Improved Updating of Euclidean Distance Maps and Voronoi Diagrams

Boris Lau Christoph Sprunk Wolfram Burgard

***Abstract*— This paper presents novel, highly efficient approaches for updating Euclidean distance maps and Voronoi diagrams represented on grid maps. Our methods employ a dynamic variant of the brushfire algorithm to update only those cells that are actually affected by changes in the environment. In experiments in different environments we show that our update strategies for distance maps and Voronoi diagrams require substantially fewer cell visits and significantly less computation time compared to previous approaches. Furthermore, the dynamic Voronoi diagram also improves on previous work by correctly dealing with non-convex obstacles such as building walls. We also present a dynamic variant of a skeletonization-based approach to Voronoi diagrams that is especially robust to noise. All of our algorithms consider actual Euclidean distances rather than grid steps. An open source implementation is available online [1].**

I. Introduction

The Generalized Voronoi Diagram (GVD) is a data structure that has been widely used in various fields [2]. In the context of robotics it is a popular cell decomposition method for solving navigation tasks. The GVD is defined as the set of points in free space to which the two closest obstacles have the same distance [3], which motivates its application as a roadmap technique for path planning: GVDs are "sparse" in the sense that different paths on the GVD correspond to topologically different routes with respect to obstacles. This significantly reduces the search problem and can be used to generate the n-best paths for offering route alternatives to a user or for optimization-based motion planning [4]. Also, moving along the edges of a GVD ensures the greatest possible clearance when passing between obstacles. This relates the GVD to distance maps (DMs) which store in each cell the distance to its closest obstacle. Since a cell lookup requires only constant time, DMs provide efficient means for collision checks, computing traversal costs for path planning or for robot localization with likelihood fields [5].

Several algorithms for computing DMs and GVDs of given static environments have been proposed in the past [2], [6]. However, applications in dynamic environments require efficient updates of DMs and GVDs, especially when using large maps or more than two dimensions, or when computational resources are limited. Existing approaches to dynamic updates of GVDs on grid maps approximate obstacle distances with grid step distances, suffer from discretization artifacts, and fail inside non-convex obstacle compounds like building walls as shown in Fig. 1 (top-left).

All authors are with the Department of Computer Science at the University of Freiburg, Germany, {lau,sprunkc,burgard}@informatik.uni-freiburg.de. This work has partly been supported by the European Commission under grant agreement number FP7-248258-First-MM.

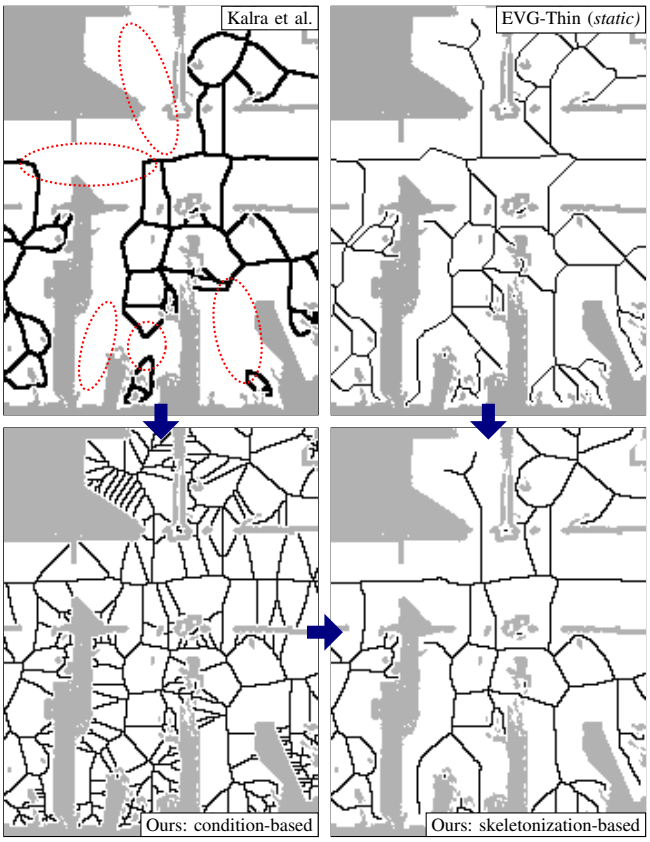

Fig. 1. Dynamic Voronoi diagrams on grid maps. Our condition-based approach improves on previous work by Kalra *et al.* [7] which generates overly thick Voronoi lines and fails inside enclosed areas (dotted ellipses). We combine this with ideas from the static Voronoi approximation method "EVG-Thin" [8] and propose a dynamic skeletonization-based approach that is especially robust to noise.

This paper presents algorithms for efficient updating of DMs (Sect. III) and GVDs (Sect. IV) that overcome the aforementioned problems, use actual Euclidean distances, and are computationally more efficient than previous approaches. Sect. V evaluates and discusses our methods compared to the state of the art.

II. Related Work

Existing approaches for static two-dimensional DMs comprise analytical methods, linear image traversal, and distance propagation with the brushfire method (see the recent survey by Fabbri *et al.* [6] for a comparison). Whenever a cell in a grid map is newly occupied or freed, the corresponding DM has to be updated to reflect that change. A trivial method is to recompute distances for patches within $2d_{\max}$ around all changed cells, where $d_{\max}$ is an upper bound on the minimum obstacle distance in the environment. However, this method

Algorithm 1 Pseudo-Code for Updating Euclidean Distance Maps

SetObstacle(s)
1: $obst_s \leftarrow s$
2: $dist_s \leftarrow 0$
3: insert$(open, s, 0)$

RemoveObstacle(s)
4: clearCell(s)
5: $toRaise_s \leftarrow$ true
6: insert$(open, s, 0)$

UpdateDistanceMap()
7: **while** $open \neq \emptyset$ **do**
8: $s \leftarrow$ pop$(open)$
9: **if** $toRaise_s$ **then**
10: raise(s)
11: **else if** isOcc$(obst_s)$ **then**
12: $voro_s \leftarrow$ false
13: lower(s)
14: **return** $dist$

raise(s)
15: **for all** $n \in \text{Adj}_8(s)$ **do**
16: **if** $(obst_n \neq$ cleared $\wedge \neg toRaise_n)$ **then**
17: **if** $\neg$isOcc$(obst_n)$ **then**
18: clearCell(n)
19: $toRaise_n \leftarrow$ true
20: insert$(open, n, dist_n)$
21: $toRaise_s \leftarrow$ false

lower(s)
22: **for all** $n \in \text{Adj}_8(s)$ **do**
23: **if** $\neg toRaise_n$ **then**
24: $d \leftarrow \|obst_s - n\|$
25: **if** $d < dist_n$ **then**
26: $dist_n \leftarrow d$
27: $obst_n \leftarrow obst_s$
28: insert$(open, n, d)$
29: **else** chkVoro(s, n)

usually updates substantially more cells than necessary, e.g., if $d_{\max}$ is high due to large open spaces or if changed cells cover a wide area. Kalra *et al.* recently proposed to update DMs and GVDs with a dynamic brushfire algorithm [7] based on D^*, which starts propagating wavefronts at newly occupied or deleted obstacle cells. These wavefronts accumulate 8-connected grid steps to approximate obstacle distances. However, this overestimates the true Euclidean distances by up to 8.0% [9], which for a robot implies either a collision risk or overly conservative movements.

Scherer *et al.* adopted and corrected Kalra's algorithm for DM updates [10]. 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 [9]. According to Cuisenaire and Macq [11], the shortest distance at which this propagation error can occur is 13 pixels, which yields a maximum relative error of 0.69%. Our approach follows the same principle and applies it to GVD generation as well. It improves on the work by Scherer *et al.* by requiring significantly lower computational time for the same task due to a substantially reduced number of cell visits. Similar to their algorithm, our approach can also be trivially extended to 3D and incorporate a limit on the propagated distance.

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 moving input points [12] or points that have been inserted or deleted [13]. 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.

The approach for updating GVDs proposed by Kalra *et al.* [7] addresses this issue by introducing obstacle identifiers that are uniquely assigned to a compound of connected obstacle cells. 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 mentioned in Sect. I. Additionally, it does not generate Voronoi lines in the interior of concave obstacle compounds like rooms or corridors as shown in Fig. 1 (top-left), which destroys the connectivity of the GVD and is disadvantageous for path planning. 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 consider actual Euclidean distances and present a condition-based approach that a) does not rely on obstacle identifiers and b) generates thin Voronoi lines, as shown in Fig. 1 (bottom-left).

Zhang and Suen [14] proposed an approach for GVDs on static grid maps that erodes the free space starting with the free cells next to obstacles. For each cell under inspection, a set of so-called "thinning patterns" determines if a cell can be removed or not. At the end of this iterative process the remaining skeleton approximates the GVD. Beeson's C++ implementation "EVG-Thin" [8] follows this approach. An example output is shown in Fig. 1 (top-right). This GVD approximation is robust to noise and fairly thin, i.e., the Voronoi lines are only one-pixel wide. However, a few double-cell lines still occur and the skeleton uses many straight and diagonal lines where the actual Voronoi lines would be curved. Furthermore, it cannot be updated dynamically. In addition to the condition-based approach described above, this paper presents a GVD generation algorithm that combines the former with an erosion process that uses true Euclidean distances. The resulting method is robust to noise and allows for efficient updates in dynamic scenarios. An example output is shown in Fig. 1 (bottom-right).

III. Updating Euclidean Distance Maps

Similar to the method by Kalra *et al.* [7], our approach for updating Euclidean DMs uses a dynamic variant of the brushfire method to iteratively compute distances. Following the notation of Kalra *et al.*, the pseudo-code of our algorithm is given in Alg. 1. Movement, insertion, or deletion of objects causes individual cells in an occupancy grid map to flip their state from free to occupied or vice versa. A change 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. When calling the function **UpdateDistanceMap()** the collected cell changes are propagated to all affected cells which completes the update.

Newly occupied cells initiate "lower" wavefronts that update the closest obstacle distance of affected cells. Similarly, "raise" wavefronts start at newly freed cells and clear the distance entries of all cells whose closest obstacle was the deleted one, as shown in Fig. 2. The processing of raise and lower wavefronts is interwoven and controlled by a single

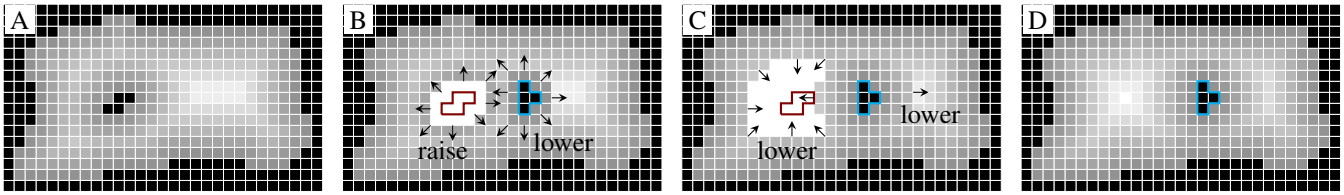

Fig. 2. Distance map update between two configurations (A) and (D). Black represents occupied cells, brightness increases with distance. The inserted obstacle (blue) initiates a "lower" wavefront shown in the intermediate steps (B) and (C) that updates the distances in the cells up to the point where a different obstacle is closer. The removed obstacle (red outline) starts a "raise" wavefront (B) that clears the cells that lost their closest obstacle. When it comes to a halt it initiates a "lower" wavefront (C) that recomputes the distances for the cleared cells (white) on the basis of the remaining obstacles.

priority queue that sorts the enqueued cells by distance. Both the raise wavefronts (Alg. 1, line 20) and the lower wavefronts (line 28) enqueue the neighbors of a processed cell to propagate the wavefront.

A. Data fields and initialization

The output of an update step is a distance map $dist$, which stores in each cell s its Euclidean distance to the closest occupied cell in the corresponding grid map. The obstacle reference map $obst$ stores for all cells the coordinates of their closest occupied cell. If s is occupied, it has a distance of $dist_s = 0$ and refers to itself as closest obstacle location, i.e., $obst_s = s$. The function **isOcc(**s**)** returns whether the latter condition is true for a given cell s. Upon initialization, all values are undetermined, i.e., $dist_s = \infty$ and $obst_s =$ cleared. The function **clearCell(**s**)** used in the algorithm also resets s to these values.

All cells that need processing by either a lower or raise wavefront are inserted into a priority queue $open$ that is sorted by a distance value. The function **pop(**$open$**)** returns the cell s with the lowest enqueued distance and removes it from the queue. The method **insert(**$open, s, d$**)** inserts s into the queue with distance d or updates the priority if s is already enqueued. An additional flag $toRaise$ is used to ensure proper processing of cells in the wavefronts, in particular where raise and lower wavefronts overlap.

B. Raise and lower: propagating wavefronts

While the priority queue is not empty, the function **UpdateDistanceMap()** repeatedly retrieves the next unprocessed cell s (lines 7–8). If s has been cleared but has not yet propagated a raise wavefront, the function **raise(**s**)** is called (lines 9–10). If s however has a valid closest obstacle, the function **lower(**s**)** is called to propagate the lower wavefront (lines 11–13).

The function **raise(**s**)** processes each cell n in the 8-connected neighborhood $\text{Adj}_8(s)$ of s that is not yet part of a raise wavefront and refers to a closest obstacle $obst_n$ (lines 15–16). If $obst_n$ is not occupied, n is cleared, marked to propagate the raise wavefront, and inserted into the priority queue (lines 17–20). Otherwise, the raise wavefront comes to a halt at n, leaves n unchanged but still enqueues it to initiate a lower wavefront (line 20), as shown in Fig. 2 (C).

The function **lower(**s**)** considers each cell n in the 8-connected neighborhood $\text{Adj}_8(s)$ of s that is not marked to be part of a raise wavefront (lines 22–23). The Euclidean distance from n to the closest obstacle of s is compared to the current closest obstacle distance of n (lines 24–25). 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 26–28).

To avoid superfluous raise wavefronts where they would overlap with lower wavefronts, the condition in line 25 can be extended to also overwrite cells with equal distance that refer to a deleted obstacle. The line then reads "if $d < dist_n \vee (d = dist_n \wedge \neg\text{isOcc}(obst_n))$ then".

The lines 12 and 29 update the GVD on the fly during the update of the distance map as described in Sect. IV. If only distance maps are required, the lines can be omitted.

C. Implementation Details

The distance map algorithm described above computes and compares real-valued Euclidean distances stored in $dist$. As previously done by Scherer *et al.* [10] and others, we resort to integer squared distances which saves computational expenses for the square-root.

A central data structure in our algorithm is the sorted priority queue used for the queues $open$ and $voroQ$. We implement these queues using the bucketing technique presented by Cuisenaire and Macq [11]. 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 constant time.

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.

IV. Reconstruction of Voronoi Diagrams

Voronoi graphs in continuous spaces consist of infinitely thin lines and curves. When 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 Voronoi graph looses its sparseness property, i.e., not all paths in the GVD correspond to topologically different routes with respect to obstacles.

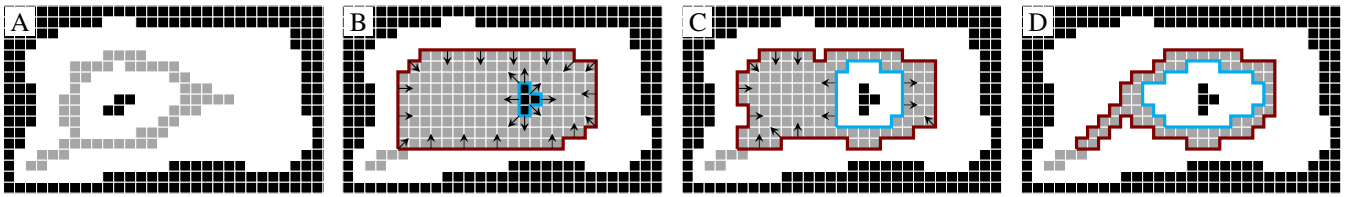

Fig. 3. Skeletonization-based Voronoi update between two configurations (A) and (D). The erosion (B) starts at cells adjacent to newly inserted obstacles (blue boundary), and where raise or lower wavefronts came to a hold during the distance map update (red boundary). In an iterative process, the cells are eroded depending on a set of conditions while ensuring the connectivity of the GVD (C). The remaining skeleton is the updated (unpruned) GVD (D).

Algorithm 2 Condition-based Voronoi

chkVoro(s, n)

30: **if** $(dist_s > 1 \vee dist_n > 1) \wedge obst_n \neq$ cleared
$\wedge\ obst_n \neq obst_s \wedge obst_s \notin \text{Adj}_8(obst_n)$ **then**
31: **if** $\|s - obst_n\| \leq \|n - obst_s\|$ **then** $voro_s \leftarrow$ true
32: **if** $\|n - obst_s\| \leq \|s - obst_n\|$ **then** $voro_n \leftarrow$ true

Algorithm 3 Skeletonization-based Voronoi

rebuildVoronoi$(voroQ)$

33: **while** $voroQ \neq \emptyset$ **do**
34: $s \leftarrow pop(voroQ)$
35: **if** $\neg$patternMatch$(s) \wedge \nexists n \in \text{Adj}_8(s) : conditions(s, n)$
$\wedge\ \text{comp}(obst_s) \neq \text{comp}(obst_n)$ **then**
36: $voro_s \leftarrow$ false
37: **for all** $n \in \text{Adj}_8(s)$ **do**
38: **if** $voro_n \neq$ false **then** $insert(voroQ, n, dist_n)$

When using an 8-connected grid model, the GVD appears to be thinner by visual inspection. However, due to the additional connections, such GVDs often violate the sparseness condition, which is not the case for 4-connected ones.

The remainder of this section presents two approaches to update GVDs on grid maps that mitigate the above-mentioned problems. Both methods can generate 4- and 8-connected GVDs. An additional pruning step deals with artifacts due to discretization, i.e., double lines and erroneous connections, and thus ensures sparseness of the GVD.

A. Condition-based GVD

Our *condition-based* approach to updateable GVDs improves on the algorithm by Kalra *et al.* [7] by performing less cell visits and operating without extra knowledge about obstacle identifiers. We represent the GVD by a map $voro$, 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: lower wavefronts remove elements from the GVD (line 12), and potentially add them after checking a number of conditions (line 29).

In continuous space, a point is part of the GVD if the distance to its two closest obstacles is identical. This condition cannot directly be applied to discretized cell coordinates. Instead, we have to select the cells for the GVD that contain Voronoi lines in their associated area. 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, it calls **chkVoro**(s, n). This function tests if at least one $c \in \{s, n\}$ is not adjacent to its closest obstacle (see Alg. 2, line 30), which increases the robustness to noise. If n has a valid closest obstacle that is different and not adjacent to the closest obstacle of s, both cells are GVD candidates. The function then 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 referenced obstacle to the one of the competing neighbor. If the increase is the same both ways, both cells are inserted (lines 31–32). For 8-connected GVDs, the "$\leq$" are replaced by "$<$" for diagonal neighbors s and n, since no cells need to be inserted in the case of equal increase.

B. Updating Skeletonization-based GVDs

Algorithms like EVG-Thin [8] skeletonize the free space to generate GVD approximations, starting with the cells adjacent to obstacles. By iteratively applying thinning patterns that preserve the connectivity, an evenly progressing erosion of the free space is ensured. The remaining skeleton approximates the GVD, as shown in Fig. 1 (top-right). These approaches are robust to noise and thus generate clean GVD approximations. This section presents an approach to generate and update skeletonization-based GVDs that consider actual Euclidean distances.

To ensure correct updates after topological changes, we assign a unique identifier to each connected group of occupied cells. These identifiers are stored by comp(s) for each occupied cell s, and can be updated efficiently in $O(\log n (\log \log n)^3)$ time [15].

The erosion of our dynamic method starts next to newly inserted obstacles and where raise wavefronts turned into lower wavefronts while updating the DM as shown in Fig. 3. These cells are enqueued in $voroQ$, sorted by their distances for even erosion.

The function **rebuildVoronoi**$(voroQ)$ iteratively retrieves the cell s with the lowest distance from the queue, see Alg. 3, lines 33–34. Cells were unconditionally added to the GVD by the raise wavefronts (not shown in pseudo code), and are now removed again (line 36) if all of the following conditions hold: (a) the cell is not required for the GVD as determined by connectivity patterns (b) no cell n adjacent to s fulfills the conditions in Sect. IV-A and has an obstacle with a different identifier comp$(obst_n)$ than the obstacle of s (line 35).

The patterns shown in Fig. 4 match whenever the center cell s provides connectivity for one or more of its adjacent cells. Here, a "1" matches $voro =$ true, while "0" stands for $voro =$ false, and empty fields are ignored. The patterns are applied in every rotation where indicated by arrows. If not

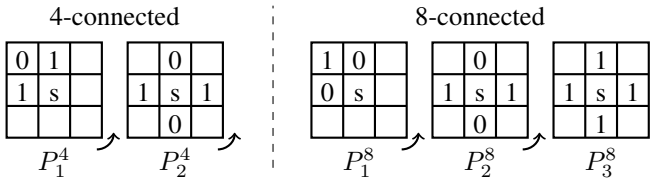

Fig. 4. Patterns used by the function **patternMatch**(s), which returns true if any of the patterns match. Arrows indicate application of rotated copies.

prevented by any of these conditions, the cell is removed and all its neighbors that still are on the GVD are inserted into $voroQ$ for inspection (lines 37–38).

C. Pruning

As discussed in Sect. I, 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. In contrast, both our approaches generate thin GVDs wherever this can be done unambiguously. Our optional pruning step erodes 2-pixel-wide Voronoi lines that occur where a theoretical Voronoi line passes exactly between two cells. Therefore, all new Voronoi cells are inserted into a priority queue and processed as follows.

In a first phase the pruning algorithm merges Voronoi lines that are erroneously connected due to the finite map resolution. This is done by adding unoccupied cells that are enclosed by 4-connected Voronoi lines to the GVD. They are detected by matching pattern P_3^8.

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 none of the connectivity patterns match at the cell's location, it can be removed from the GVD without affecting its topology.

V. Experiments

We tested our algorithms on laser range data of walking people in different environments (see Fig. 5), recorded by a moving robot. 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), and the maximum closest obstacle distance $d_{\max}$ in these two maps 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 $d_{\max} = 44$ cells (2.2 m). Similar to Kalra *et al.* [7], the random obstacles were placed within 5 m radius around a moving center, which simulates a moving observer with limited perception. The tests were done using our C++ implementation of the algorithms, running on an Intel® Core™ i7 2670 MHz. The source code and the employed data sets are available online [1].

Fig. 5. Maps of the environments where our experiments were carried out.

TABLE I
Update performance of distance maps and Voronoi diagrams

		Map & Approach	Time per frame [s] mean	min	max	Cell visits per frame 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 Skel	0.008	0.004	0.011	181,716	39,676	408,248
		*Ours Cond	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 Skel	0.060	0.027	0.091	2,480,006	880,617	3,715,450
		*Ours Cond	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 Skel	0.021	0.011	0.026	689,189	252,674	906,647
		*Ours Cond	0.017	0.009	0.021	391,555	113,889	515,343

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

A. Computational performance of dynamic updates

To demonstrate the computational benefit of dynamically updateable DMs, we compared our algorithm with state-of-the-art static methods implemented by Fabbri *et al.* [6], namely the algorithms by Cuisenaire and Macq [11] and Maurer *et al.* [16]. These approaches 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.* [10]. Since no source code was available, we implemented this algorithm in C++ with the assistance of Scherer. Our condition-based GVD "Ours Cond" and the skeletonization-based approach "Ours Skel" are compared to the static EVG-Thin method [8] and the dynamic approach by Kalra *et al.* [7]. The performance results are shown in Tab. I, and summarizing plots are presented in Fig. 6. For each sequence the computation time and cell visits per frame are given by their mean, minimum, and maximum values of all frames. 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 Maurer *et al.*,

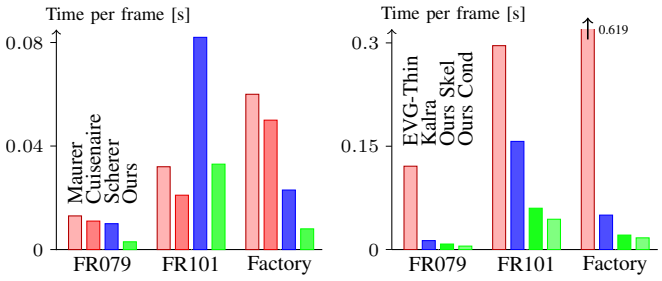

Fig. 6. Performance of our algorithms for updating distance maps and Voronoi diagrams compared to related work. The plots show the average computation time per frame.

except for the distance maps in the open space of FR101, where most updates affect a large fraction of the map. In all environments, 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.* [10]. 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 22). 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 GVD update algorithms shows similar results: the dynamic methods clearly outperform the static method EVG-Thin. In addition, both of our approaches can reduce the runtime considerably compared to the previous dynamic approach by Kalra *et al.* [7], since this method uses the same update strategy as Scherer's approach for DMs and thus visits more cells. Our proposed skeletonization-based approach has slightly higher computational costs than our condition-based one due to the erosion process. However, it still requires substantially less cell visits and computation time compared to Kalra's approach.

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.

B. Generated distance maps and Voronoi diagrams

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. II.

The GVDs are of equal or better quality, exemplary outputs of the static erosion-based approach by Beeson [8], the dynamic algorithm by Kalra *et al.* [7] and our two 4-connected approaches are given in Fig. 1: while Kalra's GVD misses Voronoi lines inside rooms and corridors, our condition-based approach captures the connectivity of the floor plan completely. The output of the static EVG-Thin approach is robust to noise. However, it only approximates the real GVD and prefers straight and diagonal lines. Our skeletonization-based approach shows the same robustness to noise as EVG-Thin, but generates actual Euclidean GVDs. Furthermore, it only updates the affected cells, which can be seen in the results in Tab. I and Fig. 6.

VI. Conclusion

In this paper we presented techniques for updating Euclidean distance maps and Voronoi diagrams. Compared to previous approaches, our methods require about 60-70% less cell updates and computation time, and at the same time provide equal or more accurate results without any drawbacks. Our condition-based Voronoi approach is easy to implement and, unlike previous approaches, handles non-convex obstacle compounds correctly. The proposed skeletonization-based approach is robust to noise and allows for efficient updating of Voronoi diagrams. Our algorithms have been implemented and tested on real-world data sets. Experiments demonstrate that our approaches allow to update distance maps with minimum frame rates of 20 fps and Voronoi diagrams with minimum frame rates of 15 fps even on very large maps. This property is especially useful for robot navigation in dynamic environments. The source code of our implementation is available online [1].

References

[1] B. Lau, C. Sprunk, and W. Burgard, "Open source implementation of dynamically updateable distance maps and voronoi diagrams," http://www.informatik.uni-freiburg.de/~lau/updatingGVD.

[2] F. Aurenhammer, "Voronoi diagrams – a survey of a fundamental geometric data structure," *ACM Computing Surveys (CSUR)*, vol. 23, no. 3, 1991.

[3] H. Choset, "Sensor based motion planning: The hierarchical generalized voronoi graph," Ph.D. dissertation, California Institute of Technology, Pasadena, CA, USA, 1996.

[4] B. Lau, C. Sprunk, and W. Burgard, "Kinodynamic motion planning for mobile robots using splines," in *IEEE/RSJ Intl. Conference on Intelligent Robots and Systems (IROS)*, 2009.

[5] S. Thrun, "A probabilistic on-line mapping algorithm for teams of mobile robots," *International Journal of Robotics Research (IJRR)*, vol. 20, no. 5, 2001.

[6] R. Fabbri, L. da Fontoura Costa, J. C. Torelli, and O. M. Bruno, "2D Euclidean distance transform algorithms: A comparative survey," *ACM Computing Surveys*, vol. 40, no. 1, 2008.

[7] N. Kalra, D. Ferguson, and A. Stentz, "Incremental reconstruction of generalized voronoi diagrams on grids," *Robotics and Autonomous Systems*, vol. 57, pp. 123–128, 2009.

[8] P. Beeson, "EVG-Thin: A thinning approximation to the extended voronoi graph," 2006. [Online]. Available: http://www.cs.utexas.edu/users/qr/software/evg-thin.html

[9] P.-E. Danielsson, "Euclidean distance mapping," *Computer Graphics and Image Processing*, vol. 14, pp. 227–248, 1980.

[10] S. Scherer, D. Ferguson, and S. Singh, "Efficient C-Space and cost function updates in 3d for unmanned aerial vehicles," in *Intl. Conf. on Robotics and Automation (ICRA)*, Kobe, Japan, 2009.

[11] O. Cuisenaire and B. Macq, "Fast euclidean distance transformation by propagation using multiple neighborhoods," *Computer Vision and Image Understanding*, vol. 76, pp. 163–172, 1999.

[12] C. M. Gold, P. R. Remmele, and T. Roos, "Voronoi methods in GIS," in *Algorithmic Foundations of Geographic Information Systems*. Springer Berlin / Heidelberg, 1997, vol. 1340.

[13] I. Lee and M. Gahegan, "Interactive analysis using voronoi diagrams: Algorithms to support dynamic update from a generic triangle-based data structure," *Transactions in GIS*, vol. 6, no. 2, pp. 89–114, 2002.

[14] T. Zhang and C. Suen, "A fast parallel algorithm for thinning digital patterns," *Communications of the ACM*, vol. 27, no. 3, 1984.

[15] M. Thorup, "Near-optimal fully-dynamic graph connectivity," in *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, New York, NY, USA, 2000, pp. 343–350.

[16] C. R. Maurer, Jr., R. Qi, and 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*, vol. 25, no. 2, pp. 265–270, 2003.