ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute for Dynamic Systems and Control
IDSC
Institut für Dynamische Systeme und Regelungstechnik

Johannes Boghaert

Global Localization

Autonomous Mobility on Demand

Final Report GOTO-1

Institute for Dynamic Systems & Control
Swiss Federal Institute of Technology (ETH) Zürich

Supervision

Merlin Hosner
Gioele Zardini

Dr. Jacopo Tani
Dr. Andrea Censi

December 2019

Abstract

The goal of this project is to navigate a single Duckiebot within a predefined Duckietown lay-out from any starting point A to a randomly generated arrival point B. The Duckiebot is - with GOTO-1 - able to start driving and autonomously navigate through Duckietown in such way that it uses the existing infrastructure only to localize and navigate, that it follows the shortest path possible and that it reaches the desired arrival point with an acceptable accuracy. This means that GOTO-1 uses existing, standard infrastructure as landmarks to locate itself within the city, and that it uses these landmarks as nodes to calculate the shortest path within the predefined Dijkstra graph representing Duckietown. Although in a development phase, we hope this project can form the basis for future localization & navigation projects.

There and back again - A tale of death and rebirth

Global Localization

1. Mission & Scope

The goal of the GOTO-1 project is to **localize** a single Duckiebot within Duckietown (DT), efficiently **navigate** between two randomly generated points and to **stop** the Duckiebot with an acceptable level of accuracy.

1.1 Motivation

Currently, there is no system - without the watchtower set-up - available to localize a single Duckiebot within a given Duckietown map, and to subsequently navigate into the city and stop at a desired end point (e.g. a docking station or parking area). The motivation behind this project is hence to add functionality to Duckietown, and to reflect the realistic commute of single Duckiebots within a city. The latter being similar to single navigation systems used in today's commute.

1.2 Existing solution

The GOTO-1 project has not been tried before, and no existing solutions were at hand for the Duckiebot to localize itself and travel to a predefined point of arrival. In other words, the GOTO-1 project features a newly developed global localization package (addressing localization, planning, navigation and stopping - further explained in the next sections). Yet, the entire system required for the mission of global localization is built upon the existing framework of *indefinite navigation* (daffy version). The latter thereby provides in a.o. apriltag (AT) detection, lane following and intersection navigation. All of which can be found here.

Note: An existing framework to define the Duckietown map configuration automatically is duckietown-world. Due to the reduced scope and manpower of this project, the duckietown-world functionality has not (yet) been integrated within GOTO-1.

1.3 Opportunity

As explained earlier, the GOTO-1 project adds a desirable functionality to Duckietown for a single Duckiebot within a city not fitted with a watchtower system. In addition, it is a unique opportunity to have a localization and path planning module implemented within Duckietown, reflecting today's navigation systems.

1.4 Preliminaries

In order for the reader to fully grasp the GOTO-1 context and functioning, it is advised to familiarize oneself with the Duckietown project, and the development version (DAFFY) of the Duckietown environment on Github. Of particular interest are the appearance specifications of Duckietown, and the *indefinite navigation* demo.

2. Problem Definition

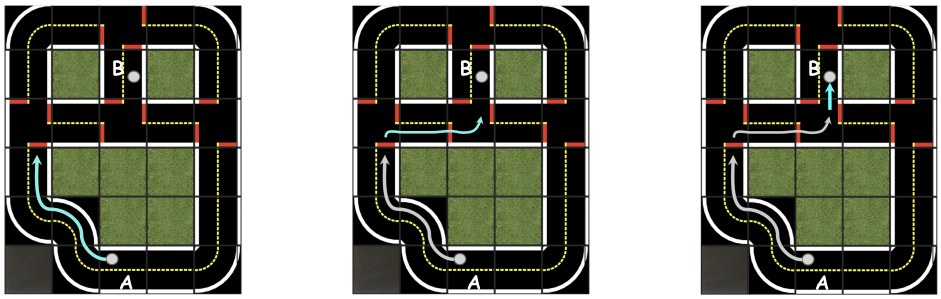
In order to grasp the complexity and approach of the GOTO-1 project, the problem definition of this project is outlined, and divided into subproblems.

2.1 Task description

As mentioned in the previous section, the goal of GOTO-1 is to get information on the location of the Duckiebot, and from there navigate as efficiently as possible by following the shortest path towards a final destination point and stop the Duckiebot there.

2.2 Task Decomposition

As a result of the aforementioned task description, the GOTO-1 project addresses the following three sub-problems as seen in the figure.



(a) Localization / Planning (b) Navigation (c) Arrival / Stopping

Fig.1 The three sub-problems visually represented in a template Duckietown configuration with A the starting position of the Duckiebot and B the final destination

2.3 Performance and Evaluation Metrics

Corresponding to the three subtasks, three performance criteria can be defined for each subtask, while the overall robustness and stability of the GOTO-1 solution is assessed by the success rate of each subtask and of the entire pipeline.

- The Duckiebot is able to localize itself in Duckietown upon AT detection
- The Duckiebot is able to define the shortest path from its current location to the predefined arrival point, and is able to navigate itself accordingly
- The Duckiebot is able to stop at the arrival point with a certain accuracy

Evaluating (quantifying) the first two criteria is done by respectively verifying the determined Duckiebot position and shortest path to the actual one in a (virtual) Duckietown map representation. This is easily done for smaller Duckietown configurations, and can naturally be extended to the case of larger configurations. The final criteria was (initially) assessed as follows, in order of increasing performance:

- Arrival lane stopping accuracy
- Tile-level stopping accuracy (within 60cm of the destination point)
- Sub-tile-level stopping accuracy (within 30cm of the destination point)

2.4 Assumptions & Restrictions

In order for these performance metrics to be reachable and make sense, the following assumptions are made:

- The map of Duckietown is defined as in the *path planning* module (reducing implementation flexibility) - also explained further
- There are no varying lighting conditions (reducing robustness)
- There are no external factors s.a. obstacles on the roads (reducing robustness)
- An acceptably functioning *indefinite navigation* demo version is available (reducing stability through interdependencies)

In addition, we restrict ourselves to the following scenario within Duckietown:

- Only a limited density of apriltags (AT) can be used, i.e. no other AT's but the ones from the Duckietown appearance guideline can be used.
- Traffic rules within Duckietown need to be taken into account (i.e. no U-turns, driving directions, driving within the lanes, stopping at intersections)

Apart from that, it is assumed that the Duckiebot used for GOTO-1 fulfills the basic requirements for driving in Duckietown. In particular:

- a well calibrated Duckiebot, i.e. using wheel calibration and camera calibration
- a well set-up laptop (preferably Ubuntu), further explained here
- a well established connection between Duckiebot and (any) desktop

3. The GOTO-1 Project

3.1 Pipeline Description

As mentioned in the previous section, the global localization problem can be divided into three subtasks, which is reflected in the pipeline structure of GOTO-1. The driving input for localization in GOTO-1 are the intersection AT's, as these provide a reliable (a certain AT will never return another AT id) and robust way to locate the Duckiebot in the city. This approach firstly originated from the need to have a certain landmark - mapped within a predefined map - that could serve as a localization tool. These then become - by extension - also the nodes for the graph that is used for the subsequent path planning (using Dijkstra). Other algorithms and more information on graph theory can be found here. This is especially the case since the predefined map would already require to have all AT's mapped in order to allow localization from all possible starting points within the map. Also, using the AT's as nodes satisfies the constraint of not making U-turns within Duckietown. As a result, at each intersection an AT is read out and benchmarked/validated against the generated sequence of nodes composing the shortest path, for robustness (*). The in parallel generated sequence of turn commands then publishes the turn command to the *unicorn intersection* node of *indefinite navigation*, which is responsible for the execution of intersection navigation and control. Once the final AT is encountered, the final turn command is passed and a state feedback loop takes over to go the last mile to the desired arrival point B. It passes a stop command once a desired distance from the last intersection (the arrival point) is met.

(*) **Note**: the red stop lines at intersections could provide the trigger function for turn commands as well, but these proved to be less reliable during testing within the *indefinite navigation* framework.

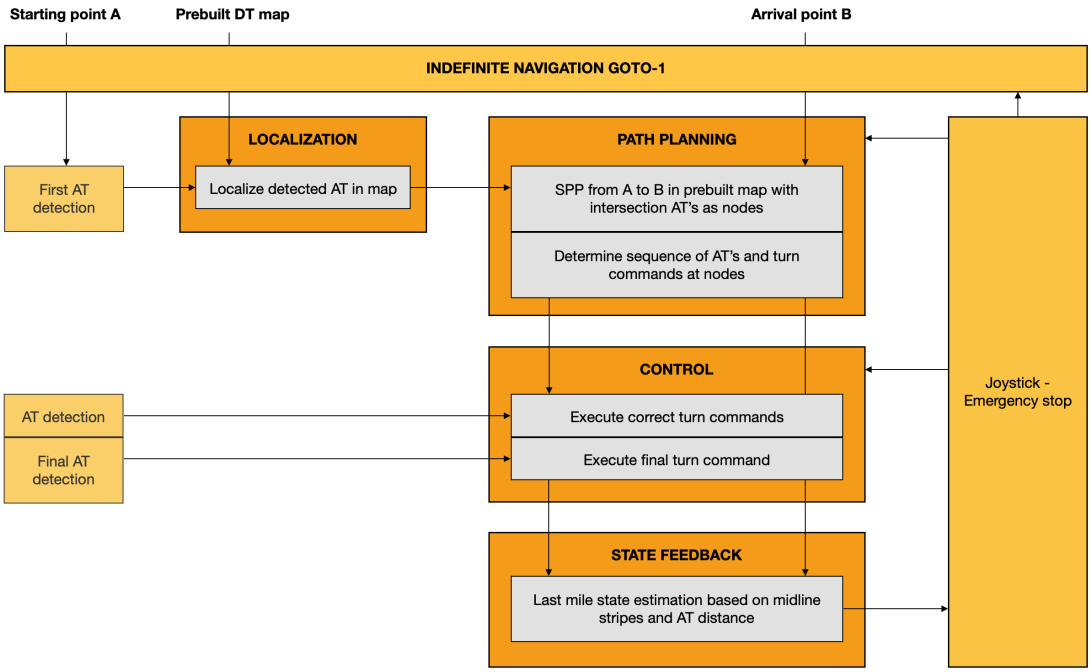


Fig.2 Overall GOTO-1 pipeline: Unmarked inputs are given or self-determined, yellow blocks are running in the existing framework of *indefinite navigation*, and orange-marked items represent the custom blocks developed for GOTO-1

The overall pipeline will require inputs for defining the arrival point B, as well as the map representation of Duckietown as a Dijkstra graph. More specifically, the following inputs are passed to GOTO-1, and yield the following outputs further handled by the *indefinite navigation* framework:

Inputs:

- **goal_input:** AT (id) defining the final lane (orientation)
- **goal_distance:** distance (in cm) between last intersection and arrival point B
- **map weights:** matrix representation of the stage costs between each node of the predefined Duckietown map configuration
- **map policies:** matrix representation of the allowable turn commands the Duckiebot can take at a certain node in the predefined Duckietown map configuration

Outputs:

- **turn command:** desired turn command to the *unicorn intersection* node of the *indefinite navigation* framework in order to follow the generated shortest path to the destination point
- **stop command:** command to stop upon arrival or in case of emergency

3.2 System Interfaces:

The GOTO-1 project builds upon the *indefinite navigation* framework, yet requires certain adaptations in order for the implementation to work. In particular, the following packages (explained in further detail in the next section) were altered or built from scratch:

- the **random april tag turns** node within *indefinite navigation* is responsible for interpretation of a detected, postprocessed AT, and transforming that input into a readible turn command that is further sent to the *unicorn intersection* node. It is in other words responsible for navigation of the Duckiebot, which GOTO-1 aims to replace with an optimal path planner. The new node would therefore send the desired turn commands to the *unicorn intersection* node that make the Duckiebot follow the most optimal path. This new navigation approach and turn command publisher is handled by a **global localization** node in GOTO-1.
- the aforementioned replacement of nodes requires the implementation of a shortest path planner - i.e. a Dijkstra algorithm - to generate the shortest path from the current location to the desired arrival point. This is done by calling an external **path planning** class, which is used by the *global localization* node, aforementioned.
- a final node that was designed is the **state estimation** node, which takes the camera image stream from the *camera node* as input to define the current, travelled distance in order for the *global localization* node to gain feedback on the proximity to the final destination point.

In contrast, all other packages included in the *indefinite navigation* demo remain unaltered. As will be discussed later in this report, it might be optimal to replace certain existing modules with the ones from newly developed projects for increased stability and accuracy.

4. Module Logic & Required Packages

As mentioned in the previous section, and in line with the subtasks of the global localization problem, three packages are defined that support the GOTO-1 functionality.

4.1 Global_localization (node):

This node **localizes** the duckiebot and uses an external *path planning* class to generate the shortest path to get from the localized point to a given destination point. It is the main code of GOTO-1 and passes the desired turn commands per intersection, as well as the stop command upon arrival - i.e. it **navigates** the Duckiebot through Duckietown. The driving input of this code are the intersection AT's, serving as nodes for the Dijkstra graph within the *path planning* class outlined next, and as trigger to publish turn commands. Upon arrival, the node publishes a **stop** command and shuts down the GOTO-1 package.

Inputs:

- **detected AT:** AT message incoming from *apriltag postprocessing* node
- **goal_input:** given AT (id) defining the final lane (orientation)
- **goal_distance:** given distance (in cm) between last intersection and arrival point
- **state feedback:** number of midline stripes encountered after last intersection
- **FSM state:** only process an incoming AT if at an intersection

Outputs:

- **turn command:** desired turn command to the *unicorn intersection* node of the *indefinite navigation* framework in order to follow the generated path to the destination
- **state estimation trigger:** trigger command to start state estimation or not
- **stop command:** command to stop upon arrival or in case of emergency

Logic:

When the Duckiebot is started in *lane following* mode as part of *indefinite navigation*, it will stop at an intersection and read out the apriltag associated with the direction the Duckiebot is coming from. An additional filter is implemented such that this node is not activated when the FSM state INTERSECTION_SOMETHING is False, this is done in order to avoid false AT readings during *lane following*. If the desired FSM state is True, the AT id is passed to the path planning class and the position can be determined as the AT id's are represented in the Dijkstra graph. A shortest path towards the given arrival point is generated as a sequence of AT id's and turn commands, and is updated at every next intersection - i.e. when the next AT in the sequence is encountered. When the last AT id in the sequence is encountered, the node triggers the *state estimation* node by publishing the value True. Position feedback (proximity to the destination point) is obtained by subscribing to this node. If the desired position is reached, the *global localization* node publishes a shutdown command to the *joy mapper* node which is running in *indefinite navigation*.

4.2 Path_planning (class):

This class is imported by the *global localization* node and calculates the **shortest path** given an input and output AT id (node) within the predefined DT map. The predefined DT map is hardcoded in this class, and should be adapted to the actual Duckietown configuration used. For the path planning itself, the class uses the Dijkstra algorithm to define the shortest path. This then gives a sequence of AT's (nodes) to define this path, as well as the corresponding turn commands it should execute in order to navigate/perform the shortest path.

Inputs:

- **detected AT:** AT message incoming from *apriltag postprocessing* node
- **Dijkstra graph:** two graphs (being the cost between every two nodes and the turn command between these two nodes) representing the used Duckietown configuration

Outputs:

- **turn command sequence:** desired sequence of turn commands to the *global localization* to post-process/execute the shortest path
- **apriltag sequence:** desired sequence of AT's to be encountered on the shortest path

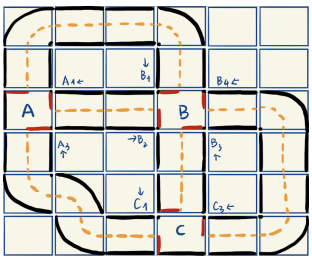
Logic:

The required Dijkstra graph can be defined in a purely deterministic manner by taking the weights between every two AT id's (nodes). The weights are defined according to the equation below, and are based on the distance (number of tiles) and the number of turns between the respective nodes. Note that - since the distance from a certain node to that same node is defined as infinity - the Dijkstra algorithm will not return the policy of a U-turn (which would require a new allowable input definition).

$$C(i,j) = c_n * N(i,j) + c_t * T(i,j) \tag{1}$$

Where: C is the cost from node i to node j, N the number of tiles and T the number of turns between node i and node j. The coefficients c_n and c_t are the individual weights that are given to increase/decrease the relative cost of distance and number of turns.

The allowable policy matrix in its turn is defined using the inputs that can be passed to the *unicorn intersection* node of *indefinite navigation*. In particular, 0 as turning left, 1 as going straight and 2 as turning right. Note that the map used during the demo version of GOTO-1 differs from the one below, and can be found here.



(a) Map definition

AT	8	15	199	231	63	66	59	9	61	11
8	∞	∞	∞	∞	3.7	∞	2.9	∞	∞	∞
15	∞	∞	∞	∞	∞	∞	∞	2.9	2.7	∞
199	∞	∞	∞	∞	∞	∞	∞	6.8	∞	6.7
231	∞	∞	∞	∞	8.4	7.5	∞	∞	∞	∞
63	∞	∞	∞	∞	∞	∞	∞	∞	7.3	8.0
66	∞	2.9	3.0	3.7	∞	∞	∞	∞	∞	∞
59	6.7	6.8	7.5	∞	∞	∞	∞	∞	∞	∞
9	∞	∞	∞	∞	∞	6.4	6.5	∞	∞	∞
61	8.4	9.1	∞	8.3	∞	∞	∞	∞	∞	∞
11	3.7	∞	2.9	3.0	∞	∞	∞	∞	∞	∞

(b) Cost matrix

AT	8	15	199	231	63	66	59	9	61	11
8	∞	∞	∞	∞	0	∞	2	∞	∞	∞
15	∞	∞	∞	∞	∞	∞	∞	2	0	∞
199	∞	∞	∞	∞	∞	∞	∞	1	∞	0
231	∞	∞	∞	∞	1	0	∞	∞	∞	∞
63	∞	∞	∞	∞	∞	∞	∞	∞	1	2
66	∞	2	1	0	∞	∞	∞	∞	∞	∞
59	2	1	0	∞	∞	∞	∞	∞	∞	∞
9	∞	∞	∞	∞	∞	2	1	∞	∞	∞
61	1	0	∞	2	∞	∞	∞	∞	∞	∞
11	0	∞	2	1	∞	∞	∞	∞	∞	∞

(c) Input matrix

Fig.3 Transformation of the Duckietown map configuration into the two input matrices (graph) used in the *path planning* class

4.3 State_Estimation (node):

This node executes the **last mile problem** of the GOTO-1 project by converting the input distance (between the last intersection and arrival point) to a desired, discrete number of midline stripes, and visually counting these until the desired position is reached. The number of stripes is published to the *global localization* node and functions as a state feedback mechanism. Once the desired number of stripes is reached, the nodes shut down and the Duckiebot is stopped.

Inputs:

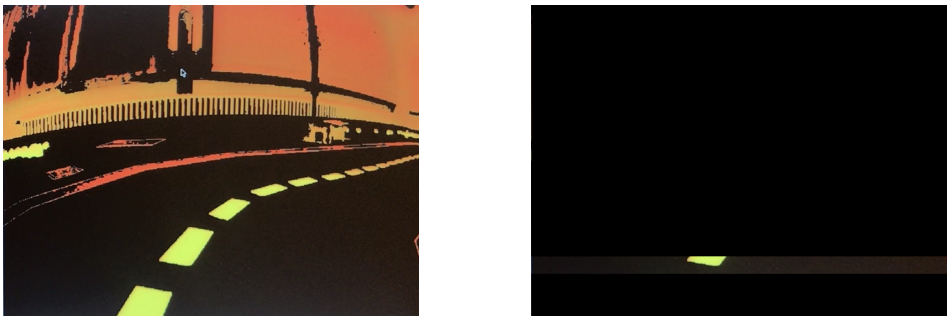
- **camera stream:** images of the lane, post-processed to filter out the midline stripes
- **state estimation trigger:** trigger command to start state estimation or not
- **FSM state:** only enter state estimation once final intersection has been passed

Outputs:

- **state feedback:** number of midline stripes encountered after last intersection

Logic:

This node is activated once the *global localization* node publishes the necessary trigger and once the FSM state is no longer INTERSECTION_SOMETHING. I.e. when it has passed the intersection. From that moment, the number of midline stripes is counted. Counting the number of midline stripes - which adhere to the aforementioned appearance specifications of Duckietown - is done by filtering out the yellow (HSV) colour range of the stripes, and cropping the incoming images to only a few pixels high, horizontal line of the image. The cropped image is then summed over the HSV values, where 0 equals to a fully black image, and non-zero would equal to a yellow stripe in the image. Then, whenever a sequence of a yellow - black image is encountered, a discontinuity or stripe is added to the number of stripes encountered. This number is continuously published to the *global localization* node in order to stop the Duckiebot once the desired number of stripes have been passed. The *global localization* node and the *state estimation* node are both shutdown, and a new global localization trial can be run by restarting the *global localization* node from the terminal with (new) input variables.



(a) Yellow mask (b) Cropped mask

Fig.4 Cropping the mask and summing over a limited pixel height of the image allows GOTO-1 to track discontinuities of the midline stripes and calculate the number of stripes encountered

Note: in the current code there is a set-up to trigger two types of state estimation. A feedback-based one, where the number of midline stripes are count, and a feedforward-based one where the linear velocity and distance to the final destination are used to decide after how much time the Duckiebot should be stopped and therefore has reached the final destination. However, only the first version is currently implemented (closed loop approach).

5. Demo and Trial Run

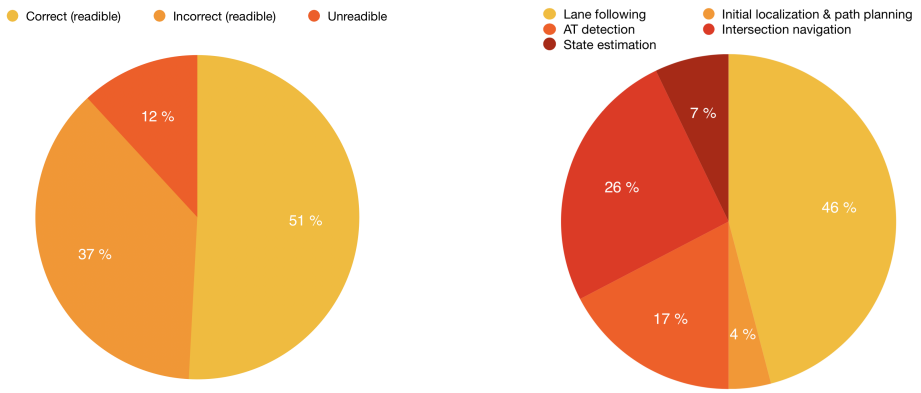
A full description of running the GOTO-1 package, implemented in the *indefinite navigation* framework can be found here. The full GOTO-1 code, troubleshooting actions and a summary README-version of this report can be found here as well. An example log output of the GOTO-1 system can be found in Appendix. A successful run of GOTO-1 can be viewed here (localization, path planning and navigation) and here (navigation and state estimation with a lowered linear velocity).

6. Performance Evaluation

Quantifying the performance metrics is done over three implementations (building the GOTO-1 image from zero, explained in the previous section) of the GOTO-1 package, and running ten attempts per implementation. This was done, in order to mitigate any short-term effects and ratify the behaviour of GOTO-1 over a longer period. In fact, different success rates were seen when building the same images over a single week. Possible causes for these fluctuations might lie in the newly pulled images of *dt-core*, *dt-car-interface* and *dt-duckiebot-interface*, and in external factors such as lighting, AT positioning or network load.

As a result, extensive use of logging services was used during testing. This also allowed us to ensure that each individual module worked in a stable manner before adding the next functionality to GOTO-1. However, to go through the entire pipeline of GOTO-1 was only possible when sufficient time was spent on finetuning the trim values for intersection navigation and lane following. During tuning, one should be aware that during testing, certain values only work on a 1-per-1 basis, and were thus not converging to an uniform, optimal value. The GOTO-1 package allows to pass these tuning values during runtime in two ways: upon starting the node (in which case the passed values are only used at the desired moment in global localization) or from the root line of another Docker container using the *rosparam set* command.

From these trials, it became clear that the interdependencies within *indefinite navigation* caused the majority of unsuccessful (full pipeline) runs. I.e. the AT detection, lane following and intersection navigation caused a lot of undefined behaviour during the trials. Examples of these can respectively be found here (where the Duckiebot is supposed to go straight, but reads out an incorrectly oriented AT), here (where the Duckiebot has trouble following the curved lane) and here (where the Duckiebot has trouble with taking turns). The outcomes of respectively the first performance metric and the efficiency (time allocated) of GOTO-1 shown next are thus on a per-module basis, and do not show the success rate of running the entire pipeline.



(a) AT detection correctness

(b) Time allocation

Fig.5 Evaluation of respectively the efficiency of GOTO-1 and the correctly read out AT's

The outcomes of the other performance metrics are as follows:

- 100% success rate of defining the shortest path between two nodes, given a correctly read out AT,
- although the Duckiebot is stopping when the state estimation (virtually) reaches the final destination, reaching the actual destination is inaccurate. It is possible for the Duckiebot to stop near the destination point, but its precision is insufficient as parameter tuning was not done extensively due to time constraints and at this moment did not yet yield a robust, stable behaviour.

7. Future Improvements

As for any project, there are certain aspects of the GOTO-1 package and the involved framework of *indefinite navigation* that can be improved. In especially, the following submodules could benefit from the following proposed resolutions:

- the *apriltag detection* could make use of the AT pose in order to filter out only the correctly oriented AT's (as AT's parallel to the line of sight currently can be favoured over the ones that are perpendicular to the line of sight) to increase robustness,
- the *state estimation* module could be improved by increasing the rate of analyzed frames, and lowering the upper bound for linear velocity for increased accuracy (note that the latter also requires to finetune the other *lane following* parameters),
- the *intersection navigation* parameters are currently passed as a feedforward command, and are not tailored to a specific Duckiebot (developing a better feedback-based intersection navigation is currently part of another project),

- the *lane control* parameters are currently very unstable, and are again not tailored to a specific Duckiebot (developing a better adaptive lane control module is currently part of another project as well). During the development of GOTO-1, it was opted to use an updated version of dt-core, with the improved modules from the adaptive lane control project implemented and the *random apriltag turns* node deactivated (further information is given in the GOTO-1 repository) for increased efficiency and robustness. The approach would not require building dt-core from scratch, but would instead build upon it. Note that this particular approach has not yet been tested, but was successfully implemented by the adaptive lane control project for *lane following*,

- implementation of the duckietown-world package as basis for the Dijkstra graph, in order to switch more dynamically between various Duckietown configurations for increased robustness and uniformity. For that, a mapping from the position of the AT's in the dt-world format to the format of the Dijkstra graph in the *path planning* class should be build. Integrating this feature would mean that the global localization package becomes more efficient and less expensive in terms of effort than other localization packages that require an entire watchtower system set-up for larger Duckietown configurations.

Appendix

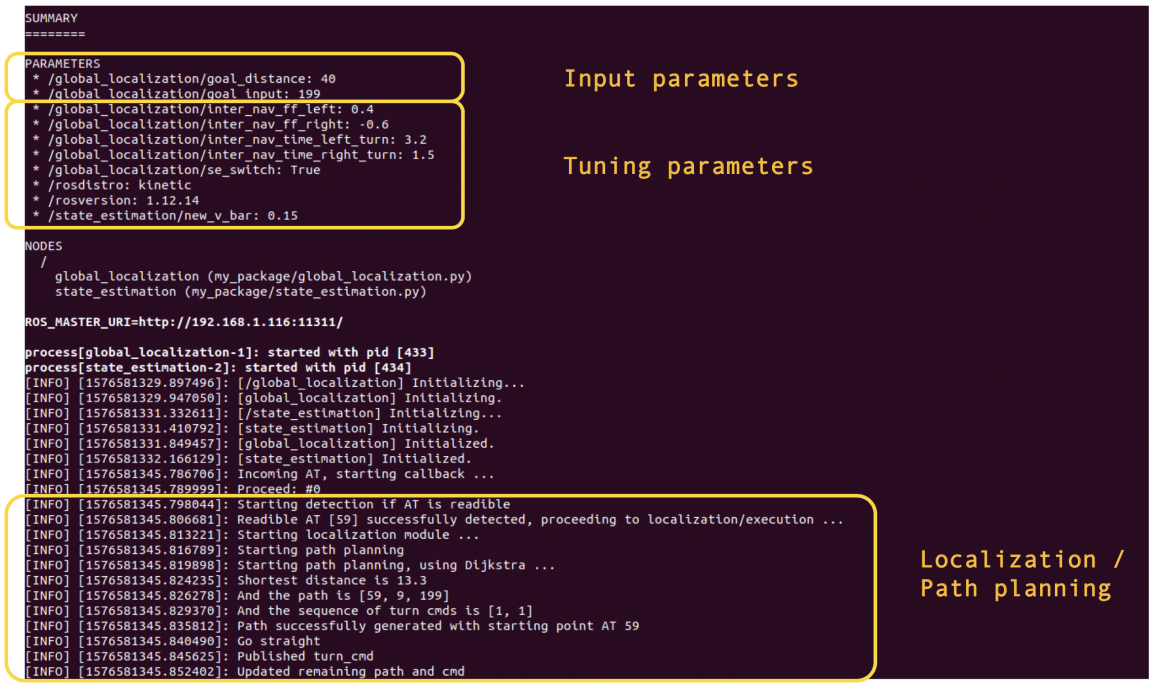


Fig.6 Initialization of GOTO-1 from the terminal with a listing of all input and tuning parameters. The localization and path planning action are also logged as executed after lane following was started (from another terminal window)

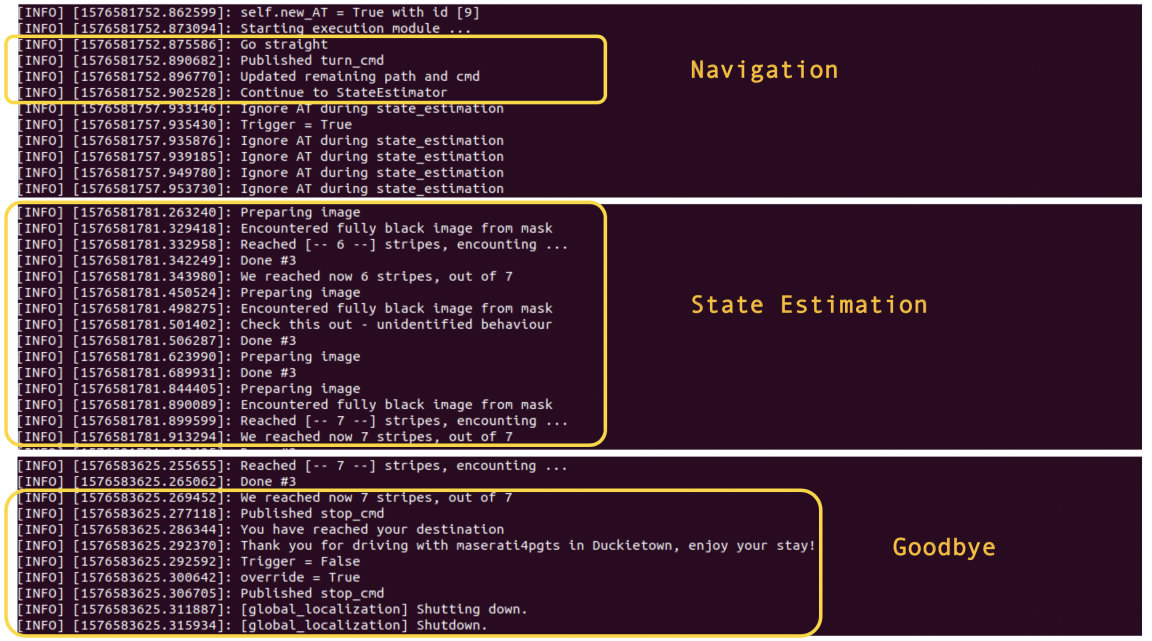


Fig.7 Logging of respectively navigation, state estimation of the last mile problem and the stopping (shutdown) of the Duckiebot upon arrival.