2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
Macau, China, November 4-8, 2019

FIESTA: Fast Incremental Euclidean Distance Fields for Online Motion Planning of Aerial Robots

Luxin Han, Fei Gao, Boyu Zhou and Shaojie Shen

***Abstract*— Euclidean Signed Distance Field (ESDF) is useful for online motion planning of aerial robots since it can easily query the distance and gradient information against obstacles. Fast incrementally built ESDF map is the bottleneck for conducting real-time motion planning. In this paper, we investigate this problem and propose a mapping system called FIESTA to build global ESDF map incrementally. By introducing two independent updating queues for inserting and deleting obstacles separately, and using Indexing Data Structures and Doubly Linked Lists for map maintenance, our algorithm updates as few as possible nodes using a BFS framework. Our ESDF map has high computational performance and produces near-optimal results. We show our method outperforms other up-to-date methods in term of performance and accuracy by both theory and experiments. We integrate FIESTA into a completed quadrotor system and validate it by both simulation and onboard experiments. We release our method as open-source software for the community[1].**

I. Introduction

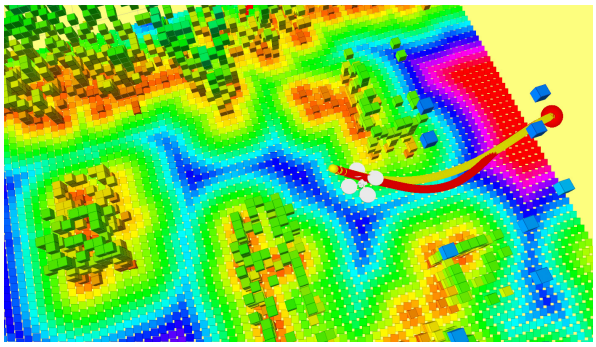
For a fully autonomous Micro Aerial Vehicle (MAV), the perception-planning-control pipeline takes environmental measurements as input and generates control commands. The mapping module, which provides a foundation for onboard motion planning, is one of the most essential components in this system.

There are many well-developed data structures and algorithms for mapping. The basic requirement of a mapping system is to balance the accuracy of fusing depth measurements and the overhead of storing a fine representation of the environment. Representative mapping frameworks include Octomap [2] and TSDF(Truncated Signed Distance Field) [3], etc. However, for the purpose of autonomous quadrotor navigation, what is truly useful is the information of free space, instead of obstacles. A desirable map for planning must have the capability of fast querying free/occupied status, or getting the distance information to obstacles, such as the Euclidean Signed Distance Field (ESDF) map.

ESDF map has the advantage to evaluate the distance and gradient information against obstacles and is, therefore, necessary for gradient-based planning method, such as CHOMP [4]. The gradient-based method tends to push the generated trajectories away from obstacles to improve path clearance. Therefore it is particularly useful in quadrotor planning as presented in [5], [6] and [7]. For real-time onboard motion planning, the efficiency and accuracy in maintaining and updating the ESDF map is always the bottleneck. In [8], an efficient method using distance parabolic curve is proposed to compute the ESDF globally. However, for quadrotor platform where the sensing range and onboard computing resource are both very limited, incrementally updating the ESDF map is demanding for real-time onboard planning. Another way to update ESDF map in real-time is to keep and maintain only a small local map sliding with the quadrotor, such as presented in [7]. Although this compromise works in quadrotor local (re-)planning, it discards all past map information and can not be used for applications where global or repeatable planning is needed.

Voxblox [9] are also proposed to incrementally build the ESDF map. It computes the ESDF map directly from a Truncated Signed Distance Field (TSDF) map. It leverages the distance information already contained within the truncated

(a)



(b)

Fig. 1. Motion planning experiment using FIESTA for calculate ESDF Map. The quadrotor plans to fly to a designated destination through the maze. [1] is used for motion planning. (a) shows the quadrotor in the experiment, (b) is the corresponding visulization of FIESTA, where cubes are obstacles, plane is one slice of ESDF map, yellow and red lines are the planned trajectory. Details about the experiments are shown in this video[2].

All authors are with the Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong, China. {luxin.han, fgaoaa, bzhouai, eeshaojie}@ust.hk

[1]https://github.com/hlx1996/FIESTA

[2]https://www.youtube.com/watch?v=pgRi8LOnT6Y

radius in the TSDF map and then expands to all the voxels in the map using BFS (Breadth-First Search). It can work in real-time on a dynamically growing map, however, there are actually two categories of errors existing in the final ESDF map. Firstly, it conducts the BFS and updates the ESDF according to the quasi-Euclidean distance. Therefore the computed distance may have larger error compared to the actual Euclidean distance. Secondly, it relies on TSDF-based mapping, but the TSDF projective distance may overestimate the actual Euclidean distance to the nearest surface.

In this paper, we propose **FIESTA** (**F**ast **I**ncremental **E**uclidean Di**STA**nce Fields), which is a lightweight and flexible mapping framework for building ESDF map incrementally. Our method takes the pose estimations and depth measurements as input, and updates the ESDF map globally with the minimal computational overhead. This is done by our elaborately designed data structures and ESDF updating algorithm. Our data structures build the foundation for our high-efficiency ESDF updating algorithm, and also provides an option to trade-off the time complexity and space complexity. Our proposed algorithm expands as few as possible nodes using a BFS framework and results in much more accurate ESDF values. What's more, unlike Voxblox [9] depending on TSDF, our algorithm doesn't rely on any specific types of mapping framework. It means that our method can be applied to any general mapping framework, including occupancy grid map and TSDF-based map. The contributions of this work are:

- Elaborately designed data structures for fast updating the ESDF map incrementally.
- A novel ESDF updating algorithm which expands as few as possible nodes and obtains near-optimal results.
- Theoretical and practical analysis of time and space complexity, accuracy, and optimality.
- Integration of the proposed ESDF map into a completed quadrotor system and demonstrations of onboard MAV motion planning.
- Release the proposed mapping framework as open-source software.

II. Related Work

There are a lot of different mapping frameworks used in quadrotor planning. Some of them are derived from a general map used in robotic perception. In [10], a grid map is used for fast collision checking and occupancy evaluation. Using the occupancy map, a field-based path searching is conducted, followed by a hard-constrained piecewise trajectory generation. In [11], the authors utilize the Octomap to group large free space for searching a collision-free path and generating safe trajectories. Moreover, point clouds are directly used for planning in [12], where the safe space of the environment is extracted by randomly querying the nearest neighbor using a Kd-tree of the point cloud. Some other mapping frameworks tailored for robotic navigation are also proposed recently. Topomap [13] builds convex clusters based on sparse features points from a visual SLAM system. The authors build a sparse topological graph using these convex clusters, where a path can be found very efficiently. Sparsemap [14] shares a similar idea with Topomap. The authors propose a complete pipeline to extract a 3D Generalized Voronoi Diagram (GVD) based on an ESDF and obtain a thin skeleton diagram representing the topological structure of the environment. MAV planning is conducted in this topological map. Nanomap [15] propose a novel map structure, which can be built and queried range search efficiently. In Nanomap [15], the authors propose to keep only recently depth raw measurements to do collision checking, instead of explicitly maintaining a fused map. Fast quadrotor flight is supported by this map structure.

ESDF map is widely used in gradient-based robotics motion planning, where the distance and gradient information to obstacles are necessary. CHOMP [4] uses the method [8] to compute the ESDF map. It starts with a deterministic occupancy grid map, which means the value is either 0 or 1, and compute the EDT (Euclidean Distance Transform) for the map. Computing the EDT using [8] is efficient. However, motion planning suffers from repeating calculating ESDF value again and again. It can't satisfy both the map accuracy and updating frequency, which in return, limits the performance of the planning module running onboard.

The most relevant work to ours is Voxblox [9], which also focus on building the ESDF map incrementally. Voxblox firstly integrates sensor data into a TSDF Map and then calculates ESDF from TSDF using a BFS(Breadth-First-Search) framework. Voxblox fully utilizes the distance information in the TSDF map, has real-time performance on CPU and allows dynamically-growing map size. However, this approach has some drawbacks. Firstly, the ESDF values are far away from accurate. There are actually two categories of errors existing in the final ESDF map. (i) It uses BFS to update the ESDF information of neighbor points and updates them with the length of a broken line, which is also called quasi-Euclidean expanding. In this way, the final ESDF is not the real Euclidean distance, which may have larger error compared to the actual Euclidean distance. (ii) It relies on TSDF-based mapping, where the TSDF projective distance may overestimate the actual Euclidean distance to the nearest surface. Instead, our work incrementally calculates ESDF directly from an occupancy grid map, one of the simplest mapping data structures, which eliminates error from (ii) fundamentally. As for (i), we also use BFS but updates its neighbors in another way. It is not accurate, either, as we prove that in Section V-E that all ESDF updating algorithms based on BFS cannot be accurate. However it is much better than a quasi-Euclidean way in an order of magnitude in both theory and practice, which will be shown in Section V-E and Section VI-A, respectively. Secondly, Voxblox highly depends on TSDF mapping, which results in its high complexity and low extensibility. Its complexity also makes it hard to be transplanted into other applications. And its updating starts from changes of TSDF voxels, makes the whole algorithm hard to analyze and optimize. On the contrary, our proposed system is a lightweight solution which introduces two independent updating queues for inserting and

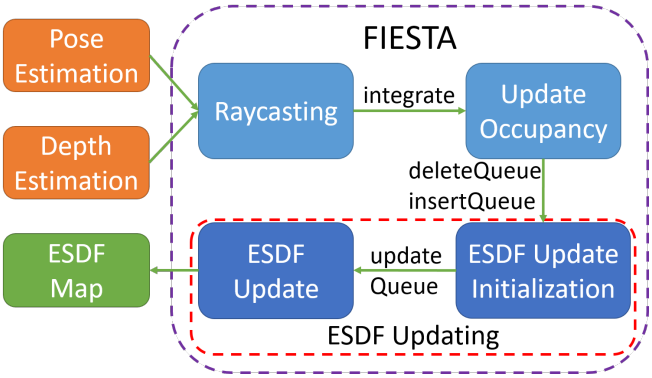


Fig. 2. System overview of our proposed ESDF mapping framework.

Name	Meaning	Abbreviation
position	voxel coordinate	pos
occupancy	probability of occupancy	occ
ESDF	Euclidean distance to the closest obstacle	dis
Closest Obstacle Voxel Coordinate	the voxel coordinate of the closest obstacle	coc
observed	whether this voxel is ever observed	obs
prev, next, head	used in DLLs	$prev$, $next$, $head$
Doubly Linked List (method)	all voxels that the closest obstacle is this voxel	dll
Neighhoods (method)	all observed neighbors of this voxel	$nbrs$

TABLE I
MEMBERS AND METHODS OF A VIS AND ABBREVIATION IN PSEUDOCODE

deleting obstacles separately, gives a natural explanation of the starting points of the BFS, makes it easier to analyze and optimize.

III. System Framework

The overview of our system FIESTA is shown in Fig. 2. Firstly, we get depth measurements from stereo, RGB-D sensors or monocular depth estimation, and pose measurements from external devices such as GPS and Vicon, or internal pose estimation such as VIO (Visual-Inertial Odometry). Then we use **raycasting** to integrate them into the occupancy grid map, which is the data structure we choose to store the occupancy information in **FIESTA**. In this process, all voxels that change their occupancy status are added into two queues named $insertQueue$ and $deleteQueue$ respectively. After that we merge these two queues into one queue named $updateQueue$ by a procedure called **ESDF Update Initialization** and update all the voxels that ESDF may change using ESDF Update Algorithm based on BFS.

IV. Data Structures

This section describes all data structures used in FIESTA. Occupancy Grip Map is used to integrate occupancy information. Voxel Information Structures store all information with respect to voxels. Indexing Data Structure is the mapping from a voxel coordinate towards corresponding Voxel Information Structure. And Doubly Linked Lists are used especially for updating ESDF efficiently when an occupied voxel becomes free.

A. Occupancy Grid Map & Voxel Information Structure

Occupancy Grid Map is used in our method to store the probability of occupancy of voxels. It keeps integrating new occupancy information data obtained from raycasting when a new depth measurement is given. The probabilistic occupancy information is stored in a Voxel Information Structures (VIS) of voxels. Besides that, there are some other important members in VIS. Table I lists all useful members with their meanings and abbreviations in the pseudocode of this paper. The last two rows in it are two methods of a voxel. dll will be described in Section IV-C, and $nbrs$ means all observed neighbors, with the designated connectivity, such as 6-connectivity or 26-connectivity.

B. Indexing Data Structure

Depends on whether the bounding box of the planning area is pre-known or not and the memory reserved for this algorithm, an array or a hash table is used for indexing work, i.e. mapping a voxel coordinate towards the corresponding pointer of VIS.

In detail, if the bounding box for the planning area is pre-known and the memory is large enough, pointers of VIS of all voxels can be simply put into an array. In this way, when a voxel is queried by its coordinate, we just need to compute the index of it inside that array and then return the corresponding pointer of VIS. On the contrary, if the memory is inadequate or the bounding box is unknown, a hash table is required to convert a voxel coordinate to its corresponding pointer of VIS. With this method, memory consumption is minimized, because all voxels in this data structure have been observed. However, since the number of look-up operations in a hash table is enormous, this will make the performance much worse than the array implementation.

Thus we give an operatable trade-off of these two by making $(block\_size)^3$ voxels to a block, just like what Voxel Hashing [16] did. Hash table here is only used to manage blocks. After we calculate block coordinate from voxel coordinate, that hash table is used to find the corresponding block. All pointers of VIS of all voxels in that block are stored in an array with respect to that block. Because the size of the hash table for blocks is much smaller than that for voxels, the performance can be improved by more usage of memory. We can even use bitwise operation to speed up the converting work, by using $block\_size$ as an integer power of 2, such as 4, 8, 16 etc.

No matter which data structure is used, the time complexity for look-up operation is always $\Theta(1)$ on average. Though the speed-up factor of different data structures is only constant-level, it is crucial for systems requiring real-time performance on limited resources.

C. Doubly Linked Lists

Doubly Linked Lists(DLLs) are used especially for updating ESDF efficiently when an occupied voxel becomes free (more details in Section V-C). A DLL is a linked data structure that consists of a set of sequentially linked nodes,

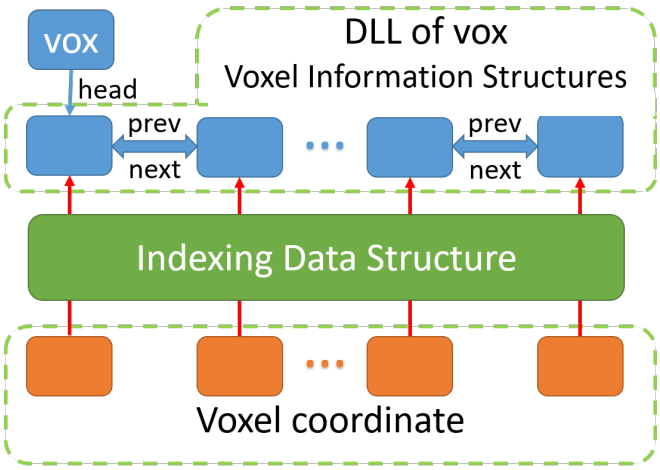


Fig. 3. Diagram among differnt data structures: Indexing Data Structure, Voxel Information Structures and Doubly Linked Lists.

with two link fields, reference to the previous and the next node. We use dll of a voxel vox, i.e. dll method in Table I, to represent all voxels whose closest obstacle is vox. They are linked into a DLL, and the head pointer of this DLL can be obtained by the $head$ member of VIS of vox. The dll method is actually iterating the whole DLL from the head pointer. Fig. 3 shows the relationship among Indexing Data Structure, Voxel Information Structures and Doubly Linked Lists.

To the beginning, the closest obstacles of all voxels are initialized to **Ideal Point**, namely Point at Infinity. We use $\mathcal{IP}$ to represent it. When a voxel is newly observed (more details in Section V-D), it will be added to a dll of $\mathcal{IP}$. As a consequence, all voxels that are newly observed but not yet updated are linked in the dll of $\mathcal{IP}$.

With the help of the Indexing Data Structure and the DLLs, insert into or delete from a DLL is easily implemented in $\Theta(1)$, because (1) find corresponding VIS using Index Data Structure is $\Theta(1)$, and (2) insert or delete a given node in DLL is $\Theta(1)$. We use INSERTINTODLL and DELETEFROMDLL to denotes these two procedures. They have two parameters, which are the header of DLL and the voxel need to be inserted or deleted.

V. ALGORITHMS

Since ESDF value can be easily calculated by the difference of EDF (Euclidean Distance Field) values for both occupancy grid map and its logical complement [4] within the double time of the EDF calculation. In this section, we only explain how to calculate an EDF map. To avoid confusion of notations, we still use ESDF to represent the map we build.

A. Occupancy Integration

Every time we get a pose estimation and a depth estimation with aligned timestamp, a point cloud of obstacles can be obtained. We use raycasting method to integrate the new occupancy information into the Occupancy Grid Map. After that, all voxels that becomes newly observed are inserted into dll of $\mathcal{IP}$ and marked observed. All voxels that change their occupancy state to occupied or unoccupied are added into two queues named $insertQueue$ and $deleteQueue$ respectively.

Algorithm 1 ESDF Updating Algorithm

Require: $updateQueue$ is the queue from ESDF Updating Initialization;
Ensure: All voxels need to be updated should be updated with the minimal computational overhead.

```
1:  while not updateQueue.EMPTY() do
2:      cur ← updateQueue.FRONT()
3:      updateQueue.POP()
4:      for each nbr ∈ cur.nbrs do
5:          if DIST(cur.coc, nbr.pos) < nbr.dis then
6:              nbr.dis ← DIST(cur.coc, nbr.pos)
7:              DELETEFROMDLL(nbr.coc, nbr.pos)
8:              nbr.coc ← cur.coc
9:              INSERTINTODLL(nbr.coc, nbr.pos)
10:             updateQueue.PUSH(nbr)
11:         end if
12:     end for
13: end while
```

B. ESDF Updating Algorithm for Insert-only Case

The ESDF Updating Algorithm is based on a BFS algorithm. To explain it easier, a simplified case where obstacles can only be inserted only is explained firstly. It is the case using a deterministic occupancy grid map where all occupancy integrations only make some voxels become occupied by obstacles. In this case, $deleteQueue$ is empty, we simply add everything in $insertQueue$ into $updateQueue$ after simple initialization (see Alg. 2 Line 1 - 9). We give a not accurate but useful assumption (see more in Section V-E) that a new obstacle will only influence ESDF and the closest obstacle of voxels in a continuous (with respect to the designated connectivity) bounded area. Under this assumption, BFS is used to update neighbors. The complete ESDF Updating Algorithm is described in Alg. 1, where line 5 - 11 is the Euclidean updating way we proposed, like what SDF_Tools [17] did.

C. ESDF Updating Algorithm for Fully Dynamic Case

In the fully dynamic case, obstacles can be either inserted or deleted. They are in the two queues named $insertQueue$ and $deleteQueue$ provided by Section V-A. The measure to deal with $insertQueue$ remains the same. For each voxel in $deleteQueue$, all voxels from the dll of it will be iterated. Status of each term will be cleared back to the observed but not yet updated status, i.e. set the closest obstacle to $\mathcal{IP}$ and set ESDF to ∞. After that, try to update their ESDF by existing closest obstacles of their neighbors. If ever updated, insert this voxel to the DLL of its closest obstacle and add it into $updateQueue$. The complete algorithm that merges $insertQueue$ and $deleteQueue$ to $updateQueue$ is shown

Algorithm 2 ESDF Updating Initialization

Require: $insertQueue$ is all voxels changed to occupied. $deleteQueue$ is all voxels changed to unoccupied.
Ensure: After Alg. 1 using given $updateQueue$, all voxels need to be updated should be updated

```
1: while not insertQueue.EMPTY() do
2:     cur ← insertQueue.FRONT()
3:     insertQueue.POP()
4:     DELETEFROMDLL(cur.coc, cur.pos)
5:     cur.coc ← cur.pos
6:     cur.dis ← 0
7:     INSERTINTODLL(cur.coc, cur.pos)
8:     updateQueue.PUSH(cur)
9: end while
10:
11: while not deleteQueue.EMPTY() do
12:     cur ← deleteQueue.FRONT()
13:     deleteQueue.POP()
14:     for each vox ∈ cur.dll do
15:         DELETEFROMDLL(vox.coc, vox.pos)
16:         vox.coc ← IP
17:         vox.dis ← ∞
18:         for each nbr ∈ vox.nbrs do
19:             if nbr.coc still existing,
          ∧ DIST(nbr.coc, vox) < cur.dis then
20:                 vox.dis ← DIST(nbr.coc, vox)
21:                 vox.coc ← nbr.coc
22:             end if
23:         end for
24:         if vox.coc = IP then
25:             INSERTINTODLL(IP, vox.pos)
26:         else
27:             INSERTINTODLL(vox.coc, vox.pos)
28:             updateQueue.PUSH(vox)
29:         end if
30:     end for
31: end while
```

in Alg. 2. After that, the ESDF Updating Algorithm in Alg. 1 is executed.

D. ESDF Updating Algorithm for Limited Observations

One thing we ignored in the previous two sections is the influence of limited observations. Voxels newly observed may only update its neighbors, without updated by previous existing voxels, which will make the whole system inconsistent. For example, in a 1-D case, only an obstacle at (0) is observed at the beginning. In the next occupancy integration, $(1)-(3)$ are observed and (3) is an obstacle. After executing the Alg. 1, the ESDF of (1) will be updated to 2, which should be 1 from obstacle (0). This happens because in Alg. 1, ESDF of the newly observed ones may not be updated from previous existing voxels. So we need to modify the algorithm, to add Alg. 3 between Line 3-4 in Alg. 1.

Algorithm 3 Patch Code for Limited Observations

Require: Put these code between Line 3-4 in Alg. 1
Ensure: The whole map is updated accurately even with limited observations

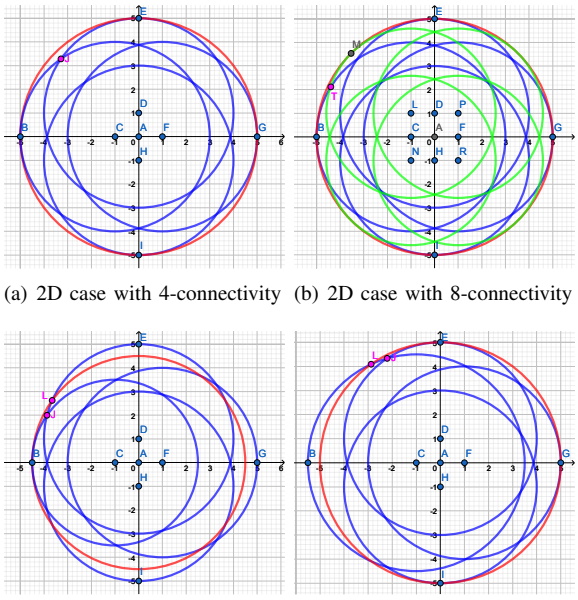
```
1: flag ← false
2: for each nbr ∈ cur.nbrs do
3:     if DIST(nbr.coc, cur.pos) < cur.dis then
4:         cur.dis ← DIST(nbr.coc, cur.pos)
5:         DELETEFROMDLL(cur.coc, cur.pos)
6:         cur.coc ← nbr.coc
7:         INSERTINTODLL(cur.coc, cur.pos)
8:         flag ← true
9:     end if
10: end for
11: if flag then
12:     updateQueue.PUSH(cur)
13:     continue
14: end if
```

E. Theoretical Analysis

1) Optimality: Our algorithm is equivalent to update ESDF value of a voxel by the shortest Euclidean distance between this voxel and closest obstacles of its neighbors. The ESDF value is always an "Euclidean" distance. It is obviously accurate than quasi-Euclidean, because the latter uses the length of the broken line as ESDF value. However, no matter which connectivity we choose, ESDF updating algorithm based on BFS cannot be exactly accurate. We will prove this conclusion below, and gives the relationship between connectivity and optimality.

Think about a 2D case whose connectivity is 4. The point need to be updated is the origin, where its neighbors are $nbrs = \{(0,1),(0,-1),(1,0),(-1,0)\}$. Assume the closest obstacle of theirs are $obs_{nbrs} = \{(0,5),(0,-5),(5,0),(-5,0)\}$ correspondingly. According to the ESDF update algorithm, the distance of origin point will be updated to 5, with the closest obstacle be one of obs_{nbrs}. As shown in the Fig. 4(a), the center of the red circle is at the origin and its radius is 5, centers of blue circles are points in $nbrs$ and their radii are 4. With the assumption that obs_{nbrs} are the closest obstacles of $nbrs$, there are no obstacles inside any blue circle. However, there still remains space (called remaining space) that is outside the blue circles while inside the red circle. The distance between points in that space and $nbrs$ are farther than the distance between obs_{nbrs} and $nbrs$, while the distance between points in that space and origin are closer than the distance between obs_{nbrs} and origin. Fig. 4(b) also shows a case whose connectivity is 8. In fact, no matter which connectivity we choose, this algorithm cannot be accurate, because using finite small circles inside a big circle to cover the big circle is impossible, and there always remains space to give counter-examples.

Actually, this is the worst case for ESDF Updating Algorithm using BFS. If one of obs_{nbrs} moves to another



(a) 2D case with 4-connectivity (b) 2D case with 8-connectivity

(c) Making obstacle B (the left-most point) closer to origin (d) Making obstacle B (the left-most point) farther to origin

Fig. 4. Circles Illustration for the Optimality Analysis

point on or inside its original corresponding blue circle, as shown in Fig. 4(c), the closest obstacle of origin will be changed and ESDF of origin will be smaller, which in return makes the red circle and remaining space smaller. If it moves outside its original corresponding blue circle, as shown in Fig. 4(d), that blue circle becomes bigger and also makes remaining space smaller. The error comes from the difference between the calculated distance from the algorithm and the actual distance. From the illustration, error decreases as radius increases. With the consideration of few occurrences of the worst cases, and the fact that only integer points inside the remaining space will influence our algorithm, the Root Mean Squared (RMS) Error of our algorithm is acceptable in practice. The experimental results will be shown in Section VI-B.

2) Time Complexity: For the Alg. 2, every voxel which is either in $insertQueue$ or whose closest obstacle is in $deleteQueue$ is handled only once. If we use an FIFO queue, the time complexity is $\Theta(k)$, where k is the number of all necessary voxels needed to be handled.

And for the Alg. 1, if we use a priority queue to do the BFS procedure, we are sure that all voxels are popped from $updatequeue$ and handled only once to update its neighbors. Then the time complexity is $\Theta(n \log n)$, where n is the number of all voxels need to be updated, and the $\log n$ factor comes from the nature of priority queue.

3) Space Complexity: As we described in Section IV-B, comparing to the pure occupancy grid map based on a hash table, only VIS is modified for our system. so it is only constant-level higher than pure occupancy grid map. From the data structures we design, we can even make it equal to $\Theta(m)$ if we choose to use $block\_size = 1$, where m is the number of all ever observed voxels, though it is not high-

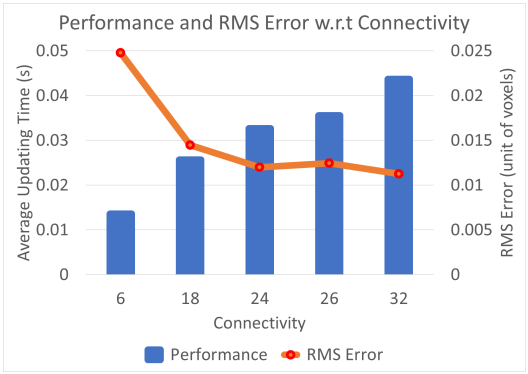


Fig. 5. Time and RMS Error with respect to different Connectivities. The x-axis is the type of connectivity. The left y-axis is the performance measured by average updating time, the right y-axis is the RMS Error in the unit of voxels.

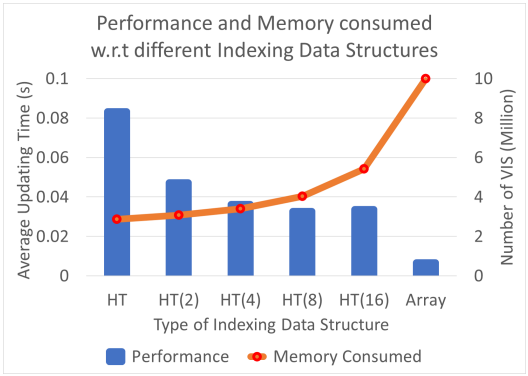


Fig. 6. Time and Space with respect to different Indexing Data Structures. The x-axis is the type of Indexing Data Structure, where HT(n) means Hash Table with $block\_size = n$. The left y-axis is the performance measured by average updating time, the right y-axis is the memory consumption measured by the number of VIS.

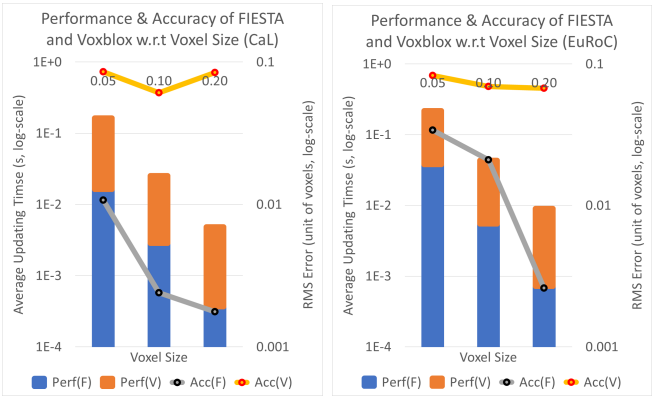
efficient. Thus it is significant to have highly customizable Indexing Data Structures with consideration of requirements of the real system.

VI. Results

A. Experiments on Real-world Datasets

In this section, we test different parameters of our system FIESTA, and then we compare it with Voxblox in different level of voxel size. All the experiments in this part are using i7-8700k at 3.7GHz and only one thread is used.

1) Parameters Tuning: There are a lot of parameters inside our system, such as Connectivity and which Indexing Data Structure is used. A set of experiments will be provided to test the optimal parameters for our system with respect to the accuracy, performance and memory consumption. Our system asks for ESDF updates at a frequency of every 0.5s, and the performance is measured by the average updating time. The largest memory consumption is VIS, so we use a number of VIS as a measurement of memory consumption. After running the whole dataset, we establish a K-D Tree using the occupancy information obtained from an occupancy grid map and then query for the distance for each voxel inside map as ground truth. After that Root Mean Squared (RMS)



(a) Cow_and_Lady Dataset (b) EuRoC Dataset

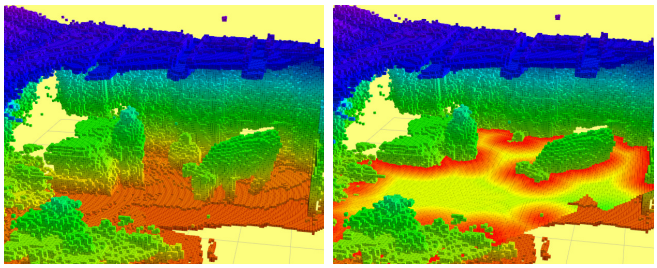
Fig. 7. Performance(Perf) and Accuracy(Acc) between FIESTA(F) and Voxblox(V) with respect to Voxel Size compared using Cow_and_Lady and EuRoC Datasets. The x-axis is voxel size. The left y-axis is the performance measured by average updating time, the right y-axis is the RMS Error in the unit of voxels.

Error is used in the unit of voxels to demonstrate the scale of error. In this section, Cow_and_Lady dataset[3] [9] with an RGB-D camera, which contains point cloud data and pose information, is used as the dataset.

First, we fix the Indexing Data Structure to be a hash table with $block\_size = 8$ and test accuracy with respect to connectivity. We try 6-(faces), 18-(faces and edges), 26-(faces, edges, and corners), 24-(faces, edges and 2-step faces, i.e. Manhattan distance less than or equal to 2) and 32-(combine of previous two) connectivities. More connectivity results in more neighbors to be updated therefore gives a higher accuracy with worse computational performance. Thus we need to choose one trade-off in both performance and accuracy. Fig. 5 shows the result. From the illustration, 24-connectivity is one of the best choices for both RMS Error and performance. Then, different Indexing Data Structures are compared. As shown in the Section IV-B, we give a lot of choices to provide a trade-off between performance and memory consumption. We fix the connectivity to 24, and then test array-implemented and hash-table-implemented with $block\_size = 1, 2, 4, 8, 16$. Fig. 6 shows the result. According to the result, if the map boundary is unknown, a hash table with $block\_size = 8$ is one of the best choices for both time and space complexity. Otherwise, the array has the best performance.

2) Comparison with Voxblox: A comparison between FIESTA and Voxblox in term of performance and accuracy in different voxel size is provided. Cow_and_Lady dataset with an RGB-D camera and EuRoC dataset with a stereo camera are used as datasets for this comparison. We choose to use a hash table with $block\_size = 8$ as the Indexing Data Structure and 24-Connectivity for this comparison. Fig. 8 shows the results, it is clear that our system outperforms Voxblox in an order of magnitude in both performance and accuracy. Visualization of Occupancy Grid Map and a slice of ESDF Map using our system with $voxel\_size = 0.05$, running Cow_and_Lady dataset is shown in Fig. 7.

[3]https://projects.asl.ethz.ch/datasets/doku.php?id=iros2017



(a) Occupancy Grid Map (b) Slice of ESDF Map

Fig. 8. Visulization of Occupancy Grid Map and a slice of ESDF Map using our system with $voxel\_size = 0.05$, running Cow_and_Lady Dataset

B. Tests of Quadrotor Motion Planning in Simulation

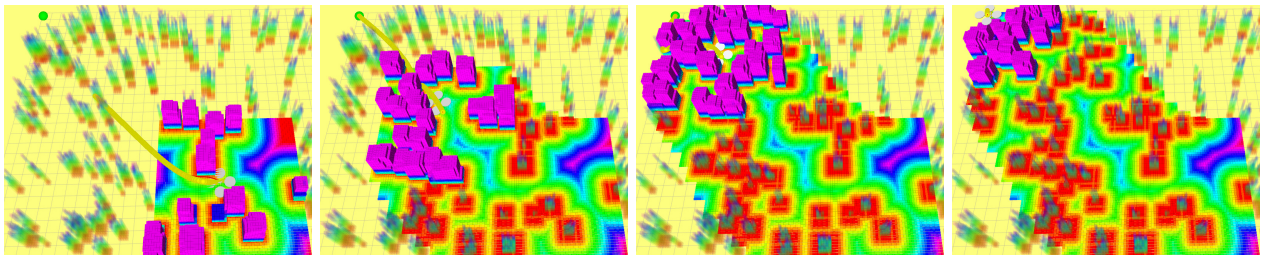
To prove that our proposed map can be used for motion planning, we present simulated quadrotor flight, where the motion planning method is adopted from [1]. In simulation, a map as well as the starting and destination points of the MAV are generated randomly. Only obstacles within a radius around the current position of MAV can be sensed. Our system FIESTA builds global ESDF Map incrementally, to help to motion planning algorithm running efficiently. Our sample results are shown in Fig. 9.

C. Autonomous Quadrotor Onboard Experiments

Onboard experiments are conducted in unknown cluttered environments using the same motion planning approach as in Section VI-B. The platform we use is a quadrotor equipped with a Velodyne VLP-16 3D Lidar, which is used for both pose estimation and depth measurement. All state estimation, motion planning, control, and our proposed mapping system are running onboard on a dual-core 3.00GHz Intel i7-5500U processor. Although our ESDF map can finish an update within 20ms, we update ESDF at 20Hz, due to the low requirement of the flight speed in our experiments. Also, updating ESDF map at a relatively low frequency saves a lot of onboard recourses for other modules such as the planning and localization. Snapshots of the experiment and the incremental mapping and planning results are given in Figs. 1 and 10.

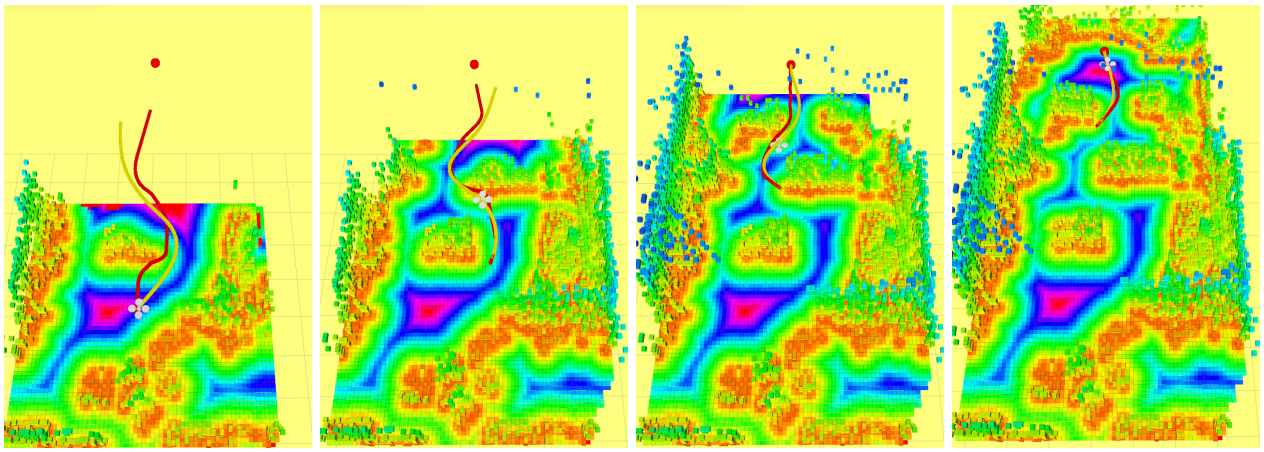
VII. Conclusion

In this paper, a lightweight and flexible mapping framework, FIESTA, was proposed. FIESTA builds a global ESDF map incrementally, which is important to motion planning. By introducing two independent updating queues for inserting and deleting obstacles separately, and using Indexing Data Structures and Doubly Linked Lists to maintain voxels, our algorithm updates as few as possible nodes using a BFS framework. Both theoretical and practical analysis in term of time and space complexity, accuracy and optimality are given in this paper. We proved that the updating rule of our method outperforms quasi-Euclidean one for Voxblox. In practical, using a hash table with $block\_size = 8$ and 24-Connectivity,



(a) (b) (c) (d)

Fig. 9. Simulation Experiments Results



(a) (b) (c) (d)

Fig. 10. Onboard Experiments Results

our system beats Voxblox in an order of magnitude in term of accuracy and time complexity. The proposed ESDF map is also integrated into a completed quadrotor system. With the simulation and onboard experiments, we validate our system by building global ESDF map incrementally high-efficiently.

REFERENCES

[1] B. Zhou, F. Gao, L. Wang, C. Liu, and S. Shen, "Robust and efficient quadrotor trajectory generation for fast autonomous flight," *IEEE Robotics and Automation Letters (RA-L), Under Review*, 2019. [Online]. Available: https://www.dropbox.com/s/0ryz611l5l8cdxgx/ral_iros2019_boyu.pdf?dl=0

[2] K. M. Wurm, A. Hornung, M. Bennewitz, C. Stachniss, and W. Burgard, "Octomap: A probabilistic, flexible, and compact 3d map representation for robotic systems," in *Proc. of the IEEE Intl. Conf. on Robot. and Autom. (ICRA)*, vol. 2, Anchorage, AK, US, May 2010.

[3] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon, "Kinectfusion: Real-time dense surface mapping and tracking," in *2011 IEEE International Symposium on Mixed and Augmented Reality*. IEEE, 2011, pp. 127–136.

[4] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "Chomp: Gradient optimization techniques for efficient motion planning," in *Proc. of the IEEE Intl. Conf. on Robot. and Autom. (ICRA)*, May 2009, pp. 489–494.

[5] H. Oleynikova, M. Burri, Z. Taylor, J. Nieto, R. Siegwart, and E. Galceran, "Continuous-time trajectory optimization for online uav replanning," in *Proc. of the IEEE/RSJ Intl. Conf. on Intell. Robots and Syst.(IROS)*, Daejeon, Korea, Oct. 2016, pp. 5332–5339.

[6] F. Gao, Y. Lin, and S. Shen, "Gradient-based online safe trajectory generation for quadrotor flight in complex environments," in *Proc. of the IEEE/RSJ Intl. Conf. on Intell. Robots and Syst.(IROS)*, Sept 2017, pp. 3681–3688.

[7] V. Usenko, L. von Stumberg, A. Pangercic, and D. Cremers, "Real-time trajectory replanning for mavs using uniform b-splines and a 3d circular buffer," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 215–222.

[8] P. F. Felzenszwalb and D. P. Huttenlocher, "Distance transforms of sampled functions," *Theory of computing*, vol. 8, no. 1, pp. 415–428, 2012.

[9] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto, "Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning," in *2017 Ieee/rsj International Conference on Intelligent Robots and Systems (iros)*. IEEE, 2017, pp. 1366–1373.

[10] F. Gao, W. Wu, Y. Lin, and S. Shen, "Online safe trajectory generation for quadrotors using fast marching method and bernstein basis polynomial," in *Proc. of the IEEE Intl. Conf. on Robot. and Autom. (ICRA)*, Brisbane, Australia, May 2018.

[11] J. Chen, T. Liu, and S. Shen, "Online generation of collision-free trajectories for quadrotor flight in unknown cluttered environments," in *Proc. of the IEEE Intl. Conf. on Robot. and Autom. (ICRA)*, Stockholm, Sweden, May 2016, pp. 1476–1483.

[12] F. Gao, W. Wu, W. Gao, and S. Shen, "Flying on point clouds: Online trajectory generation and autonomous navigation for quadrotors in cluttered environments," *Journal of Field Robotics*. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21842

[13] F. Blochliger, M. Fehr, M. Dymczyk, T. Schneider, and R. Siegwart, "Topomap: Topological mapping and navigation based on visual slam maps," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 1–9.

[14] H. Oleynikova, Z. Taylor, R. Siegwart, and J. Nieto, "Sparse 3d topological graphs for micro-aerial vehicle planning," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 1–9.

[15] P. R. Florence, J. Carter, J. Ware, and R. Tedrake, "Nanomap: Fast, uncertainty-aware proximity queries with lazy search over local 3d data," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 7631–7638.

[16] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger, "Real-time 3d reconstruction at scale using voxel hashing," *ACM Transactions on Graphics (ToG)*, vol. 32, no. 6, p. 169, 2013.

[17] UM-ARM-Lab, "sdf_tools," https://github.com/UM-ARM-Lab/sdf_tools, 2014.