Nicolá Michel Henry Riedmann

A Semantic Map Implementation for a Long-Term Autonomous Robot

Bachelor's Thesis

Graz University of Technology

Institute for Softwaretechnology

supervised by
Dr. Gerald Steinbauer

July, 2016

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, 31st July, 2016 

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am 31 Juli, 2016 

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

This thesis presents an implementation of a Semantic Map that is to be used as part of a long-term autonomous robot system in the Long-Term Autonomy Project by the TU Graz, Institute for Software Technology.

To provide an understanding of the theoretical and practical foundations of the implementation, the thesis opens with a discussion of important concepts like Semantic Maps and Costmaps, as well as descriptions of related developments and utilized technologies.

The implemented Semantic Map is part of a ROS[2] based system and offers interfaces to OPRS[3] - the reasoning system used in the project.

Before the implementation, existing Semantic Map implementations were researched and considered for the project. The considered frameworks are presented and evaluated, but were not used due to various reasons.

The individual components of the implemented Semantic Map module are described in the Implementation section, which details the purpose and function of each component, as well as providing design-document like descriptions of classes, member variables and methods and ROS and OPRS specific functionality.

[2]Robot Operating System
[3]Open Procedural Reasoning System

Contents

1. Introduction

This thesis was done as a part of the Institute for Software Technology's Long-Term Autonomy robot project, which aims to have a wheeled robot patrol a building autonomously for a long period of time.

Using the Robot Operation System (ROS) [3] the project's robot software uses a occupancy grid for navigation, a system which simply put - and more elaborately described in section 2.2 - uses a grid whose entries signify either empty or occupied space, which can be used to plan paths around obstacles/occupied space.

This type of map poses an inherit constraint, as it can not model which objects are occupying space in the world, but only that space is occupied.

While this has a larger importance in applications like assistance robots that may, for example, need to know which object is a plate and which is a dishwasher, in order to complete the goal of putting the dishes in the dishwasher, adding some information about objects to the map and 'knowledge' of the robot software is a valuable addition to the aim of achieving long-term autonomous operation of the robot.

In this application the robot is to be given additional information about which objects in the world are doors, extending its knowledge by the state and probability of doors to be open or closed. This additional information can then be utilized by the planning software to make decisions about the general paths needed to patrol a building more efficiently.

The aim of this thesis is first, to give basic definitions of underlying concepts like semantic maps, occupancy grid based navigation and procedural reasoning.

Secondly, to examine existing semantic map implementations and their potential to be used for the project.

Lastly, it details the implementation of a basic semantic map representing the knowledge about doors in the world, as existing implementation were deemed unfit to be used in the project.

2. Related Research

2.1. Semantic Maps

This work's basic definition of a Semantic Map, which was already alluded to in the introduction, is that it is a map extended with or providing *semantic* information about *which* objects are occupying the world, other than just the information that they are occupying it.

In the introduction to their article 'Towards Semantic Maps for Mobile Robots' Nüchter and Hertzberg posit the following definition:

> A *semantic map* for a mobile robot is a map that contains, in addition to spatial information about the environment, assignments of mapped features to entities of known classes. Further knowledge about these entities, independent of the map contents, is available for reasoning in some knowledge base with an associated reasoning engine. [20]

Several more exact definitions exist, like Lang and Paulus' definition of a Semantic Map as a tuple $\mathbf{M}_{sem} =< \mathbf{M}, \mathbf{L}, \mathbf{A} >$, where $\mathbf{M}$ is a set of maps, $\mathbf{L}$ is a set of links between those maps and $\mathbf{A}$ is "a structure, which represents knowledge about the relation between entities, classes and attributes, also known as common-sense knowledge" [17]. This definition itself is an extension of Buschka's definition of hybrid maps [13], which posits the concept of a map consisting of a set of maps and a set of links between them.

2.1.1. Relation of The Implemented Semantic Map Module to the Definitions

The Semantic Map module implemented as part of this thesis conforms to both of those definitions.

The module consist of a database of doors, which contains both spatial information as well as further knowledge like the probability of a door entity to be open.

The spatial information is relayed to the robot's navigation software as a layer of the Costmap[1] used for navigation by the move_base package of ROS. This layered Costmap is a hybrid map, the Semantic Map's spatial information is thus an element of the Costmap's set of maps $\mathbf{M}$.

The reasoning engine used in the project is the Open Procedural Reasoning System (OPRS) to which the Semantic Map module provides interfaces, so that the semantic information can be used for reasoning.

2.1.2. Current Research

Most current research focuses on the automatic creation of maps with added semantic information, like the classification of encountered objects.

[1] The concept of Costmaps will be detailed in the following Section

This thesis and related implementation only cover adding existing semantic knowledge to a system. Whether this information is pre-compiled manually, or created by some form of automated classification based on sensor measurements is of no importance to that.

2.2. Costmaps

The concept of a Costmap is rather simple. Objects in the world are represented as cost values on a grid, usually with a threshold value defining when a cost represents an object that can not be passed, making lower values usable for comparing the cost of individual paths.

Entries in a Costmap can be inflated - widened by a given radius that is - to define regions that pose problems once the robot enters them, thus leading to avoiding getting to close to obstacles. Costmaps can be two or three dimensional grids, onto which the position of real world objects are projected.

In order to create a Costmap representing a robots surroundings automatically, it is necessary to map data captured by the robots sensors onto the grid representation while reducing the influence of noise that is often present in sensor measurements.

2.2.1. Occupancy Grids

A method described in detail by Elfes in 1989 [14] is using Occupancy Grids for the creation of a Costmap from a stream of multiple sensor measurements.

Occupancy Grids, like the basic Costmap concept, are a grid-based representation of space. Each grid cell contains a probabilistic random variable describing its state (e.g. occupied or empty). Elfes describes the problem of recovering a world representation from a robots sensor data as an estimation problem and thus advocates the use of probabilities to represent the occupation of a given cell in the model. As sensor data changes - as the robot moves, or noise in the measurements changes - the probability of a cell to have a given occupancy state is updated depending on its prior probabilities.

Together with the concepts of sensor-integration or -fusion [19] - using the data from several different sensors together - Occupancy Grids form the basis of obtaining Costmaps in current systems like ROS. [7]

2.2.2. Layered Costmaps

Unrelated to how a Costmap is obtained, it can benefit from the concept of hybrid maps mentioned before.

A layered Costmap thus consists of several individual Costmaps, whose composite forms the main map grid.

This can offer several benefits, many of which are described in detail by Lu, Hershberger and Smart [18].

Most importantly and directly related to the usage in this thesis, a layered Costmap allows to add occupancy data to a Costmap without influencing existing Costmaps.

It also allows for a layer of occupancy data to retain its context. For this thesis that means, that the grid representing the existence of doors in the world is organized in its own layer in a multilayer Costmap. At any time it is clear *what* obstacles in this

door-layer are, thus already adding some form of semantic information to a system that could otherwise consist of just one large grid of obstacles.

Additionally, layers can be used to influence a robot's behavior towards a type of obstacle. A layer whose obstacles are doors could have a larger inflation radius, leading to the robot keeping more distance and reducing the potential danger of being hit by an opening door.

2.2.3. Navigating with a Costmap

There are many different algorithms to implement path-planning, but the basis of all is the need to find the best (lowest-cost) path from one point to another, while evading all obstacles on the route.

While the lowest-cost path may simply be defined as the shortest path from A to B, the nature of a Costmap makes it possible to also consider the cost of obstacles that are not "lethal" - obstacles that can be passed by the robot - to decide on the cost of a path.

As an example, given a robot that has the capability to climb stairs, albeit slowly, the decision between a short path using stairs and a longer path around the stairs, can take into consideration a cost-value assigned to the stair area in the robot's Costmap.

3. Related Implementations

3.1. ROS - Robot Operating System

The Robot Operating System project is a framework meant to aid the creation of robot software by offering "tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms" [3].

Originally developed by robotics company Willow Garage, it was released as open source in 2007 and has since then steadily expanded and improved, being used in academic as well as commercial projects and often benefiting from the results and developments of these projects.

ROS forms the basis of the robot software used for the long term autonomy project, its modules taking care of things like driving the robot-platform's motors, planning paths based on a Costmap or creating that Costmap from sensor data.

As there are no papers on the functionalities of the ROS implementations, the information below is based on the online documentation of ROS.

3.1.1. Costmap Implementation

The ROS Costmap module uses two layered 2D Costmaps. [7]

One global Costmap, containing the model of the entire known surroundings of the robot, and thus mainly being filled with static data like a non-changing floorplan, and being used for long-term global planning. And one local Costmap representing the more immediate surroundings of the robot, filled with current information from sensor measurements and used for planning obstacle avoidance and other local movements. [10]

A ROS Costmap is based on Occupancy Grids and generally consists of several layers, both predefined and additionally implemented. What layers a Costmap contains, as well as specific configuration for these layers, can be defined in a configuration file for both the local and global Costmap.

The generally used predefined layers are, a static layer - modeling unchanging parts of the world like the basic floorplan -, an obstacle layer - modeling obstacles that are measured by sensors - and an inflation layer which contains the added grid entries around obstacles.

The cell entries in a ROS Costmap are either Free, Occupied/Lethal Obstacle or Unknown, each of which are self-explanatory named and are used by ROS' path-planning methods.

While the Costmap is a 2D grid, the obstacle layer uses 3D Voxel data by default, so that some form of 3D Costmap can be achieved using ROS.

3.1.2. Concepts and Functionality Used in the Implementation

This section gives a short overview of a few basic concepts that are used in the implementation outlined in the following chapter.

ROS Nodes

A ROS system consists of a number of Nodes - programs that run in the ROS environment, use ROS functions and offer their own functionalities, generally to other Nodes - and a ROS Core that offers all basic functionality needed for Nodes to run and "communicate with one another using streaming topics, RPC services, and the Parameter Server." [9].

The implemented Semantic Map Server that handles loading and modifying the database of doors as well as the OPRS interface implement ROS Nodes. The door layer plugin is run as part of ROS' move_base Node.

ROS Topics and Messages

ROS offers a publish/subscribe communication framework that allows Nodes to publish on, or subscribe to Topics. [11] The Node publishing a Topic defines its name and the type of Messages sent to subscribers. Subscribers subscribe to the topic via its name and must implement a function that is able to handle the Message type of the topic.

ROS offers a range of predefined Messages, but new Messages can be declared in .msg files. [8] Such a Message description consists only of Message values and their associated type, defined as [type] [name], each on a new line. Available types are all Message types, meaning that both pre-defined types (like int32, float64, etc.) as well as self-defined Messages can be used as types inside a Message.

ROS' buildsystem generates code for all Messages defined in a package's *msg* folder and declared in a specific *add_message_files* block in the package's Makefile.

Some samples of Message definitions can be found in Section 4.2.5 in the Implementation chapter.

Actionlib Actions

Nodes can offer Actions that can be called by clients via the methods offered by ROS. An Action is a specified service offered and implemented by a node. While ROS already offers a mechanism to define services, Actions have the benefit of having a status (like 'in-progress' or 'done') and offering the possibility to be canceled during their execution. [6]

Similar to Messages, Actions are defined in .action files. These need to be placed in a package's *action* folder and declared in a special block in the Makefile for ROS to create corresponding code when building the package.

Action definitions are somewhat more complex than Message definitions, consisting of three blocks, each containing variables in the same way as messages. An Action definition consists of a goal Message, which is sent by a client to an Action-Server to call an Action; a result Message, which is sent by the server to the client once the Action has finished (either by executing normally or being finished in any other way); and a feedback Message which can be sent by the server to the client at any point during the execution of an Action, e.g. to notify the client of the current execution progress. Additionally a client can send a cancel Message, which is not specifically defined in the .action file.

Some samples of Action definitions can be found in Section 4.2.4 in the Implementation chapter.

Move Base and Costmap2D

The move_base ROS node is an integral part of ROS navigation functionality, consisting of the aforementioned Costmaps and planners and linking those to the inputs of sensor and odometry data, transformation definitions and an output to the robots movement controller.

The Costmap2D package provides the implementations of Costmaps that can also be used to define new Costmap layers.

3.2. Procedural Reasoning Systems

Procedural reasoning systems are architectures ''for representing and reasoning about actions and procedures in a dynamic domain". [16]

Such an architecture consists of a database of current beliefs defined by logical predicates, a set of goals the system is trying to achieve and a library of plans that consists of the procedural knowledge defining the system's capabilities.

Based on its current goals and beliefs, the system is capable of choosing the correct plans it needs to execute, to reach its goals. At every step an executed procedure may lead to new beliefs or sub-goals to be added to the systems database, resulting in it choosing additional or different plans to execute.

The mentioned Procedural Knowledge, is knowledge about how certain procedures are to be performed. In the context of this thesis an example procedure might be "charge batteries" detailing the steps that must be carried out to reach the goal of "full batteries" and possibly resulting in sub-goals that trigger procedures like "go to charging station" or "dock at charging station".

A PRS is thus a system that, based on its knowledge about the world, can choose the correct predefined procedures to reach its current goals, and can react to the changes to the world its actions might entail, by choosing other fitting procedures.

3.2.1. Concepts of PRS

Predicates

One part of PRS' Procedural Knowledge is formed by logical Terms and Predicates - boolean functions that given a Term, or several Terms return true or false.

Functions

Functions in PRS are functions on Terms, that return a Term.

Actions

Actions are PRS' way of triggering the execution of some activity. Syntactically similar to Functions, they work on and return Terms, but generally have some consequence outside of the logical system. An example from the OPRS documentation would be the Action OPEN_VALVE in an automated pressure control system, that, as the name suggests, opens a valve, a physical action that can thus be triggered by the reasoning system.

3.2.2. Procedural Reasoning with OPRS

The Open Procedural Reasoning System [15] is an implementation of a Procedural Reasoning system, more precisely an open source continuation of the original PRS system developed by SRI International and used at NASA.

It allows for predicates, functions and actions used by the reasoning system to be defined in a C library and thus can interface with other systems like ROS.

In the Long-Term Autonomy Project, OPRS is used as the main planning/reasoning component.

3.3. Existing Semantic Map Frameworks and Reasons for Not Using Them in This Project

Before the decision was made to implement the Semantic Map module from scratch, several frameworks were researched and their potential to be used in the project was evaluated. The following sections contain descriptions of these frameworks, their benefits and the reasons that eventually lead to deciding against using them.

The URLs of the projects' repositories can be found in Appendix D.

3.3.1. KnowRob

KnowRob is described as a "knowledge processing system" combining several types of knowledge, ranging from a robot's gathered sensor data to procedural knowledge. All knowledge in the KnowRob system is defined in the Web Ontology Language [5] and stored in ontologies as part of a global Knowledge Base held by the system. This Knowledge Base is used as the basis of the system's reasoning, which is defined in first-order logic. [21]

The project combines its Knowledge Base with several reasoning systems, functionality modules and interfaces to systems like ROS. [2]

KnowRob does not only offer a Semantic Map implementation, it is largely based on using the semantic knowledge that its Knowledge Base is good at representing, as part of its reasoning systems.

The main problem presented by KnowRob is thus, that it is such a large framework that it would be necessary to use it for other parts of the project than just the Semantic Map. The decision was made, that using KnowRob's Knowledgebase-based reasoning instead of OPRS was not desirable for this project.

Another problem that presented itself when researching KnowRob's Semantic Map implementation, is that it, as well as the whole project, seems targeted towards 3D representations and high level knowledge about indoor environments. A large amount of projects, papers and thus current framework development focus on acquiring and using semantic knowledge in complex assistance robot applications. The KnowRob Semantic Map seems more focused on complex semantic knowledge like "this is a kitchen, the goal is to put the dishes into the dishwasher. Commonsense knowledge dictates that the dishwasher is near the sink, etc.". Again KnowRob solves more complex problems than handling the simple added semantic information that is supposed to be added to this project.

KnowRob's problems are also its benefits though. It is a large and well documented framework, offering a vast array of functionalities for projects that need it and are willing to commit to using its knowledge and reasoning system as their basis.

3.3.2. STRANDS

STRANDS (Spatio-Temporal Representation and Activities for Cognitive Control in Long-Term Scenarios) is a EU funded research project focused on researching and providing software for long-term autonomous robots acting in human environments. [4]

Like KnowRob the project has an understandable focus on 3D environments, also made evident by the fact that its Semantic Map implementation is part of its 3D mapping framework.

While initially researching the implementation it seemed light on features, sparsely documented and proved to be highly interconnected to the projects navigation framework, which is built as an extension of ROS' navigation stack with an emphasis on navigation through human environments.

At the time of writing this thesis, after completing the implementation, STRAND's Semantic Map implementation has been updated, now showing a focus on creating room information as part of a mapping process. Showing the projects connection the current research topic of automated semantic mapping, but entirely ruling out the potential of using it for the project's intended purpose.

Again the Semantic Map approach of the framework is too specialized for the needs and framework of the project, to be used independently, and the large, in-progress and comparably badly documented state of the projects rules it out as a candidate to be used as a larger-scale basis of the project.

3.3.3. Other open-source implementations

Some smaller open-source Semantic Map implementations were discovered and evaluated during the research phase of this thesis.

Each of these are again specialized solutions as part of larger frameworks. In each of the three cases they are part of commercial developments that were open-sourced. The benefit of each project is that they are ROS based implementations, so that they at least somewhat influenced the design process and implementation of the eventually implemented semantic map module.

Yuyin Robot's open source libraries include a *Semantic Navigator* integrating the system's waypoint based navigation with some semantic information about those waypoints. The dependence on the rest of Yuyins navigation framework and focus on semantic information for waypoints or potentially regions, made the implementation unusable for this project.

Also copyrighted partially as owned by Yuyin is Jorge Santos Simóns world_canvas ROS framework. Implemented in Python it uses a map implementation based on occupancy grids, but not obviously equivalent to the standard ROS Costmap. Added to a map server that receives its maps over a connection to some warehouse server, is an annotation-server and scripts offering functions to mark map regions as a certain type (e.g. chair/wall/etc.).

Robots in Concert, a project developed as part of ROS, aims to integrate robots into larger systems making them "part of the solution" [1] and using robots as part of a service architecture. ROCONs open-sourced projects include a no longer developed module for handling semantic information about regions. As it does not directly interface with the ROS Costmap and is written in python it turned out to be of little use.

4. Implementation

4.1. Basic Concept

To fulfill its task as defined in section 2.1, the Semantic Map module has to both extend the map used by the robot for navigation, as well as offer additional semantic information about objects in the world. Specifically this information is to be used for higher level planning and reasoning, a task that is handled in this project by OPRS.

As this Semantic Map implementation only concerns itself with pre-existing knowledge about where in the world objects are located, it uses a database of such objects, initially loaded at startup and expandable during runtime. The limitation of only representing doors, makes the basic semantic information of what type of object an obstacle is, implicit.

The door-database contains the entire knowledge about doors in the world, both their position as obstacles for navigation as well as their semantic attributes.

The main component of the semantic map module, called the server and described in detail below, holds the current database.

Two high-level interfaces exist, connecting the server's door database to the Navigation and the Reasoning systems.

4.1.1. The Map - Extension of the Multi-Layered ROS Costmap

The ROS costmap2D module provides a way to extend its layered Costmap by additional layers.

The Semantic Map module uses this to extend the Costmap by a door layer, which contains the 2D obstacle information used by the ROS move_base navigation system.

This layer contains obstacle values representing a door's position, by placing a 2D line on the map grid for each closed door.

This is implemented in the lta_semantic_map_layer component.

4.1.2. The Semantic Information - Integration to OPRS

OPRS provides methods to define additional actions, evaluable predicates and evaluable functions in a C/C++ library.

The lta_semantic_map_oprs library defines several such user actions, predicates and functions, allowing the reasoning system to use the information about doors for planning, as well as to make changes to the database.

The design and implementation of each of the module's components is described in detail in the following sections. Figure A.1 shows an overview of how the components interact and depend on each other.

4.2. Server Component

4.2.1. Purpose

The lta_semantic_map_server is the main component of the module.

It implements a ROS node, loads the door database and holds it at runtime, making it necessary for the server to be running for other components to function.

The door database is loaded from a .yaml file[1], whose path is passed to the server as a parameter at startup. From the defined database in the file, the server constructs a collection of Door objects. The datastructure used to hold the Door objects is a Map - a structure linking a Key to an Object or set of objects - using a Door's ID as key linking to the Door object.

The server uses ROS' messaging functionality to publish the current door-database whenever a change to it occurs. A latch mechanism that can be activated when creating a ROS publisher, ensures that new subscribers to the published topic receive the latest published message as they subscribe. This ensures that all components that depend on receiving the door database messages receive the current state of the database, independent of when they subscribe to the topic. This mechanism prevents the problem that a subscriber may not have any database until a change occurs and it is republished. The latch mechanism is more convenient and bandwidth conserving, than having to frequently publish the potentially large unchanged door-database to ensure new subscribers receive it.

The server implements four Action-Servers that define actions to modify the database. The offered actions are defined in more detail below, but in short allow to add, remove or modify database entries, as well as save the current Map to a yaml file.

4.2.2. Foundations / Dependencies

The server component is dependent on ROS, as the server's main functionalities are based on a ROS Publisher and ActionServers. It was tested with ROS Indigo, but as the functionality used is basic, it should work with other distributions, at least newer ones.

Database YAML Structure

The server can load a database of doors from a .yaml file. Yaml object definitions consist of keys and values, a collection of objects is simply several such objects, each on their own line. Entries in the door database file must follow format defined in Figure 4.1 below, parts marked with a $ are variables that should be replaced with the respective value.

Like in the Map held at runtime, door objects have an ID that is used as a key to identify them. When creating a database file, assigned IDs should be unique, as doors are referenced by their ID all throughout the system.

All values except for the status are floating point real numbers. The status is either *OPEN, CLOSED, UNKNOWN*, if the server is not able to read OPEN or CLOSED as the status, it will assign UNKNOWN as the created Door object's status.

[1] YAMLAin'tMarkupLanguage, a standard for data serialization designed to be easily readable by humans and used throughout ROS

```
$ID:
        position:
                x: $X_POSITION
                y: $Y_POSITION
                theta: $THETA
        width: $WIDTH
        status: $STATUS
        closed_prob: $PROBABILITY_CLOSED
        open_prob: $PROBABILITY_OPEN
```

Figure 4.1.: Schema of the yaml definition of a Door.

4.2.3. Data Structures

As mentioned above, the server holds the door-database in a Map of Door objects.
A diagram of the classes detailed below can be found in Figure A.3 in the appendix.

Door

The Door class defines a Door based on the representation used for the yaml database.

It has the same member variables as defined in the yaml listing above, with three exceptions.

The id is not only used as the Door object's key in the database, it is also a member of the Door.

The position is an object of the Position class described in the following section.

The status is a variable of type Door_Status, which is an enum defined in the Door class. The values of the Door_Status enum are the same three as described previously and the enum is defined as public in the Door class and used for comparisons in other parts of the module.

The Door class defines the typical getter and setter functions for its private member variables [2] , as well a few additional functions.

bool isOpen() and **bool isClosed()** are convenience methods to get information about the Door's state, without needing to compare to the Door_State enum.

setOpen(), setClosed(), setUnkown() are convenience methods for setting the state of a Door object, without having to use the enum as a function parameter.

Getter and Setter for the state are also available.

std::string getYamlString() creates and returns a string representation of the Door. This string adheres to the definition of a door entry in the yaml database and the function is used when storing the current Map as a yaml file.

[2] In keeping with object-oriented design principles all member variables of C++ classes defined in this module are private and accessed only through class methods.

Position

The Position class defines a position consisting of an x and y position and an angle theta, and defines the typical getter and setter functions for these member variables. All members of a Position are of type double.

4.2.4. Actions

The lta_semantic_map_server defines ActionServers for each of the following actions.

AddDoor

The AddDoor Action allows a client to create a new Door entry in the server's database.

The goal message defines all properties of a Door object, except of the id, which is set to the next available index in the Map by the server.

The set id is returned to the client in the the result message, if the Door object could be created and added successfully.

The Action also defines a feedback message, which is currently never sent by the server, due to how quickly the required tasks can be performed.

The definition of the AddDoor action can be found in Figure B.1 in the appendix.

RemoveDoor

The RemoveDoor Action allows a client to remove a door from the server's database.

The goal message contains the id of the Door to be removed.

The server returns a boolean signifying if the Door could be removed successfully.

The Action also defines a feedback message, which is currently never sent by the server, due to how quickly the required tasks can be performed.

The definition of the RemoveDoor action can be found in Figure B.2 in the appendix.

SaveDoorDB

The SaveDoorDB Action allows a client to request the server to store its current database.

The goal message specifies the filename or path the data should be saved to. The server saves the database as a yaml file, using each Door's toYamlString() function to write the entries of the file.

The server returns a boolean signifying if the database could be saved successfully.

The Action also defines a feedback message, which is currently never sent by the server, due to how quickly the required tasks can be performed.

The definition of the SaveDoorDB action can be found in Figure B.3 in the appendix.

UpdateDoor

The UpdateDoor Action allows a client to update a Door entry in the server's database.

The goal message consists of the Doors id, a boolean for each member of a Door object, signifying if this value should be updated, and a value for each member. The

booleans ensure that only values the client wants to change are changed, without the client needing to know and specify the current values in the message.

The server returns a boolean signifying if the entry was updated successfully.

The Action also defines a feedback message, which is currently never sent by the server, due to how quickly the required tasks can be performed.

The definition of the UpdateDoor action can be found in Figure B.4 in the appendix.

4.2.5. Messages

The server publishes its current database to the */lta_semantic_map_door_db* topic as previously described.

It sends a message of the DoorMap type, which contains an array of DoorEntry messages. These represent a Door by consisting of all of a Door's members. The Door's status is sent in the message as a string and parsing from the string to the Door_State enum must be handled by the callback method of topic subscribers.

The definitions of the DoorMap and DoorEntry messages can be found in Figures C.1 and C.2 in the appendix.

4.2.6. Members

As the server is not a object oriented class, but simply a C++ program with some global variables and functions, it does not technically have member variables, it has several important variables essential to its execution though, and these are listed here.

Publisher db_publisher used to publish DoorMap messages on the door_db topic.

map <int, Door*> door_db which holds the database of Door objects, or more precicly pointers to Door objects.

string door_db_filename which defines the path the door database is loaded from when the server is started. It is set when it is passed as a parameter.

Created in the server's main-method, the server also defines one ActionServer for each of its offered Actions and a ROS NodeHandle that is needed to define the Publisher and ActionServers as part of the component's ROS Node.

4.2.7. Functions

bool load_door_db(string yaml) loads the door database from the passed yaml file. It returns false if any error occurs, true otherwise.

bool store_door_db(string yaml) stores the door database to a yaml file whose filepath is specified by the passed parameter. It returns false if any error occurs, true otherwise.

int add_door(const Position position, const double width, const Door:: Door_State state, const double prob_open, const double prob_closed) creates a new Door and adds it to the database. It returns the Doors id, which is assigned as

the first free index at the end of the door database.

bool remove_door(int id) removes the Door with the passed id from the database. It returns true if the Door existed and was removed successfully, false otherwise.

bool update_door_pos(int id, const Position position) sets the position of the Door with the given id to the passed value. It returns true if the Door exists and the position was changed successfully, false otherwise.

bool update_door_width(int id, const double width) sets the width of the Door with the given id to the passed value. It returns true if the Door exists and was changed successfully, false otherwise.

bool update_door_state(int id, const Door::Door_State state) sets the state of the Door with the given id to the passed value. It returns true if the Door exists and was changed successfully, false otherwise.

bool update_door_prob_open(int id, const double prob_open) sets the open probability of the Door with the given id to the passed value. It returns true if the Door exists and was changed successfully, false otherwise.

bool update_door_prob_closed(int id, const double prob_closed) sets the closed probability of the Door with the given id to the passed value. It returns true if the Door exists and was changed successfully, false otherwise.

void execute_add(const lta_semantic_map::AddDoorGoalConstPtr& goal, AddServer* as) is the callback function executed when the AddServer receives a Goal message. It parses the received message and calls the add_door function.

void execute_update(const lta_semantic_map::UpdateDoorGoalConstPtr& goal, UpdateServer* as) is the callback function executed when the UpdateServer receives a Goal message. It parses the received message and calls all required update_door_* function to make the requested changes.

void execute_remove(const lta_semantic_map::RemoveDoorGoalConstPtr& goal, RemoveServer* as) is the callback function executed when the RemoveServer receives a Goal message. It parses the received message and calls the remove_door function.

void execute_storedb(const lta_semantic_map::SaveDoorDBGoalConstPtr& goal, SaveDBServer* as) is the callback function executed when the StoreDBServer receives a Goal message. It parses the received message and calls the store_door function.

lta_semantic_map::DoorMap make_db_message() creates a DoorMap message from the door database, by creating a DoorEntry message from each of the Doors in the database and adding those to the message's array. It is used to create the message

published whenever the database changes.

4.3. Costmap Layer Component

4.3.1. Purpose

The lta_semantic_map_layer implements a costmap2d layer and provides edited launch and parameter files that add the layer to the system's layered Costmap.

This layer places a line of obstacle filled cells into the Costmap grid for every closed Door, and sets these cells to free space if the Door is open, or to unknown space if its state is not known.

4.3.2. Foundations / Dependencies

The door_layer consists of a DoorLayer class that extends both the ROS costmap2d::Layer and Costmap2D classes. It does this in order to be loaded as part of a layered Costmap and to be able to access a full Costmap grid in which to mark the occupied Door positions.

As mentioned in the server's description, the DoorLayer subscribes to the published door_db topic and constructs its own Map of Door objects from the received message. It is thus necessary for the server to be running and publishing a database for the layer to place any objects on the Costmap.

Bresenham's Line algorithm

The layer uses the compact version of Bresenham's line algorithm, an algorithm for drawing a line between two points on a mesh grid first published in 1965. [12]

Using a Door's known Position - x,y and angle Θ - as well as its width, its end point can be calculated. Knowing these points, the line drawing algorithm draws the gird approximation of the line between them into the Costmap grid.

The basic functionality of the algorithm is to progress one grid cell from the current position - initially the start position - in the direction of the end position at every step, until the end position is reached.

While the original algorithm needs to differentiate based on the octant the line is currently directed in and updates only in x- or y-direction, the compact version operates with only quadrants of the plane, using a sign variable for x and y and can update both x- and y-directions in one step.

The compact algorithm uses the difference between end and start position x and y values to calculate the difference variables dx and dy. These are added together to form the initial error. The mentioned sign variables sx and sy are set to 1 or -1, depending on the line's direction, and are not only defining the sign, but also used to progress the current position. Thus it is only necessary to differentiate based on the direction once, when initially setting sx and sy.

The doubled error is used in every step to compare to the values dx and dy, and increase both the error and current position based on the result of the comparison.

The compact algorithm as it is implemented in the DoorLayer class is printed in Figure 4.2 below.

```
int dx = abs(x1-x0);
int dy = -abs(y1-y0);
int sx = x0<x1?1:-1;
int sy = y0<y1?1:-1;
int err = dx+dy;
int e2;
while(1){
        set(x0,y0)      /*ROS and implementation specific
                          costmap setting code omitted*/
        if (x0==x1 && y0==y1){
                break;
        }
        e2 = 2*err;
        if (e2 > dy){
                err += dy;
                x0 += sx;
        }
        if (e2 < dx){
                err += dx;
                y0 += sy;
        }
}
```

Figure 4.2.: Compact Bresenham's algorithm as implemented in door_layer.cpp.

4.3.3. Data Structures

The DoorLayer depends on the Door and Position classes, as it holds a Map of Doors just like the server component.

As mentioned, the component is implemented in the DoorLayer class, which extends both the costmap2d Layer and Costmap2D classes and is a ROS node. A class diagram of DoorLayer can be found in figure A.2 in the appendix.

4.3.4. Messages

The DoorLayer subscribes to the lta_semantic_map_door_db topic published by the server component. It parses the DoorEntry messages in the received DoorMap message in order to create its own Map of Door object that is equivalent to the servers current database.

Descriptions of the Messages received by this component can be found in section 4.2.5 above.

4.3.5. Members

NodeHandle nh the DoorLayer ROS Nodes handle.

ros::Subscriber sub used to subscribe to the door_db topic published by the server component.

std::map <int,Door*> door_db the internal Map of Doors created from the received messages.

dynamic_reconfigure::Server <costmap_2d::GenericPluginConfig> *dsrv_ a reconfigure server that ROS plugins must offer, to allow for them to be reconfigured at runtime. Needed because the door layer is loaded by the layered costmap as a plugin.

4.3.6. Functions

Public Functions

DoorLayer() the DoorLayer's constructor, in the current implementation nothing happens in it.

void onInitialize() the initialization method of the costmap. Map initialization as well as registering the reconfigure server and subscribing to the door_db topic is handled in this function

void updateBounds(double robot_x, double robot_y, double robot_yaw, double* min_x, double* min_y, double* max_x, double* max_y) is the internal update function of the Costmap2D. In this function the compact Bresenham algorithm is used to set the costmap cells corresponding to Door positions. The cells are set to either *LETHAL_OBSTACLE, FREE_SPACE* or *NO_INFORMATION* based on a Door's state.

void updateCosts(costmap_2d::Costmap2D& master_grid, int min_i, int min_j, int max_i, int max_j) is the function in which the Layer's costmap values are set in the costmap grid of the master costmap it is a part of. Only cells that contain information (Obstacle or Free) are set in the master grid.

bool isDiscretized() a costmap method, set to return true.

virtual void reset() used to reset the costmap and to clear the internal Door database. Can be called externally and is also called whenever a new database message is received from the server.

void matchSize() sets the costmap dimensions of the costmap layer to those of the master costmap.

Private Functions

void door_db_message_callback(const lta_semantic_map::DoorMapConstPtr& msg) is the callback function in which received DoorMap messages are parsed and Door objects are created and put into the components Map of Doors.

Position get_end_point(Position p, double width) is a method used to calculate the end point of a line, based on its start position, width and angle.

void reconfigureCB(costmap_2d::GenericPluginConfig &config, uint32_t level) is the callback method for the reconfigure server needed by a plugin.

4.4. OPRS Interface Component

4.4.1. Purpose

The OPRS Interface component utilizes OPRS' offered functionality of defining evaluable predicates, functions and actions in a C library that can be loaded when running OPRS.

It is the interface between the reasoning system and the map server, allowing OPRS to access information about, and making changes to the database of Doors.

4.4.2. Foundations / Dependencies

The component is implemented as a C++ library that imports several OPRS C++ files and uses the defined functions and types to declare functionalities for OPRS. It needs to be loaded in a OPRS kernel to have any function. An OPRS include file and opf file that allows calling the defined actions can be found in the module's oprs_test folder.

4.4.3. Data Structures

The OPRS-Interface depends on the Door and Position classes, as it holds a Map of Doors just like the server component.

4.4.4. Actions

The component implements clients for the UpdateDoor and RemoveDoor Actions. See section 4.2.4 for their definitions.

The clients are used to call the ROS Actions, when the fitting OPRS actions are called. The UpdateDoor client is used by several OPRS actions, as they split the Update functionality into several actions (e.g. SET_DOOR_OPEN or SET_OPEN_PROBABILITY) to make calling these actions in OPRS easier.

The standard functionality of clients to wait for a result message from the server (or a specified timeout) before continuing, seems to interfere with OPRS. Using the waitForResult method will sometimes result in an endless loop, making using OPRS impossible. This can occur even when a timeout is set. At this time I am not aware of the exact cause of this problem, but it may be due to Action Clients handling their execution in a separate thread and this somehow interfering with OPRS' execution. The problem seems to occur irregularly, except when all OPRS trace options are set, in which case it always occurs.

As a workaround, clients used in the OPRS Actions do not wait for a result after sending a goal to their respective server. On the OPRS side, they simple report success, once the goal message was sent. This is not a good solution, but as the Actions are performed quickly and generally succeed, this should pose less of a problem than the execution breaking bug that may occur otherwise.

When using the Actions one should thus not rely on whether they return true or not.

4.4.5. Messages

The OPRS-Interface subscribes to the lta_semantic_map_door_db topic published by the server component. It parses the DoorEntry messages in the received DoorMap message in order to create its own Map of Door object that is equivalent to the servers current database.

Descriptions of the Messages received by this component can be found in section 4.2.5 above.

4.4.6. Functions

The component contains several functions that define the functionality of declared OPRS functions, predicates or actions, these are listed in the OPRS section below.

The other functions of the component are listed in this section.

void door_db_message_callback(lta_semantic_map::DoorMapConstPtr& msg) is the callback function in which received DoorMap messages are parsed and Door objects are created and put into the component's Map of Doors.

bool call_door_update(int id, bool set_as_open, bool set_as_closed, bool change_prob_open, bool change_prob_closed, double prob) calls the UpdateAction to make changes based on the provided parameters. It returns true if the Action was completed successfully. It is used by all OPRS actions that make changes to Doors and thus need to call UpdateActions.

void declare_user_eval_funct(void) uses the function defined by OPRS to declare all evaluable functions defined by the OPRS-Interfaces component.

void declare_user_eval_pred(void) uses the function defined by OPRS to declare all defined evaluable predicates.

void declare_user_action(void) uses the function defined by OPRS to declare all defined actions.

extern "C" void init_semantic_map_oprs(void) is the function that needs to be called when loading the library in OPRS. It initializes a ROS Node, subscribes to the door_db topic and creates the action clients. It also creates an AsyncSpinner used to be able to receive new Messages, without blocking the execution of the rest of the program.

This function also calls the three declare functions described above, and thus ensures that the OPRS-Interfaces can be used by OPRS.

4.4.7. OPRS

The following sections describe the user defined predicates, functions and actions that the OPRS-Interfaces library adds to OPRS, it also notes their respective C++ functions. The definitions are to be read similarly to the definitions in the OPRS documentation, the notation being *RETURN-TYPE* **NAME** *(PARAMETER-TYPE).*

All of the defined predicates/functions/actions that need a door as a parameter, take the Door parameter as an Atom Term of the form *DOOR_$ID*, where $ID is to be replaced by the referenced door's ID.

Evaluable Predicates

PBOOLEAN **DOOR_STATE_KNOWN** *(ATOM)* is the predicate to check if the state of a given Door is known. It is defined for 1 term. It returns TRUE if the state of the Door is not unknown, FALSE otherwise.

The respective C++ function is **PBoolean door_state_known(TermList tl)** which parses the input and checks whether it was valid and the Door exists, before returning a PBoolean.

PBOOLEAN **DOOR_OPEN** *(ATOM)* is the predicate to check if a given Door is open. It is defined for 1 term. It returns TRUE if the Door is open, FALSE otherwise.

The respective C++ function is **PBoolean door_open(TermList tl)** which parses the input and checks whether it was valid and the Door exists, before returning a PBoolean.

PBOOLEAN **DOOR_CLOSED** *(ATOM)* is the predicate to check if a given Door is closed. It is defined for 1 term. It returns TRUE if the Door is closed, FALSE otherwise.

The respective C++ function is **PBoolean door_open(TermList tl)** which parses the input and checks whether it was valid and the Door exists, before returning a PBoolean.

Evaluable Functions

FLOAT **PROBABILITY_DOOR_OPEN** *(ATOM)* is a function to get the probability of the given Door to be open. It is defined for 1 term. It returns the probability as FLOAT, if the Door exists and has an open probability, NIL otherwise.

The respective C++ function is **Term* door_open_probability(TermList tl)** which parses the input and checks whether it was valid and the Door exists, before returning a Term pointer.

FLOAT **PROBABILITY_DOOR_CLOSED** *(ATOM)* is a function to get the probability of the given Door to be closed. It is defined for 1 term. It returns the probability as FLOAT, if the Door exists and has a closed probability, NIL otherwise.

The respective C++ function is **Term* door_closed_probability(TermList tl)** which parses the input and checks whether it was valid and the Door exists, before returning a Term pointer.

PBOOLEAN **GET_CLOSED_DOOR** *(VARIABLE)* is a function to get any closed Door. It is defined for 1 VARIABLE term that will be assigned with an ATOM representing a closed door, if one is found. It returns TRUE if a closed Door exists, NIL otherwise.

The respective C++ function is **Term* get_closed_door(TermList tl)** which parses the input and checks whether it was valid and a closed Door exists, before assigning the passed variable and returning a Term pointer.

PBOOLEAN **GET_OPEN_DOOR** *(VARIABLE)* is a function to get any open Door. It is defined for 1 VARIABLE term that will be assigned with an ATOM representing a open door, if one is found. It returns TRUE if a open Door exists, NIL otherwise.

The respective C++ function is **Term* get_closed_door(TermList tl)** which parses the input and checks whether it was valid and a open Door exists, before assigning the passed variable and returning a Term pointer.

Actions

PBOOLEAN **REMOVE_DOOR** *(ATOM)* is the action to remove the given Door from the database. It is defined for 1 term. It returns TRUE, if the remove goal was sent successfully, NIL otherwise.

The respective C++ function is **Term* remove_door(TermList tl)** which parses the input and checks whether it was valid, before calling the RemoveAction and returning a Term pointer.

ATOM **SET_DOOR_OPEN** *(ATOM)* is the action to set the status of the given Door to open. It is defined for 1 term. It returns TRUE, if the 'set to open' goal was sent successfully, NIL otherwise.

The respective C++ function is **Term* set_door_open(TermList tl)** which parses the input and checks whether it was valid, before using the call_door_update method and returning a Term pointer based on the value returned by the method.

ATOM **SET_DOOR_CLOSED** *(ATOM)* is the action to set the status of the given Door to closed. It is defined for 1 term. It returns TRUE, if the 'set to closed' goal was sent successfully, NIL otherwise.

The respective C++ function is **Term* set_door_closed(TermList tl)** which parses the input and checks whether it was valid, before using the call_door_update method and returning a Term pointer based on the value returned by the method.

ATOM **SET_OPEN_PROBABILITY** *(ATOM FLOAT)* is the action to set the open probability of the given Door to the given float value. It is defined for 2 terms, an ATOM defining the Door to be changed, and the FLOAT value the probability shall be set to. It returns TRUE, if the goal to change the probability was sent successfully, NIL otherwise.

The respective C++ function is **Term* set_open_probability(TermList tl)** which parses the input and checks whether it was valid, before using the call_door_update method and returning a Term pointer based on the value returned by the method.

ATOM **SET_CLOSED_PROBABILITY** *(ATOM FLOAT)* is the action to set the closed probability of the given Door to the given float value. It is defined for 2 terms, an ATOM defining the Door to be changed, and the FLOAT value the probability shall be set to. It returns TRUE, if the goal to change the probability was sent successfully,

NIL otherwise.

The respective C++ function is **Term* set_closed_probability(TermList tl)** which parses the input and checks whether it was valid, before using the call_door_update method and returning a Term pointer based on the value returned by the method.

4.5. Server Action Clients

The implementation also includes two clients for the server component's offered Actions.

The lta_semantic_map_action_test_client simply calls a few of the server's actions and was implemented as the name suggest to initially test the implementation of the Actions.

During the development of the semantic_map module a second client was implemented to make testing easier.

The lta_semantic_map_cmdline_tool offers a simple command-line interface to call all the Actions provided by the server component. While proving valuable during testing the module, the command-line tool might still be useful to make changes to the Door database or trigger saving it, when the module is in practical use.

The test_client was also left in place and may be seen as a sample implementation of using the Action messages in a client implementation.

5. Conclusion

The main aim of this thesis - finding or creating a suitable Semantic Map implementation for the Long-Term Autonomy Project - was fulfilled.

Sadly none of the researched frameworks were suitable for this application, as they were either too large in scope, or too specialized to a problem domain or software stack.

A recurring problem notable in the STRANDS project and the small open-source projects, is the existence of many special-case semantic map implementations, but few to no general purpose semantic map frameworks that interface with ROS.

The software implemented as part of this thesis retains some of these problems, itself solving the specialized problem of creating a semantic map representing only doors and interfacing with OPRS as well as the ROS Costmap implementation.

It offers the potential of being extended to different domains of objects though, by making the necessary changes to the data and Message structure, changing the update Actions and the way that objects are drawn into the Costmap grid.

Additional interfaces to reasoning or planning systems can be built the same way the OPRS interfaces were built, working with the published list of doors and defined actions.

The object oriented nature of the C++ implementation also offers the possibility of a more generalized implementation, with classes like the Door inheriting from a Object super-class and offering important methods like the Costmap draw function.

As creating a general Semantic Map implementation was not the aim of this thesis, this extension may be a project worth pursuing in the future, but the specialized implementation fulfills its requirements for now.

Bibliography

[1] http://www.robotconcert.org/ robots in concert - wiki. http://www.robotconcert.org/index.php/Main_Page, note = Accessed: 2016-07-21.

[2] knowrob.org knowrob. http://www.knowrob.org/knowrob. Accessed: 2016-07-21.

[3] ros.org about ros. http://www.ros.org/about-ros/. Accessed: 2016-07-21.

[4] strands.acin.tuwien.ac.at project. http://strands.acin.tuwien.ac.at/project.html, note = Accessed: 2016-07-21.

[5] w3.org owl - web ontology language. http://www.w3.org/TR/owl-features/. Accessed: 2016-07-24.

[6] wiki.ros.org actionlib. http://wiki.ros.org/actionlib/DetailedDescription. Accessed: 2016-07-23.

[7] wiki.ros.org costmap_2d. http://wiki.ros.org/costmap_2d. Accessed: 2016-07-21.

[8] wiki.ros.org messages. http://wiki.ros.org/Messages. Accessed: 2016-07-23.

[9] wiki.ros.org nodes. http://wiki.ros.org/Nodes. Accessed: 2016-07-23.

[10] wiki.ros.org robotsetup - costmap configuration. http://wiki.ros.org/navigation/Tutorials/RobotSetup. Accessed: 2016-07-21.

[11] wiki.ros.org topics. http://wiki.ros.org/Topics. Accessed: 2016-07-23.

[12] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[13] Pär Buschka. *An investigation of hybrid maps for mobile robots*. PhD thesis, Örebro University, Department of Technology, 2005.

[14] Alberto Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57, 1989.

[15] Francois F. Ingrand. Oprs development environment. 2014. Version 1.1b7.

[16] Francois F. Ingrand, Michael P. Georgeff, and Anand S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert: Intelligent Systems and Their Applications*, 7(6):34–44, December 1992.

[17] Dagmar Lang and Dietrich Paulus. Semantic maps for robotics. In *Proc. of the Workshop "Workshop on AI Robotics" at ICRA*, 2014.

[18] David V Lu, Dave Hershberger, and William D Smart. Layered costmaps for context-sensitive navigation. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 709–715. IEEE, 2014.

[19] Hans P Moravec. Sensor fusion in certainty grids for mobile robots. *AI magazine*, 9(2):61, 1988.

[20] Andreas Nüchter and Joachim Hertzberg. Towards semantic maps for mobile robots. *Robot. Auton. Syst.*, 56(11):915–926, November 2008.

[21] Moritz Tenorth. *Knowledge Processing for Autonomous Robots*. Dissertation, Technische Universität München, München, 2011.

A. Diagrams

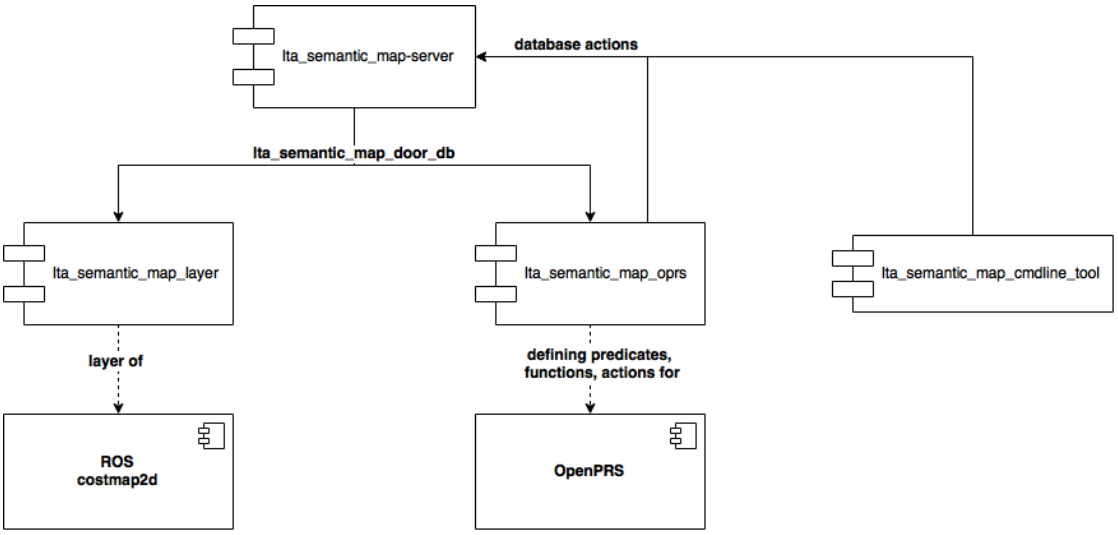

Figure A.1.: Diagram detailing the relations between the components of the module.

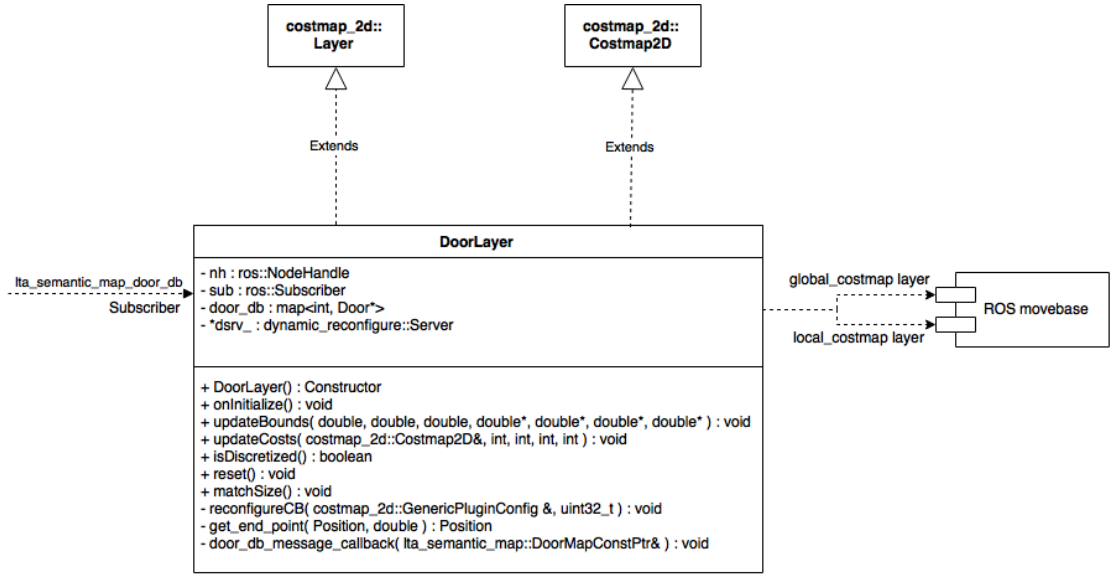

Figure A.2.: Classdiagram of the DoorLayer class of the costmap component.

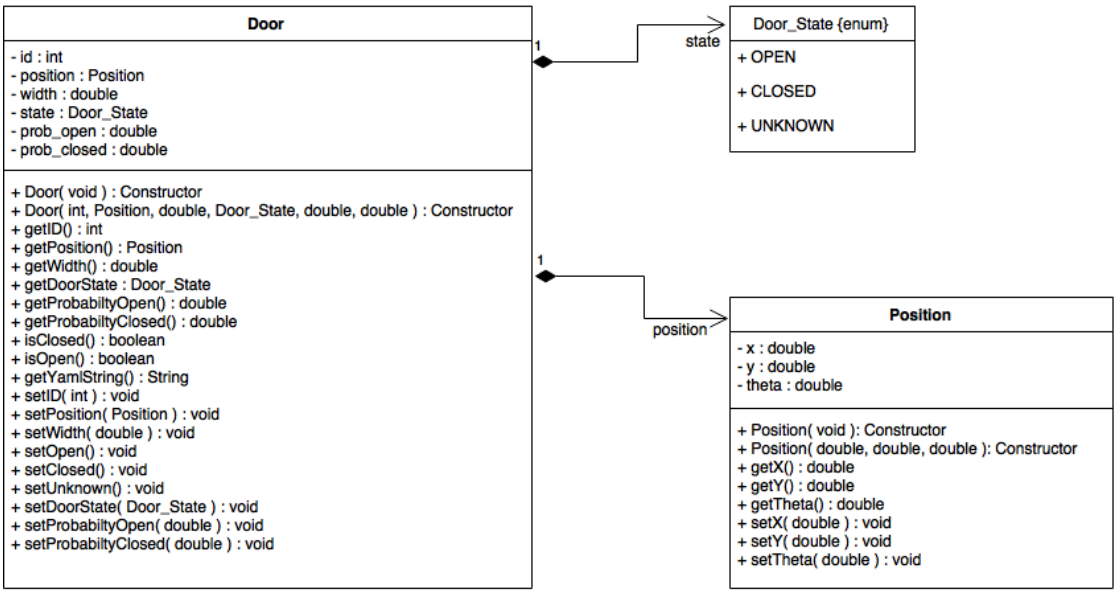

Figure A.3.: Classdiagram of the Server data classes.

B. ROS Action Definitions

```
#goal
float64 pos_x
float64 pos_y
float64 pos_theta
float64 width
string door_state
float64 prob_open
float64 prob_closed
---
#result
uint32 door_id
---
#feedback
float32 percent_complete
```

Figure B.1.: AddDoor.action definition.

```
#goal
uint64 door_id
---
#result
bool door_removed
---
#feedback
float32 percent_complete
```

Figure B.2.: RemoveDoor.action definition.

```
#goal
string file_name
---
#result
bool stored_successful
---
#feedback
float32 percent_complete
```

Figure B.3.: SaveDoorDB.action definition.

```
#goal
uint64 door_id
bool change_pos
bool change_width
bool change_open
bool change_prob_open
bool change_prob_closed
float64 pos_x
float64 pos_y
float64 pos_theta
float64 width
string door_state
float64 prob_open
float64 prob_closed
---
#result
bool door_updated
---
#feedback
float32 percent_complete
```

Figure B.4.: UpdateDoor.action definition.

C. ROS Message Definitions

```
DoorEntry[] door_db
```

Figure C.1.: DoorMap.msg definition.

```
uint32 door_id
float64 pos_x
float64 pos_y
float64 pos_theta
float64 width
string door_state
float64 prob_open
float64 prob_closed
```

Figure C.2.: DoorEntry.msg definition.

D. Semantic Map Frameworks - Software Repositories

KnowRob	`https://github.com/knowrob/knowrob`
STRANDS	`https://github.com/strands-project`
Yuyin Semantic Navigator	`https://github.com/yujinrobot/yujin_ocs`
World Canvas	`https://github.com/corot/world_canvas`
Rocon Semantic Regions	`https://github.com/stonier/graveyard_rocon_database`

E. Test of the Influence of the Semantic Map on the ROS-based Navigation

E.1. Test Setup

In order to test the influence the Semantic Map implementation has on the ROS move-base navigation, multiple tests were run using the Stage[1] simulator.

For the test the simulated robot needs to reach 100 randomly chosen goals in the simulated world. The floorplan of the university building floor on which the IST Robotics Lab is located was used as the base map of the simulation.

Two .yaml databases containing all doors on the floor were created - one to be used with the Semantic Map and one for a program that adds obstacles to the Stage simulation representing the actual doors in the world.

Three additional small programs were created that fulfill several tasks needed to run the simulation.

These can be found in the /test folder of the lta_semantic_map_package and are the DoorDaemon, GoalSender and StageDoorSensor.

DoorDaemon

In order to simulate a real-world scenario in which doors are opened and closed over time, the DoorDaemon program is used.

Upon startup it calculates the likelihoods for a door to be opened or closed (toggled will be used for simplicity from here on) based on the time t at which it has last been toggled.

It uses a normal distribution based on the assumption that a door's likelihood to be toggled increases for a while after it has last been toggled and then begins to decrease over time. To represent this, the normal distribution has $\mu = 6.0, \sigma = 4.0$ (values below 0 are ignored) . People may leave a room shortly after they have entered it (e.g. if it is a bathroom), or a while after they have entered (e.g. if is a break-room or their office), but the longer a door is open or closed, the less likely it gets for it to be toggled (think of some seldom used storage closet, or an entry way that is simply kept open). To ensure that doors do not get stuck in one state for too long a base-probability for change is added, so that it may change, even when the probability from the distribution has become low.

In order for the test to offer any valuable results it is assumed, that doors are not generally kept shut, but that their state simply changes from open to closed.

The DoorDaemon holds a list of all doors as well as a list containing the timestamp at which any door was last toggled.

These timestamps are simply an incremented number, not the physical time the door was toggled.

After 24 iterations the door's timestamp will reset to 0, as the changes were originally intended to happen hourly, which proofed too large a delay for the simulation. As it

[1] `http://playerstage.sourceforge.net/doc/Stage-3.2.1/index.html`

makes for no notable difference, and frequent changes of the door states were inteded that mechanism was left in place, all thought the delay between iterations was reduced.

While the test is running, the DoorDaemon will generate a random variable for every door, and compare it to the probability entry that relates to time since the doors last toggle. If the random variable is equal or larger to the defined probability, the door will be toggled - which means the DoorDaemon will use an action call to prompt the Stage object representing the door to be moved - either to it's closed position in it's respective door frame, or to an open position outside the map.

Between every iteration the DoorDaemon will delay for 15 minutes, not changing the state of doors during that timeframe.

StageDoorSensor

To be able to make changes to the entries of the Semantic Map a mock sensor that simulates a sensor that is capable of recognizing doors in the world was implemented.

For this the StageDoorSensor program subscribes to the ros topics on which the semantic map door database and the position of the robot are published.

It also *listens* on any action goals sent in relation to the Stage door objects and builds an internal database of the simulated doors and whether they are currently open or closed.

During the test the StageDoorSensor will check whether the robot is currently close to door object in the simulation and will then check if there is also a Semantic Map entry for the door.

If the state of the simulated door and entry differ (i.e. if the Stage object is in place (closed), but the semantic map entry is set to be open, or vice-versa), the mock sensor will correctly identify the discrepancy with 90 percent accuracy (again a random variable is generated to the check the probability) and update the entry in the semantic map database.

GoalSender

The GoalSender is the main program of the simulation test.

It sends a specified number of goals to the movebase, and waits for the robot to reach one goal before sending another.

Each goal is a randomly chosen 2D point on the map and may be unreachable to the robot - either due to it being in occupied space on the map, or by being in a place that is currently blocked by a closed door.

The second function of the GoalSender program is timekeeping and thus producing the comparable data used to evaluate the test.

It creates records of

- the total runtime,
- the total time spent reaching goals,
- the average, fastest and slowest times to reach a goal
- the number of tried goals

- the number of goals that could not be reached
- the number of goals that could not be reached after the robot had to drive there
  - this data was acquired based on how long the movebase took to notify the GoalSender that a goal could not be reached. It was found that it would reject a goal after about 20 seconds if it did not manage to plan a path to it. Any unreached goal that took longer to be reject is counted for this figure.

The last point is of importance as these are the points, for which the global planner could not immediately decide that they could not be reached, so they represent cases in which the robot had to drive to a position to find the it's path is blocked by a now closed door.

E.1.1. Test Runs

Two types of tests were run. Initial tests were run with entirely (pseudo) random values for goals and to check probabilities, while for a second run the same randomized values were used for each test setup.

In each test there were four distinct setups which are detailed in the section below.

For the initial random test two test-runs were executed for each test setup.

For the subsequent *equally randomized* test just one test-run was executed for each setup.

In each test-run the robot had to reach 100 goals.

Setups

The first test was run using no semantic map, but the standard costmap setup containing an obstacle layer in the global costmap. This means that laser scans of obstacles are present in the map used for global path-planning.

The second test was run using the semantic map, StageDoorSensor and the standard costmap.

The third test was run using no semantic map and a costmap setup that does not include obstacles in the global costmap.

The fourth test was run using the semantic map, StageDoorSensor and a costmap setup that does not include obstacles in the global costmap.

E.2. Test Results

In the following tables the test setups that did not include obstacles in the global costmap are marked with a star (*). Additionally the tests are included in the tables in the same order as their descriptions above (first to fourth).

The percentage entries in the tables denote how many percent of the total time the average, fastest and slowest times represent, which is useful for comparing results.

E.2.1. Entirely Random Tests

While two runs of 100 goals were executed for each test setup, the results shown in the tables are condensed into one entry representing the combined results of both tests, to increase readability.

RUNTIMES										
	Total		for Goals		Average		Fastest		Slowest	
	sec	h	sec	h	sec	%	sec	%	sec	%
No Map	66181	18.4	53519	14.9	121	0.18	0[2]	0	27366	41.35
Map	37794	10.5	22387	6.2	96	0.25	0	0	301	0.80
No Map*	57162	15.9	23279	6.5	113	0.2	0	0	319	0.56
Map*	65695	18.2	42981	12	109	0.17	0	0	21728	33.07

NUMBER OF GOALS			
	# goals tried	# goals unreachable after driving	% of total
No Map	582	40	6.87
Map	864	55	6.37
No Map*	1665	187	11.23
Map*	1707	71	4.16

E.2.2. Fixed Randomized Tests

RUNTIMES										
	Total		for Goals		Average		Fastest		Slowest	
	sec	h	sec	h	sec	%	sec	%	sec	%
No Map	21629	6	11885	3.3	118	0.55	0	0	278	1.29
Map	20387	5.7	13833	3.8	138	0.68	0	0	349	1.71
No Map*	20006	5.7	11565	3.2	115	0.58	0	0	264	1.32
Map*	20036	5.6	10841	3.0	108	0.54	0	0	256	1.28

NUMBER OF GOALS			
	# goals tried	# goals unreachable after driving	% of total
No Map	511	20	3.91
Map	372	11	2.96
No Map*	219	49	22.37
Map*	383	36	9.4

[2]Throughout all tests the GoalSender generated at at least one goal that could be reached by the robot in less than a second. Thus the recurring value 0, and a field that is of little use for this discussion of results.

Figures E.1 to E.4 show histograms of the runtimes for the fixed randomized test-runs. The runtimes were split into 50 bins for the histogram, and a dashed red line showing the Gaussian distribution for the data's mean and standard deviation is overlayed. The plots show that the data does in some cases follow the normal distribution slightly, while being notably different in some other cases, even to the total and average times are close to one another for all runs.

E.3. Discussion of Results and Conclusions

E.3.1. Entirely Random Tests

What is immediately notable from these results is that they vary, this is of course due to the inherit randomness of the whole test. Not only is the selection of goals random, but also the toggling of doors. As such, in some test more goals had to be tried and in some test the robot may have gotten locked in and stuck in a room - the data suggests that this occurred at least once in test No Map (run # 2) and Map* (run #2), each showing that over 30% of the total time was spent on the slowest goal run.

That is why the discussion of the results will focus on the respective percentages and not the actual values.

Preliminary to the test the assumption was formed, that the mechanisms of the semantic map would not actually lead a notable improvement of the performance of the path planner.

As mentioned above, the usual configuration of the move_base's costmaps contains the obstacles discovered by laserscans in the map used for global path planning. Thus the planner already has access to the map data represented by the semantic map and its costmap layer - which contains occupied space where closed doors are, or free space where doors are open. This data is updated by the StageDoorSensor when the robot is close to a door, and thus in exactly the same occasion as the laser scan is integrated into the global costmap.

To prove or disprove these assumptions, the tests without an obstacle layer in the global costmap were ran, the expected outcome being that the tests without the semantic map would perform worse than before, and the ones using the semantic map should have similar performance as other tests.

The actual runtime figures have little informative power.

Ignoring the two outliers in which the slowest run took over 30% of the total runtime, the robot spent around 3 hours of every test-run reaching goals, the total run-times being between 4 and 6 hours.

Reaching single goals took the robot about 2 minutes or 0.2% of the total time in most runs, regardless of whether the semantic map was used or not and also regardless of the presence of obstacles in the global costmap.

The number of goals that had to be come close to, before being able to determine that they could not be reached offers more insights than the run-times.

In the test runs without the semantic map a combined 6.87% of tried goals had to be visited to be the deemed unreachable.

In the test runs with the semantic map it was a combined 6.37%.

In the test runs without the obstacles in the global costmap, but with the semantic map it a combined 4.16% of unreachable goals had to be visited first. While this number is lower than the others, it can be assumed to be due to other circumstances, like the large total amount of unreachable goals in the second run.

In these three tests - all offering information about the state of doors in the world to the global path planner - a very similar and small amount of goals had to be come close to to discover they could not be reached. These cases are most like to be instances in which the robot has passed an open door before, but it has since been closed.

The tests without a global obstacle layer and without the semantic map saw the robot having to drive to a combined 11.23% of unreachable goals.

As this figure is close to double the amount from the other tests and the other figures are close to each other, this strongly suggests that the initial assumption was correct, and the semantic map and global obstacle layer have a very similar effect on the performance.

E.3.2. Fixed Randomized Tests

The tests using the same randomized variables were executed to remove the differences between runs caused by the randomness of goal coordinates and the state of doors.

While some random differences still seem to be present in these tests, in general results are very close to one another.

Looking at the run-times, each test took around 6 hours to execute, with 3-3.8 hours spent reaching goals.

The average time to reach goals was around 2 minutes for each run, the slowest times being between 4.4 - 5.8 minutes.

No real significant differences between setups can be discerned from the runtimes, as they are all quite close together.

Again the number of goals that had to be come close to, before deciding that they were unreachable - those instances in which it can be assumed, that a door closed since the robot last passed it and updated it's information (be it the semantic map entries or the obstacle layer in the global costmap) - provides more useful data.

While the number of goals tried in total varies between runs, we will again look at the percentage of goals that were only deemed unreachable after driving close to them.

The runs with a global obstacle layer and either using or not using the semantic map are quite close to one another, with around 3% and 4% of unreachable goals having to be come close to. There is a slight improvement of 1% when using the semantic map, although with the slight differences in runtimes, that may lead to the robot being at different locations when the DoorDaemon changes door states, the single run is too little evidence to assume this is an actual improvement due to the semantic map.

The run using no global obstacle layer and no semantic map performed notably worse than all others, having to drive to 22.37% of all unreachable goals.

The important run is the one without global obstacle layer but using the semantic map, which performed notably better than the run not using the semantic map, but worse than either of the runs using the obstacle layer in the global costmap.

This may be due to the mock door sensor having only a 90% probability of correctly identifying a door as open or closed and then still being dependent on a ROS action to be called and executed successfully before the semantic map's door database is updated.

Conclusions

In conclusion it was shown that the semantic map offers little to no benefit to the movebase's path-planning. At best it functions similarly to the (standard) inclusion of an obstacle layer in the global costmap used for path-planning to goals.

Nevertheless the Semantic Map could potentially increase path-planning performance if combined with a mechanism of *smart* doors, that are able to notify the system of their current status, based on which the robots door database could be updated and current data be fed into the global path planner.

As this test does only consider one part of the semantic map - it's spatial mapping information - these results should not be viewed as evidence that the semantic map is of no use.

It can still be useful to a higher-level planner that is not planning how to reach a goal, but planning which goal should be tried next.

Together with the potential information of how likely a door is to be open or closed - something that can be inferred from observations whenever passing a known door - a global planner could decide on the most efficient way to patrol the floor, potentially visiting doors that are likely to be open first and leaving doors that are very unlikely to be open at a later point.

An improved version of the system may even store and take into account at what time a given door is likeliest to be open.

Furthermore the semantic map still offers the *semantic* information that obstacles aren't merely obstacles, but doors - that by a fittingly equipped system could be opened.

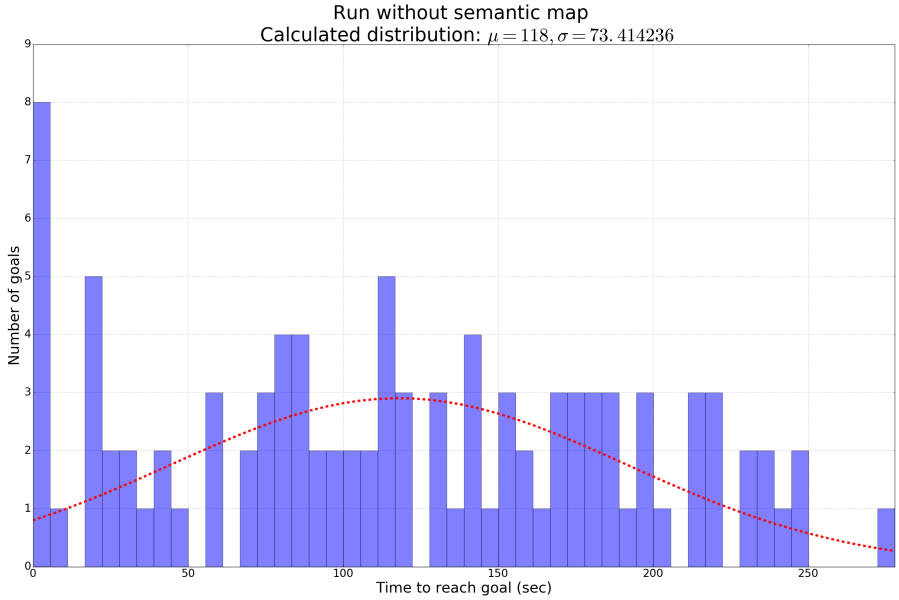

Figure E.1.: Histogram of runtimes for test-run without semantic map (NoMap).

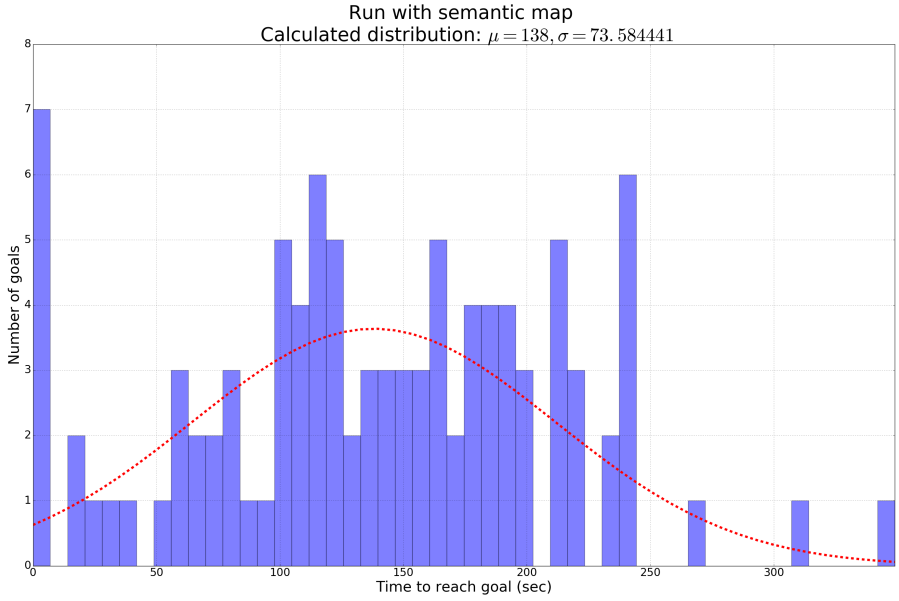

Figure E.2.: Histogram of runtimes for test-run with semantic map (Map).

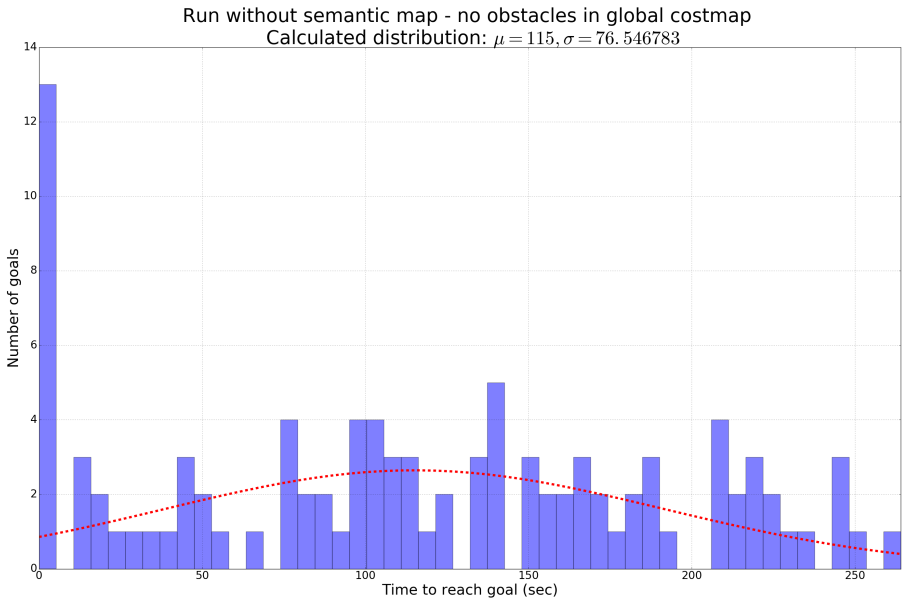

Figure E.3.: Histogram of runtimes for test-run without semantic map and no global obstacle layer (NoMap*).

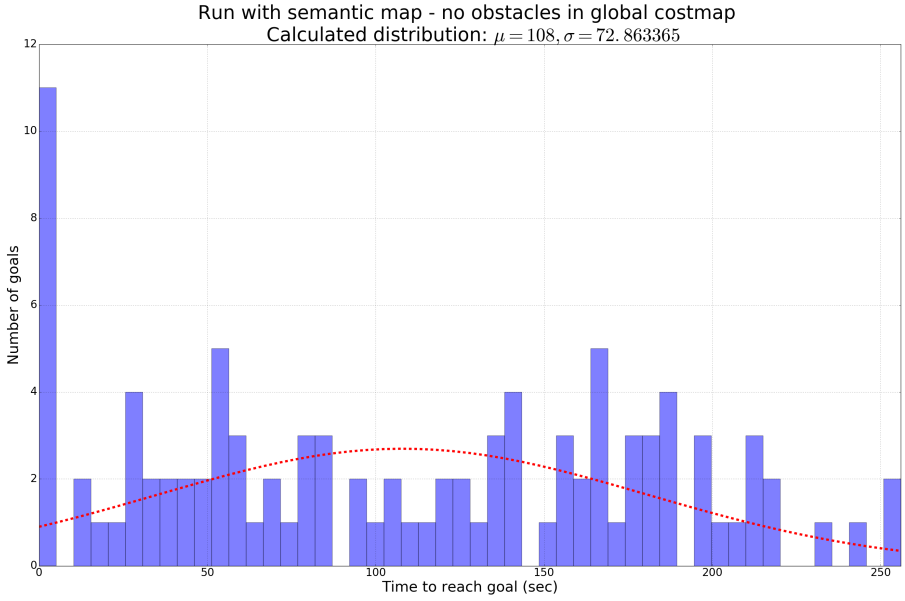

Figure E.4.: Histogram of runtimes for test-run with semantic map and no global obstacle layer (Map*).