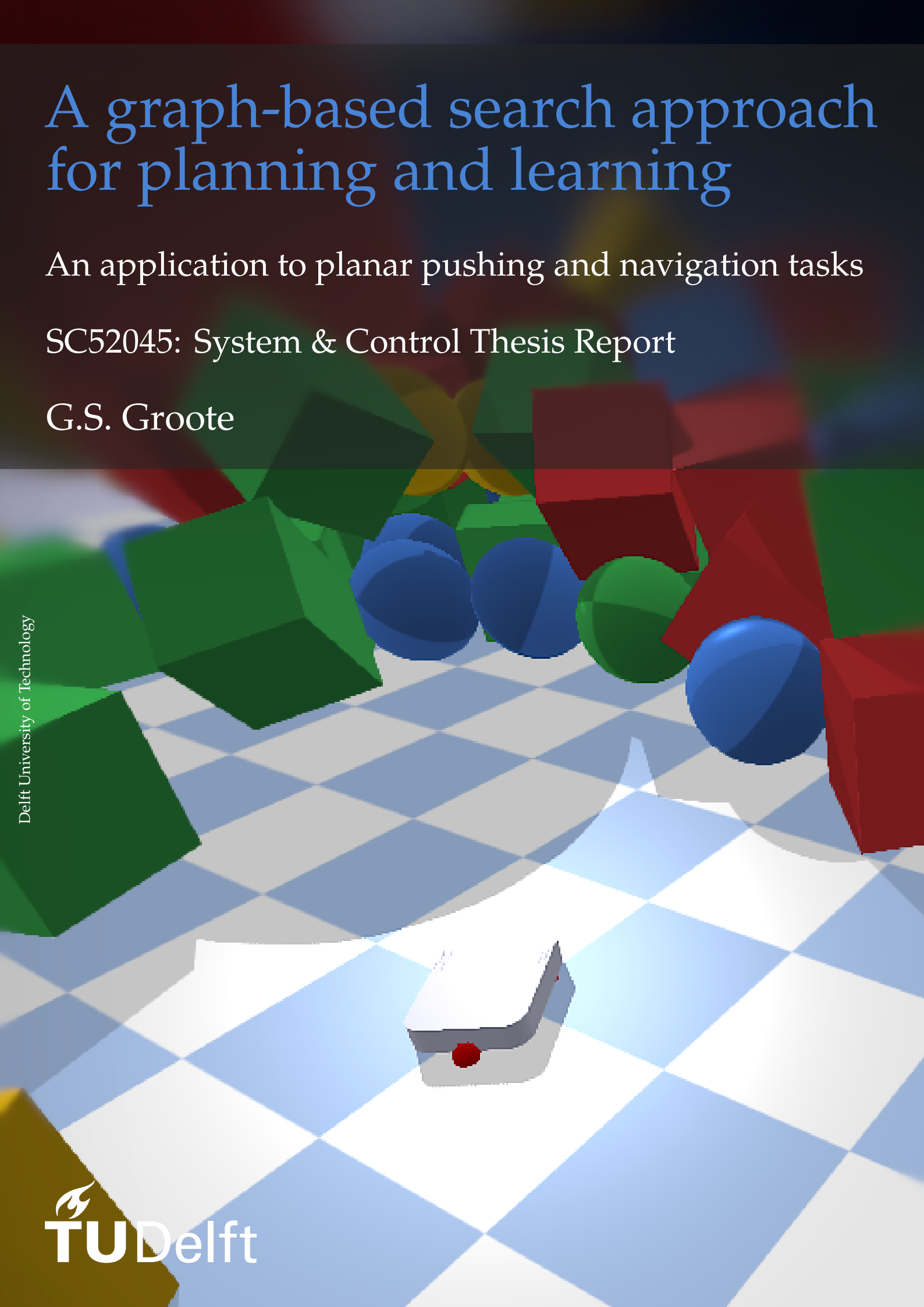
A graph-based search approach for planning and learning

An application to planar pushing and navigation tasks

SC52045: System & Control Thesis Report

G.S. Groote



Delft University of Technology

TU Delft

page intentionally left blank.

A graph-based search approach for planning and learning

An application to planar pushing and navigation tasks

by

G.S. Groote

Student Name	Student Number
Gijs S. Groote	4483987

Supervisors:	C. Smith, M. Wisse
Daily Supervisor:	C. Pezzato
Project Duration:	Nov, 2021 - June, 2023
Faculty:	Faculty of Cognitive Robotics, Delft

Cover: Simulation environment used during the thesis [40].

Style: TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft — Delft Center for Systems and Control — AiR The AI for Retail Lab Delft — Cognitive Robotics

Abstract

In the field of robotics, consider the following problem scenario: In a robot environment a simple robot must push objects to reference places while figuring out which objects can be pushed, what the best manipulation strategy is, or which objects are static, and cannot be pushed. The problem scenario can be decomposed in three research topics which individually have received much attention from the research community; *learning object dynamics* [8, 38], *Navigation Among Movable Objects (NAMO)* [7, 14, 22, 24] and *nonprehensile pushing* [2, 5, 26, 42, 43, 44]. A combination of these three topics could lead to improvements in planning, execution time, and reasoning, but it has not been explored in the literature.

This thesis proposes a robot framework that combines these three research topics. This framework comprises of three key components: the **hypothesis algorithm**, the **hypothesis graph**, and the **knowledge graph**. The hypothesis algorithm computes an hypothesis on how to relocate an object to a new pose by computing possible action sequences given certain robot skills. In doing so, the hypothesis algorithm creates an hypothesis graph that encapsulates the structure of the action sequences and ensures the robot eventually halts. Once an hypothesis is carried out on the robot, information about the execution, such as the outcome, the prediction error the type of controller used and other metrics, are stored in the knowledge graph. The knowledge graph is populated over time, allowing the robot to learn, for instance, object properties and then refine the hypothesis computed to increase task performance, such as success rate and execution time.

A new planning algorithm is proposed that can detect a when a path is blocked by an object, the hypothesis algorithm relies on the newly proposed planner to generate action sequences and to free blocked paths. This planner extends the double tree optimised Rapidly-exploring Random Tree algorithm [7]. The planner constructs a configuration space for an object and is provided with starting- and target pose for that object. The planner then convert these poses to points in configuration space to then search for a path connecting the starting configuration to the target configuration. For the new planner, objects are initially classified as "unknown" and can later be categorized as either "movable" or "unmovable". The object type information is then used when constructing the configuration space for the newly proposed planning algorithm, configuration space consists of the conventional free- and unmovable- (or obstacle) space, and the newly proposed unknown- and movable space.

To carry out the investigation, a mobile robot in a robot environment with movable and unmovable objects is created. The robot is given a task that involves relocating a subset of the objects in the robot environment through driving and nonprehensile pushing. The task can be broken down into individual subtasks that consist of an object and a target pose. Planning for a push or drive action occurs with the newly proposed planning algorithm that, if successful, completes a given task and populates the knowledge graph with learned object information. Information that can be used to determine which objects to manipulate, and what strategy performs best to manipulate a specific object.

By combining these three topics, the proposed framework shows better task execution thanks to experience gained. The proposed framework performs equivalent or better compared to the state-of-the-art frameworks that are specialized in only two out of three research topics [15, 35, 37, 46, 48].

G.S. Groote
Delft, June 2023

Contents

Abstract i

List of Tables iv

List of Figures vii

List of Acronyms vii

List of Symbols viii

1 Introduction 1
- 1.1 Research Question . . . 6
- 1.2 Problem Description . . . 6
  - 1.2.1 Task Specification . . . 7
  - 1.2.2 Assumptions . . . 7
  - 1.2.3 Robot and Objects Examples . . . 8
- 1.3 Report Structure . . . 9

2 Required Background 10
- 2.1 System Identification Setup . . . 10
- 2.2 Control Methods . . . 13
- 2.3 Planning . . . 13
  - 2.3.1 Estimating Path Existence . . . 14
  - 2.3.2 Path Planning . . . 18

3 Proposed Path Planner 24
- 3.1 Planning in Four Subspaces . . . 24
- 3.2 Kinodynamic Planning . . . 27

4 Proposed Robot Framework 28
- 4.1 Overview of the Proposed Framework . . . 28
- 4.2 Hypothesis Graph . . . 29
  - 4.2.1 Definition . . . 29
  - 4.2.2 Edge statusus . . . 31
  - 4.2.3 Legend . . . 32
- 4.3 Hypothesis Algorithm . . . 35
  - 4.3.1 The Search and the Execution Loop . . . 36
  - 4.3.2 Fault Detection . . . 40
  - 4.3.3 The Blocklist . . . 41
  - 4.3.4 Encountering a Blocked Path . . . 42
- 4.4 Knowledge Graph . . . 49
  - 4.4.1 Definition . . . 49
  - 4.4.2 The Testing and Converging Phases . . . 50
  - 4.4.3 Legend and Example . . . 50

5 Results 54
- 5.1 Driving and Pushing Experiments . . . 56
  - 5.1.1 A Driving Task . . . 57
  - 5.1.2 A Pushing Task . . . 61
- 5.2 Comparison with State-of-the-Art . . . 65

6 Conclusions 68

7 Future work 70

References **72**

Appendix **76**

List of Tables

1.1 Overview of state-of-the-art literature with an indication of which topics they incorporate from the three topics; learning object dynamics, the NAMO problem and specifying object target poses. . . . 6

2.1 The planning-related terminology is juxtaposed in the left column alongside its corresponding description in the right column. . . . 14
2.2 The variables and functions employed in the Algorithms 1 to 4. . . . 23

4.1 The proposed-framework-related terminology is juxtaposed in the left column alongside its corresponding description in the right column. . . . 34
4.2 The action edge status is presented in the left column, the corresponding action taken by the *MakeReady* function to prepare an action edge for execution in the right column. An action edge increments its status as indicated in Figure 4.3. . . . 39
4.3 The functions employed by the hypothesis algorithm (halgorithm) in Algorithm 6. . . . 45
4.4 Comprehensive description regarding the actions executed by the blocks in Figure 4.15. 48

5.1 Method metrics employed to measure task performance by the proposed robot framework in the left column. Motivation on relevance of the metric is provided in the corresponding right column. . . . 54
5.2 The left column displays the available models of drive- and push systems, accompanied by a description in the corresponding right column. . . . 55
5.3 The tuning parameters to initialize a random environment . . . 57
5.4 The selected tuning parameters for the randomized drive environment. . . . 58
5.5 The selection of the (Model Predictive Control (MPC), *lti-drive-model*) parameterization versus selecting the (Model Predictive Path Integral (MPPI), *lti-drive-model*) parameterization for drive actions during the randomized driving tasks. The leftmost column indicates that tasks execution is performed with the knowledge graph (kgraph) action suggestions, and without action suggestions. . . . 60
5.6 The selected tuning parameters for the randomized push environment. . . . 61
5.7 The selection of the (MPPI, *nonlinear-push-model-1*) parameterization versus selecting the (MPPI, *nonlinear-push-model-2*) parameterization for push actions during the randomized pushing tasks. The leftmost column indicates that tasks execution is performed with the kgraph action suggestions, and without action suggestions. . . . 64
5.8 Overview of recent state-of-the-art papers that include a subset of the 3 topics (learning system models, NAMO, and nonprehensile pushing). The method metric indicates the testing method used by the paper, where the underlined metric is used to compare against the proposed framework. . . . 66
5.9 Average execute-, search and total times for a task that involves learning object dynamics and the NAMO problem. The results are average over three solved tasks for Wang et al., and ten solved tasks for the proposed framework. A visualization of the task if presented in Figure 5.11. . . . 67

List of Figures

1.1 Two robots in the robot environment 8
1.2 Two objects in the robot environment 9

2.1 A top view of the point robot with five input sequences. The input sequences are executed on the robot and the output responses are recorded, together the in- and output sequences form a Input-Output (IO) data set. 11
2.1 The roobt collecting an IO data sequence by first driving toward the initial pose at the start of an arrow. Then the corresponding input is sent toward the robot, and the robot and the object's output response is collected to form a IO sequence. The IO data set then consists of all six IO data sequences. 12
2.2 Robot environment with the point robot, unmovable yellow walls and movable brown boxes. 16
2.3 The configuration space for the point robot with different cell sizes. The robot environment corresponds to the environment presented in Figure 2.2. 16
2.4 Flowchart displaying both drawbacks when checking path existence. 17
2.5 Snapshot of a two dimensional configuration with free- and unmovable spaces. Where unmovable space is indicated by the unmovable object, the rest of configuration space (including the movable object) is free space. The start- and target nodes, denoted as "Start" and "Target" respectively, serve as the source nodes for the start- and target connectivity tree. 19
2.6 Create new node and connect to parent node. 20
2.7 Connect new node to parent node that results in the lower $Cost_{toInit}$. 21
2.8 Check if the newly added node can lower cost for nearby nodes and connect the start- to the target tree. 22
2.9 Path planner that found a path from start- to target node marked in red. 23

3.1 Driving task and two planned paths. 26

4.1 Flowchart representation of the proposed robot framework. 29
4.2 Finite State Machine displaying the status of an identification edge 31
4.3 Finite State Machine displaying the status of an action edge 32
4.4 Legend for the hypothesis graph. 33
4.5 First stages of the hypothesis graph (hgraph) when the halgorithm searches for an hypothesis to a driving task. 35
4.6 Multiple stages of the hgraph when the halgorithm executes the hypothesis found in figure 4.5. 36
4.7 The search (upper) and execution (lower) loop, that make up the main part of the proposed halgorithm. The figure's goal is to present the two loops, the flowchart in the background is presented full page in Figure 4.15. 37
4.8 First stages of the hgraph when the halgorithm creates an hypothesis for a pushing task. 38
4.9 The hypothesis for a pushing task becomes valid and is executed. Then a path for the pushing edge is planned which generates new nodes to drive toward the best push pose against the box. 39
4.10 The hgraph after the pushing task was successfully completed. 40
4.11 Multiple stages of the hgraph. The two hypotheses both failed during execution because a fault was detected. The failed edges are added to the blocklist, preventing the regeneration of edges with the same parameterization. The halgorithm concludes the task to be unfeasible. 42
4.12 Multiple stages of the hgraph for a driving task, where an blocked path is encountered. 43

4.13 Snapshot of the hgraph during drive task, where a blocked path is encountered. The (red) current node indicates that next action is to drive toward the best push pose against the blocking object. . . . 44
4.14 hgraph final stages before successfully completing a driving task, during which a blocked path is encountered. . . . 44
4.15 Flowchart displaying the halgorithm workflow. . . . 47
4.16 Legend for the knowledge graph. . . . 51
4.17 The kgraph that has collected action feedback on two objects. Two edge parameterizations for driving the robot and one edge parameterization to push the box. . . . 51
4.18 Flowchart displaying the knowledge graph's workflow. . . . 52

5.1 Schematic overview of point robot pushing a box object. . . . 56
5.2 A random environment initialized by tuning parameters presented in Table 5.4 and randomness. . . . 57
5.3 Search-, execution- and total time to complete a drive task **whilst** using kgraph action suggestions. The horizontal axis indicates the number of task experience in a run. The vertical axis displays a boxplot of the search-, execution- and total time over ten runs grouped by number of tasks in experience. . . . 59
5.4 Search-, execution- and total time to complete a drive task by randomly selecting edge parameterizations. The horizontal axis indicates the number of task experience in a run. The vertical axis displays a boxplot of the search-, execution- and total time over ten runs grouped by number of tasks in experience. . . . 59
5.5 Comparing average total time to complete a task out of then runs for a driving task. For the edge parameterizations once with the use of kgraph action suggestions are leveraged indicated by "with kgraph" and once with random selection indicated by "no kgraph". 60
5.6 Two random environments where the pushing task contains a single subtask displayed by a target ghost pose. . . . 62
5.7 Search-, execution- and total time to complete a pushing task **with** kgraph action suggestions. The horizontal axis indicates the number of task experience in a run. A run contains ten tasks, starting the run with an empty kgraph that collects action feedback as the robot gains experience. The vertical axis displays a boxplot of the search-, execution- and total time over ten runs, where the sum of search- and execution time equals total time. 63
5.8 Search-, execution- and total time to complete a pushing task **without** kgraph action suggestions. The horizontal axis indicates the number of task experience in a run. A run contains ten tasks, starting the run with an empty kgraph that collects action feedback as the robot gains experience. The vertical axis displays a boxplot of the search-, execution- and total time over ten runs, where the sum of search- and execution time equals total time. 63
5.9 Comparing average total time to complete a task out of then runs for a pushing task. For the edge parameterizations once with the use of kgraph action suggestions are leveraged indicated by "with kgraph" and once with random selection indicated by "no kgraph". 64
5.10 Box plot of the Prediction Error for solving reshuffled tasks. The horizontal axis indicates the number of task experience in a run. A run contains six tasks, starting the run with an empty kgraph that collects action feedback as the robot gains experience. The vertical axis displays a boxplot of Prediction Error over six runs. For the edge parameterizations once with the use of kgraph action suggestions are leveraged indicated by "with kgraph" and once with random selection indicated by "no kgraph". . . . 65
5.11 Two similar environments and tasks, the robots are tasked to drive toward a target pose indicated with the green ghost poses. In both environment a direct path is blocked by the red box. . . . 67

1 The blockade environment where the robot is tasked to push the blue box toward the target pose indicated with the target ghost pose. A direct path is blocked by the movable light blue cylinder object and the unmovable yellow walls. . . . 78
2 The swap environment where the robot is tasked with swapping the poses of the cylinder and the box object. . . . 79

3 The surrounded environment where the robot is tasked with escaping the surrounding enclosure by driving to the target ghost pose displayed on the right side in the figure. Every box objects is unmovable except the red box which is movable. 79
4 System $G(t)$ with input $u(t)$, output $y(t)$ and MPC controller with input $y(t)$ reference signal $y_{ref}(t)$, parameterisation p and constraint sets $\mathbb{X}$, $\mathbb{U}$, $\mathbb{Y}$ 83
5 MPPI controlled race car using a control horizon of 3 time steps, with 3 rollouts all having their respected inputs as $u_{i,j}$ where i is the rollout index and j indicates the time step [27]. 84

List of Acronyms

DOF Degrees Of Freedom 4
FSM Finite State Machine 31
halgorithm hypothesis algorithm iv
hgraph hypothesis graph v
IO Input-Output v
kgraph knowledge graph iv
LTI Linear Time-Invariant 55
MDP Markov Decision Process 5
MPC Model Predictive Control iv
MPPI Model Predictive Path Integral iv
NAMO Navigation Among Movable Objects i
NP non-deterministic polynomial-time 77
NP-hard non-deterministic polynomial-time hard 3
PE Prediction Error 41
PRM Probabilistic Road Map 3
PRM* optimised Probabilistic Road Map 3
RRT Rapidly-exploring Random Tree 3
RRT* optimised Rapidly-exploring Random Tree 3
TE Tracking Error 41

List of Symbols

$\mathbb{R}$	Set of Real numbers
$\mathbb{Z}_{\geq 0}$	Set of non-negative integers
n	Number of Degrees Of Freedom
obj	Object in the robot environment
Obj	Set of objects
m	Number of objects in the environment
Origin	Origin of the environment with with x (north to south), y (west to east) and z (down to up) axis
Ground Plane	The robot environment ground plane
Eq	Set of motion equations in the robot environment
k	time step index
ϵ^{pred}	Prediction Error
ϵ^{track}	Tracking Error
C-space	Configuration Space, Dim(C-space) $\in \mathbb{R}^2 \vee \mathbb{R}^3$
c	Configuration, point in configuration space
$\hat{c}$	Estimated configuration
x	distance to Origin over the x-axis
y	distance to Origin over the y-axis
θ	angular distance toward the the Origin's positive x-axis around the z axis
x_{grid}	length of grid in direction of x-axis
y_{grid}	length of grid in direction of y-axis
s_{cell}	length and width of a cell
v	Node in motion or manipulation planner
V_{MP}	Set of nodes for motion or manipulation planner
E_{MP}	Set of edges for motion or manipulation planner
P	Set of paths
S	Task, Tuple of objects and corresponding target configurations.
s	subtask, tuple of an object and an target configuration
h	hypothesis, Sequence of successive edges in the hypothesis graph, an idea to put a object at it's target configuration
$G^{hypothesis}$	hypothesis graph
$G^{knowledge}$	knowledge graph

v	Node in hypothesis or knowledge graph
V_H	Set of nodes for hgraph
V_K	Set of nodes for kgraph
e	Edge in hypothesis graph or knowledge graph
E_H	Set of edges for hgraph
E_K	Set of edges for kgraph
NODE_STATUS	Status of a node
EDGE_STATUS	Status of a edge
OBJ_CLASS	classification of an object
ob	observation from the robot environment
α	Success factor for an edge in kgraph
Δk	Time step [sec]

1

Introduction

This chapter narrows the broad field of robotics down to a largely unsolved problem, combining the three topics; learning object dynamics, NAMO and nonprehensile pushing. For that problem, state-of-the-art methods are presented, and their shortcomings are highlighted in the upcoming paragraphs. Then the gap in the literature regarding the combination of the aforementioned topics is summarized in Section 1.1 in the form of the main and sub-research questions. The combination of the three topics is then narrowed down to the scope of this thesis in the problem description, Section 1.2. The chapter finishes by presenting all upcoming chapters in the report structure, Section 1.3.

For robots, navigating and acting in new, unseen environments remains a complicated problem. From the emerged challenges robots face in such environments, three topics are selected, namely: **learning object dynamics** [16], **Navigation Among Movable Objects (NAMO)** [41], and **nonprehensile pushing** [43]. The main goal of this thesis is to combine these three topics. A secondary goal is to investigate how these topics can strengthen each other over time. When investigating the influence of the topics on each other, questions arise, such as: How do learned objects' system models improve global task planning for a robot with nonprehensile push manipulation abilities over time? How to combine learning and planning for push and drive applications? To what extend is the combination of the three topics; learning object dynamics, the NAMO problem and nonprehensile pushing influenced by environmental experience? The three topics are now shortly described.

Learning object dynamics enables the robot to manipulate unforeseen objects, NAMO allows the robot to move around in an environment even if an object is blocking the robot's target location, and nonprehensile pushing allows the robot to change the environment. Combining these three topics covers any task that involves relocating objects by pushing in unforeseen robot environments, such as exploration missions or clearing debris on construction sites. An unfamiliar environment may emerge by changing a familiar environment, such as a supermarket with the presence of people. Learning abilities are crucial when the robot can encounter many different objects.
Navigation Among Movable Objects can be described as; the robot having to navigate to a goal pose in an unknown environment that consists of static and movable objects. The robot may move objects if the goal can not be reached otherwise or if moving the object may significantly shorten the path to the goal [17].
Nonprehensile pushing is a form of manipulation widely available for robots, even if they are not intentionally designed for pushing. Mobile robots can drive (and thus push) against objects, and a robot arm with a gripper can push against objects even if the gripper is already full. Many robots can push, a manipulation action many robots should leverage.
The NAMO problem and nonprehensile pushing are overlapping topics because both involve object manipulation. However, a given task to relocate objects in an environment consisting of movable and unmovable objects do not fit in either of the two topics individually. Relocating objects fits the combination of the NAMO topic and the nonprehensile pushing topic. It is conceivable that a robotic system possessing the capability to both navigate through an environment and manipulate objects holds greater utility in contrast to a robot confined solely to navigation within the robot environment.

Research into approaches tackling the three topics just described can be split into two categories. The bulk falls into the category of hierarchical approaches [15, 23, 37, 46, 48]. The remainder falls in category locally optimal approaches [28, 34, 35]. Both approaches are elaborated upon later in this chapter. First, the configuration space that builds up to the composite configuration space [46] is discussed. Then secondly, two challenges are highlighted that are related to the composite configuration space growth of dimension and the fact that the composite configuration space is piecewise-analytic.

Configuration Space Planning for a single action and its relation to the configuration space is now discussed. Finding a path for a single action (such as robot driving or robot pushing) is known as a *motion-* or *manipulation planning problem* and is planned in configuration space. *Configuration space* for an object *obj* can be described as an n-dimensional space, where n is the number of degrees of freedom for that single object *obj*. A point in this n-dimensional configuration space fully describes where that object is in the workspace. Then the workspace obstacles are mapped to configuration space to indicate for which configurations the object *obj* is in collision with an obstacle in the workspace. The subset of configurations in configuration space for which *obj* is in a collision is called *obstacle space*. The remainder of obstacle space subtracted from configuration space is free space, in which the object can move freely. For every object in the environment, a configuration space can be constructed. A mathematical configuration space description can be found in Section 2.3.1. When a configuration space is constructed, dedicated path planners query the configuration space to determine if a configuration lies in free or obstacle space.

Composite Configuration Space Planning for an action sequence and its relation to the composite configuration space is now investigated. For a robot environment that involves relocating objects among the presence of other movable objects, planning for a single action is not enough because manipulating an object directly to a target pose is often unfeasible. For example, manipulating an object obj_A to a target pose is feasible if the object obj_B that blocks that path is first removed. Clearly, removing blocking object obj_B influences the feasibility of manipulating obj_A to its target pose. For planning, there must be a connection between the configuration space of obj_A and obj_B, where the composite configuration space emerges. A *composite configuration space* or composite space emerges when an object's configuration space is augmented with the configuration space of other objects [45, 46]. A composite configuration in composite space fully describes where the robot and objects are in the workspace. In recent literature, the composite configuration space is also named room configuration [35], joint configuration space [46] or finding "bridges" between configuration spaces [18]. The term composite configuration space has been adopted because it indicates multiple configuration spaces composed together. Path planners (in contrast to the planning in configuration space) have great difficulty in connecting a starting composite configuration to a target composite configuration [33] for reasons that are discussed in the upcoming paragraph.

Challenges Three challenges are now listed, two related to planning in composite space and one due to unknown environments. The first challenge is that the composite configuration space grows exponentially with the number of movable objects in the environment. The dimension of a robot environment's composite configuration space can be written as:

$$m^{env}_{composite} = \sum_{obj \in Obj|_{\text{OBJ\_CLASS}=movable}} m^{obj}_{configuration}$$

Thus the dimension of the composite space grows linearly with the number of objects in the robot environment, and the composite configuration space itself grows exponentially, also known as the *curse of dimensionality*.

The second challenge is due to constraint sets. In configuration space, a single constraint set applies, where constarints originate from kinematic-, dynamic- and/or geometrical properties of the robot and the environments. For example, the boxer robot cannot drive sideways because of the constraints that

orignate from the non-holonomic properties, visualise the boxer robot in Figure 1.1b. The non-holonomic constraints must be respected during planning for the path planner to yield feasible paths. Only a single constraint set exists in configuration space, also called *mode of dynamics* [18], because configuration space relates to a single action. Dedicated path planners such as Probabilistic Road Map (PRM) [19], optimised Probabilistic Road Map (PRM*), Rapidly-exploring Random Tree (RRT) or optimised Rapidly-exploring Random Tree (RRT*) [20] can find paths in configuration space whilst respecting the constraint set and guarantee asymptotic optimality [20]. In composite space, multiple modes of dynamics exist; the composite configuration must respect the constraints set of the mode of dynamics it resides in. These different modes of dynamics make the composite configuration space *piecewise-analytic*. Path planners have great difficulty crossing the boundary from one mode of dynamics to another mode of dynamics [46].

The appendix contains a section explaining complexity classes which may help understand this paragraph better. Finding an optimal action sequence to a NAMO and nonprehensile pushing problem requires a search in composite space. Due to the challenges described above, such a problem falls in the category of non-deterministic polynomial-time hard (NP-hard) problems, for which we now provide a reduction. If the search for an optimal solution in composite configuration space is simplified by completely removing relocating objects to new poses from the task, a purely NAMO problem is what remains. Suppose the problem is simplified even further by assuming that every object is an unmovable obstacle. In that case, the problem falls in the category of NP-hard problems because a reduction exists from the piano mover's problem, which is known to be NP-hard [31]. That a simplified version is NP-hard indicates the difficulty of finding an optimal path in composite configuration space.

The last challenge introduced is the uncertainty of actions in unknown environments. Planning an action sequence with limited or no environmental knowledge inevitably leads to unfeasible action sequences, such as pushing unmovable obstacles. Updating the environmental knowledge and replanning the action sequence is the cure to the uncertainty introduced by a lack of environmental knowledge. Trying to complete unfeasible action sequences is time and resources lost. Additionally, it can lead to the task itself becoming unfeasible. For example, a pushing robot pushes an object into a dead end due to an action sequence planned with limited environmental knowledge. Now that the object is stuck, the task has become unfeasible.

The search for an *optimal* solution in composite space is unthinkable; no recent literature actually plans directly into composite space. Significant simplifications are applied to a search in the composite space, which will be discussed in the next paragraph. Finding paths for multiple actions without opting for an optimal solution, also known as multi-model planning [18], can be achieved using one of the two methods. The first method connects multiple configuration spaces, *hierarchical approaches*, and the second method introduces simplifications to the composite configuration space that allow finding a path, *locally optimal approaches*.

To summarize, the main challenge is to *find an action sequence for a given task to relocate objects that consist of push and drive actions in new and unforeseen environments*. For two reasons, a search cannot be performed in the composite configuration space because it would require solving an NP-hard problem. First, the composite space grows exponentially with the number of movable objects. Secondly, the composite configuration space piecewise-analytic path planners have great difficulty crossing boundaries from one dynamical mode to another. A multi-modal planning algorithm is sought to robustly find paths in composite configuration space whilst avoiding the emerging challenges with composite space and handling the uncertainty introduced by the lack of environmental knowledge. Several researchers tackled this problem. In the following, we provide a categorization and a summary of the most relevant state-of-the-art methods, advantages and disadvantages.

Locally Optimal Approaches As indicated in the previous paragraph, finding a path in the composite configuration space cannot computationally be found in a reasonable time (orders of magnitude slower than real-time, with no guarantees if no path exists). Only by leveraging simplifications applied to the composite configuration space can a search be performed, such as considering a heavily simplified

probabilistic environment [45], considering a single manipulation action [6], discretization [34] or a heuristic function combined with a time horizon [34]. Such techniques prevent searching in configurations relatively far from the current configuration. Local optimality guarantees can be given, and real-time implementations have been shown.

The most relevant locally optimal approach is presented by Sabbagh Novin et al. She presents an optimal motion planner [34] and applies it to a robot in a hospital environment [28] to later improve upon her work [35]. The optimal motion planner avoids obstacles in the workspace and respects the kinematic and dynamic constraints of a robot arm [34]. Examples of the motion planner are provided using a 3- and 4- Degrees Of Freedom (DOF) planar robot arm. Sampling in the composite configuration space is simplified using discretization (by disjunctive programming [12]) of the composite configuration space and by using a receding horizon. The disjunctive programming concept is applied to convert the continuous problem of path planning into a discrete form. In other words, a continuous path is made equivalent to some points with equal time distances representing the entire path. After discretization, the composite configuration space remains untraceable. Thus a search is performed close to the current configuration by combining a heuristic function with a receding horizon concept. A specially developed heuristic function points *toward* a target configuration. The planner then plans between the current configuration and a point toward the target configuration for a predetermined time horizon. The concept of a receding horizon is used to obtain the optimal path for every time step in the time horizon, but apply only the first term and repeating this process until the end-effector meets the final pose.

The optimal motion planner [34] is then converted toward path planning for a non-holonomic mobile robot with a gripper [28]. With the 3-fingered gripper, the robot can grasp legged objects such as chairs or walkers. The targeted workspace is a hospital where the robot is tasked with handing walkers (or other legged objects) to patients to lower the number of falling patients. The variety of legged objects motivates an object model learning module that learns dynamic parameters from experimental data with legged objects. The dynamic parameters are learned using a Bayesian regression model [37]. An MPC controller then tracks the path and compensates for modelling errors. An essential contribution is that the planner can decide to re-grasp one of the object's legs to improve path tracking.

Real-world experiments show the effectiveness of Novin's locally optimal approach [35]. She has presented a manipulation planning framework focused on moving legged objects in which the robot must choose which leg to push or pull. The framework can operate in real-time, and the local optimality has been shown. From the three topics this thesis focuses on, Sabbagh Novin et al. includes learning object dynamics and prehensile manipulation of objects to target poses, missing only the NAMO problem because a path is assumed to be free during object manipulation. Because Novin uses a gripper to manipulate objects, her research falls into the category of prehensile manipulation. Nonprehensile manipulation bears an additional challenge over prehensile manipulation. With prehensile manipulation, a gripper ensured multiple contact points, geometrically locking the object with respect to the gripper. Until the gripper opens, the gripper and gripped object can be considered a unified object. Nonprehensile manipulation, unlike prehensile manipulation, does not have the advantage of locking the object in place, making it more challenging.

Hierarchical Approaches The second class of approaches to finding a path in composite configuration space is classified as hierarchical approaches [15, 23, 37, 46, 48] that can be described as follows. A hierarchical structure generally consists of a high-level and a low-level component. The high-level task planner has an extended time horizon, including several atomic actions and their sequencing. Whilst a low-level controller acts to complete a single action in a single mode of dynamics, e.g. drive toward the object, push object. The high-level planner has a prediction horizon consisting of an action sequence, a long prediction horizon compared to the low-level planner, whose prediction horizon is, at most, a single action.

The most relevant hierarchical approach is presented by Scholz et al. [37]. He presents a planner for the NAMO problem that can handle environments with under-specified object dynamics. The robot's workspace is split into various regions where the robot can move freely. Such regions can be connected

if an object separates two regions and can be manipulated by the robot to connect both regions. The manipulation action is uncertain because objects have constraints that the robot has to learn, e.g. a table has a leg that only rotates but cannot translate. A Markov Decision Process (MDP) is chosen as a graph-based structure, where the nodes represent a free space region and objects separating the regions are edges in the MDP. Finding a solution for the MDP leads to an action sequence consisting of some drive and object manipulation actions to eventually drive the robot toward a target pose. The under-specified object dynamics introduce uncertainty in object manipulation. During action execution, object constraints are captured with a physics-based reinforcement learning framework that results in improving manipulation planning when replanning is triggered.

Scholz et al. presented a NAMO planner that makes use of a hierarchical MDP combined with a learning framework, resulting in online learning of the under-specified object dynamics. An implementation on a real robot has shown the method's effectiveness in learning and driving toward a target location. From the three topics this thesis focuses on, Scholz et al. includes learning and the NAMO problem, missing only push manipulation toward target locations. By not including manipulating objects to target poses Scholz et al. can find a global path without running into high dimensional spaces. In other words, by driving only the robot toward a target location, a global path will encounter objects only once. By running into objects only once, manipulating an object does not affect the feasibility of the global path, hence the simplification.

Individually a considerable amount of research is done on these three topics (learning object dynamics [8, 38], NAMO [7, 14, 15, 22, 24, 48], nonprehensile pushing [2, 5, 26, 42, 43, 44]). Combining two topics received little attention from the scientific community, and combining all three topics (to the best of my search) not at all. The most relevant work for local optimal [35] and hierarchical [37] approaches are discussed, both having advantages and disadvantages. Local optimal approaches converge to a local optimal plan. To avoid the curse of dimensionality, simplifications must be used to sample the composite configuration space to be computationally feasible. Such simplifications determine the quality of solutions found. Hierarchical structures generally provide computationally efficient solutions but are hierarchical, meaning the solutions found are the best feasible solutions in the task hierarchy they search. The quality of the solution depends on the hierarchy, which is typically hand-coded and domain-specific [46]. Table 1.1 presents state-of-the-art literature and which portion of the three topics they include in their research. Note that both relevant works focus on prehensile manipulation, whilst this thesis focuses on nonprehensile push manipulation. This thesis combines the three topics partly indicated in the table below by ✗/✓ because learning object dynamics is only theoretically included and is not tested properly.

Table 1.1: Overview of state-of-the-art literature with an indication of which topics they incorporate from the three topics; learning object dynamics, the NAMO problem and specifying object target poses.

			NAMO		Specify object target poses	
Author	Citation	Learns object dynamics	prehensile	nonprehesile	prehensile	nonprehesile
Ellis et al.	[15]	✓	✗	✓	✗	✗
Sabbagh Novin et al.	[35]	✓	✓	✗	✓	✗
Scholz et al.	[37]	✓	✓	✗	✗	✗
Vega-Brown and Roy	[46]	✗	✓	✗	✓	✗
Wang et al.	[48]	✓	✗	✓	✗	✗
Groote	Proposed Framework	✗/✓	✗	✓	✗	✓

1.1. Research Question

The following research questions have been selected to investigate the effect of learning on action selection and action planning.

Main research question:

> How do learned objects' system models improve global task planning for a robot with nonprehensile push manipulation abilities over time?

The main research question is split into two smaller, more detailed subquestions.

Research subquestion:

1. How to combine learning and planning for push and drive applications?
2. To what extend is the combination of the three topics; learning object dynamics, the NAMO problem and nonprehensile pushing influenced by environmental experience?
3. How does the proposed framework compare against the state-of-the-art?

This thesis's main contribution is combining all three topics. These topics are learning object dynamics, the NAMO problem and nonprehensile push manipulation. The proposed framework combines these three topics with the *hypothesis algorithm*. The algorithm builds a graph-based structure with nodes and edges, named the *hypothesis graph*. Planning directly in the composite configuration space is avoided due to the inherent complexity associated with planning in composite space. Instead, the hypothesis algorithm plans only in a single mode of dynamics and searches for a global path with a technique known as a backward search [23]. Learned object dynamics are stored in a knowledge base called the *knowledge graph*. The hypothesis algorithm, hypothesis graph and knowledge graph are introduced in Chapter 4.

1.2. Problem Description

To help answer the research questions, tests are performed in a robot environment. A simple environment is desired because that simplifies testing, yet the robot environment should represent many real-world environments in which robots operate. Thus a 3-dimensional environment is selected. The environment consists of a flat ground plane since many mobile robots operate in a workspace with a flat floor, such as

a supermarket, warehouse or distribution center. An environment with a flat floor and a flat robot can be treated as a 2-dimensional problem because the robot and objects can only change position over x and y axis (xy plane parallel to the ground plane) and rotate around the z axis (perpendicular to the ground plane). A flat robot is selected because it has a low center of gravity, which lowers the chance of tipping over.

Let us start with defining the environment. Let the tuple $\langle \text{Origin}, \text{Ground Plane}, Obj, \text{Eq} \rangle$ fully define a robot environment where:

Origin	Static point in the environment with a x-, y- and z-axis. Any point in the environment has a linear and an angular position and velocity with respect to the origin
Ground Plane	A flat plane parallel with the Origin's x- and y- axis. Objects cannot pass through the ground plane and meet sliding friction when sliding over the ground plane.
Obj	A set of objects, $Obj = (ob_1, ob_2, ob_3, \ldots, ob_i)$ with $i \geq 1$, an object is a 3-dimensional body with shape, can be unmovable or movable. In the latter case, the mass is uniformly distributed. The robot itself is considered an object, and an environment thus contains one or more objects. Examples of objects are given in Figure 1.2.
Eq	A set of motion equations describing the behaviour of objects.

A configuration consists of the linear position of an object's center of mass with respect to the environment's origin and the angular position of an object's orientation with respect to the environment's origin.

Formally, a **configuration**, $c_{id}(k)$ is a tuple of $\langle pos_x(k), pos_y(k), pos_\theta(k) \rangle$ where $pos_x, pos_y \in \mathbb{R}$, $pos_\theta \in [0, 2\pi)$
k indicates the time step and can be dropped to simplify the notation, *id* is short for identifier and indicates the object to which this configuration belongs.

1.2.1. Task Specification

To answer the research questions, some tasks are designed, which are defined as a subset of all objects with an associated target configuration.

$$\text{task} = \langle Obj_{task}, C_{targets} \rangle$$

Where $Obj_{task} \subseteq Obj$, $C_{target} = (c_1, c_2, c_3, \ldots c_k)$ and $k > 0$.

A task is completed when the robot manages to push every object to its target configuration within a specified error margin.

1.2.2. Assumptions

Several assumptions are taken to simplify the pushing and learning problem, which are listed below.

Closed-World: *Objects are manipulated, directly or indirectly, only by the robot. Influences from outside the environment cannot manipulate objects.*

Perfect Object Sensor: *The robot has full access to the poses and geometry of all objects in the environment at all times.*

Tasks are Commutative: *Tasks consist of multiple objects with specified target poses. The order in which objects are pushed toward their target pose is commutative.*

3-dimensional robot environment can be represented as 2-dimensional environment *All objects in the environment can be projected onto the ground plane.*

The assumptions taken serve to simplify the problem of task completion. Note that insight is given to remove all assumptions in Chapter 7. By removing assumptions completing tasks becomes a more complicated problem but a more realistic problem closer to real-world applications.

Assumptions might have certain implications, which are listed below. The **closed-world assumption** implies that objects that stand still and do not interact with the robot remain at the same pose. Completed subtasks are therefore assumed to be completed for all times after completion time.

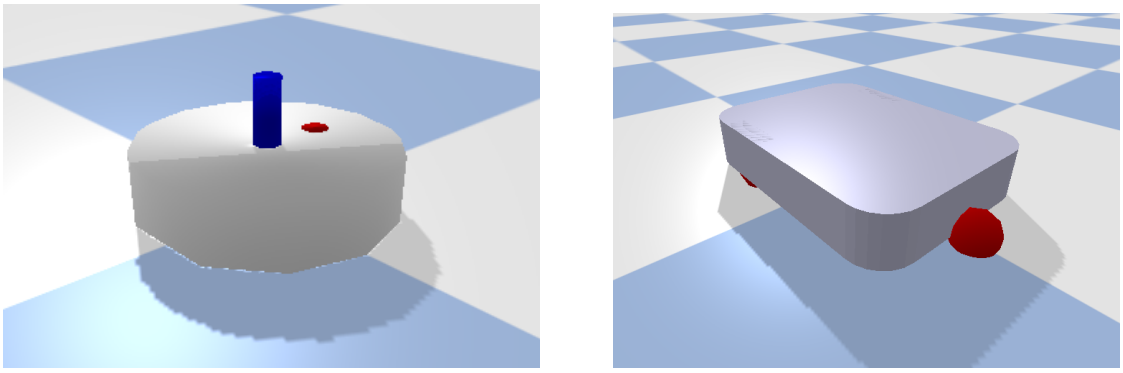
The **perfect object sensor assumption** simplifies a sensor setup, it prevents Lidar-, camera setups and tracking setups with aruco or other motion capture markers. The existence of a single perfect measurement erases the need to combine measurements from multiple sources with sensor fusion algorithms, such as Kalman filtering [47].

Certain tasks are only feasible if performed in a specific order (e.g. the Tower of Hanoi). The **tasks are commutative assumption** allows focusing only on a single subtask since it does not affect the completion or feasibility of other subtasks.

The **3-dimensional world can be represented as 2-dimensional** ensures that the objects in the environment can be projected onto the ground plane. Additionally, it ensures that objects do not tip over. In practice, objects will not be higher than the minimum width of the object. This experimental strategy seemed sufficient whilst running experiments but does not guarantee that objects cannot tip over.

1.2.3. Robot and Objects Examples

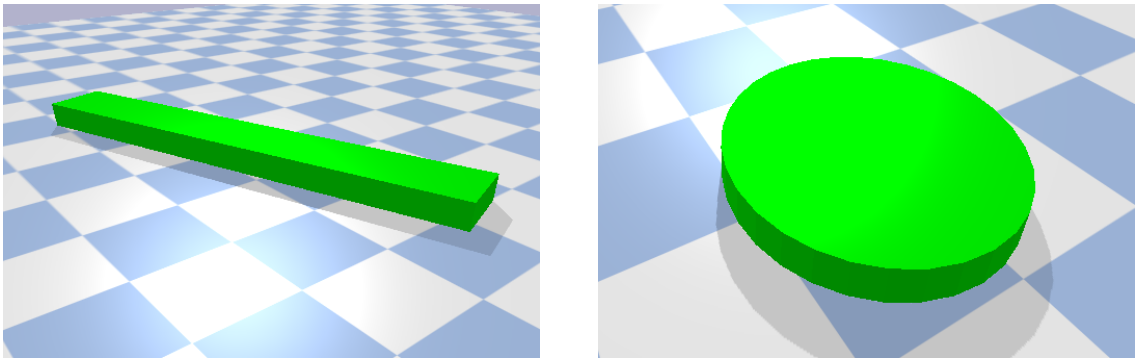
To get a sense of what the robots and the objects look like, see the two robots used during testing in Figure 1.1 as well as two example objects displayed in Figure 1.2.



(a) The holonomic point robot with velocity input in x and in y direction

(b) The non-holonomic boxer robot, the input is forward and rotational velocity

Figure 1.1: Two robots in the robot environment



(a) A box object **(b)** A cylinder object

Figure 1.2: Two objects in the robot environment

For complete environments with accompanying tasks, see Chapter 5.

1.3. Report Structure

The proposed framework heavily relies on a number of methods and functions. This required background is conveniently grouped in Chapter 2. Then an extension is made on an existing planning algorithm in Chapter 3 to be integrated into the proposed framework that is presented and discussed in Chapter 4. Testing the proposed framework is presented in Chapter 5, and allows to draw conclusions in Chapter 6. A list of symbols and a list of acronyms summarize all variables and acronyms used troughout the thesis, by hovering with the mouse over any variable or acronym, a popup presentes more information of the acrynom or variable. The popup also works for references toward figures, citations, chapters, sections and subsections and should prevent much scrolling and jumps to pages. The last two chapters dedicate themselves to the future work and the appendix.

2

Required Background

This chapter presents the components on which the proposed framework relies. These components are system identification, control methods and planning. System identification converts IO data into a system model, Section 2.1 presents a categorization of system models and describes the procedure on how to collect IO data. In the scope of this thesis, a system model is required by control methods to create stable control and track a reference signal, discussed in Section 2.2. Path planning is a core component of the proposed framework and is responsible for finding a path in configuration space. Such a path acts as a reference signal for the controller. Path planning is split into path estimation in Section 2.3.1 and path planning in Section 2.3.2. An existing path planner [48] is extensively discussed, to then expand upon in Chapter 3.

2.1. System Identification Setup

In this thesis understanding how objects behave as a result of input sent to the robot is captured by *system models*. System identification aims to create a system model that best relates an input sequence to an output sequence. Three types of system models are categorized, data-driven-, hybrid- and analytic system models. Data-driven models are generated by only using IO data, and hybrid system models provide a predetermined structure between the input and output with several variables to be determined by IO data, such as the object's dimensions or weight. Analytic models are fully defined and thus do not use any IO data. Note, this section eleborates on how to collect an IO data set, and does not eleborate on system identification methods themselves. System identification methods are relocated to the future work section.

An example of a system model is the drive model for the robot. The model estimates the robot's trajectory when input is sent to the robot. For a specified range of inputs that can be sent to the robot, a system model can predict the possible future states that a robot can reach for a small number of time steps. Using this method, a prediction can be made that estimates if states are reachable from the current state and which states cannot be reached.

To generate a data-driven- or hybrid system model, IO data is required that is collected by sending input to the robot and recording the output response. Now a method for data collection is presented to collect data for drive and push applications.

The IO data set is defined as:

$$\text{IO data set} = \left[(u_1(k), y_1(k)), \ldots, (u_g(k), y_g(k))\right]$$

$$= \left[(u_1(1), \cdots u_1(a), y_1(1), \cdots y_1(a)), \ldots (u_g(1), \cdots u_g(a), y_g(1), \cdots, y_g(b))\right]$$

Where k is the time step, (u_{id}, y_{id}) is a IO sequence with identifier id, m is the number of sequences in the IO data set, a is the number of inputs and outputs in sequence with identifier 1, b is the number of inputs and outputs in sequence with identifier g. With $g, a, b \in \mathbb{Z}_{\geq 0}$

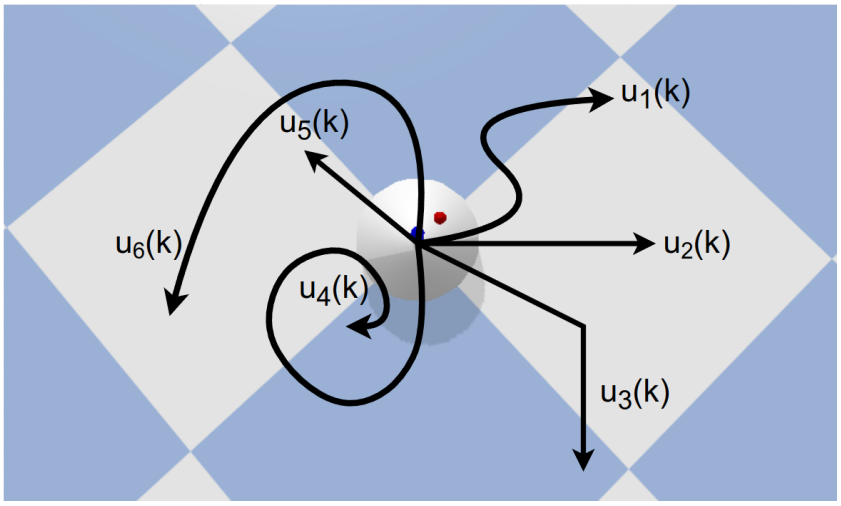
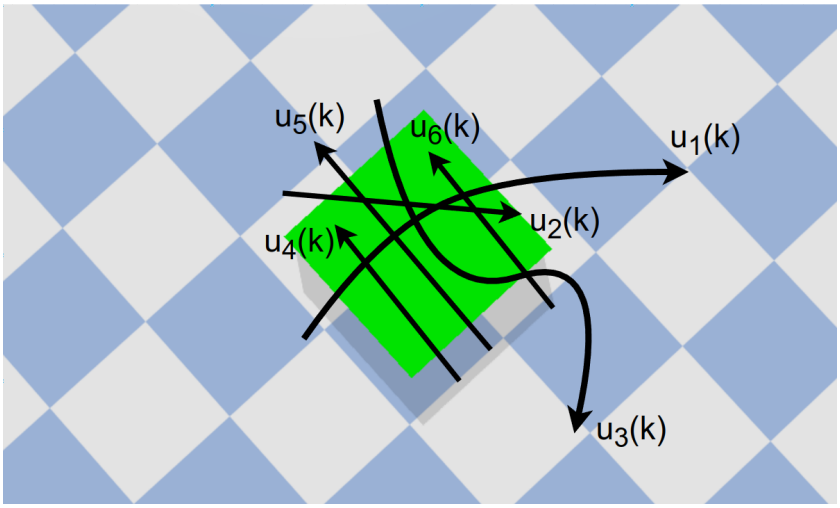
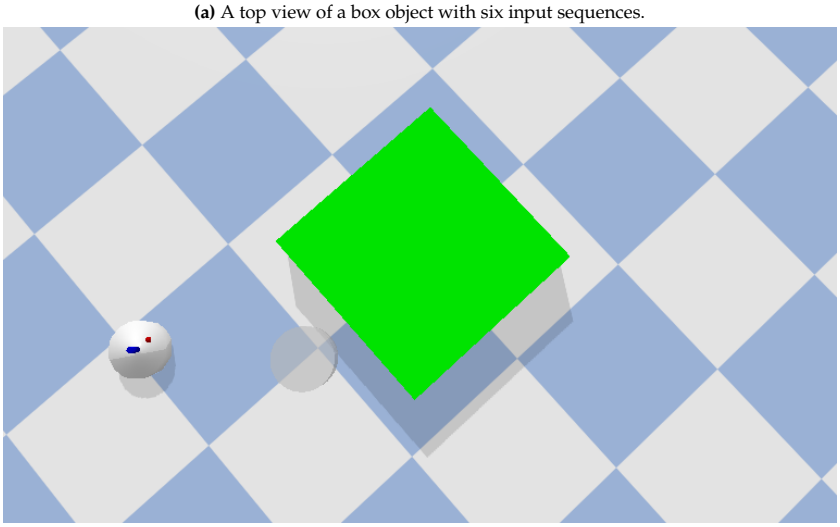


Figure 2.1: A top view of the point robot with five input sequences. The input sequences are executed on the robot and the output responses are recorded, together the in- and output sequences form a IO data set.

The initial pose does not affect a driving model's collection of an IO sequence. At the very least, the robot should not collide with other objects during data collection. Opposite to collecting IO data for a driving model, sending input to generate a IO sequence for the robot-pushing model requires a predefined initial pose. That pose is relative to the object to push, as visualized in the following figure.



(a) A top view of a box object with six input sequences.



(b) The robot driving to toward the initial pose to then apply input sequence $u_1(k)$ and collect output sequence $y_1(k)$.

Figure 2.1: The roobt collecting an IO data sequence by first driving toward the initial pose at the start of an arrow. Then the corresponding input is sent toward the robot, and the robot and the object's output response is collected to form a IO sequence. The IO data set then consists of all six IO data sequences.

A top view of the point robot with five input sequences. The input sequences are executed on the robot and the output responses are recorded, together the in- and output sequences form a IO data set.

The quality of a system model resides in the dynamics it can capture from the system. In order to create models that capture a large portion of the dynamics, system identification methods need an IO data set that captures these dynamics. It must be recorded when the system is persistently exited to create IO sequences that capture the dynamics. Creating input that persistently excites the system is out of the scope of this thesis.

Only a method for data collection is preset, which is fully integrated with the proposed framework. System identification is not implemented or tested; instead, analytic models are used for two time-related reasons. Firstly creating the implementation module itself takes too much time. Secondly, collecting enough IO data to generate a system model is time wise very costly. Thus, time is saved by using several analytic system models instead of implementing a system identification module. The replacement moves to focus from "which system identification method yields a system model that most accurately describes a dynamic model?" to "which system model in the set of available models most accurately describes a dynamic model?". The analytic system models are not opting for modelling the drive or push model as accurately as they possibly can, thus severe model mismatch should be expected.

Related to system identification is *system classification*, a symbolic model that indicates the accordance of objects. In this thesis objects are initially classified as UNKNOWN, a test determines if objects are movable or unmovable and classify them as MOVABLE or UNMOVABLE, indicating that they can or cannot be pushed. That test is presented in Section 4.3.

2.2. Control Methods

This section elaborates on why control is required and which control methods are best suitable for various control applications. Predictive control methods have been selected from many control methods because they heavily rely on a dynamic model of the system they control. During this thesis, the effect of the robot interacting with objects is captured by dynamic system models. In addition to predicting output with system models, predictive control methods use the system models to determine action input. A requirement for a controller is that it should yield a stable closed-loop control because that guarantees converging toward a set point. In the proposed framework, controllers are later selected for yielding desired metrics grouped in Section 4.3.2. The two control methods that are used during testing are discussed below.

Model Predictive Control The basic concept of MPC is to use a dynamic model to forecast system behaviour and optimize the forecast to produce the best decision for the control move at the current time. Models are central to every form of MPC [30]. The best feasible input is found every time step by optimizing the objective function and respecting constraints. Tuning is accomplished by modifying the objective function's weight matrices or the constraint set. Minimizing an objective function to find the best feasible input generally yields robust control. A more elaborate description of MPC control can be found in the appendix.

Model Predictive Path Integral Control The core idea is from the current state of the system using a system model and randomly sampled inputs to simulate several "rollouts" for a specific time horizon, [27]. These rollouts indicate the system's future states if the randomly sampled inputs are applied. The future states can be evaluated by a cost function penalizing undesired states and rewarding desired future states. A weighted sum over all rollouts determines the input which will be applied to the system. The main advantage MPPI has over MPC control is that it is better suited for nonlinear system models. Whilst linear models can accurately estimate drive applications, push applications are harder to estimate with a linear model. Thus MPPI is selected mainly for push applications. A more elaborate description of MPPI control can be found in Chapter 7.

The properties of MPC suggest that it is best suitable for drive actions because of easy tuning and robustness. MPPI control is compatible with nonlinear system models, making it more suitable for push actions. It is worth mentioning that the goal of this thesis is not to find the best optimal controller. The goal is to gradually, over time, choose control methods in combination with system models that result in better performance, and the performance is measured with various metrics, discussed in Chapter 4.

2.3. Planning

This section explains *path planning*, which consists of 2 steps. Firstly, path estimation, and secondly, motion or manipulation planning. The path estimator can detect non-existent paths and concludes such paths as unfeasible [51]. If a path exists, the double tree RRT* planning algorithm is responsible for finding a path from starting point to a target point in configuration space whilst respecting applicable constraints [7].

To clarify terminology used in this thesis, the following Table 2.1 is presented.

Table 2.1: The planning-related terminology is juxtaposed in the left column alongside its corresponding description in the right column.

Global Planning:	Planning for a task by focusing on one single subtask at a time.
Local Planning:	Validating if two closeby configurations that are maximal *step_size* apart can be connected whilst respecting constraints using a system model.
Path Estimation:	Estimating the existence of a path for a push or drive action.
Motion Planning:	Planning a drive action
Manipulation Planning:	Planning for an push action.
Action Planning:	Planning for an drive or push action.
Path Planning:	Path estimation and action planning for an drive or push action.

2.3.1. Estimating Path Existence

In this subsection, motivation and explanation for estimating path existence are presented. We can describe the path estimation algorithm as *The idea is to discretize the configuration space. The emerging cells act as nodes in the graph, cells connect through edges to nearby cells. Graph-based planners start from the cell containing the starting configuration and search for the cell containing the target configuration while avoiding cells in obstacle space.*

If the path estimator concludes the existence of a path planning problem, then it is validated that it is geometrically possible for an object to go from start to target configuration in small successive steps without colliding with an unmovable obstacle. The check prevents the planner from attempting a search for a start and target configuration for a problem that is unfeasible due to not being detected by the path estimator because it does not check for non-holonomic constraints. By neglecting non-holonomic constraints, the path estimator is magnitudes faster than the planner; the planner is responsible for checking the non-holonomic constraints.

Discritizing the Configuration Space For general geometric shapes, a configuration space can be constructed and discretized. During this thesis, configuration space was implemented for cylinders and rectangular prisms. First, the configuration space for a cylindrical-shaped object in a 3-dimensional environment is presented. That configuration space for cylindrical objects is defined as a (x, y)-plane, and the z-axis is omitted. During the projection from a 3-dimensional environment to a 2-dimensional plane, cylinders (flat side facing down) become circles, and rectangular prisms become rectangles. The definition of configuration space is presented below for a circular object, such as the point robot, without loss of generality. In this thesis, three dimensional objects can be projected to the xy-plane. As a result a three dimensional spherical robot or object can has a configuration space that is two dimensional (x and y dimension). Objects in the shape of rectangular prisms that are projected onto the xy-plane become rectangles, and have a three dimensional configuration space (x, y and θ dimension).

A grid of cells represents configuration space. Let s_{cell} be the width and height of a square cell. Let x_{grid} be the vertical (north to south) length and let y_{grid} be the horizontal length (west to east) of the configuration space, point $(0, 0)$ is at the center of the grid.

The two dimensional configuration space is defined as:

$$\text{C-space}^{2D} = \begin{bmatrix} c_{(0,0)} & c_{(0,1)} & \cdots & c_{(0,j_{\max})} \\ c_{(1,0)} & c_{(1,1)} & \cdots & c_{(1,j_{\max})} \\ \vdots & \vdots & \ddots & \vdots \\ c_{(i_{\max},0)} & c_{(i_{\max},1)} & \cdots & c_{(i_{\max},j_{\max})} \end{bmatrix}$$

With $0 \leq i \leq i_{\max} = \frac{x_{grid}}{s_{cell}}, \quad 0 \leq j < j_{\max} = \frac{y_{grid}}{s_{cell}}$.

Where $c_{(i,j)}$ in matrix C-space represents to which subspace the cell at indices (i, j) belongs. The four subspaces considered in this thesis are free-, unmovable-, movable- and unknown space indicated with the integers 0, 1, 2 and 3, respectively. Multiple subspaces can reside in a single cell; by default, a cell represents free space. The subspaces are ordered from least to most important: free, movable, unknown, and unmovable space. A cell displays the subspace with the highest order of importance that resides in the cell. Thus a cell that contains part unknown and part unmovable space will evaluate as unmovable space since unmovable space is more important than the unknown space.

A mapping function $f_{chart\_to\_idx}(i, j)$ maps the Cartesian (x, y) coordinates to their associated (i, j) indices:

$$f_{chart\_to\_idx}(i, j) : \mathbb{R}^2 \mapsto {\mathbb{Z}_{\geq 0}}^2$$

Defined for:

$$x = \{x_{grid} \in \mathbb{R} : -\frac{x_{grid}}{2} \leq x \leq \frac{x_{grid}}{2}\}, \quad y = \{y_{grid} \in \mathbb{R} : -\frac{y_{grid}}{2} \leq y \leq \frac{y_{grid}}{2}\}$$

A mapping function $f_{idx\_to\_chart}(i, j)$ maps the indices (i, j) to their associated Cartesian (x, y) coordinates:

$$f_{chart\_to\_idx}(i, j) : {\mathbb{Z}_{\geq 0}}^2 \mapsto \mathbb{R}^2$$

Defined for:

$$i = \{\frac{x_{grid}}{s_{cell}} \in \mathbb{Z}_{\geq 0} : 0 \leq i \leq \frac{x_{grid}}{s_{cell}}, \quad j = \{\frac{y_{grid}}{s_{cell}} \in \mathbb{Z}_{\geq 0} : 0 \leq j \leq \frac{y_{grid}}{s_{cell}}\}$$

The three dimensional configuration space for object is defined as:

$$\text{C-space}^{3D} = \begin{bmatrix} \begin{bmatrix} c_{(0,0,0)} & c_{(0,1,0)} & \cdots & c_{(0,j_{\max},0))} \\ c_{(1,0,0))} & c_{(1,1,0))} & \cdots & c_{(1,j_{\max},0))} \\ \vdots & \vdots & \ddots & \vdots \\ c_{(i_{\max},0,0))} & c_{(i_{\max},1,0))} & \cdots & c_{(i_{\max},j_{\max},0))} \end{bmatrix} \\ \begin{bmatrix} c_{(0,0,1)} & c_{(0,1,1)} & \cdots & c_{(0,j_{\max},1))} \\ c_{(1,0,1))} & c_{(1,1,1))} & \cdots & c_{(1,j_{\max},1))} \\ \vdots & \vdots & \ddots & \vdots \\ ,1)) & c_{(i_{\max},1,1))} & \cdots & c_{(i_{\max},j_{\max},1))} \end{bmatrix} \\ \begin{bmatrix} c_{(0,0,k_{\max})} & c_{(0,1,k_{\max})} & \cdots & c_{(0,j_{\max},k_{\max}))} \\ c_{(1,0,k_{\max}))} & c_{(1,1,k_{\max}))} & \cdots & c_{(1,j_{\max},k_{\max}))} \\ \vdots & \vdots & \ddots & \vdots \\ & c_{(i_{\max},1,k_{\max}))} & \cdots & c_{(i_{\max},j_{\max},k_{\max}))} \end{bmatrix} \end{bmatrix}$$

With $0 \leq i \leq i_{\max} = \frac{x_{grid}}{s_{cell}}, \quad 0 \leq j \leq j_{\max} = \frac{y_{grid}}{s_{cell}}, \quad k \in \mathbb{Z}_{\geq 0}$.

Similar to $f_{chart\_to\_idx}(x, y)$ and $f_{idx\_to\_chart}(i, j)$ the mapping functions $chart\_to\_idx(x, y, \theta)$ and $f_{idx\_to\_pose}(i, j, k)$ exist and map between the (x, y, θ) pose and the (i, j, k) indices.

An example configuration space for the point robot is presented below.

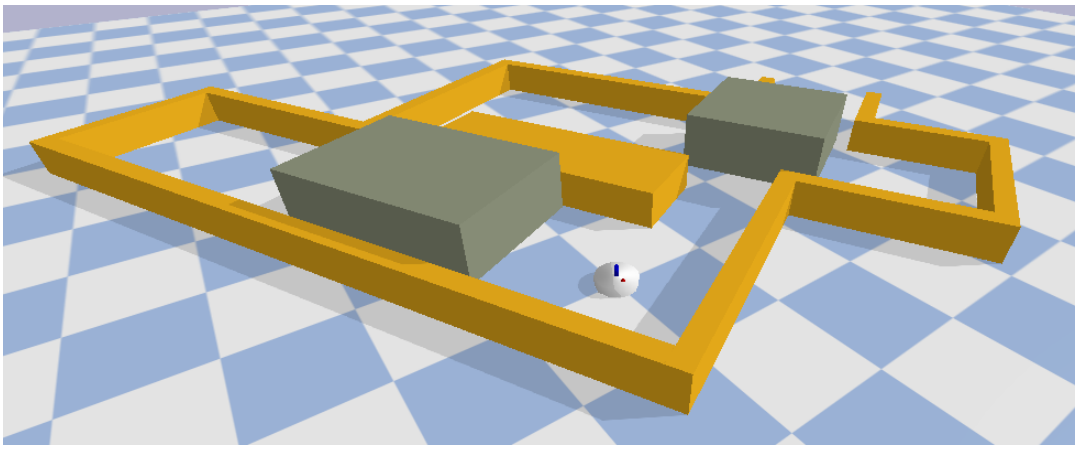
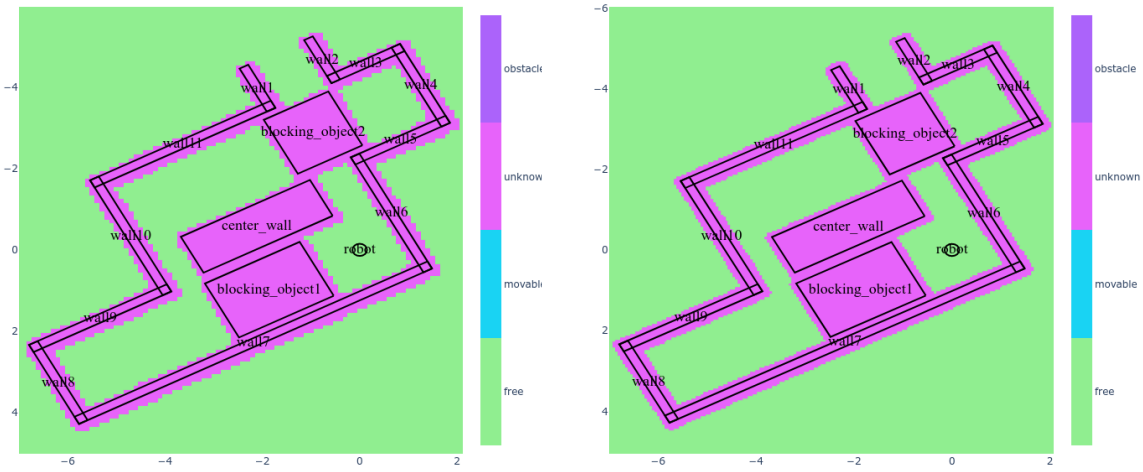


Figure 2.2: Robot environment with the point robot, unmovable yellow walls and movable brown boxes.

The point robot has a cylindrical shape; thus, a two dimensional configuration space can represent the free and obstacle space which can be seen in Figure 2.3.



(a) C-space^{circle} with $s_{cell} = 0.1$

(b) C-space^{circle} with $s_{cell} = 0.05$

Figure 2.3: The configuration space for the point robot with different cell sizes. The robot environment corresponds to the environment presented in Figure 2.2.

The resolution of the configuration space at Figure 2.3b is higher compared to Figure 2.3a because of a smaller grid size. A high resolution is better at detecting paths through small corridors and tight corners, but it comes at the cost of a more extended creation and search time. The robot classifies all objects as unknown since it is unaware of the objects' classes.

Path Existence Algorithm In the context of this thesis, we can describe path existence as: *For an object's discretized configuration space, there exists a list of neighbouring cells from starting to target configuration that does not lie in unmovable space.*

A path in configuration space is detected using the implemented $f_{shortest\_path}(c_{start}, c_{target},)$ function. This function takes a start- and target configuration, c_{start} and c_{target} and returns the shortest path between them that lies in free space. The *shortest_path* function searches for the shortest path using the Dijkstra algorithm [11] on the discreted configuration space. The shortest path validates the existence of a path (note that only geometric constraints are respected), and can help path planning discussed in Section 2.3. The shortest path helps path planning by providing an initial number of path planner nodes before the path planner creates nodes by random sampling. Such a conversion is also referred to as a "warm start". If no path can be found, the $f_{shortest\_path}$ function raises an error that will prevent path planning from occurring.

Unfeasible solutions and an undecidable problem The path estimator does not take system constraints into account. Thus, the path estimator can find a list of neighbouring cells from the start to the target configuration and conclude that a path exists. In reality, this path is unfeasible. An example is driving the boxer robot displayed in Figure 1.1b through a narrow, sharp corner. Whilst geometrically, the robot would fit through the corner, the non-holonomic constraints of the robot prevent it from steering through such a tight corner. It is for the action planner to detect that the path is unfeasible.

The path estimator suffers from another drawback, finding proof that there exists a path that is undecidable [51]. This is due to the chosen cell size during discretizing the configuration space. An example is a corridor having the same width as the robot. The robot fits exactly through this corridor. Detecting such a path requires many neighbouring cells that lie exactly in the center line of the corridor. Only with a cell size going to zero and the number of cells going to infinity such a path is guaranteed to be detected. Path non-existence, on the other hand, is easier to prove because the path estimator provides an upper bound on existing paths and a lower bound on non-existing paths [51]. The following flowchart neatly presents why the existence of paths is an *estimation* rather than an guarantee.

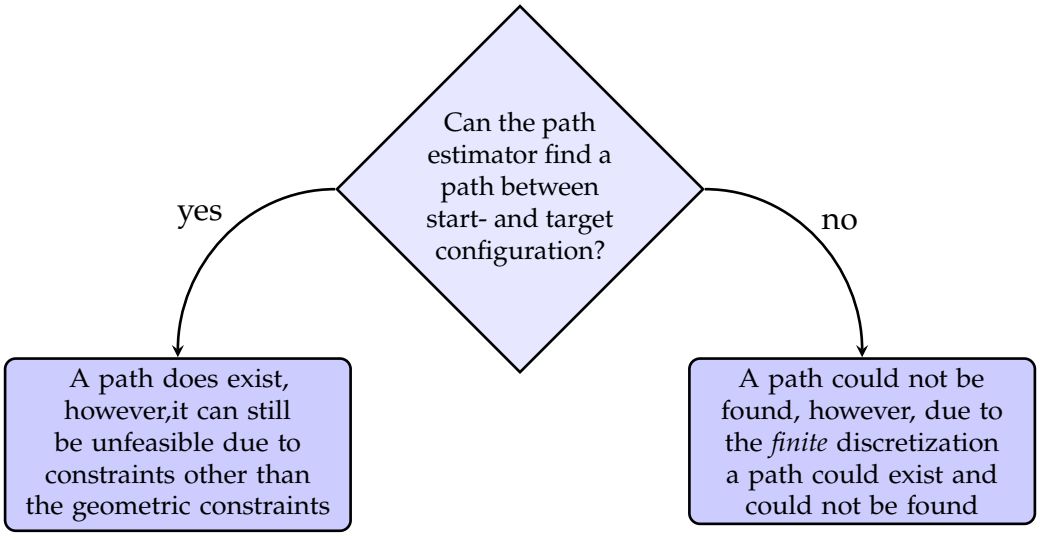


Figure 2.4: Flowchart displaying both drawbacks when checking path existence.

The path existence algorithm can detect non-existence paths. Thus checking path existence before motion or manipulation planning filters out several non-existent paths saving time and resources. Even if, in exceptional cases, the path estimator can yield unfeasible paths and fails to detect existing paths. Checking path existence before path planning filters some non-exist paths and is additionally motivated by two reasons. First, path estimation is orders of magnitude faster compared to path planning. Secondly,

the path estimation algorithm can provide a number of initial nodes to the path planner that can act as a "warm start".

2.3.2. Path Planning

Controllers discussed in Section 2.2 can track a path from start to target, given that all necessary ingredients are provided. One essential ingredient is a path to follow, and providing a path is the planners' responsibility; planners seek inside the configuration space for a path from the start to the target configuration. A practical example of such a path is a list of successive points in configurations space. How far the successive points can lie apart is a tuning parameter of the planner. Seeking a path in configuration space whilst avoiding unmovable objects is referred to as *path planning* and can be described as:

"The main idea is to avoid the explicit construction of the object space and instead conduct a search that probes the configuration space with a sampling scheme. This probing is enabled by a collision detection module, which the path planning algorithm considers a "black box." [24]"

The path planner is defined with the following tuple:

$$PathPlanner = \langle V_{MP}, E_{MP}, P \rangle$$

Where V_{MP} is a set of nodes, E_{MP} a set of edges and P a set of paths, a path is defined as:

$$path = [c_{start}, c_2, c_3, \cdots, c_{n-1}, c_{target}]$$

Where $n \in \mathbb{Z}_{\geq 2}$ is the number of configurations in the path.

The goal of the path planner is to find a path between a given start- and target configuration that results in the lowest *TotalPathCost*, defined as:

$$TotalPathCost = \sum_{i=1}^{n-1} Distance(c_i, c_{i+1}) \sum_{i=1}^{n-1} = \|c_i - c_{i+1}\|$$

Now pseudocode of the RRT* algorithm is presented that is split into three parts that are grouped by color as can be seen in Algorithm 1. Notice that the colour correspond to the later pseudocode and example figures. Each part is then discussed, as well as a number of variables and functions used in that respected part. When discussing each part an example is analyzed in which the path planner adds a single node to the connectivity graph. The variables and functions are neatly grouped after discussing the example in Table 2.2. Note that, the path planner in this section plans in free- and obstacle (or unmovable) space, later in Chapter 3 that will be extended to free-, unknown-, movable- and unmovable space.

A node x in the path planner consists of the following tuple:

$$x = \langle c, cost\_to\_source, key, prev\_node\_key, in\_tree \rangle$$

Where c is a point in configuration space, *cost_to_source* the cost toward the source node, *key* a unique key for the node, *prev_node_key* they parent key to which the node is directly connected with an edge, *in_tree* an indicator that the node is connected to the *start*- or *target* connectivity tree. The path planner starts by adding the first two initial nodes, the start- and target node. The path planner then enters a loop that adds randomly sampled nodes, until the stopping criteria is reached with the *NotReachStop* function.

Algorithm 1 Pseudocode for double tree RRT* algorithm. The pseudocode is split into three parts, Algorithms 2 to 4 that correspond to the blue, yellow and green coloured blocks.

1: $V_{MP} \leftarrow x_{init}$
2: **while** *NotReachStop* **do**
▷ Create, project and validate a new random node
▷ Find and connect new node to parent node
▷ Check if the newly added node can lower cost for nearby nodes and if a both connectivity trees can be connected
3: **end while**

The example that adds a single node starts in Figure 2.5. In this example, one node is added to the starting connectivity tree. Adding this node involves the following steps: generating a new random node, projecting the node to the nearest node, rewiring nearby nodes and connecting the start to the target tree.

The start connectivity tree consists of the nodes connected by edges containing the starting node, and vice versa for the target connectivity tree containing the target node. The algorithm grows the two *connectivity trees* by randomly sampling configurations and adding them as nodes to the start or target connectivity tree. The algorithm explores configuration space by growing these connectivity trees.

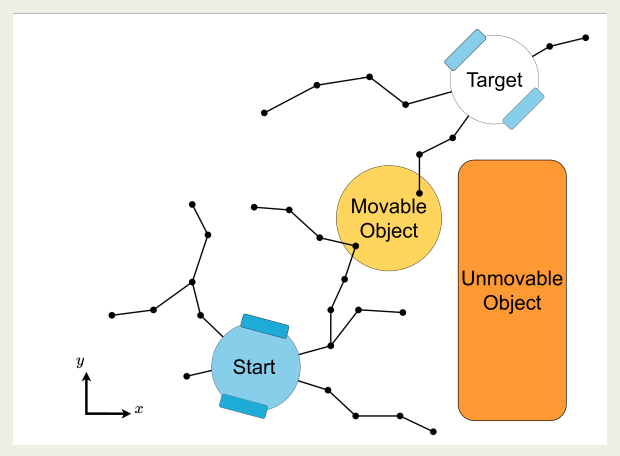


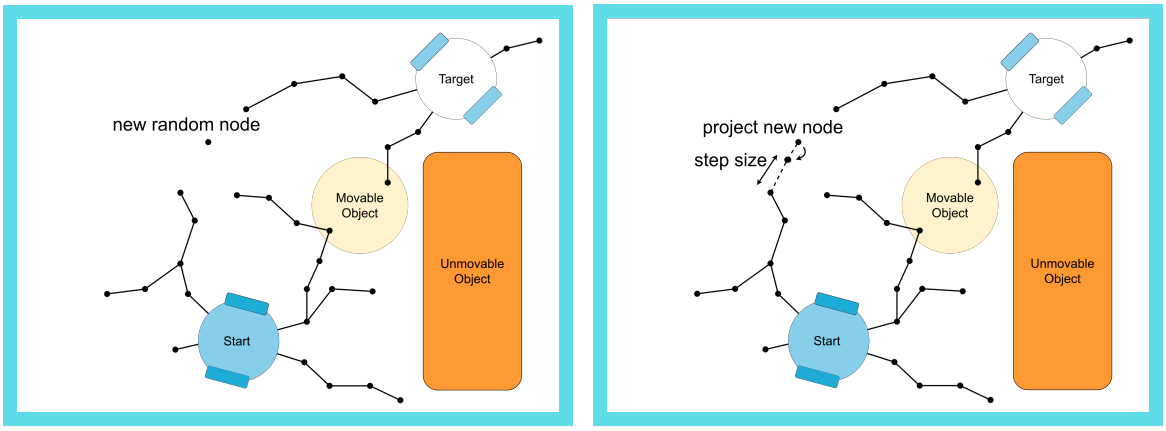
Figure 2.5: Snapshot of a two dimensional configuration with free- and unmovable spaces. Where unmovable space is indicated by the unmovable object, the rest of configuration space (including the movable object) is free space. The start- and target nodes, denoted as "Start" and "Target" respectively, serve as the source nodes for the start- and target connectivity tree.

The algorithm has two tuning parameters that can be tweaked; first, the *step size*, the maximal normalized distance between connected nodes in the connectivity trees, see Figure 2.6b for a visual example. Choosing a high step size will increase search speed because the connectivity trees grow faster. A higher step size comes at the cost of smoothness; the resulting path will be bumpier with sharper corners. Additionally, the path has an increased chance of collision with unmovable objects because, for two connected configurations in a path, the individual configurations can both lie in free space whilst a direct line between the configurations crosses through unmovable space. The second tuning parameter is the *search size*, which is a subspace around the newly sampled node (see, Figure 2.8).

Algorithm 2 Pseudocode to create, project and validate a new random node.

$v_{rand} \leftarrow Sample_{random}$
$v_{nearest} \leftarrow Nearest(v_{rand}, V_{MP})$
$v_{temp} \leftarrow Project(v_{rand}, v_{nearest})$
if $CollisionCheck(v_{temp})$ **then**
    $v_{new} = v_{temp}$
    $Cost_{toInitMin} \leftarrow +\infty$
else
    Continue
end if

The $Sample_{random}$ function creates a random node in free- or unmovable space. This random node is projected to the closes existing node using two functions. First, the $Nearest(x, V)$ returns the nearest node from x in V. Second, if the nearest node is further than *step size* Euclidean distance from the randomly sampled node, then the $Project(x, x')$ function projects x toward x', such that the Euclidean distance between x and x' is the *step size*. The projected random node can reside in free- or unmovable space, ensuring that the node is in free space is validated with the $CollisionCheck(x)$ function that returns true if x is in free space. Newly sampled nodes are added structurally, guaranteeing an optimal path is found with infinite sampling [7]. Where optimality is defined as the path with the lowest possible cost.



(a) A new random node is generated.

(b) The new node is projected toward the closest node, that becomes its parent node.

Figure 2.6: Create new node and connect to parent node.

The new node resides in free space and should now be added to either the start- or target connectivity graph. Adding the new node to either connectivity graph is an operation that creates a new edge between the new node and a to be determined parent node. The operation requires three functions, first a $NearestSet(x, V)$ function that returns set of nearest nodes from x in V that lie in the search space. The parent node is selected from the set of nearest nodes, from this set the node that results in the lowest $Cost_{toInit}$ is sought, defined as:

$$Cost_{toInit} = CostToInit(v_{near}) + Distance(v_{near}, v_{new})$$

The $Cost_{toInit}$ is proportional to the path length from a node toward the initial node in the connectivity graph. It can be seen as half of the $Cost_{path}$ that will be later defined as te cost for the entire path from starting- to target node. The second and third functions are the $Distance(x, x')$ function that returns the distance between node x and x', and a $CostToInit(x)$ function that finds the total cost from x to the initial node.

Algorithm 3 Pseudocode to find and connect new node to parent node.

$X_{near} \leftarrow NearestSet(v_{new}, V_{MP})$
for $v_{near} \in X_{near}$ **do**
  $Cost_{temp} \leftarrow CostToInit(v_{near}) + Distance(v_{near}, v_{new})$
  if $Cost_{temp} < Cost_{toInitMin}$ **then**
    $Cost_{toInitMin} \leftarrow Cost_{temp}$
    $v_{minCost} \leftarrow v_{near}$
  end if
end for
if $Cost_{toInitMin} == \infty$ **then**
  Continue
else
  $V_{MP}.add(v_{new})$
  $E.add(v_{minCost}, v_{new})$
end if

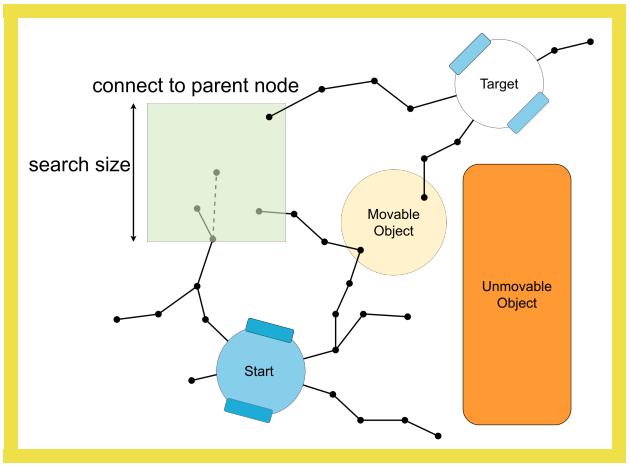


Figure 2.7: Connect new node to parent node that results in the lower $Cost_{toInit}$.

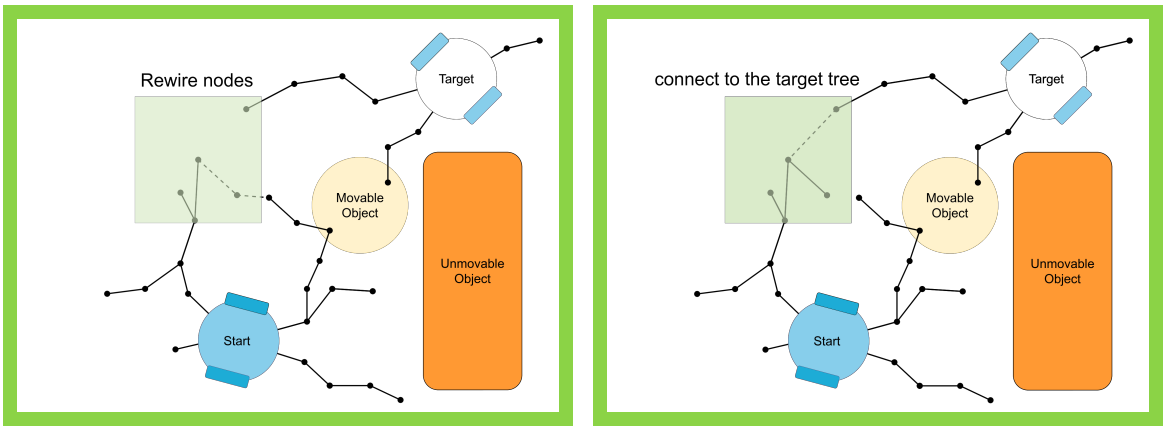
The new node is connected to a parent node as can be seen in Figure 2.8. A final step remains to be done, connecting the two trees or rewiring. For this step the $InSameTree(x, x')$ indicates if both x and x' are or are not in the same tree. For every node in the set of nearest nodes that is in the same tree as the new node, it is validated if rewiring would result in a lower $Cost_{toInit}$ for that node. The rewire procedure changes the parent node by removing and adding an edge, rewiring can be visually seen in Figure 2.8a. The node in the set of nearest nodes that are in the other three that results in the lowerst $Cost_{path}$ is connected to the new node. Thereby creating a path from start node to target node, the path is added to the set of paths with corresponding $Cost_{path}$, defined as:

$$Cost_{path} = CostToInit(v_{new}) + Distance(v_{new}, v_{other\_tree}) + CostToInit(v_{other\_tree})$$

Algorithm 4 Pseudocode to check if the newly added node can lower cost for nearby nodes with the rewire procedure, and if both trees can be connected, yielding a path.

```
1:  Cost_pathMin ← +∞
2:  for v_near ∈ X_near do
3:      if InSameTree(v_near, v_new) then
4:          if CostToInit(v_new) + Distance(v_new, v_near) < CostToInit(v_near) then
5:              E.rewire(v_near, v_new)
6:          end if
7:      else                                          ▷ Add lowest cost path to the list of paths
8:          Cost_temp ← CostToInit(v_new) + Distance(v_new, v_near) + CostToInit(v_near)
9:          if Cost_temp < Cost_pathMin then
10:             Cost_pathMin ← Cost_temp
11:             v_pathMin ← v_near
12:         end if
13:     end if
14:     if Cost_pathMin == ∞ then
15:         Continue
16:     else
17:         P.addPath(v_new, v_pathMin, Cost_pathMin)
18:     end if
19: end for
```

When the start connectivity tree is close enough (inside the search size of a newly added node) to the target connectivity tree, a path from start to target is found. The Increasing the search size improves the choice of the parent node and improves cost due to rewiring, but it also exponentially increases computation time.



(a) Rewire a node for which the cost can be lowered by removing and adding an edge.

(b) Connect start- to target connectivity tree.

Figure 2.8: Check if the newly added node can lower cost for nearby nodes and connect the start- to the target tree.

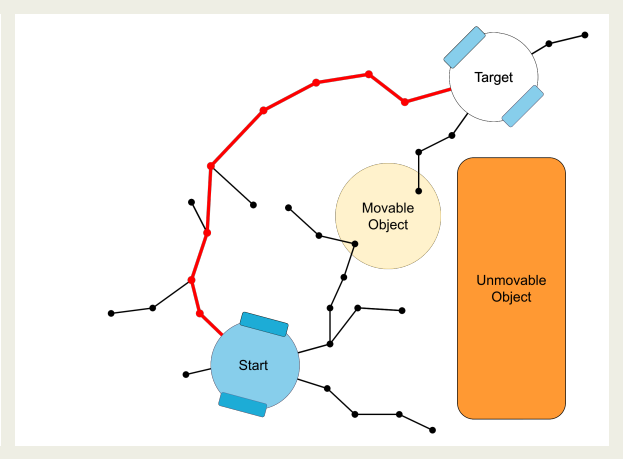


Figure 2.9: Path planner that found a path from start- to target node marked in red.

The existing path planner has now been discussed, the following table summarized the variables and functions. In the upcoming chapter, the path planner is extended to detect blocking paths.

Table 2.2: The variables and functions employed in the Algorithms 1 to 4.

x:	A node consisting of the following tuple: $\langle c, cost\_to\_source, key, prev\_node\_key, in\_tree \rangle$ where c a point in configuration space, *cost_to_source* the cost toward the source node x_{init}, *key* a unique key for the node, *prev_node_key* they parent key to which the node is directly connected, *in_tree* an indicator that the node is connected to the *start-* or *target* connectivity tree.
x_{init}:	The two initial nodes, x_{start} and x_{target}
NotReachStop:	True if the stopping criteria are not reached
$Sample_{random}$:	Creates a random node in free-, movable- or unknown space
$Nearest(x, V)$:	Returns the nearest nodes from x in V
$NearestSet(x, V)$:	Returns set of nearest nodes from x in V
$Project(x, x')$:	Project x toward x'
$CollisionCheck(x)$:	Returns true if x is in free space
$Distance(x, x')$:	Returns the distance between node x and x'
$CostToInit(x)$:	Find the total cost from x to the initial node
$InSameTree(x, x')$:	Returns true if both x and x' are in the same tree, otherwise return false

3

Proposed Path Planner

Finding a path between the start- and target configuration whilst avoiding collisions with an existing RRT path planner [7] was presented at the end of the previous chapter. This chapter extends that existing path planner's configuration space from free- and obstacle space to also incorperate movable- and unknown space. Movable space originates from objects in the environment that are known to be movable, unknown space originates from objects for which it is unknown weither they are movable or unmovable. The planner avoids obstacle space, is incentivezed to plan only in free space but can pass through unknown or movable space. A direct path lies in free space only, a blocked path passes through unknown- or movable space. To free a blocked path and make it a direct path, the object that causes the unknown- or movable space should be moved out of the way. Then a theoretic extension is mentioned to generate paths which respect dynamic constraints.*

3.1. Planning in Four Subspaces

The modified algorithm has four tuning parameters that can be tweaked; The *step size* and *search size* were discussed in Section 2.3.2. The third and fourth tuning parameters are the *UnknownSpaceCost* and *MovableSpaceCost*, which are fixed costs for crossing through unknown or movable space. Crossing through is defined as one or more nodes in the path lying in that subspace. If a path does not contain a node in unknown space, *UnknownSpaceCost* will be 0, equivalent to movable space and *MovableSpaceCost*. These cost are added to the $Cost_{path}$, which is redefined as:

$$Cost_{path} = MovableSpaceCost + UnknownSpaceCost + \sum_{i=1}^{n-1} Distance(c_i, c_{i+1})$$

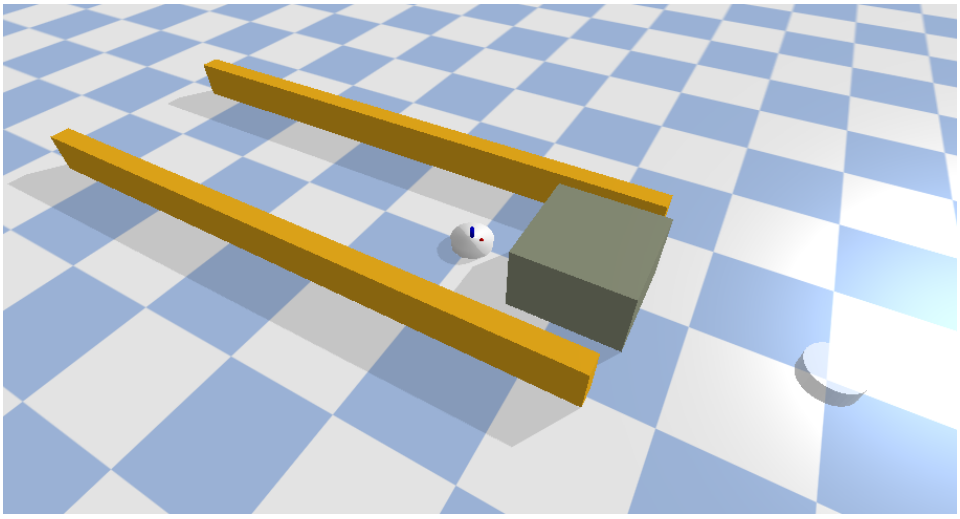
Where $n \geq 2$ configuration points make up the path starting at c_{start} and ending at c_{target}.

The RRT* algorithm searches for a path with the lowest cost, by adding a penalty for crossing through unknown or movable space the path planner is incentivised to find the shortest path around objects but prefers moving an object over making a large detour. Tuning the additional fixed cost for a path crossing through movable or unknown space balances the robot's decision between the length of a detour the robot is willing to drive, compared to pushing an object to free the path. Removing an unknown object bears more uncertainty than a movable object, motivating a higher cost to remove an unknown object compared to an known object. The pseudocode from Algorithm 1 is presented again, with changes due to the extension indicated with a red colour.

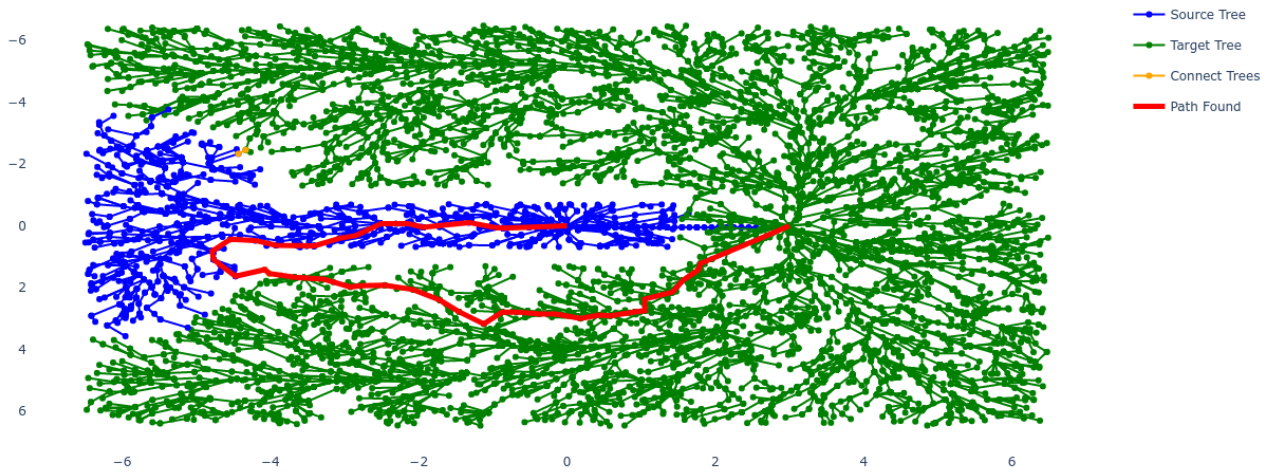
Algorithm 5 Pseudocode for extended RRT* path planning algorithm. Lines that contain changes compared to Algorithm 1 are indicated with the red colour.

1: $V_{MP} \leftarrow x_{init}$
2: **while** *NotReachStop* **do**
3: $v_{rand} \leftarrow Sample_{random}$ ▷ Create, project and validate a new random sample
4: $v_{nearest} \leftarrow Nearest(v_{rand}, V_{MP})$
5: $v_{temp} \leftarrow Project(v_{rand}, v_{nearest})$
6: **if** $CollisionCheck(v_{temp})$ **then**
7: $v_{new} = v_{temp}$
8: $Cost_{toInitMin} \leftarrow +\infty$
9: **else**
10: Continue
11: **end if**
12: $X_{near} \leftarrow NearestSet(v_{new}, V_{MP})$ ▷ Find and connect new node to parent node
13: **for** $v_{near} \in X_{near}$ **do**
14: $Cost_{temp} \leftarrow CostToInit(v_{near}) + Distance(v_{near}, v_{new}) + ObjectCost(v_{near}, v_{new})$
15: **if** $Cost_{temp} < Cost_{toInitMin}$ **then**
16: $Cost_{toInitMin} \leftarrow Cost_{new}$
17: $v_{minCost} \leftarrow v_{near}$
18: **end if**
19: **end for**
20: **if** $Cost_{toInitMin} == \infty$ **then**
21: Continue
22: **else**
23: $V_{MP}.add(v_{new})$
24: $E.add(v_{minCost}, v_{new})$
25: **end if**
26: $Cost_{pathMin} \leftarrow +\infty$
27: **for** $v_{near} \in X_{near}$ **do** ▷ Check if the newly added node can lower cost for nearby nodes and if a both connectivity trees can be connected
28: **if** $InSameTree(v_{near}, v_{new})$ **then**
29: **if** $CostToInit(v_{new}) + Distance(v_{new}, v_{near}) + ObjectCost(v_{new}, v_{near}) < CostToInit(v_{near})$ **then**
30: $E.rewire(v_{near}, v_{new})$
31: **end if**
32: **else** ▷ Add lowest cost path to the list of paths
33: $Cost_{temp} \leftarrow CostToInit(v_{new}) + Distance(v_{new}, v_{near}) + CostToInit(v_{near})$
34: **if** $Cost_{temp} < Cost_{pathMin}$ **then**
35: $Cost_{pathMin} \leftarrow Cost_{temp}$
36: $v_{pathMin} \leftarrow v_{near}$
37: **end if**
38: **end if**
39: **if** $Cost_{pathMin} == \infty$ **then**
40: Continue
41: **else**
42: $P.addPath(v_{new}, v_{pathMin}, Cost_{pathMin})$
43: **end if**
44: **end for**
45: **end while**

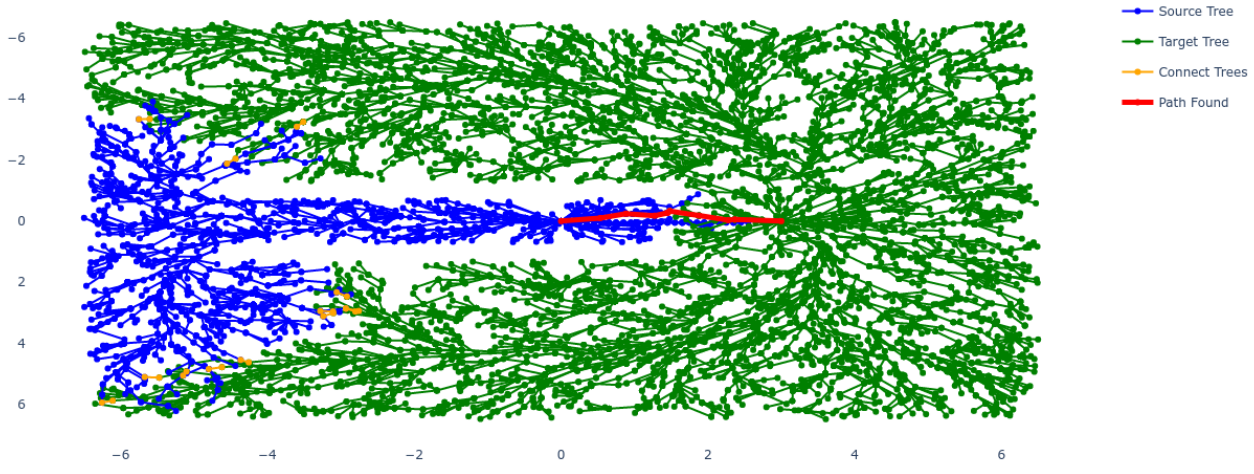
The proposed algorithm prevents planning a path through blocking objects except when no other option is available or a large detour can be prevented. No performance tests have been conducted on the modified path planner, apart from visual inspection. Figure 3.1 clearly shows the effect of varying *UnknownSpaceCosts*.



(a) Robot environment with the point robot, two yellow unmovable walls and an unknown brown box. The robot tasked to drive toward the opposite side of the brown box.



(b) Visualization of the planned path around the brown box and yellow obstacles, with *UnknownSpaceCost* = 1.



(c) Visualization of the planned path going through the brown box, *UnknownSpaceCost* = 0.5.

Figure 3.1: Driving task and two planned paths.

3.2. Kinodynamic Planning

Path planning whilst respecting dynamic constraints (e.g. non-holonomic constraints or constrains on the derivatives of the planned path) is known as kinodynamic planning [13]. A non-holonomic robot, such as the boxer robot in Figure 1.1b should respect the dynamic constraints, thus a path to track for the robot should be provided by a kinodynamic planner. To take dynamic constraints into account, a dynamic model should be incorporated. The popular method is to use motion primitives [36, 20]. Whilst other methods to respect dynamical constraints during path planning do exist (solving a two-point-boundary problem [25], setting curvature constraints with a steering function such as Dubins path or Reeds-Shepp curves [24]) a description is provided to incorporate a dynamic model using motion primitives in the upcoming paragraph.

When sampling a new node in the path planner a dynamic model can be incorporated. Currently a new node is created by generating random number within the boundaries of every dimension of the configuration space. Then to project that configuration point to the closes node, followed by a check if the newly sampled configuration does not lie in unmovable space, see lines 3-11 in Algorithm 5. To involve a dynamic model newly samples nodes are created by selecting a existing node, propagate a dynamic system model for a predefined number of time steps using randomly sampled input and the existing node as initial state. If the simulated future state does not lie in unmovable space, it is converted to a newly sampled node for the path planner. That new node, will respect the dynamic constraints since it is is created as a future simulation using a system model that respects the dynamic constraints.

This thesis focuses on the improvement of task execution as a result of learning dynamical models on how to manipulate objects in the robot environment. The proposed path planning algorithm can be extended into a kinodynamic planning algorithm to generate paths that respect non-holonomic and dynamic constraints. As will become clear later in this thesis, the selection of a dynamic model for a certain manipulation action will improve over time when the robot becomes more experienced in the robot environment. To incorporate a system model into path planning leads to the conclusion that: *When using dynamic models that improve modeling the system in term of capturing dynamics and lowering model mismatch, the path planner that incorporates a dynamic model to generate new nodes will improve the paths generated. Where path improvement implies firstly, that the system can be controlled to track a path found, secondly paths that do not respect the dynamic constraints are not generated.* Later in this thesis, during testing, the holonomic point robot is used. Since the point robot has no dynamic or non-holonomic constraints that must be respected, no extension of the proposed planning algorithm is made that makes the path planner a kinodynamic path planner.

Now that the modified path planner is discussed, the proposed robotic framework is discussed. The proposed framework relies on the required background from previous chapter, and relies on the modified path planner from this chapter.

4

Proposed Robot Framework

This chapter is dedicated to introducing and defining the proposed framework. The proposed framework consists of the hypothesis algorithm, the hypothesis graph and the knowledge graph. The ***hypothesis algorithm (halgorithm)*** *acts on the* ***hypothesis graph (hgraph)*** *and is responsible for searching and executing action sequences to complete a specified task. Section 4.2 is dedicated to introducing and defining the hgraph. Then the halgorithm is discussed and defined in Section 4.3. The chapter finalizes with the kgraph in Section 4.4.*

4.1. Overview of the Proposed Framework

Figure 4.1 presents a schematic overview of the interconnection of the kgraph, halgorithm and the robot environment.

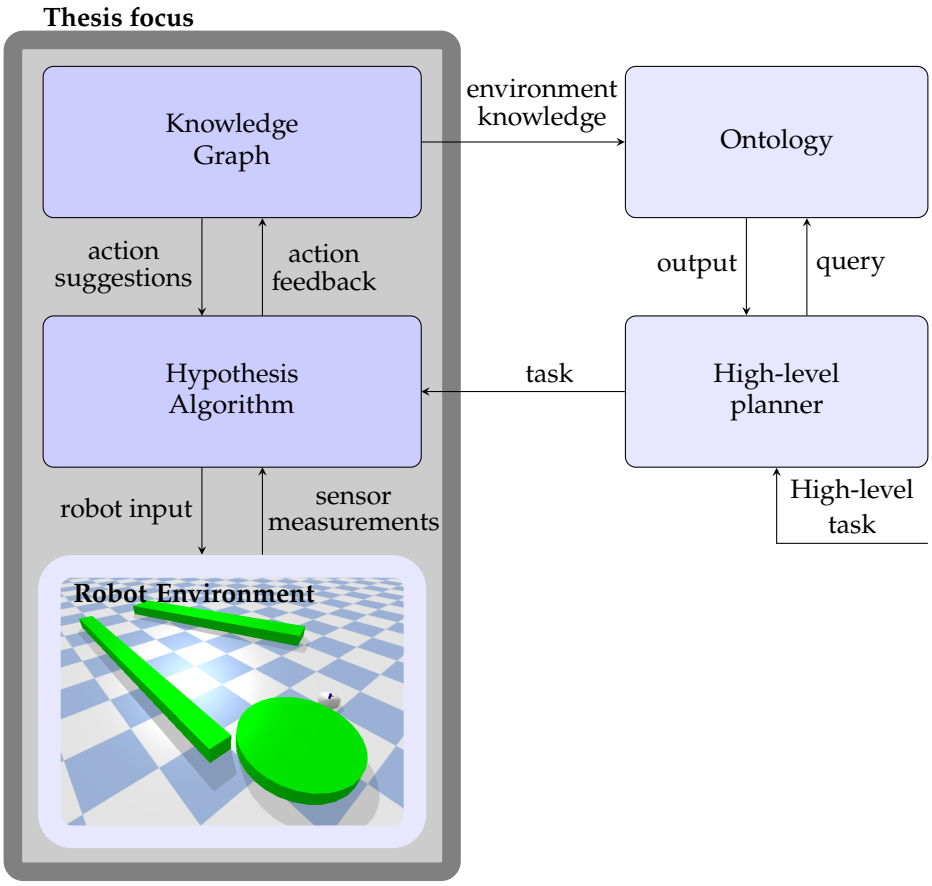


Figure 4.1: Flowchart representation of the proposed robot framework.

The above figure shows that the thesis focus could be augmented with an ontology and high-level planner. Such an augmentation would create a framework capable of completing high-level tasks such as cleaning or exploring.

4.2. Hypothesis Graph

The hypothesis graph (hgraph) consists of a set of nodes and edges. The node corresponds to an object at a configuration, and the edges correspond to actions. Overall, the hgraph represents a search in the composite configuration space. The halgorithm creates and updates nodes and edges in the hgraph and is discussed in Section 4.3. A search in the composite configuration space is avoided because an edge only operates in a single mode of dynamics, in the scope of this thesis, a driving mode or pushing mode. The hgraph is explicitly created for a task with a single start and a single target node for every subtask in the task. When the halgorithm halts, and the task is completed, the hgraph is no longer needed and is discarded.

The upcoming section defines and discusses the hgraph in Section 4.2.1. The halgorithm is then discussed in Section 4.3, where an explanation is provided on how the halgorithm searches for a solution in the composite configuration space. The section is concluded with an extensive example.

4.2.1. Definition

Before defining the hgraph, some definitions are provided on which the hgraph depends. First, recall the **configuration** definition.

Formally an **configuration**, $c_{id}(k)$ is a tuple of $\langle x(k), y(k), \theta(k) \rangle$

where $x, y \in \mathbb{R}, \quad \theta \in [0, 2\pi)$

An object is represented as its shape and configuration
Formally, a **object**, $obj_{id}(k) = \langle c(k), shape \rangle$

where *shape* is linked to a 3D representation of the object, *id* is an identifier for the object.

An object node represents an object in a configuration.
Formally, a **objectNode**, $V_{id}^{obj} = \langle \text{status}, obj(k) \rangle$.

An edge describes how a node transitions to another node in the hgraph. In the robot environment, an edge represents an object's configuration change. Edges are split into three categories since they accomplish different goals. *System identification edges* that have as a goal to collect IO data and to generate a system model with that data, *action edges* steer a system toward a target configuration and *emtpy edges* connect nodes that contain different objects. The edges are formally defined as:

A **identification edge**,

$$e_{(from,to)} = \langle \text{status}, id_{from}, id_{to}, \text{Identification Method}, \text{controller}, \text{input} \rangle$$

With id_{from} and id_{to} indicating the node identifier where the edge points from and towards, respectively, the identification method indicates the used method, and the controller contains the control method used for driving the robot during the collection of IO data. The input contains multiple input sequences to apply to the system. The yielded system model must meet the controller requirements, otherwise, they are incompatible.

A **action edge**,

$$e_{(from,to)} = \langle \text{status}, id_{from}, id_{to}, \text{verb}, \text{controller}, \text{dynamic model}, \text{path} \rangle$$

With id_{from} and id_{to} indicating the node identifier of the node in the hgraph where the edge starts from, and points to, verb is an English verb describing the action the edge represents (driving, pushing), the controller contains the control method used for driving the robot, the dynamic model is the dynamic model used by the control method and the path a list of configurations indicating the path connecting a start- to target node.

A **empty edge**,

$$e_{(from,to)} = \langle \text{status}, id_{from}, id_{to} \rangle$$

Where the status for an emtpy edge can be INITIALIZED or FAILED.

Connecting nodes with edges is eleborated upon in the upcoming section. Two nodes can only be connected if they both contain the same object. The empty edge main goal is to connect nodes that contain different object, to involve multiple objects in a single action sequence.

Now that the nodes and edges have been defined, the hgraph can be defined.

Formally, a **hypothesis graph**, $G^{hypothesis} = \langle V_H, E_H \rangle$
Where V_H is a set of nodes and E_H a set of edges, defined as: $V_H = \{V_{H_i}^{obj}\}, \quad E_H \in \{e_{(i,j)} | E_{Hi}, E_{Hj} \in \{V_H{}^{obj}\}, i \neq j\}$.

Most hgraph components have been defined. The status of an identification or action edge remains undefined and requires further explanation.

4.2.2. Edge statusus

The edges are split into two categories, identification edges and action edges. An identification edge sends an input sequence and records the system output. That IO sequence and assumptions on the system are the basis for system identification, techniques on various system identification methods are discussed in Section 2.1. The goal is to create a dynamic model augmented with a corresponding controller that forms closed-loop stable control. The status of an identification edge can be visualized in the following Finite State Machine (FSM).

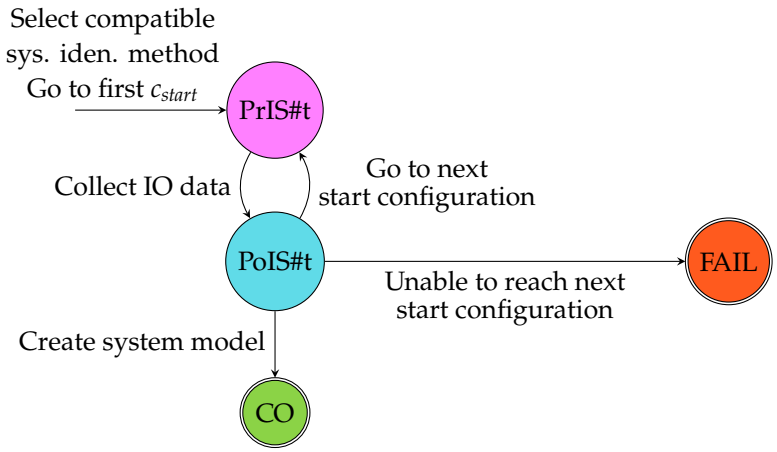


Figure 4.2: Finite State Machine displaying the status of an identification edge

PRE INPUT SEQUENCE number t (PrIS#t):	Go to target configuration to apply the input sequence.
POST INPUT SEQUENCE number t (PoIS#t):	Collect the output sequence.
COMPLETED (CO):	The edge has driven the system toward its target configuration, and its performance has been calculated.
FAILED (FAIL):	An error occurred, yielding the edge unusable.

An identification edge corresponds to an action edge because its goal is to generate a system model to hand over its corresponding action edge. The system identification method is selected after the action edge is selected to yield a system model compatible with the controller that resides in the action edge. Two types of system models are generated: system models that describe the driving behaviour of the robot and system models that describe the robot's push behaviour and an object's.

After initialization, an action edge starts propagating its status as indicated in Figure 4.3. An action edge's eventual goal is to track a path. First, the existence of a path is estimated, a system model must be provided, and action planning must be performed.

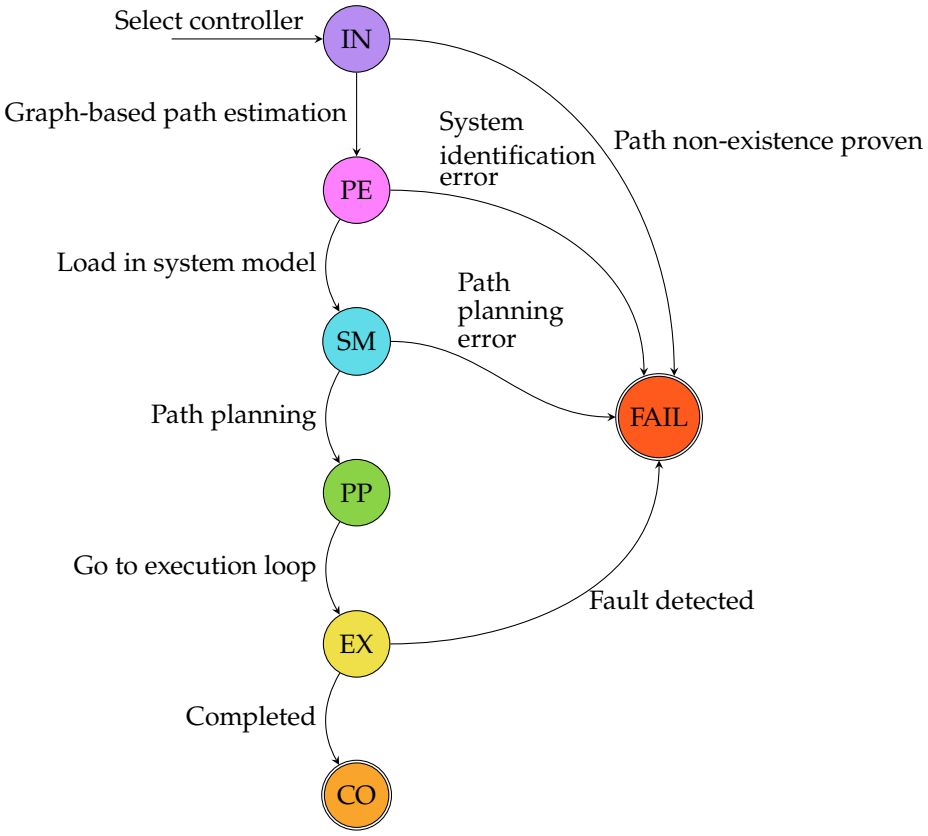


Figure 4.3: Finite State Machine displaying the status of an action edge

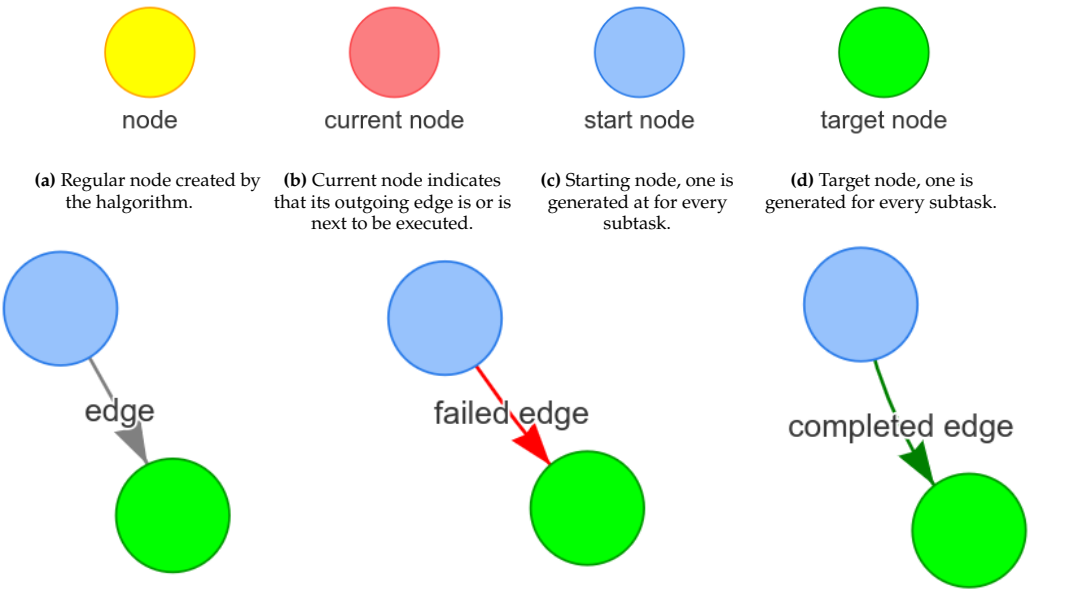
INITIALIZED (IN):	The edge is created with a source and target node, which are present in the hgraph. A choice of controller is made by random selection.
PATH EXISTS (PE):	A graph-based search is performed to validate whether the target configuration is reachable, assuming the system is holonomic.
SYSTEM MODEL (SM):	A dynamics system model is provided to the controller residing in the edge.
PATH PLANNED (PP):	Resulting from a sample-based planner, a path from start to target configuration is provided.
EXECUTING (EX):	The edge receives observations from the robot environment and sends back robot input.
COMPLETED (CO):	The edge has driven the system toward its target configuration, and its performance has been calculated.
FAILED (FAIL):	An error occurred, yielding the edge unusable.

4.2.3. Legend

hmmm, you want this?

Figure 4.3 shows that many steps must successfully be completed before the edge can be executed. Before executing edges, edges must be initialized, which is where the next section is dedicated to.

The following figure presents an legend for the hgraphs that will be presented in the next section.



(a) Regular node created by the halgorithm.

(b) Current node indicates that its outgoing edge is or is next to be executed.

(c) Starting node, one is generated at for every subtask.

(d) Target node, one is generated for every subtask.

(e) Edge with status IN, PE, SM, PP or EX.

(f) Edge with status FAILED (FAIL)

(g) Edge with status COMPLETED (CO)

Figure 4.4: Legend for the hypothesis graph.

This chapter has terminology that is conveniently grouped in Table 4.1.

Table 4.1: The proposed-framework-related terminology is juxtaposed in the left column alongside its corresponding description in the right column.

Task:	Tuple of objects and corresponding target configurations. $$\text{task} = S = \left\langle Obj_{task}, C_{targets} \right\rangle$$
Subtask:	A single object and a single target configuration. $$\text{subtask} = s = \left\langle obj_{subtask}, c_{target} \right\rangle$$
Object Class	Classification assigned to an object. $$\text{OBJ\_CLASS} = \text{Unknown} \vee \text{Obstacle} \vee \text{Movable}$$
Node:	A node in the hgraph represents an object in a configuration with node status that indicates if the halgorithm manages to place the object to that configuration. $$\text{node} = v = \langle \text{status}, obj, c \rangle$$
Node Status:	Status of a node indicates if a node is initialized, the halgorithm was able to bring the object to the configuration or whether the halgorithm fails to bring the object to its configuration. $$\text{NODE\_STATUS} = \text{Initialised} \vee \text{Completed} \vee \text{Failed}$$ elaborate information on the edge statuses can be found in Figure 4.3.
Edge:	Edge connecting a node to another node in the hgraph or kgraph. $$\text{edge} = e = \left\langle \text{status}, id_{from}, id_{to}, \text{verb}, \text{controller}, \text{dynamic model}, \text{path} \right\rangle$$
Edge Status:	Status of an edge, $$\text{EDGE\_STATUS} = \text{Initialised} \vee \text{PathExists} \vee \text{SystemModel}$$ $$\vee \text{PathPlanned} \vee \text{Executing} \vee \text{Completed} \vee \text{Failed}$$
Non-Failed Status:	Node- or edge status other than FAILED.
Hypothesis:	Sequence of successive edges in the hgraph, an idea to put an object at its target configuration. If executed and completed, a subtask is completed. $$\text{hypothesis} = h = [\, e_1, e_2, e_3, \ldots e_m \,] \; m > 0$$
Hypothesis Algorithm:	Graph-based algorithm that searches for hypothesis in the hgraph to complete subtasks, eventually completing a task.
Hypothesis Graph:	Collection of nodes and edges. For every subtask, a start and target node exist in the hgraph, and the halgorithm searches for a path through nodes and edges to connect the start to the target node. $$\text{hgraph} = G^{hypothesis} = \langle V_H, E_H \rangle$$
Knowledge Graph:	Collection of nodes and edges. The kgraph acts as a knowledge base and can be queried for an action suggestion. $$\text{kgraph} = G^{knowledge} = \langle V_K, E_K \rangle$$

4.3. Hypothesis Algorithm

this section starts with a simple example that generates and executes the hypothesis to drive toward a target pose. Then the search and execution loop are discussed, that constitute the principal components of the proposed halgorithm. Step wise, the terminology is elaborated upon whilst an example of a pushing task is discussed. Then two more examples are provided that involve a blocked path and detecting faults during task execution. Finally, the pseudocode can be resented, supported by a proposed halgorithm flowchart.

Two arguments initialize the halgorithm, first, a set of object geometry that contains the dimensions of the objects in the robot environment for internal representation, and second a task to solve.

Additionally several parameters must be specified, such as the grid size, maximum allowed input to the robot and tuning parameters for the path estimator, the path planner and controllers. When all arguments and parameters are provided, the halgorithm can be is initialized. There is only a single access point toward the halgorithm, the *Respond(observation)* function. This function takes an environment *observation* that updates the internal objects' poses. The function *Respond(·)* returns control input for the robot. In this thesis, the sensor measurements coincide with the poses objects in the environment. Recall that the perfect-sensor assumption, assumption 1.2.2, makes access to every object's exact configuration possible.

Now a relatively simple example is presented to indicate how the halgorithm operates, later every step will be extensively elaborated. The task in the example consists of driving to a single target pose. The leftmost subfigure in figure 4.5 visualizes the initialization of a start and target node, which are connected with a drive action edge in the center figure. To make the drive edge ready for execution in the center subfigure, a system model must be provided to the controller residing in the drive action edge. Motivating the *sys. iden* edge and the c_{robot_model} node in the rightmost subfigure.

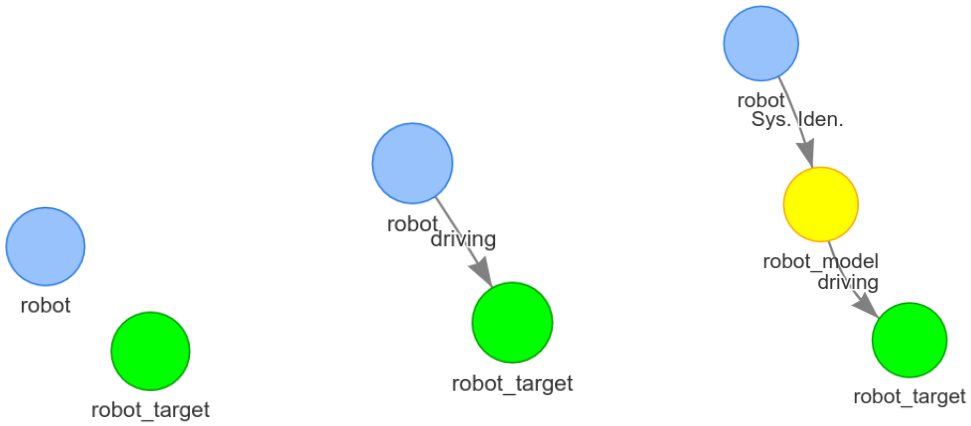


Figure 4.5: First stages of the hgraph when the halgorithm searches for an hypothesis to a driving task.

Now that an hypothesis is created that consists of an identification- and an action edge, the halgorithm alternates from the search loop to the execution loop, both loops are addressed shortly. In the execution loop, the halgorithm executes the edges by sending input toward the robot, which can be visualized in the figure below.

The generated and executed example for a driving task just discussed is provided to show a simple example. It leaves many details out, which are now elaborated. Start with initializing start- and target nodes, then the search- and execution loop are elaborated.

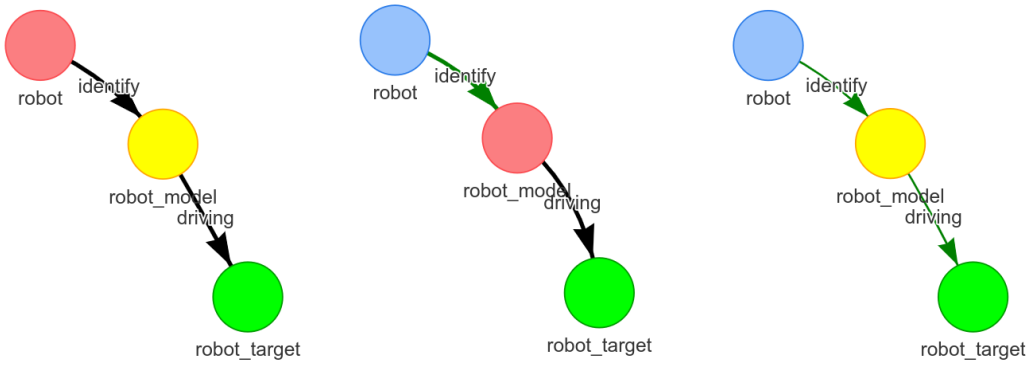


Figure 4.6: Multiple stages of the hgraph when the halgorithm executes the hypothesis found in figure 4.5.

Initialization of the halgorithm the halgorithm is initialized with a task that consists of one or more subtasks. Start- and target nodes are created for every subtask, and their status is set to INITIALIZED. Then the goal of the halgorithm is to connect every starting node to its corresponding target node with a hypothesis. The target node's status is set to COMPLETED when a hypothesis is completed successfully. If the halgorithm could not find a hypothesis that completes a subtask, the halgorithm concludes it cannot complete that subtask, and the target node's status is set to FAILED.

4.3.1. The Search and the Execution Loop

the proposed algorithm comprises two main parts, a search loop and an execution loop. The halgorithm searches for a hypothesis in the search loop. In the executions loop, the halgorithm tests hypotheses by executing the edges which form the hypothesis. This chapter finalized with a flowchart in figure 4.15 that will be familiar to the following figure where the two main loops can be identified.

Hypotheses are formed while the halgorithm resides in the search loop. Forming a hypothesis generates nodes, edges, and progressing their status as described in figures 4.2 and 4.3. In the execution loop *an edge is being executed*, a phrase to describe that the controller residing in an edge is sending control input toward the robot. The halgorithm operates synchronously. The result is that the robots cannot operate whilst the halgorithm resides in the search loop, and during execution, no hypothesis can be formed or updated. The halgorithm alternates between the search and execution loop; when in the search loop, a hypothesis is generated, that hypothesis is tested in the execution loop. The execution loop executes the edges that form the hypothesis one by one until either a fault is detected or the hypothesis is completed. Upon fault or completion, the halgorithm alternates back to the search loop

When entering or re-entering the search loop, the first thing to determine is if there are unfinished subtasks and, if unfinished subtasks exist, which nodes to connect in order to form a hypothesis that completes that subtask. For such functionality three functions are created; *SubtaskNotFinished*, *GoBackward(v)*, *FindCorrespondingNode(v)*. These functions are now discussed.

When elaborating the halgorithm, an example presents a visual example with every step in the halgorithm. In this example, the robot generates a hypothesis to complete a pushing task that contains a single subtask, initialization and the first generated edges are presented in figure 4.8.

finding unfinished subtasks determining if there exists an unfinished subtask is validated with the *SubtaskNotFinished(S)* function. It checks the status for every target node in hgraph. The three statuses are; initialized, completed and failed. A target node with an initialized status corresponds to an uncompleted subtask and is returned by the *SubtaskNotFinished(S)* function. If all existing target nodes have either a completed or failed status, the halgorithm concludes that the task is completed.

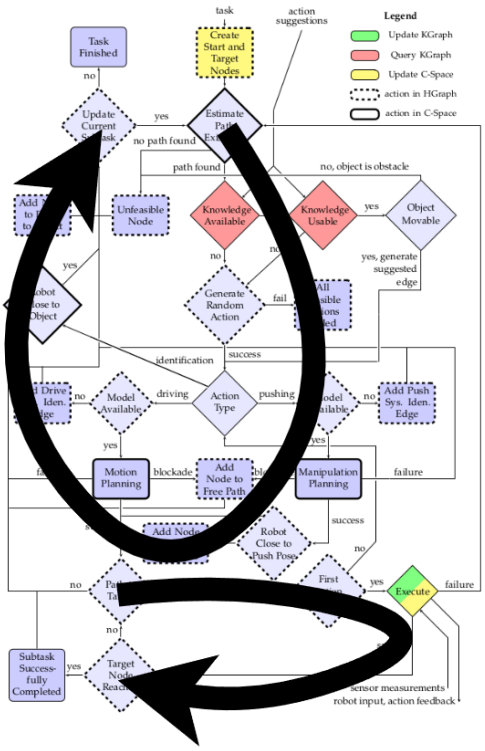


Figure 4.7: The search (upper) and execution (lower) loop, that make up the main part of the proposed halgorithm. The figure's goal is to present the two loops, the flowchart in the background is presented full page in Figure 4.15.

creating a hypothesis for a subtask suppose the *SubtaskNotFinished* returns a target node corresponding to an unfinished subtask. In that case, the halgorithm starts searching for a hypothesis connecting the start node to the target node. In figure 4.8a, the nodes to connect are the *box* node to the c_{box_target} node. These two nodes are a start- and a target node, the nodes to connect are not necessarily start- and target nodes themselves, as seen in figure 4.8b. Here the c_{robot} node must be connected to the c_{box} node. These nodes are both starting nodes. The first challenge is to find the two nodes to connect from an unfinished target node.

The halgorithm relies on a backward search technique. The backward search technique can be described as *start the search at a goal state and work backwards until the initial state is encountered [24].* A motivation for a backward search over a forward search is that it might be the case that the branching factor is significant when starting from the initial state. In such cases, it might be more efficient to use a backward search. If the *SubtaskNotFinished* returns an unfinished subtask, the halgorithm starts searching for a hypothesis connecting the start node to the corresponding target node. The first step is to find the right nodes in the hgraph, which is now discussed.

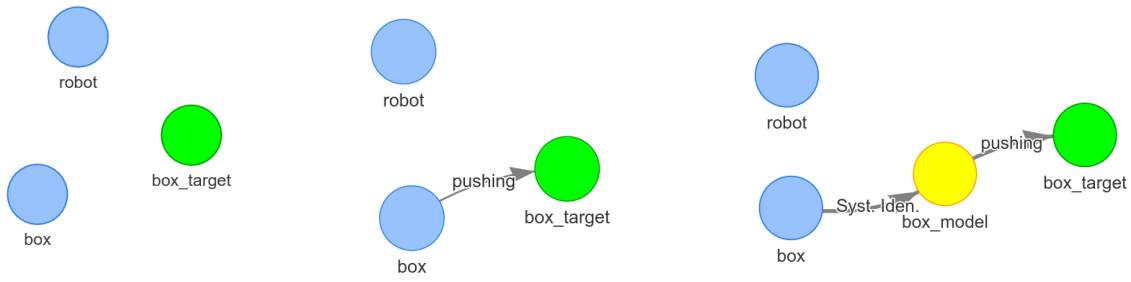
The *GoBackward*(v_{target}) function takes a target node v_{target} that corresponds to a unfinished subtask. It then traverses backwards via non-failed edges to find the node that points toward the target node. The function stops traversing back when it encounters a node with a FAILED status or when no edge exists to traverse backwards over. It returns the last node, that node points toward the target node over a sequence of edges with a status other than failed, and all these edges point toward nodes with a status other than failed. In figure 4.8a the *GoBackward*(v_{box_target}) function returns the v_{box_target} node, in figure 4.8b the *GoBackward*(v_{box_target}) returns the v_{box} node.

The *GoBackward*(v_{target}) finds a node to connect to, a corresponding node is sought to connect from, which the *FindCorrespondingNode(v)* does. *FindCorrespondingNode(GoBackward(v))* takes a node as parameter and returns an existing node that contains the same object as its arguments node; if such a node does not exist, a new node is created. In both figures 4.8b and 4.8c, *FindCorrespondingNode(GoBackward*(v_{target})*)*

returns node v_{box}. The nodes that the halgorithm desires to connect are renamed to prevent long function names:

$$v_{to} = GoBackward(v_{target})$$

$$v_{from} = FindCorrespondingNode(GoBackward(v_{target}))$$



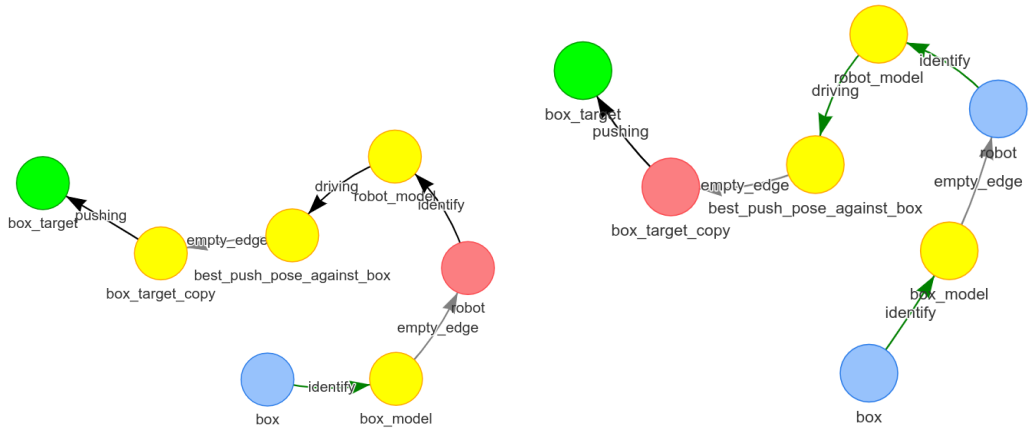
(a) Initialize start- and target nodes. **(b)** Creation of a push action edge. **(c)** Creation of a identification edge.

Figure 4.8: First stages of the hgraph when the halgorithm creates an hypothesis for a pushing task.

creating edges the *ConnectWithEdge*(v_1, v_2) function connects two nodes with an edge, such as the nodes v_{from}, v_{to} just introduced. In this thesis the robot can take two actions, drive and push. It is required that both nodes contain the same object. The push action edge generated and displayed in figure 4.8b is between two nodes containing the *box* object. An *EmtpyEdge* is introduced to involve nodes that contain different objects. The emptyEdge serves only to connect nodes that contain different objects and can have status initialized or failed. The halgorithm can traverse over emptyEdge if the status is INITIALIZED.

Push action edges require more than only initializing and preparing for execution, also they spawn new nodes by default and validate if objects are movable. The robot must first drive toward a push pose against or close to the object to push. When a push action edge has planned a path, and updates its status to PATH IS PLANNED, the halgorithm then creates a $v_{best\_push\_position}$ node, which configuration depends on the object's planned path. Thus, a path is planned, and then the best push pose is determined. The newly created node is connected before the push action edge, where an empty edge points from $v_{best\_push\_position}$ to e_{drive}'s source node. The first occurrence of an $v_{best\_push\_position}$ can be visualized in Figure 4.9a. When it is unknown if objects are movable or unmovable, a push action edge performs a tests to determine the class of an object. If the push action is unable to move the object in the first 50 time steps, the object is classified as UNMOVABLE, otherwise it is classified as MOVABLE.

Valid Hypotheses Before a hypothesis can be executed, the hypothesis must be valid. A hypothesis is valid when two conditions are met. First, it starts at the start node and points toward the target node over a sequence of edges with a non-failing status that all point toward nodes with a non-failing status. Second, the first edge in the hypothesis must be ready for execution which the next paragraph will elaborate upon further. To indicate a node or edge has a status other than the FAILED status, that node or edge is called a non-failed node or -edge. To check if an hypothesis is valid the *IsConnected*(v_1, v_2) is created. This function checks if there exists a path in the hgraph from v_1 to v_2 over a sequence of non-failing nodes and -edges. In the pushing task example, the first occurrence of a valid hypothesis is presented in Figure 4.9a.



(a) Executing the hypothesis and generated new nodes.

(b) Executing the pushing edge.

Figure 4.9: The hypothesis for a pushing task becomes valid and is executed. Then a path for the pushing edge is planned which generates new nodes to drive toward the best push pose against the box.

Preparing edges for Execution In contrast to identification edges, action edges must first take several actions in preparation before they are ready to send input toward the robot. The status of an edge indicates at which step of preparing the action edge is and can be visualized in Figure 4.3. After initialization, the action edge performs path estimation, loads in a system model, performs path planning and then, it is ready for execution. Two functions are created to make edges ready for execution. The *ReadyForExecution(e)* validates if an edge is ready for execution. Identification edges are ready for execution when they bear a non-failed status, action edges are ready for execution when they bear the PATH PLANNED or EXECUTING status. The *MakeReady(e)* function takes an edge and takes action depending on its status presented in the following table.

Table 4.2: The action edge status is presented in the left column, the corresponding action taken by the *MakeReady* function to prepare an action edge for execution in the right column. An action edge increments its status as indicated in Figure 4.3.

Action edge status	action taken by *MakeReady* function
INITIALIZED	Create a path estimator and estimate path existence. If no path can be estimated, the status is updated to FAILED. If a path can be estimated, a shortest path is found that acts as a "warm start" for the path planner.
PATH EXISTS	Load in a system model.
SYSTEM MODEL	Create path planner and plan path. The edge status is updated to FAILED if no path can be found. If a path is found, it acts as a reference signal for the controller. In the case of an push action edge, a $v_{best\_push\_position}$ is created. Additionally, the path the planner finds can indicate that an object is blocking. In such cases, the halgorithm must first push that object to free the path. An example of such a case is provided in Figure 4.12.

Hypothesis Execution When the halgorithm creates a valid hypothesis, it switches from the search loop to the execution loop. Executing a hypothesis is managed by three functions. The first edge in the hypothesis is ready for execution and thus contains a controller and a path to track. That edge is executed, and its controller sends input toward the robot to track the path. The *SteerTowardTarget(e)* calculates the input that steers the robot toward the path. The *TargetNotReached(e)* validates if the robot has reached the target, which is the last configuration in the path. A margin is set by which the *TargetNotReached(e)* concludes that the robot is close enough to the final target pose. For drive actions,

that margin is set to 0.1 meters measured in Euclidean instance between the robot and the robot's target position.

For push action, that margin is set to 2 meters, measured in Euclidean distance between the object and the object's target position. These values are tuned by trial and error and is one of the improvements that can be made in future work. The large margin for pushing tasks is set to ensure that the target pose is reached. With a lower margin, the object is often pushed further than the target position. A fault is then detected because the deviated too much from the path, the robot then drives toward the object's opposite side to again, push it over the target position. Fault detection is discussed in upcoming paragraph. After the successful completion of an edge, the next edge in the hypothesis is selected by the *IncrementEdge* function. Two possible outcomes exist, the next edge is ready for execution, then the halgorithm remains in the execute loop. Or, the next edge is not yet ready, then the halgorithm goes from the execution loop toward the search loop to prepare the next edge for execution.

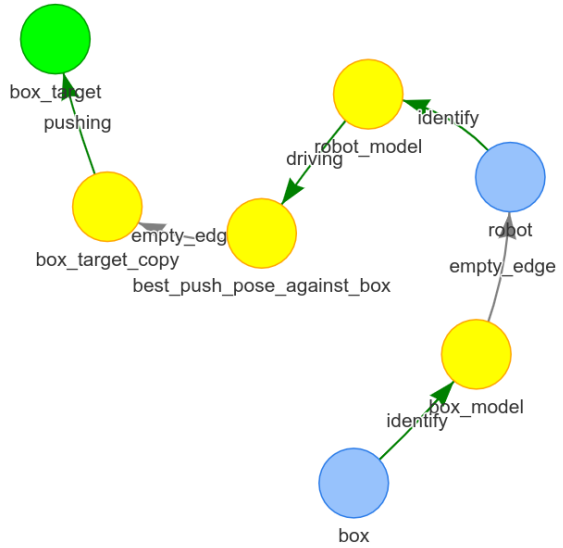


Figure 4.10: The hgraph after the pushing task was successfully completed.

Completing Hypotheses and Edges When the last edge in an hypothesis is completed, that subtask is completed. The hgraph randomly selects a next subtask to complete until there are no unfinished subtasks left, the halgorithm then concludes that the task is completed. Opposed to successful completion, subtasks can unsuccessfully complete, which is discussed in the upcoming paragraph.

4.3.2. Fault Detection

The proposed framework implements a fault metric named the *monitored metrics* which are presented shortly, first motivation for the monitoring metrics is given. When the halgorithm resides in the execution loop, it cannot search for action sequences, which is performed in the search loop. During the execution of an action, the hypothesis algorithm is unable to perform any other action. This blocking behavior has some implications, mainly that the controller can steer the system to a configuration from which it cannot independently reach the target configuration, as a result, it will never halt. For example, a controller tries to drive the robot toward a target configuration but there is an unmovable obstacle in the way. Another example is the controller is closed-loop unstable and never reaches its target configuration. Both examples do not occur is well defined simulation environments, because of the *closed-world assumption*. In the real world, an unexpected blocking obstacle or unstable controllers are more likely to occur.

Detecting controller faults is a large robotic topic [21], properly implementing a fault detection and diagnosis module is out of the scope of this thesis. Instead, two simple metrics will be monitored during

execution, by the *FaultDetected(e)* function. Upon detection of a fault, the *HandleFault(e)* function then updates the executing edge's status to FAILED, and the halgorithm switches from the execution loop to the search loop in search for a new hypothesis. The first monitoring metric is Prediction Error, and can be described as:
With the current configuration of the system, calculated system input and a system model, a prediction info the future is made every time step. Then the system input is applied to the system, the time step incremented and the configuration is measured. The prediction error is than the difference between predicted and measured configuration.

The Prediction Error (PE) is defined as:

$$\epsilon^{pred}(k) ::= ||\hat{c}(k|k-1) - c(k)||$$

Where $\hat{c}(k|k-1)$ is a prediction of the configuration and $c(k)$ is the measured configuration.

During execution a sudden high PE indicates unexpected behavior occurs, such as when the robot has driven into an object which it was not expecting. A high PE, which persists indicates that the robot is continuously blocked. A few high prediction errors are allowed, but when the PE exceeds a pre-defined threshold and persists over a pre-defined time, the hgraph concludes that there was an fault detected during execution and the edge's status is updated to FAILED.

The second monitoring metric is the Tracking Error that can be described as:
A controller tracks a path that consists of a list of configurations by steering the system to the upcoming configuration in the list. When that configuration is reached, the upcoming configuration is updated to the next configuration in the path. The Tracking Error (TE) is the difference between the current configuration of the system and the upcoming configuration.

The TE is defined as:

$$\epsilon^{track}(k) ::= ||c_{upcoming} - c(k)||$$

Where $c_{upcoming}$ is the target configuration in the path that the controller tries to steer toward, and $c(k)$ is the measured configuration.

The system should not diverge too far from to path it is supposed to track, if the robot diverges more than a pre-defined threshold the hgraph concludes that there was an error during execution and the edge fails. $c_{upcoming}$ does not update every time step, whilst $c(k)$ does update every time step. As a result, a "good" TE is expected to take the form of a saw tooth function inverted over the horizontal x-axis.

The predefined thresholds are split for drive and push actions because driving actions have much lower average PE and TE compared to push actions. For drive action edges, when the average of the last 25 recorded PE's is higher than 0.05 meter, or the TE is higher than 2 meters, a fault is concluded. For push actions only a TE is used, which is split into two parts. One ensures the object follows the path, and another ensures that the robot does not deviate too far from the object. If, for a pushing edge, the object deviates more than 2 meters from the path or the robot deviates more than 2 meters from its push position determined by the object pose, a fault is concluded.

4.3.3. The Blocklist

The blocklist prevents the regeneration of failed edges. The infinite loop of creating an edge that fails only to be regenerated is prevented. The blocklist keeps a list of edge parameterization as well as the node identifier if failed on. Newly generated edges are checked against this blocklist, if they are on the blocklist, initialization of the edge is prevented. The possible parameterizations are filtered when two nodes are connected with an action edge. Thus, any parameterization on the blocklist for a specific node (to which the action edge would point to) cannot be created again for the lifetime of the hgraph.

In the following example, Figure 4.11 faults are detected, these edges are added to the blocklist, the

first hypothesis fails to complete, and the halgorithm tries to generate a new hypothesis that also fails to complete.

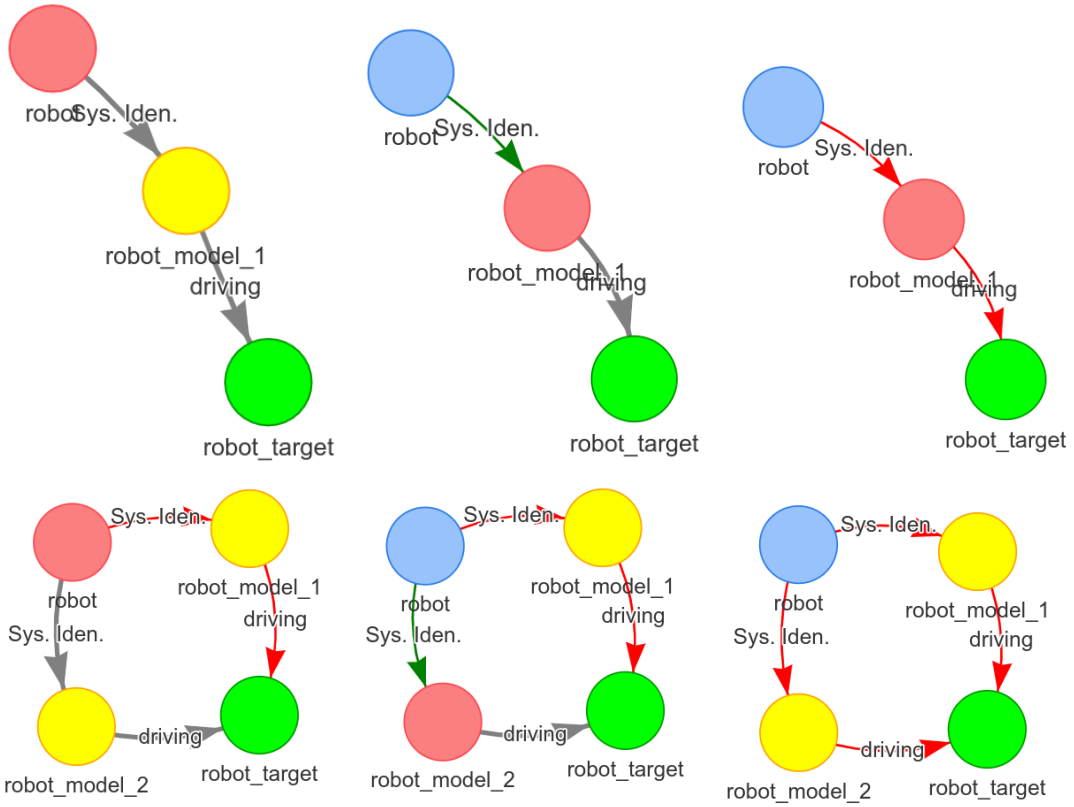


Figure 4.11: Multiple stages of the hgraph. The two hypotheses both failed during execution because a fault was detected. The failed edges are added to the blocklist, preventing the regeneration of edges with the same parameterization. The halgorithm concludes the task to be unfeasible.

In Figure 4.11, only two parameterizations of drive controllers and system models were available. Thus after two failed hypotheses, the halgorithm concludes that the task is unfeasible. All functionality is now discussed and is neatly summarized in the following table. Then, pseudocode for the proposed halgorithm is presented.

4.3.4. Encountering a Blocked Path

During the propagation of an action edge's status, path planning occurs discussed in Section 2.3.2 and chapter 3. A blocking object is detected when the path found crosses through unknown of movable space. An example is now discussed that elaborates upon the halgorithm response when encountering a blocked path. An example of such an environment can be visualized in Figure 3.1a where the *UnknownSpaceCost* is set to 0.5 meter. The example hgraph that will now be discussed generalizes over drive tasks that can be completed if an blocking object is pushed to free the path.

The halgorithm creates the start- and target node for the robot in Figure 4.12a, and creates a first hypothesis which can be visualized in Figure 4.12b. When propagating the drive action edge status, path planning occurs, during which a blocking object is detected that blocks a direct path. To free that path two nodes are generated, first, a target node for the blocking object *blocking_object_target* that target node indicates the blocking object at a pose that is no longer blocking the path. The second node represents the blocking object at its current pose, because there does not yet exist a node for the blocking object at its current pose, the halgorithm generates a new node for the blocking object at it's current

pose named *blocking_object* that can be seen in Figure 4.12c.

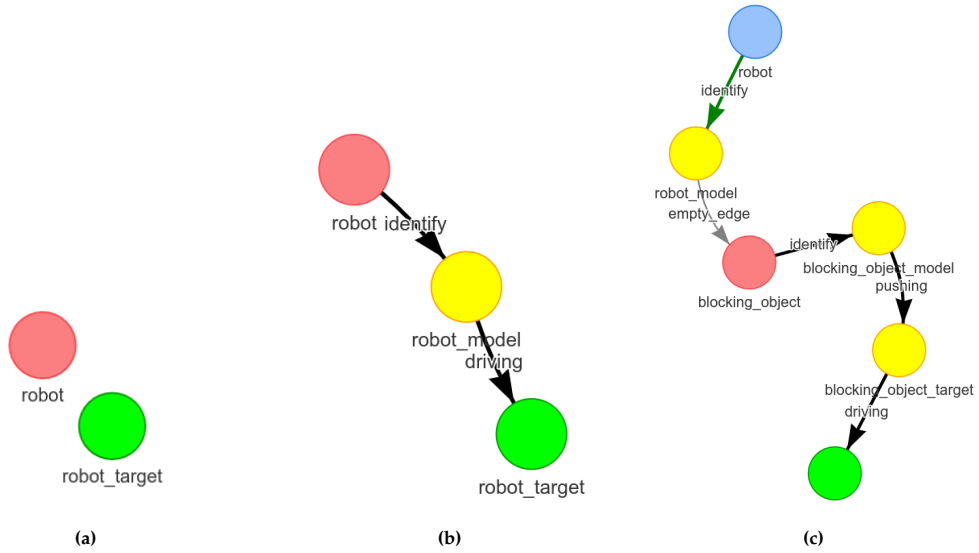


Figure 4.12: Multiple stages of the hgraph for a driving task, where an blocked path is encountered.

The newly generated pushing edge between node *blocking_object_model* and *blocking_object_target* is initialized with a randomly selected controller. To fully parameterize the pushing edge a controller with a compatible system model is required. To generate a system model, the halgorithm generates a pushing identification edge and node *blocking_object_model*.

After identifying a system model that describes pushing against the blocking object, path planning for the blocked object from its current pose toward it's target pose occurs. The initial pose for the robot against the blocked object can then be determined because that initial pose is dependent on the planned path. Figure 4.13 displays the drive edge toward the best initial push pose indicated by the *best_push_pose_against_blocking_object* node. Equivalent to the generation of a blocking object node at its current pose in Figure 4.8c, a robot node is generated at the current robot's pose, named *robot_copy* in Figure 4.13, because a node at the current robot pose did not yet exist. A drive edge is generated between the *robot_copy* and *best_push_pose_against_blocking_object* nodes. The identification edge is later generated as can be seen in Figure 4.14a.

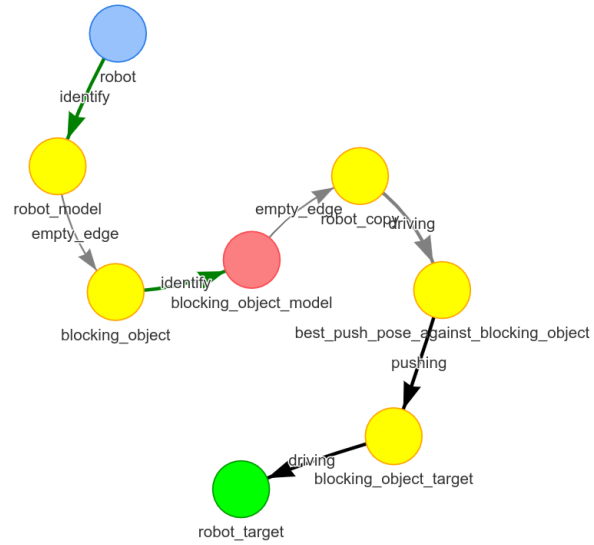


Figure 4.13: Snapshot of the hgraph during drive task, where a blocked path is encountered. The (red) current node indicates that next action is to drive toward the best push pose against the blocking object.

Figures 4.14a and 4.14b visualize driving toward the best push pose against the blocking object, pushing the blocked object to its target pose, and finally driving toward the robots target pose.

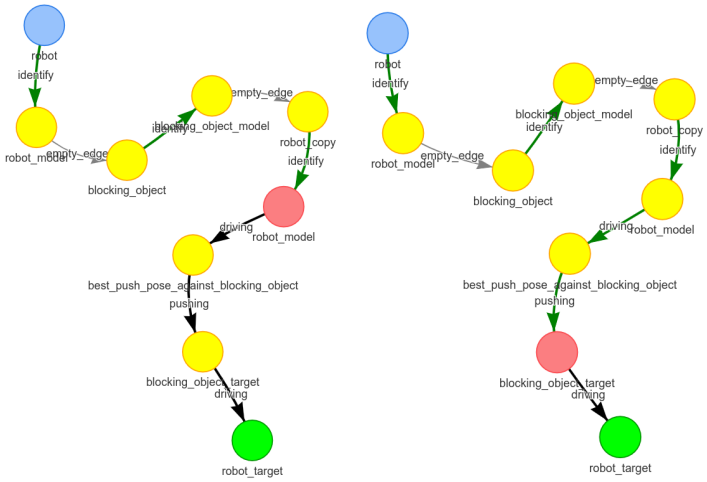


Figure 4.14: hgraph final stages before successfully completing a driving task, during which a blocked path is encountered.

Now that the halgorithm is defined and discussed, pseudocode is presented. The following table summarizes the functions that will be used by the pseudocode in Algorithm 6.

Table 4.3: The functions employed by the halgorithm in Algorithm 6.

SubTaskNotFinished(s):	Return False if the subtask s is completed or it is concluded to be unfeasible
IsConnected(v_1, v_2):	Return True if there exist a path in the hgraph from node v_1 to node v_2 through a number of non-failed edges
ReadyForExecution(e):	Return True if the edge e is ready to execute
TargetNotReached(e):	Return True edge e has not reached it target configuration
FaultDetected(e):	Return True if a fault has been detected during execution of edge e
HandleFault(e):	Update edge e status to FAILED and remove edge from hypothesis
SteerTowardTarget(ob):	Update controller with observation ob and compute response that steers the system to target configuration
ReadyForExecution(e):	Check if edge e has the PATH IS PLANNED status and contains all components to track the path
IncrementEdge:	Mark current edge as completed, set next edge in h as current edge if ready for execution, otherwise, enter search loop
MakeReady(e):	Perform actions (see Table 4.2 for detailed information) to make the edge e ready for execution
GoBackward(v):	Find the source node that point toward v through a number of non-failed edges
FindCorrespondingNode(v):	Find the node containing the same object as v
ConnectWithEdge(e_1, e_2):	Randomly generate edge between nodes v_1 and v_2 or use kgraph to suggest an edge

The pseudocode uses variables defined in Table 4.1, such as a subtask: s, task: S, hgraph: $G^{hypothesis}$, hypothesis: h, observation: ob and node: v.

Algorithm 6 Pseudocode for the proposed hypothesis algorithm.

```
for s ∈ S do
    while SubTaskNotFinished(s) do                                  ▷ Search Loop
        if G^hypothesis.IsConnected(s.start, s.target) then
            if h.CurrentEdge.ReadyForExecution then
                while TargetNotReached(h.CurrentEdge) do            ▷ Execution Loop
                    if FaultDetected(h.CurrentEdge) then
                        HandleFault(h.CurrentEdge)
                        break
                    end if
                    h.CurrentEdge.SteerTowardTarget(ob)
                    if TargetReached(h.CurrentEdge) then
                        if ReadyForExecution(h.CurrentEdge) then
                            h.IncrementEdge
                        else
                            break
                        end if
                    end if
                end while
            else
                MakeReady(h.CurrentEdge)
            end if
        else
            v_localtarget ← G^hypothesis.GoBackward(v.target)
            v_localstart ← G^hypothesis.FindCorrespondingNode(v_localtarget)
            G.ConnectWithEdge(v_localstart, v_localtarget)
        end if
    end while
end for
```

A flowchart of the halgorithm is presented in Figure 4.15. Compared to the pseudocode presented above, the flowchart provides more detail, especially in the elaborate description accompanying the flowchart in Table 4.4. The flowchart includes a connection point to the kgraph and robot environment. The blocks in the flowchart indicate the resources used and changes indicated in the legend. Compared to the flowchart, the pseudocode is an abstract version, leaving many details explicitly related to the robot used in this thesis. Pseudocode encompasses a broader field of robots. So can the pseudocode also be applied to a robot with manipulation abilities other than nonprehensile pushing.

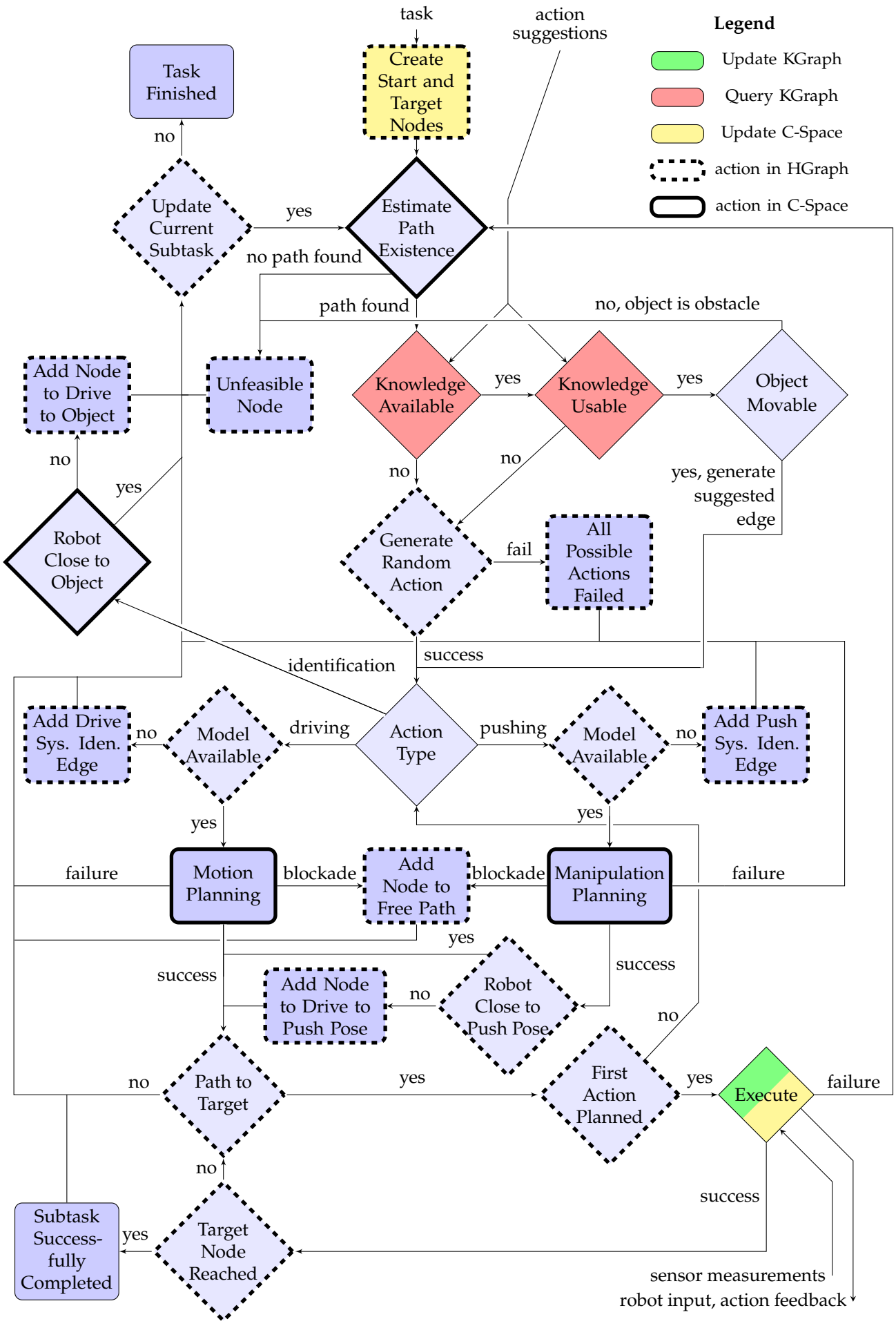


Figure 4.15: Flowchart displaying the halgorithm workflow.

Table 4.4: Comprehensive description regarding the actions executed by the blocks in Figure 4.15.

Node name	Description of actions taken
Task Finished	log all metrics for the hgraph, then deconstruct hgraph.
Create Start and Target Nodes	Generate a robot node and the start and target nodes for every subtask in the task.
Update Current Subtask	Select an unfinished subtask or update the current subtask. Use the backward search technique. The *current_start_node* and *current_target_node* are updated. When all subtasks have been addressed, conclude task is finished.
Estimate Path Existence	Check if a path exists between *current_start_node* and *current_target_node* whilst assuming that the object is holonomic.
Add Node to Drive to Object	Add a node before the *current_target_node*.
Unfeasible Node	Update node's status to unfeasible because it can not be completed, log failed Edge.
Knowledge Available	Query the kgraph for action suggestion to connect *current_target_node* to *current_target_node*
Knowledge Usable	Check if a suggested action is not on the blocklist.
Object Movable	Check if the object is classified as movable
Robot Close to Object	Check if the object is inside directly reachable free space of the robot
Generate Random Action	Randomly sample a controller with a compatible system identification method not on the blocklist.
All Possible Actions Failed	Every possible action is on the blocklist for the *current_target_node*, update *current_target_node* status to failed.
Add Drive System Identification Edge	Adds an identification edge between a newly generated node and the drive action edge's source node.
Model Available	Checks if the drive action edge contains a system model.
Action Type	Checks the action type.
Model Available	Check if the push action edge contains a system model.
Add Push System Identification Edge	Adds identification edge compatible with push action edge.
Motion Planning	Search a path for the *current_edge*, detect blocking objects.
Add Node to Free Path	Search close by pose for an object to free the path. Create a node to push the object toward that pose.
Manipulation Planning	Search a path for the *current_edge*, detect blocking objects.
Add Node to Drive to Push Pose	Create node to drive toward push pose, add before action edge.
Robot Close to Push Pose	Check if the robot overlaps the best push pose.
Path to Target	Is there a path from robot to target node in the hgraph, then set the first edge to *current_edge* otherwise update subtask.
First Action Planned	Check if motion/manipulation planning was performed.
Execute	Execute the *current_edge*, update hgraph after completion, log failed hypothesis if a fault is detected.
Subtask Successfully Completed	Log hypothesis metrics.
Target Node Reached	Check if the target node is reached.

The halgorithm is now discussed with a description, examples, pseudocode and a flowchart. In the next section, the executed edges will be reviewed and stored in a knowledge base named the knowledge graph (kgraph).

4.4. Knowledge Graph

The hgraph, discussed in the previous section, has a lifetime that spans over a single task. In the hgraph only the object class is stored. When a task is completed, the hgraph is deconstructed and learned objects classes are lost. The kgraph's responsibility is to store objects classes for future tasks, and to collect action feedback, to then later suggest edge parameterizations based on the action feedback that is stored. An edge paremeterization comprises of a controller and system model. Estimating which parameterization would be the best candidate is an entire field of research. In this thesis the ordering is made by collecting action feedback on executed action edges and summarizing that feedback in a single metric, the *success factor*. This metric combines the Prediction Error, the number of failed edges and the number of successfully completed edges.

The name "knowledge graph" originates from the environmental knowledge it contains and its graph structure. Both the hgraph and kgraph are newly proposed frameworks built from the ground up, with only inspiration from an already existing technique, a backward search. The kgraph is in an early stage of development and does therefore not (yet) adhere to any standard that apply to knowledge bases, such as first order-logic [4, 32], relational databases [3] or more practical, a language such as prologe [49].

4.4.1. Definition

The kgraph consists of center nodes, side nodes and edges. A center node's responsibility is to represent an object and its classification.

Formally, a **center node**, $v_{id}^{center} = \langle id, id_{obj}, \text{OBJ\_CLASS} \rangle$

Where id an identifier for the center node, id_{obj} an identifier for the object
OBJ_CLASS the classification of that object which can be either MOVABLE or UNMOVABLE.

A side node is a placeholder for the edge to point toward.

Formally, a **side node**, $v_{id}^{side} = \langle id \rangle$, Where id an identifier for the side node.

An edge points from center- to side node and has two responsibilities. First, it stores the parameterization used to manipulate an object that resides in the center node. Second, it stores the performance of that parameterization which is summarized in the success factor.

Formally an **edge**, $e_{(from,to)} = \langle id_{from}, id_{to}, \alpha(a), \text{System Model}, \text{Controller} \rangle$

Where id_{from} and id_{to} corresond the the identifiers of the center- and side node respectively, $\alpha(a)$ is the success factor that summarized the feedback of $a \geq 0$ reviewed action edges and (System Model, Controller) together is referred to as edge parameterization.

The successfactor rates an edge parameterization based on experience in the robot environment. The successfactor is created or updated when an action edge with that parameterization failed during execution time or successfully completed.

Formally the **success factor** = α:

$$\alpha(a+1) = \begin{cases} 0.1^{\epsilon^{pred}{}_{\text{avg}}} & \text{if } a = 0 \\ 0.1 + 0.9\alpha(a) & \text{if } a > 0 \text{ and the reviewed edge was successfully completed} \\ 0.9\alpha(a) & \text{if } a > 0 \text{ and the reviewed edge failed during execution time} \end{cases}$$

Where $\epsilon^{pred}{}_{\text{avg}}$ is the average PE of the action edge that is reviewed. a is the number of times an action edge with such a edge parameterization reached the EXECUTING status.

Now that the nodes and edges have been defined, the kgraph can be defined.

Formally, a **knowledge graph**, $G^{knowledge} = \langle V_K, E_K \rangle$
comprising $V_K = \{v^{center}, v^{side}\}, \quad E_K \in \{e_{(i,j)} | i \in V_{K_{ids}}^{center}, j \in V_{K_{ids}}^{side}\}$.

Where $V_{K_{ids}}^{center}$ are the identifiers of the set of center nodes, and $V_{K_{ids}}^{side}$ are the identifiers of the set of side nodes.

4.4.2. The Testing and Converging Phases

When the halgorithm determines is must manipulate an object to fullfill the given task, it creates a action edge with a random edge parameterization. After executing that action edge that parameterization is stored in the kgraph. When encountering the need to manipulate the object again a different random parameterization is selected. First, all possible edge parameterizations are tested and stored on a specific object, this can be seen as the testing phase on how to manipulate an object. Then second, when all possible parameterizations are tested, the kgraph suggests the edge parameterization with the highest success factor, this phase can be seen as the converging phase. Where either, the edge parameterization with the highest success factor receives an even higher success factor, or the success factor is lowered, and the second best edge parameterization becomes the highest ranking edge parameterization in terms of success factor.

The kgraph has three interface functions. The *add_object(obj, OBJ_CLASS)* function adds an object and its class to the kgraph. Espacially usefull to indicate which objects are unmovable. The *add_review(e)* function takes an action edge from the hgraph and depending on the kgraph does the following. If a corresponding center node (containing the object the edge manipulated) does not yet exist, create a new center node. Then calculate the success factor and add a new edge or update the success factor of the existing edge with equal edge parameterization in the kgraph.

or update the the edge that containts the edge parameterization. If the edge parameterization exist in one of the outgoing edges from the center node, existthe of the corersponding object if it does not exist. creates or updates the corresponding edge the kgraph is updated with a new success factor as described in the formula above. The *action_suggestion* returns the best parameterization it contains for an object.

4.4.3. Legend and Example

A legend for the kgraph is now presented.

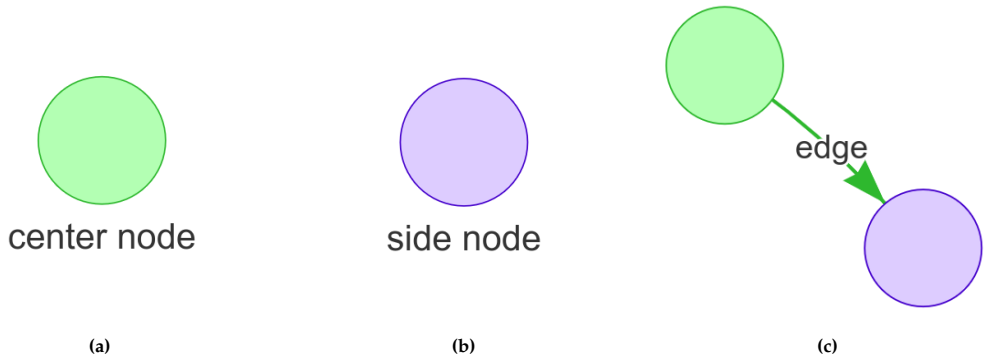


Figure 4.16: Legend for the knowledge graph.

An example kgraph can be visualized in Figure 4.17, where edge parameterizations are displayed over the edges.

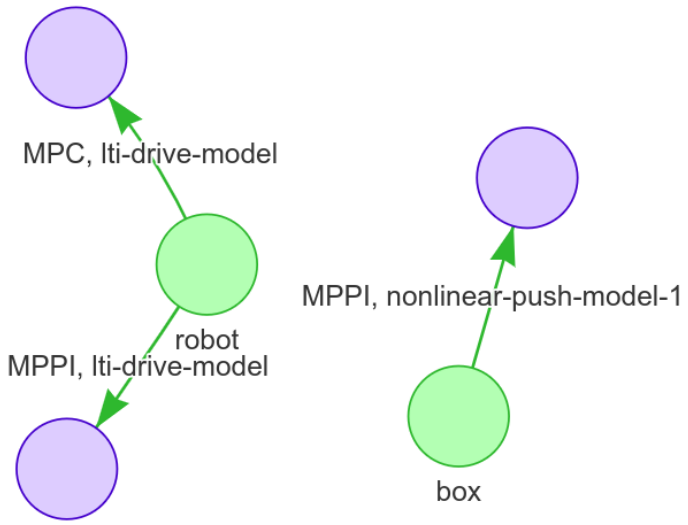


Figure 4.17: The kgraph that has collected action feedback on two objects. Two edge parameterizations for driving the robot and one edge parameterization to push the box.

Figure 4.18: Flowchart displaying the knowledge graph's workflow.

What by now hopefully became clear to the reader is that the halgorithm autonomously searches for hypotheses in the hgraph to solve a task, one subtask at a time. The halgorithm switches between the search and execution loop. Switching from the search loop toward the execution loop when a hypothesis is found and switching back when a hypothesis is completed, or a fault is detected.

The limited number of possible edge parameterization (every combination of a system identification method with a compatible control method) guarantees that the robot tries to complete a subtask. However, it concludes that it cannot complete a subtask if all possible edges have failed.

This thesis proposes to combine the three topics (one, learning object dynamics, two, the NAMO problem, and three, nonprehensile push manipulation to target pose). The halgorithm can solve NAMO problems because the robot can drive toward target poses even if reaching such a pose requires objects to be moved first. The proposed algorithm learns to classify objects by updating the object's class from unknown to movable or unmovable. The halgorithm can push objects to target poses by identifying a system model and then pushing the object toward its target pose. However, the system model that

system identification yields is of short use because it is only given to the corresponding action edge.

5

Results

This chapter presents various robot environments and tasks that challenge the proposed framework. The test results provide evidence that supports the answers to the research questions. The chapter starts by presenting metrics that measure task performance, the simulation environment is introduced, then a number of system models is presented and tuning variables are specified. With the system models defined and tuning parameters set test are presented, a randomized environment that is split in task that mainly involve driving and tasks that mainly involve pushing. The chapter finishes with a comparison with the state-of-the-art in Section 5.2.

Metrics to Measure Task Performance The proposed frameworks task performance is measured in metrics, these include the PE, search-, execute- and total time to complete a task. We can draw conclusions from the evolution of the metrics over time. Furthermore, the method metrics will be used to compare the proposed framework to state-of-the-art. Now, the method metrics are presented in Table 5.1.

Table 5.1: Method metrics employed to measure task performance by the proposed robot framework in the left column. Motivation on relevance of the metric is provided in the corresponding right column.

Prediction Error [meter]	The PE is high in two scenario's, first when unexpected behaviour occurs, second when the system model used to calculate the PE describes the system poorly. When the PE is low, that would indicate the robot encounters expected behaviour and that the model describes the system accurate. If the PE is lowering during task execution over time, that would indicating the robot is learning to select models that are more accurate and less unexpected behavior occurs.
search time [sec]	The time the proposed framework spends searching for hypotheses. The search time is dominated by path planning because compared to path estimation, updating the hgraph it is time consuming. The number of subtasks in a task, the environment and the number of unknown and classified objects influence the number of paths to be planned and the duration of planning, and thus the search time.
execution time [sec]	The time the proposed framework spends executing hypotheses. The execution time is dominated by path length, driving or pushing toward a target pose on the other side of the workspace takes more time compare to a target pose closeby the initial pose.
total time [sec]	search time + execution time.

The Simulation Environment Testing in a simulation environment has been done using the URDF Gym Environment [40], a 100% python environment build upon the PyBullet library [9]. The code created during the thesis can be found on GitLab and GitHub. Experiments are taken on laptop with specifications: Laptop: HP ZBook Studio x360 G5, OS: Ubuntu 22.04.1 LTS x86_64, CPU: Intel i7-8750H (12) @ 4.100GHz, GPU: NVIDIA Quadro P1000 Mobile. The simulation environment provides many different robots, of which one robot is selected to perform tests, the point robot, which is are displayed in Figure 1.1. The point robot takes an velocity input along the x- and in y-axis.

Robot input is defined as:

$$u(k) = [u_x(k), u_y(k)]^\top$$

During testing three controllers have been used. A MPC and MPPI controller for driving the robot, and a MPPI controller for robot pushing. The MPC and MPPI control methods were introduced in Section 2.2. Three system models are implemented, a Linear Time-Invariant (LTI) model describing robot driving, and two nonlinear model describing the robot pushing an object. First a short textual description of the available system models is provided below. Second, the state-space model is provided for the three implemented models.

Table 5.2: The left column displays the available models of drive- and push systems, accompanied by a description in the corresponding right column.

lti-drive-model	A second order LTI model that is compatible with both the MPC and the MPPI drive controller. The state variables are the robot's x and y position. The next state $x(k+1)$ is based on the current state $x(k)$, the time step Δk and robot system input $u(k)$.
nonlinear-push-model-1	A nonlinear model that is compatible with only the MPPI push controller. The state variables are the robot's and the pushed object's x and y position. The next state $x(k+1)$ is based on the current state $x(k)$, the time step Δk and robot system input $u(k)$.
nonlinear-push-model-2	A nonlinear model that is compatible with only the MPPI push controller. The state variables are the robot's and the pushed object's x and y position and the objects orientation θ_{obj}. The next state $x(k+1)$ is based on the current state $x(k)$, the time step Δk, robot system input $u(k)$ and geometrical variables v_p and s_t.

State-space representation of the *lti-drive-model*:

$$x_{lti-drive-model}(k+1) = \begin{bmatrix} x_{robot}(k+1) \\ y_{robot}(k+1) \end{bmatrix} = \begin{bmatrix} x_{robot}(k) + \Delta k u_x(k) \\ y_{robot}(k) + \Delta k u_y(k) \end{bmatrix} \tag{5.1}$$

State-space representation of the *nonlinear-push-model-1*:

$$x_{nonlinear-push-model-1}(k+1) = \begin{bmatrix} x_{robot}(k+1) \\ y_{robot}(k+1) \\ x_{obj}(k+1) \\ y_{obj}(k+1) \end{bmatrix} = \begin{bmatrix} x_{robot}(k+1) + \Delta k u_x(k) \\ y_{robot}(k+1) + \Delta k u_y(k) \\ x_{obj}(k+1) + \frac{1}{2}\Delta k u_x(k) \\ y_{obj}(k+1) + \frac{1}{2}\Delta k u_y(k) \end{bmatrix} \tag{5.2}$$

State-space representation of the *nonlinear-push-model-2*:

$$x_{nonlinear-push-model-2}(k+1) = \begin{bmatrix} x_{robot}(k+1) \\ y_{robot}(k+1) \\ x_{obj}(k+1) \\ y_{obj}(k+1) \\ \theta_{obj}(k+1) \end{bmatrix} = \begin{bmatrix} x_{robot}(k) + \Delta k u_x(k) \\ y_{robot}(k) + \Delta k u_y(k) \\ x_{obj}(k) + \Delta k \cos(\theta_{obj}(k))(1 - |\frac{2s_t}{H}|)v_p \\ y_{obj}(k) + \Delta k \sin(\theta_{obj}(k))(1 - |\frac{2s_t}{H}|)v_p \\ \theta_{obj}(k) + \frac{2*\Delta k * v_p * s_t}{H} \end{bmatrix} \tag{5.3}$$

Where distance s_t and velocity v_p can be visually seen in Figure 5.1. A positive s_t indicates the object will rotate anticlockwise, a negative s_t indicates a clockwise rotation. The width of an object is defined as H and is set to 2 meters.

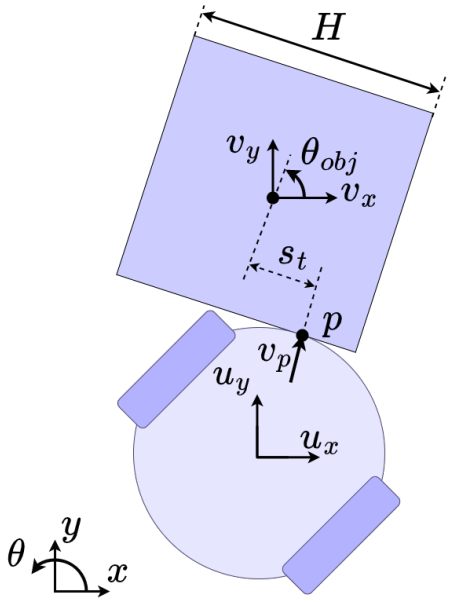


Figure 5.1: Schematic overview of point robot pushing a box object.

Where v_p is the velocity of the robot perpendicular to the object defined as:

$$v_p = u_x \cos(\theta_{obj}(k)) + u_y \sin(\theta_{obj}(k))$$

Two drive action edge parameterizations are available: (MPC, *lti-drive-model*) and (MPPI, *lti-drive-model*), and two push action edge parameterizations are available: (MPPI, *nonlinear-push-model-1*) and (MPPI, *nonlinear-push-model-2*).

The goal of this thesis is not to find optimal control, or to model the environment with great accuracy. The goal is to select the best edge parameterization in the available set of edge parameterizations. Eleborating the control design and system model design is therefore out of the scope of this thesis, the control design parameters such as prediction horizon, cost function, number of rollouts can be found in the implementation.

5.1. Driving and Pushing Experiments

To improve the relevance of the experimental results, driving and pushing tasks are performed with randomized settings. A set of multiple tasks is given to the robot, named a *run*. When the robot is tasked with the first task in the run the kgraph is empty, and when the last task in the run is completed

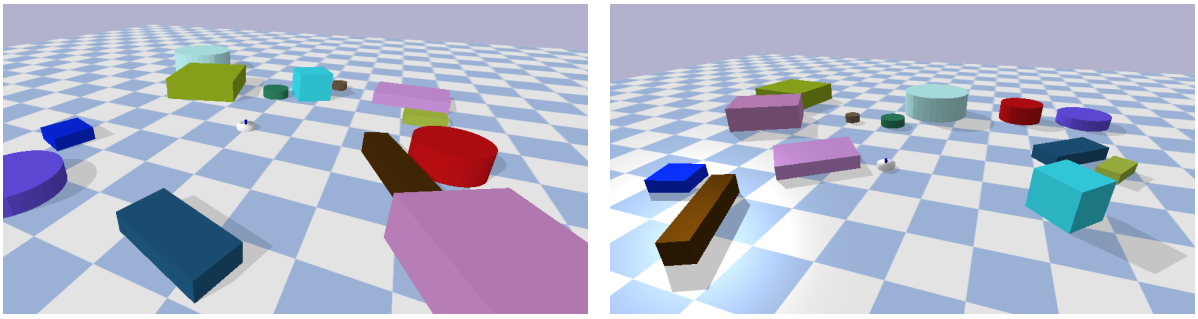
the kgraph is filled with the experience gained. For each run, a set of random objects and properties are generated. This remains the same for the whole run, however for each task in the run the environment is reshuffled. A reshuffled does three things, first, it resets the robot pose, secondly, it gives the objects in environment new initial pose. Thirdly, new target poses are generated for the subtask in the new task. By solving multiple reshuffled tasks in a reshuffled environment, the robot gains experience that is stored in the kgraph.

Table 5.3 presents a set of parameters that initializes the random environment. Two type of tasks solved by the robot, a driving task, where the robot must drive toward multiple random target poses, and a pushing task, where the robot must push a movable object toward a random target pose.

Table 5.3: The tuning parameters to initialize a random environment

The *size of the grid*	length and width of the ground plane in x and y direction.
The *minimal and maximal size of objects*	A box will have sides with a length that lie in the specified range from minimal to maximal length. Cylinders will have a diameter and height that is within the specified range, additionally, cylinders are not higher than the radius of the cylinder to prevent cylinders from tipping over.
The *maximal weight*	which is uniformly distributed for the environment objects, minimal weight is set by default to 1 kilogram.
The *number of unmovable objects*	Specify the amount of unmovable objects.
The *number of movable objects*	Specify the amount of movable objects.
The *number of subtasks in a task*	Specify the amount of subtasks in a task.
The *number of boxes and cylinders*	For every new object generated there is a 50% chance it becomes a box and 50% chance it becomes a cylinder.

An randomly initialized environment that is then reshuffled can be visualized in Figure 5.2.



(a) Environment after initialization. **(b)** Environment after a reshuffle.

Figure 5.2: A random environment initialized by tuning parameters presented in Table 5.4 and randomness.

5.1.1. A Driving Task

For the driving task the random environment is created with the following tuning parameters.

Table 5.4: The selected tuning parameters for the randomized drive environment.

grid size	x=12 m, y=12 m
object size	$min\_length = 0.2m$, $max\_length = 2m$
object weight	$max\_weight = 1000kg$
number of objects	$num\_unmovable\_obj = 3$, $num\_movable\_obj = 5$
number of tested runs	$num\_runs = 10$
number of tasks in a run	$num\_tasks = 10$
number of subtasks in a task	$num\_subtasks = 3$

These parameters have been specifically selected, starting with the size of the ground floor. The ground floor should be large enough such that objects can be pushed around, note that, for a driving task, pushing is involved when a path must be freed. An enormous (100 by 100 meter) ground floor would result in a longer computational time for path planning, which is undesired. A 12 by 12 meter ground floor is selected because the floor is large enough for objects to be pushed around. The range that determines the size of objects is set such that objects can be as large as the robot itself, and be around 10 times as large as the robot. With these sizes the robot is unable to grasp objects, a gripper would be too small to grasp objects. The comparatively large size fits the objective of nonprehensile pushing, there simply is no other method to manipulate such large objects other than pushing. A real-life example are can be found in harbours where tug boats push giant cargo ships around that are many times over the size of the tug boat. The ratio of solid obstacles vs. movable objects determines if a task is more navigation (only solid obstacles) or more NAMO (only movable objects). A task that tends toward NAMO is favoured because that is the target environment in this thesis. There should be some unmovable obstacles that reward the robot learning such objects are unmovable (to then not interact with them). Thus there are more movable objects than solid obstacles chosen, whilst still having 2 solid obstacles around. Ten runs are taken, each run consisting of 10 tasks, the results are averaged to reach statistic relevance in a randomized environments. The number of subtasks is set to 3, a low number of drive subtasks that can be completed in under 2 minutes.

Lastly, a number of tuning parameters must be set for the halgorithm. These are the maximal robot speed, set to 1 $\frac{meter}{s}$, the *cell size* for the path estimator set to 0.1 meter, the path planner takes four tuning parameters; the *step size* set to 0.2 meter, the *search size* set to 0.35 meter, the *MovableSpaceCost* set to 2 meter and the *UnknownSpaceCost* set to 3.5 meter.

Given the above parameters, all parameters are now set. Ten runs are solved by the robot where every run is composed of ten tasks. The first generated environment in the run determines the type of objects and their properties that remain unchanged for the entire run. The subsequent environments in the run are reshuffled versions of the first environment. After ten runs the task performance is collected that is expressed in the metrics just described. The ten runs metrics are vertically augmented such that they are grouped by the same number of tasks in experience. First, a boxplot is presented displaying execution-, search- and total times over the ten runs whilst using kgraph action suggestions in Figure 5.3. The number of solved subtasks can be calculated by multiplying the number of runs times the number task in a run by the number of subtasks in a task, which is $10 \cdot 10 \cdot 3 = 300$ subtasks.

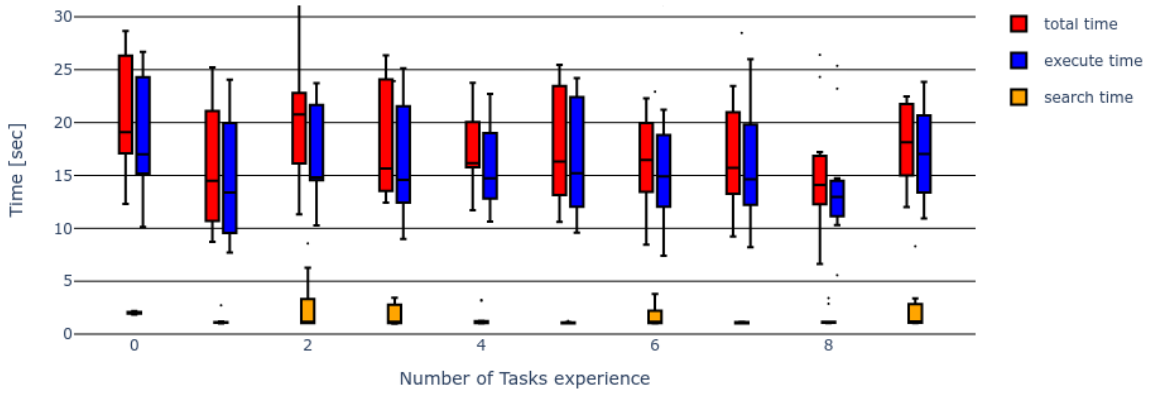


Figure 5.3: Search-, execution- and total time to complete a drive task **whilst** using kgraph action suggestions. The horizontal axis indicates the number of task experience in a run. The vertical axis displays a boxplot of the search-, execution- and total time over ten runs grouped by number of tasks in experience.

The above figure displays results where the halgorithm initially selects a random edge parameterization, after executing an action edge it sends action feedback to the kgraph. When every possible edge parameterization for drive actions has corresponding action feedback in the kgraph, the kgraph starts suggesting edge parameterizations. Now a boxplot is presented in Figure 5.4 that displays task execution without using the kgraph action suggestions, instead a random parameterization is selected for every action edge. By fixing a seed the environments used for Figure 5.3 are equal to the environments used for Figure 5.4.

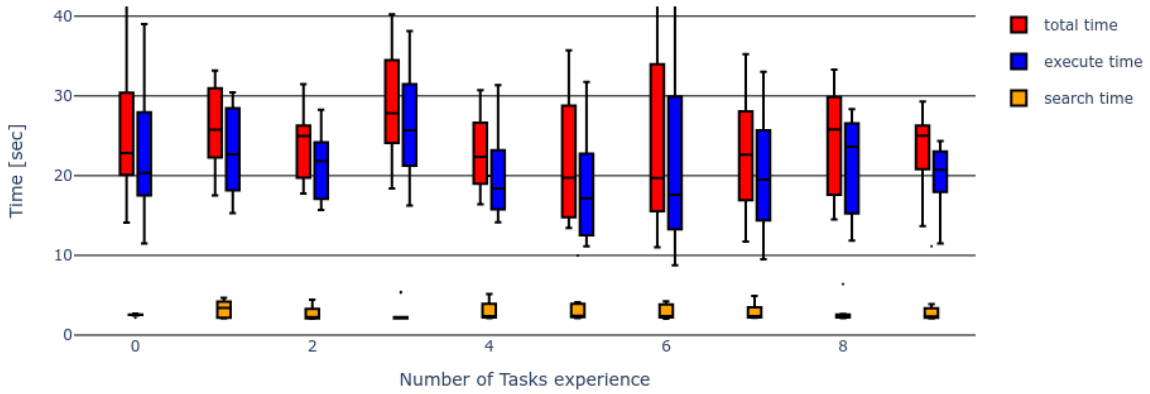


Figure 5.4: Search-, execution- and total time to complete a drive task by randomly selecting edge parameterizations. The horizontal axis indicates the number of task experience in a run. The vertical axis displays a boxplot of the search-, execution- and total time over ten runs grouped by number of tasks in experience.

Lets compare the means from Figures 5.3 and 5.4 and compare task execution with and without the kgraph action suggestions. However overall Figure 5.3 shows significant improvement over Figure 5.4 which becomes better visible if only the means are compared in Figure 5.5.

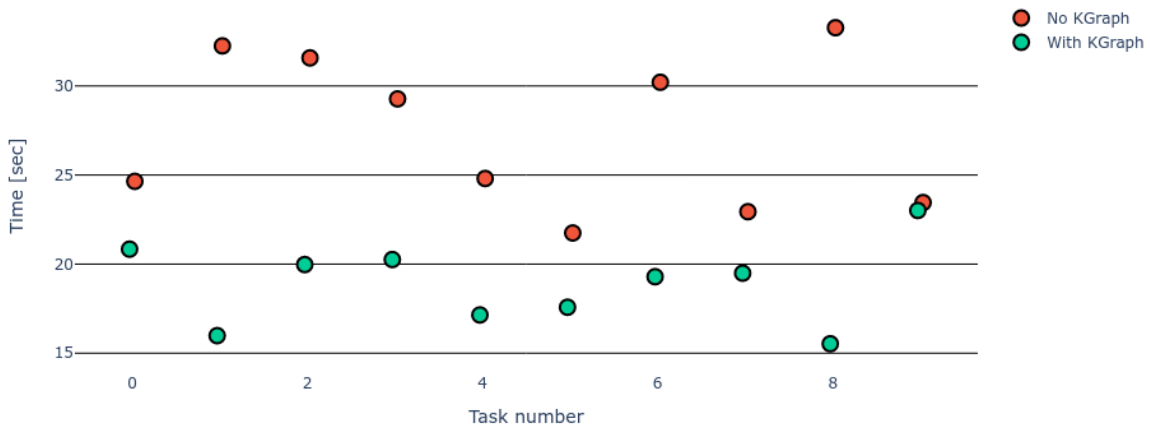


Figure 5.5: Comparing average total time to complete a task out of then runs for a driving task. For the edge parameterizations once with the use of kgraph action suggestions are leveraged indicated by "with kgraph" and once with random selection indicated by "no kgraph".

Two driving edge parameterizations are available, the MPC and the MPPI parameterization that both use lti-drive-model. The kgraph prefers the MPC parameterization over the MPPI parameterization as can be seen in Table 5.5. In this table is can be seen that the kgraph needs at most one task of experience to suggest the MPC parameterization. An explanation why there was no trend to be found in Figure 5.3, it was already converged at the first task.

Table 5.5: The selection of the (MPC, *lti-drive-model*) parameterization versus selecting the (MPPI, *lti-drive-model*) parameterization for drive actions during the randomized driving tasks. The leftmost column indicates that tasks execution is performed with the kgraph action suggestions, and without action suggestions.

Number of Tasks in experience		0	1	2	3	4	5	6	7	8	9
With kgraph suggestions	Number of MPC parameterizations	22	33	34	33	33	33	33	34	33	32
	Number of MPPI parameterizations	11	0	0	0	0	0	0	0	0	0
	MPC selected in total drive actions [%]	67	100	100	100	100	100	100	100	100	100
	MPPI selected in total drive actions [%]	33	0	0	0	0	0	0	0	0	0
Without kgraph suggestions	Number of MPC parameterizations	14	14	12	12	17	16	17	19	16	11
	Number of MPPI parameterizations	19	19	21	21	17	17	18	14	17	22
	MPC selected from total drive actions [%]	42	42	36	36	50	48	49	58	48	33
	MPPI selected from total drive actions [%]	58	58	64	64	50	52	51	42	52	67

The halgorithm successfully completes 300 driving subtasks (3 subtasks per task, 10 tasks per run, 10 runs are completed). Once while storing edge feedback in the kgraph that suggest edge parameterization,

and once without kgraph. In Table 5.5 the number of MPC and MPPI parameterizations should be more than 30 for any number of tasks in experience. In many cases it is more than 30, which indicates more than 30 drive edges were created to complete 30 drive subtasks. A fault has been detected that terminates the execution of an edge, to complete the task, a new edge is created, resulting in more than 30 MPC and MPPI edge parameterizations. When comparing both a positive effect is measured of the use of the kgraph on the total task required to complete a task compared to random selection of edge parameterization. Now a task is taken that compares the use of the halgorithm with and without kgraph for a pushing task.

put this here: First the search- and execution time of a task are investigated, and its development over multiple tasks. Then a comparison is made between solving and reshuffling multiple tasks once with the kgraph that suggests edge parameterizations, and once without suggestions, by randomly selecting a available edge parameterization.

5.1.2. A Pushing Task

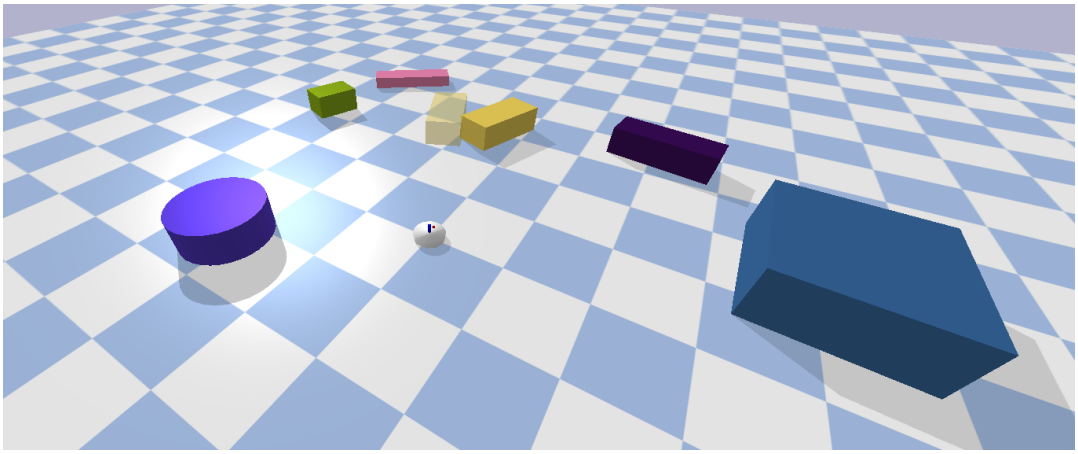
The push task in the randomized environment consists of a single subtask. To complete this push task the robot must push the object toward its specified target pose. The tuning parameters that make up the random environment for the push task can be visualized in Table 5.6.

This test of yours, 10 is the magic number does Winters support that claim?G

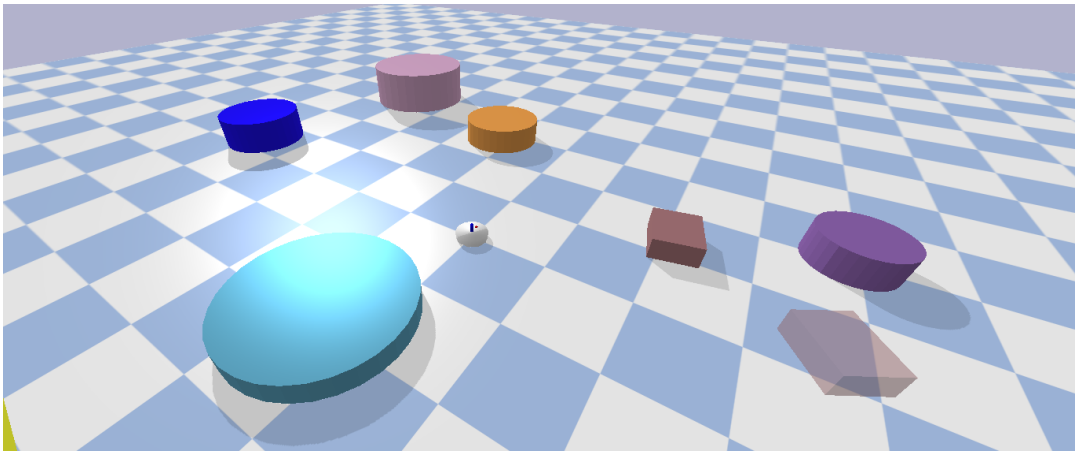
[10] can this <- winter guy add something to my conclusions, especially because it does have outliers wich are far out.

Table 5.6: The selected tuning parameters for the randomized push environment.

grid size	x=12 m, y=12 m
object size	$min\_length = 0.2m$, $max\_length = 2m$
object weight	$max\_weight = 1000g = 1kg$
number of objects	$num\_unmovable\_obj = 3$, $num\_movable\_obj = 5$
number of tested runs	$num\_runs = 10$
number of tasks in a run	$num\_tasks = 6$
number of subtasks in a task	$num\_subtasks = 1$



(a)



(b)

Figure 5.6: Two random environments where the pushing task contains a single subtask displayed by a target ghost pose.

All tuning parameters are set up, now the results are presented, first the pushing task is completed using kgraph suggestions. The search-, execute- and total time for task completion is presented in Figure 5.7. Then the same tasks are completed without help of the kgraph suggestions in Figure 5.4.

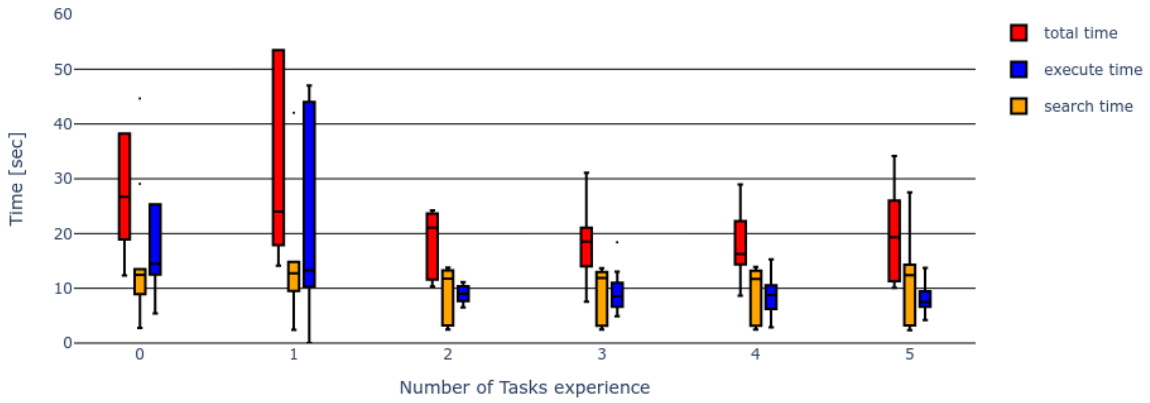


Figure 5.7: Search-, execution- and total time to complete a pushing task **with** kgraph action suggestions. The horizontal axis indicates the number of task experience in a run. A run contains ten tasks, starting the run with an empty kgraph that collects action feedback as the robot gains experience. The vertical axis displays a boxplot of the search-, execution- and total time over ten runs, where the sum of search- and execution time equals total time.

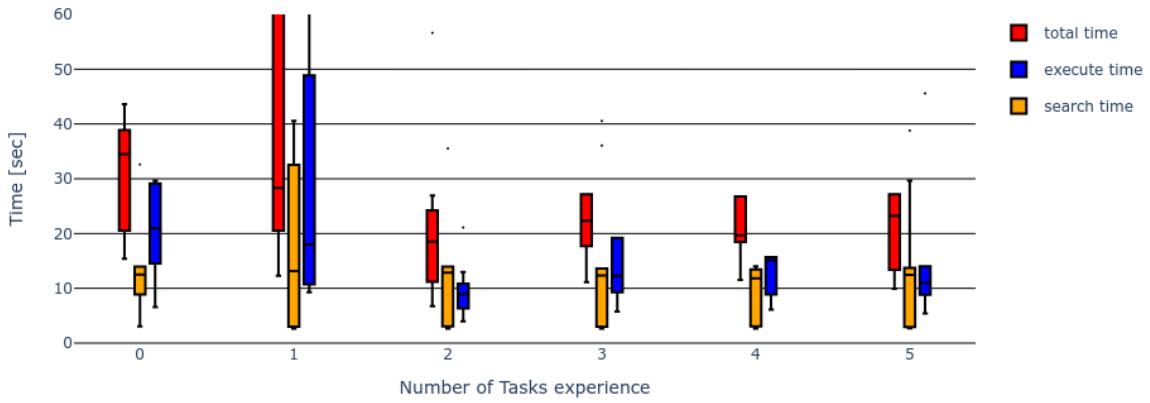


Figure 5.8: Search-, execution- and total time to complete a pushing task **without** kgraph action suggestions. The horizontal axis indicates the number of task experience in a run. A run contains ten tasks, starting the run with an empty kgraph that collects action feedback as the robot gains experience. The vertical axis displays a boxplot of the search-, execution- and total time over ten runs, where the sum of search- and execution time equals total time.

Both Figure 5.7 and Figure 5.8 look very similar due to solving the same tasks. Whilst gaining more experience the halgorithm that uses the kgraph suggestions yields a better total task completion time. Which is easier to see if only the mean of the total tasks times are plotted in Figure 5.9.

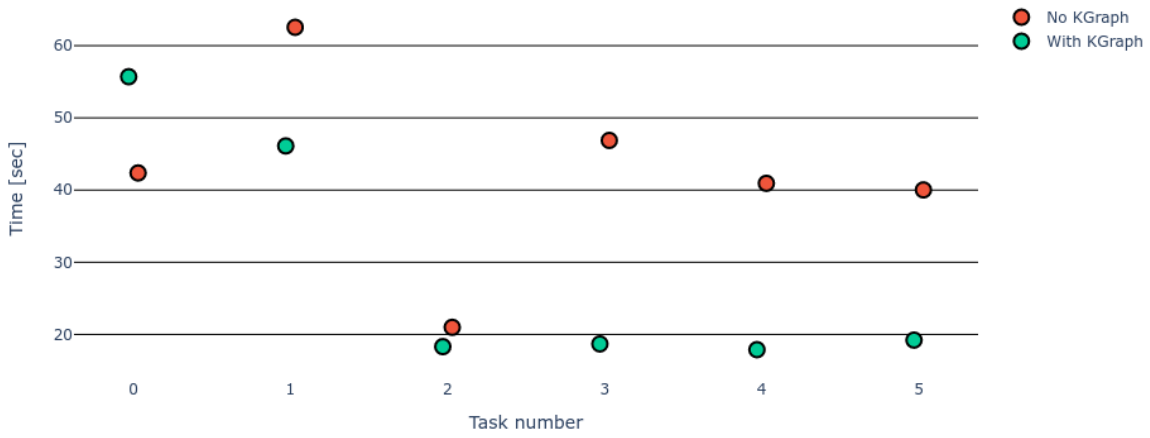


Figure 5.9: Comparing average total time to complete a task out of then runs for a pushing task. For the edge parameterizations once with the use of kgraph action suggestions are leveraged indicated by "with kgraph" and once with random selection indicated by "no kgraph".

Table 5.7: The selection of the (MPPI, *nonlinear-push-model-1*) parameterization versus selecting the (MPPI, *nonlinear-push-model-2*) parameterization for push actions during the randomized pushing tasks. The leftmost column indicates that tasks execution is performed with the kgraph action suggestions, and without action suggestions.

Number of Tasks in experience		0	1	2	3	4	5
With kgraph suggestions	Number of *nonlinear-push-model-1* parameterizations	4	5	8	10	10	10
	Number of *nonlinear-push-model-2* parameterizations	5	4	2	0	0	0
	nonlinear-push-model-1 selected in total push actions [%]	44	56	80	100	100	100
	nonlinear-push-model-2 selected in total push actions [%]	56	44	20	0	0	0
Without kgraph suggestions	Number of *nonlinear-push-model-1* parameterizations	4	3	7	5	4	5
	Number of *nonlinear-push-model-2* parameterizations	5	5	3	5	6	4
	nonlinear-push-model-1 selected in total push actions [%]	44	38	70	50	40	56
	nonlinear-push-model-2 selected in total push actions [%]	56	62	30	50	60	44

The kgraph favours the MPPI controller with *nonlinear-push-model-1* as can be seen in Table 5.7. The nonlinear-push-model-1 parameterization does not only have a lower execution time compared to the nonlinear-push-model-2 parameterization, it also has a lower PE as can be seen in Figure 5.10.

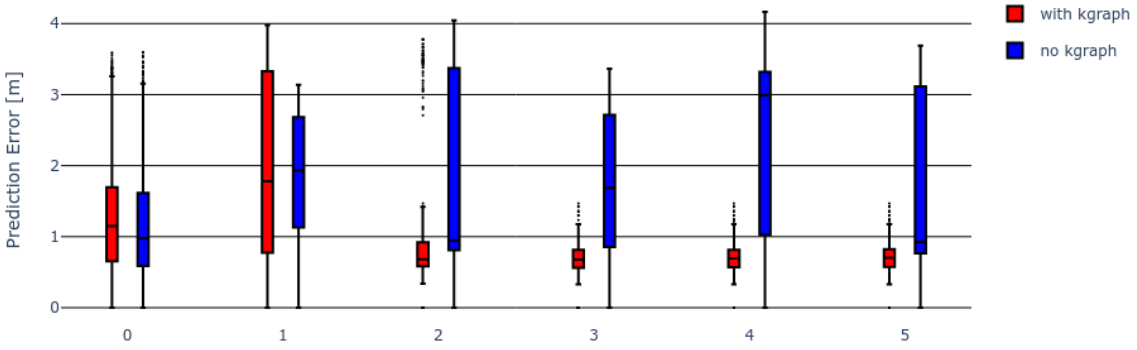


Figure 5.10: Box plot of the Prediction Error for solving reshuffled tasks. The horizontal axis indicates the number of task experience in a run. A run contains six tasks, starting the run with an empty kgraph that collects action feedback as the robot gains experience. The vertical axis displays a boxplot of Prediction Error over six runs. For the edge parameterizations once with the use of kgraph action suggestions are leveraged indicated by "with kgraph" and once with random selection indicated by "no kgraph".

The proposed framework has been compared against itself, it has shown that learning improves the method metrics by a notable margin as can be seen in Figure 5.10. Now the proposed framework is compared to the state-of-the-art in the upcoming section.

5.2. Comparison with State-of-the-Art

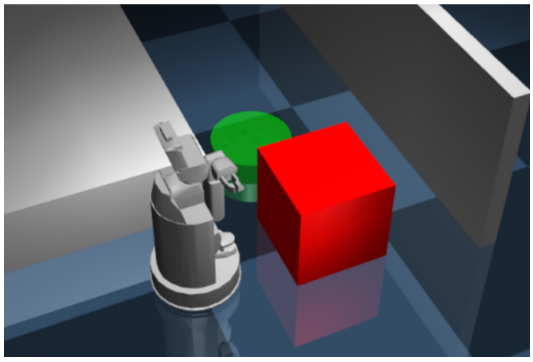
In the introduction Table 1.1 was presented. That table presents state-of-the-art methods and the subset of the three main topics that they include (the three topics; learning system models, the NAMO problem and nonprehesile pushing). Now that table presented again in Table 5.8, an additional column is augmented to the table on the right side. This extra column indicates the testing metric that state-of-the-arts uses and underlines the metric that is compared between the proposed framework and the state-of-the-art.

Table 5.8: Overview of recent state-of-the-art papers that include a subset of the 3 topics (learning system models, NAMO, and nonprehensile pushing). The method metric indicates the testing method used by the paper, where the underlined metric is used to compare against the proposed framework.

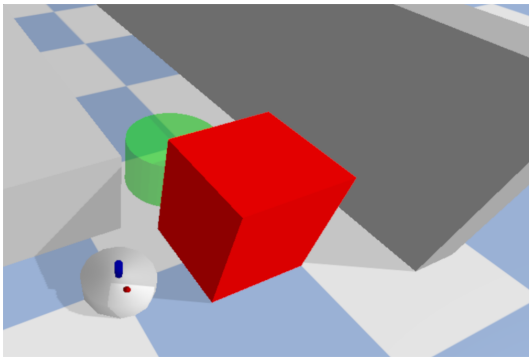
			NAMO		Specify object target poses		
Author	Citation	Learns object dynamics	prehesile	nonprehesile	prehesile	nonprehesile	method metric
Ellis et al.	[15]	✓	✗	✓	✗	✗	success rate
Sabbagh Novin et al.	[35]	✓	✓	✗	✓	✗	success rate, execution time prediction error, final pose error
Scholz et al.	[37]	✓	✓	✗	✗	✗	runtime, planning time, number of replanning number of calls to update model
Vega-Brown and Roy	[46]	✗	✓	✗	✓	✗	computation time
Wang et al.	[48]	✓	✗	✓	✗	✗	computation and execution time
Groote	Propose Frame-work	✗/✓	✗	✓	✗	✓	

A comparison with one state-of-the-art paper is made, that is accomplished by recreating the environment that the state-of-the-art has used during testing. With Wang et al. the computation- (or search) and execution time is compared.

Comparing Computation and Execution time with Wang et al. Wang et al. combines the NAMO problem with learning object dynamics. He tests his method with a task to drive toward a target pose, where a chair is blocking the path [48]. Wang et al. proposed framework takes an accordance approach compared with a contact-implicit motion planning algorithm [48]. A test is performed in a real-world environment, where the robot drive toward a target pose, whilst the path is blocked by a chair. The real-world robot environment is implicitly presented as simulation environment. The environment is mimicked with three walls and a red box on the spot where is chair stands. The implicit representation of the real-world environment and the mimicked robot environments can be seen in Figure 5.11.



(a) Wang et al. simulation environment.



(b) Recreated robot environment.

Figure 5.11: Two similar environments and tasks, the robots are tasked to drive toward a target pose indicated with the green ghost poses. In both environment a direct path is blocked by the red box.

Wang et al. solves the task three times, the average search-, execute- and total time over these three executions is displayed in Table 5.9. Note that, during testing Wang et al. splits the search time in two categories; time to estimate accordance of an object and time to optimize the trajectory. The recreated or mimicked environment is displayed in Figure 5.11b and is solved ten times, every time starting without environmental knowledge and thus an empty kgraph. The average search, execute- and total time over these then executions is also displayed in Table 5.9.

Table 5.9: Average execute-, search and total times for a task that involves learning object dynamics and the NAMO problem. The results are average over three solved tasks for Wang et al., and ten solved tasks for the proposed framework. A visualization of the task if presented in Figure 5.11.

Author	Wang et al.	Groote
search time [sec]	109	26
execution time [sec]	67	4
total time [sec]	176	30

The proposed framework outperforms the contact-implicit motion planning framework proposed by Wang et al. Mainly the search time improves by a margin. The execution time improves, but because Wang et al. tests in an real-world environment where motion equations are more complex compared to the simulated environment, an improvement in the execution time is expected.

Would it be possibel to create extra results that support your work?, or some benchmark tests?

6

Conclusions

This thesis aims to create a robot framework that can combine three topics; learning object dynamics, the NAMO problem and nonprehensile pushing. The overwhelming research in the individual topics stand is contrast to the sparse number of recent papers that combine the three aforementioned topics. This absence of research can be motivated by the emerged challenges when combining the three topics such as the uncertainty in planning and the complexity class of the combined problem.

The main research question, that is: *How do learned objects' system models improve global task planning for a robot with nonprehensile push manipulation abilities over time?* Is answered in Chapter 4 that describes the proposed framework. Action sequences are generated by starting a search from the desired outcome and searching backward to the current configuration with the backward search technique. The chapter shows how learned system knowledge can be leveraged by alternating between searching for an action sequence and executing an action sequence.

There were two aspects that improve global task planning as a result of learning object models. The first improvement is due to classification of objects, that improves the success rate of generated action sequences. Mainly because, probabilistic action sequences that may involve pushing unmovable objects cannot occur when object classification is present. During task execution the proposed framework gathers experience by storing action feedback after successful or unsuccessful actions. As a result the proposed framework converges toward an best strategy to manipulate objects which is the second improved aspect. The best strategy consists of the best combination of controller and system model that yield desirable metrics (success rate and Prediction Error) out of the available combinations of controllers and system models. Evidence for improvements of global task planning as result of learned object dynamics is found in Chapter 5 that presents the results. In this chapter the proposed robot framework is tested by providing a robot with a multiple tasks in various environments. These tests point out that task execution improves when the classification of an object is known, and when experience is gathered and leveraged in the robot environment.

Research subquestion 1, that is: *How to combine learning and planning for push and drive applications?* An answer is given in Chapters 2 to 4. Here it is shown how the proposed framework, that consists of the hypothesis algorithm, the hypothesis graph and the knowledge graph work together to combine learning and planning with the technique of backward search. The curse of dimensionality and an NP-hard problem are bypassed by searching only in a single mode of dynamics instead of searching in composite space. The hgraph is a graph-based structure that presents how the halgorithm is trying to complete a subtask. The structure of the hgraph is used to enforce the backward search technique, in which the halgorithm searches from the target configuration toward the start configuration. Newly gained environmental knowledge is stored in the kgraph, which firstly classifies objects as movable or unmovable, and secondly holds information on how they can be best manipulated. The kgraph can be filled with newly learned environmental knowledge and can be queried for action suggestions.

Research subquestion 2, that is: *To what extend is the combination of the three topics; learning object*

dynamics, the NAMO problem and nonprehensile pushing influenced by environmental experience? The answer is provided in Chapter 5 that present the results. By once solving tasks in a randomized environment and remembering environmental experience, and once solving the same tasks without storing environmental experience. The results indicate that leveraging environmental experience significant improvements are made. These improvements were measured with the total task execution time and Prediction Error, such improvements are due to selecting the best available controller and system model combination. Two important factors determine convergence toward an optimal selection of controller and system model combination. First the success factor, because it determines which experience of control and system model combination is the best choice to manipulate a specific object. Second, the set of available controllers and system models, because the proposed framework can only converge to the best available controller in the set of available controllers. During testing analytic models were used that cannot model a wide variety of systems accurately. Table 5.8 shows therefore a ✗/✓, concluding that the proposed framework can partly combine the three topics. The analytic models could be replaced by adding a system identification module that is moved to the future work section, if implemented the proposed framework can claim that the three topics can fully be combined.

Research subquestion 3, that is: *How does the proposed framework compare against the state-of-the-art?* The proposed framework was able to perform better compared to the state-of-the-art. The proposed framework was tested against an existing methods that only combines a subset of the 3 main topics. Five state-of-the-art papers are selected to compare against where the success rate, planning time, execution time and number of replanning can be used to compare. Only a single comparison was made where search-, execute- and total time to complete a task is compared. From these results, it can be concluded that the proposed framework learns relatively fast compared to Wang et al.

The proposed framework combines the three topics, the NAMO is tackled by extending an existing path planner that detects blocked paths. The halgorithm then combines the extended path planner with the backward search technique to search for action sequences. Action sequences consists of drive and push actions that improve over time by reviewing actions and storing action review in the kgraph. Thereby, nonprehensile pushing, learning and the NAMO problem are combined in the proposed framework. Since the proposed framework classifies objects and stores action feedback, it is concluded that the three topics are partly combined. The proposed framework improves upon the state-of-the-art by a significant margin in search- and execution time.

7

Future work

extend the planner to a kinodynamic planner?

System identification Module

The thesis provides an module that collects IO data for a robot driving system, and for a system that describes the robot pushing an object. It has been fully integrated into the proposed robot framework, but has been replace by analytic models during testing. The appendix provides a categorization that categorized system models in three categories; data-driven model, hybrid models and analytic models. A valuable extension on is to add system identification methods, and to investigate task execute performance with data-driven- and hybrid system models.

Convergence to a Manipulation Strategy

Whilst gaining more experience in an robot environment, the proposed framework converges toward a specific edge parameterization for each object. Such an edge parameterization or manipulation strategy exists of the controller and system model, for robot driving (MPC, *lti-drive-model*), for object pushing (MPPI, *nonlinear-push-model-1*). During testing every manipulated object converged toward the same edge parameterization. Three aspects could be improved; more variation if objects; more experimenting with success factors to generate edge feedback; a larger set of parameterization. All three influence the parameterization to which the halgorithmconverges. It is expected but not shown, that objects that are dissimilar in properties have different parameterization to which they converge.

Benchmark Tests

Three benchmark test have been created to test the proposed robot framework. As a result of poor planning and unforeseen implementation failures, the aforementioned environments have failed to yield any results. The environments that accompany these benchmark test have been moved toward the appendix.

Removing Assumptions

Every assumption taken in Chapter 1 serves to simplify the problem and to narrow the scope of this thesis. They can however all be removed. Starting with assumption **closed-world assumption**, without this assumption the proposed framework must be much more robust. The proposed framework can with the help of this assumption conclude many conclusions deterministically. Examples are the feedback on edges and the classification of an object as unmovable or movable. In real-world applications unmovable objects can become movable, to make the transition toward removing the closed-world assumption the proposed framework must be converted to a probabilistic variant.

Moving from simulation toward the real world must remove the **perfect object sensor assumption**. Sensors and sensor fusion is required to estimate the configuration of the robot itself and objects in the environment. A start is to test the proposed framework with noise added to the perfect sensor.

The **tasks are commutative assumption** assumption makes it possible to randomly select a subtask without influencing the feasibility of the task. Removing this assumption can require an additional rearrangement algorithm [23] to be run to determine a feasible order of handling subtasks.

The **objects do not tip over assumption** prevents objects from tipping over, but at the same time limits the number of objects. There are many possibilities to handle tipped objects. First, adding a tipping detector can detect when an object has tipped over. Then the proposed framework can threaten the objects as a new object and reclassify it. Another method would be a dedicated subroutine to place it in an upright position.

References

[1] Ian Abraham et al. "Model-Based Generalization Under Parameter Uncertainty Using Path Integral Control". In: *IEEE Robotics and Automation Letters* 5.2 (Apr. 2020), pp. 2864–2871. ISSN: 2377-3766, 2377-3774. DOI: 10.1109/LRA.2020.2972836. URL: https://ieeexplore.ieee.org/document/8988215/ (visited on 01/31/2022).

[2] Ermano Arruda et al. "Uncertainty Averse Pushing with Model Predictive Path Integral Control". In: *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*. 2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids). Birmingham: IEEE, Nov. 2017, pp. 497–502. ISBN: 978-1-5386-4678-6. DOI: 10.1109/HUMANOIDS.2017.8246918. URL: http://ieeexplore.ieee.org/document/8246918/ (visited on 04/29/2022).

[3] Paolo Atzeni and Valeria De Antonellis. *Relational Database Theory*. Redwood City, Calif: Benjamin/Cummings Pub. Co, 1993. 389 pp. ISBN: 978-0-8053-0249-3.

[4] Jon Barwise. "An Introduction to First-Order Logic". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 90. Elsevier, 1977, pp. 5–46. ISBN: 978-0-444-86388-1. DOI: 10.1016/S0049-237X(08)71097-8. URL: https://linkinghub.elsevier.com/retrieve/pii/S0049237X08710978 (visited on 05/23/2023).

[5] Maria Bauza, Francois R. Hogan, and Alberto Rodriguez. "A Data-Efficient Approach to Precise and Controlled Pushing". Oct. 9, 2018. arXiv: 1807.09904 [cs]. URL: http://arxiv.org/abs/1807.09904 (visited on 03/01/2022).

[6] Dmitry Berenson et al. "Manipulation Planning on Constraint Manifolds". In: *2009 IEEE International Conference on Robotics and Automation*. 2009 IEEE International Conference on Robotics and Automation (ICRA). Kobe: IEEE, May 2009, pp. 625–632. ISBN: 978-1-4244-2788-8. DOI: 10.1109/ROBOT.2009.5152399. URL: http://ieeexplore.ieee.org/document/5152399/ (visited on 12/05/2022).

[7] Long Chen et al. "A Fast and Efficient Double-Tree RRT*-Like Sampling-Based Planner Applying on Mobile Robotic Systems". In: *IEEE/ASME Transactions on Mechatronics* 23.6 (Dec. 2018), pp. 2568–2578. ISSN: 1083-4435, 1941-014X. DOI: 10.1109/TMECH.2018.2821767. URL: https://ieeexplore.ieee.org/document/8329210/ (visited on 04/14/2022).

[8] Lin Cong et al. "Self-Adapting Recurrent Models for Object Pushing from Learning in Simulation". July 27, 2020. arXiv: 2007.13421 [cs]. URL: http://arxiv.org/abs/2007.13421 (visited on 04/06/2022).

[9] Erwin Coumans and Yunfei Bai. *PyBullet, a Python Module for Physics Simulation for Games, Robotics and Machine Learning*. 2016–2021. URL: http://pybullet.org.

[10] J.C.F. de Winter. "Using the Student's t-Test with Extremely Small Sample Sizes". In: (). DOI: 10.7275/E4R6-DJ05. URL: https://scholarworks.umass.edu/pare/vol18/iss1/10/ (visited on 05/23/2023).

[11] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0029-599X, 0945-3245. DOI: 10.1007/BF01386390. URL: http://link.springer.com/10.1007/BF01386390 (visited on 02/08/2023).

[12] *Disjunctive Programming*. New York, NY: Springer Berlin Heidelberg, 2018. ISBN: 978-3-030-00147-6.

[13] Bruce Donald et al. "Kinodynamic Motion Planning". In: *Journal of the ACM* 40.5 (Nov. 1993), pp. 1048–1066. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/174147.174150. URL: https://dl.acm.org/doi/10.1145/174147.174150 (visited on 06/08/2023).

[14] Mohamed Elbanhawi and Milan Simic. "Sampling-Based Robot Motion Planning: A Review". In: *IEEE Access* 2 (2014), pp. 56–77. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2014.2302442. URL: https://ieeexplore.ieee.org/document/6722915/ (visited on 11/30/2022).

[15] Kirsty Ellis et al. “Navigation among Movable Obstacles with Object Localization Using Photorealistic Simulation”. In: July 2022. URL: https://www.researchgate.net/publication/362092621_Navigation_Among_Movable_Obstacles_with_Object_Localization_using_Photorealistic_Simulation.

[16] A. René Geist and Sebastian Trimpe. “Structured Learning of Rigid-body Dynamics: A Survey and Unified View from a Robotics Perspective”. In: *GAMM-Mitteilungen* 44.2 (June 2021). ISSN: 0936-7195, 1522-2608. DOI: 10.1002/gamm.202100009. URL: https://onlinelibrary.wiley.com/doi/10.1002/gamm.202100009 (visited on 05/24/2023).

[17] Hai-Ning Wu, M Levihn, and M Stilman. “Navigation Among Movable Obstacles in Unknown Environments”. In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2010). Taipei: IEEE, Oct. 2010, pp. 1433–1438. ISBN: 978-1-4244-6674-0. DOI: 10.1109/IROS.2010.5649744. URL: http://ieeexplore.ieee.org/document/5649744/ (visited on 05/06/2023).

[18] Kris Hauser and Jean-Claude Latombe. “Multi-Modal Motion Planning in Non-expansive Spaces”. In: *The International Journal of Robotics Research* 29.7 (June 2010), pp. 897–915. ISSN: 0278-3649, 1741-3176. DOI: 10.1177/0278364909352098. URL: http://journals.sagepub.com/doi/10.1177/0278364909352098 (visited on 12/05/2022).

[19] D. Hsu, J.-C. Latombe, and R. Motwani. “Path Planning in Expansive Configuration Spaces”. In: *Proceedings of International Conference on Robotics and Automation*. International Conference on Robotics and Automation. Vol. 3. Albuquerque, NM, USA: IEEE, 1997, pp. 2719–2726. ISBN: 978-0-7803-3612-4. DOI: 10.1109/ROBOT.1997.619371. URL: http://ieeexplore.ieee.org/document/619371/ (visited on 05/03/2023).

[20] Sertac Karaman and Emilio Frazzoli. “Sampling-Based Algorithms for Optimal Motion Planning”. In: *The International Journal of Robotics Research* 30.7 (June 2011), pp. 846–894. ISSN: 0278-3649, 1741-3176. DOI: 10.1177/0278364911406761. URL: http://journals.sagepub.com/doi/10.1177/0278364911406761 (visited on 03/15/2022).

[21] Eliahu Khalastchi and Meir Kalech. “On Fault Detection and Diagnosis in Robotic Systems”. In: *ACM Computing Surveys* 51.1 (Jan. 31, 2019), pp. 1–24. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3146389. URL: https://dl.acm.org/doi/10.1145/3146389 (visited on 12/13/2022).

[22] Zachary Kingston, Mark Moll, and Lydia E. Kavraki. “Sampling-Based Methods for Motion Planning with Constraints”. In: *Annual Review of Control, Robotics, and Autonomous Systems* 1.1 (May 28, 2018), pp. 159–185. ISSN: 2573-5144, 2573-5144. DOI: 10.1146/annurev-control-060117-105226. URL: https://www.annualreviews.org/doi/10.1146/annurev-control-060117-105226 (visited on 05/06/2022).

[23] Athanasios Krontiris and Kostas Bekris. “Dealing with Difficult Instances of Object Rearrangement”. In: July 2015. DOI: 10.15607/RSS.2015.XI.045.

[24] Steven Michael LaValle. *Planning Algorithms*. Cambridge ; New York: Cambridge University Press, 2006. 826 pp. ISBN: 978-0-521-86205-9.

[25] Yanbo Li, Zakary Littlefield, and Kostas E. Bekris. “Asymptotically Optimal Sampling-Based Kinodynamic Planning”. In: *The International Journal of Robotics Research* 35.5 (Apr. 2016), pp. 528–564. ISSN: 0278-3649, 1741-3176. DOI: 10.1177/0278364915614386. URL: http://journals.sagepub.com/doi/10.1177/0278364915614386 (visited on 06/08/2023).

[26] Tekin Meriçli, Manuela Veloso, and H. Levent Akın. “Push-Manipulation of Complex Passive Mobile Objects Using Experimentally Acquired Motion Models”. In: *Autonomous Robots* 38.3 (Mar. 2015), pp. 317–329. ISSN: 0929-5593, 1573-7527. DOI: 10.1007/s10514-014-9414-z. URL: http://link.springer.com/10.1007/s10514-014-9414-z (visited on 01/31/2022).

[27] neuromorphic tutorial. *LTC21 Tutorial MPPI*. June 7, 2021. URL: https://www.youtube.com/watch?v=19QLyMuQ_BE (visited on 05/31/2022).

[28] Roya Sabbagh Novin et al. "Dynamic Model Learning and Manipulation Planning for Objects in Hospitals Using a Patient Assistant Mobile (PAM)Robot". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). Madrid: IEEE, Oct. 2018, pp. 1–7. ISBN: 978-1-5386-8094-0. DOI: 10.1109/IROS.2018.8593989. URL: https://ieeexplore.ieee.org/document/8593989/ (visited on 05/05/2022).

[29] Manoj Pokharel. "Computational Complexity Theory(P,NP,NP-Complete and NP-Hard Problems)". In: (June 2020).

[30] James Rawlings, David Mayne, and Moritz Diehl. *Model Predictive Control: Theory, Computation and Design*. 2nd edition. Santa Barbara: Nob Hill Publishing, LLC, 2020. ISBN: 978-0-9759377-5-4.

[31] John Reif and Micha Sharir. "Motion Planning in the Presence of Moving Obstacles". In: *26th Annual Symposium on Foundations of Computer Science (Sfcs 1985)*. 26th Annual Symposium on Foundations of Computer Science (Sfcs 1985). Portland, OR, USA: IEEE, 1985, pp. 144–154. ISBN: 978-0-8186-0644-1. DOI: 10.1109/SFCS.1985.36. URL: http://ieeexplore.ieee.org/document/4568138/ (visited on 05/05/2022).

[32] Arend Rensink. "Representing First-Order Logic Using Graphs". In: *Graph Transformations*. Ed. by Hartmut Ehrig et al. Red. by David Hutchison et al. Vol. 3256. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 319–335. ISBN: 978-3-540-23207-0 978-3-540-30203-2. DOI: 10.1007/978-3-540-30203-2_23. URL: http://link.springer.com/10.1007/978-3-540-30203-2_23 (visited on 05/23/2023).

[33] Nicholas Roy. "Hierarchy, Abstractions and Geometry". May 6, 2021. URL: https://www.youtube.com/watch?v=mP-uK1PFqfI (visited on 05/03/2023).

[34] Roya Sabbagh Novin, Mehdi Tale Masouleh, and Mojtaba Yazdani. "Optimal Motion Planning of Redundant Planar Serial Robots Using a Synergy-Based Approach of Convex Optimization, Disjunctive Programming and Receding Horizon". In: *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering* 230.3 (Mar. 2016), pp. 211–221. ISSN: 0959-6518, 2041-3041. DOI: 10.1177/0959651815617883. URL: http://journals.sagepub.com/doi/10.1177/0959651815617883 (visited on 05/05/2022).

[35] Roya Sabbagh Novin et al. "A Model Predictive Approach for Online Mobile Manipulation of Non-Holonomic Objects Using Learned Dynamics". In: *The International Journal of Robotics Research* 40.4-5 (Apr. 2021), pp. 815–831. ISSN: 0278-3649, 1741-3176. DOI: 10.1177/0278364921992793. URL: http://journals.sagepub.com/doi/10.1177/0278364921992793 (visited on 05/05/2022).

[36] Basak Sakcak et al. "Sampling-Based Optimal Kinodynamic Planning with Motion Primitives". In: *Autonomous Robots* 43.7 (Oct. 2019), pp. 1715–1732. ISSN: 0929-5593, 1573-7527. DOI: 10.1007/s10514-019-09830-x. URL: http://link.springer.com/10.1007/s10514-019-09830-x (visited on 06/08/2023).

[37] Jonathan Scholz et al. "Navigation Among Movable Obstacles with Learned Dynamic Constraints". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). Daejeon, South Korea: IEEE, Oct. 2016, pp. 3706–3713. ISBN: 978-1-5090-3762-9. DOI: 10.1109/IROS.2016.7759546. URL: http://ieeexplore.ieee.org/document/7759546/ (visited on 04/29/2022).

[38] Neal Seegmiller et al. "Vehicle Model Identification by Integrated Prediction Error Minimization". In: *The International Journal of Robotics Research* 32.8 (July 2013), pp. 912–931. ISSN: 0278-3649, 1741-3176. DOI: 10.1177/0278364913488635. URL: http://journals.sagepub.com/doi/10.1177/0278364913488635 (visited on 02/16/2022).

[39] Jitkomut Songsiri. *Input Signals: Persistent Excitation*. Jan. 8, 2022. URL: https://www.youtube.com/watch?v=IDaCw3LjzC4 (visited on 06/06/2023).

[40] Max Spahn. *Urdf-Environment*. Version 1.2.0. Aug. 8, 2022. URL: https://github.com/maxspahn/gym_envs_urdf.

[41] Mike Stilman et al. "Planning and Executing Navigation among Movable Obstacles". In: *Advanced Robotics* 21.14 (Jan. 2007), pp. 1617–1634. ISSN: 0169-1864, 1568-5535. DOI: 10.1163/156855307782227408. URL: https://www.tandfonline.com/doi/full/10.1163/156855307782227408 (visited on 05/24/2023).

[42] Jochen Stüber, Marek Kopicki, and Claudio Zito. "Feature-Based Transfer Learning for Robotic Push Manipulation". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)* (May 2018), pp. 5643–5650. DOI: 10.1109/ICRA.2018.8460989. arXiv: 1905.03720. URL: http://arxiv.org/abs/1905.03720 (visited on 03/15/2022).

[43] Jochen Stüber, Claudio Zito, and Rustam Stolkin. "Let's Push Things Forward: A Survey on Robot Pushing". In: *Frontiers in Robotics and AI* 7 (Feb. 6, 2020), p. 8. ISSN: 2296-9144. DOI: 10.3389/frobt.2020.00008. arXiv: 1905.05138. URL: http://arxiv.org/abs/1905.05138 (visited on 02/24/2022).

[44] Marc Toussaint et al. "Sequence-of-Constraints MPC: Reactive Timing-Optimal Control of Sequential Manipulation". Mar. 10, 2022. arXiv: 2203.05390 [cs]. URL: http://arxiv.org/abs/2203.05390 (visited on 04/29/2022).

[45] Jur van den Berg et al. "Path Planning among Movable Obstacles: A Probabilistically Complete Approach". In: *Algorithmic Foundation of Robotics VIII*. Ed. by Gregory S. Chirikjian et al. Red. by Bruno Siciliano, Oussama Khatib, and Frans Groen. Vol. 57. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 599–614. ISBN: 978-3-642-00311-0 978-3-642-00312-7. DOI: 10.1007/978-3-642-00312-7_37. URL: http://link.springer.com/10.1007/978-3-642-00312-7_37 (visited on 06/24/2022).

[46] William Vega-Brown and Nicholas Roy. "Asymptotically Optimal Planning under Piecewise-Analytic Constraints". In: *Algorithmic Foundations of Robotics XII*. Ed. by Ken Goldberg et al. Vol. 13. Cham: Springer International Publishing, 2020, pp. 528–543. ISBN: 978-3-030-43088-7 978-3-030-43089-4. DOI: 10.1007/978-3-030-43089-4_34. URL: http://link.springer.com/10.1007/978-3-030-43089-4_34 (visited on 06/23/2022).

[47] M. Verhaegen and Vincent Verdult. *Filtering and System Identification: A Least Squares Approach*. Cambridge ; New York: Cambridge University Press, 2007. 405 pp. ISBN: 978-0-521-87512-7.

[48] Maozhen Wang et al. "Affordance-Based Mobile Robot Navigation Among Movable Obstacles". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). Las Vegas, NV, USA: IEEE, Oct. 24, 2020, pp. 2734–2740. ISBN: 978-1-72816-212-6. DOI: 10.1109/IROS45743.2020.9341337. URL: https://ieeexplore.ieee.org/document/9341337/ (visited on 06/28/2022).

[49] Jan Wielemaker et al. "SWI-Prolog". In: *Theory and Practice of Logic Programming* 12.1-2 (Jan. 2012), pp. 67–96. ISSN: 1471-0684, 1475-3081. DOI: 10.1017/S1471068411000494. URL: https://www.cambridge.org/core/product/identifier/S1471068411000494/type/journal_article (visited on 05/23/2023).

[50] Grady Williams, Andrew Aldrich, and Evangelos Theodorou. "Model Predictive Path Integral Control Using Covariance Variable Importance Sampling". Oct. 28, 2015. arXiv: 1509.01149 [cs]. URL: http://arxiv.org/abs/1509.01149 (visited on 05/02/2022).

[51] Liangjun Zhang, Young J. Kim, and Dinesh Manocha. "A Simple Path Non-existence Algorithm Using C-Obstacle Query". In: *Algorithmic Foundation of Robotics VII*. Ed. by Srinivas Akella et al. Vol. 47. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 269–284. ISBN: 978-3-540-68404-6 978-3-540-68405-3. DOI: 10.1007/978-3-540-68405-3_17. URL: http://link.springer.com/10.1007/978-3-540-68405-3_17 (visited on 06/16/2022).

Appendix

The appendix contains additional information that may help better understand the thesis.

Complexity Classes

Problems in class P have a solution which can be found in polynomial time, problems in non-deterministic polynomial-time (NP) are problems for which no polynomial algorithms have been found yet, and of which it is believed that no polynomial time solution exist. For problems in NP, when provided with a solution, verifying that the solution is indeed a valid solution can be done in polynomial time. NP-hard problems are a class of problems which are at least as hard as the hardest problems in NP. Problems that are NP-hard do not have to be elements of NP. They may not even be decidable [29]. This thesis or other recent studies in the references do not attempt to find an optimal solution. Instead, they provide a solution whilst guaranteeing properties such as near-optimality or probabilistic completeness. As the piano's mover problem can be reduced to the NAMO problem combined with relocating objects to target poses, the conclusion can be drawn that this NAMO problem is NP-hard.

Benchmark Tests

Three benchmark test are presented, the blockade, swap and surround environment. The environments are created but have not been used for testing the proposed halgorithm. The three environments are presented and a short description is provided that elaborates which parts of the proposed method are tested. Now the blockade environment is presented.

In the blockade environment the robot is tasked with placing a box in a target pose that is blocked by a cylinder object.

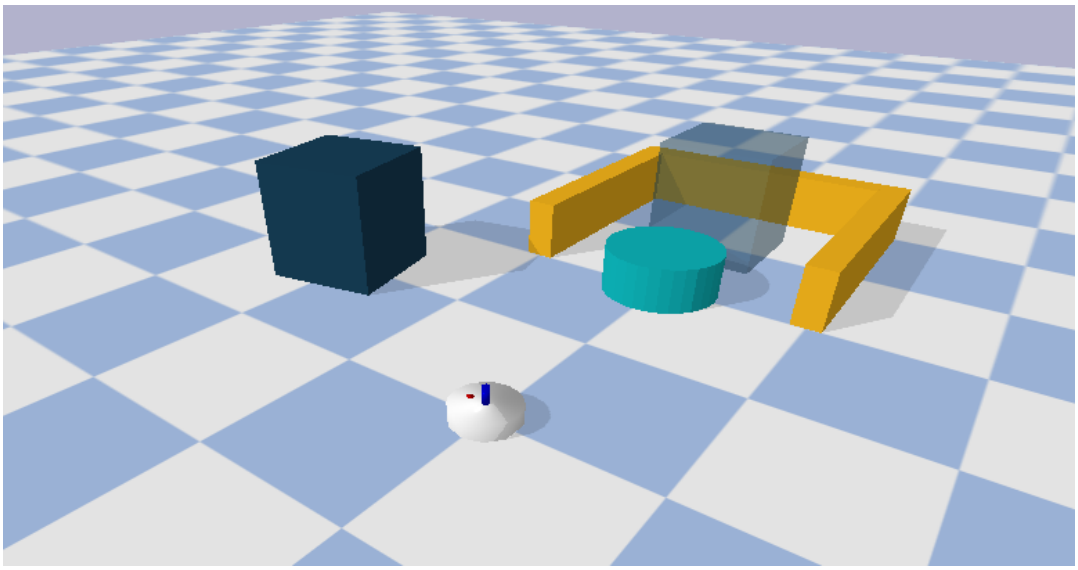


Figure 1: The blockade environment where the robot is tasked to push the blue box toward the target pose indicated with the target ghost pose. A direct path is blocked by the movable light blue cylinder object and the unmovable yellow walls.

The blockade environment presented in the figure above would show the backward search technique in action. The backward search technique first plans for the blue box toward the target ghost pose, then an blocking object is detected. That blocking object can be either one of the walls or the cylinder object. The blocking object is pushed to free the path, in the case of the unmovable walls the kgraph is updated indicating that the walls are inmovable, in the case of the cylinder object, it is moved out of the way and the path is freed.

In the swap environment the robot is tasked to swap the poses of the two objects in the environment.

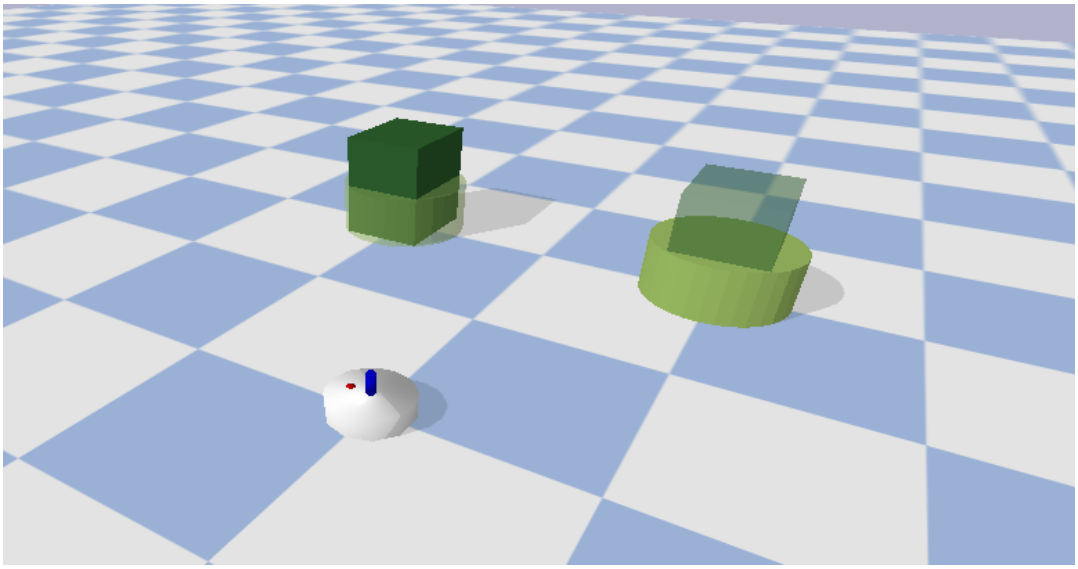


Figure 2: The swap environment where the robot is tasked with swapping the poses of the cylinder and the box object.

Just as the previously presented blockade environment, the swap environment's purpose is to show the backward search technique that results in a generated action sequence. In that action sequence the cylinder or box can be eitehr obj_A and the other object is obj_B. The action sequence moves obj_A to free the path for obj_B that is then pushed to it's target pose, the task is completed by pushing obj_A toward it's target pose. The proposed halgorithm does not perform global path optimization, and it can be shows using the swap environment. In the current figure above the initial robot-box, and robot-cylinder distances are equal. If these were unequal, for example, place the robot initial pose next to the box object and far from the cylinder object, there would be best object to manipulate first (the closer object). Since there is no global path optimized the robot selects to first manipulate the box or the cylinder randomly, resulting in half of the times driving further compared to a globally optimized path.

In the surrounded environment the robot has to learn which box is movable to escape the enclosure of boxes.

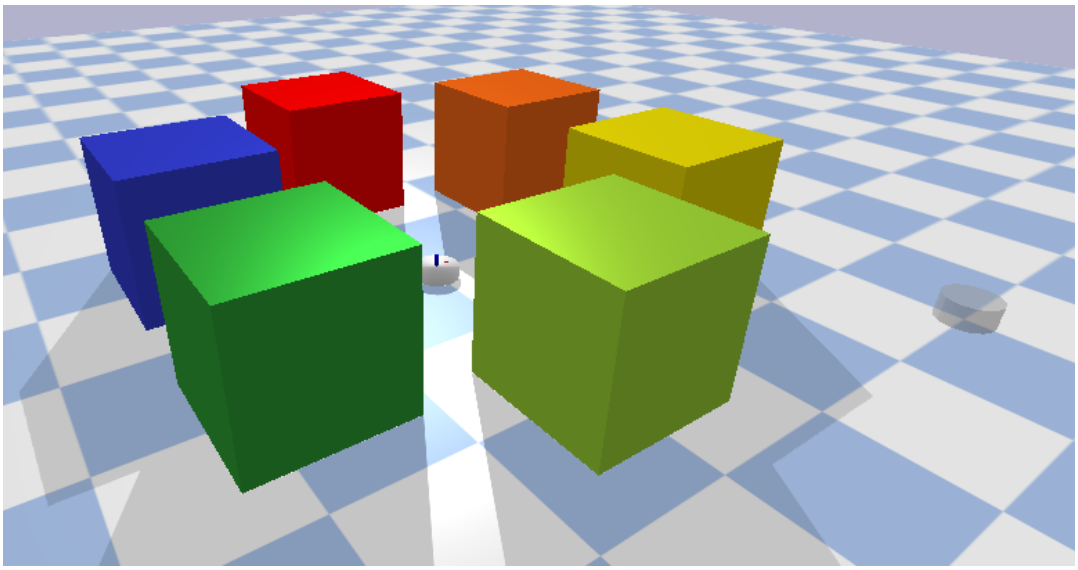


Figure 3: The surrounded environment where the robot is tasked with escaping the surrounding enclosure by driving to the target ghost pose displayed on the right side in the figure. Every box objects is unmovable except the red box which is movable.

The surrounded environment's purpose is to present the effect of storing object classes. Initially the robot needs to find which object is movable and which are unmovable. By storing which object class information a generated action sequence immediately decides to push the movable red box.

System Models

Robot pushing is modeled using system models, here an overview is presented that classifies pushing system models in three categories, analytical-, hybrid- and data-driven models.

Analytical models

Historically, analytical models are the first models to emerge, most prominently used are *state-space* representations, *transfer functions* and *differential equations*. Building an analytical model requires thorough knowledge of the system it models, because every system parameter, such as mass, damping coefficient, the center of gravity, geometry, friction coefficient or inertia. Analytical approaches rely on accurate identification of physical parameters which makes analytical models unfit for manipulation while learning system models [2], [42].

Nevertheless, the work in [5] manages to create a stable controller for push manipulation using an analytical model. Because thorough model identification of the pushable object was performed, the trajectory error stayed within reasonable boundaries.

Data-driven models

Among recent studies, data-driven models shown an uptrend in popularity [26, 5, 42, 43]. Fully data-driven methods don't model any structure or use a generalised model of the system it describes. A system is viewed as a black box, which is fed input and gives back output. This reduces the need for prior information about the system significantly. IO data is analysed to estimate the structure of the black box. The IO data analysed which solemnly serves the creation of a model is called the *model train set*. The advantage of requiring a minimum of prior information comes at the cost of the amount of IO data required. If there is not sufficiently much data an identified model will not be accurate. If there is enough data, but the data does not encapsulate the systems behaviour enough, then a identified model will also not be accurate. The encapsulating system behavior should reside in the IO data collected. During data collection the system should be persistently excited which depends upon the input sequence. The *persistent excitation order of input signal* determines the amount of system behaviour ends up in the collected IO data. Persistent excitation order of input signal is the degree to which an input contains enough information for exciting a system for system identification purpose [39]. The need for so much, rich IO data can be showed with an example. [5] compared a purely analytical approach with a data-driven approach in push manipulation. The data-driven approach can take up to 200 samples of IO data to sufficiently match the performance of an analytical controller. With more IO data, data-driven approaches lower output errors and increase performance, outperforming analytical approaches but also outperforming hybrid approaches, which are discussed in the next subsection. Data-driven approaches outperform because data-driven approaches capture even tiny dynamical details of the true dynamics. That is, assuming that the dynamical details reside in the IO data.

The nonlinear effects which dominate systems that describe object manipulation reside in IO data, because data-driven approaches models are not assuming any structure which could limits capturing nonlinear effects, the data-driven approaches are a worthy method for estimating true dynamics. It must be mentioned that data-driven methods outperform other model classes with enough data, for which the training time is in robotics not always available. If other modelling approaches are available which estimating true dynamics accurate enough, the data-driven approach should be avoided because of the long lasting training time.

Hybrid models

Hybrid models are an extension of analytical approaches with data-driven methods. Whilst the interactions between objects are still represented analytically, some quantities of interest are estimated based on observations (e.g. the coefficients of friction) [43]. Recent literature reveals the foremost hybrid methods are parameterisable differential equations. Parameterisable state-space models and parameterisable transfer models do exist, though the most widely used parameterisable model remains a parameterisable differential model, which takes the form:

$$\frac{dx}{dt} = f(x, u, p) \tag{1}$$

where x is the state vector, u is the input vector and p is the parameterisation which needs to be found such that $f(x, u, p)$ accurately estimates the true dynamics. With a random or educated initial guess of the parameterisation p, a system model is provided without full knowledge of all system parameters. An initial guess as parameterisation may not be a very accurate model, but it does allow to skip a tedious system identification period. Online adaptation allows to converge to a local minimum during execution. Whether this local minimum also coincides with the global minimum is dependent on the optimisation technique and the initial guess. Parameterisable differential models are very powerful in situations where the general structure of dynamics is known, but certain parameters e.g. weight, the friction coefficient is unknown or change over time [38].

In a environment with unknown objects, the ability to rapidly interact with objects is provided by hybrid models. During interaction hybrid approaches can improve their model accuracy whilst also adapting to changing systems. To fully capture the push mechanics, data-driven methods should be used, because only data-driven methods are able to capture a large portion of the nonlinear dynamics.

Control Methods

MPC Control

In recent literature involving predictive methods Model Predictive Control (MPC) methods are dominating, before moving on to MPC and variations of MPC, MPC will briefly be explained. The basic concept of MPC is to use a dynamic model to forecast system behaviour and optimise the forecast to produce the best decision for the control move at the current time. Models are therefore central to every form of MPC. Because the optimal control move depends on the initial state of the dynamic system [30]. A dynamical model can be presented in various forms, let's consider a familiar differential equation.

$$\frac{dx}{dt} = f(x(t), u(t))$$
$$y = h(x(t), u(t))$$
$$x(t_0) = x_0$$

In which $x \in \mathbb{R}^n$ is the state, $u \in \mathbb{R}^m$ is the input, $y \in \mathbb{R}^p$ is the output, and $t \in \mathbb{R}$ is time. The initial condition specifies the value of the state x at $t = t_0$, and a solution to the differential equation for time greater than t_0, $t \in \mathbb{R}_{\geq 0}$ is sought. If little knowledge about the internal structure of a system is available, it may be convenient to take another approach where the state is suppressed, no internal structure about the system is known and the focus lies only on the manipulable inputs and measurable outputs. As shown in figure 4, consider the system $G(t)$ to be the connection between u and y. In this viewpoint, various system identification techniques are used, in which u is manipulated and y is measured [30]. From the input-output relation, a system model is estimated or improved. The system model can be seen inside the MPC controller block in figure 4.

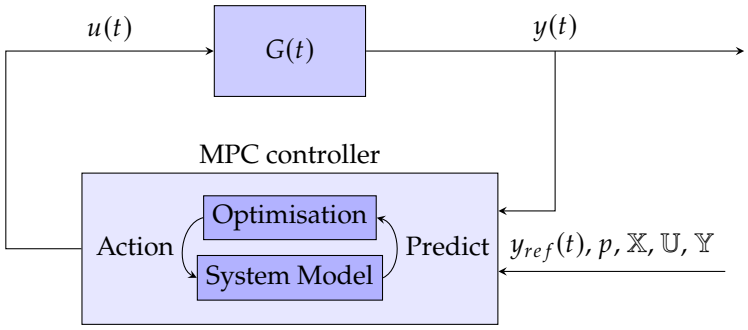


Figure 4: System $G(t)$ with input $u(t)$, output $y(t)$ and MPC controller with input $y(t)$ reference signal $y_{ref}(t)$, parameterisation p and constraint sets $\mathbb{X}$, $\mathbb{U}$, $\mathbb{Y}$

Some states of the system might be inside an obstacle region, such a region is undesirable for the robot to be in or go toward. The robot states are allowed in free space, which is all space minus the obstacle region. The free space is specified as a state constrained set $\mathbb{X}$. Allowable input can be restricted by the input constraint set $\mathbb{U}$, a scenario in which input constraints are required is for example the maximum torque an engine produces at full throttle. Lastly, the set of allowed outputs is specified in the output constraint set $\mathbb{Y}$. State, input and output constraints must be respected during optimisation, the optimiser takes the state-, input- and output constraint sets $\mathbb{X}$, $\mathbb{U}$, $\mathbb{Y}$ and if feasible, finds an action sequence driving the system toward the reference signal while constraints are respected. The MPC system model predicts future states where the system is steered toward as a result of input actions.

The optimisation minimises an objective function $V_N(x_0, y_{ref}, \mathbf{u}_N(0))$, where $\mathbf{u}_N(k) = (u_k, u_{k+1}, \ldots, u_{k+N})$. The objective function takes the reference signal as an argument together

with the initial state and the control input for the control horizon. The objective function then creates a weighted sum of some heuristic function. States and inputs resulting in outputs far from the reference signal are penalised more by the heuristic function than outputs closer to the reference signal. Because the objective function is a Lyapunov function, it has the property that, it has a global minimum for the optimal input $\mathbf{u}_N^*$. If the system output reaches the reference signal y_{ref}, x_{ref} then u_{ref} will be mapped to the output reference signal as such $y_{ref} = h(x_{ref}, u_{ref})$. As a result solving the minimisation problem displayed in equation (2) gives the optimal input which steers the system toward the output reference signal while at the same time respecting the constraints.

$$\begin{aligned} \underset{u_k, u_{k+1}, \ldots, u_{k+N}}{\text{minimize}} \quad & V_N(x_0, y_{ref}, u_k, u_{k+1}, \ldots, u_{k+N}) \\ \text{subject to} \quad & x(k+1) = f(x(k), u(k)), \\ & x \in \mathbb{X}, \\ & u \in \mathbb{U}, \\ & y \in \mathbb{Y}, \\ & x(0) = x_0 \end{aligned} \tag{2}$$

After solving the minimisation problem, equation (2), the optimal input sequence is obtained $\mathbf{u}_N^*$ (given that the constraints are respected for such input), from which only the first input is executed for time step k to $k+1$. Then all indices are shifted such that the previous time step $k+1$ becomes k, the output is measured and the reference signal, parameterisation, and constraints sets are updated and a new minimisation problem is created, which completes the cycle.

MPPI Control

Introduced by [50] MPPI control arose, which was followed by MPPI control combined with various system identification methods and system models [1, 8, 2]. The core idea is from the current state of the system with the use of a system model and randomly sampled inputs to simulate in the future a number of "rollouts" for a specific time horizon, [27]. These rollouts indicate the future states of the system if the randomly sampled inputs would be applied to the system, the future states can be evaluated by a cost function which penalised undesired states and rewards desired future states. A weighted sum over all rollouts determines the input which will be applied to the system. If a goal state is not reached, the control loop starts with the next iteration. An example is provided, see figure 5.

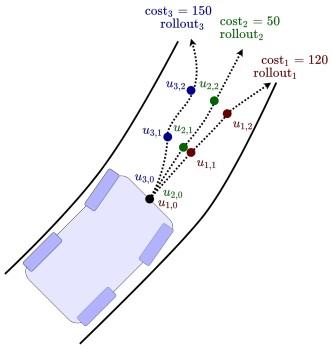


Figure 5: MPPI controlled race car using a control horizon of 3 time steps, with 3 rollouts all having their respected inputs as $u_{i,j}$ where i is the rollout index and j indicates the time step [27].

Here 3 rollouts are displayed, The objective function is designed to keep the car driving on the center of the road by penalising rollouts which are further away from the center of the road relatively more. resulting in a high cost for rollout$_1$ and rollout$_3$ compared to rollout$_2$. As a result, the input send to the system as a weighted sum of the rollouts is mostly determined by rollout$_2$. The weighted sum determining the input is displayed in equation (3), from [27].

$$u(k+1) = u(k) + \frac{\sum_i w_i \delta u_i}{\sum_i w_i} \tag{3}$$

Where δu_i is the difference between $u(k)$ and the input for rollout i, the weight of rollout$_i$ is detemined as: $w_i = e^{-\frac{1}{\lambda}\text{cost}_i}$, λ is a constant parameter.