FH Vorarlberg
University of Applied Sciences

An Implementation Approach of the Gap Navigation Tree Using the TurtleBot 3 Burger and ROS Kinetic

Master Thesis
for gaining the academic degree

Master of Science in Engineering (MSc)

Vorarlberg University of Applied Sciences
Computer Science – Software and Information Engineering

Advisor
Dr. Ralph Hoch
Prof. (FH) Dr. Hans-Joachim Vollbrecht

Submitted by
Daniel Thomas Groß, BSc
Dornbirn, December 2020

Acknowledgements

I would like to thank everyone who supported me throughout the thesis in one way or another. Many thanks to my advisors Dr. Ralph Hoch and Prof. (FH) Dr. Hans-Joachim Vollbrecht for their help through the thesis. Finally, I want to thank my family who supported me during the entire course of my studies, and without whom this would not have been possible.

Kurzreferat

Die Erstellung eines räumlichen Modells der Umgebung ist eine wichtige Aufgabe, um das Planen einer Route durch die Umgebung zu ermöglichen. Abhängig von der Anzahl der Sensoren sind verschieden Arten der Erstellung eines räumlichen Modells möglich. Diese Arbeit stellt einen Implementierungsansatz des Differenz Navigation Baum vor, welcher für die Verwendung mit Robotern konzediert ist, welche nur eine Limitierte Anzahl von Sensoren besitzt. Der Differenz Navigation Baum ist eine Baum Struktur, welche auf Tiefensprüngen basiert, welche aus den Daten eines Laser Scanner ermittelt werden. Unter Verwendung des Simulierten TurtlebBot3 Burger und ROS Kinetic wird ein Programmiergerüst entwickelt welches die Theorie des Differenz Navigation Baum umsetzt. Das Programmiergerüst ist so aufgebaut, dass eine Verwendung mit verschieden Robotern und Sensoren möglich ist. Dies wir erzielt in dem die Erkennung der Tiefensprünge getrennt von der Konstruktion und Aktualisierung des Differenz Navigation Baum erfolgt.

Abstract

The creation of a spatial model of the environment is an important task to allow the planning of routes through the environment. Depending on the number of sensor inputs different ways of creating a spatial environment model are possible. This thesis introduces an implementation approach of the Gap Navigation Tree which is aimed for usage with robots that have a limited amount of sensors. The Gap Navigation Tree is a tree structure based on depth discontinuities constructed from the data of a laser scanner. Using the simulated TurtleBot 3 Burger and ROS kinetic a framework is created that implements the theory of the Gap Navigation Tree. The framework is structured in a way that allows using different robots with different sensor types by separating the detection of depth discontinuities from the building and updating of the Gap Navigation Tree.

Contents

List of Figures

Listings

1. Introduction

Navigating from one place to another and remembering the environment is no problem for many kinds of animals and humans. The information they use is not necessarily accurate or quantitatively, they define the space by remembering a few landmarks [YJC12]. Navigating through an environment is an important task for mobile autonomous robots [Thr02]. To be able to plan a route and navigate from point A to point B they need a representation of their environment like humans and animals do. The representation of the environment allows the robot localise itself in the environment, plan the route and then navigate. The abilities of the robot define how a spatial model can be constructed and also how accurate this spatial model is.

1.1. Problem Statement

Assuming a robot has only a very limited sensing ability. Let the robot have a LIDAR sensor and the ability to know if it is driving forward or backwards and if it is rotating clockwise or counter-clockwise. It does have the local odometry information to determine the local x and y position in the environment and orientation. The robot shall be able to create a local spatial model so once it has discovered his environment, it is able to plan routes from A to B.

1.2. Aim of the Work

The aim of the work is to create a Robot Operating System (ROS) framework which allows a robot with limited sensing capabilities the construction of a spatial model. It is assumed that the robot has a Light Detection and Ranging (LIDAR) sensor, the ability of detecting if it is turning clockwise or counter-clockwise and knows if it is driving forwards or backwards. For this purpose, the TurtleBot 3 Burger platform is chosen. The implemented framework shall be implemented in a way that it consists of multiple ROS nodes. This shall allow easy adaption to different sensors which returns spatial information.

1.3. Methodological Approach

First a ROS node was implemented which extracts the valuable information from the environment and makes this information available to other ROS nodes. After having the valuable information, a second ROS node was created which takes the information from

the first ROS node to analyse it and extract the information which allows to create a spatial model of the environment. The third ROS node then takes the information from the second ROS node and creates the spatial model which again is made available for other ROS nodes.

1.4. Structure of Work

In Chapter 2 the reader is introduced to basics of robotics. The robotic framework ROS is explained as well as the used robot TurtleBot 3 Burger and the used simulation software. Furthermore, an introduction to mapping and navigation is given. Chapter 3 discusses theories and implementation approaches which use limited sensing to create a spatial model of the environment. The concept for the implemented framework is discussed in Chapter 4. It introduces the foundations which are then implemented in Chapter 5. In Chapter 6 the accuracy and robustness of the implemented framework is discussed. This is done by defining some test environments and driving the robot through those environments while running the implemented framework. The last Chapter 7 discusses the implemented framework with its limitations and also what needs to be improved in the future.

2. Technical Background

This chapter discusses information necessary for understanding the remainder of this thesis. In the following sections, important terms are defined and explained. First the framework used to develop the software is explained, followed by the robot used and the simulator for it. Furthermore, the terms mapping, localisation and navigation are defined and explained.

2.1. ROS

Robot Operating System (ROS) is the used system for the communication, specifically the version kinetic [Fou18]. When reading Operating System, one might think of process management and scheduling, which is not the case for ROS [gee20]. ROS is a framework which makes it possible to decentralise the control of the robot and encourages the development of reusable code by separating the software into small modules called nodes. The idea is to separate the computing machines into onboard machines and offboard machines where those nodes are run [Adn15][Qui+09]. Offboard machines are the nodes not located on the robot and onboard machines are those that are located on the robot. The connection between the onboard and offboard machines is usually established using WiFi.

ROS makes development of software for the robot possible in multiple languages instead of being restricted to a single programming language. Supported languages are C++, Python, Octave and LISP [Qui+09]. Robot software development is done by creating software modules, the so called nodes, which are ideally designed for one specific task. The modules are called nodes because because several nodes communicating with each other can build a graph network. The developed nodes can run on an offboard machine or onboard machine. An onboard machine for example can be a Raspberry Pi mounted on the robot. Due to the limited computing power of a Raspberry Pi it is not suited for computational heavy tasks. Therefore, on the robot only nodes will run that run tasks which does not require high computing power. Such a task can be controlling the motors, reading the sensor data. Computational more expensive tasks, for example, image processing might be better suited for an offboard machine because there is more computing power is available.

ROS enables easy data exchange between nodes and offers multiple possibilities to implement communication between them. The possible communication methods between nodes are topics, services and actions. Topics are a publisher and subscriber communication method. Multiple nodes can subscribe to a topic or publish to a topic. This method of communication is asynchronous, because there is no communication between

publisher and subscribers which ensures that a message was received. For a function like behaviour ROS offers the service functionality to communicate with other nodes. A node can offer one or multiple services for other nodes to be called. The service defines the information it requires and the response after processing the request. So far, the mentioned communication possibilities do not have the option to interrupt or get information about the processing status. The so-called actions offer the functionality of interruption and feedback. The action server which offers the action functionality can give the calling node feedback during processing. Another functionality the other communication method does not have is the possibility to cancel the request during it is already being processed.

For the communication between the nodes messages are used which are defined by msg-files. The used description method is done using a language-neutral Interface Definition Language (IDL) [Qui+09]. Each msg-file contains number of fields and each field is defined by a datatype and name. This allows ROS to support cross-language development. There are pre-defined datatypes, but it is also possible to use another message as the data type. From this definition native implementations are generated by a code generator for each supported language. Actions and services also have IDL descriptions. The description of a service contains definitions how the semantic description of the message is setup. For the action the description has three section consisting of the goal definition, result definition and feedback.

ROS also has a powerful development toolset which contains tools for debugging, introspection, plotting and visualisation of the state of the system to name a few of them. All core functionality and introspection tools can be used from the command line from remote without a Graphical User Interface (GUI). Rviz is one of the graphical tools provided by ROS [Ope20a][Ope20c]. The tool can visualise the robot itself, sensor data in two-dimensional (2D) and three-dimensional (3D) space and camera images. Another tool provided by ROS is rqt, which is based on the Qt framework and allows the development of a GUI for the robot [Ope20a][Ope20b]. It comes with a library of plugins that can be used to create an interface that visualises various data of the robot. If no standard plugin fits the needs it is also possible to write custom plugins.

2.2. TurtleBot

In this thesis the mobile robot TurtleBot 3 Burger is used. The TurtleBot 3 Burger is chosen because it fulfils the minimal sensor requirements for this thesis and the platform is designed to be used with ROS. Furthermore, the familiarity with it due to the use in lectures at Vorarlberg Applied Science University is a plus. It was designed for use in education, research, hobby and product prototyping and is small and affordable [ROB20c]. The robot is equipped with two motors which each driving one wheel, a Light Detection and Ranging (LIDAR) sensor, gyroscope, acceleration sensor, 3-axis-magnetometer and a Raspberry Pi [gen20b]. With its two separately driven wheels it is classified as a differential wheeled robot. Changing the direction is done by letting the wheels turn with different speeds [ASN09]. With the LIDAR sensor the robot can

detect obstacles with a maximum distance of 3,5m. The sampling rate of LIDAR is 1,8kHz [gen20a]. Development is done using the simulator and the simulated TurtleBot 3 Burger. The needed files to simulate the TurtleBot 3 Burger with Gazebo are provided with the ROS installation [ROB20a].

2.3. Gazebo

In this thesis a simulator is used to develop the software without requiring a physical robot and a place to set up a test environment. Simulators can provide the developer with virtual version of a real robot or a theoretical physical system [KN11]. Furthermore, a simulator allows direct debugging, which is also mentioned as a benefit by Laue et al.[LSR06] and Afzal et al.[Afz+20]. When working with a simulator it is important to know that they cannot represent physical reality and therefore have a gap between simulation and reality because the simulator is an abstraction of the reality [Afz+20]. The simulator used in this work is Gazebo [Ope14a]. Gazebo is the primary simulator when working with ROS and is integrated in ROS by providing a set of Gazebo plugins which allow ROS to communicate with the simulator [Rob20]. Gazebo comes with a list of pre-defined models (see [Fou14]). The pre-defined models include various robots and other elements like a wall or a door which can be used to create a test environment. Gazebo supports building custom models by the means of providing an editor with a graphical editor to create own models [Ope14b]. A more advanced method of creating and editing models is using the Unified Robotic Description Format (URDF) and the Simulation Description Format (SDF) [ROB20b][Fou20]. The URDF specifies the kinematic and dynamic properties of the robot. To describe joint loops (parallel linkages), friction and other properties the SDF is used. A simulation also needs an environment which Gazebo calls a world. For this purpose, Gazebo comes with a building editor which is a graphical interface to create worlds.

2.4. Mapping

The problem of acquiring a spatial model of a physical environment is called mapping. To build a truly autonomous mobile robot, mapping is one of the most important problems that need to be solved. Enabling the robot to navigate through its environment is a common use case. Navigation often comes with the requirement of localisation, which requires a spatial model of the physical environment [Thr02]. Therefore, mapping needs to deal with the selection of the right map representation, localisation and loop closing which will be explained briefly in the next three sections.

2.4.1. Map Representation

The representation of the environment can be done using different decomposition techniques. These techniques include exact decomposition, cell decomposition, fixed decomposition, adaptive decomposition or topological decomposition. The different techniques

differ in the amount of detail the map contains and the complexity of it. An example for the exact decomposition is the continuous-valued map, where the features of an environment can be precisely annotated in continuous space. However, this type of representation is only suitable for 2D representation since the computational complexity explodes with an additional dimension. A map using continuous representation with polygons as environmental obstacles is shown in Figure 2.1.

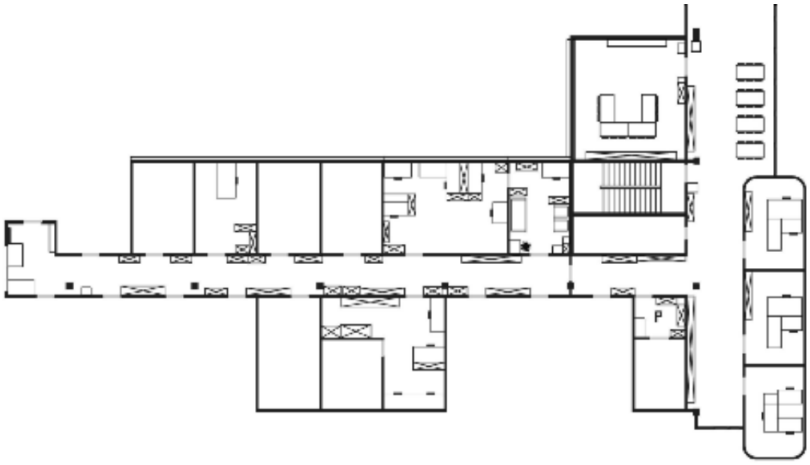


Figure 2.1.: Environment in continuous representation [SNS11].

A simpler approach of continuous representation is a continuous-valued line representation. The features in the environment are represented as straight lines. This can be achieved using line extraction where a line is fitted through several points. Figure 2.2 shows an example of continuous-valued line representation. In a) the real map is shown and in b) the continuous-values line representation. Note that in b) the captured features are those that can be represented with straight lines such as those found at corners and along walls.

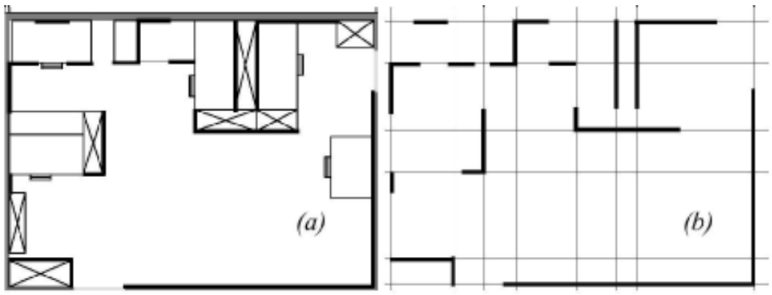


Figure 2.2.: Environment in continuous-valued line representation. a) real map. b) representation with a set of lines [SNS11].

An example for exact cell decomposition is shown in Figure 2.3. For this type of map representation, the given space is separated into areas with free space. In this example the obstacles are polygons and the vertical lines are drawn so they touch a corner of the

polygon. Each free space can be represented as a single node and therefore results in a very compact representation. It is assumed that the robot does not need to know the exact position within the free area. Instead, the ability of the robot to move from one free space to the next adjacent free space is of interest and can be achieved with the resulting graph.

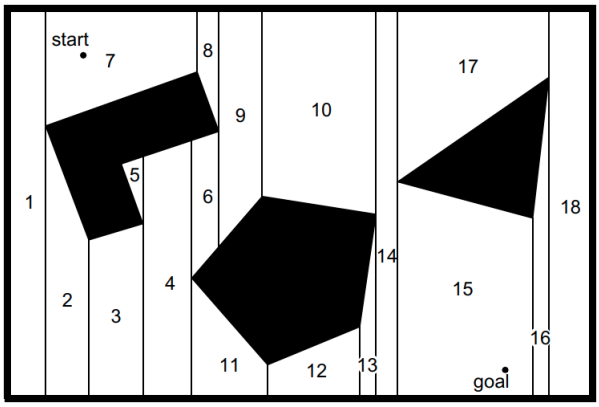


Figure 2.3.: Environment in exact cell decomposition [SNS11].

Using fixed decomposition, one gets a discrete approximation of the real environment. Figure 2.4 shows an example of a fixed decomposition. The occupancy grid, shown in Figure 2.5 is a popular version of the fixed decomposition. Its accuracy depends on the resolution of the grid. If the resolution is chosen too low narrow passageways might disappear.

The adaptive decomposition approach tries to create cells that are as large as possible. The algorithm starts with one large cell which will be broken down until it only contains one type (free or obstacle). An adaptive decomposition example is given in Figure 2.6.

The last decomposition option is the topological decomposition. Figure 2.7 illustrates the topologic map of an indoor office space. The topologic map does not use direct measurements of the environment to gain geometric information and store it. Instead it searches for characteristics in the environment that are most important to localise the robot. The representation is a graph built of nodes which represent areas in the world with certain characteristics. Two areas that are connected and can be traversed by the robot are denoted by two nodes connected by a arc. In the given example of Figure 2.7 a sensor was used to find intersections between halls and rooms [SNS11, p284-293]. All those different decomposition techniques have different advantages and disadvantages and therefore different usages.

2.4.2. Localisation

Localisation is an essential ability during the process of getting a representation of the environment and navigating through the environment. The localisation is the process of determining the position in the environment. It typically uses the robots odometry and

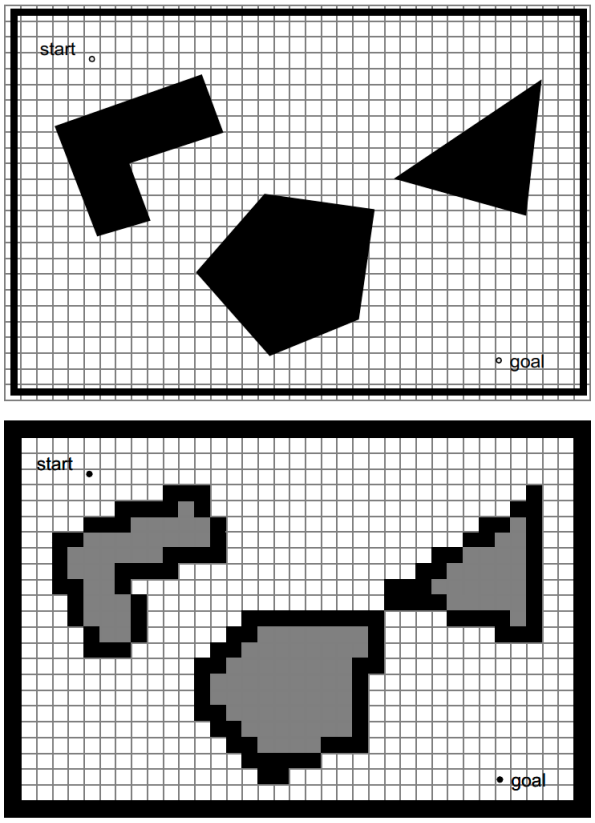


Figure 2.4.: Environment in fixed decomposition [SNS11].

exteroceptive sensors (e.g. ultrasonic, laser, vision sensor). When the robot starts moving from a precisely known position, then the current position during the movement can be tracked using the robot's odometry. Inaccuracy of the odometry causes the position uncertainty to grow as movement progresses. Localisation of the robot in relation to its environment map prevents the uncertainty from growing because with the information from the exteroceptive sensor in combination with the odometry information the robot is then able to localise itself as well as possible in the environment [SNS11, p.296-299].

2.4.3. Simultaneous Localisation and Mapping (SLAM)

Simultaneous Localisation and Mapping (SLAM) is the process of learning a map and at the same time determining the position of the robot. Loop closure is a problem that arises throughout the SLAM process. As heard before when creating a map, it is important for the robot to know where its position in the environment and therefore on the map is. So, when driving and mapping the environment the robot probably will come to a place where it has been before. The robot should then be able to recognise that it has visited that place before. This knowledge can then be used to correct the map [SNS11, p.348ff].

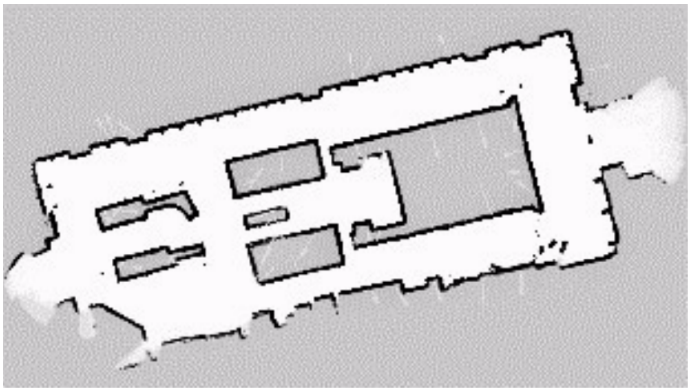


Figure 2.5.: Environment in occupancy grid [SNS11].

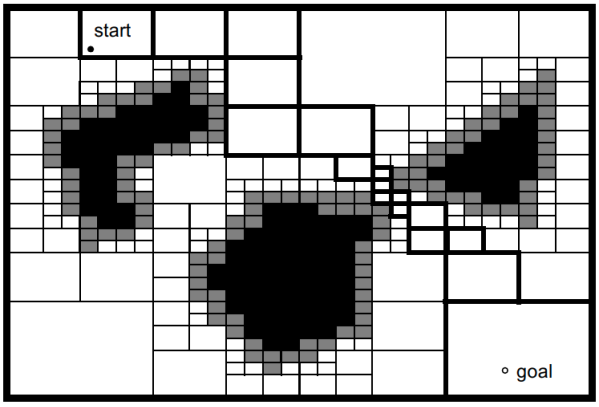


Figure 2.6.: Environment in variable cell decomposition [SNS11].

2.5. Navigation

The challenge of finding a way to a desired destination is called navigation. For finding a way to a desired destination the robot needs to know where it currently is (localisation), where it needs to go (goal) and how it gets there (path). The goal can be any position in the environment. With the current position and the goal, a path needs to be found to get from the current position to the goal. When a path was found the robot needs to follow that path. While following the path the robot needs to react on unforeseen events, for example there could be an obstacle it was not aware while planning the route [SNS11][TB96].

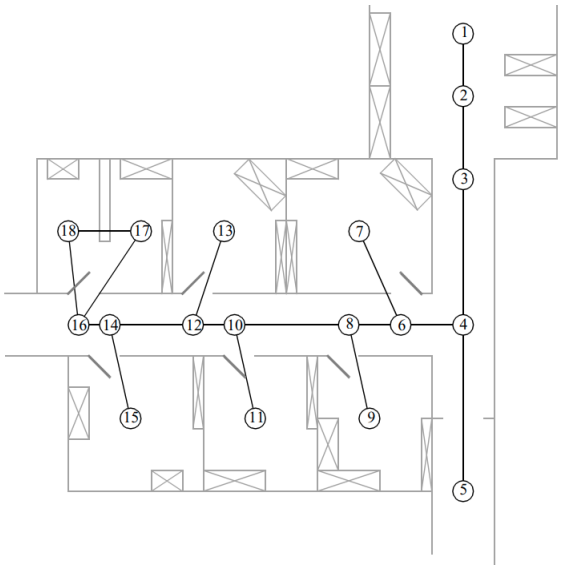


Figure 2.7.: Environment in topological decomposition [SNS11].

3. Related Work

Only little other work use the approach of having a robot with limited sensing and lack of knowledge of the exact position [Zha+10] [TML07]. This chapter discusses approaches which rely on robots with limited sensing capabilities to create a spatial model.

3.1. Real-Time Indoor Mapping for Mobile Robots with Limited Sensing

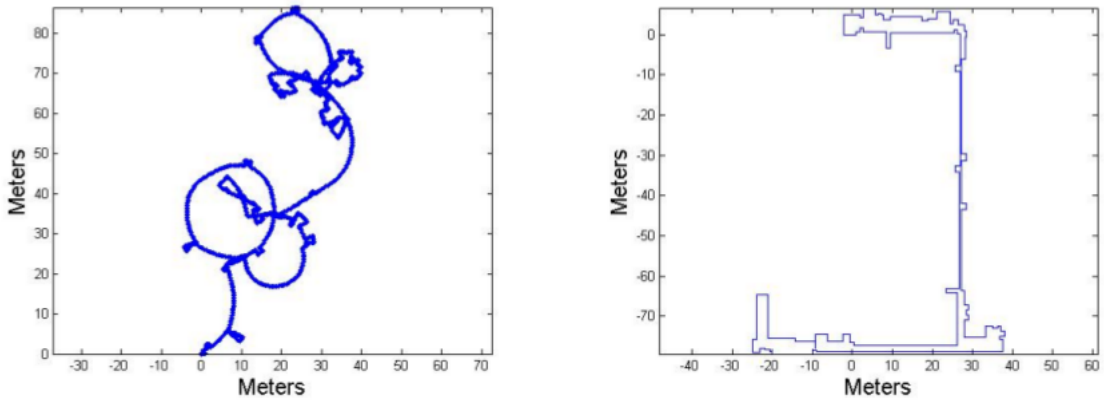
Zhang et al.[Zha+10] propose an approach for creating a map of an indoor environment with a robot that only has a few sensors. The used robot is the *iRobot Create*. It can measure the driven distance, a odometry for estimation purpose of the wall position and the angle turned, a wall sensor on the right side to measure the distance to the wall and two bumper sensors, that are located on the front left and front right side. With the wall sensor having a range of under 10 cm and the bumper sensors being short-rang sensors. The result of their mapping is the outline of the environment as shown in Figure 3.1. In a) snapshot of the hallway being mapped is shown and b) shows the floor plan with the area to be mapped being coloured.

(a) Hallway snapshot (b) Hallway floor plan

Figure 3.1.: Example environment which is a hallway in an office building [Zha+10].

They use a wall-following behaviour to explore the indoor environment and generate the outline. Due to the noise in the odometry its data is only used as an estimation

of the wall position instead of using it as the actual position of the robot. To correct the estimation, they make use of the pre knowledge that the environment consists of straight walls and rectilinear corners. With this knowledge they perform a rectification of the raw odometry angles by classifying them into multiples of $\pi/2$. To prevent errors caused by small furniture or chairs the history of the angles is taken additionally. With a probabilistic approach they maintain N candidate maps. If a hypothesis is consistent with the input data, it gets a higher probability to be the correct one. Figure 3.2 shows two versions of the obtained outline. In a) the non-corrected version of the outline is shown and in b) the corrected version of the outline. To be able to perform loop closing they introduce the Accumulated Turn Counts (ATC). It sums up all the left and right turns to the current distance with a left turn being 1 and a right turn being –1. A loop is indicated by the ATC when it is high positive or low negative.



(a) Robot trajectory from odometry (b) An outline obtained from mapping

Figure 3.2.: Environment representation before correction and after correction [Zha+10].

3.2. Gap Navigation Tree

Tovar et al.[TML07] propose the Gap Navigation Tree (GNT) which is a tree like structure created with one sensor that tracks depth discontinuities. The sensor described is an abstract sensor which can detect discontinuities in depth and tracks them with a so-called gap sensor that has an infinite range. The GNT is then constructed from the information of the sensor. Tovar et al.[TML07] use a robot which is modeled as point with no dimensions. They claim that its abstract sensor can be implement with a variety of sensors such as a laser range finder, sonar, camera or with another sensing system with whom it is possible to detect depth continuities. The gap sensor returns a sequence of gaps denoted as $G(x) = [g_1, \ldots, g_k]$. Where the gap is a notion for a depth discontinuity. With the environment being $\mathbb{R} \subset \mathbb{R}^2$, $G(x)$ represents the sequence at $x \in \mathbb{R}$. $G(x)$ is a cyclic ordering if x lies in the interior of $\mathbb{R}$. The cyclic ordering allows

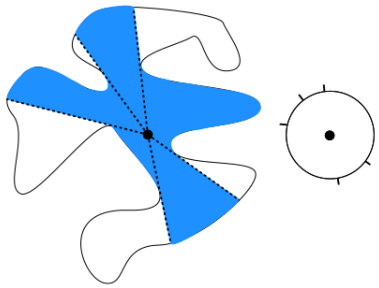


Figure 3.3.: On the left side the environment is shown with the robot being the black dot, blue is the area that can be seen by the LIDAR and the black dotted lines are the laser beams detecting a depth discontinuity. The circle on the right represents the 360° view with the detected depth discontinuities [TML07].

statements such as $[g_1, g_2, ..., g_k] = [g_2, ..., g_k, g_1]$. Figure 3.3 shows an environment on the left and the resulting gaps on the right with the robot being the black dot. The gaps can be caused by soft edge boundary or a edge boundary (see Figure 3.4). If the robot moves along a soft edge boundary like in a) the gap will move clockwise. A gap caused by a edge as shown in b) will stay at the same position.

A gap appearance to the left of x is defined the same way. During gap detection each gap $g_1 \in G(x)$ with $x \in \mathbb{R}$ gets its unique label. Besides a unique label the gap does not contain any other information. If the robot moves only a small amount no change in the sequence will happen. The appearance, disappearance, merging or splitting of a gap are fundamental changes and can occur occasionally. A appearance will occur in the beginning when the detection starts and no information is available. The disappearance occurs if the robot for example approaches a gap that is caused by the edge of a protrusion and the robot moves into it. The merge occurs when an edge causing a gap covers another edge which also causes a gap. In case of a split the robot drives towards an edge if the robot is close enough another edge appears behind the approached gap. A gap that appears will always get a new unique id even if it was a gap that has been tracked before, disappeared and later gets detected again. The control of the robot is defined by a gap chasing motion *chase(g)* with $g \in G(x)$. This motion is a combination of a rotation and forward movement. A rotation is performed to align the robot with the gap. When the robot is aligned with the gap it moves towards it in a straight line. The *chase(g)* terminates when the chased gap disappears. To construct the gap navigation the previous mentioned events of appearance, disappearance, merge and split are used. The four different events appear, disappear, merge and split are shown in Figure 3.5.

Based on these four events the GNT is constructed incrementally as the robot moves along a path τ. The root vertex represents the robot and each detected gap is then represented as a leaf vertex of the root vertex. An update of the GNT is performed when one of the critical events occurs. When a new gap g appears a new child vertex

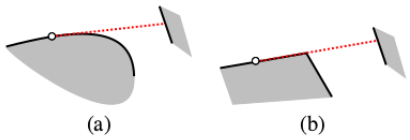


Figure 3.4.: Environment with two different types (a)soft edge, b)edge) of boundaries causing depth discontinuities [TML07].

g is added to the root vertex. If a gap g disappears the vertex g gets removed from the root vertex if it is a leaf vertex. When two gaps g_1 and g_2 merge into g the two vertices g_1 and g_2 become children of the new vertex g. The vertex g is then added to the root as a child. If a gap g splits into g_1 and g_2 two different scenarios are possible which are creating new vertices or using the existing ones. The first is when vertex g has two children g_1 and g_2. In this case vertex g_1 and g_2 become a child of the root vertex and the vertex g gets removed. If the vertex g has no children two new vertices g_1 and g_2 are created and added to the root vertex. The vertex g gets removed from the root vertex. When adding a vertex or vertices the cyclic ordering must be preserved. Figure 3.6 shows the process of constructing a GNT for the given example environment. The robot and the root of the GNT are represented as a black dot. The light dotted lines in Figure 3.6.a show the boundaries of the aspect cells.

Initially the robot detects g_1 and g_2 which is shown in Figure 3.6.a because it is not sure if the gaps can split, they are drawn as circle in the tree. In Figure 3.6.b the robot is moving towards gap g_1. Now the gap g_3 is detected and added to the tree as rectangle leaf because this node cannot split due to the knowledge at the position in Figure 3.6.a. As the robot continues to move towards gap g_1 the gaps g_3 and g_4 merge into g_4 which is shown in Figure 3.6.c. The merged gap g_4 is drawn as a circle because it is known that this gap can split. With gap g_1 disappearing in Figure 3.6.d the only gap to chase is gap g_4. Gap g_4 will then split into g_3 and g_4 while gap g_1 gets visible again for the robot Figure 3.6.e.

Now the gap g_1 is visualised as a rectangle because it is known that it can't split. The robot will continue with moving towards g_2 because it is uncertain if the gap can split or not. When chasing gap g_2 the gap g_3 will disappear and g_5 will appear as it is shown in Figure 3.6.f. Gap g_2 will disappear because the robot is now at the position where g_2 was. If the robot now would move back towards g_1 the gap g_2 would be visualised as a rectangle because it is known that it can't split.

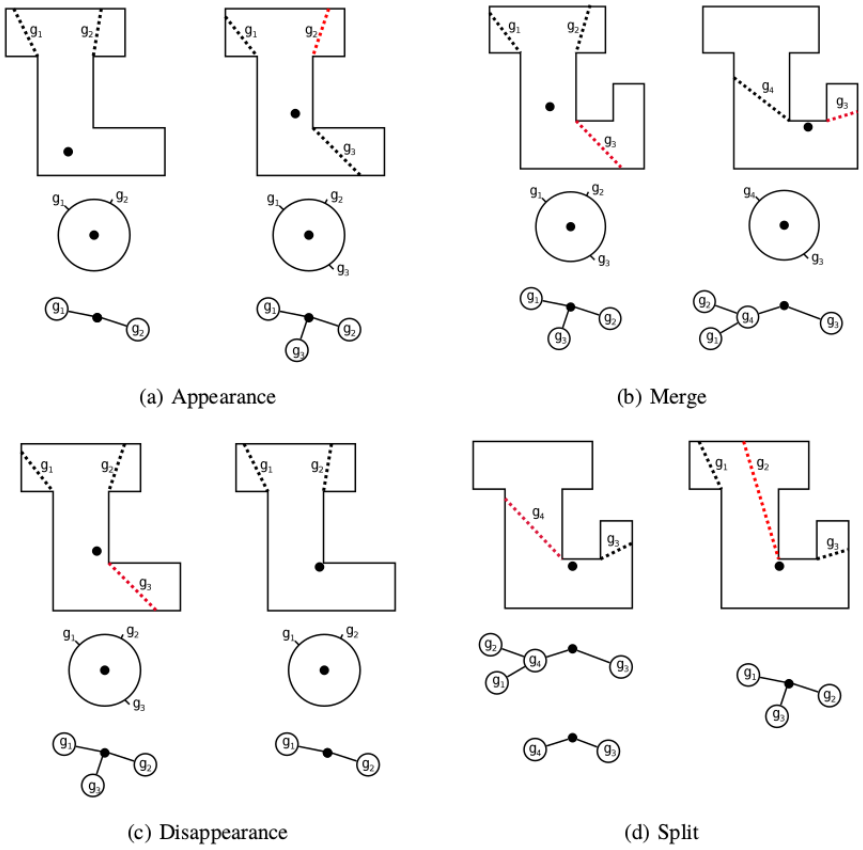


Figure 3.5.: From a) to d) the four different critical events appearence, merge, dissappearence and split are shown with its inpact on the tree [TML07].

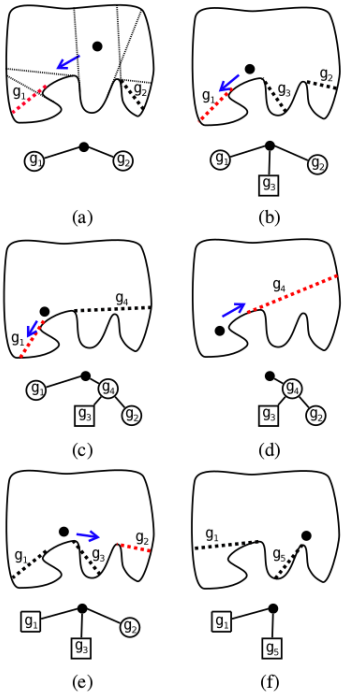


Figure 3.6.: Ilustration how the tree gets built while the robot moves through the environment [TML07].

4. Concept

This chapter first discusses limitations of the GNT found during initial experiments and the chosen solutions. Then the structure of the software is discussed by defining how the depth discontinuities can be detected, how the critical events and move events can be detected and how the tree is constructed.

4.1. Limitations of the Gap Navigation Tree

Analysing the approach of Tovar et al.[TML07] and initial experiments showed that the given approach has limitations that are not clear at first sight. As mentioned in Section 2.2, for this thesis the robot TurtleBot 3 Burger is used. The robot is equipped with LIDAR that has a range of 3,5m. Tovar et al.[TML07] assume that the robot can see all depth discontinuities no matter how far away they are. This makes it possible to track a depth discontinuity until it either splits or merges. With only having a range of 3,5m the proposed concept of Tovar et al.[TML07] fails. For illustration let us assume the LIDAR of the TurtleBot 3 Burger detects a depth discontinuity in 2,5m distance. The robot then starts moving in the opposite direction of the depth discontinuity, which causes the distance to the depth discontinuity getting larger. At one point the depth discontinuity will be out of range for the LIDAR of the TurtleBot 3 Burger. Due to the definition that a disappearing depth discontinuity will be labelled new if it reappears all information connected to it gets lost. For the GNT this means every information connected to that node in the tree is lost and therefore information of the environment.

To fulfil the requirement of not losing a depth discontinuity the environment must be less or equal than 3,5m x 3,5m in size. As stated by Khader et al.[KC20] LIDAR sensors can have different ranges. The detection range can be up to 200 meters depending on the type of system. To be able to test complex larger environments with the TurtleBot 3 Burger the decision was made to modify the model in the simulation. In this folder the file turtlebot3_burger.gazebo.xacro needs to be modified. This file gives gazebo the information how the model needs to be simulated. Listing A.1 shows the definition of the LIDAR, which needs to be changed. From line 16 to 20 the range of the LIDAR is defined. Here the value of <max> needs to be changed to 10, the current value should be 3,5. With this change the LIDAR is now simulated with a range of 10 meters.

Robot used by Tovar et al.[TML07] is modelled as a point which brings up issues when applying the chase(g) motion to a real robot. Recall, *chase(g)* is the movement towards a detected gap. It is described as "*the robot rotates to align its heading with the gap and moves forward in unit speed*" by Tovar et al.[TML07]. It is added that "*the robot uses sensor feedback to continue the motion, which is guaranteed to be collision free, except*

for tangential motions along the boundary"[TML07]. Termination of the chase motion happens as soon as the chased gap is no longer visible. With a real robot two things will happen. The first thing is that the robot will bump into the wall because of the straight alignment towards the gap. Secondly, termination will not be triggered because the robot will get stuck in front of the gap and therefore the discontinuity will still be detected. The chase motion will need to consider that the chased gap must move to the left or the right side of the robot at one point. Furthermore, it needs to perform a wall following motion.

Another question that arises is how the width of the robot is considered during the gap detection. A point representation of a robot will theoretically fit through every gap. With an actual robot or simulated version of a real robot a check needs to be performed if the robot fits through the gap or not. For the tree it is important to only represent gaps that the robot can fit through.

When the robot is close to a wall the search for depth discontinuity using a LIDAR returns false gaps if the wall is long enough. This happens due to two consecutive data points of the LIDAR having a large enough change in the detection distance. This problem is not addressed and needs to be solved to provide valid data for the GNT.

4.2. Software Structure

This section covers the structure of the software and the task of each module contained in the software structure. First the detection of the depth discontinuities is discussed, followed by the detection of critical events, the construction of the tree, how to chase gaps and finally how the complete system looks like.

4.2.1. Detection of Depth Discontinuities

Tovar et al.[TML07] describe a very abstract sensor to detect the depth discontinuities. In thesis the TurtleBot 3 Burger is used which comes with a LIDAR that gives the depth information. The LIDAR of the TurtleBot 3 Burger returns a vector containing 360 distance measurements. The reading of the LIDAR is counter-clockwise with 0° being to the front of the robot as illustrated in Figure 4.1. Let the range data from the LIDAR sensor be $rdata$, which is a vector with 360 distance measurements. The discontinuities can be calculated by comparing consecutive distance measurements and determine the difference between them. This can be expressed as

$$diff\_single\_scan = \left| \begin{bmatrix} rdata_0 \\ . \\ . \\ . \\ rdata_{359} \end{bmatrix} - \begin{bmatrix} rdata_1 \\ . \\ . \\ . \\ rdata_0 \end{bmatrix} \right|. \tag{4.1}$$

To determine the depth discontinuities a threshold can now be applied to $diff\_single\_scan$. If the value is greater than the threshold then the value is set to 1

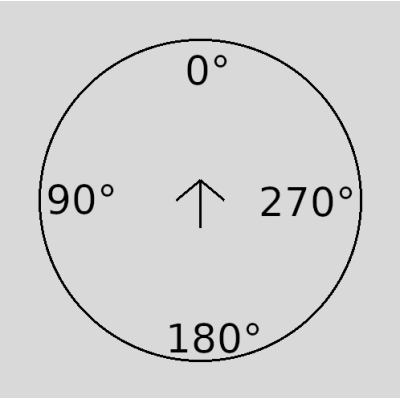


Figure 4.1.: Direction of LIDAR reading.

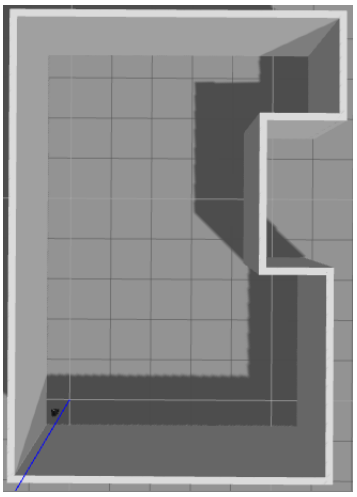
and otherwise to 0. The result is a vector with 0 and 1 with 1 indicating a depth discontinuity at this particular angle. In section 4.1 we mentioned that there is a perspective problem with the depth discontinuity detection proposed by Tovar et al.[TML07]. Figure 4.2 shows the simulated environment with the robot positioned close to the wall in a). In b) the LIDAR reading is shown and in c) the detected depth discontinuities. Due to the perspective in the distance a change of 1° in the angle causes a large change in the distance measurement. If the change is larger than the threshold a depth discontinuity is detected.

To prevent the detection of false depth discontinuities an additional method needs to be implemented that filters them out from the discontinuity reading. The previously described approach uses one LIDAR scan to detect the depth discontinuities. An alternative approach for detecting depth discontinuities is using two consecutive LIDAR scans. Let $rdata^{(t)}$ be the current reading and $rdata^{(t-1)}$ the previous reading of the LIDAR scanner. Both are vectors containing 360 distance measurements. Then calculate the differences between the two vectors

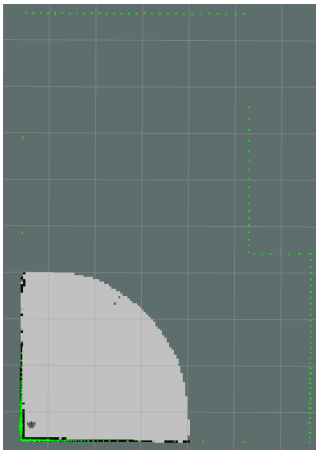
$$diff\_two\_scans = \left| rdata^{(t)} - rdata^{(t-1)} \right|. \tag{4.2}$$

Now we are using two LIDAR scans from different time stamps and check if there is a difference at the same angle from the previous LIDAR scan to the current one. After applying the threshold value to $diff\_two\_scan$ we get the depth discontinuities where the change in distance was large enough to meet the threshold and therefore indicate a discontinuity. With a high enough scan frequency, false depth discontinuities are not detected because the reading will give an equal reading. However, with the scan frequency of 5Hz from the TurtleBot 3 Burger the step resolution is not high enough to provide good results. To solve this problem the model of the TurtleBot 3 Burger was edited with a higher scan frequency. Listing A.2 shows the edited turtlebot3_burger.gazebo.xacro file with the update rate set to 90Hz.

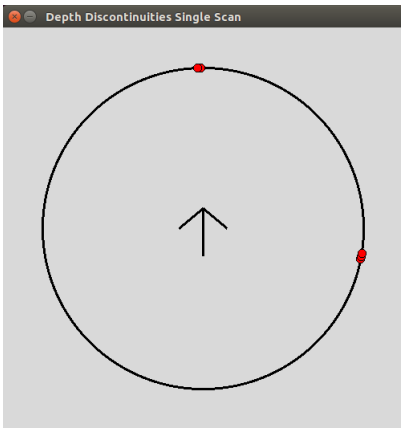
The vector $diff\_two\_scans$ can now be used to detect the changes of depth discontinuities. With the vector $diff\_single\_scan$ the depth discontinuities can be tracked when there is no detection in $diff\_two\_scans$ and to determine the exact angle of



(a) Environment in Gazebo



(b) Rviz visualising the LIDAR reading as green dots.



(c) Detected depth discontinuities by the robot.

Figure 4.2.: False detection of depth discontinuities due to the perspective when being close to the wall and the wall being long.

either the near point or the distance point.

To filter single occurring depth discontinuities, it is checked how often the depth discontinuity is detected. This not only prevents single occurring depth discontinuities, but also enables robustness on dropouts of a tracked depth discontinuity. Using a threshold, allows defining when a tracked depth discontinuity can be taken as valid. To be able to define in which direction the depth discontinuity should have moved, the odometry and drive commands are used. This allows to forecast where to search for the moved depth discontinuities from one LIDAR scan to the next LIDAR scan.

After processing the LIDAR scan and updating the detected depth discontinuities the information is passed on. The information is provided to other modules by publishing it to a topic. Other modules can subscribe to this topic and can receive the message consisting of a header, depth discontinuities, LIDAR scan, rotation and linear x movement.

4.2.2. Detection Critical Events and the Movement

For the construction of the tree the four critical events appearance, disappearance, merge and split, which are proposed by Tovar et al.[TML07], need to be detected. To be able to detect those events the depth discontinuities of t-1 are compared with the depth discontinuities of t. The information of the depth discontinuities can be retrieved from the topic where the depth discontinuities are published. With the knowledge in which direction the robot has rotated and if it moved forward or backwards allows to predict where the depth discontinuities at t-1 should move in t. During the process of updating the t-1 state to the state of t the four critical events can be detected. For further processing the information about critical events and movement are published to topics.

4.2.3. Tree Construction

For the construction of the tree the module takes the information of the critical event and movement detection. A 360° view of the environment is maintained with nodes where a depth discontinuity is detected. With each event or move published the tree gets updated. Merge events result in a new node with two childes. The same applies for a split but the other way around. The children are placed on the top-level and if there are none then new child nodes are created. An appearance results in creating a new node. If a node disappears it gets removed with all its children if there are any. Again, the updated tree is then published to a topic.

4.2.4. The Complete System

Figure 4.3 shows how the different tasks connect to each other. On top is the robot providing the sensor data (in this case LIDAR scans) and its motors that are controlled. The depth discontinuity detection uses the sensor data and monitors the drive commands provided to the motors. To detect critical events and the movement of gaps, the results from the depth discontinuity detection is used. In further consequence the information

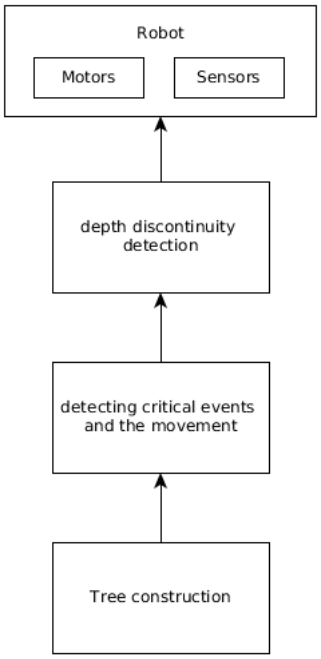


Figure 4.3.: Overview of how the different tasks connect to each other.

gained during the detection of critical events and movement of depth discontinuities are used to create and update the tree.

To allow the usage of different sensors the depth discontinuity detection and detection of critical events and movement are put in different modules. This allows reuse of the parts that are not specific to a sensor. For any sensor only the discontinuity detection part needs to be implemented.

5. Implementation Details

In the previous chapter the concept of the implementation of the Gap Navigation Tree was provided. The chapter discussed ideas which give the basis for this chapter. In this chapter the implementation is discussed looking at the different ROS Nodes. The first node covered is the *depth_jump_sensor* which is responsible for the detection of the depth discontinuities and verification that the occurred depth discontinuities were detected more than once. This is followed by the gap_sensor node which detects the critical events and move events. With the information gained by the *gap_sensor* the node *gap_navigation_tree* constructs and updates the tree.

5.1. Detection of Depth Discontinuities

The detection of the depth discontinuities is achieved by the node *depth_jump_sensor*. Its task is the detection of the depth discontinuities as well as making sure that a discontinuity not just appeared once. This part is divided into four sections. First it is discussed what data the node needs and how that data is structured. Then the detection of the depth discontinuities and filtering of false depth discontinuities will be discussed. This is followed by the tracking of the discontinuities to verify that a discontinuity not only occurred once. Finally, the data published by this node and its structure is discussed.

5.1.1. Input Data and Its Structure

In this implementation approach of the GNT a LIDAR is used to detect the depth discontinuities. The TurtleBot 3 Burger comes with a LIDAR which data is accessible by subscribing to the topic *scan*. Listing B.1 shows the definition of the LaserScan message. The field which is interesting for the calculation of the depth discontinuities is the field ranges. This field is a array of range measurements with the index of the element being the angle. The length of the array is defined by the *angle_increment* and *angel_min* as the start angle and *angel_max* as the end angle. With the LIDAR of the TurtleBot 3 Burger the angle ranges from 0° to 359° with an increment of 1°. This results in an array with 360 entries. Additional information needed is the current robot yaw and if the robot is moving forwards or backwards. This information is used to track the depth discontinuities from t to $t+1$.

The current robot yaw is determined using the odometry information which can be accessed by subscribing to the topic *odom*. From the odometry message the *PoseWithCovariance* field contains another field which is the *pose*. To determine the rotation

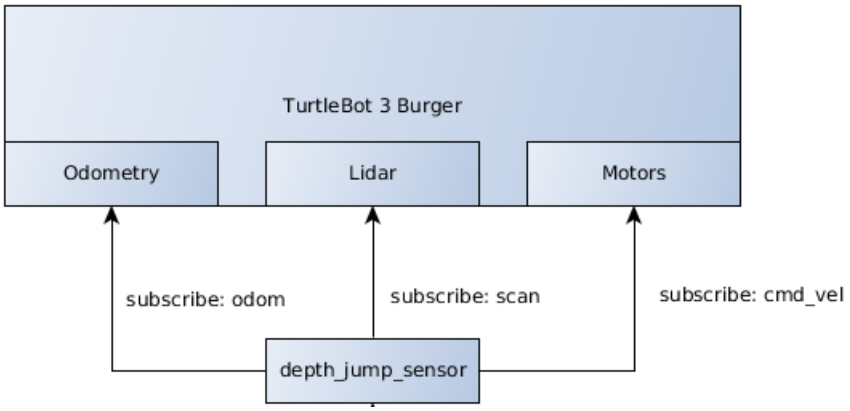


Figure 5.1.: Connections of the node *depth_jumps_sensor* to the TurtleBot 3.

direction first the yaw of the robot needs to be determined. The orientation can be determined from the quaternion information of the pose. To get the yaw from the quaternion representation a conversion needs to be done. For this purpose, ROS offers a package called *tf* [ROS20]. *tf* knows and stores the information how different coordinate systems relate to each other and offers different transformations. The transformation from quaternion(x,y,z,w) to euler(roll, pitch, yaw) is one of the transformations offered. Resulting from the conversion is a yaw that has the boundaries $-\pi < x \leq \pi$. To get an angle between 0° and 360° two more steps are required. The first step is to convert the range of yaw to $0 \leq x < 2\pi$ with following calculation $((yaw + 2 * \pi)\%(2 * \pi))$ and then converting the result to degree with $(yaw_2 * \pi * 360)/(2 * \pi)$. Listing B.2 shows the calculation of the robot yaw. For the calculation only the odometry is used, this results in an angle that might not represent the actual angle. In this case it is not a problem because the angle will only be used to determine in which direction the robot is turning.

If the robot is moving forward or backwards can be determined by subscribing to the topic *cmd_val*. The topic is used to set the motor speed and, in this case, it is used to listen to it to determine the driving direction. Listing B.3 shows the message definition for the *Twist* message and *Vector3*. The *Twist* message is the one received when subscribing to the topic *odom*. From this message we need the linear field which is of the message type *Vector3*. The definition of *Vector3* is shown below the *Twist* definition in Listing B.3. The x field of *linear* contains the forward/backwards speed. If the value is positive the robot is driving forward and if it is negative the robot drives backwards.

Figure 5.1 illustrates the connections of the node to the robot by subscribing to respective topics.

5.1.2. Depth Discontinuity Detection and Filtering

This section covers the extraction of depth discontinuities from the LIDAR data and the filtering of false depth discontinuities based on the concept introduced in section 4.2.1. Equation 4.1 and equation 4.2 introduced a way of calculating the depth discontinuities

by applying a threshold to the result of each equation. When implementing this approach infinite range measurements need to be considered. Such infinite measurements can occur when no object was detected within the range. In this case the value of the reading is *inf*. Even though this case will not appear in the test environment this is an important case to keep in mind for the calculation. First the implementation of detecting the depth discontinuities using a single scan is discussed followed by the implementation for using two scans. Then it is covered how those two approaches can be combined to filter the false depth discontinuities.

Listing B.4 shows the Python code to detect discontinuities using a single laser scan. The input parameter for the method is the LIDAR reading which is an array of distance measurements. First the variable *depth_jumps* is defined, which is the output array, with the same length as the input array and all values being 0. Then an iteration is performed over every element of the array *scan*. Each element is then compared with its successor to determine the depth discontinuity is large enough. If both elements have a value which is not inf it is checked if the depth discontinuity is larger than the threshold. When the depth discontinuity is larger than the threshold the closest point is determined. In the *depth_jump* array the value of the corresponding element is set to 1 to indicate that a depth discontinuity is at this angle. If only one of the elements has a value of *inf* the element with the none *inf* value indicates the depth discontinuity. Again, the value of the corresponding element in the *depth_jump* array is set to 1.

Figure 5.2 shows an example of the detection of the depth discontinuities. For a), b) and c) the rows represent the time, the columns represent the angle. In a) the LIDAR measurements are shown for 5 consecutive time steps and 6 angle steps. In a) the raw measurement values are shown with the measurements indicating a depth discontinuity highlighted in green. At each time step the algorithm from Listing B.4 calculates the absolute difference between measurement at α and is successor. The results of this operations are shown in b). All results being larger than the threshold which is $0,4$ are now highlighted in green. In c) the results of the algorithm are shown. Every value being less than the threshold is now 0 and every value larger than the threshold is 1. Note that the 1 is written where the closest point is to the robot based on the measurements of a). The last row in b) and c) is empty because the calculation uses the measurement of the successor which is not available at this point.

Listing B.5 shows the Python code to detect discontinuities using two laser scans which are one timestep apart. The method takes two arrays of range measurements as input. One array contains the measurements at time t and the other array contains the measurements at time $t-1$. To make use of the numpy operations both arrays are converted to numpy arrays. In the next step all *inf* values are replaced with the max reading distance and adding the threshold to that value. This allows the later subtraction work because subtracting an *inf* value would cause an error. In the next step the absolute value subtraction is performed which results in a array containing values which describe the differences between elements at the same position. To filter the depth discontinuities now all values below the threshold are set to 0. This results in non-0 vales where a depth discontinuity is detected. Because the actual difference is not important any more all remaining values non-0 are set to 1. This results in an array

a)

Threshold=0,4		Angle					
		α	α+1	α+2	α+3	α+4	α+5
Time	t-4	1.70349467	1.724552631	1.746068716	0.956208646	0.910209656	0.884375095
	t-3	1.72750342	1.715811849	1.749725938	0.93573302	0.915759087	0.871334434
	t-2	1.70432711	1.733810067	1.72469902	1.748922348	0.903548062	0.888293326
	t-1	1.70986998	1.710629702	1.733921409	1.753883004	0.935699284	0.889224708
	t	1.69102561	1.716376662	1.752417445	1.747634649	0.93319416	0.871134639

b)

Time	t-4	0.02105796	0.021516085	0.78986007	0.045998991	0.02583456	
	t-3	0.01169157	0.033914089	0.813992918	0.019973934	0.044424653	
	t-2	0.02948296	0.009111047	0.024223328	0.845374286	0.015254736	
	t-1	0.00075972	0.023291707	0.019961596	0.81818372	0.046474576	
	t	0.02535105	0.036040783	0.004782796	0.814440489	0.062059522	

c)

Time	t-4	0	0	0	1	0	
	t-3	0	0	0	1	0	
	t-2	0	0	0	0	1	
	t-1	0	0	0	0	1	
	t	0	0	0	0	1	

Figure 5.2.: Detection of depth discontinuities based on the implementation in Listing B.4. a) shows the raw measurement with the measurements indicating a discontinuity highlighted. b) shows the results of the absolut difference between the consecutive measurments and highlights those exceeding the threshold of 0,4. c) shows the resulting depth discontinuities marked with a 1.

with a 1 at the angle where a discontinuity is detected. For later calculations the values are then converted to integer values because no floating-point information is needed.

Figure 5.3 shows an example of the detection of the depth discontinuities using two scans for the calculation. The measurements used in a) are the same as in Figure 5.2 a). Again, the interesting values are highlighted in green. In a) one can see by comparing the values of the cells highlighted in green that a discontinuity is detected. The values of b) result from the absolute subtraction with the value at the same α of the LIDAR scan. Highlighted in green is the value which is above the threshold of $0,4$. Appling the threshold and setting every value below it to 0 and all values greater or equal to 1 lead to the table presented in c). In b) and c) the first row is empty because the calculation uses the previous measurement which is not available at $t-4$.

Comparing the output of the algorithm from Listing B.4 and Listing B.5 using Figure 5.2 and Figure 5.3 shows two different characteristics. The algorithm for detecting depth discontinuities using a single LIDAR scan gives a consecutive depth discontinuity position. Whereas the algorithm using two consecutive LIDAR scans only outputs a depth discontinuity when its position changed due to moving through the environment. From now on let the result of the detection using a single LIDAR scan be called *discontinuities_single_scan* and the result of the detection using two LIDAR scans *discontinuities_two_scans*. This information combined with additional checks allows to filter the false depth discontinuities. Figure 5.4 shows *discontinuities_single_scan*(a) and *discontinuities_two_scans*(b) for false depth discontinuities. The y-axis represents time with time increasing downwards and the x-axis the angle with increasing angle to the right. Column G is the reading at 3° and the seen movement is a rotation to the right which causes the discontinuities move counter clockwise. The recording of the depth discontinuities is taken in the environment with the long wall as seen in Figure

a)

Threshold=0,4		Angle					
		α	α+1	α+2	α+3	α+4	α+5
Time	t-4	1.70349467	1.724552631	1.746068716	0.956208646	0.910209656	0.884375095
	t-3	1.72750342	1.715811849	1.749725938	0.93573302	0.915759087	0.871334434
	t-2	1.70432711	1.733810067	1.72469902	1.748922348	0.903548062	0.888293326
	t-1	1.70986998	1.710629702	1.733921409	1.753883004	0.935699284	0.889224708
	t	1.69102561	1.716376662	1.752417445	1.747634649	0.93319416	0.871134639

b)

Time	t-4						
	t-3	0.02400875	0.008740783	0.003657222	0.020475626	0.005549431	0.013040662
	t-2	0.02317631	0.017998219	0.025026917	0.813189328	0.012211025	0.016958892
	t-1	0.00554287	0.023180366	0.009222388	0.004960656	0.032151222	0.000931382
	t	0.01884437	0.005746961	0.018496037	0.006248355	0.002505124	0.018090069

c)

Time	t-4						
	t-3	0	0	0	0	0	0
	t-2	0	0	0	1	0	0
	t-1	0	0	0	0	0	0
	t	0	0	0	0	0	0

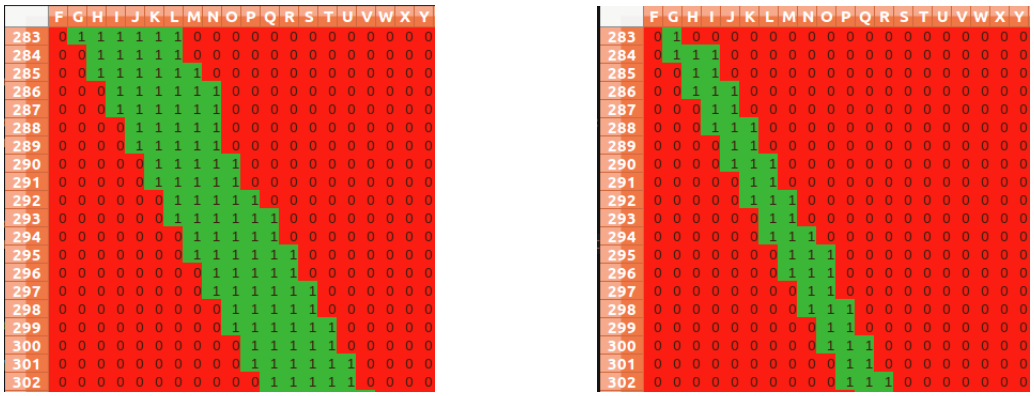
Figure 5.3.: Detection of depth discontinuities based on the implementation in Listing B.5. a) shows the raw measurement with the measurements highlighted indicating an discontinuity. b) shows the results of the absolut difference between the consecutive measurments and highlights those exceeding the threshold of 0,4. c) shows the resulting depth discontinuities marked with a 1.

5.5. The measurements in the far distance cause consecutive depth discontinuities which indicate that those depth discontinuities are false.

To solve this problem the number of consecutive depth detections can be counted. Therefore, *discontinuities_single_scan* and *discontinuities_two_scans* both are checked to have no direct neighbouring depth discontinuities within 3°.

The check is performed during the process of adding a discontinuity to the currently known ones. For the purpose of determining if a discontinuity is known or if it is new the node holds a array which matches the LIDAR array size, let this array be called *depth_jumps*. Initially *depth_jumps* has no depth discontinuities set. For each LIDAR scan first *discontinuities_single_scan* and *discontinuities_two_scans* are calculated. The *discontinuities_two_scans* array is then used to perform a check if the depth discontinuity is new or if it is a previously known depth discontinuity. If the checked value in *discontinuities_two_scans* is 1 it is checked if *discontinuities_single_scan* has a 1 at the same entry. If not, the element ± 1 from the viewed one is checked to determine the actual position of the discontinuity. For the search it is important to take the movement of the robot into account.

The movement of the robot has an impact on how the depth discontinuities move and is important for matching depth discontinuities at t with the depth discontinuities at time $t-1$. Figure 5.6 illustrates how depth discontinuities move when the robot performs different movements. The arrows in red indicate the movement of the robot. Drawn in blue is the position of the depth discontinuity before the movement. In green the movement direction of the depth discontinuity is visualised. When the robot rotates clockwise the depth discontinuities move counter-clockwise as shown in a). If the robot rotates counter-clockwise the depth discontinuities move clockwise which is shown in b). The forward and backwards movement of the robot causes a slightly different behaviour



(a) False detection single scan.

(b) False detection two scans.

Figure 5.4.: Recording of false detection for *discontinuities_single_scan* *discontinuities_two_scans*

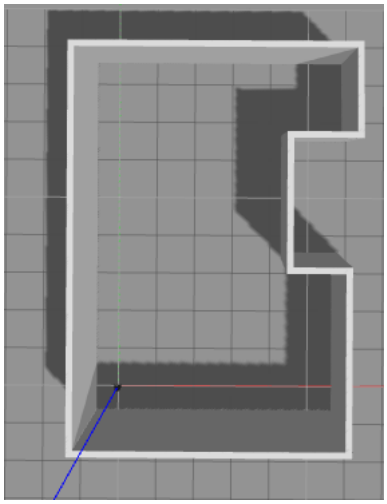
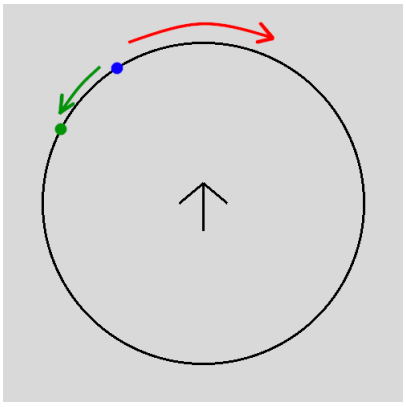
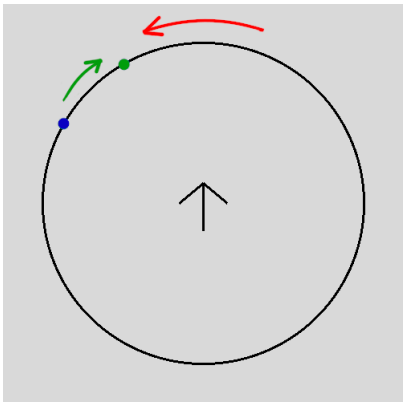


Figure 5.5.: Test environment for the perspective problem during depth discontinuity detection. The robot is positioned in the bottom left corner. A rotation to the right produces the output shown in Figure 5.4.

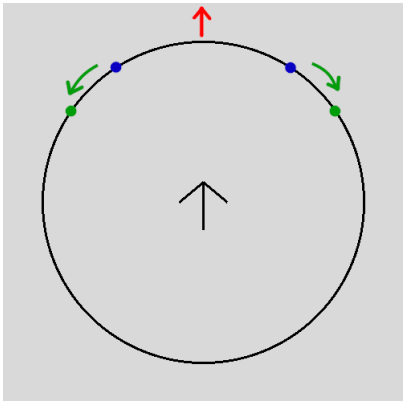
in movement for the depth discontinuity. It is different for the left side and right side of the robot. With the left side being 0° to 180° and right side being 180° to 360° and 0° in the front of the robot. If the robot is moving forward the depth discontinuities on the left move towards 179° so the angle increases. On the right side the depth discontinuities move towards 180° so the angle decreases. The forward movement is shown in c). If the robot moves backwards the movement on the left and right invert compared with the forward movement. On the left the depth discontinuities move towards 0° so the angle decreases and on the right side the depth discontinuities move towards 360° so the angle increases which is shown in d).



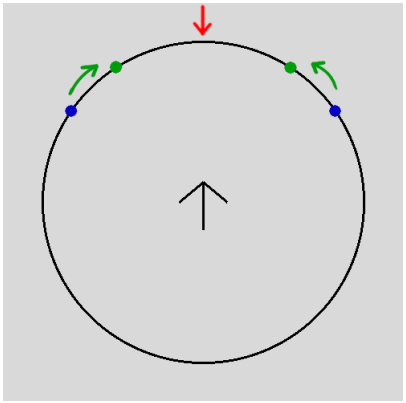
(a) Robot rotates clockwise.



(b) Robot rotates counter-clockwise.



(c) Robot moves forwards.



(d) Robot moves backwards.

Figure 5.6.: Movement of the discontinuities depending on the robot movement. The robot movement is drawn in red. Blue is the depth discontinuity at time t. Green is the depth discontinuity at time $t+1$ and the movement direction of discontinuity.

So, the searching direction for a depth discontinuity of *discontinuities_two_scans* in *discontinuities_single_scan* depends on the movement of the robot. As an example, let us assume the robot turns counter-clockwise. For this case we first want to search in the clockwise direction and then the counter-clockwise direction. With a turn counter-clockwise the *discontinuities_two_scans* all discontinuities move clockwise on the de-

tection. Therefore, we first want to check in the counter-clockwise direction and then in the clockwise direction. In the next step it needs to be determined if the detected depth discontinuity was detected at $t-1$ by checking *depth_jumps*. Here we want to search in the opposite direction than to where the depth discontinuities are expected to move. The search range used is $\pm 1°$. If no depth discontinuity was found this discontinuity needs to be added. Otherwise, a tracking is performed to verify that the discontinuity not only occurred once which is covered in section 5.1.3. Before the depth discontinuity is added to *depth_jumps* the earlier mention a check for multiple neighbouring is performed and the discontinuity is only added if there are no neighbouring depth discontinuities. Listing B.6 shows how the neighbour check is performed. This algorithm is called for *discontinuities_two_scans* and *discontinuities_single_scan*. If both return that no neighbouring depth discontinuities are found the new depth discontinuity is added.

The decision on how the *discontinuities_two_scans* array needs to be processed is done by the method shown in Listing B.7. As input the method requires *discontinuities_two_scans*, *discontinuities_single_scan*, rotation, movement and the discontinuities from the last time step. First it is checked if the robot rotated and if the start index, end index, increment for the loop and search direction are set. Then the correction is executed using this information. The second step is to check if the robot moved forward or backwards. In Figure 5.6 a) and b) it is shown how the depth discontinuities move. The correction is now performed different for the range from 0° to 179° and 180° to 359°. The settings for the correction of forward and backwards movement are shown in Listing B.8. To avoid errors due to a drift in standstill this case is also checked.

Listing B.9 shows the described correction method as pseudo code. As described earlier the *discontinuities_two_scans* are checked if a depth discontinuity is indicated. When a discontinuity is detected the check for the exact position is performed as well as the check if the discontinuity is previously known. The tracking and verification process in line 7 and Line 13 to 20 will be discussed in section 5.1.3. If the discontinuity is not known Listing B.6 is performed twice and added if no neighbours were detected.

5.1.3. Verification

This section covers the tracking of the depth discontinuities. The idea of tracking is to prevent following to cases. First, prevent the detection of a discontinuity which only appeard once. Second, prevent a depth discontinuity to be marked as disappeared due to a single detection fail. To prevent those two events the number of detections is counted. Each time a depth discontinuity from $t-1$ is found in t the counter is incremented. The incrementation is done until a threshold is reached. If a discontinuity from $t-1$ is not found in t the counter is decreased. The counter is realised using the *depth_jump* array. Line 13 to line 20 of Listing B.9 shows how discontinuities are handled when *discontinuities_two_scan* does not detect a change. If at $t-1$ a depth discontinuity was detected the counter for this discontinuity will be greater than 0. In this case *discontinuities_single_scan* is checked to determine if the discontinuity is still there. If the discontinuity is still there the counter gets increased until the threshold is reached.

Otherwise, the counter is decreased. When decreasing the counter, it eventually will get to 0 which proves that the discontinuity no longer exists. The method at line 7 of Listing B.9 handles tracking for the case of a merge, split or move of a depth discontinuity to allow the counter to be updated correct. Listing B.10 shows the algorithm behind line 7 of Listing B.9. First a check for split and merge is done. Based on the results of those checks the according operation is performed. In case of a merge the array is updated so that the counter from one of the two merged depth discontinuities is copied to the merged discontinuity and incremented. Then the counter of the two merged depth discontinuities is set back to 0. When a split is detected the counter of the splitting depth discontinuity is copied to the depth discontinuities resulting from the split and incremented. The counter of the previous depth discontinuity is set back to 0. A move of a gap is the third possible event. The counter gets copied from the position at $t-1$ to the position at t of the gap. Again, the counter gets incremented and the counter at $t-1$ set back to 0. Note, the counter is only incremented if it is below the threshold.

To determine which depth discontinuities can be published the *depth_jumps* array is processed after the update algorithm (Listing B.7) is finished. Therefore, an iteration over the whole array is performed. A depth discontinuity is valid if the counter is greater or equal to the threshold. An output array is created matching the size of the *depth_jumps* array with all elements initially set to 0, let this array be called discontinuities. For each valid depth discontinuity, the element at the same index of the discontinuities array is set to 1.

5.1.4. Published Data and Its Structure

The node publishes the detected discontinuities to the topic *depth_jumps*. Listing B.11 shows how the message published to the topic looks like. The message contains a header, the discontinuities array (*depth_jumps*), the LIDAR measurements (*range_data*), the information in which direction the robot rotated (*rotation*) and the information if the robot moved forwards or backwards (*liniear_x*). In the header the time stamp and sequence field are set. The field *depth_jumps* contains the data of the discontinuities array from the verification process. Rotation and forward, backwards movement are the movements the robot made to obtain this depth discontinuity reading.

5.2. Critical Events and Discontinuity Movement

This section covers the detection of the four critical events appear, disappear, merge and split as well as detecting the movement of the discontinuities. It will be discussed how each of these events is detected and how the information is made available for other nodes. The node responsible for this task is the *gap_sensor*.

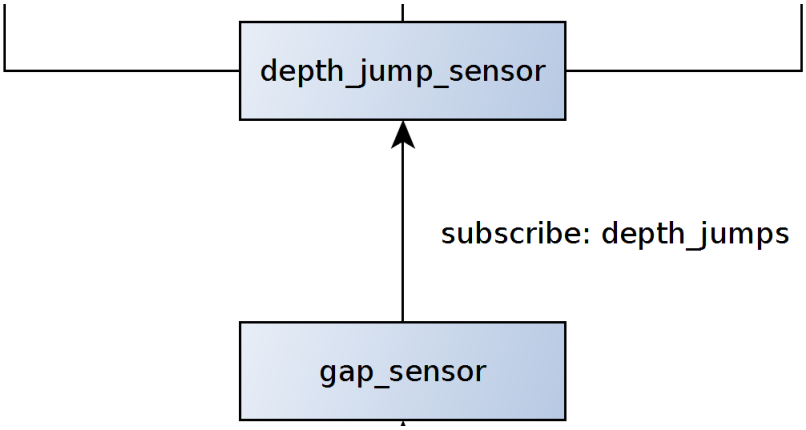


Figure 5.7.: The node *gap_sensor* subscribes to the topic *depth_jumps* of the node *depth_jumps_sensor*

5.2.1. Input Data and Its Structure

The node *gap_sensor* subscribes to the topic *depth_jumps* as shown in Figure 5.7 and is awaiting the message of Listing B.11. This shall allow the reuse of the critical event detection and movement detection for the tree construction.

5.2.2. Detection of Critical Events and Discontinuity Move

To detect the critical events and discontinuity moves the *gap_sensor* holds a global array *depth_jumps_last* which is of the same size as the field *depth_jumps* from the message *DepthJumps*. The *gap_sensor* implements a detection algorithm called *detect_critical_events* which uses the information *rotation* and *linear_x* of the message *DepthJumps* to determine what actions need to be performed. Listing B.12 shows the *detect_critical_events* method which decides what action needs to be performed. The method takes the received depth discontinuity reading, rotation and movement. Based on the rotation and movement the method updates the local view of the depth discontinuities and searches for the critical events as well as discontinuity movements. First the matching of the rotation is discussed followed by the forward and backwards movement matching. After discussing those to processes the detection of appear, disappear, merge, split and move are discussed in detail.

5.2.2.1. Match the Rotation

Listing B.13 shows the decision making on how to perform the matching. Notice that the searching direction changed compared to Listing B.7. This is because the approach for the matching is done differently. In section 5.1 the approach was to determine if the detected discontinuity existed previously. Now the approach is to check if a discontinuity detected at time $t-1$ still exists at time t. If the robot has, for example, a depth discontinuity at 10° and the turns 5° clockwise the new position will be at 15°. To find

this new position a search in positive direction (increasing angle) needs to be performed. Assuming we have a array with 360 entries this means, if the robot rotates clockwise the search is performed starting at 0° and increasing the angle until it reaches 359°. When the robot rotates counter-clockwise the search if performed starting at 359° and decreasing the angle until it reaches 0°.

Listing B.14 shows the algorithm that handles matching of the new depth discontinuity reading to the locally stored representation. First a copy of the *dept_jumps* array is created because we do not want to modify the original array. Then an iteration is performed over the *depth_jumps_last* array. The range and iteration direction are defined by the *start_index*, *end_index* and *increment*. In the given range the elements at index of *depth_jump_last* and *depth_jumps_cp* are checked for change. The first loop is designed to check for a move, merge or disappear and the second to check for a move which was not detected in the first loop or if it is a appear. An indication for a move, merge or disappear is that *depth_jumps_last* is 1 and *depth_jumps_cp* is 0. In this case first the new position of the depth discontinuity is tried to be found. The *find_new_pos_of_depth_jump* first performes a search in the direction indicated by the parameter *increment* starting from index and then in the opposite direction. Searching into the *increment* direction first because this is where the depth discontinuity is expected to move due to the rotation. After searching for the new position, the check is performed if it is a move, merge or disappear which can be distinguished by the value of *index_new* the details on how this method is implemented is discussed in section 5.2.2.4. When *depth_jumps_last* and *depth_jumps_cp* both are 1 the copy can be set to 0 so it will not be processed in the second loop. The second loop checks of any depth discontinuities in *depth_jumps_cp* to not be processed by the first loop. When *depth_jumps_cp* is 1 and *depth_jumps_last* is 0 at this point this most likely indicates a new appeared depth discontinuity. The other possibility is a failed move of the gap at the first time and therefore is tried again to match. Because one of those two cases will apply the last step is to set *depth_jumps_cp* to 0.

5.2.2.2. Match the Forward and Backwards Movement

When matching the depth discontinuities during a forward or backwards movement the range from 0° to 179° and 180° to 359° need to be searched differently, compared to as mentioned in Figure 5.6. First *match_forward_backwards* at line 16 of Listing B.12 checks in which direction the robot is moving and determines the corresponding method to call which either is *match_forward* or *match_backwards*. Listing B.15 shows how both methods iterate over the *depth_jumps_last*. The method *match_forward* iterates over the *depth_jumps_last* starting at 0° to 179° and 359° to 180°. For the *match_backwards* the direction of the iteration changes to 179° to 0° and 180° to 359°. The methods *check_positive_direction* and *check_negative_direction* perform the checks for move, merge, disappear, appear and split.

Listing B.16 shows the method *check_positive_direction*. First a check is performed if there might have been a move, merge or disappear. The indication for this is *depth_jumps_last* being 1 or higher and *depth_jumps_cp* being 0. If this applies the

new position is searched in positive direction in a range of 5 degree. It is possible that the detection was in negative direction due to measurement noise. Therefore, a search is performed in negative direction if the search in the positive direction was unsuccessful. With the received new position which might be *Null* the *check_ move_ merge_ disappear* is executed. The *check_ move_ merge_ disappear* is discussed in detail in section 5.2.2.4. A possible split or appear is indicated by *depth_ jumps_ cp* being 1 and *depth_ jumps_ last* being 0. For this case the *check_ split_ appear* is executed to determine if it is a split or appearance. It is also possible that *depth_ jump_ last* and *depth_ jump_ cp* both have the value 1. For this case now it is sufficient to set the *depth_ jumps_ cp* back to 0 so it does not get processed again. The method then returns the index which might have changed in case of a split by being set to where the second depth jump of the split was found. For the *check_ negative_ direction* four lines change compared to *check_ positive_ direction*. The first line that changes is *line 6* which changes to *search_ x_ degree_ negative* but with the same parameters. *Line 8* changes to *check_ x_ degree_ positive* but the parameters stay unchanged. On *line 9* the last parameter changes from *1 to –1*. The last line that change is on *line 14* where the *last parameter changes from +1 to –1*.

5.2.2.3. Match Drift While Still Stand

To prevent information loss the matching is also performed when no rotation and movement forwards or backwards is detected because of slow drift of the robot. Listing B.17 shows the detection of a move, merge, disappear or appearance. The iteration is performed over the *depth_ jumps_ last* and *depth_ jumps_ cp* array. It is checked for *depth_ jumps_ last* being 1 and *depth_ jumps_ cp* being 0 which indicates a move, merge or disappear. The discontinuity is expected to move only one degree. Therefore, the neighbouring elements of *depth_ jumps_ cp[index]* are checked for being 1. The result is used to perform the move, merge and disappear check. If *depth_ jumps_ last* is 0 and *depth_ jumps_ cp* is 1 a discontinuity appeared at this position. The method *discontinuity_ appear* creates a message, updates global *depth_ jumps_ last* and sets the element at index to 2 which marks it as new appeared. This is done for debugging proposes to make tracking easier. A critical event message is created and published to the topic critical event as well as added to a global event array.

5.2.2.4. Differentiation Between Move, Merge and Disappear

The decision if a depth discontinuity moved, merged or disappeared is done by the method
check_ move_ merge_ disappear in Listing B.18. The simplest case is when *index_ new* has no value. This means the previously detected depth discontinuity disappeared and the *discontinuity_ disappear* method is called. The method updates the global *depth_ jumps_ last* array by removing the discontinuity at *index_ old* and creating a disappear event message which is published to the topic *critical_ event* and added to a global event array. When *index_ new* is set the two possible events are move and merge. Figure 5.8 shows depth discontinuities at time $t-1$ and t for a move and merge. The

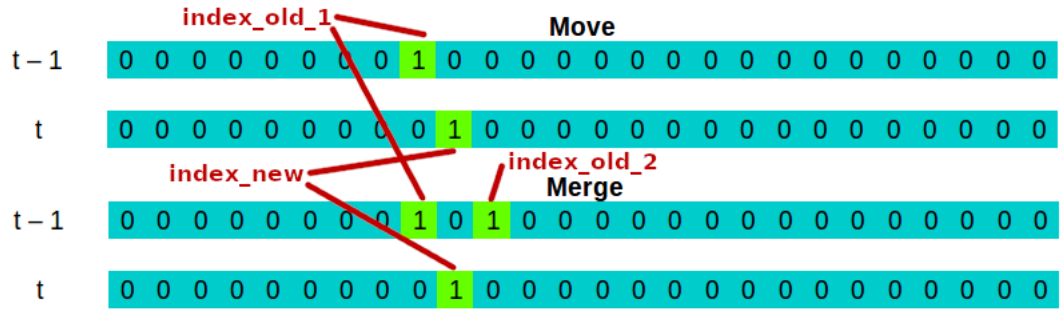


Figure 5.8.: Move and merge in comparison and the parameters from Listing B.18

method gets *index_ old_ 1* and *index_ new* as parameters. To determine if it is a move or merge it needs to be determined if there is a second depth discontinuity at $t-1$ which is two degrees apart of *index_ old_ 1*. This check is done inside *check_ merge*. The search direction for *index_ old_ 2* is defined by the parameter *search_ increment* and depends on the movement of the robot. Depending on the result of the check for a merge either *discontinuity_ moved* or *discontinuity_ merge* is called. The *discontinuity_ moved* method updates the global *depth_ jumps_ last* array by setting the element at *index_ old* to 0 and the element at *index_ new* to 1. A move message is created which is published to the topic *gap_ move* and added to a global array which holds all moves. The *discontinuity_ merge* method sets the elements of the global stored *depth_ jumps_ last* array at *index_ old_ 1* and *index_ old_ 2* to 0. To highlight the merged depth discontinuity during debugging the element at *index_ new* is set to 1. A critical event message is created which is then published to the topic *critical_ event* and added to the global event array.

5.2.2.5. Differentiation Between Split and Merge

The decision if a depth discontinuity is a split or appeared is done by the method *check_ split_ appear* in Listing B.19. First it needs to be determined if the found depth discontinuity has a position at $t-1$. Therefore, depending on the parameter *search_ increment* a search is done two degrees from the current position. Figure 5.9 shows an example for appear and split with example positions for depth discontinuities and the variables storing the positions. If a previous position was found the next step is to determine if there is a second depth discontinuity at t within two degrees. To be sure that the depth discontinuity at $t-1$ split it is checked to not have any other discontinuities within three degrees. Neighbouring depth discontinuities could indicate that it is a move rather than a split. If the both conditions are fulfilled the check returns the index of the second depth discontinuity as *index_ new_ 2* resulting from the split. Now if *index_ old* and *index_ new_ 2* is set the *discontinuity_ split* is called, and the arrays are updated. If it is not a split it must be a new depth discontinuity appearing and *discontinuity_ appear* is called. The method split sets the element at *index_ old_ 1* of the global *depth_ jumps_ last* to 0. For the tracking purpose during debugging the split depth discontinuities at *index_ new_ 1* and *index_ new_ 2* are set to 3. The critical event

Figure 5.9.: Appear and split in comparison and the parameters from Listing B.19

message is created, published to the topic *critical_ event* and added to the global list of critical events.

5.2.3. Published Data and Its Structure

This section discusses how the critical events and move events are published. The events are published in three different ways. First, the events are publised on each event detection. This means every time a appear, disappear, merge, split or move is detected the respective event is published. Second, all move events and all critcial events are publised to the respective topics after proccesing. Third, all move events and all critical events are published to one topic as a collection after processing. The message of the type *CriticalEvent* is published to the topic *critical_ event* with every detection. Listing B.20 shows the definition of the message from line 5 to line 9. Depending on the event types listed on line 12 to 16 the fields are set. For the event of an appearing discontinuity the *event_ type* field is set to 1 and the *angle_ new_ 1* is set with the angle value. On a disappear event the *event_ type* field is set to 2 and the field *angel_ old_ 1* is set with the angle where the discontinuity disappeared. For a split the field *event_ type* is set to 3, the field *angle_ old_ 1* is set with the angle of the discontinuity that split, *angle_ new_ 1* and *angle_ new_ 2* are the new angles of the discontinuities resulting from the split. The last event to cover is the merge. In case of a merge the field *event_ type* is set to 4, *angle_ old_ 1* and *angle_ old_ 2* is set with the angles of the discontinuities that merged and *angle_ new_ 1* is set with the angle of discontinuity resulting from the merge. The definition of the collection of critical events is shown on line 2 of Listing B.20. Listing B.21 shows on line 5 to line 6 how the move message is defined. The message simply contains the old angle and the new angle. Like the critical events the move event is published after every detection and as a collection which is defined on line 2 of Listing B.21. The message for sending the collection of critical events and collection of mov events together is shown in Listing B.22. The message is published after processing an update on the topic *depth_ jumps*. After publishing the collections are reset.

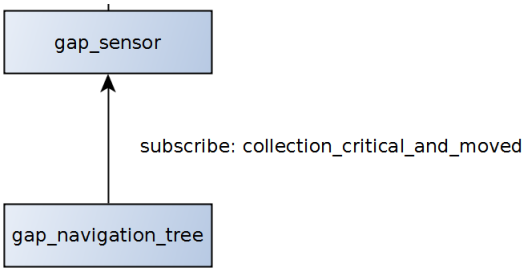


Figure 5.10.: The node *gap_ navigation_ tree* subscribes to the topic *collection_ critical_ and_ moved* of the node *gap_ sensor*

5.3. Tree Construction

The node *gap_ navigation_ tree* is responsible for the construction of the tree. With the information of the topic *collection_ critical_ and_ moved* the tree is updated according the events. First it is discussed how the messages received from *collection_ critical_ and_ moved* shown in Figure 5.10 are processed followed by how the constructed tree is made available.

5.3.1. Events Messages Processing

The construction and updating of the tree is done based on the information received from the topic *collection_ critical_ and_ moved* which data is described in section 5.2.3. While processing it is ensured that only one message is processed at a time. The constructed tree is stored globally as a 360° representation which is an array named *root*. This is done to maintain the information at which degree the discontinuity is located. In the beginning each of these elements has no node attached. The implementation of the tree node class is shown in Listing B.23. It is constructed of a field for the *id* and an array of *children*. The *id* is a consecutive number for each newly created node. First the critical event messages are processed. According to the event type the respective action is performed. If the event is an appearance a new node is created and added to the root at the angle of the appearance. On disappearance the disappearing depth discontinuity is removed from root including all its children. When two depth discontinuities merge first a new node is created. Then the two nodes which merge are added to the newly created node as children. The merging nodes are then removed from the *root* and the newly created node is added to the *root* at the corresponding angle. A split of a depth discontinuity has two different actions in the tree depending on if the node representing the split depth discontinuity having children or not. If the node has children those two are added according to the angles and the node representing the split depth discontinuity is removed. When the node has no children first two new nodes are created and then added to the root. After having processed all critical events the move events are processed. The move event is a removing of the node at the given *angle_ old* and moving it to *angle_ new*. Figure 5.11 visualises how the node sees the constructed tree.

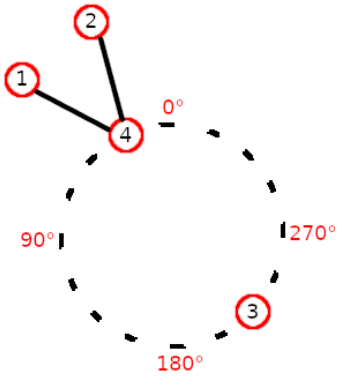


Figure 5.11.: Visualisation how the tree is stored in the node *gap_navigation_tree*

5.3.2. Published Data and Its Structure

The constructed tree is published every time the message from the *collection_critical_and_moved* topic contains one or more critical events. Before it can be published the tree needs to be converted to the message type the topic *gap_tree* requires. Listing B.24 shows how the tree node is defined in the ROS message. A definition of the tree node like the Python implementation in Listing B.23 which would look like Listing B.25 is not possible. The reason for this is that it is not possible to define a default value for custom messages. When trying to generate the message in Listing B.25 an endless loop starts because the ROS build tool searches for data type which has defined default values. The definition in isting B.24 is the alternative way for contracting the tree. It contains all nodes as a list and each node has an id and a list of child ids. Based on this information the receiver needs to construct the tree.

5.4. The Complete System

Figure 5.12 shows all nodes and the topics they subscribe. At top the TurtleBot 3 Burger is visualised with its sensor and the *depth_jump_sensor* node which subscribes to the odometry, LIDAR and motors. This node is specific for to the application with the TurtleBot 3 Burger which uses a LIDAR sensor. The node *gap_sensor* and *gap_navigation_tree* are designed to be reused with any other system which can extract depth discontinuities.

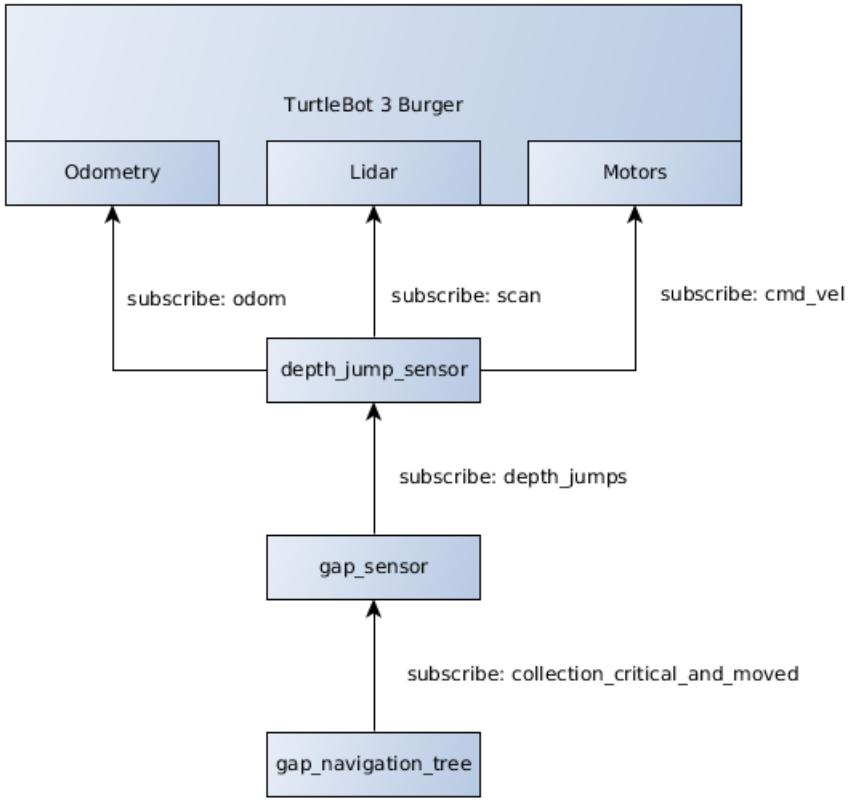


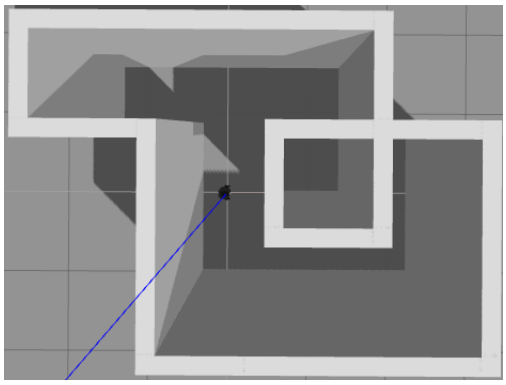
Figure 5.12.: All nodes and the communication between them.

6. Results

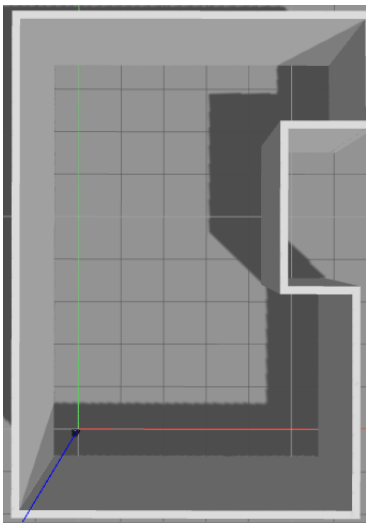
This chapter discusses the results gained during testing the implemented framework. For the tests three different environments were chosen which are shown in Figure 6.1. The environment shown in a) is a small environment with the outer boundaries being 3,5m x 3,5m. Due to its shape it is perfect to test the discontinuity detection and critical events. By travelling to the bottom right a merge should occur and when traveling back this merged discontinuity should split. The environment in shown b) allows to test the perspective problem. Discontinuities will appear in the distance on the wall if the robot depending on his orientation towards it. Furthermore, the environment allows to check the effect of long distance on the detection of discontinuities. The last environment shown in c) is a more complex one. Multiple discontinuities will appear at the same time and this will show how the framework can handle multiple discontinuities.

First let us discuss the results of the test in environment b) of Figure 6.1. To test if the concept discussed in section 4.2.1 the robot was put close to the wall so in the distance will appear discontinuities where no discontinuities are. First it was tested if the filtering of false discontinuities works during the rotation of the robot. Therefore, the robot was turned clockwise and counter-clockwise several times. The tests for this setup gave mixed results. There have been tests where the *depth_jump_sensor* node filtered the false discontinuities successful but in general not a 100% correct detection. Figure 6.2 shows the result of one of the tests where in a) the *depth_jump_sensor* node accepted a false depth discontinuity which is the left one. The robot was in the bottom left corner and looking upwards as shown in Figure 6.1 b). In Figure 6.2 b) and c) it can be seen that there is a miss match to a). The discontinuity reading in b) shows to detections on the right. Those are from the top right corner. In a) they are not shown as this representation constantly updates and at the time of taking the image the second depth discontinuity was not detected. The nonconstant detection in a) resulted in the wrong representation in c) where the gap_nativigation_tree node shows the current representation of the environment. During the rotation the detection changes caused several new depth discontinuities. They were not recognised and not detected as disappearing which resulted in several left over discontinuities as shown in d). The measurement noise, which is simulated by gazebo, caused false representations. Figure 6.3 shows some images of another rotation test. This time the *depth_jump_sensor* node managed to detect the false discontinuities and filtered them as shown in a). In b), c) and d) can be seen that the discontinuity of the corner in the far-top right corner again caused some false results and therefore the representation gained by the *gap_navigation_tree* node is not correct.

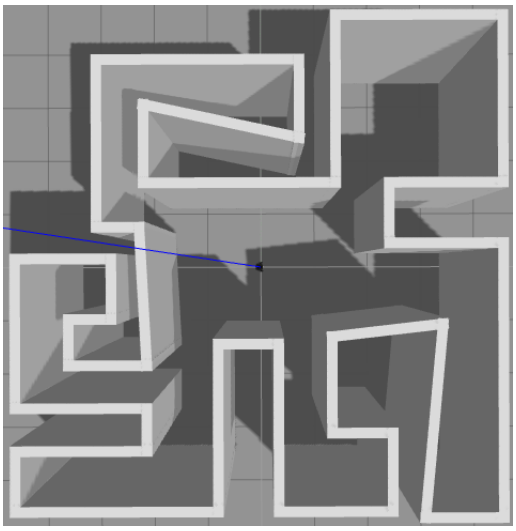
Figure 6.4 shows some images of the perspective test in environment b) of Figure 6.1. The *depth_jump_sensor* node can detect the false depth discontinuities due to the



(a) Simple test environment.

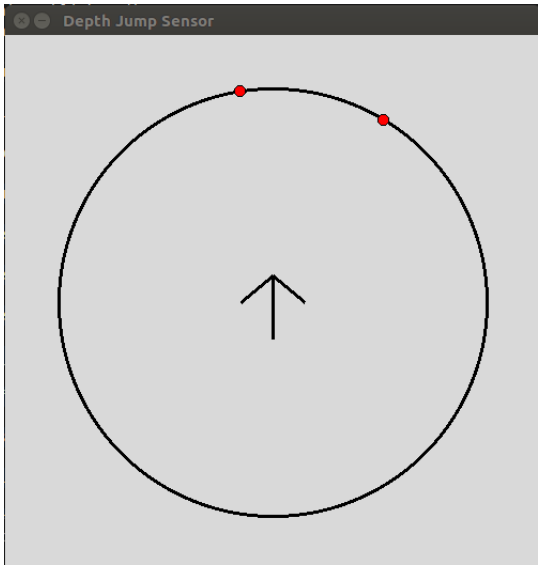


(b) Environment for perspective test.

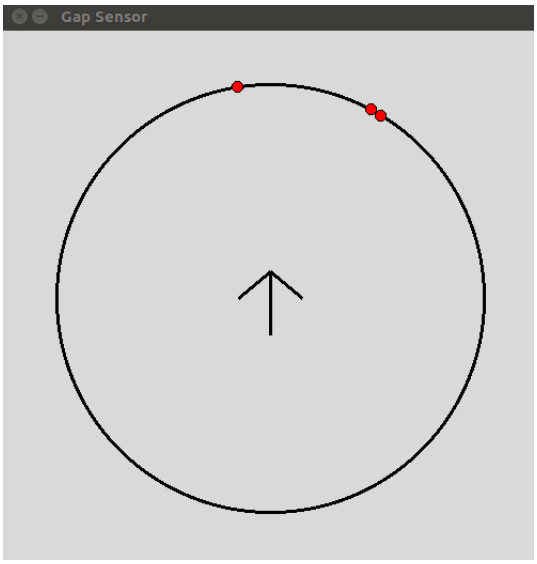


(c) Complex test environment.

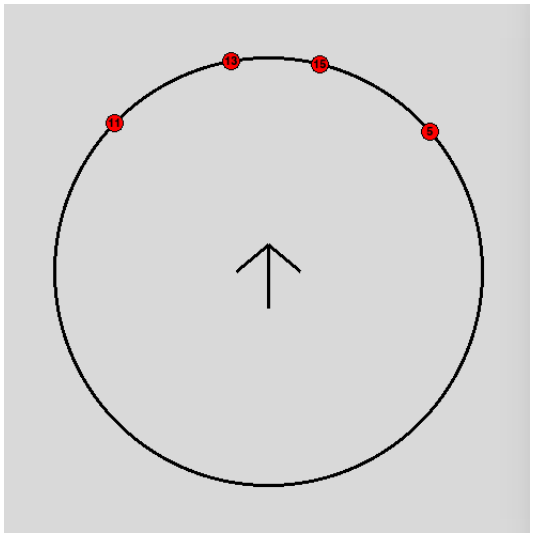
Figure 6.1.: Environments used to test the framework.



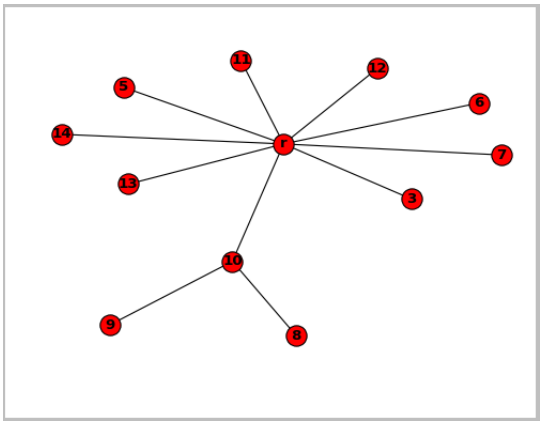
(a) Readings *depth_jump_sensor* node.



(b) Readings *gap_sensor* node.

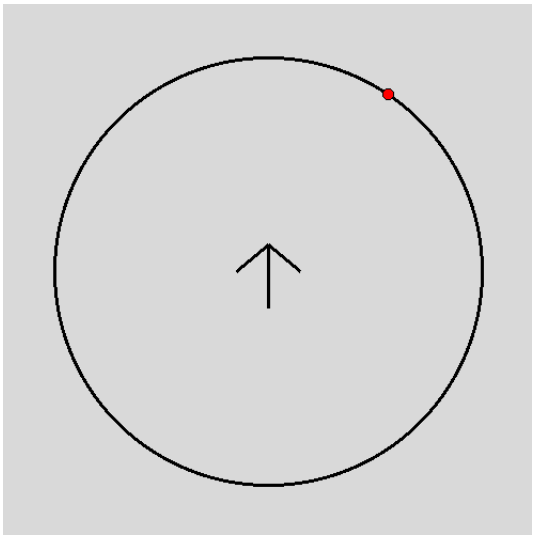


(c) Readings *gap_navigation_tree*, discontinuity positions.

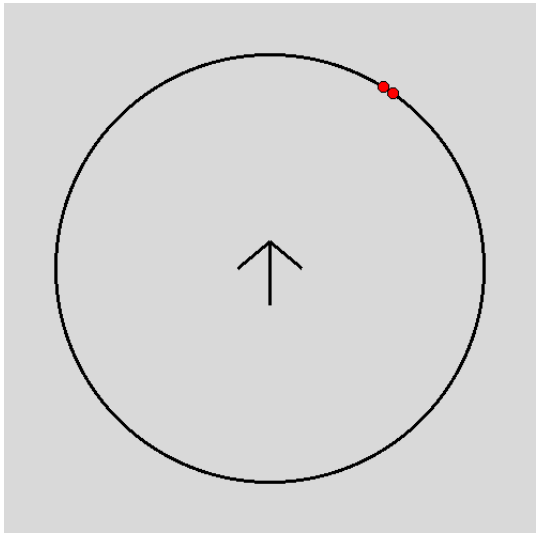


(d) Readings *gap_navigation_tree*, tree view.

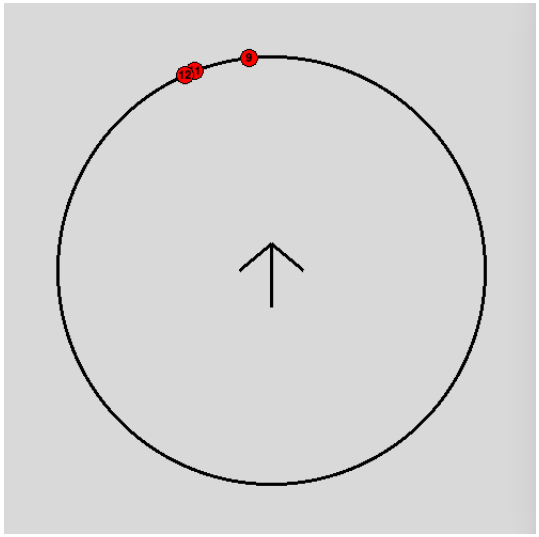
Figure 6.2.: Rotation test 1 in environment 6.1.b .



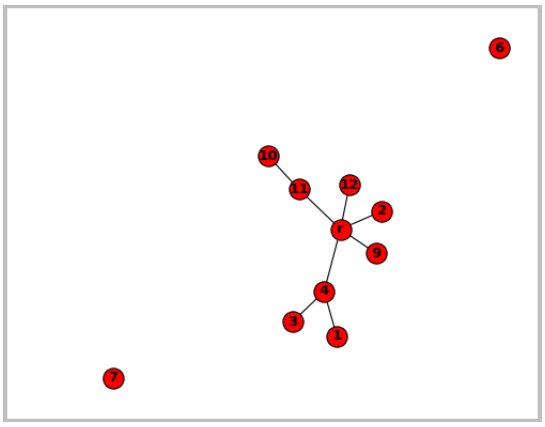
(a) Readings *depth_ jump_ sensor* node.



(b) Readings *gap_ sensor* node.



(c) Readings *gap_ navigation_ tree*, discontinuity positions.



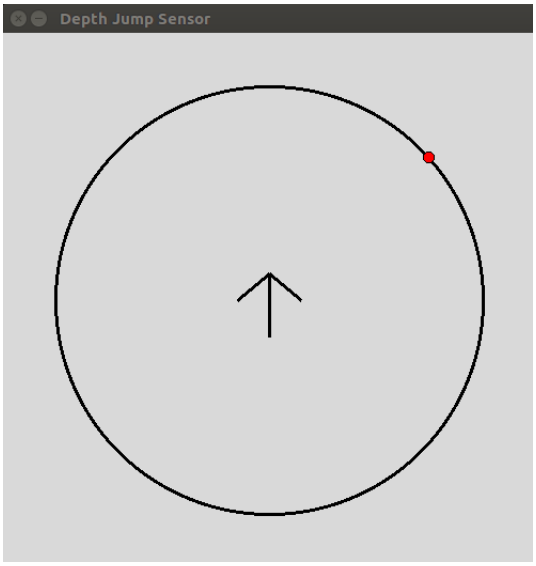
(d) Readings *gap_ navigation_ tree*, tree view.

Figure 6.3.: Rotation test 2 in environment 6.1.b .

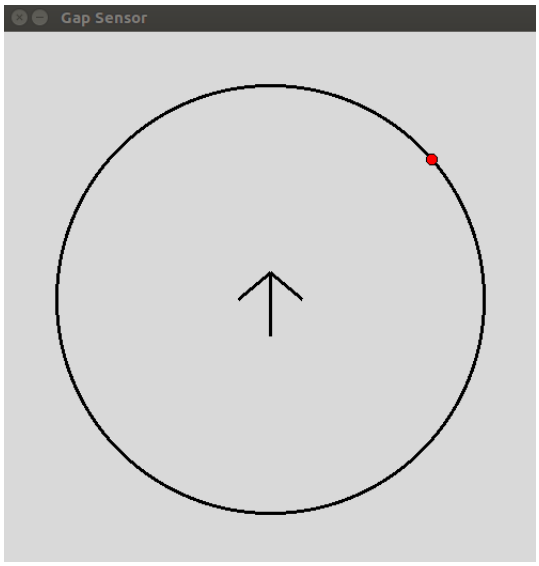
perspective and filters it. Compared to Figure 6.2 and 6.3 there are not two detections in the top right corner in b) and c). But d) reveals that there must have been some wrong detections in the top right because nodes are disconnected from the robot. Floating nodes in the graph indicate that the detection of the depth discontinuities most likely had some false detections and therefore caused a false information during the detection of the critical events or move events. The actual reason for this needs to be determined with more tests.

To test the detection of the critical events and move events the environment a) in Figure 6.1 was used. The robot was navigated through the environment manually on the path shown in Figure 6.5 a). The start is in the top right corner were the robot first is moved straight ahead to the junction. At the junction a 90° counter-clockwise turn was performed so the robot will face downwards when looking from the top. The robot is then moved forwards until the left side is free for the robot. It is then performed again a90° counter-clockwise turn and moved forward. When the wall to the left disappears again a turn 90° counter-clockwise is done. The robot is then moved forward to approximately the centre. From this point the robot is then moved back to the starting point on the same route. The expected number of the discontinuities on the path can be determined for this small environment manually. In b) the sectors and number of discontinuities expected to be detect is added in blue to the previously discussed path. The letters in red are names for the sections for easier referencing. Critical events are expected to occur between G and H. When going from G to H two depth discontinuities will merge into one which is the critical event merge. When traveling from H to G the merged discontinuities split again which is the critical event split. In Figure c) and d) the splitting and merging depth discontinuities between G and H are drawn in red.

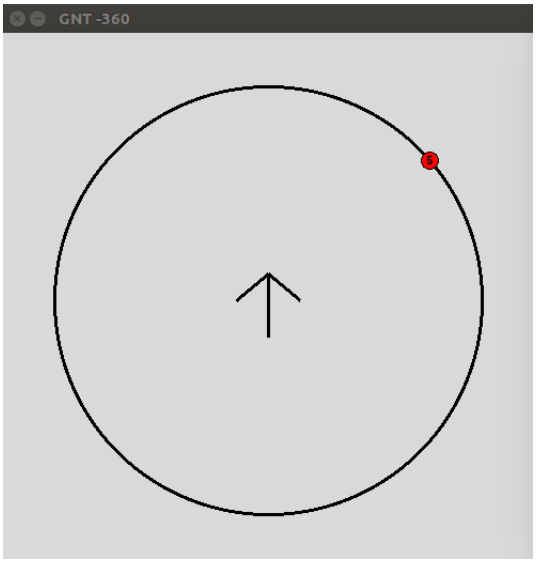
Figure 6.6 shows results from the test in the environment Figure 6.1 a). The first column (a, c, e) shows the constructed tree and the second column (b, d, f) the depth discontinuity positions. As the driving speed the maximal possible speed of the TurtleBot 3 Burger is chosen. The maximum translational velocity is 0,22m/s and the maximum rotational velocity is 2.84 rad/s. Several tests were done, and detection issues were discovered during this test. When moving between sector F, G and H the results were not stable. In a) and b) readings for sector F is given. Note that the nodes are numbered with 1, 2 and 3. When the robot moves to sector H the tree nodes 2 and 3 are expected to merge and create a new node 4. In c) and d) the reading after driving into sector H coming from sector F through G is shown. Looking at the nodes of the tree in c) one can see that the result is different to the expected results. The merged node has as children node 3 and 5 and the previous node 1 changed to node 4. This behaviour was observed during the test multiple times. The cause for this seems to be the transition from G to H where depth discontinuities are detected as disappearing and then appear again. Because of the definition that appearing depth discontinuities get a new id the number of the node is new even if it is the same than before. To prevent this behaviour caused by measurement noise the detection of depth discontinuities needs to be improved. Driving back from sector H to F the previously merged depth discontinuities are expected to split and the children of the splitting nodes. The result in e) and f) are not as expected. Due to disappearing and appearing the representation of the environment is wrong. Node 3



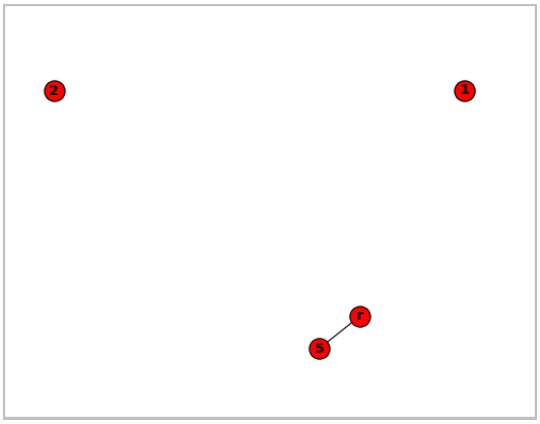
(a) Readings *depth_jump_sensor* node



(b) Readings *gap_sensor* node.



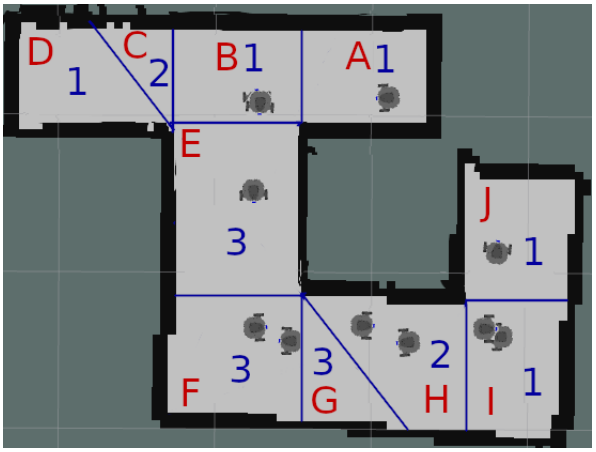
(c) Readings *gap_navigation_tree*, discontinuity positions.



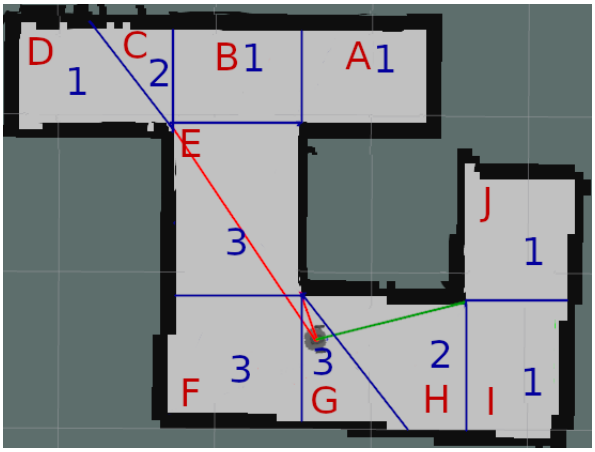
(d) Readings *gap_navigation_tree*, tree view.

Figure 6.4.: Forward driving in environment 6.1.b .

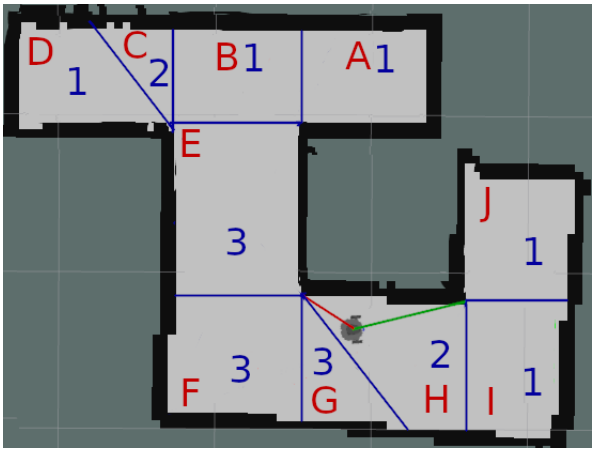
(a) Driving path.



(b) Sections with number of depth discontinuities detected.



(c) Discontinuity reading in section G.



(d) Discontinuity reading in section H.

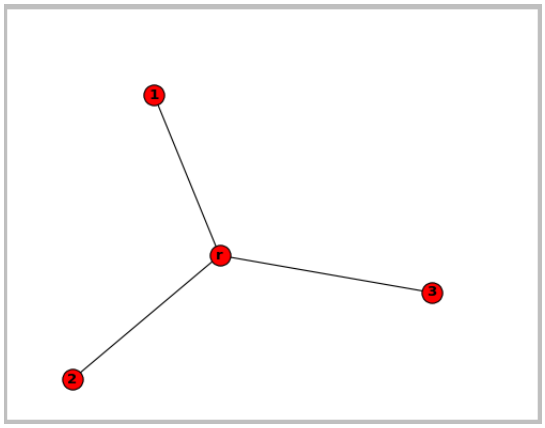
Figure 6.5.: Driving in environment 6.1.a.

and 5 which were expected to appear from the split of node 6 are shown as merged into node 8. Furthermore, the node 4 was detected as disappearing and then reappeared. To solve this issue some more tests need to be done in the future to analyse the exact cause and how this can be prevented and adapt the algorithms accordingly.

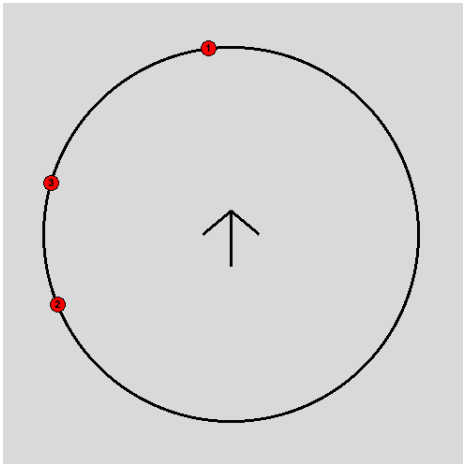
The third test was performed in the environment shown in Figure 6.1 c). Compared to the other environments it is a very complex environment. When the robot moves around the space in the centre multiple depth discontinuities can be detected. Therefore, this environment is convenient to test how the framework can handle multiple depth discontinuities at a time. When running the simulation at first everything works as expected. As soon as the robot turns the discontinuities appear on the visualisation of the *depth_jump_sensor* node and *gap_sensor* node without any delay. By the time the first depth discontinuities are detected a delay of 6 seconds can be detected until the movements show up on the visualisation. Listing 6.1 shows the average processing time of one LIDAR scan for the *depth_jump_sensor* and *gap_sensor* in environment a) and c) of Figure 6.1. When comparing the average processing time, a small difference can be determined but this should not lead to a delay of 6 seconds. It turns out that the visualisation cannot handle the fast updates. While navigating the robot through the environment the same behaviour was observed as in the small environment. By the numbers of the nodes in the tree representation shown in Figure 6.7 can be determined that several appear and disappear events got triggered. There are no consecutive numbers and two nodes which are not connected. Therefore, no valid result could be achieved.

```
1 ### Complex Environment ###
2 gap_sensor: 0.721991212554 ms
3 depth_jump_sensor: 3.94728794769 ms
4
5 ## Simple Environment ###
6 gap_sensor: 0.568701975369 ms
7 depth_jump_sensor: 3.41519563684 ms
```

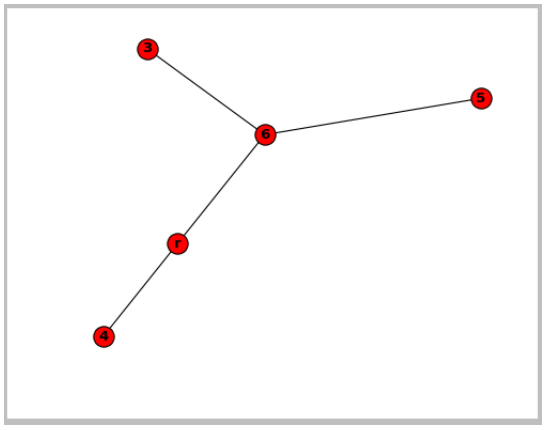
Listing 6.1: Average processing time of one LIDAR scan and discontinuity reading for the simple and complex environment.



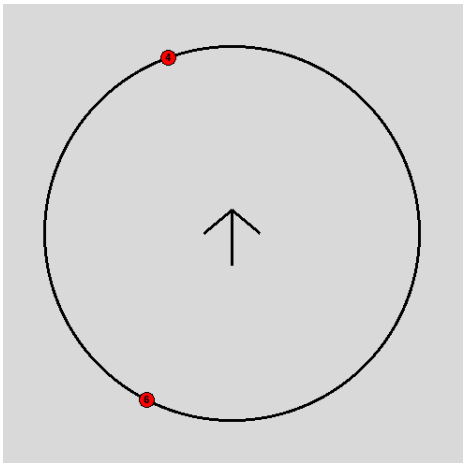
(a) Section F: Tree representation.



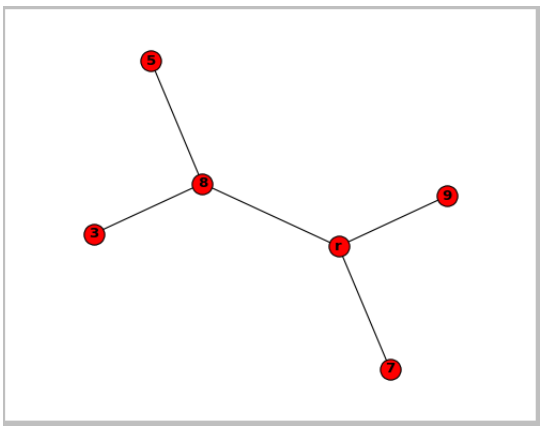
(b) Section F: Discontinuitiy reading.



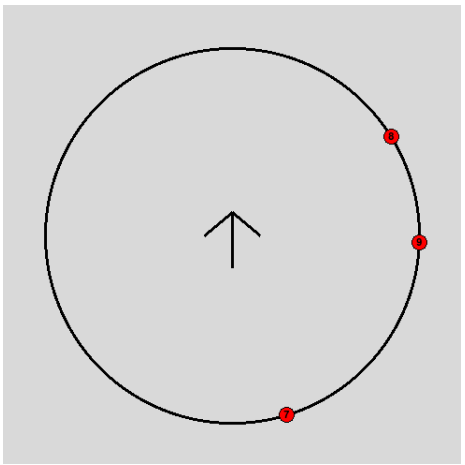
(c) Section H coming from F: Tree representation.



(d) Section H coming from F: Discontinuitiy reading.



(e) Section F coming from H: Tree representation.



(f) Section F coming from H: Discontinuitiy reading.

Figure 6.6.: Results from driving from F to H and back to F in the environment shown in Figure 6.5

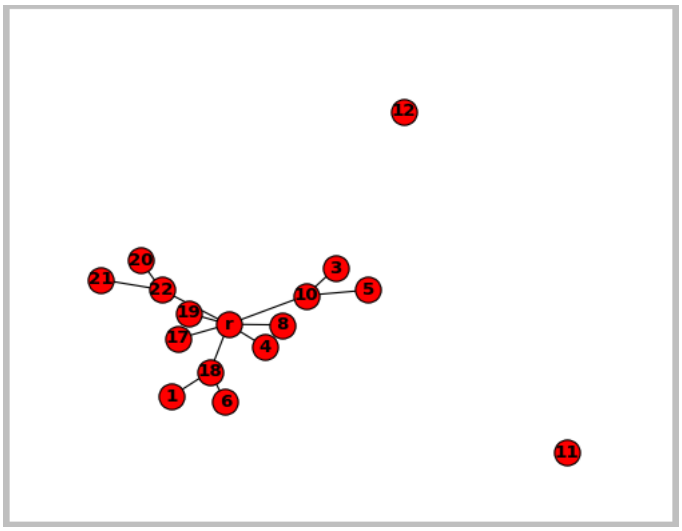


Figure 6.7.: Resulting graph after moving around in the complex environment from Figure 6.1.c.

7. Discussion and Conclusion

This chapter summarises the outcome of this work. The result is critically reflected and limitations are discussed. At the end, some insights and final thoughts are provided.

7.1. Critical Reflection

This work presents an implementation approach of generating a representation of the environment using minimal sensing information based on the theory proposed by Tovar et al.[TML07]. The framework is designed to allow the use of different sensors to extract the depth discontinuities. With the current implementation the framework is a prototype approach, because the framework does not produce stable results. Parameters of the different nodes currently are only changeable by manipulating the code and not with passing parameters to the nodes during start-up.

The development was only done using the simulator Gazebo and it can be expected that the framework will behave different in a real environment. This is because the simulator cannot recreate the real worlds noise of for example the LIDAR measurement. Furthermore, the used environments were very abstract and do not represent a real environment which might return different results.

Choosing the TurtleBot 3 Burger might have been a wrong decision. The robot fulfils the needs of having minimal sensing by only having a LIDAR and odometry but it has a sensor which only has a range of 3,5m and a resolution of 1°. This limits the robot to environments of 3,5m x 3,5m to fulfil the requirement of tracking a discontinuity. Furthermore, the update rate of the LIDAR sensor as well as the odometry is an important part to allow accurate tracking.

Transferring the ideal assumption of Tovar et al.[TML07] to an actual sensor was underestimated and took more time than expected. A lot of time was put into achieving a depth discontinuity detection which can handle the detection of false discontinuities and accurate tracking. This time was missing at the end to take a closer look at why the construction of the tree is not reliable.

7.2. Limitations

At the current state of the framework parameters can only be changed from within the code. Furthermore, it is not ready for real world use and some more research needs to be put into it to make it ready for it. The robot currently needs to be controlled manually. Using ROS the tool for this purpose is the *teleop_twist_keyboard* which allows to set

the rotation, forwards and backwards speed and giving the drive commands using the keyboard.

7.3. Future Work

To make the framework more correct the robustness needs to improve. Therefore, the detection of depth discontinuities and the detection of critical events and moves of discontinuities need more research and testing in different scenarios and different robot speeds. An important part for autonomous driving is the check if a robot fits through a gap that is indicated by a depth discontinuity. In the real environment a discontinuity might be caused by some obstacle which has a gap which is large enough that it causes depth discontinuities that are recognised but too small for the robot to fit through it. Another important step for autonomous driving is the chasing of depth discontinuities. Therefore, an algorithm needs to be developed which uses wall following and obstacle to detection to chase a depth discontinuity. This needs to be done because the robot would hit the wall when it just approaches the depth discontinuity in a straight line as described in Tovar et al.[TML07] where the robot was modelled as a point. Testing the framework in a real environment with a real robot is also a task that needs to be done.

Bibliography

[Adn15] Adnan Ademovic. *An Introduction to Robot Operating System: The Ultimate Robot Application Framework*. Toptal Engineering Blog. 2015. URL: https://www.toptal.com/robotics/introduction-to-robot-operating-system (visited on 10/22/2020).

[Afz+20] Afsoon Afzal et al. "A Study on the Challenges of Using Robotics Simulators for Testing". In: *arXiv:2004.07368 [cs]* (Apr. 15, 2020). arXiv: 2004.07368. URL: http://arxiv.org/abs/2004.07368 (visited on 10/22/2020).

[ASN09] Farshad Arvin, Khairulmizam Samsudin, and M. Ali Nasseri. "Design of a differential-drive wheeled robot controller with pulse-width modulation". In: *2009 Innovative Technologies in Intelligent Systems and Industrial Applications*. 2009 Conference on Innovative Technologies in Intelligent Systems and Industrial Applications (CITISIA). Kuala Lumpur, Malaysia: IEEE, July 2009, pp. 143–147. ISBN: 978-1-4244-2886-1. DOI: 10.1109/CITISIA.2009.5224223. URL: http://ieeexplore.ieee.org/document/5224223/ (visited on 10/22/2020).

[Fou14] Open Source Robotics Foundation. *Gazebo models*. 2014. URL: http://models.gazebosim.org/ (visited on 10/22/2020).

[Fou18] Open Source Robotics Foundation. *kinetic - ROS Wiki*. Aug. 1, 2018. URL: http://wiki.ros.org/kinetic (visited on 10/22/2020).

[Fou20] Open Source Robotics Foundation. *SDFormat Home*. 2020. URL: http://sdformat.org/ (visited on 10/22/2020).

[gee20] geeksforgeeks. *Introduction to ROS (Robot Operating System)*. GeeksforGeeks. Section: Advanced Computer Subject. Jan. 3, 2020. URL: https://www.geeksforgeeks.org/introduction-to-ros-robot-operating-system/ (visited on 10/22/2020).

[gen20a] generationrobots. *LDS-01 360 Laser Distance Sensor*. Génération Robots. 2020. URL: https://www.generationrobots.com/en/402984-lds-01-360-laser-distance-sensor.html (visited on 10/22/2020).

[gen20b] generationrobots. *TurtleBot3 Burger educational robot*. Génération Robots. 2020. URL: https://www.generationrobots.com/en/402707-turtlebot3-burger-mobile-robot.html (visited on 10/22/2020).

[KC20] Motaz Khader and Samir Cherian. "An Introduction to Automotive LIDAR". In: (2020), p. 7.

[KN11] Podishetty Naveen Kumar and Y. Shivraj Narayan. "Simulation in Robotics". In: Jan. 11, 2011.

[LSR06] Tim Laue, Kai Spiess, and Thomas Röfer. "SimRobot – A General Physical Robot Simulator and Its Application in RoboCup". In: *RoboCup 2005: Robot Soccer World Cup IX*. Ed. by Ansgar Bredenfeld et al. Red. by David Hutchison et al. Vol. 4020. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 173–183. ISBN: 978-3-540-35437-6 978-3-540-35438-3. DOI: 10.1007/11780519_16. URL: http://link.springer.com/10.1007/11780519_16 (visited on 10/22/2020).

[Ope14a] Open Source Robotics Foundation. *Gazebo*. 2014. URL: http://gazebosim.org/ (visited on 10/22/2020).

[Ope14b] Open Source Robotics Foundation. *Gazebo : Tutorial : Model Editor*. 2014. URL: http://gazebosim.org/tutorials?tut=model_editor (visited on 10/22/2020).

[Ope20a] Open Robotics. *ROS.org | Core Components*. 2020. URL: https://www.ros.org/core-components/ (visited on 10/22/2020).

[Ope20b] Open Robotics. *rqt - ROS Wiki*. 2020. URL: http://wiki.ros.org/rqt (visited on 10/22/2020).

[Ope20c] Open Robotics. *rviz - ROS Wiki*. 2020. URL: http://wiki.ros.org/rviz (visited on 10/22/2020).

[Qui+09] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: (2009), p. 6.

[ROB20a] ROBOTIS. *TurtleBot3 Simulation*. ROBOTIS e-Manual. 2020. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/ (visited on 10/22/2020).

[ROB20b] ROBOTIS. *urdf - ROS Wiki*. 2020. URL: https://wiki.ros.org/urdf (visited on 10/22/2020).

[ROB20c] Your ROBOTIS. *Turtlebot3 Burger overview*. ROBOTIS e-Manual. 2020. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/ (visited on 10/22/2020).

[Rob20] Open Robotics. *ROS.org | Integration*. 2020. URL: https://www.ros.org/integration/ (visited on 10/22/2020).

[ROS20] ROS. *tf - ROS Wiki*. 2020. URL: http://wiki.ros.org/tf (visited on 11/16/2020).

[SNS11] Roland Siegwart, Illah R. Nourbakhsh, and Davide Scaramuzza. *Introduction to Autonomous Mobile Robots (Second Edition)*. 2011. ISBN: 978-0-262-01535-6.

[TB96] Sebastian Thurn and Arno Bücken. "Learning Maps for Indoor Mobile Robot Navigation". In: *CMU-CS-96-121* (Apr. 1996), p. 38.

[Thr02] Sebastian Thrun. "Robotic Mapping: A Survey". In: (2002), p. 31.

[TML07] B. Tovar, R. Murrieta-Cid, and S.M. LaValle. "Distance-Optimal Navigation in an Unknown Environment Without Sensing Distances". In: *IEEE Transactions on Robotics* 23.3 (June 2007), pp. 506–518. ISSN: 1552-3098. DOI: 10.1109/TRO.2007.898962. URL: http://ieeexplore.ieee.org/document/4252181/ (visited on 04/26/2020).

[YJC12] Chuho Yi, Seungdo Jeong, and Jungwon Cho. "Map Representation for Robots". In: 2.1 (2012), p. 11.

[Zha+10] Ying Zhang et al. "Real-time indoor mapping for mobile robots with limited sensing". In: *The 7th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (IEEE MASS 2010)*. 2010 IEEE 7th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS). San Francisco, CA, USA: IEEE, Nov. 2010, pp. 636–641. ISBN: 978-1-4244-7488-2. DOI: 10.1109/MASS.2010.5663778. URL: http://ieeexplore.ieee.org/document/5663778/ (visited on 11/11/2020).

Statuatory Declaration

I declare that I have developed and written the enclosed work completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others or literal quotations are clearly marked. This Master Thesis was not used in the same or in a similar version to achieve an academic degree nor has it been published elsewhere.

Dornbirn, 20 December 2020 Daniel Gross

Appendices

A. Listings of chapter 4

```
<gazebo reference="base_scan">
  <material>Gazebo/FlatBlack</material>
  <sensor type="ray" name="lds_lfcd_sensor">
  <pose>0 0 0 0 0 0</pose>
  <visualize>$(arg laser_visual)</visualize>
  <update_rate>5</update_rate>
  <ray>
    <scan>
      <horizontal>
        <samples>360</samples>
        <resolution>1</resolution>
        <min_angle>0.0</min_angle>
        <max_angle>6.28319</max_angle>
      </horizontal>
    </scan>
    <range>
      <min>0.120</min>
      <max>10</max> <!--default: 3.5-->
      <resolution>0.015</resolution>
    </range>
      <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.01</stddev>
      </noise>
  </ray>
  <plugin name="gazebo_ros_lds_lfcd_controller" filename="
      libgazebo_ros_laser.so">
    <topicName>scan</topicName>
    <frameName>base_scan</frameName>
  </plugin>
  </sensor>
</gazebo>
```

Listing A.1: The changed turtlebot3_burger.gazebo.xacro file to extend the laser range.

```
<gazebo reference="base_scan">
  <material>Gazebo/FlatBlack</material>
  <sensor type="ray" name="lds_lfcd_sensor">
  <pose>0 0 0 0 0 0</pose>
  <visualize>$(arg laser_visual)</visualize>
  <update_rate>90</update_rate> <!--default: 5-->
  <ray>
    <scan>
      <horizontal>
        <samples>360</samples>
        <resolution>1</resolution>
        <min_angle>0.0</min_angle>
        <max_angle>6.28319</max_angle>
      </horizontal>
    </scan>
    <range>
      <min>0.120</min>
      <max>10</max> <!--default: 3.5-->
      <resolution>0.015</resolution>
    </range>
      <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.01</stddev>
      </noise>
  </ray>
  <plugin name="gazebo_ros_lds_lfcd_controller" filename="
      libgazebo_ros_laser.so">
    <topicName>scan</topicName>
    <frameName>base_scan</frameName>
  </plugin>
  </sensor>
</gazebo>
```

Listing A.2: The changed turtlebot3_burger.gazebo.xacro file to increase the update rate.

B. Listings of chapter 5

```
1 # Single scan from a planar laser range-finder
2 #
3 # If you have another ranging device with different behavior (e.g. a sonar
4 # array), please find or create a different message, since applications
5 # will make fairly laser-specific assumptions about this data
6
7 Header header           # timestamp in the header is the acquisition time
      of
8                        # the first ray in the scan.
9                        #
10                       # in frame frame_id, angles are measured around
11                       # the positive Z axis (counterclockwise, if Z is
                             up)
12                       # with zero angle being forward along the x axis
13
14 float32 angle_min      # start angle of the scan [rad]
15 float32 angle_max      # end angle of the scan [rad]
16 float32 angle_increment  # angular distance between measurements [rad]
17
18 float32 time_increment   # time between measurements [seconds] - if your
      scanner
19                       # is moving, this will be used in interpolating
                             position
20                       # of 3d points
21 float32 scan_time      # time between scans [seconds]
22
23 float32 range_min      # minimum range value [m]
24 float32 range_max      # maximum range value [m]
25
26 float32[] ranges       # range data [m] (Note: values < range_min or >
      range_max should be discarded)
27 float32[] intensities    # intensity data [device-specific units].  If
      your
28                       # device does not provide intensities, please
                             leave
29                       # the array empty.
```

Listing B.1: Definition of the ROS LaserScan message

```
# transform quarternion to euler to get robot yaw
quaternion = (
    data.pose.pose.orientation.x,
    data.pose.pose.orientation.y,
    data.pose.pose.orientation.z,
    data.pose.pose.orientation.w)
euler = tf.transformations.euler_from_quaternion(quaternion)
roll = euler[0]
pitch = euler[1]
yaw = euler[2]
# convert to range 0 degree to 360 degree
self.robot_yaw = (((yaw + 2*math.pi) % (2*math.pi)) * 360)/(2*math.pi)
```

Listing B.2: Calculation of the robots yaw.

```
Twist.msg:
# This expresses velocity in free space broken into its linear and angular
    parts.
Vector3  linear
Vector3  angular

---

Vector3.msg:
# This represents a vector in free space.
# It is only meant to represent a direction. Therefore, it does not
# make sense to apply a translation to it (e.g., when applying a
# generic rigid transformation to a Vector3, tf2 will only apply the
# rotation). If you want your data to be translatable too, use the
# geometry_msgs/Point message instead.

float64 x
float64 y
float64 z
```

Listing B.3: Definition of the ROS messages Twist and Vector3

```
1  def _find_depth_jumps_using_one_scan(self, scan):
2          # works for rotation, forwards and backwards but detects two
               points on the same wall as a gap if they are far enough apart
3          depth_jumps = np.zeros((len(scan),), dtype=int)
4          for angle in range(0, len(scan)):
5              tmp_angle = -2
6              # check if jump is large enough
7              if (scan[angle] != np.inf and scan[(angle + 1) % 360] != np.
                   inf and abs(scan[angle] - scan[(angle + 1) % 360]) >= self.
                   min_depth_jump):
8                  # find the shortest r
9                  tmp_angle = angle
10
11                 if scan[angle] > scan[(angle + 1) % 360]:
12                     tmp_angle = (angle + 1) % 360
13
14             elif scan[angle] == np.inf and scan[(angle + 1) % 360] != np.
                   inf:
15                 tmp_angle = (angle + 1) % 360
16
17             elif scan[angle] != np.inf and scan[(angle + 1) % 360] == np.
                   inf:
18                 tmp_angle = angle
19
20             if tmp_angle != -2 and scan[tmp_angle] < self.
                   max_r_to_depth_jump:
21                 # shortest r must be smaller then max distance
22                 depth_jumps[tmp_angle] = 1
23
24         return depth_jumps
```

Listing B.4: Implementation depth discontinuity detection using a single scan.

```
1  def _find_depth_jumps_using_two_scans(self, scan, scan_old):
2          scan = np.asarray(scan)
3          scan_old = np.asarray(scan_old)
4          scan[scan == np.inf] = self.max_r_to_depth_jump
5          scan_old[scan_old == np.inf] = self.max_r_to_depth_jump
6
7          depth_jumps = abs(scan - scan_old)
8          depth_jumps[depth_jumps < self.min_depth_jump] = 0
9          depth_jumps[depth_jumps > 0] = 1
10         depth_jumps = depth_jumps.astype(int)
11
12         return depth_jumps
```

Listing B.5: Implementation depth discontinuity detection using a two scans.

```
neighbour_count = 0
direct_neighbours = True
i = 1
while direct_neighbours and i <= range:
    a = discontinuities[(index - i) % len(discontinuities)] == 1
    b = discontinuities[(index + i) % len(discontinuities)] == 1
    direct_neighbours &= a or b
    if direct_neighbours:
        neighbour_count += 1
    i += 1
return count
```

Listing B.6: Checking for neighbour discontinuities to prevent false detection.

```
def _update(self, depth_jumps_last, discontinuities_two_scans,
    discontinuities_single_scans, rotation, robot_move):
  # rotation
  if rotation != 0:
      start = None
      end = None
      for_increment = None
      search_increment = None

      if rotation > 0:
      # rotation right
      start = 0
      end = len(depth_jumps_last)
      for_increment = 1
      search_increment = 1
      elif rotation < 0:
      # rotation left
      start = len(depth_jumps_last) - 1
      end = -1
      for_increment = -1
      search_increment = -1

      depth_jumps_last = self._correction(...)

  # forwards backwards movement
  if robot_move > 0:
      depth_jumps_last = self._correction_forward(...)
  elif robot_move < 0:
      depth_jumps_last = self._correction_backwards(...)

  if robot_move == 0 and rotation == 0:
      start = 0
      end = len(depth_jumps_last)
      for_increment = 1
      search_increment = 1
      depth_jumps_last = self._correction(...)

  return depth_jumps_last
```

Listing B.7: Algorithm to update the depth discontinuities.

```
# --- Forward correction settings ---
# 0 -> 179
start = 0
end = len(depth_jumps_last) / 2
for_increment = 1
search_increment = -1

# 359 -> 180
start = len(depth_jumps_last) - 1
end = len(depth_jumps_last) / 2
for_increment = -1
search_increment = 1#1

# --- Backwards correction settings ---
# 180 -> 0
start = len(depth_jumps_last) / 2
end = -1
for_increment = -1
search_increment = 1

# 180 -> 359
start = len(depth_jumps_last) / 2
end = len(depth_jumps_last)
for_increment = 1
search_increment = -1
```

Listing B.8: Configuration for the correction of forward and backward movement.

```
correction(depth_jumps, start_index, end_index, for_increment,
    search_increment, discontinuities_two_scan, discontinuities_single_scan
    ):
for i in discontinuities_two_scan:
  if discontinuities_two_scan[i] == 1:
    find_exact_position()
    determine_if_previously_known()
    if previously known:
      track_and_verify()
    else:
      check_for_neighbours()
      if no_neighbours:
        add to depth_jumps
  else:
    if depth_jumps[i] > 0:
      if discontinuities_single_scan[i] == 1:
        if depth_jumps[i] < max_depth_jump_recognition_count:
                                depth_jumps[i] +=
                                    recognition_increase_rate
      else:
        depth_jumps[i]--
    else:
      depth_jumps[i] = 0
  return depth_jumps
```

Listing B.9: Algorithm which matches the depth discontinuities at t with $t-1$.

```
split_detect, index_2 = check_for_split()
merge_detect = check_for_merge()

if merge_detect and not split_detect:
  merge(index, index_old_1, index_old_2)
  increase_count()
elif split_detect and not merge_detect:
  index_old = None
        if index_old_1 != None:
            index_old = index_old_1
        else:
            index_old = index_old_2
  split(index_old, index, index_2)
  increase_count()
elif index_old_1 != None or index_old_2 != None:
  index_old = index_old_1
        if index_old == None:
            index_old = index_old_2
  move(index_old, index)
  increase_count()
```

Listing B.10: Algorithm to for tracking and verification of depth discontinuities.

```
std_msgs/Header header
int32[] depth_jumps
float32[] range_data
int32 rotation
int32 liniear_x
```

Listing B.11: ROS message definishen for publishing the depth discontinuity information.

```
detect_critical_events(depth_jumps, rotation, movement):
  #rotation
  if rotation != 0:
    self._match_rotation_left_right(depth_jumps_last, depth_jumps,
        rotation)

  # forwards backwards
  if movement != 0:
    self._match_forward_backwards(self.depth_jumps_last, depth_jumps,
        movement)

  if movement == 0 and rotation == 0:
    self._match_drift_while_still_stand(self.depth_jumps_last, depth_jumps
        )
```

Listing B.12: Algorithm deciding what action to perform.

```
match_rotation_left_right(depth_jumps_last, depth_jumps, rotation):
        # rotation
        if rotation < 0:
            self._match_rotation(0, len(depth_jumps_last), 1,
                depth_jumps_last, depth_jumps)
        elif rotation > 0:
            self._match_rotation(len(depth_jumps_last) - 1, -1, -1,
                depth_jumps_last, depth_jumps)
```

Listing B.13: Algorithm to decide how to iterate over the array.

```
match_rotation(start_index, end_index, increment, depth_jumps_last, depth_jumps):
  depth_jumps_cp = copy.copy(depth_jumps)
    for index in range(start_index, end_index, increment):
        index_new = None

        # when at t-1 a depth jump was detected at this position, then try to find the new position of it
  # check for >= 1 because appear sets to 2, split to 3 and merge to 4
        if depth_jumps_last[index % len(depth_jumps_last)] >= 1:
            if depth_jumps_cp[index % len(depth_jumps_cp)] == 0:
                index_new = self._find_new_pos_of_depth_jump(depth_jumps_cp, index, increment)
                self._check_move_merge_disappear(depth_jumps_last, index, index_new, increment)
            else:
                # positions match, set copy to 0 so it can not get match with an other discontinuity
                depth_jumps_cp[index] = 0
                self.depth_jumps_last[index] = 1

    # depth_jumps now contains all new depth jumps
    for index in range(start_index, end_index, increment):
        if depth_jumps_cp[index % len(depth_jumps_cp)] == 1:
            if depth_jumps_last[index % len(depth_jumps_last)] == 0:
                index_old = self._find_new_pos_of_depth_jump(depth_jumps_last, index, increment)
                if index_old != None:
                    #move
                    self._discontinuity_moved(index_old, index)
                else:
                    # appear
                    self._discontinuity_appear(index)
            # positions matched at this point, set copy to 0 so it can not get match with an other discontinuity
            depth_jumps_cp[index] = 0
```

Listing B.14: Algorithm to match the rotation.

```
match_forward(depth_jumps_last, depth_jumps):
  depth_jumps_cp = copy.copy(depth_jumps)
        # 0 -> 179
        index = 0
        while index < (len(depth_jumps_cp) / 2):
    index = check_positive_direction()
          index += 1

        # 359 -> 180
        index = len(depth_jumps_cp) - 1
        while index >= (len(depth_jumps_cp) / 2):
    index = check_negative_direction()
              index -= 1

match_backwards(depth_jumps_last, depth_jumps):
  depth_jumps_cp = copy.copy(depth_jumps)
        # 179 -> 0
        index = (len(depth_jumps_cp) / 2) - 1
        while index >= 0:
    index = check_negative_direction()
              index -= 1

        # 180 -> 359
        index = len(depth_jumps_cp) / 2
        while index < len(depth_jumps_cp):
              index = check_positive_direction()
              index += 1
```

Listing B.15: Algorithm how the forward and backward movement is matched.

```
check_positive_direction(self, depth_jumps_last, depth_jumps_cp, index)
  index_new = None
    # move, merge, disappear
  # check for >= 1 because appear sets to 2, split to 3 and merge to 4
    if (depth_jumps_last[index] >= 1 and depth_jumps_cp[index] == 0):
        index_new = self._search_x_degree_positiv(depth_jumps_cp, index,
            5)
        if index_new == None:
            index_new = self._search_x_degree_negativ(depth_jumps_cp,
                index, 3)
        self._check_move_merge_disappear(depth_jumps_last, index,
            index_new, 1)

    if index_new == None:
        if depth_jumps_last[index] == 0 and depth_jumps_cp[index] == 1:
            # appear or split: try find node near (with in $3^{\circ}$)
            index_new_1, index_new_2 = self._check_split_appear(
                depth_jumps_last, depth_jumps_cp, index, +1)
            if index_new_2 != None:
                index = index_new_2
            else:
                index = index_new_1
        elif depth_jumps_last[index] == 1 and depth_jumps_cp[index] == 1:
            # gap stayed at the same index, set to 1
            depth_jumps_last[index] = 1
            # depth jump got processed at this point
            depth_jumps_cp[index] = 0
        else:
            # depth jump got processed at this point
            depth_jumps_cp[index] = 0
    return index
```

Listing B.16: Algorithm for checking from current index into positive direction.

```
1  match_drift_while_still_stand(depth_jumps_last, depth_jumps):
2      """
3      Match the depth jumps from previous step with the current when the
           robot is not moving by its values but slowly drifting of.
4
5      Parameters:
6      depth_jumps_last (int[]): Array indicating the depth jumps at t - 1
7      depth_jumps (int[]): Array indicating the depth jumps at t
8      """
9      depth_jumps_cp = copy.copy(depth_jumps)
10     for index in range(0, len(depth_jumps_last)):
11   # check for >= 1 because appear sets to 2, split to 3 and merge to 4
12         if (depth_jumps_last[index] >= 1 and depth_jumps_cp[index] == 0):
13             # check existing
14             index_new = None
15
16             if depth_jumps_cp[index - 1] == 1:
17                 index_new = index - 1
18             elif depth_jumps_cp[(index + 1) % len(depth_jumps_cp)] == 1:
19                 index_new = (index + 1) % len(depth_jumps_cp)
20
21             rospy.logdebug("drift␣while␣still␣stand␣␣-␣seq:␣" + str(self.
                   current_sequence_id) + "␣-␣index:␣" + str(index) + "␣-␣
                   index_new:␣" + str(index_new))
22             self._check_move_merge_disappear(depth_jumps_last, index,
                   index_new, +1)
23
24         elif (depth_jumps_last[index % len(depth_jumps_last)] == 0 and
               depth_jumps_cp[index % len(depth_jumps_cp)] == 1 and
               depth_jumps_last[index - 1] == 0 and depth_jumps_last[(index +
               1) % len(depth_jumps_last)] == 0):
25             # appear
26             self._discontinuity_appear(index)
27             depth_jumps_cp[index] = 0
```

Listing B.17: Algorithm for checking from current index into positive direction.

```
check_move_merge_disappear(depth_jumps_last, index_old_1, index_new,
    search_increment):
  if index_new != None:
    index_old_2 = self._check_merge(depth_jumps_last, index_old_1,
        search_increment)
    if index_old_2 == None:
        # move
        self._discontinuity_moved(index_old_1, index_new)
    else:
        # merge
        self._discontinuity_merge(index_old_1, index_old_2, index_new)
  else:
    # disappear
    self._discontinuity_disappear(index_old_1)
```

Listing B.18: Algorithm for checking for move, merge or disappear.

```
check_split_appear(depth_jumps_last, depth_jumps, index_new_1,
    search_increment):
    index_old = None
    index_new_2 = None

    # try to find index that splitted
    if search_increment > 0:
        index_old = self._search_x_degree_positiv(depth_jumps_last,
            index_new_1, 2)
    else:
        index_old = self._search_x_degree_negativ(depth_jumps_last,
            index_new_1, 2)

    if index_old != None:
        index_new_2 = self._find_second_depth_jump_from_split(
            depth_jumps_last, depth_jumps, index_new_1, index_old)

    if index_old != None and index_new_2 != None:
        # split
        self._discontinuity_split(index_old, index_new_1, index_new_2)

        depth_jumps_last[index_old] = 0
        # depth jump got processed at this point
        depth_jumps[index_new_1] = 0
        depth_jumps[index_new_2] = 0
    else:
        # appear
        self._discontinuity_appear(index_new_1)
        depth_jumps[index_new_1] = 0

    return index_new_1, index_new_2
```

Listing B.19: Algorithm for checking for move, merge or disappear.

```
# CriticalEvent.msg
CriticalEvent[] events

# CriticalEvent.msg
int32 event_type
int32 angle_old_1
int32 angle_old_2
int32 angle_new_1
int32 angle_new_2

# Event types
NONE = 0
APPEAR = 1
DISAPPEAR = 2
SPLIT = 3
MERGE = 4
```

Listing B.20: ROS message for publishing the critical events.

```
# MovedGaps.msg
GapMove[] gap_moves

# GapMove.msg
int32 angle_old
int32 angle_new
```

Listing B.21: ROS message for publishing the move of discontinuities.

```
# CollectionCriticalAndMoved.msg
CriticalEvents events
MovedGaps gap_moves
```

Listing B.22: ROS message to publish the critical events together with the moves.

```
class TreeNode:
    # 0 is reserved for the root node
    node_count = 1

    def __init__(self):
        self.id = TreeNode.node_count
        TreeNode.node_count = TreeNode.node_count + 1

        self.children = []
```

Listing B.23: Definition of the tree node in Python.

```
# GapTreeNodes
GapTreeNode[] tree_nodes

# GapTreeNode
int32 id
int32[] children_ids
```

Listing B.24: Working definition of the tree node as a ROS message.

```
# GapTreeNode
int32 id
GapTreeNode[] children
```

Listing B.25: Python definition of tree node from Listing B.23 transferred directly to ROS. This definition does not work due to the restriction that custom messages do not have a default value.