Robot Navigation with PolyMap, a Polygon-Based Map Format

Chapter · January 2020

DOI: 10.1007/978-3-030-29513-4_82

Guillaume Lozenguez
IMT Lille Douai
41 PUBLICATIONS **346** CITATIONS
SEE PROFILE

Luc Fabresse
Institut Mines-Télécom
93 PUBLICATIONS **440** CITATIONS
SEE PROFILE

Noury Bouraqadi
IMT Lille Douai
118 PUBLICATIONS **677** CITATIONS
SEE PROFILE

Robot Navigation With PolyMap, a Polygon-based Map Format

Johann Dichtl, Xuan S. Le, Guillaume Lozenguez, Luc Fabresse, and Noury Bouraqadi

IMT Lille Douai, Univ. Lille, Unité de Recherche Informatique et Automatique, F-59000 Lille, France
firstName.lastName@imt-lille-douai.fr

Abstract. Vector-based map formats such as PolyMap are more compact in term of memory footprint compared with regular occupancy grid map formats. The sparse nature of a PolyMap along with the explicit representation of traversable space and frontiers makes it suitable for mono and multi-robot autonomous exploration.
In this paper, we propose a PolyMap-based navigation stack suitable for autonomous exploration. This navigation stack internally relies on a topological graph that is automatically computed from a PolyMap. This graph is critical to significantly reduce the search space when computing a global path compared with traditional occupancy grid maps. We also discuss two optimizations to speed up path planning computation on this graph. Our experiments demonstrate the efficiency of our PolyMap-based navigation stack.

Keywords: Mobile Robot Navigation; Vector-Based Map; Topological-based Navigation

1 Introduction

Mobile robots heavily rely on accurate representations of their environment (named *maps*) to fulfill their tasks. Maps are often restricted to 2D as a direct result of the capabilities of the sensors mounted on the robot. Building a map is a Chicken-and-Egg-Problem because it requires the current location of the robot while localization requires a map. *Simultaneous Localization and Mapping* (SLAM) tackle this problem using few major techniques such as: *Bayes-based Filters* [20] (*Kalman Filters* [9], *Particle Filters* [6]) and *Graph-Based SLAM* [8,11]. Most SLAM algorithms use an occupancy grid representation for the map (*grid-based* maps) in which each cell of the grid depicts a small area as either obstacle, free space, or unknown space.

The size of grid-based maps is directly related to the size of the mapped environment and the resolution of the map *i.e.* how much space is represented by one cell. This memory consuming representation is a critical issue especially in the context of a multi-robot autonomous exploration scenario [17]. Indeed, while robots are exploring the environment, they continuously update their map of the environment. The updated map must then be shared over wireless with other robots or even human operators/supervisors if any. In these scenarios, the occupancy grid map format is inefficient because it requires a lot of bandwidth and it limits the frequency of map exchanges [1].

There are other map formats such as *feature-based* or *vector-based* maps. The work presented in this paper relies on PolyMap [3] because it is a compact vector-based format and it is suitable for autonomous exploration thanks to an explicit representation of *frontiers* [4,10]. In a nutshell (cf. Section 3 for further details), a PolyMap is made of *simple* polygons represented by line segments. A line segment can represent different things such as a *frontier* that delimits the transition between explored and unexplored space.

In multi-robots autonomous exploration scenarios, frontiers are used to prioritize areas of interest and reduce the global exploration time by optimizing the assignment of frontiers to different robots. This optimization problem requires to compute a lot of paths from target frontiers to robots' positions. Indeed, each robot must select (or be assigned) a frontier to explore among several candidates accordingly to the path to reach them (e.g. select the closest frontier). This optimization problem is dynamic because the global map changes as well as frontiers. In this context, computing paths is key and must be efficient.

This papers focuses on global navigation on a PolyMap i.e. computing a path from the robot's current position to a target area. This task cannot be directly performed on a PolyMap. Transforming a PolyMap into a grid-based map is not an option in terms of memory and processing consumptions. In this paper, we propose a navigation stack that computes a topological graph from a PolyMap. From the theoretical point of view, this topological graph reduces the search space when computing paths. We also report on experimentations that shows the efficiency of this solution when it is implemented with specific data structures.

This paper is organized as follows. Section 2 presents a state of the art of navigation solutions for mobile robots. Section 3 presents the PolyMap format used to represent the robot environment. In Section 4, we describe our main contribution that consists in a PolyMap-based navigation stack. This involves a building process of a topological graph out of a PolyMap and its challenges. Section 5 reports on experiments that validate the approach in simulation on different terrains. We also present two different optimizations that make our navigation stack more efficient as shown in the latest results. Section 6 concludes this papers and draws some perspectives of this work for multi-robots systems.

2 State of the Art of Robot Navigation

Autonomous navigation for exploration requires the robot to perceive and to estimate the obstacle shapes to plan a safe path to a target position. This state of the art presents four families of approaches that tackle this problem. Given our objective presented in introduction, we are particularly interested in the required computing resources and the dependencies to a specific map model of each family.

2.1 Reactive navigation

Reactive navigation consists in computing the robot movement instantaneously accordingly to its current perception. Bug algorithm [16] for instance, guarantees the robot to attain a reachable target, in static environment, by moving along obstacles on the robot's trajectory. Reactive navigation requires: the local perception of obstacles, an efficient continuous localization of the robot, and the position of the goal. No specific localization technique or map format are required. However, the lack of global planning typically leads to suboptimal robot trajectories.

2.2 Grid-based Navigation

Several approaches are based on a regular decomposition of the environment into cells such as Occupancy Grid Maps to compute an efficient trajectory from the robot position to a target position. For instance, an approach based on potential field [2] guides the robot with repulsive forces spread from the obstacles and an attractive force directed toward the target. Like other regular-cells-navigation approaches (as based on Markov Decision Process [5]), the action performed in a cell depends on an evaluation of neighboring cells. The trajectory planning requires several iterations on the complete map to converge to a stable configuration, driving a robot towards one and only one target. The resulting trajectory could be classified as optimal regarding the cell dimension. However, it requires computation that rapidly become unpractical in large environment or in multi-task and multi-robot scenario.

2.3 Heuristic approaches

To speed up computation, heuristic approaches compute paths on a graph (adjacent traversable cells) that is built from Occupancy Grid Maps produced by most SLAM algorithms. A textbook case is provided with the Robot Operating System (ROS) module *move_base*[1][15]. Some well-known path planning algorithms, such as *Dijistra, A^*, Navfn, etc.* are employed on a graph generated from a Grid Map where obstacle shapes are inflated. This way, the approach returns the shortest path to a target position while the robot keep safe distances to obstacles.

These approaches based on converting Grid Maps usually lead to generate large and hard to process graphs making path planning slow. Approaches based on Rapidly exploring Random Tree [12] tackle this problem by performing the A^* algorithm while iteratively building a tree (instead of a graph) at the same time. At each iteration, random movements toward the target position generate new reachable positions from the most promising ones. In case of a dead end (all random movements fail because of obstacles in the map), a backtrack mechanism is activated. The algorithm ends when the target is reached or when no more position in the tree can be extended. Each path computation requires to build a new tree.

2.4 Topology-based Navigation

Generally, topology-based navigation (as proposed in [14]) relies on an hybrid reactive/deliberative architecture. Reactive modules control the robot movement toward "easily reachable" positions and are supervised by deliberative modules. Deliberative

[1] https://wiki.ros.org/move_base - Author: Eitan Marder-Eppstein

modules are responsible for SLAM and path planning where a path is defined as a succession of reachable positions accordingly to the reactive capabilities. Ideally, a robot should use a graph with a minimum number of nodes and edges to fastly compute those path between any couple of positions. Nodes represent significant area (room corner, intersection, etc.) and edges model the capability of a robot to move from a node position to another.

Topological navigation could be directly based on topological mapping [13] with a loss on the obstacles shaping. More classically, a topological map would be generated from a grid map as a *Voronoi* [19] (graph built with the farthest paths to the obstacles) or a Visibility Graph (obstacle corner positions connected while a free straight line exists). From our knowledge, there is no Visibility Graph approach proposed to handle on-line computations in exploration context.

2.5 Summary

In this paper, we investigate topology-based navigation with a topological graph generated from a *PolyMap*. The *PolyMap* structure is close to a topological graph and requires fewer computations, which will permit to efficiently supervise robot movements to a target.

3 Overview of the PolyMap Format

This PolyMap [3] format provides advantages over occupancy grids in terms of memory footprint, and does not fall behind in visualization compared to pure topological maps. This is a significant advantage when the map needs to be updated frequently over a wireless connection. Additionally the explicit modeling of frontiers helps with autonomous exploration. The sparse nature of the map format also provides advantages when it comes to navigation tasks, as we will show in this paper. But, first we need to further present more formally the *PolyMap* format before building our new navigation stack on top of it.

3.1 Model

PolyMap represents the environment using simple polygons. *Simple* polygons are non-self-intersecting closed polygons [7]. Our polygons model only explored space, i.e. the inside of all polygons is traversable space. Formally, a PolyMap M is defined as:

$$type: \quad T = \{obstacle, frontier, sector\} \tag{1}$$

$$vector: \quad V = \{(a, b, t) | a, b \in \mathbb{R}^2; a \neq b; t \in T\} \tag{2}$$

$$polygon: \quad P = \{v_i | i \geq 3; v_i \in V\} \tag{3}$$

$$PolyMap: \quad M = \{p_i | 1 \leq i \leq n; p_i \in P\} \tag{4}$$

with p_i being a *simple* polygon, and n being the number of polygons forming the map.

We define three types of line segments :

Obstacles: they represent the outline of obstacles.
Frontiers: they model the transition from free/traversable space to unknown/unexplored space.
Sectors: they delimit a traversable transition between the polygon the vector belongs to and another adjacent polygon.

Each line segment has exactly one type, polygons can contain line segments of different types. The direction of line segments is chosen, so that clockwise oriented polygons contain traversable space inside. Figure 1 shows an example of map containing two different types of line segments: red line segments represent obstacles and yellow line segments represent sectors.

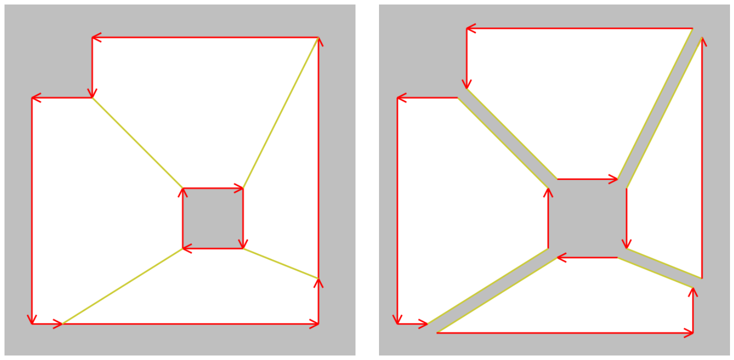

Fig. 1: A PolyMap example. The left figure shows the map itself, the right figure has the polygons moved apart to show the four individual polygons that make up this map.

3.2 BSP-Tree Representation

As with other map formats (e.g. grid map), a PolyMap can be obtained via a SLAM process. We have developed PolySLAM, an algorithm that builds a PolyMap map directly from raw sensor data (e.g. laser data). We will not detail this algorithm here since it is beyond the scope of this writing and deserves a dedicated paper. However, to introduce the BSP-Tree representation of PolyMaps, we need to introduce the concept of *Keyframes* used by PolySLAM.

Starting from raw sensor scans, PolySLAM produces *Keyframes*. Those are simple polygons, that represent the space surrounding a robot at a given point within the map's referential. Each new keyframe represents new acquired knowledge of the environment. It is then merged into the existing map during the SLAM.

To improve the speed of the merging keyframes into a PolyMap, we represent the map using a Binary Space Partitioning Tree (BSP-Tree) [18]. A BSP-Tree is a tree data structure. Each node of a BSP-Tree holds a hyperplane that partitions the hyperspace in two parts, while the leaves hold objects localized in the areas of hyperspace, separated by hyperplanes. PolyMap is a format for 2D maps. Hence, objects are polygons, while hyperplanes are 2D splitting lines. The left sub-tree for a splitting line gathers polygons located on one side of the line. Polygons in the right sub-tree are located at the other side. Figure 2 shows a PolyMap where splitting lines of the BSP-tree visible.

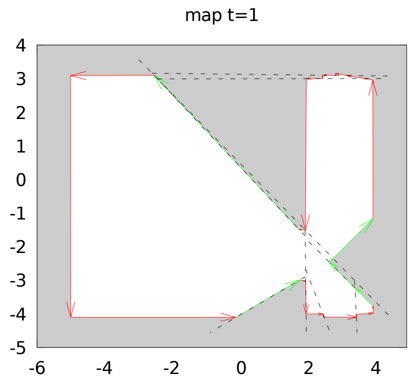

Fig. 2: Example of a PolyMap represented in a BSP-tree. Splitting lines are displayed as black dashed lines.

To merge a new keyframe into a PolyMap, the polygon that represents the keyframe is inserted into the BSP-tree. We start at the splitting line that is the root of the BSP-tree, and go down until reaching a leave. For each splitting line, we go either left or right depending the side of the splitting line where the polygon is located.

If a polygon intersects with a splitting line then it is cut into smaller polygons. The decomposition is done in a way to ensure that each of these new polygons lies on only one side of the 2D line. Last, these new polygons are inserted into the BSP-tree.

4 PolyMap-based Navigation

A PolyMap is a compact map format suitable to accurately represent a space. But, it uses a BSP-Tree data structure that is not directly suitable for navigation. Therefore, we need to build a topological graph out of the BSP-Tree.

4.1 Numerical Problems to Consider

To build a topological graph, we need to determine which polygons share overlapping vectors of type *sector*. While this looks like an easy task at first glance, there are a few details that make this more tricky then expected. The first problem is, that we cannot just look for an identical vector with start and end points reversed, since vectors may overlap without sharing any common start or end points. This is illustrated in Figure 3 in which the opposing horizontal vectors either share the start or end point, but never both. Note, that in this example the vertical sector borders do share both start and end points.

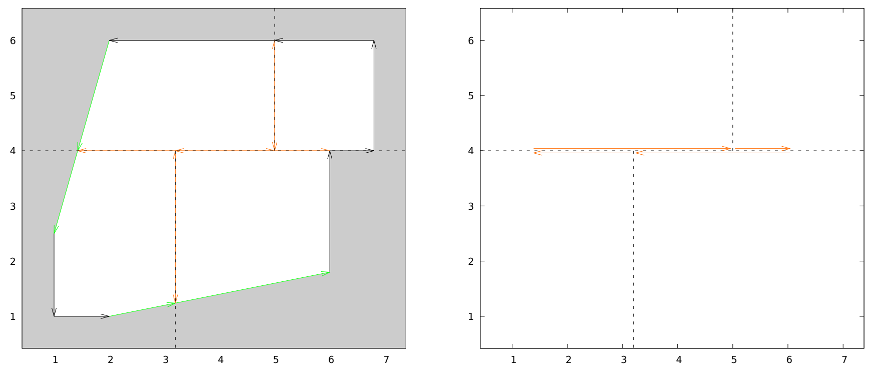

Fig. 3: An example of sector borders partial overlapping with vectors on the other side of the splitting line. The left figure shows the map embedded into a BSP tree with three splitting lines. The right figure highlights the overlapping vectors in question.

The second problem is, that computing whether two given lines (extending two vector that we want to test) are collinear is numerically challenging. For example, the nature of how computers store float point numbers makes it highly likely, that two lines/vectors that should be considered collinear have a slightly different orientation. And for similar reasons, it is possible that for example the start or end point of a vector does not exactly lie on a line that is collinear to the vector. Adding a tolerance to similarity tests on the other hand provides us with the problem, that we may encounter false positives when testing for overlapping vectors. We circumvent this problem by using the BSP-tree structure that created the sector-typed vectors in the first place, as explained in the next section.

4.2 From BSP-Tree to Topological Graph

By construction, every sector-typed vector is adjacent to a a splitting line in the BSP-Tree. Therefor, instead to consider all vectors of the map, we can narrow down potentially overlapping vectors to those that lie on the same splitting line. The structure of the BSP-Tree makes this convenient and relatively fast, assuming that the tree itself is balanced. Once we have collected all vectors that lie on the same splitting line, we compute which ones overlap. Now we can add edges to the graph for each polygon that is sharing an overlapping vector with a polygon on the other side of the splitting line. This is done by adding the corresponding neighbors to the node that represents the polygon in the graph (and creating new nodes if the respective polygons are not already represented in the graph). This process creates a topological graph based on the PolyMap, which is considerably more sparse than a graph based on a conventional grid map.

```
Data: BSP-Tree $bspTree$
Result: top. graph $graph$
begin
    Dictionary $dict := \{splittingLine \to (collection\ of\ \{vector;\ polygon\})\}$
    foreach $leaf$ in $bspTree$ do
        $node := leaf$
        $polygon := leaf\ polygon$
        while $node$ has parent do
            $node := node\ parent$
            $line := node\ splittingLine$
            foreach vector $v$ in $polygon$ do
                if $v$ is collinear with $line$ then
                    add $line \to \{v;\ polygon\}$ to $dict$
                end
            end
        end
    end
    foreach $(splittingLine \to (collection\ of\ \{vector;\ polygon\}))$ in $dict$ do
        foreach entry $\{v;\ p\}$ in $(collection\ of\ \{vector;\ polygon\})$ do
            foreach entry $\{v2;\ p2\}$ in $(collection\ of\ \{vector;\ polygon\})$ do
                if $v$ direction = $-1\ *\ v2$ direction then
                    if $v$ overlaps with $v2$ then
                        find or create node $N$ for polygon $p$
                        find or create node $N2$ for polygon $p2$
                        add neighbor $N2$ to node $N$
                        add neighbor $N$ to node $N2$
                    end
                end
            end
        end
    end
end
```

Algorithm 1: Building a topological graph from a BSP Tree

4.3 Path Planning on a Topological Graph

Once a topological graph has been build it can be used to compute the path to a target location in the environment. The first step is to determine the current location of the robot in the topological graph. For this we select the polygon that contains the center of the robot. This can be done efficiently since we have the map stored in a BSP-Tree which allows us to just walk down the tree on whichever side of the splitting lines the robot's center is located. The polygon can then be used to identify the corresponding node in the topological graph. This will act as our starting node for a conventional $A^\star$ or $D^\star light$ graph search. For the final result of our graph search, we replace the first node with a temporary node that contains the robot's current location instead, since we otherwise would use the polygon's center for further computations.

A simple example is illustrated in Figure 4, which shows navigation on a small map. The robot is located at the bottom part of the map, labeled 'start'. The target area is in the top right corner of the map, labeled 'goal'. With the map placed in a BSP-Tree, it is partitioned into smaller polygons, indicated by the yellow lines. Each leaf in the BSP-Tree is linked to a node in the graph via the polygon that both share. In this example the topological graph (colored pink) contains eleven nodes. With an $A^\star$ graph search we are able to find a valid path from the robot's position to the target area, as shown in the right figure.

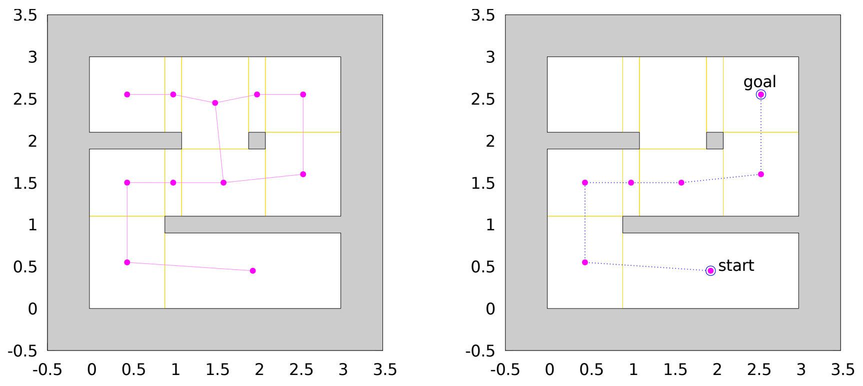

Fig. 4: Navigation example on a small map. The left figure shows the map portioned into smaller areas, with the topological graph overlayed. The right figure shows the computed path that leads from the start location to the target area.

5 Experiments and Optimizations

5.1 First results

First tests show that the concept of building a topological graph from a PolyMap is feasible. But at the same time, we find that the topological graph created this way has not yet reached its full potential. When building a PolyMap with keyframes that contain noise, we no longer have perfect alignment of vectors, and as a result, the PolyMap contains a considerable amount of small polygons that are clustered around obstacles. Furthermore, the incremental map building from keyframes promotes long thin polygons along the newly explored areas. There are nodes of the topological graph that are essentially inaccessible by any robot because they are too close to obstacles is obviously undesirable. And long thin polygons are also less desirable than more compact shaped polygons, since the latter are more “local” representation of the environment, and leads in general to less zigzag-shaped trajectories. Both issues can be solved easily, as shown in the following.

5.2 Using Grid Partitioning on the PolyMap

Our solution to prevent a build-up of excessively long thin polygons is to impose a grid structure on the map. All polygons that span over a grid line are split on the respective line, and the new polygons are treated separately. This breaks up all large polygons into smaller polygons and provides a more even distribution of nodes in the topological graph as a result. Compared to conventional grid maps, our cell dimension is relatively large: 1 by 1 meter by default. Each occupied cell contains a BSP-Tree, like the original PolyMap format, but now limited in its dimension to the size of the grid cell. This setup allows us to quickly access polygons, first by determining which grid cell we need, and then by searching within the BSP tree of the cell. Lookup of the cell typically is done in $O(1)$ time complexity (basically a 2D table lookup), and finding elements in the BSP tree (if balanced) takes $log(N)$ time, with N being the number of elements in the tree. This also removes any worries about unbalanced BSP-Trees, since the individual trees are now much smaller, so that even a degenerated tree wont pose a problem performance-wise.

We formally define the GridPolyMap as follows:

$$cell: \quad C = \{r | r \in M\} \tag{5}$$

$$GridPolyMaph: \quad G = \{m_{i,j} | 1\mathbb{N}; m_{i,j} \in C\} \tag{6}$$

The GridPolyMap G is a two dimensional grid where each entry contains a cell C. A cell in return contains a map M as defined in (4), stored in a BSP-Tree.

An direct comparison between a map with and without grid overlay can be seen in Figure 5. By separating parts of the maps into individual cells we prevent the larger polygons to be split into smaller ones when the merging process introduces new splitting lines, because the splitting lines in one cell do not affect any other cells.

5.3 Removing Inaccessible Nodes from the Graph

When the size of the robot is known, we can take the robot footprint into account to remove nodes from the graph that are not accessible by the robot. As a quick heuristic we remove all nodes from the graph where the polygons center point is closer to an obstacle than the radius of the robot. Since we typically experience a lot of small polygons around obstacles, this reduces the size of the graph by a significant amount. In fact, the

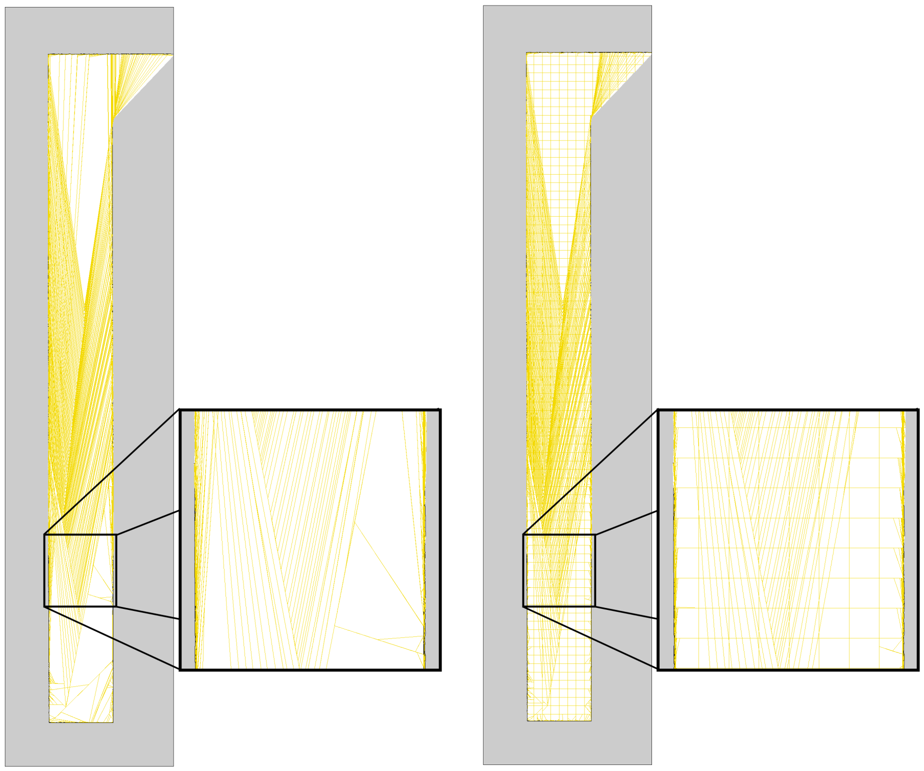

Fig. 5: Comparison of a PolyMap versus a GridPolyMap. The environment is a long wide corridor that exceeds the robots maximum sensor range. The incremental updates from the moving robot add long thin slices to the explored area, which are visible in the two figures. However the right figure has a grid overlay, which splits long polygons into smaller ones.

number of nodes and edges is reduced by about two orders of magnitude in our tests. An example can be seen in Figure 6, where the number of nodes in the graph drops from 62427 to 688. The robot radius used in this example was 0.25m.

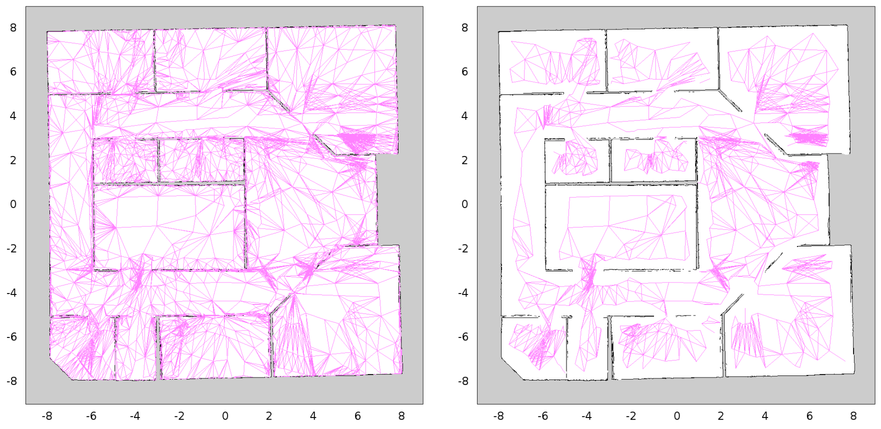

Fig. 6: The left figure shows the full topological graph of the map, drawn on top of the actual map. The right figure shows the pruned graph with all inaccessible nodes removed.

5.4 Experiment Results

To test the impact of our optimizations, we created a topological graph for a few of maps. The map has been created with our PolySLAM algorithm, using data collected from a simulated environment. Table 1 shows the results with and without pruning the topological graph. We can see a significant reduction of the graph size in all cases. We also notice, that the grid overlay increases the number of nodes by only a moderate amount.

Map Format		full graph	pruned
PolyMap		60786	676
GridPolyMap (grid size in meter)	0.5	78972	1837
	1.0	63279	885
	2.0	62427	688
	4.0	62289	670

Table 1: Size of the topological graph (node count) for various map configurations.

We also created a topological graph from an occupancy grid representing an office environment (16m by 16m in size) and compared the performance. The occupancy grid has been inflated by 0.25m to make the results comparable to our pruned graph of the GridPolyMap, which also removed nodes that are closer than 0.25m to any obstacle. When creating the graph for the occupancy grid, we consider the 8 closest neighbors for each cell, allowing for horizontal, vertical, and diagonal edges. Table 2 and Figure 7 are showing the results of our experiment. With 72475 nodes, the graph computed from the occupancy grid is relatively large with respect of the environment. In comparison, the graph for the GridPolyMap only contains 688 nodes – two orders of magnitude less. Similarly, the A^* search requires significantly less iterations to reach the target on our GridPolyMap-derived topological graph. However, while the path for the occupancy grid contains more nodes, the actual distance is shorter. The reason for this is that the path for the GridPolyMap is not as close to the obstacles, and contains more zigzag-like movements.

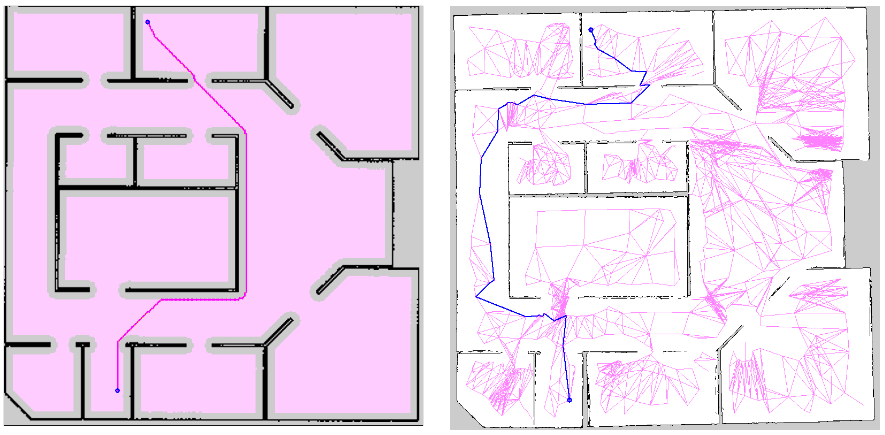
Fig. 7: Navigation on an Occupancy Grid in comparison to a GridPolyMap. The left image shows the resulting path on the occupancy grid, the right image on a GridPolyMap. The pink background in the left image visualizes the traversable area after inflating all obstacles. The pink lines in the right image represent the edges of the topological graph.

Graph source	node count	path node count	path length
Occupancy Grid	72475	344	19.45m
GridPolyMap	688	39	25.06m

Table 2: Comparison of the topological graph created from an occupancy grid and created from a GridPolyMap. The computed path in visualized in Figure 7.

6 Conclusion

In this paper, we introduce a solution for autonomous robot navigation based on PolyMap, a lightweight polygon-based map format. PolyMap represents a terrain using a BSP-Tree where leaves are polygons of traversable space. We have shown how to build a topological graph out of this BSP-Tree. Thanks to the vectorial nature of PolyMap, the resulting topological graph is considerably sparse. Compared to traditional grid-based maps, our solution allows faster path planning, while still using a conventional A^* or D^*light algorithms.

Although these results are already interesting per se, we have demonstrated that we can go beyond thanks to two optimizations. First, based on a grid-based partitioning of the map, we have drastically reduced the complexity of the searching for polygons within PolyMap. The second optimization consists into taking into account the robot footprint while building the topological graph. This allows to dismiss unreachable nodes, resulting into a smaller topological graph. Experiments validate the effectiveness of these optimizations.

As with future work, we plan to use our solution to perform multi-robot exploration. We expect to benefit from the lightweight nature of PolyMap, and the resulting topological graph to reduce the network bandwidth required by robot data exchange. This is critical in a context where robots need to setup an ad hoc communication network.

Acknowledgment

This work was completed in the framework of DataScience project which is co-financed by European Union with the financial support of European Regional Development Fund (ERDF), French State and French Region of Hauts-de-France.

References

1. Baizid, K., Lozenguez, G., Fabresse, L., Bouraqadi, N.: Vector Maps: A Lightweight and Accurate Map Format for Multi-robot Systems. In: International Conference on Intelligent Robotics and Applications, Springer International Publishing (2016) 418–429
2. Barraquand, J., Langlois, B., Latombe, J.C.: Numerical potential field techniques for robot path planning. IEEE Transactions on Systems, Man and Cybernetics **22** (1992) 224–241
3. Dichtl, J., Fabresse, L., Lozenguez, G., Bouraqadi, N.: Polymap: A 2d polygon-based map format for multi-robot autonomous indoor localization and mapping. In: International Conference on Intelligent Robotics and Applications, Springer (2018) 120–131
4. Faigl, J., Kulich, M.: On benchmarking of frontier-based multi-robot exploration strategies. In: 2015 European Conference on Mobile Robots (ECMR). (Sept 2015) 1–8
5. Foka, A.F., Trahanias, P.E.: Real-time hierarchical POMDPs for autonomous robot navigation. Robotics and Autonomous Systems **55**(7) (2007) 561–571
6. Grisetti, G., Stachniss, C., Burgard, W.: Improved techniques for grid mapping with rao-blackwellized particle filters. IEEE Transactions on Robotics **23**(1) (Feb 2007) 34–46
7. Grünbaum, B.: Convex Polytopes. Volume 221. Springer Science & Business Media (2013)
8. Hess, W., Kohler, D., Rapp, H., Andor, D.: Real-time loop closure in 2D LIDAR SLAM. Proceedings - IEEE International Conference on Robotics and Automation **2016-June** (2016) 1271–1278
9. Huang, S., Dissanayake, G.: Convergence and consistency analysis for extended kalman filter based slam. IEEE Transactions on Robotics **23**(5) (Oct 2007) 1036–1049
10. Juliá, M., Gil, A., Reinoso, O.: A comparison of path planning strategies for autonomous exploration and mapping of unknown environments. Autonomous Robots **33**(4) (2012) 427–444
11. Konolige, K., Grisetti, G., Kümmerle, R., Burgard, W., Limketkai, B., Vincent, R.: Efficient sparse pose adjustment for 2d mapping. In: 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems. (Oct 2010) 22–29
12. Kuffner, J., LaValle, S.: RRT-connect: An efficient approach to single-query path planning. In: International Conference on Robotics and Automation. Volume 2. (2000) 995–1001
13. Kuipers, B.: The Spatial Semantic Hierarchy. Artificial Intelligence **119** (2000) 191–233
14. Lozenguez, G., Adouane, L., Beynier, A., Martinet, P., Mouaddib, A.I.: Interleaving Planning and Control of Mobiles Robots in Urban Environments Using Road-Map. In: International Conference on Intelligent Autonomous Systems. (2012)
15. Marder-Eppstein, E., Berger, E., Foote, T., Gerkey, B., Konolige, K.: The office marathon: Robust navigation in an indoor office environment. In: Robotics and Automation (ICRA), 2010 IEEE International Conference on, IEEE (2010) 300–307
16. McGuire, K., de Croon, G., Tuyls, K.: A comparative study of bug algorithms for robot navigation. CoRR **abs/1808.05050** (2018)
17. Saeedi, S., Trentini, M., Li, H., Seto, M.: Multiple-robot Simultaneous Localization and Mapping - A Review 1 Introduction 2 Simultaneous Localization and Mapping : problem statement. Journal of Field Robotics (2015)
18. Thibault, W.C., Naylor, B.F.: Set operations on polyhedra using binary space partitioning trees. SIGGRAPH Comput. Graph. **21**(4) (August 1987) 153–162
19. Thrun, S.: Learning Metric-Topological Maps for Indoor Mobile Robot Navigation. Artificial Intelligence **99**(1) (1998) 21–71
20. Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics (Intelligent Robotics and Autonomous Agents). The MIT Press (2005)