazy Theta* over the octree data structure. As it can be seen, the resulting path avoids the many obstacles. The scenario is composed by the aforementioned dataset, a start point and a goal that are 10 meters apart but have several obstacles between them.

The resulting path (in green) does not cross any obstacles. On the other hand, the start and end points are not the same in the input points (in blue) and in the resulting path. A less obvious feature of the resulting path is that the distance kept from the obstacles is variable.



(a) (b)

Fig. 15 Amount of iterations used in the same octree, for two different path lengths for different compositions of voxel space. Each case consists on a variation of start and/or goal. (a) For one meter lines. (b) For ten meter lines.

The offset between the start and end of each line reflects that each point is identified by the center of the voxel it is contained by. Another consequence is the irregular distance the path maintains from the obstacles. At the first obstacle, the waypoint is very close while at the second obstacle the chosen waypoint is much further away, creating a steep dive. At the first obstacle, the free voxel is small while in the second obstacle it is much larger. As both are part of the free space, this is not directly observable in the figure but is verified numerically. The effect is observable in the voxels representing the occupied space. As each cube represents the center of the voxels, it often creates the illusion of holes in this type of visualization. Examples are the holes in the ground to the left of the path and the floating blocks in the middle of the solid shape in line with the first obstacle.

6 Conclusions and Future Work

In this paper, the impact of the underlying data structure on processing the world representation has been analyzed both to find frontier cells for exploration and to generate paths that avoid obstacles. The focused data structure is the implementation of an octree done by the octomap framework.

In the context of the frontier cells, some characteristics emerge as beneficial for efficiency. The baseline used was a regular grid with the same resolution as the octree. The number of iterations needed to find frontier cells was smaller in all cases by, at least, one order of magnitude. Grouping regions with the same status and skipping the unknown cells explain these results. The number of frontier cells is smaller for the sparse grid in all of the datasets while covering a similar amount of space. This fact indicates that these cells better

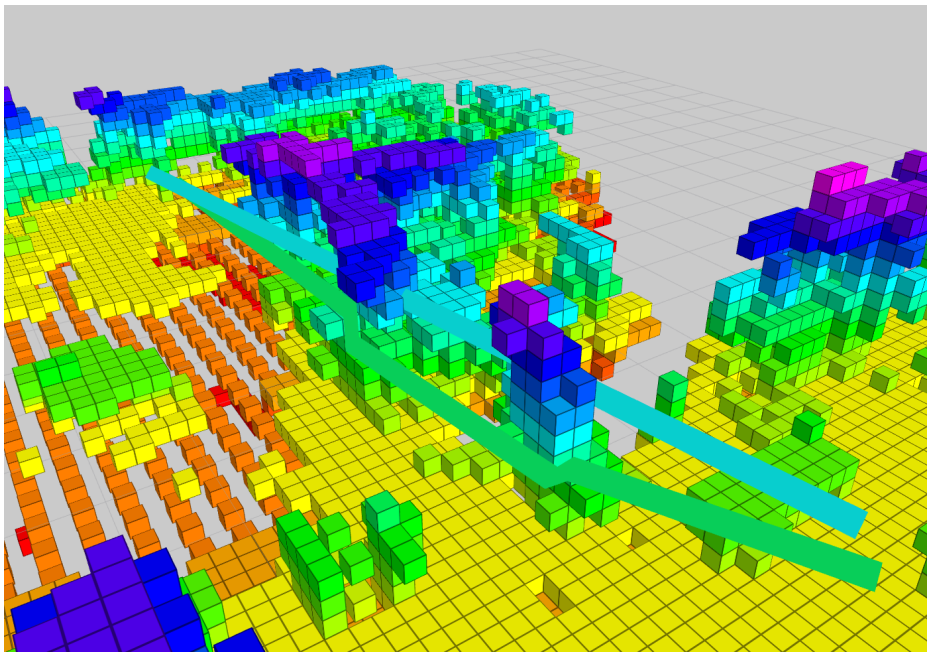


Fig. 16 Example of a path avoiding obstacles generated using the Lazy Theta* implementation. The octree used was collected by a multirotor during the evaluation of the second stage of the EUROC competition.

summarize the frontier regions, providing some degree of clustering. Although it will be beneficial for any cell processing, the difference is not large enough to allow to conclude it will create a sizable impact.

A possibility emerged from the combination of the results of both algorithms: the type of terrain dictates the level of sparseness of the grid. Although this empirically seems apparent, a structured classification of the terrain type would provide software developers with a valuable tool when designing a system. Should the intuition be proven, it would formally validate the choice of sparse grids for large unknown and concentrated environments. In other words, settings where the obstacles are large and uninterrupted, like a pillar as opposed to a tree. Offshore environments especially fit this description as they seldom contain vegetation. On land, the ground is more uniform than on land (no plants) although it is in constant movement. The structures that compose the initially targeted use case are also exceptionally well suited to this description due to the frequent occurrence of walls, pillars, and other solid objects.

Due to the choice of using the center of the cell as a waypoint, the sparse grid has the fringe benefit of acting as a Voronoi map up to some degree. The path is always at least half the resolution of the octree away from the obstacles.

The combination of test-driven development and automated proof of correctness provided a valuable tool towards stable and reliable code. The automated tests provided means to gather statistical data about the generated path, which is a feature mentioned in [3] as relevant for parameter finding. In this study, it is provided an overview of the improvements achieved as opposed to the baseline. There was a reduction in the number of iterations needed to find the solution in 75% of the cases. This decrease reflects a substantial

amount of different configurations of the underlying octree that leads to a reduction of the necessary computational power. Here the baseline is equivalent to how many voxels would be evaluated if all the space required maximum resolution voxels and the heuristic always surfaced the next waypoint of the correct solution. These results are explained by the ability to evaluate several (maximum resolution and same state) voxels at a time. This capability avoids the evaluation of each one individually.

The effort to encapsulate technical details in higher level structures made possible a semantic level of inspection of the code. This insight allowed to identify some initial misinterpretations and enabled the certification that the algorithm was indeed implemented as intended.

Several different future directions of research are of interest. On one hand, it would be of great benefit to integrate more data structures and to add more implementations of the octree such as the PCL cloud. On the other hand, it would be valuable to compare entirely different data structures. One type of data structure of particular interest is a triangle mesh extracted with Delaunay triangulation. Again several implementations exist (CGAL, PCL), and comparing them would bring greater insight into their application. Another research direction is to evaluate the performance of the search algorithm throughout a mission. Such survey would permit a thorough comparison between increasing amounts of information in the same scenario while using each different data structure. Finally, our mid-term objective is to apply these results to UAS exploring extensive areas in real-time. With the combination of exploration and path planning, the UAS would be able to detect when all reachable point cloud information had been sampled and autonomously deliberate to stop.

References

1. Armin Hornung et al. Octomap. [Online; accessed 28 sep 2017].
2. M. Faria, I. Maza, and A. Viguria. Analysis of data structures and exploration techniques applied to large 3D marine structures using UAS. In *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1277–1284, June 2017.
3. Francisco J Perez-grau, Fernando Caballero, Ricardo Ragel, Antidio Viguria, and Anibal Ollero. An Architecture for Robust UAV Navigation in GPS-denied Areas. *Journal of Field Robotics (JFR), Special Issue on High Speed Vision-Based Autonomous UAVs*, 2017.
4. Samuel Hornus, Olivier Devillers, and Clément Jamin. dD triangulations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.9 edition, 2016.
5. Flemming Schøler, Morten Bisgaard, and Anders Cour-harbo. Generating Configuration Spaces and Visibility Graphs from a Geometric Workspace for UAV Path Planning. *Autonomous Robots*, pages 1–14, 2013.
6. Rosli Omar and Rosli Omar. PATH PLANNING FOR UNMANNED AERIAL VEHICLES USING VISIBILITY LINE-BASED METHODS Thesis submitted for the degree of Doctor of Philosophy at the University of Leicester by Department of Engineering VISIBILITY LINE-BASED METHOD. (March), 2011.
7. Steven M. LaValle. Planning algorithms. *Planning Algorithms*, 9780521862059:1–826, 2006.
8. Alexander Greß and Reinhard Klein. Efficient representation and extraction of 2-manifold isosurfaces using kd-trees. *Graphical Models*, 66(6):370–397, 2004.
9. Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, 2013.
10. Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.
11. Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. Intelligent robotics and autonomous agents. The MIT Press, Cambridge (Mass.) (London), 2005.
12. Brian Yamauchi. A frontier-based approach for autonomous exploration. In *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97. 'Towards New Computational Principles for Robotics and Automation'*, pages 146–151. IEEE Comput. Soc. Press, 1997.

13. Friedrich Fraundorfer, Lionel Heng, Dominik Honegger, Gim Hee Lee, Lorenz Meier, Petri Tanskanen, and Marc Pollefeys. Vision-based autonomous mapping and exploration using a quadrotor MAV. *IEEE International Conference on Intelligent Robots and Systems*, pages 4557–4564, 2012.
14. Leonardo Romero, Eduardo Morales, and Enrique Sucar. A Robust Exploration and Navigation Approach for Indoor Mobile Robots Merging Local and Global Strategies. *Advances in Artificial Intelligence: International Joint Conference 7th Ibero-American Conference on AI 15th Brazilian Symposium on AI IBERAMIA-SBIA 2000 Atibaia, SP, Brazil, November 19–22, 2000 Proceedings*, pages 389–398, 2000.
15. Christian Dornhege and Alexander Kleiner. A frontier-void-based approach for autonomous exploration in 3d. In *2011 IEEE International Symposium on Safety, Security, and Rescue Robotics*, pages 351–356. IEEE, nov 2011.
16. Gavin Paul, Stephen Webb, Dikai Liu, and Gamini Dissanayake. Autonomous robot manipulator-based exploration and mapping system for bridge maintenance. *Robotics and Autonomous Systems*, 59(7-8):543–554, 2011.
17. Christian Potthast and Gaurav S. Sukhatme. A probabilistic framework for next best view estimation in a cluttered environment. *Journal of Visual Communication and Image Representation*, 25(1):148–164, 2014.
18. Wolfram Burgard, Mark Moors, Cyrill Stachniss, and Frank E. Schneider. Coordinated multi-robot exploration. *IEEE Transactions on Robotics*, 21(3):376–386, 2005.
19. Howie Choset and Keiji Nagatani. Topological simultaneous localization and mapping (SLAM): Toward exact localization without explicit localization. *IEEE Transactions on Robotics and Automation*, 17(2):125–137, 2001.
20. Luigi Freda, Giuseppe Oriolo, and Francesco Vecchioli. Sensor-based exploration for general robotic systems. *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS*, pages 2157–2164, 2008.
21. Shaojie Shen, Nathan Michael, and Vijay Kumar. Autonomous multi-floor indoor navigation with a computationally constrained MAV. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 20–25, 2011.
22. J.C. Latombe. *Robot Motion Planning*. The Springer International Series in Engineering and Computer Science. Springer US, 1991.
23. H.M. Choset. *Principles of Robot Motion: Theory, Algorithms, and Implementation*. A Bradford book. Prentice Hall of India, 2005.
24. Chung Tin. Robust multi-UAV planning in dynamic and uncertain environments. *Work*, 2004.
25. F. Schler, A. la Cour-Harbo, and M. Bisgaard. Generating approximative minimum length paths in 3d for uavs. In *2012 IEEE Intelligent Vehicles Symposium*, pages 229–233, June 2012.
26. Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.
27. Steven M LaValle and James J Kuffner Jr. Rapidly-exploring random trees: Progress and prospects. 2000.
28. Robert Bohlin and Lydia E Kavraki. Path planning using lazy prm. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 1, pages 521–528. IEEE, 2000.
29. Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.
30. James J Kuffner and Steven M LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE, 2000.
31. Helen Oleynikova, Michael Burri, Zachary Taylor, Juan Nieto, Roland Siegwart, and Enric Galceran. Continuous-time trajectory optimization for online uav replanning. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 5332–5339. IEEE, 2016.
32. Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.
33. Alex Nash, Kenny Daniel, Sven Koenig, and Ariel Felner. Theta^*: Any-angle path planning on grids. In *AAAI*, pages 1177–1183, 2007.
34. Joseph Carsten, Dave Ferguson, and Anthony Stentz. 3d field d: Improved path planning and replanning in three dimensions. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 3381–3386. IEEE, 2006.
35. Alex Nash and Sven Koenig. Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D. 2010.
36. Mihail Pivtoraiko, Daniel Mellinger, and Vijay Kumar. Incremental micro-uav motion replanning for exploring unknown environments. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 2452–2458. IEEE, 2013.

37. Liang Yang, Juntong Qi, Jizhong Xiao, and Xia Yong. A literature review of UAV 3D path planning. *Proceedings of the World Congress on Intelligent Control and Automation (WCICA)*, 2015-March(March):2376–2381, 2015.
38. Valeri Kroumov, Jianli Yu, and Keishi Shibayama. 3D path planning for mobile robots using simulated annealing neural network. *International Journal of Innovative Computing, Information and Control*, 6(7):2885–2899, 2010.
39. Y Volkan Pehlivanoglu, Oktay Baysal, and Abdurrahman Hacioglu. Path planning for autonomous uav via vibrational genetic algorithm. *Aircraft Engineering and Aerospace Technology*, 79(4):352–359, 2007.
40. E Masehian and G Habibi. Robot Path Planning in 3D Space Using Binary Integer Programming. *International Journal of Mechanical Systems Science and Engineering*, 1(5):1255–1260, 2007.
41. Ellips Masehian and M. R. Amin-Naseri. A voronoi diagram-visibility graph-potencial field compound algorith for robot path planning. *Journal of Robotic Systems*, 21(6):275–300, 2004.
42. Lifeng Liu and Shuqing Zhang. Voronoi diagram and gis-based 3d path planning. In *2009 17th International Conference on Geoinformatics*, pages 1–5, Aug 2009.
43. Helen Oleynikova, Zachary Taylor, Roland Siegwart, and Juan Nieto. Safe Local Exploration for Replanning in Cluttered Unknown Environments for Micro-Aerial Vehicles. *arXiv*, 2017.
44. Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.
45. GRVC-CATEC team. European robotics challenge, 2017. [Online; accessed 26 sep 2017].
46. RobMoSys: Composable Models and Software for Robotic Systems. User stories. [Online; accessed 12 fev 2018].
47. RobMoSys: Composable Models and Software for Robotic Systems. General purpose modeling languages and dynamic-realtime-embedded domains. [Online; accessed 12 fev 2018].
48. Open source ros wiki. Cis. [Online; accessed 12 fev 2018].
49. Tully Foote, Isaac Saito, Phillip Reed Reed, Jeremy Adams, and Florian Weihardt. Cis. [Online; accessed 12 fev 2018].
50. Google. Google c++ testing framework, 2017. [Online; accessed 28 sep 2017].
51. Shaojie Shen, Nathan Michael, and Vijay Kumar. Stochastic differential equation-based exploration algorithm for autonomous indoor 3D exploration with a micro-aerial vehicle. *The International Journal of Robotics Research*, 31(12):1431–1444, oct 2012.