Efficient Grid-Based Spatial Representations for Robot Navigation in Dynamic Environments

Boris Lau*, Christoph Sprunk, Wolfram Burgard

Autonomous and Intelligent Systems, University of Freiburg, D-79110 Freiburg, Germany

Abstract

In robotics, grid maps are often used for solving tasks like collision checking, path planning, and localization. Many approaches to these problems use Euclidean distance maps (DMs), generalized Voronoi diagrams (GVDs), or configuration space (c-space) maps. A key challenge for their application in dynamic environments is the efficient update after potential changes due to moving obstacles or when mapping a previously unknown area. To this end, this paper presents novel algorithms that perform incremental updates that only visit cells affected by changes. Furthermore, we propose incremental update algorithms for DMs and GVDs in the configuration space of non-circular robots. These approaches can be used to implement highly efficient collision checking and holonomic path planning for these platforms. Our c-space representations benefit from parallelization on multi-core CPUs and can also be integrated with other state-of-the-art path planners such as rapidly-exploring random trees.

In various experiments using real-world data we show that our update strategies for DMs and GVDs require substantially less cell visits and computation time compared to previous approaches. Furthermore, we demonstrate that our GVD algorithm deals better with non-convex structures, such as indoor areas. All our algorithms consider actual Euclidean distances rather than grid steps and are easy to implement. An open source implementation is available online.

Keywords: Incremental algorithms, Voronoi diagrams, Distance maps, Configuration space, Collision checking, Robot navigation

1. Introduction

Many approaches in robot navigation rely on occupancy grid maps to encode the obstacles of the area surrounding a robot. These maps can be learned from sensor data, they are well suited to solve problems like path planning, collision avoidance, or localization, and they can easily be updated to reflect changes in the environment.

In the past, several grid-based spatial representations have been developed that can be derived from grid maps, e.g., distance maps, Voronoi diagrams, and configuration space maps. These representations are important building blocks for many different robotic applications, since they can be used to speed-up algorithms that solve the aforementioned problems. This paper proposes incremental update algorithms to facilitate the online use of these representations in dynamic environments. We also apply our methods to update distance maps and Voronoi diagrams in the configuration space of non-circular robots, e.g., to speed-up path planning or collision avoidance for these types of platforms. This paper extends our previous work on these topics [2, 3] and includes additional experiments for 3D distance maps and incremental updates of distance maps and Voronoi diagrams in the context of simultaneous localization and mapping (SLAM).

The generalized Voronoi diagram (GVD) is defined as the set of points in free space to which the two closest obstacles have the same distance [4]. It is a discrete form of the Voronoi graph, which has been widely used in various fields [5]. In the context of robotics, Voronoi graphs are a popular cell decomposition method for solving navigation tasks. Their application as roadmaps is an appealing technique for path planning, since they are “sparse” in the sense that different paths on the graph

*Corresponding author

Email address: `lau@informatik.uni-freiburg.de` (Boris Lau) correspond to topologically different routes with respect to obstacles. This significantly reduces the search problem and can be used for example to generate the *n*-best paths for offering route alternatives to a user [6]. Also, moving along the edges of a Voronoi graph ensures the greatest possible clearance when passing between obstacles. When Voronoi graphs are discretized and stored as a map, they can lose their sparseness property due to erroneous interconnections between neighboring Voronoi lines. Our method to compute GVDs overcomes this problem with additional conditions that ensure the sparseness of the generated GVDs.

The cells in a distance map (DM) encode the distance to the closest cell that is occupied according to the corresponding occupancy map. Since a cell lookup only requires constant time, DMs provide efficient means for collision checks, to compute traversal costs for path planning, and for robot localization with likelihood fields [7]. Since the computation of this transform is carried out without considering the shape of the robot, direct application of plain DMs is restricted to circular approximations of the robot's footprint.

For non-circular robots in passages narrower than their circumcircle, however, circularity is too crude an assumption, and collisions have to be checked for in the three-dimensional configuration space (c-space) of robot poses. Also, even for robots moving on a plane as considered in this paper, 3D obstacles and collisions can be important: applications such as robotic transporters, wheelchairs, or mobile manipulators can require the robot to partially move underneath or above obstacles as shown in Fig. 1. In these cases, collision checks easily become a dominant part of the computational effort in path planning. However, by convolving a map with the discretized shape of the robot, one can precompute a collision map that marks all colliding poses. With such a map, a collision check requires just a single lookup, even for 3D obstacle representations.

In changing environments, precomputed GVDs, DMs, and c-space maps have to be updated regularly to always reflect the current state of the corresponding occupancy map. These changes can be caused by moving people or vehicles, newly explored areas during mapping, or when correcting a map after closing a loop in SLAM.

In this paper, we present efficient methods to compute and update these representations. Since our algorithms perform all updates in an incremental way, i.e., recomputing only parts affected by changes, they can be applied online even with large maps or with more than two dimensions. In comparison to previous approaches, our methods require less computational effort, are easy to implement, and work in both indoor and outdoor environments.

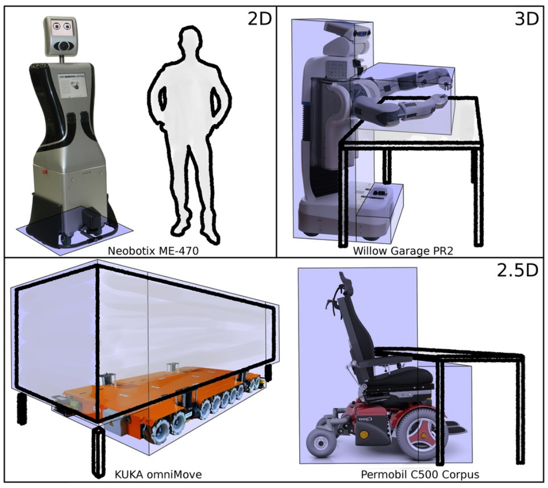

Figure 1: For some applications, representing obstacles and robots by their 2D footprints can be sufficient (top-left). For overhanging parts of robots, their load, or obstacles, 2.5D representations are needed (bottom), whereas interaction tasks can also require actual 3D obstacle and robot models (top-right). Robot shape approximations as used in our experiments are depicted in blue.

Additionally, we combine DMs and GVDs with c-space collision maps, and propose distance transformed c-space maps and c-space Voronoi diagrams. These can be used for efficient collision checking and path planning of non-circular robots. With our algorithms described in this paper, these representations can be updated in an incremental way as well.

After discussing related work in Sect. 2, we describe the brushfire algorithm in Sect. 3. It can be used to compute static DMs, and it is an important foundation for our dynamic DM and GVD algorithms proposed in Sects. 4 and 5. Sect. 6 describes our dynamic c-space collision maps, followed by the c-space DM and c-space GVD in Sect. 7. Our experiments are presented in Sect. 8 before we conclude our paper in Sect. 9.

2. Related Work

In the past, many different approaches have been proposed to compute DMs, GVDs, and c-space collision maps. With the goal of applying them online in dynamic environments, a lot of effort has been spent on developing more efficient algorithms. However, unlike ours, most of these approaches do not exploit the potential of incremental updates. The remainder of this section presents related work for the different spatial representations and discusses the contribution of our methods.

2.1. Distance Maps

Many different approaches have been proposed to compute static two-dimensional DMs, e.g., linear image traversal [8], dimensional decomposition [9], and distance propagation with the brushfire algorithm [10]. We review the brushfire algorithm in Sect. 3. For a comparative review of other approaches please refer to the survey by Fabbri et al. [9].

Whenever a cell in a grid map is newly occupied or vacated, the corresponding DM has to be updated to reflect this change. A trivial method is to recompute distances for patches within $\hat{d}$ around all changed cells, where $\hat{d}$ is an upper bound on the minimum obstacle distance in the environment. However, this method usually updates substantially more cells than necessary, e.g., if $\hat{d}$ is high due to large open spaces or if the changed cells cover a wide area. Furthermore, efficiently determining the minimal update area is not trivial, if the changes affect the occupancy of several cells.

Kalra et al. proposed a dynamic brushfire algorithm that incrementally updates DMs and GVDs by propagating wavefronts starting at newly occupied or vacated cells [11]. While their method is based on the incremental path planning algorithm D^* by Stentz [12], the algorithm proposed here is directly derived from the brushfire algorithm and requires substantially less computational time for the same task due to a considerably reduced number of cell visits.

The wavefronts of Kalra et al. accumulate 8-connected grid steps to approximate obstacle distances [11]. This overestimates the true Euclidean distances by up to 8.0% [13], which for a robot implies either a collision risk or overly conservative movements. Scherer et al. adopted and corrected Kalra's algorithm for their DM update method [14]. They propagate obstacle locations rather than grid step counts to determine Euclidean distances, which reduces the absolute overestimation error below an upper bound of 0.09 pixel units [13]. According to Cuisenaire and Macq, the shortest distance at which this propagation error can occur is 13 pixels [15], which yields a maximum relative error of 0.69%. Additionally, by propagating obstacle references, our representations can provide the location of the closest obstacle rather than just the distance to it, which can be appealing for collision avoidance methods. In a recent publication, Scherer et al. build on our original method for DM updates and combine it with their approach to map scrolling [16].

This paper extends our DMs presented in [2] to 3D by adding the possibility to limit the propagated distances to maintain online feasibility in large open spaces and outdoors as proposed by Scherer et al. [14].

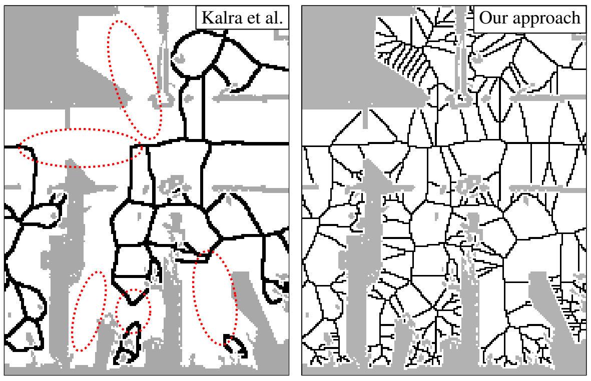

Figure 2: GVD of an indoor map, computed by the approach of Kalra et al. [11] and our method. The ellipses mark missing Voronoi lines, see Sect. 2.2. Our approach generates thin Voronoi lines, such that different paths on the GVD correspond to topologically different routes.

Additionally, we describe how to further reduce the number of visited neighbor cells, which increases the efficiency for 3D DMs, and we present additional experiments.

2.2. Voronoi Diagrams

Traditional Voronoi algorithms compute parametric lines or curves that separate singular obstacle points or line segments represented in continuous space. There are approaches to update such Voronoi graphs, e.g., for newly discovered obstacles during exploration [17, 18], moving input points [19], or points that have been inserted or deleted [20]. However, analytic approaches are not practical for use with grid maps, since they would attach Voronoi lines between all pairs of occupied cells, even for larger obstacles and walls.

Several approaches exist to incrementally construct Voronoi diagrams, e.g., [21, 22]. However, most of them are not suitable for dynamic environments or incremental mapping with SLAM, since they do not support clearing previously occupied map cells, which is necessary to handle dynamic environments and map corrections caused by loop-closures in SLAM.

Tao et al. propose line fitting to overcome this problem for a SLAM application [17]. Their algorithm can incrementally construct analytic Voronoi graphs during exploration, which comprises the addition of obstacles, but not their removal. Furthermore, one would have to update the line fitting as well, which can cause sudden changes in the Voronoi graph.

The approach for updating GVDs proposed by Kalra et al. directly operates on grid maps, but introduces obstacle identifiers that are uniquely assigned to a compound of connected obstacle cells [11]. If two adjacent cells have different closest obstacles according to their identifier, both cells are added to the GVD. This condition however generates two-cell-wide lines that violate the sparseness property of the GVD. Additionally, it does not generate Voronoi lines in the interior of concave obstacle compounds like rooms or corridors as shown in Fig. 2 (left). This destroys the connectivity of the GVD and is problematic for path planning, especially in indoor environments. Furthermore, since Kalra et al. use 8-connected step distances for the distance maps, the Voronoi lines also follow this metric and thereby only approximate the GVD.

In this paper we describe our condition-based approach to incrementally update GVDs, first proposed by Lau et al. [2]. It considers actual Euclidean distances and uses a new criterion that determines if a cell is part of the GVD or not, without requiring obstacle identifiers as the approach by Kalra et al. [11]. Furthermore, our approach correctly handles indoor environments and generates thin Voronoi lines that can be used for the *n*-best computation of topologically different paths as shown in Fig. 2. Additionally, it benefits from the speed-up of our dynamic brushfire algorithm described above.

2.3. Configuration Space Maps

Configuration space (c-space) maps encode if a given robot pose leads to a collision with the environment or not. Algorithms for efficient collision checking between three-dimensional objects continue to be an active area of research. Being in an overlap area between motion planning and computer graphics, most approaches represent the environment and the obstacles with polygon meshes. For example, Tang et al. recently proposed a connection collision query algorithm that detects collisions of triangle meshes moving between given states [23]. Hence, it can be used for sampling-based path planning. For online feasibility, Pan and Manocha use multi-core GPUs for collision queries [24]. Still, the cost per collision check depends on the number of polygons used to represent the tested objects.

For a collision avoidance system, Schlegel proposed to precompute collision distances for circular arc trajectories as a function of relative obstacle location and curvature [25]. Thus, the kinematic analysis is done offline and collision distances can be obtained with one lookup per obstacle. Instead, precomputing c-space representations further reduces the online effort for collision checks to a single lookup. Since the publication of the seminal paper by Lozano-Perez on c-space planning among static polyhedric obstacles [26], many approaches were proposed to reduce the cost for computing c-space obstacles, see for example the survey by Wise and Bowyer [27]. Linan and Zhenmin for example proposed a method to incrementally grow polygonal c-space obstacles for multiple robots, but did not consider changes other than ongoing exploration [28]. Because of the relevance of this problem, especially in dynamic environments, researchers are still working on improving the efficiency [29].

Convolving a grid map of a robot's environment with an image of its footprint yields a discrete c-space map. In order to reflect the current state of previously unknown or moving obstacles at all times, these maps need to be updated regularly. Kavraki proposed to use the fast Fourier transform (FFT) to reduce the computational cost of the convolution [30], and Therón et al. added parallelization for an additional speed-up [31]. Later, the same group proposed a multi-resolution approach to reduce memory and computational load in large workspaces [32]. To speed up path planning for autonomous cars, Ziegler and Stiller decompose the shape of the vehicle into circular disks [33].

As a first dynamic approach for changing environments, Wu et al. proposed to precompute colliding robot poses for each potentially occupied cell in the workspace of a manipulator [34]: taking the union of the colliding poses for a given set of occupied cells yields the c-space collision map without further recomputation. For mobile robots, however, the size of the operational area can render the database storage or the online computation of the union infeasible. In contrast, our method for updating c-space collision maps is truly incremental: it executes a regular map convolution in an offline phase, and during online application only updates the cells affected by changes in the environment [3].

For path planning with circular robots, 2D Voronoi diagrams are appealing roadmaps since they cover all topologically different paths in a map with a small number of cells. For rectangular robots however, 2D Voronoi planning loses its completeness property, which requires repairing paths in narrow areas where following the Voronoi diagram leads to collisions, e.g., by using rapidly-exploring random trees (RRTs) as proposed by Foskey et al. [35]. In this paper we combine dynamic distance maps and Voronoi diagrams with our novel incrementally updatable c-space collision maps. In this way, we overcome the aforementioned problem and can perform complete Voronoi planning in the configuration space of non-circular robots. Due to the ability to perform incremental updates, the resulting systems are suitable for online application in dynamic en-

Algorithm 1 Brushfire algorithm

```
computeDistanceMap()
 1: for all $s$ do
 2:     if $M(s) = 1$ then
 3:         $D(s) \leftarrow 0$
 4:         $obst(s) \leftarrow s$
 5:         insert($open, s, 0$)
 6:     else $D(s) \leftarrow \infty$
 7: while $open \neq \emptyset$ do
 8:     $s \leftarrow$ pop($open$)
 9:     lower($s$)
10: return $D$

lower($s$)
11: for all $n \in \mathrm{Adj}_8(s)$ do
12:     $d \leftarrow \|obst(s)-n\|$
13:     if $d < D(n)$ then
14:         $D(n) \leftarrow d$
15:         $obst(n) \leftarrow obst(s)$
16:         insert($open, n, d$)
```

vironments.

Although we use A* planning as an example application, our approach can be combined with other planners, e.g., D* Lite [36], or RRTs with Voronoi-biased sampling [37, 38].

3. The Brushfire Algorithm

The brushfire algorithm, as described by Verwer et al. computes static distance maps with a shortest path search similar to Dijkstra's algorithm with multiple sources [10]. By using a priority queue that orders the expansion of cells by the distance to their closest obstacle, the propagation spreads in wavefronts that start at the location of obstacles as shown in Fig. 3.

The pseudo-code of our algorithm is given in Alg. 1. As discussed in Sect. 2, we have modified it to store the location of the closest obstacle of each visited cell in the obstacle reference map $obst(s)$. Given a map $M(s)$, it initializes each free cell s of the distance map $D(s)$ with infinite distance (line 6). The occupied cells are initialized with zero distances and then inserted into the priority queue *open*: the function insert($open, s, d$) inserts s into the queue with distance d, or updates the priority if s is already enqueued (line 5).

As long as this queue contains cells, the algorithm iteratively calls the function pop(*open*) which returns the cell s with the lowest enqueued distance and removes it from the queue (line 8). It then updates the cells in the 8-connected neighborhood $\mathrm{Adj}_8(s)$ of s: if the distance d from a neighbor n to the closest obstacle of s as specified by $obst(s)$ is smaller than the current value $D(n)$ (line 13), the distance value and closest obstacle of n are updated with the obstacle of s (lines 14–15). Furthermore, each updated neighbor cell n is inserted into the priority queue with its new distance value to continue the propagation (line 16). Thus, each obstacle induces a propagation wavefront that expands in circles, and updates the distance values of the visited cells to the Euclidean distance to the obstacle. Since this update can only reduce the distance value associated with a cell, we call this type of propagation a "lower" wavefront[1].

If a wavefront cannot lower the distance of any neighbor of the visited cells, no further cells are enqueued. After all wavefronts came to a halt this way, the priority queue is empty and the algorithm returns the distance map.

The implementation of the priority queue can be done with a complexity of $O(1)$ for adding and removing elements, as described in Sect. 4.2. Then, the brushfire algorithm is in the same complexity class as a simple image passing algorithm, i.e., $O(n^2)$ for an $n \times n$ input map.

4. Dynamic 2D Euclidean Distance Maps

Movement, insertion, or deletion of objects causes individual cells in a binary occupancy grid map M to flip their state from free (0) to occupied (1) or vice versa. This section presents an approach to update Euclidean distance maps to reflect such changes using a dynamic variant of the brushfire method. After registering an arbitrary set of newly occupied and newly freed cells, the algorithm performs the update in an incremental way, i.e., exploiting its previous results. As discussed in Sect. 2, our approach is directly derived from the brushfire algorithm presented in Sect. 3, unlike the method by Kalra et al. which is derived from D* Lite [11].

The example in Fig. 4 shows how the DM from Fig. 3 is updated after removing an obstacle and inserting a different one. Frame (A) shows the initial state, which is equivalent to the final state (D) of Fig. 3.

When performing the update, newly occupied cells (blue outline) initiate "lower" wavefronts (B) that update the closest obstacle distance of affected cells similarly to the static variant of the algorithm. These wavefronts are propagated up to the point where a different obstacle is closer (C). In addition, "raise" wavefronts start at newly freed cells (red outline) and clear the distance entries of all cells whose closest obstacle was the deleted one (B). When they come to a halt at cells with a different closest obstacle, they initiate new "lower" wavefronts that recompute the distances for the cleared cells on the basis of the remaining obstacles (C).

Both the raise and lower wavefronts propagate themselves by enqueueing the neighbors of a processed cell into the same priority queue. Since the queue sorts its

[1]This corresponds to the nomenclature by Kalra et al. [11].

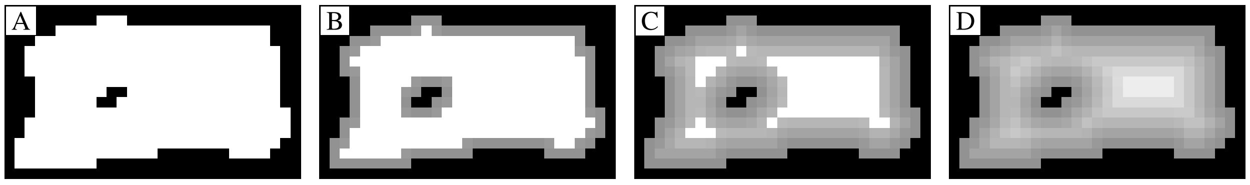

Figure 3: Computing a distance map with the static brushfire algorithm described in Sect. 3. The distance values are initialized with 0 (black) for occupied and infinity (white) for empty cells (A). The occupied cells initiate wavefronts that propagate the increasing distances, denoted by increasing brightness in (B) and (C). A wavefront stops if no further distance values can be lowered. After all wavefronts have stopped, the full distance map is computed (D).

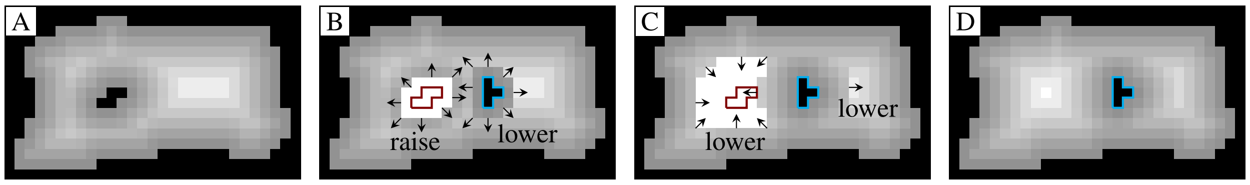

Figure 4: The dynamic brushfire algorithm presented in Sect. 4 is used to update the distance map shown in (A). To propagate the changes (B), a raise wavefront starts to delete the invalid values for a removed obstacle (marked red), and a lower wavefront propagates the new distances for an inserted obstacle (blue). Where the raise wavefront hits cells with a different (valid) closest obstacle, it halts and initiates a new lower wavefront to restore the invalidated distance values (C). After all wavefronts came to a halt, the update is completed (D).

elements by distance, the processing of raise and lower wavefronts is interwoven. After all wavefronts have stopped, the queue is empty and the update is completed (D).

4.1. The Dynamic Brushfire Algorithm Explained

The pseudo-code of the dynamic brushfire algorithm is given in Alg. 2.[2].

The algorithm is initialized with a given distance map $D(s)$ and an obstacle reference map $obst(s)$ for a corresponding occupancy grid map $M(s)$. If a cell s is occupied according to $M(s)$, it has a distance of $D(s) = 0$ and refers to itself as the closest obstacle location, i.e., $obst(s) = s$. Otherwise, $D(s)$ is the distance value to the closest occupied cell, whose location is stored in $obst(s)$. A change in the occupancy of a cell s is registered by calling the function setObstacle(s) or removeObstacle(s), which updates s and inserts it into a priority queue. Thereby, the function clearCell(s) resets s to $D(s) = \infty$ and $obst(s) = \text{cleared}$[3]. An additional flag *toRaise* is used to ensure proper processing of cells in the wavefronts, in particular where raise and lower wavefronts meet. It indicates for each cell, whether it has to process its neighbors with a raise wavefront (true) or not (false).

After registering a set of changes using setObstacle(s) and removeObstacle(s), the priority queue *open* is filled with the updated cells. Calling the function updateDistanceMap() performs the update by propagating the changes to all affected cells. While the priority queue is not empty, it repeatedly retrieves the next unprocessed cell s (lines 23–24). If s has still to propagate a raise wavefront, the function raise(s) is called (lines 25–26). If this is not the case and if s has a valid closest obstacle, the function lower(s) is called to propagate the lower wavefront (lines 27–29). The function isOcc(s) tests if a cell s is occupied by checking if $obst(s) = s$.

The function raise(s) processes each cell n in the 8-connected neighborhood $\text{Adj}_8(s)$ of s that has not been raised and still refers to a closest obstacle $obst(n)$ (lines 31–32).

The cell n is inserted into the priority queue with its old distance value (line 33). If the cell referenced by $obst(n)$ is no longer occupied, n is cleared and marked to propagate the raise wavefront (lines 34–36). Otherwise, the raise wavefront comes to a halt at n, leaves n unchanged, and propagates a lower wavefront, as shown in Fig. 4 (C). After processing the neighbors, the raise update of s is completed and *toRaise*(s) is set to false (line 37).

The function lower(s) considers each cell n in the 8-

[2]This corrects the typo in the original pseudo-code in [2] mentioned by Scherer et al. [16].

[3]Without a precomputed distance map, e.g., when mapping a new area, the algorithm can also be initialized by clearing all cells and registering occupied cells with setObstacle(s).

Algorithm 2 Pseudo-code for updating Euclidean distance maps

```
setObstacle(s)
17: $obst(s) \leftarrow s$
18: $D(s) \leftarrow 0$
19: insert($open, s, 0$)

removeObstacle(s)
20: clearCell($s$)
21: $toRaise(s) \leftarrow$ true
22: insert($open, s, 0$)

updateDistanceMap()
23: while $open \neq \emptyset$ do
24:     $s \leftarrow$ pop($open$)
25:     if $toRaise(s)$ then
26:         raise($s$)
27:     else if isOcc($obst(s)$) then
28:         $voro(s) \leftarrow$ false
29:         lower($s$)
30: return $D$

raise(s)
31: for all $n \in \mathrm{Adj}_8(s)$ do
32:     if ($obst(n) \neq$ cleared
            $\wedge \neg toRaise(n)$) then
33:         insert($open, n, D(n)$)
34:         if $\neg$isOcc($obst(n)$) then
35:             clearCell($n$)
36:             $toRaise(n) \leftarrow$ true
37: $toRaise(s) \leftarrow$ false

lower(s)
38: for all $n \in \mathrm{Adj}_8(s)$ do
39:     if $\neg toRaise(n)$ then
40:         $d \leftarrow \|obst(s) - n\|$
41:         if $d < D(n)$ then
42:             $D(n) \leftarrow d$
43:             $obst(n) \leftarrow obst(s)$
44:             insert($open, n, d$)
45:         else checkVoro($s, n$)
```

connected neighborhood $\mathrm{Adj}_8(s)$ of s. If a cell n is not marked to be part of a raise wavefront (lines 38–39), it is updated as in the static version of the algorithm: the Euclidean distance from n to the closest obstacle of s is compared to the current closest obstacle distance of n (lines 40–41). If it is smaller, the values for distance and closest obstacle of n are updated to reflect that $obst(s)$ is now the closest obstacle of n as well. Also, n is inserted into the priority queue to propagate the lower wavefront (lines 42–44). To avoid superfluous raise wavefronts where they would overlap with lower wavefronts, the condition in line 41 can be extended to also overwrite cells with equal distance that refer to a deleted obstacle. With this modification the line reads

if $d < D(n) \vee (d = D(n) \wedge \neg\mathrm{isOcc}(obst(n)))$ **then**.

The lines 28 and 45 are hooks to incrementally update a Voronoi diagram on the fly during the update of the distance map (see Sect. 5). If only distance maps are required, these lines can be omitted.

4.2. Implementation Details

The distance map algorithm described above computes and compares real-valued Euclidean distances stored in $D(s)$. As previously done by Scherer et al. [14] and others, we resort to integer squared distances in practice which saves the computational expenses for the square-root. Due to the strict monotony of the square root function for positive inputs, this does not change the behavior of the algorithm.

A central data structure in our algorithm is the sorted priority queue *open*. Such queues are often implemented using search on a binary tree, which yields a complexity of $O(\log n)$ for the insert operation where n is the number of enqueued elements. Since the processing of cells is ordered by distances and cells only enqueue their direct neighbors, many elements in the priority queue have identical distance values. We exploit this by implementing the queue using the bucketing technique presented by Cuisenaire and Macq [15]. It pools cells with the same distance in unsorted lists and keeps track of the next non-empty container. Thereby it reduces the insertion costs from $O(\log n)$ to $O(1)$.

Algorithm 3 Improved expansion of neighbors for lower wavefronts in 3D, replaces line 38 in Alg. 2

```
46: $w \leftarrow (s - obst(s))$
47: for all $n \in \mathrm{Adj}_{26}(s)$ do
48:     $\Delta \leftarrow (n - s)$
49:     if $\exists c \in \{x, y, z\} : w_c \cdot \Delta_c < 0$ then continue
```

To implement priority queues with unique entries and increasable priorities, we actually insert the elements whenever they are updated, and carry a Boolean flag *toProcess* for each cell s. It is set to true by insert($open, s, d$) and reverted to false by pop($open$). The function pop($open$) iteratively dequeues elements until it reached an s with $toProcess(s)$ = true, and thus discards duplicated entries.

4.3. Extension to Higher Dimensions

In the context of robotics, distance transforms have mostly been applied to two-dimensional maps. However, for flying robots or manipulators, 3D distance maps are also very appealing. Our dynamic brushfire algorithm can directly be extended to 3D. Obviously, the obstacle locations, the obstacle reference map, and the distance map itself have to be 3D vectors and arrays in this case. Each cell on a three-dimensional grid has 26 neighbors, so Adj_8 is replaced by Adj_{26}.

Depending on the map size and the amount of changes in the environment, maintaining a complete 3D distance map may not be feasible even with an incremental update algorithm. As proposed by Scherer et al. we introduce an upper bound $d_{\max}$ on the distances that we propagate. Whenever the increasing distances in a lower wavefront reach this threshold, the propagation is stopped. The appropriate value for $d_{\max}$ depends on the application and the available computational resources. Our experiment in Sect. 8.3 demonstrates on real data

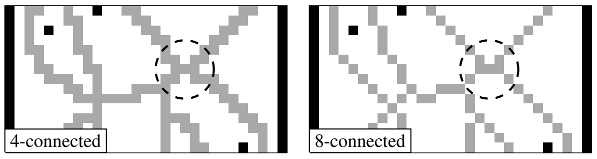

Figure 5: Voronoi diagrams on 4- and 8-connected grids. The 8-connected Voronoi lines (right) might appear thinner on visual inspection, but create interconnections (encircled) with multiple paths. In the 4-connected GVD (left), different paths correspond to topologically different routes with respect to obstacles.

how the choice of $d_{\max}$ influences the required computation time of our algorithm. To implement the distance bound in our algorithm, the condition $d < d_{\max}$ has to be added as an additional requirement in line 41. During initialization, the distance values in the empty cells are set to $d_{\max}$ rather than infinity.

The efficiency of the lower wavefront can be improved by reducing the size of the neighborhood that is expanded for each cell s: the neighbors that lie in the inverse direction of the wavefront's propagation can be skipped. The pseudo-code for this modification is shown in Alg. 3. The direction of the wavefront at s can be determined from $s - obst(s)$ (line 46), and the direction of the potential expansion to neighbor n by $n - s$ (line 48). If these vectors have opposing signs in any component x, y, z, the expansion of n can be skipped (line 49). In theory, this modification could be applied in 2D as well. In this case, however, the computational overhead exceeds the benefit of the reduced cell visits.

5. Dynamic 2D Voronoi Diagrams

In continuous space, a point is part of the Voronoi graph if the distances to its two closest obstacles are identical. For discrete GVDs, this condition cannot directly be applied to the discretized cell coordinates. Instead, the GVD is the set of cells that would contain continuous Voronoi lines in their associated area. Furthermore, the implicit grouping of occupied cells to obstacles plays an important role: treating each occupied cell as a single obstacle would cause the GVD to be cluttered, since a line would be inserted between each pair of adjacent occupied cells. In contrast, treating all connected occupied cells as a single obstacle causes missing Voronoi lines in indoor environments as shown in Fig. 2 (left).

Voronoi graphs in continuous spaces consist of infinitely thin lines and curves. Since GVDs are represented on discretized grids, artifacts in the form of erroneous connections can occur. Firstly, a pair of nearby Voronoi lines that pass through adjacent cells becomes connected and thus creates erroneous circles and interconnections in the graph. Secondly, a single Voronoi line that lies between two discrete cell locations in continuous space causes double lines in the GVD. In both cases, the GVD loses the sparseness property of the Voronoi graph, i.e., the paths in the GVD no longer correspond to topologically different routes with respect to obstacles. When using an 8-connected grid model, the GVD appears to be thinner by visual inspection. However, the additional connections often create additional path variations in adjacent cells. Thus, 8-connected GVDs often violate the sparseness condition, which is not the case for 4-connected ones (see Fig. 5 for an example).

Algorithm 4 Evaluation of the Voronoi condition

```
checkVoro(s, n)
50: if $(D(s) > 1 \lor D(n) > 1) \land obst(n) \neq$ cleared
        $\land\ obst(n) \neq obst(s) \land obst(s) \notin \mathrm{Adj}_8(obst(n))$ then
51:     if $\|s - obst(n)\| \leq \|n - obst(s)\|$ then $voro(s) \leftarrow$ true
52:     if $\|n - obst(s)\| \leq \|s - obst(n)\|$ then $voro(n) \leftarrow$ true
```

We present a set of conditions to generate GVDs that are fully connected, and at the same time have no neighboring Voronoi lines that touch each other, as shown in Fig. 2 (right). Although we focus on 4-connected GVDs, our method can generate 8-connected ones as well. An additional pruning step deals with artifacts due to discretization, i.e., double lines and erroneous connections, and thus ensures the sparseness of the GVD. The algorithm is directly integrated with our method for updating distance maps and is easy to implement.

We represent the GVD by a binary map $voro(s)$, which specifies for each cell s if it is part of the GVD ($voro(s)$ = true) or not ($voro(s)$ = false). The update of the GVD directly integrates with the update of DMs given by Alg. 2 in Sect. 4: lower wavefronts remove all visited cells from the GVD (line 28), and potentially add the cells where they come to a halt. If the lower wavefront propagated by a cell s finds an adjacent cell n whose distance cannot be lowered by adopting $obst(s)$ as closest obstacle, both cells are candidates for the GVD (line 45), and are potentially added after checking our additional conditions in checkVoro(s, n) according to Alg. 4. This function tests if at least one of the neighboring candidate cells s and n is not adjacent to its closest obstacle (line 50). Furthermore, the neighbor n has to have a valid closest obstacle that is different and not adjacent to the closest obstacle of s. If these conditions are fulfilled, a Voronoi line passes be-

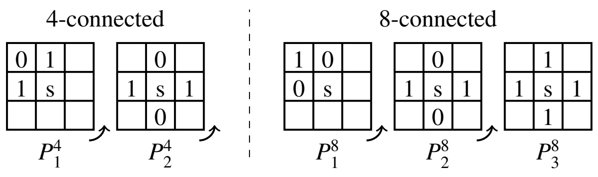

Figure 6: Image operator patterns used to test the connectivity of the GVD. Arrows indicate application of rotated copies.

tween the centers of these cells in sufficient distance to the next one, and both cells are candidates for the discrete GVD.

To avoid double lines, the function only adds the cell $c \in \{s, n\}$ that violates the continuous Voronoi condition to the lesser degree, i.e., the one with the smaller distance increase when switching from its own closest obstacle to the one of the competing neighbor. If both have the same increase, both cells are inserted (lines 51–52). To obtain 8-connected GVDs, the "$\leq$" in these lines are replaced by "$<$" for diagonal neighbors s and n, since then no cells need to be inserted in the case of equal increase.

5.1. Pruning

As discussed before, different paths on the Voronoi graph correspond to topologically different routes in the environment. To preserve this property for GVDs on grid maps, thin Voronoi lines, i.e., being one pixel wide, are desired. Previous work on dynamic GVDs by Kalra et al. however regularly generates Voronoi lines that are two or three pixels wide. Our optional pruning step erodes 2-pixel-wide Voronoi lines that occur where a continuous Voronoi line would pass exactly between two cells. Therefore, all new Voronoi cells are inserted into a priority queue and processed by the pruning stage.

The image operator patterns shown in Fig. 6 match whenever the center cell s provides connectivity for one or more of its adjacent cells. The left side shows the two patterns required for ensuring 4-connectedness, the three patterns on the right side correspond to 8-connectedness. In any pattern, a "1" matches $voro(s) =$ true, while "0" stands for $voro(s) =$ false, and empty fields are ignored. Where indicated by arrows, the same pattern is applied in all unique 90 degree rotations.

In a first phase, the pruning algorithm merges Voronoi lines that are erroneously connected due to the finite map resolution. This is done using the matching pattern P_3^8, which detects cells that are enclosed by Voronoi cells. If such a cell is free and not part of the GVD, it is added at this point. This merges Voronoi lines that are too close to be separated given the map resolution. Together with the following pruning step, this prevents

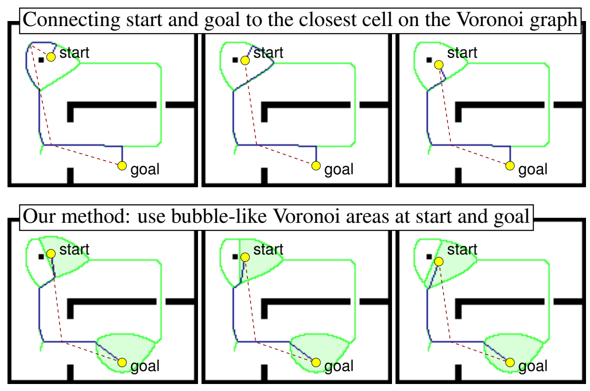
Connecting start and goal to the closest cell on the Voronoi graph
start
goal
Our method: use bubble-like Voronoi areas at start and goal

Figure 7: Connecting start and goal to the Voronoi graph (green) during planning: using the shortest connection (top), the planned path (blue) can change abruptly for small changes of the start configuration, even for sightline-pruned paths (dashed). We create Voronoi bubbles around start and goal, and use goal-directed search therein, which yields more stable paths (bottom).

erroneous connections and ensures that the generated GVD is sparse.

The second phase implements the actual pruning step. In increasing order of distance, the enqueued cells are iteratively popped from the priority queue. If such a cell has more than one neighbor on the GVD and is not required to keep the GVD connected, it can be removed from the GVD without affecting its topology. Again, this is tested using the image operator patterns: if none of the connectivity patterns match at the cell location, the cell is not required in the GVD.

5.2. Path Planning on Voronoi Diagrams

As mentioned before, a GVD is a cell decomposition method that is appealing for path planning. This section details on important aspects of Voronoi planning in dynamic environments. In general, the start and goal locations of a planning problem are not part of the GVD. Straight-forward approaches search for the closest Voronoi cell at both locations, and connect them with straight lines to the graph [39]. This is problematic in practice, since a small change of the start pose can substantially change the planned path as shown in Fig. 7 (top row). Starting a goal-driven search instead can easily expand a big part of the space that is not on the GVD.

Our approach given by Alg. 5 overcomes these problems. First, we insert virtual obstacles at the start and goal location (lines 54–55). After updating the distance map and the GVD with our incremental algorithm (line 56), these locations become enclosed by Voronoi lines that form "bubble"-like areas as shown in Fig. 7 (bottom row). With a simple brushfire expansion, we

Algorithm 5 "Bubble"-technique for path planning on a GVD

```
planPath(start, goal)
53: if $M(start) = 1 \lor M(goal) = 1$ then return
54: setObstacle(start)                  // create Voronoi bubbles
55: setObstacle(goal)
56: updateDistanceMap()
57: brushfireMark(start)                // mark bubbles as searchable
58: brushfireMark(goal)
59: push(astarqueue, start)
60: while $astarqueue \neq \emptyset$ do          // A* on Voronoi and bubbles
61:     $s \leftarrow$ pop(astarqueue)
62:     if $s = goal$ then
63:         removeObstacle(start)
64:         removeObstacle(goal)
65:         updateDistanceMap()
66:         return path from start to goal
67:     for all $n \in \mathrm{Adj}_4(s)$ do
68:         if $voro(n) = \text{true} \lor marked(n) = \text{true}$ then
69:             A* update for costs and heuristic of n
70:             push(astarqueue, n)

brushfireMark(s)
71: push(unsortedqueue, s)
72: while $unsortedqueue \neq \emptyset$ do
73:     $s \leftarrow$ pop(unsortedqueue)
74:     $marked(s) = \text{true}$
75:     for all $n \in \mathrm{Adj}_4(s)$ do
76:         if $M(n) = 1$ then continue
77:         if $voro(n) = \text{false} \land marked(n) = \text{false}$ then
78:             push(unsortedqueue, n)
```

mark all cells in the bubbles up to the enclosing Voronoi lines (lines 57–58). Now we can start a goal-directed search that is restricted to cells that are either marked or belong to the GVD. In this way, the search expands from the start onto the Voronoi graph, follows Voronoi lines, and then connects to the goal when reaching the goal bubble. Since the whole path is the result of goal-directed graph search, the consecutive paths planned for a moving robot are very similar to each other and do not change abruptly (see Fig. 7). After the shortest path is computed, we undo the changes to the GVD by removing the virtual obstacles and performing another update (lines 63–65).

In order for the algorithm to be truly incremental, the markers should be stored in a hash map structure rather than in a binary grid that has to be cleared after each frame. Additionally, one can employ an incremental replanning algorithm like D* rather than A*.

6. Dynamic C-Space Collision Maps

As discussed in Sect. 2.3, a configuration space (c-space) grid map encodes for each discretized configuration of a robot whether it causes a collision with obstacles in the environment or not. Non-circular robots moving on a plane have a three-dimensional c-space, since their poses $\langle x, y, \theta \rangle$ are given by their 2D position on the ground and their orientation θ.

Computing a c-space map usually requires convolving a map with the shape of the robot for each orientation. Recomputing these convolutions to reflect changes in dynamic environments is often not feasible at frame rates required for online applications. This section presents a method to efficiently update c-space collision maps with the obstacle models shown in Fig. 1. For the sake of clarity, we first describe our algorithm for a 2.5D representation with overhanging obstacles (bottom-right), and discuss the adaptation to other obstacle models later.

Let $M(x, y)$ be a grid map that represents the vertical clearance, i.e., the height of free space above the floor, with zeros for completely occupied cells. Consider a robot moving on the floor with continuous orientation $\tilde{\theta}$ with respect to the map coordinate system. We represent the discretized shape of the robot for a given orientation $\tilde{\theta}$ by a map $S_{\tilde{\theta}}(i, j)$, that stores the height of the robot for every cell of its footprint. $S_{\tilde{\theta}}$ has the same resolution and orientation as the grid map M, whereas its origin $S_{\tilde{\theta}}(0, 0)$ is located at the center of the robot.

A convolution-type conjunction of M and $S_{\tilde{\theta}}$ yields a count map $C_{\tilde{\theta}}(x, y)$ as shown in Fig. 8. Each cell $\langle x, y \rangle$ in $C_{\tilde{\theta}}$ stores the number of cells the robot collides with when located there:

$$C_{\tilde{\theta}}(x, y) = \sum_i \sum_j \mathrm{eval}\{M(x+i, y+j) \leq S_{\tilde{\theta}}(i, j)\} \ , \quad (1)$$

where $\mathrm{eval}(\text{true}) = 1$ and $\mathrm{eval}(\text{false}) = 0$. If we discretize $\tilde{\theta}$ and stack the $C_{\theta}(x, y)$ for all discrete θ, we obtain the robot's c-space collision count map $C(x, y, \theta)$ for M. Clearly, by testing $C(x, y, \theta) > 0$ we can check if the discretized pose $\langle x, y, \theta \rangle$ is colliding. By storing collision counts instead of just binary values as in regular c-space maps, we can update the c-space map incrementally as described below.

6.1. Incremental Update of the C-space Map

Unknown or moving obstacles cause changes in the environmental representation of a robot. For the 2.5D obstacle model, a change is given by an updated vertical clearance v_{new} for a cell $\langle x, y \rangle$ in M. To refresh C incrementally rather than computing it from scratch, we

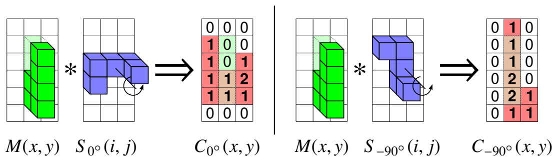

Figure 8: Convolving a map $M(x,y)$ with a representation of the robot's shape $S_\theta(i,j)$ for a given orientation θ yields a collision map $C_\theta(x,y)$, according to Eq. (1). Each cell $\langle x,y\rangle$ in C_θ counts the cells in the robot footprint that collide with occupied cells in M, given the robot is at pose $\langle x,y,\theta\rangle$.

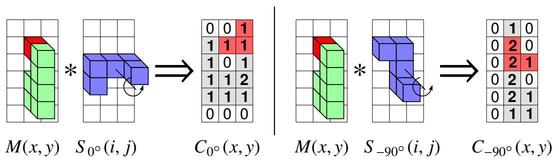

Figure 9: A newly occupied cell in the map M (red) increments the collision count C for all robot poses that cause a collision at the location of the new obstacle. In this way, the collision map is updated (red cells) without recomputing the values for unaffected (gray) cells. Alg. 6 implements this procedure as well as the corresponding case for newly emptied cells.

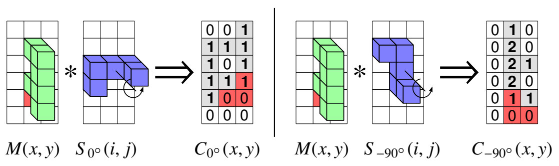

Figure 10: A newly vacated cell in the map M (red) decrements the collision count C for all robot poses that caused a collision at the location of the former obstacle. The coordinates of cells whose collision count reaches zero are configurations that become collision free.

Algorithm 6 Dynamic update of C-space collision maps

```
updateVerticalClearance($x, y, v_{\text{new}}$)
79: $v_{\text{old}} \leftarrow M(x,y)$
80: $M(x,y) \leftarrow v_{\text{new}}$
81: for all $\theta$ do
82:     for all $\langle x',y'\rangle \in \{\langle x-i, y-j\rangle \mid S_\theta(i,j) > 0\}$ do
83:         if $v_{\text{new}} \le S_\theta(i,j) \wedge v_{\text{old}} > S_\theta(i,j)$ then
84:             $C(x',y',\theta) \leftarrow C(x',y',\theta) + 1$
85:             if $C(x',y',\theta) = 1$ then newOccupied($x',y',\theta$)
86:         else if $v_{\text{new}} > S_\theta(i,j) \wedge v_{\text{old}} \le S_\theta(i,j)$ then
87:             $C(x',y',\theta) \leftarrow C(x',y',\theta) - 1$
88:             if $C(x',y',\theta) = 0$ then newEmpty($x',y',\theta$)
```

only update the affected parts of the sum in Eq. (1) according to Alg. 6. See the sequence of Figs. 8-10 for an illustration.

The algorithm separately updates the θ-layers of C, and can thus be parallelized (line 81). For each cell $\langle i,j\rangle$ of the robot shape $S_\theta(i,j)$ we visit the robot position $\langle x',y'\rangle$ that lets $\langle i,j\rangle$ fall on $\langle x,y\rangle$ (line 82). These cells can efficiently be selected using standard drawing algorithms for rasterized images.

If the new vertical clearance v_{new} in $\langle x,y\rangle$ causes a collision with $S_\theta(i,j)$ while v_{old} did not, the collision counter of $\langle x',y'\rangle$ is incremented (line 83), since this represents a new collision candidate cell. Vice versa, if v_{new} is collision-free whereas v_{old} collided, the counter is decremented (line 86), since a collision candidate was removed. Whenever the count changes from 0 to 1 or from 1 to 0, the pose $\langle x',y',\theta\rangle$ is newly occupied (line 85) or emptied (line 88), respectively. These events can be used to trigger further computation, e.g., to update the c-space distance map and Voronoi diagram discussed in Sect. 7.

6.2. *Discretization of Orientations*

An appropriate discretization of $\tilde{\theta}$ ensures that if two adjacent poses $\langle x,y,\theta_i\rangle$ and $\langle x,y,\theta_{i+1}\rangle$ are collision-free according to C, intermediate orientations in $[\theta_i,\theta_{i+1}]$ are collision-free as well. Under this constraint we seek to discretize $\tilde{\theta}$ as coarse as possible to keep the number of θ-layers in C small.

In occupancy grid maps, the actual location of obstacles can be anywhere in the cells they occupy. Therefore, one usually assumes an additional safety margin m around the robot, e.g., of $m = 1$ pixel unit. Given this margin, we can formulate a bound on the angular resolution for the discretization of $\tilde{\theta}$ as follows: if the robot rotates from θ_i to θ_{i+1}, each point on the robot moves along an arc. The maximum arc length occurs at the outmost point of the robot, which is the radius r (in pixels) of the circumcircle around its center of rotation. By choosing a resolution of $|\theta_i - \theta_{i+1}| = m/r$, we ensure that even in the worst case an obstacle collides only with the safety margin but not with the actual robot. Depending on the shape of the robot, less conservative bounds on the discretization can be formulated.

6.3. Adaptation to Other Obstacle and Robot Models

Up to this point, we assumed overhanging obstacles and a robot on the floor that can move underneath obstacles as in Fig. 1 (bottom-right). By reversing the comparisons of robot height and vertical clearance in Eq. (1) and Alg. 6 (lines 83 and 86), this can easily be adapted to obstacles elevating from the floor and robots with overhanging load or parts as in Fig. 1 (bottom-left). For plain 2D robot and obstacle models, the heights v_{new} and v_{old} are binary values that encode occupied (true) and free (false). In that case, the conditions for determining newly occupied cells in line 83 are given by

"**if** $v_{\text{new}} = \text{true} \wedge v_{\text{old}} = \text{false}$ **then**",

and for newly vacated cells in line 86 by

"**if** $v_{\text{new}} = \text{false} \wedge v_{\text{old}} = \text{true}$ **then**".

For some applications, the obstacles and the robot have to be represented in full 3D as in Fig. 1 (top-right). The height comparisons in Alg. 6, lines 83 and 86 then have to consider lists of obstacle heights. If the robot shape is approximated by a set of vertical columns with a given upper and lower end as in Fig. 1, one can also use a separate shape map for each column. If line 82 is adapted to only consider the columns that potentially collide with a given new obstacle, the c-space collision map can be efficiently updated.

In applications like mobile manipulation or autonomous transport, the shape of the robot and its payload can vary over time. To use our method in these cases, one can group the changing parts of the robot to additional models, and maintain separate c-space maps for them. Then, one can immediately switch between different configurations of the robot by using different sets of models in the collision check. If highly accurate collision checks are required, one can also create shape models for an inner and an outer approximation. Only if the outer (larger) approximation collides while the inner (smaller) one does not, more complex methods like mesh queries are required. Otherwise, the approximations are sufficient.

Robots with a symmetric shape with respect to their center cause a part of the θ-layers in $C(x, y, \theta)$ to be redundant. For example, a rectangular robot rotating around its center causes the same c-space obstacles at orientation 180° as at 0°. Omitting the respective layers when iterating over θ in Alg. 6 (line 81) saves a substantial part of the computational effort and memory consumption.

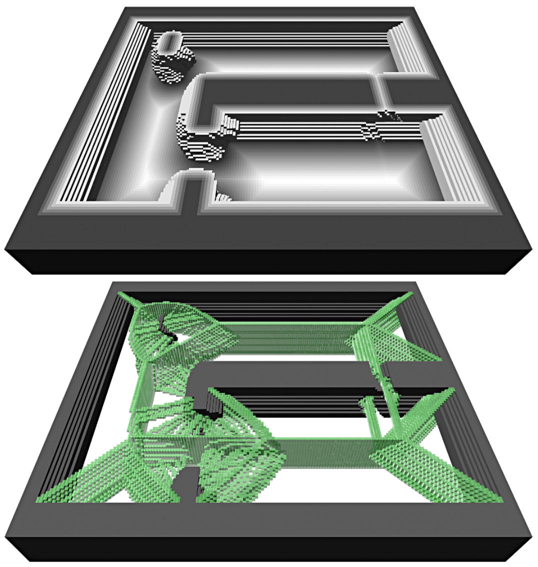
Figure 11: C-space distance map (top) and Voronoi diagram (bottom) for a rectangular robot, both obtained by stacking layers computed in 2D for different robot orientations θ. For readability, only half of the layers are shown, the other half is identical due to the symmetry of the robot. In the visualization at the top, cells above the bottom layer have a different color scaling and were removed when exceeding a distance threshold.

7. C-Space Distance Maps and Voronoi Diagrams

This section describes how to compute DMs and GVDs presented in Sects. 4 and 5 for the c-space collision map in Sect. 6. Since we provide incremental update algorithms for all these representations, these combinations are suitable for online applications as well, and thus open new possibilities for collision checking and path planning in the configuration space of mobile robots.

Given a three-dimensional c-space collision map $C(x, y, \theta)$ as defined in the previous section, we can compute a 3D distance map in this space that uses a 3D distance measure which combines Euclidean distances and the angle of rotation. Therefore, one has to consider the angle wrap-around of the θ component for the expansion of neighborhoods. The resolution of the θ discretization specifies how to balance Cartesian and angular distances. With the methods presented in Sect. 4.3, such a c-space DM can be updated incrementally, and can for example be used to efficiently perform collision checks for non-circular robots on long trajectories.

As discussed by Canny [40], it is also appealing to only consider 2D Euclidean distances per θ-layer of the c-space map. Therefore, we stack Euclidean distance maps $D_\theta(x, y)$ computed for every c-space map layer $C_\theta(x, y)$, yielding the c-space distance map $D(x, y, \theta)$ as shown in Fig. 11 (top).

In 2D, GVDs are the union of points whose two closest obstacles are at the same distance. Just as for the DMs, we compute a GVD $voro_\theta(x, y)$ for every c-space map layer $C_\theta(x, y)$. Stacking these Voronoi diagrams results in a c-space Voronoi diagram $voro(x, y, \theta)$ as shown in Fig. 11 (bottom), which is fundamentally different from computing the 3D generalized Voronoi diagram for $C(x, y, \theta)$. If θ is discretized according to Sect. 6.2, the Voronoi lines in neighboring layers join to connected surfaces.

To update the layers D_θ and $voro_\theta$, we first update the underlying c-space collision map. The events newOccupied(x', y', θ) and newEmpty(x', y', θ) in Alg. 6 are used to call setObstacle(x, y) and removeObstacle(x, y) in the respective θ-layer to register newly occupied or vacated cells. After the update of the c-space map is completed, we call updateDistanceMap() for each θ-layer, which completes the update of the c-space DM and GVD. Since their layers are independent, this can be parallelized on multi-core CPUs.

7.1. C-Space Voronoi Path Planning

Given a layered c-space GVD as described above, one can perform deterministic and complete path planning for non-circular mobile robots without a non-holonomic constraint. The search on the c-space GVD is very similar to 2D Voronoi planning. The major difference is the added dimension of the orientation with its cyclic nature, which has to be considered in the neighborhoods during expansion. The bubble planning method presented in Sect. 5.2 can easily be adapted for these purposes. Therefore, the creation and brushfire expansion of the Voronoi bubbles (lines 54–58) and the corresponding removal (lines 63–65) have to be executed for each θ-layer independently, which can be parallelized on multi-core CPUs. Obviously, the A* algorithm has also to be modified to search the subspace of the three-dimensional c-space, which is given by the GVD and the start/goal bubbles.

When using 8-connected GVDs, the brushfire expansion has to be run using a 4-connected neighborhood to ensure that the expansion is contained in the start and goal bubbles. If the GVD is 4-connected as in our examples, the brushfire expansion may also use 8-connectedness.

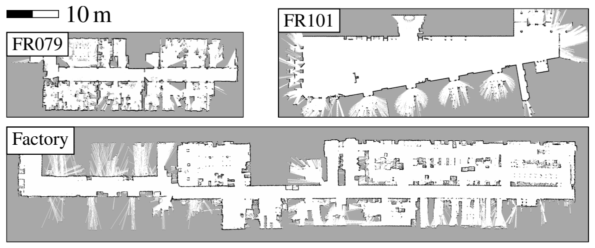

Figure 12: Maps of the environments where our experiments on 2D laser range data were carried out.

8. Experiments

This section presents experiments conducted to test our algorithms on real-world data. We analyze the computational requirements and show examples of generated output. The tests were done using our C++ implementation of the algorithms, running on an Intel® Core™ i7 2670 MHz. The source code and a big part of the employed datasets are available online [1].

8.1. 2D DMs and GVDs in Dynamic Environments

For this set of experiments we used a Pioneer robot equipped with a SICK LMS291 laser range finder. To record data, it was moving in environments where walking people heavily affected the traversable space (see Fig. 12). The sequence "FR079" consists of 369 frames recorded in an office building, and "FR101" contains 400 frames recorded in a large foyer space. The update radius around the robot was only limited by the maximum range of the laser scanner (80 m). Due to the maximum room size in the environments, the maximum closest obstacle distance in these two maps, i.e., the radius of the largest circular unoccupied area, is 29 cells (1.45 m) and 97 cells (4.85 m), respectively. The 400 frames of "Factory" were simulated by randomly inserting 200 obstacles per frame into a grid map of a large factory floor with a maximum obstacle distance of 44 cells (2.2 m). Similar to Kalra et al. [11], the random obstacles were placed within 5 m radius around a moving center, which simulates a moving observer with limited perception.

To demonstrate the computational benefit of dynamically updatable DMs, we compared our algorithm with state-of-the-art static methods implemented by Fabbri et al. [9], namely the algorithms by Cuisenaire and Macq [15] and Maurer et al. [41]. These approaches

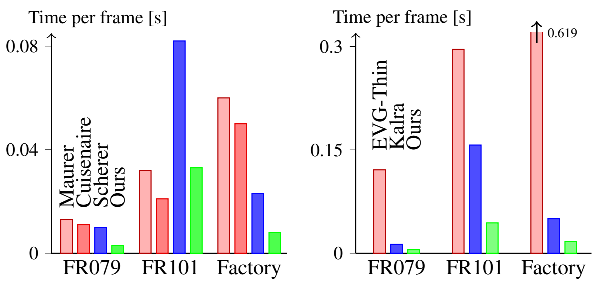

Figure 13: Performance of our algorithms for updating distance maps (left) and Voronoi diagrams (right) compared to related work. The plots show the average computation time per frame.

Table 1: Update performance of distance maps and Voronoi diagrams

			Time per frame [s]			Cell visits per frame		
Map & Approach			mean	min	max	mean	min	max
Distance Maps	FR079	Maurer	0.013	0.013	0.013	1,393,286	1,392,450	1,394,272
		Cuisenaire	0.011	0.010	0.011	302,513	301,022	304,138
		*Scherer	0.010	0.006	0.019	250,403	77,277	657,903
		*Ours	0.003	0.001	0.005	99,761	29,340	190,998
	FR101	Maurer	0.032	0.032	0.033	3,299,105	3,296,201	3,302,497
		Cuisenaire	0.021	0.021	0.021	572,345	562,474	581,158
		*Scherer	0.082	0.026	0.148	3,338,297	792,054	6,190,219
		*Ours	0.033	0.011	0.051	1,264,488	427,176	1,929,690
	Factory	Maurer	0.060	0.060	0.060	5,954,325	5,954,259	5,954,447
		Cuisenaire	0.050	0.050	0.052	959,484	957,735	961,709
		*Scherer	0.023	0.003	0.032	976,292	80,262	1,307,630
		*Ours	0.008	0.002	0.011	319,871	80,262	423,315
Voronoi Diagrams	FR079	EVG-Thin	0.121	0.120	0.122	10,030,438	10,001,973	10,059,575
		*Kalra	0.013	0.006	0.030	483,242	281,126	933,410
		*Ours	0.005	0.003	0.008	113,803	31,824	215,131
	FR101	EVG-Thin	0.296	0.282	0.310	19,892,173	19,798,905	20,005,447
		*Kalra	0.157	0.046	0.284	4,363,678	1,537,112	7,558,395
		*Ours	0.044	0.018	0.066	1,372,060	475,163	2,087,083
	Factory	EVG-Thin	0.619	0.592	0.632	35,540,331	35,525,379	35,551,489
		*Kalra	0.050	0.005	0.068	1,462,930	167,553	1,970,182
		*Ours	0.017	0.009	0.021	391,555	113,889	515,343

*dynamic method that only updates the affected parts of the map in each frame

are highly efficient, but recompute the whole distance map in every frame. We further compared our method to the recent approach by Scherer et al. [14]. Since no source code was available, we implemented this algorithm in C++ with the assistance of Scherer. Our GVD approach is compared to the static EVG-Thin method implemented by Beeson [42] and the dynamic approach by Kalra et al. [11].

In the first frame of each sequence, the algorithms were initialized with the corresponding grid map shown in Fig. 12, using a resolution of 0.05 m per grid cell. The performance is visualized in Fig. 13 and presented by numeric results in Tab. 1. For each sequence, the table provides the computation time and cell visits per frame with their mean, minimum, and maximum values. When repeating the measurements 10 times for each sequence, the standard deviations between the runs were well below 1% of the reported means.

In general, the dynamic methods are considerably faster than the static approaches by Cuisenaire and Macq and Maurer et al., except for the distance maps in the open space of FR101, where most updates affect a large fraction of the map. In all frames of all sequences, our dynamic distance map algorithm visits 60−70% fewer cells and requires 60−70% less computation time than the dynamic approach by Scherer et al. [14]. This can be mainly attributed to the raise function of their algorithm which expands the adjacent cells of the neighbors of a cell s, whereas our algorithm tests only the direct neighbors (line 38). Note that the cell visits performed by the static methods are not directly comparable to the dynamic ones due to the different amount of computation per visit.

The comparison of the GVD update algorithms shows similar results: the dynamic methods clearly outperform the static method EVG-Thin in all tested environments. In addition, our approach can reduce the runtime considerably compared to the previous dynamic approach by Kalra et al. [11], since this method uses the same update strategy as Scherer's approach for DMs and thus visits more cells. In all tested environments, the average frame rate achieved by our approaches was well above 20 fps which allows for online application of both distance maps and GVDs.

The distance maps generated by our method are equal to the ones generated by the compared methods, up to the inherent overestimation errors of 0.09 pixel units, as discussed in Sect. 2.

Exemplary outputs of our 4-connected GVD algorithm and the method by Kalra et al. [11] are shown in Fig. 2: while Kalra's GVD misses Voronoi lines inside rooms and corridors, our approach captures the connectivity of the floor plan completely. Furthermore, our method generates thin Voronoi lines, such that different paths between a given start and goal are topologically different with respect to obstacles.

8.2. 2D DMs and GVDs during SLAM

To demonstrate the suitability of our incrementally updatable DMs and GVDs for SLAM applications, we used the GMapping SLAM package [43, 44] to construct maps of the datasets "intel_lab" (indoor) and "fr_campus" (outdoor) [1], with a resolution of 0.05 m and 0.2 m per cell, respectively. We set the parameters to integrate a new scan after the robot has moved 0.5 m or rotated 0.5 radians, and used 100 particles. After each integration step, we use the map associated with the current best particle to determine the newly occupied and

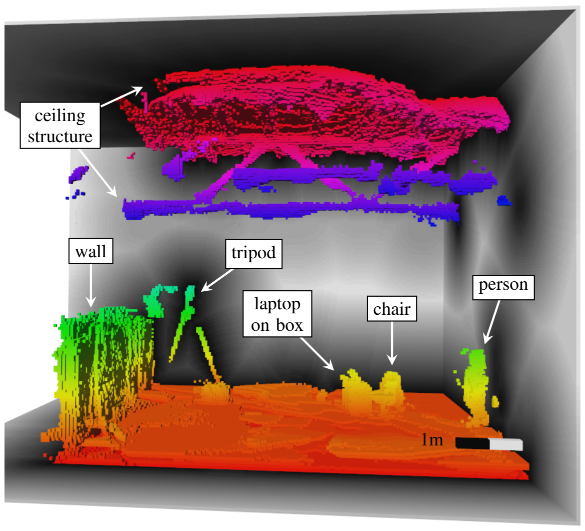

Figure 14: Excerpt of the 3D map of the hall used for our experiments, showing roughly 1/4 of the space. The color encodes the height of each cell over ground. The gray planes are slices from the 3D distance map of that space. Here, brighter cells denote increasing Euclidean distance from obstacles.

Table 2: Performance of incremental 3D distance map updates

Data set		$d_{\max}$	avg affected cells	avg time update
Hall, moving: 693 frames		0.5 m	152,613	0.0734 s
size: 14.25×14.2×7.45 m^3		1.0 m	267,097	0.1493 s
total cells	12,226,500	1.5 m	354,061	0.2174 s
avg flipped	347.0	2.0 m	396,937	0.2532 s
Hall, static: 1,443 frames		0.5 m	93,460	0.0482 s
size: 14.25×14.2×7.45 m^3		1.0 m	180,178	0.1034 s
total cells	12,226,500	1.5 m	245,062	0.1479 s
avg flipped	214.8	2.0 m	285,867	0.1782 s
Lab, static: 1,618 frames		0.5 m	41,760	0.0231 s
size: 6.0×5.15×3.15 m^3		1.0 m	48,231	0.0269 s
total cells	798,720	1.5 m	48,231	0.0275 s
avg flipped	164.7	2.0 m	48,231	0.0275 s

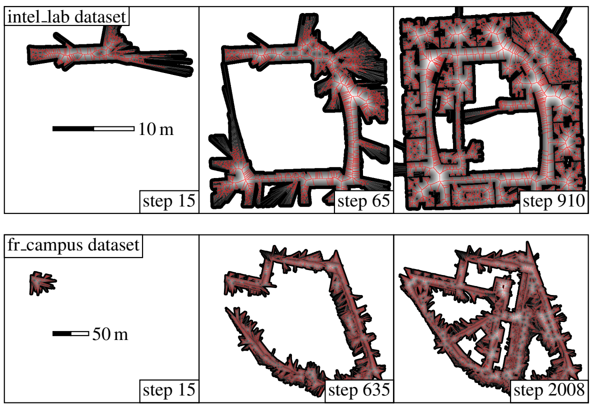

Figure 15: Incremental construction of a distance map and a Voronoi diagram during SLAM for two datasets. Pure local mapping only causes changes if new parts of the map are uncovered. Loop-closures, however, can affect large parts of the map, since individual parts often move with respect to the map coordinate system.

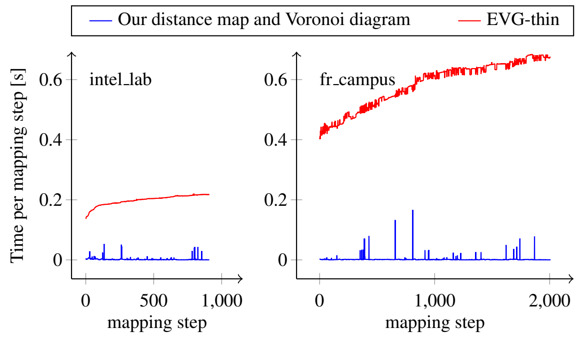

Figure 16: Computation time required to perform the updates between each mapping step for the datasets shown in Fig. 15. Our incremental methods clearly outperform the computation of the full GVD in every step as done by EVG-thin. The peaks in the computation time of our method correspond to loop closures in the SLAM process.

freed cells with respect to the previous step. Based on these changes, we incrementally update a DM and GVD in every step as shown in Fig. 15.

The computation time taken by our algorithm to perform these updates is shown in Fig. 16. For comparison, we also plot the computation required by EVG-thin to compute a GVD approximation for the grid map of each step.

Since the maps are incrementally growing during the mapping process, the computational effort for EVG-thin grows with increasing step numbers. The dynamic updates of our algorithm are substantially faster, since the number of changes between consecutive SLAM steps is rather small. The curve shows peaks of increased effort whenever the best particle changes, for example after closing a loop. In our experiments, the average computation time required to update the DM and GVD per mapping step was below 0.002 s and never exceeded 0.2 s even after loop-closures, which allows frame rates above 5 fps.

8.3. Three-dimensional DMs

For the experiments with three-dimensional DMs we recorded 3D maps in a lab room and in a large hall using the depth measurements obtained from a Microsoft Kinect 3D camera (see Fig. 14). The camera poses were determined using a MotionAnalysis motion capture system with 9 Raptor-E cameras.

The maps were constructed and stored with a resolution of 0.05 m per voxel using the probabilistic occupancy mapping routines in the OctoMap software package [45]. The map sizes and total number of cells are given in the left column of Tab. 2.

After constructing the maps, we recorded three 3D sequences of two walking people and continuously up-

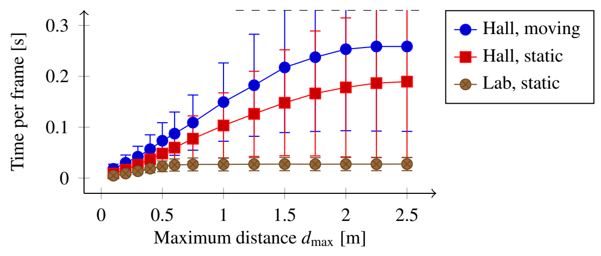

Figure 17: Performance of incremental 3D distance map updates. The plot shows the average computation time per frame for different sequences, depending on the maximum propagated distance $d_{\max}$.

dated the pre-recorded maps using the same techniques. When integrating a point cloud into the 3D map, the OctoMap library can return a list of the cells that switched from free to occupied or vice versa. In each frame, we register any flipped cell s using setObstacle(s) or removeObstacle(s), and then update the 3D distance map as described in Sect. 4.

The number of frames per sequence and the average number of flipped cells per frame are given in the left column of Tab. 2. The resulting average of affected distance map cells per frame and the required average computation time is shown in the right column. In one sequence, the camera was manually moved to follow a person, in the others it was static. An excerpt of the 3D map of the hall and 2D slices of the corresponding 3D distance maps are shown for an example frame in Fig. 14.

The average computation time per frame, depending on the maximum propagated distance $d_{\max}$, is shown in Tab. 2 and the plot in Fig. 17. The error bars show the standard deviation obtained by averaging over the frames. These values are high, since the computational requirements depend on the amount of changes in the environment, which varies over time.

The average computation time per frame grows with an increasing distance limit $d_{\max}$, up to the point where the update radius is naturally limited by the maximum distances found in the environment.

For applications like path planning, collision avoidance or localization, a distance limit $d_{\max}$ between 0.5 m and 1.0 m is mostly sufficient. The maximum computation times per frame under these conditions were achieved in the large hall and corresponded to frame rates between 2 and 5 fps, with average frame rates between 4 and 13 fps. For the smaller lab environment, the update rates do not fall below 10 fps. Note that these numbers consider the computation time required by our algorithms, but not the occupancy mapping by the OctoMap.

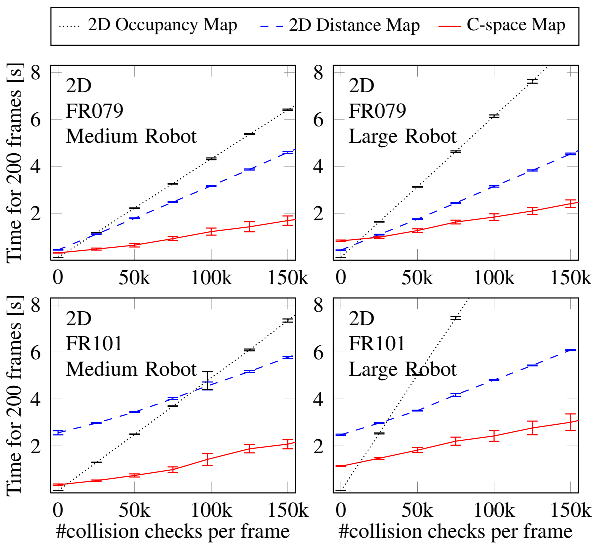

Figure 18: Computation time for different collision checking routines for two sequences and two robot models. The update required in every frame for the c-space collision map pays off starting from 10,000 collision checks per frame. The plots show mean and standard deviations averaged over 10 runs.

8.4. *C-space Obstacle Maps and Collision Checks*

This section benchmarks our incrementally updatable c-space representations on the 2D laser data sequences described in Sect. 8.1. For the updates, the maximum range of the laser scanner was limited to 5 m. To simulate 2.5D and 3D obstacles, we augmented the laser data with random height values between 0 m and the robot height.

In 2D, we assumed a medium sized rectangular robot (0.85x0.45 m) and a large one (1.75x0.85 m). In 2.5D, we modeled a wheelchair with a low front and a high rear part, as in Fig. 1 (bottom-right). In 3D, the robot was modeled like a Willow Garage PR2, with a frontal extension for the base and the fixed arms (see Fig. 1 top-right). To speed up our algorithms, we used OpenMP for parallelization with up to 6 threads.

The c-space collision map presented in Sect. 6 requires computation of the incremental update in every time step, but then, each collision check for the whole robot only requires a single map lookup. In the 2D model, we exploit the symmetry of the rectangular robot as described in Sect. 6.3.

We compare our method to a previous collision checking approach for rectangular robots that uses recursive distance queries on incrementally updatable 2D distance maps [46]. As a baseline, we also evaluate a straight-forward approach that checks every cell of the robot's footprint for collision using an up-to-date 2D oc-

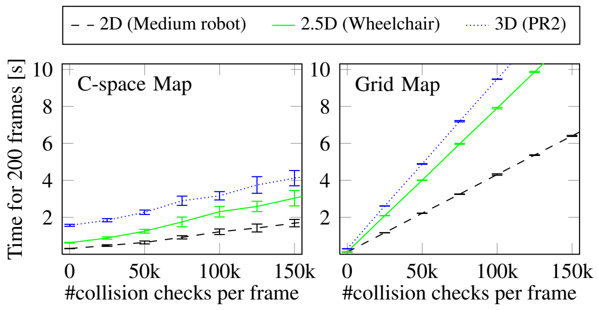

Figure 19: Collision check performance for different robot and obstacle models, using our updatable c-space collision map (left) vs. the straight-forward occupancy grid map approach (right). The costs for updating the c-space map are remedied by the faster collision checks for 10,000 checks or more per frame. The plots show mean and standard deviations averaged over 20 runs.

cupancy map.

The results of this benchmark are shown in Fig. 18. The time required for updating the distance and c-space maps is shown by the first data point of each plot (zero collision checks). The slopes of the curves depend on the cost per collision check. In contrast to the distance map approach, the update time for the c-space map grows with the size of the robot (right vs. left column), but does not suffer from the open area in FR101 (bottom vs. top row). The update for the c-space collision map pays off for 10,000 or more collision checks, which can easily be required during path planning or trajectory optimization. In comparison, the break-even point for a single disk-shaped object was at 22,400 for the disk-decomposition method by Ziegler and Stiller, and $5 \cdot 10^6$ for the full c-space [33].

We repeat the experiment, but with 2.5D and 3D obstacles and robots this time. Compared to the 2D rectangular robot (dashed), the costs for the c-space update with 2.5D and 3D are higher, since the robots are not symmetric anymore and consist of two and three parts, respectively, see Fig. 19 (left). However, the costs per collision check (slope of the plots) are the same, as opposed to the curves for the straight-forward occupancy grid map approach (right).

In all cases, the update of the c-space map takes less than 15 ms per frame. Performing 150,000 collision checks per frame additionally requires at most another 15 ms. This corresponds to $10 \cdot 10^6$ collision checks per second for arbitrary robot shapes, which clearly outperforms even modern GPU-based approaches with $0.5 \cdot 10^6$ collision checks per second for simple polygons [24].

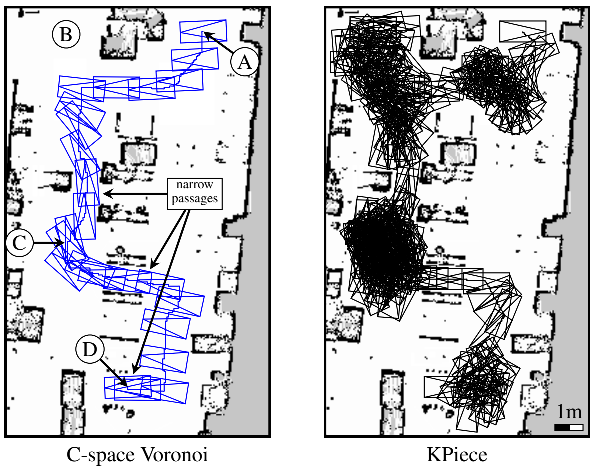

Figure 20: Map of a factory floor (9.5x15.4 m) with start location (A) and three goals (B), (C), and (D). Example paths from (A) to (D) are shown for two different planners. The sampling-based planner (right) is challenged by narrow passages, while the performance of the Voronoi planner (left) is unaffected.

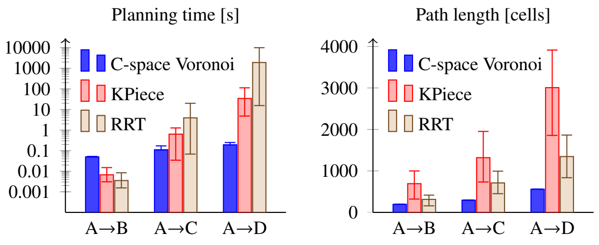

Figure 21: Planning time and path length for three planners and the three planning tasks in Fig. 20. The plot shows mean and min/max for 20 runs. In contrast to the Voronoi planner, the sampling-based planners require several orders of magnitude more planning time for each narrow passage in the path.

8.5. Path Planning using C-space Voronoi Maps

The c-space Voronoi maps presented in this paper provide means for complete grid map planning for non-circular omnidirectional robots using standard graph search algorithms like A* or D* Lite [36]. With our algorithms for incremental updates they are applicable in dynamic environments. This experiment uses the Voronoi bubble technique proposed in Sect. 5.2.

We use A* to plan paths for the large robot model (see above) on the grid map of the factory floor shown in Fig. 20. The start pose is given by (A), and three possible goal poses by (B), (C), and (D). Each of the consecutive goals requires traversing another narrow passage. For comparison, we test our method against the KPiece and RRT implementations available in the Open Motion Planning Library [47]. All planners use our c-space map for collision checking, thus the performance differences are due to the tested planner.

The average resulting planning times and path lengths for 20 runs per start-goal combination are shown in Fig. 21. Each additional narrow passage requires several orders of magnitude more planning time for the sampling-based planners, while the time taken by the Voronoi planner grows roughly linearly with the path length. Using per-cell collision checking rather than the c-space collision maps for the sampling based planners increases the computation times by a factor of 3.

As another application example, we plan the path of a PR2 robot using a c-space Voronoi map generated from real 3D point cloud data (see Fig. 22). After a precomputation phase of 0.5 s, planning a path on the incrementally updatable c-space Voronoi map takes less than 2.5 ms.

Clearly, Voronoi planning is of advantage in narrow areas as long as the grid resolution is fine enough. Our incrementally updatable c-space Voronoi representation allows to apply this idea to non-circular robots in dynamic environments, and could also be used in Voronoi sampling routines of other path planners [38].

9. Conclusion

In this paper we presented incremental algorithms to update distance maps, Voronoi diagrams, and configuration-space collision maps. These representations are initialized using a given grid map or point cloud. For efficient online operation, our methods only update cells that are affected by changes in the environment. Thus, they can be used in real-world scenarios with unexpected or moving obstacles for applications like SLAM, path planning, collision avoidance, or localization.

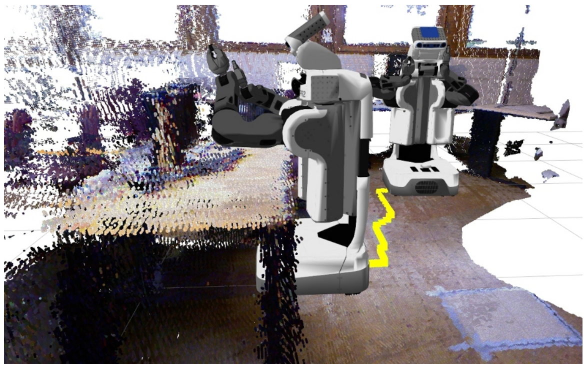
Figure 22: Table-docking with a PR2 robot in a 3D map using Voronoi planning. The yellow line shows the planned path, the rendered robots denote the start and goal pose. Note that the goal pose requires 3D collision checking, since the table overlaps with the robot's footprint.

Compared to previous approaches, our methods for updating two-dimensional Euclidean distance maps and Voronoi diagrams require about 60-70% less cell updates and computation time. At the same time they provide equal or more accurate results without any drawbacks. With modifications that limit the maximum propagated distances and reduce the neighborhood size during propagation, we can also update three-dimensional distances maps at a speed that is suitable for online applications. Our Voronoi approach is easy to implement and, unlike previous approaches, correctly handles non-convex obstacle compounds like indoor areas.

For the three-dimensional configuration space of non-circular robots we also presented methods to incrementally update collision maps, distance maps, and Voronoi diagrams. We consider different obstacle representations, namely a robot moving on a plane with overhanging obstacles, or vice versa, obstacles elevating from the ground, and a robot with overhanging parts. The approaches are also applicable to 2D and full 3D obstacle representations and can exploit symmetries in the robot shape.

Our algorithms have been implemented and tested on real-world datasets. The achieved minimum frame rates for updates, collision checks, and path planning range between 3 and 20 fps, depending on the dimensionality of the map and the size of the environment. If required, the frame rates can be further increased by limiting the propagated distances. The source code of all our algorithms is available online [1]. The 3D distance map is also available as part of the OctoMap software package [48].

Acknowledgements

This work has partly been supported by the European Commission under grant agreement numbers FP7-248258-First-MM, FP7-248873-RADHAR, and FP7-260026-TAPAS.

References

[1] B. Lau, C. Sprunk, W. Burgard, Open source implementation of dynamically updatable distance maps, Voronoi diagrams, and configuration space representations, `http://www.informatik.uni-freiburg.de/~lau/dynamicvoronoi`, 2012.

[2] B. Lau, C. Sprunk, W. Burgard, Improved Updating of Euclidean Distance Maps and Voronoi Diagrams, in: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Taipei, Taiwan, 281–286, 2010.

[3] B. Lau, C. Sprunk, W. Burgard, Incremental Updates of Configuration Space Representations for Non-Circular Mobile Robots with 2D, 2.5D, or 3D Obstacle Models, in: European Conference on Mobile Robots (ECMR), Örebro, Sweden, 49–54, 2011.

[4] H. Choset, J. Burdick, Sensor-Based Exploration: The Hierarchical Generalized Voronoi Graph, International Journal of Robotics Research (IJRR) 19 (2) (2000) 96–125.

[5] F. Aurenhammer, Voronoi diagrams – a survey of a fundamental geometric data structure, ACM Computing Surveys (CSUR) 23 (3) (1991) 345–405.

[6] C. Mandel, U. Frese, Comparison of Wheelchair User Interfaces for the Paralysed: Head-Joystick vs. Verbal Path Selection from an offered Route-Set, in: European Conference on Mobile Robots (ECMR), 2007.

[7] S. Thrun, A Probabilistic On-Line Mapping Algorithm for Teams of Mobile Robots, International Journal of Robotics Research (IJRR) 20 (5) (2001) 335–363.

[8] G. Borgefors, Distance transformations in digital images, Computer Vision, Graphics, and Image Processing 34 (3) (1986) 344–371.

[9] R. Fabbri, L. da Fontoura Costa, J. C. Torelli, O. M. Bruno, 2D Euclidean distance transform algorithms: A comparative survey, ACM Computing Surveys 40 (1) (2008) 1–44.

[10] B. J. H. Verwer, P. W. Verbeek, S. T. Dekker, An efficient uniform cost algorithm applied to distance transforms, IEEE Transactions on Pattern Analysis and Machine Intelligence 11 (4) (1989) 425–429.

[11] N. Kalra, D. Ferguson, A. Stentz, Incremental reconstruction of generalized Voronoi diagrams on grids, Robotics and Autonomous Systems (RAS) 57 (2) (2009) 123–128.

[12] A. Stentz, Optimal and Efficient Path Planning for Partially-Known Environments, in: IEEE International Conference on Robotics and Automation (ICRA), San Diego, CA, USA, 3310–3317, 2004.

[13] P.-E. Danielsson, Euclidean Distance Mapping, Computer Graphics and Image Processing 14 (3) (1980) 227–248.

[14] S. Scherer, D. Ferguson, S. Singh, Efficient C-Space and Cost Function Updates in 3D for Unmanned Aerial Vehicles, in: IEEE International Conference on Robotics and Automation (ICRA), Kobe, Japan, 3860–3865, 2009.

[15] O. Cuisenaire, B. Macq, Fast Euclidean Distance Transformation by Propagation Using Multiple Neighborhoods, Computer Vision and Image Understanding 76 (2) (1999) 163–172.

[16] S. Scherer, J. Rehder, S. Achar, H. Cover, A. Chambers, S. Nuske, S. Singh, River mapping from a flying robot: state estimation, river detection, and obstacle mapping, Autonomous Robots 33 (2012) 189–214.

[17] T. Tao, S. Tully, G. Kantor, H. Choset, Incremental construction of the saturated-GVG for multi-hypothesis topological SLAM, in: IEEE International Conference on Robotics and Automation (ICRA), 3072–3077, 2011.

[18] N. Rao, N. Stoltzfus, S. Iyengar, A 'retraction' method for learned navigation in unknown terrains for a circular robot, IEEE Transactions on Robotics and Automation 7 (5) (1991) 699–707.

[19] C. M. Gold, P. R. Remmele, T. Roos, Voronoi methods in GIS, in: Algorithmic Foundations of Geographic Information Systems, vol. 1340, Springer Berlin / Heidelberg, 21–35, 1997.

[20] I. Lee, M. Gahegan, Interactive Analysis Using Voronoi Diagrams: Algorithms to Support Dynamic Update from a Generic Triangle-based Data Structure, Transactions in GIS 6 (2) (2002) 89–114.

[21] L. J. Guibas, D. E. Knuth, M. Sharir, Randomized Incremental Construction of Delaunay and Voronoi Diagrams, Algorithmica 7 (1992) 381–413.

[22] H. Choset, S. Walker, K. Eiamsa-Ard, J. Burdick, Sensor-Based Exploration: Incremental Construction of the Hierarchical Generalized Voronoi Graph, The International Journal of Robotics Research 19 (2000) 126–148.

[23] M. Tang, Y. J. Kim, D. Manocha, CCQ: Efficient Local Planning using Connection Collision Query, in: Algorithmic Foundations of Robotics IX, vol. 68 of *Springer Tracts in Advanced Robotics (STAR)*, Springer Berlin / Heidelberg, 229–247, 2011.

[24] J. Pan, D. Manocha, GPU-based Parallel Collision Detection for Real-time Motion Planning, in: Algorithmic Foundations of Robotics IX, vol. 68 of *Springer Tracts in Advanced Robotics (STAR)*, Springer Berlin / Heidelberg, 211–228, 2011.

[25] C. Schlegel, Fast Local Obstacle Avoidance under Kinematic and Dynamic Constraints for a Mobile Robot, in: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Victoria, Canada, 594–599, 1998.

[26] T. Lozano-Perez, Spatial Planning: A Configuration Space Approach, IEEE Transactions on Computers C-32 (2) (1983) 108–120.

[27] K. D. Wise, A. Bowyer, A Survey of Global Configuration-Space Mapping Techniques for a Single Robot in a Static Environment, International Journal of Robotics Research (IJRR) 19 (8) (2000) 762–779.

[28] J. Linan, T. Zhenmin, Building configuration space for multiple UGVs, in: IEEE International Conference on Vehicular Electronics and Safety, 245–250, 2005.

[29] E. Behar, J.-M. Lien, A New Method for Mapping the Configuration Space Obstacles of Polygons, Tech. Rep. GMU-CS-TR-2011-11, Department of Computer Science, George Mason University, 2010.

[30] L. E. Kavraki, Computation of Configuration-Space Obstacles Using the Fast Fourier Transform, IEEE Transactions on Robotics and Automation 11 (3) (1995) 408–413.

[31] R. Therón, V. Moreno, B. Curto, F. J. Blanco, Configuration space of 3D mobile robots: Parallel processing, in: 11th International Conference on Advanced Robotics, vol. 1–3, 210–215, 2003.

[32] F. J. Blanco, V. Moreno, B. Curto, R. Therón, C-Space Evaluation for Mobile Robots at Large Workspaces, in: IEEE International Conference on Robotics and Automation (ICRA), Barcelona, Spain, 3434–3439, 2005.

[33] J. Ziegler, C. Stiller, Fast Collision Checking for Intelligent Vehicle Motion Planning, in: IEEE Intelligent Vehicles Symposium, San Diego, CA, USA, 518–522, 2010.

[34] X. J. Wu, J. Tang, K. H. Heng, On the Construction of Dis-

cretized Configuration Space of Manipulators, Robotica 25 (1) (2007) 1–11.

[35] M. Foskey, M. Garber, M. C. Lin, D. Manocha, A Voronoi-based hybrid motion planner, in: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Maui, HI, USA, 55–60, 2001.

[36] S. Koenig, M. Likhachev, D* lite, in: Eighteenth National Conference on Artificial Intelligence (AAAI), 476–483, 2002.

[37] S. Lindemann, S. LaValle, Incrementally Reducing Dispersion by Increasing Voronoi Bias in RRTs, in: IEEE International Conference on Robotics and Automation (ICRA), 3251–3257, 2004.

[38] L. Zhang, D. Manocha, An efficient retraction-based RRT planner, in: IEEE International Conference on Robotics and Automation (ICRA), Pasadena, 3743–3750, 2008.

[39] R. Geraerts, M. Overmars, A Comparative Study of Probabilistic Roadmap Planners, in: Algorithmic Foundations of Robotics V, vol. 7 of *Springer Tracts in Advanced Robotics (STAR)*, Springer Berlin / Heidelberg, 43–58, 2004.

[40] J. Canny, A Voronoi Method for the Piano-Movers Problem, in: IEEE International Conference on Robotics and Automation (ICRA), 530–535, 1985.

[41] C. R. Maurer, Jr., R. Qi, V. Raghavan, A linear time algorithm for computing exact Euclidean distance transforms of binary images in arbitrary dimensions, IEEE Transactions on Pattern Analysis and Machine Intelligence 25 (2) (2003) 265–270.

[42] P. Beeson, EVG-Thin: A Thinning Approximation to the Extended Voronoi Graph, Available online: `http://www.cs.utexas.edu/users/qr/software/evg-thin.html`, 2006.

[43] G. Grisetti, C. Stachniss, W. Burgard, Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters, IEEE Transactions on Robotics 23 (1) (2007) 34–46.

[44] C. Stachniss, U. Frese, G. Grisetti, OpenSLAM, URL `http://openslam.org`, 2012.

[45] K. M. Wurm, A. Hornung, M. Bennewitz, C. Stachniss, W. Burgard, OctoMap: A Probabilistic, Flexible, and Compact 3D Map Representation for Robotic Systems, in: ICRA Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation, Anchorage, 2010.

[46] C. Sprunk, B. Lau, P. Pfaff, W. Burgard, Online Generation of Kinodynamic Trajectories for Non-Circular Omnidirectional Robots, in: IEEE International Conference on Robotics and Automation (ICRA), Shanghai, 72–77, 2011.

[47] I. A. Şucan, M. Moll, L. E. Kavraki, The Open Motion Planning Library (OMPL), Available online: `http://ompl.kavrakilab.org`, 2011.

[48] K. M. Wurm, A. Hornung, OctoMap, An Efficient Probabilistic 3D Mapping Framework Based on Octrees, `http://octomap.github.com`, 2012.