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

Project 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.

Global Localization

Mission & Scope

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

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.

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 localization node, path planning script and state feedback node. Yet, the entire localization, planning, navigation and stopping system is built upon the existing framework of *indefinite navigation* (daffy version). The latter thereby provides in 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, found here. Due to the reduced scope and manpower of this project, the duckietown-world functionality has not (yet) been integrated with GOTO-1.

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.

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 daffy version of the Duckietown environment on Github. Of particular interest are the appearance specifications of Duckietown, and the *indefinite navigation* demo.

Problem Definition

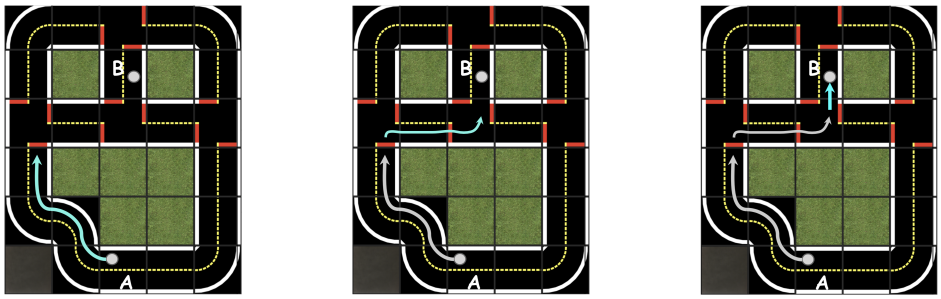
In order to grasp the complexity and approach of the GOTO-1 project, the problem definition and subtasks of this project are outlined.

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 by following the shortest path towards a final destination point.

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 (b) Navigation (c) Arrival

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

Performance and Evaluation Metrics

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

- The Duckiebot is able to localize itself 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. The final criteria was (initially) defined 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)

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 AT's 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

The GOTO-1 Project

Pipeline Description

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. In especially 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 (*). The in parallel generated sequence of turn commands then publishes the turn command to the *unicorn intersection* node, 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 the distance from the last intersection (in number of midline stripes) 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.

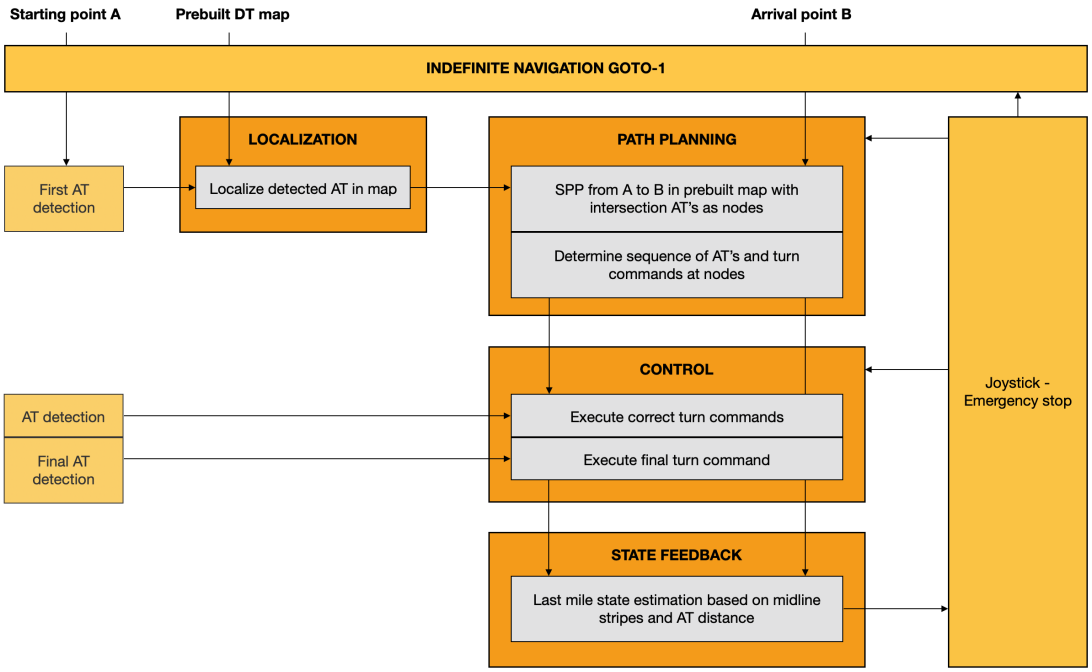


Fig: Overall GOTO-1 pipeline: Unmarked inputs are given or self-determined, yellow blocks are running in the existing framework of *indefinite navigation*, and orange-marked items represent the custom blocks developed for GOTO-1

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

System Interfaces:

Text here ...

Packages & GOTO-1 Logic

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

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** 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. Upon arrival, the node publishes a **stop** command and shuts down the GOTO-1 package.

Path_planning (class):

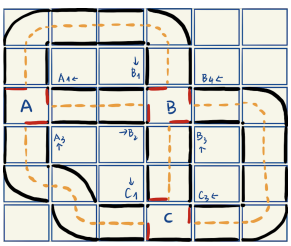
This class is imported by the *global localization* node and calculates the **shortest path** given an input and output AT id (node) within the predefined DT map. The predefined DT map is hardcoded in this class, and should be adapted to the actual Duckietown you want to use. For the path planning itself, the class uses a Dijkstra algorithm to define the shortest path. It 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.

The 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.



(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

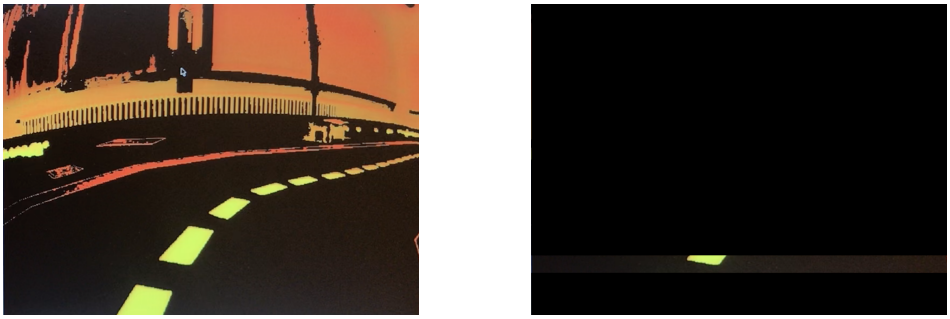
AT	8	15	199	231	63	66	59	9	61	11
8	∞	∞	∞	∞	0	∞	2	∞	∞	∞
15	∞	∞	∞	∞	∞	∞	∞	2	0	∞
199	∞	∞	∞	∞	∞	∞	∞	1	∞	0
231	∞	∞	∞	∞	1	0	∞	∞	∞	∞
63	∞	∞	∞	∞	∞	∞	∞	∞	1	2
66	∞	2	1	0	∞	∞	∞	∞	∞	∞
59	2	1	0	∞	∞	∞	∞	∞	∞	∞
9	∞	∞	∞	∞	∞	2	1	∞	∞	∞
61	1	0	∞	2	∞	∞	∞	∞	∞	∞
11	0	∞	2	1	∞	∞	∞	∞	∞	∞

(c) Input matrix

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

State_Estimation (node):

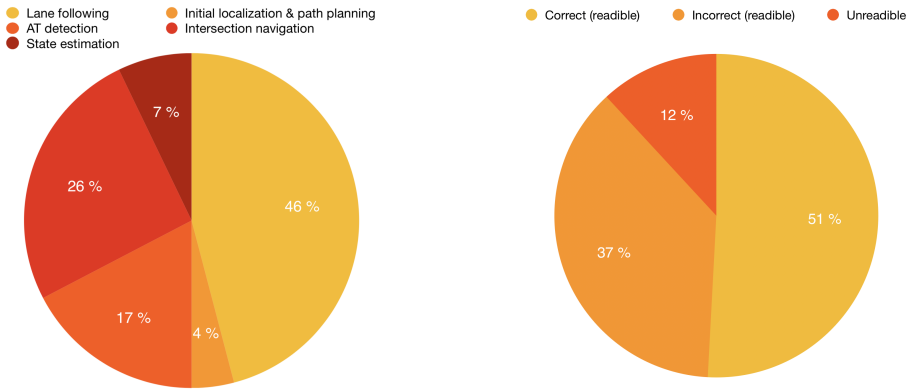
This node executes the last mile problem of the GOTO-1 project by converting the input distance (from a certain AT) to passing a desired number of midline stripes and visually counting these until the desired position is reached.



(a) Yellow mask (b) Cropped mask

Fig: 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

Performance Evaluation



(a) Time allocation (b) AT detection correctness

Fig:

Future Improvements

Appendix