

# 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

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 - Feb, 2023

Faculty:                         Faculty of Cognitive Robotics, Delft

Cover:                          Simulation environment used during the thesis [35].

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



Delft Center for  
Systems and Control



Cognitive  
Robotics

# Abstract

In the field of robotics, much attention has been given to the research topics *learning object dynamics* [6, 34], *Navigation Among Movable Objects (NAMO)* [5, 11, 18, 21] and *nonprehensile pushing* [2, 3, 22, 36, 37, 38]. However, because scientific papers that include all three topics are scarce, the combination of these research topics into one robot framework has not been explored sufficiently. A combination of the topics leads to an improvement in planning, faster execution times and rapidly concluding unfeasibility for a robot acting in new and unforeseen environments.

To solve the problem presented in the first paragraph, 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 is used to draw a 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 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 to increase task performance, such as success rate and execution time.

The hypothesis algorithm relies on planning to generate hypotheses, for the purpose of planning and freeing blocked paths that can be encountered, a new planning algorithm is proposed. This planner extends the double tree optimised Rapidly-exploring Random Tree algorithm [5]. The planners both construct a configuration space for an object and are provided with starting and target pose for that object. The planners then convert these poses to points in configuration space and searches for a path connecting the starting point to the target point. A key difference between the newly proposed planner and the existing planner lies in the ability to detect blocked paths. For the new planner, objects are initially classified as “unknown” and can later be categorized as either “movable” or “obstacle”. The object type information is then used when constructing the configuration space for the newly proposed planning algorithm. Its configuration space consists of the conventional free, obstacle, and unknown-and movable space.

To carry out the investigation, a mobile robot in a robot environment is created with movable and unmovable objects. 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, provides a path that can be tracked.

The three topics can be combined into a robot framework because results indicate that task execution improves as the robot gains more experience in its environment. The proposed framework, which covers all three research topics, performs equivalent or better compared to the state-of-the-art frameworks that are specialized in only two out of three research topics [12, 32, 33, 40, 42].

G.S. Groote  
Delft, May 2023

# Contents

<b>Abstract</b>	i
<b>Symbols</b>	vii
<b>1 Introduction</b>	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 An Example of Robots and Objects . . . . .	8
1.3 Report Structure . . . . .	9
<b>2 Required Background</b>	10
2.1 System Identification . . . . .	10
2.2 Control Methods . . . . .	13
2.3 Planning . . . . .	13
2.3.1 Estimating Path Existence . . . . .	14
2.3.2 Motion Planning . . . . .	18
2.4 Monitoring Metrics . . . . .	24
<b>3 Planning in four subspaces</b>	26
<b>4 The Hypothesis and Knowledge Graph</b>	31
4.1 Overview of the proposed framework . . . . .	31
4.2 Hypothesis Graph . . . . .	32
4.2.1 Definition of the hypothesis graph (hgraph) . . . . .	32
4.2.2 Examples . . . . .	35
4.2.3 hypothesis algorithm . . . . .	38
4.3 Monitoring Metrics . . . . .	44
4.4 Knowledge Graph . . . . .	52
4.4.1 Definition . . . . .	52
4.4.2 Example . . . . .	53
4.4.3 Edge Metrics . . . . .	54
<b>5 Results</b>	56
5.1 Proposed Method Metrics . . . . .	56
5.2 Randomization . . . . .	58
5.2.1 A Driving Task . . . . .	59
5.2.2 A Pushing Task . . . . .	63
5.3 Comparison with State-of-the-Art . . . . .	67
<b>6 Conclusions</b>	70
6.1 Future work . . . . .	71
6.1.1 System identification module . . . . .	71
6.1.2 Removing Assumptions . . . . .	71
<b>Glossary</b>	73
<b>References</b>	74
<b>7 Appendix</b>	78
.1 System Identification Methods . . . . .	79
.1.1 Analytical models . . . . .	80
.1.2 Data-driven models . . . . .	80

.1.3	Hybrid models . . . . .	81
<b>A</b>	<b>Complexity Classes</b>	<b>83</b>
<b>B</b>	<b>Control Methods</b>	<b>84</b>
B.1	Model Predictive Control (MPC) Control . . . . .	84
B.2	Model Predictive Path Integral (MPPI) Control . . . . .	86

# List of Figures

1.1	Two robots in the simulation 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 sequences are sent toward the robot as input to collect the output response of the robot. . . . .	11
2.2	To collect Input-Output (IO) data sequence, the robot must first drive toward the initial pose at the start of an arrow. The corresponding input is sent toward the robot, and the robot and the object's output response is collected. . . . .	12
2.3	Environment with the point robot, unmovable yellow walls and movable brown boxes. The robot classifies all objects as unknown since it is unaware of the objects' classes. . . . .	16
2.4	The configuration space for the point robot with different cell sizes. The robot environment corresponds to the environment presented in Figure 2.3. . . . .	16
2.5	Flowchart displaying the drawbacks when checking if there exist a path in configurations space between a start- and target configuration. These drawbacks are the reason why path existence can only be <i>estimated</i> rather than guaranteed. . . . .	17
2.6	Snapshot of the configuration space during a search from start to the target configuration.	19
2.7	Create, find and connect new node to parent node. . . . .	20
2.8	The new node is connected to the node in search space that results in the lowest cost. . . . .	21
2.9	Check if the newly added node can lower cost for nearby nodes. . . . .	22
2.10	Comparing schematic example to a visualization of the implemented algorithm. . . . .	23
3.1	The robot tasked with driving toward the other side of the brown box. . . . .	28
4.1	Flowchart representation of the proposed framework. . . . .	32
4.2	FSM displaying the status of an identification edge . . . . .	34
4.3	FSM displaying the status of an action edge . . . . .	35
4.4	Legend for hgraph's nodes an edges . . . . .	36
4.5	the hgraph in multiple stages when the hypothesis algorithm (halgorithm) searches for an hypothesis to a drive task. . . . .	38
4.6	executing the hypothesis found in figure 4.5. . . . .	39
4.7	the search (above) and execution (below) loop, that make up the two main parts of the proposed halgorithm. the full flowchart is presented in figure 4.13 . . . . .	39
4.8	initialize start and target nodes and the start of a created hypothesis to complete a pushing task. . . . .	41
4.9	The hypothesis for a pushing task becomes valid. The hypothesis contains an edge that sends actions to the robot, and thus the hypothesis is executed. . . . .	41
4.10	todo . . . . .	43
4.11	TODOhgraph for pushing the green box to the target configuration . . . . .	43
4.12	Executing two hypotheses that both failed to complete because a fault was detected. Both edges are added to the blocklist, preventing the regeneration of edges with the same parameterization. The halgorithm concludes the task to be unfeasible. . . . .	46
4.13	Flowchart displaying the hypothesis graph's workflow. . . . .	49
4.14	hgraph for driving to target configuration and encountering a blocked path . . . . .	51
4.15	knowledge graph (kgraph) with three edges for robot driving and two edges for pushing the green box. . . . .	53
5.1	A random environment initialised by tuning parameters presented in Table 5.4. After initialisation the environment is reshuffled three times. . . . .	59

5.2	Search-, execution- and total time to complete a drive task <b>whilst</b> using 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 task contains three subtasks, in other words, the robot must drive to three target poses in order to complete a task. 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. . . . .	61
5.3	Search-, execution- and total time to complete a drive task <b>without</b> using 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 task contains three subtasks, in other words, the robot must drive to three target poses in order to complete a task. 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. . . . .	61
5.4	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 is leveraged indicated by “with kgraph” and once with random selection indicated by “no kgraph”. . .	62
5.5	Two random environments where the task contains a subtask displayed by a target ghost pose. . . . .	64
5.6	Search-, execution- and total time to complete a pushing task <b>with</b> 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. . . . .	65
5.7	Search-, execution- and total time to complete a pushing task <b>without</b> 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. . . . .	65
5.8	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 is leveraged indicated by “with kgraph” and once with random selection indicated by “no kgraph”. . .	66
5.9	Box plot of the Prediction Error, a comparison between kgraph action suggestions and randomly selection. . . . .	67
5.10	Two similar environments and tasks, the robots are tasked to drive toward a target position indicated with the green ghost poses. In both environment a direct path is blocked by the red box. . . . .	69
B.1	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 $\mathbb{X}, \mathbb{U}, \mathbb{Y}$ . . . . .	84
B.2	A discrete MPC scheme tracking a constant reference signal. $k$ indicates the discrete time step, $N$ the control horizon . . . . .	86
B.3	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 [24]. . . . .	87

# List of Tables

1.1	Overview of topics; learning object dynamics, NAMO and specifying target poses in recent literature. This thesis combines the three topics; learning object dynamics, NAMO and nonprehensile pushing. . . . .	6
2.1	Terminology related to planning, terms are presented in the left column with corresponding description in the right column. . . . .	14
2.2	Functions used by the Algorithm 1. . . . .	24
2.3	Monitor metrics used to monitor if a fault occurred during the execution of an edge . . . . .	25
4.1	Terminology of terms used . . . . .	37
4.2	Steps were taken to prepare an action edge for execution. An action edge increments its status as indicated in Figure 4.3. The path estimator and path planner are elaborated in Section 2.3 and chapter 3 respectively. . . . .	42
4.3	Monitor metrics used to monitor if a fault occurred during the execution of an edge . . . . .	45
4.4	The functions used by the halgorithm. . . . .	47
4.5	Elaborate information on actions taken by blocks in ?? . . . . .	50
4.6	Edge metrics used to rank control methods from 'good' to 'bad' . . . . .	54
5.1	Method metrics used to measure task performance by the proposed robot framework, and motivation why the metric is relevant. . . . .	57
5.2	Available drive and push system models. . . . .	57
5.3	The tuning parameters that must be specified to create a random environment . . . . .	59
5.4	The selected tuning parameters for the randomised drive environment. . . . .	60
5.5	Selecting the (MPC, <i>lti-drive-model</i> ) parameterization versus selecting the (MPPI, <i>lti-drive-model</i> ) parameterization for drive actions. . . . .	62
5.6	The selected tuning parameters for the randomised push environment. . . . .	63
5.7	Influence of kgraph suggestions in selecting the (MPPI, <i>nonlinear-push-model-1</i> ) parameterization versus selecting the (MPPI, <i>nonlinear-push-model-2</i> ) parameterization for push actions. . . . .	66
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. . . . .	68
5.9	Average execute-, search and total times for a repeated task solved three times. The task involves learning object dynamics and the NAMO problem and can be visualise in Figure 5.10. . . . .	69

# 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
$\epsilon^{pred}$	Prediction Error
$\epsilon^{track}$	Tracking Error
C-space	Configuration Space, $Dim(\text{C-space}) \in \mathbb{R}^2 \vee \mathbb{R}^3$
$c$	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
$\theta$	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
$\alpha$	Success factor for an edge in kgraph
$DT$	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 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**, **Navigation Among Movable Objects (NAMO)** and **nonprehensile pushing**. 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 does learned environmental knowledge influence planning? Does the execution time decrease, and how much for a repeated task? How much can generated action sequences improve as the robot learns more?

Learning object dynamics enables the robot to manipulate unforeseen objects, NAMO allows the robot to move around in an environment even if an object blocks 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, which is a wide variety of tasks. Learning abilities allow robots to operate in new environments and to adapt to environmental changes. Examples are exploration, rescue missions or construction sites. An unfamiliar environment can emerge by slightly changing a familiar environment, such as a supermarket with 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 [13].

Nonprehensile pushing is a form of manipulation widely available for robots, even though 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, which is a manipulation action many robots should leverage.

The NAMO problem and nonprehensile pushing are overlapping topics. Moving an object out of the way to free a path (NAMO) is similar to pushing an object to a target pose. However, by combining both topics, a new situation may occur. When pushing  $obj_a$  to a new pose, the path is blocked by  $obj_b$  which must thus first be moved. When pushing  $obj_b$  to a new pose, the path is blocked by  $obj_c$ , which must

---

thus first be moved, etc. Such a situation will not occur when dealing with only the NAMO problem or with only nonprehensile pushing. A robot that can place an object at new poses on top of NAMO can handle a broader variety of tasks compared to a robot that can only navigate among movable objects.

Research into approaches tackling the three topics just described can be split into two categories. The bulk falls into the category of hierarchical approaches [12, 20, 33, 40, 42]. The remainder falls in category locally optimal approaches [25, 31, 32]. Both approaches are elaborated upon later in this chapter. First, the configuration space that builds up to the composite configuration space 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** Now planning for a single action and its relation to the configuration space are investigated. 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 leverage 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 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 position is, in many cases, unfeasible. For example, manipulating an object  $obj_A$  to a target position 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 position. 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 [39]. A composite configuration in such a composite space fully describes where the robot and objects are in the workspace. In recent literature, the composite configuration space is also named composite configuration space [40], finding “bridges” between configuration spaces [14] or room configuration [32]. The term composite configuration space has been selected because it indicates multiple configuration spaces composed together and does not confuse with robot joints (or hinges). Path planners (in contrast to the planning in configuration space) have great difficulty in connecting a starting composite configuration to a target composite configuration [30] 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:

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

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 that originates from constraints. For example, from a nonholonomic robot, the class of robots that can be described by a bicycle model [27] are nonholonomic. These robots, such as the boxer robot in Figure 1.1b, cannot drive sideways. The nonholonomic 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* [14], because configuration space relates to a single action. Dedicated path planners such as PRM [15], PRM\*, RRT or RRT\* [16] can find paths in configuration space whilst respecting the constraint set and guarantee asymptotic optimality [16]. 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 [40].

Appendix A explains complexity classes which may be helpful in understanding 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 positions from the task, a purely NAMO problem is what remains. Suppose the problem is simplified even further by assuming that every object is an unmoving 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 [29]. That a simplified version is NP-hard indicates the difficulty of finding an optimal path in composite configuration space.

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 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 [14], 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*.

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 unmoving 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.

To summarise, 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*. A search cannot be performed in the composite configuration space because it would require solving an NP-hard problem due to two reasons. 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. A number of researchers tackled this problem. In the following, we provide a categorization and a summary of the methods, advantages and disadvantages of the most relevant state-of-the-art methods.

**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 [39], considering a single manipulation action [4], discretization [31] or a heuristic function combined with a time horizon [31]. 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 [31] and applies it to a robot in a hospital environment [25] to later improve upon her work [32]. The optimal motion planner avoids obstacles in the workspace and respects the kinematic and dynamic constraints of a robot arm [31]. 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 [10]) 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 position.

The optimal motion planner [31] is then converted toward path planning for a nonholonomic mobile robot with a gripper [25]. 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 [33]. 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 [32]. 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 positions, 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 [12, 20, 33, 40, 42] 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. [33]. 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 drive the robot toward a target position eventually. 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 positions 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 [6, 34], NAMO [5, 11, 12, 18, 21, 42], nonprehensile pushing [2, 3, 22, 36, 37, 38]). Combining two topics received little attention from the scientific community, and combining all three topics (to the best of my search) not at all. Table 1.1 presents state-of-the-art literature and which portion of the three topics they include in their research. The most relevant work for local optimal [32] and hierarchical [33] 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 [40]. Note that both relevant works focus on prehensile manipulation, whilst this thesis focuses on nonprehensile push manipulation.

Author	Citation	Learns object dynamics	NAMO		Specify object target poses	
			prehesile	nonprehesile	prehesile	nonprehesile
Ellis et al.	[12]	✓	✗	✓	✗	✗
Sabbagh Novin et al.	[32]	✓	✓	✗	✓	✗
Scholz et al.	[33]	✓	✓	✗	✗	✗
Vega-Brown and Roy	[40]	✗	✓	✗	✓	✗
Wang et al.	[42]	✓	✗	✓	✗	✗
Groote	Proposed Framework	✗/✓	✗	✓	✗	✓

**Table 1.1:** Overview of topics; learning object dynamics, NAMO and specifying target poses in recent literature. This thesis combines the three topics; learning object dynamics, NAMO and nonprehensile pushing.

## 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 [20]. Learned object dynamics are stored in a knowledge base called the *knowledge graph*. The halgorithm, hgraph and kgraph 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  $\langle \text{Origin}, \text{Ground Plane}, Obj, Eq \rangle$  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,  $Obj = (ob_1, ob_2, ob_3, \dots, ob_i)$  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, 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, a number of tasks are designed, which are defined as a subset of all objects with an associated target configurations.

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

where  $Obj_{task} \subseteq Obj$ ,  $C_{target} = (c_1, c_2, c_3, \dots, c_k)$  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

To simplify the pushing and learning problem, several assumptions are taken, which are listed below.

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

**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 positions. The order in which objects are pushed toward their target position 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 in Section 6.1 insight is given to remove all assumptions. By removing assumptions completing tasks becomes a harder 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 did not interact with the robot remain at the same position. Completed subtasks are therefore assumed to be completed for all times after completion time.

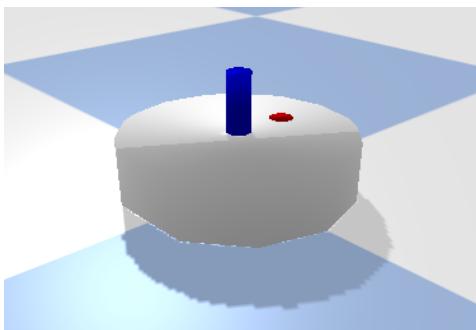
The **perfect object sensor assumption** simplifies a sensor setup, it prevents Lidar-, camera setups and tracking setups with aruco or other motion capture markers. The existence of a single perfect measurement wipes away the need to combine measurements from multiple sources with sensor fusion algorithms, such as Kalman filtering [41].

Certain tasks are only feasible if performed in a certain 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.

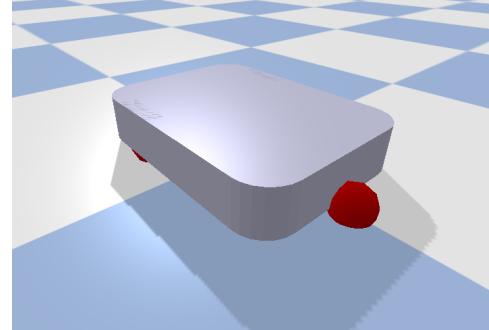
The **3-dimensional world can be represented as 2-dimensional** ensures that the objects in the environment can be projected onto the ground plane. Additionally it ensures that objects do not tip over. In practice, objects will not be higher than the minimum width of the object. This experimental strategy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

### 1.2.3. An Example of Robots and Objects

To get a sense of what the robots and the objects look like, see the two robots that are used during testing in Figure 1.1. And among many different objects, two example objects are displayed in Figure 1.2.

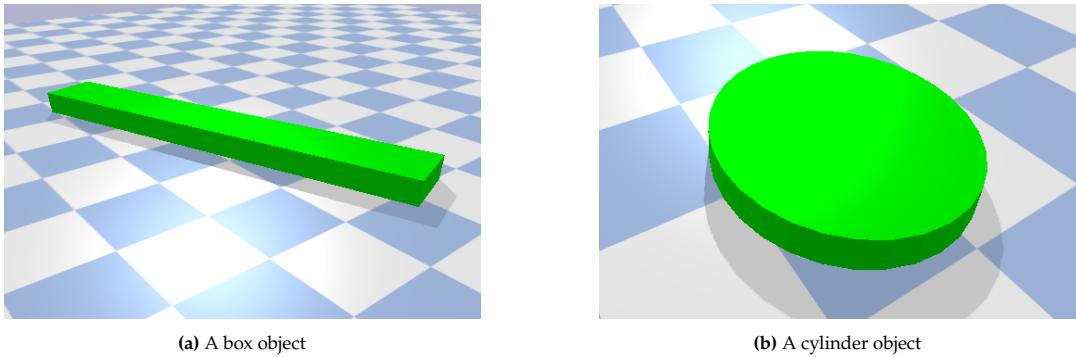


(a) The holonomic point robot with velocity input in  $x$  and in  $y$  direction



(b) The nonholonomic boxer robot, the input is forward and rotational velocity

Figure 1.1: Two robots in the simulation environment



**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. These methods and functions are conveniently grouped in Chapter 2. Then the proposed framework is presented and discussed in Chapter 4. Testing the proposed framework is presented in Chapter 5. The last chapter draws conclusions on tests and answering the research questions.

# 2

## Required Background

*This chapter presents the components on which the proposed framework relies. These components are system identification, control methods, planning and fault detection. System identification converts IO data into a system model as described in Section 2.1. In the scope of this thesis, a system model is required by control methods to create stable control and track a reference signal. Control methods are presented 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, motion planning in Section 2.3.2 and manipulation planning in ???. An existing path planner is presented in this chapter. In the next chapter, the planner is extended such that it can detect objects that are blocking the path. Lastly, fault detection halts persisting faulty behaviour and will be elaborated upon in Section 4.3.*

### 2.1. System Identification

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. An overview of advantages and disadvantages for every category of system models is provided in Appendix .1.

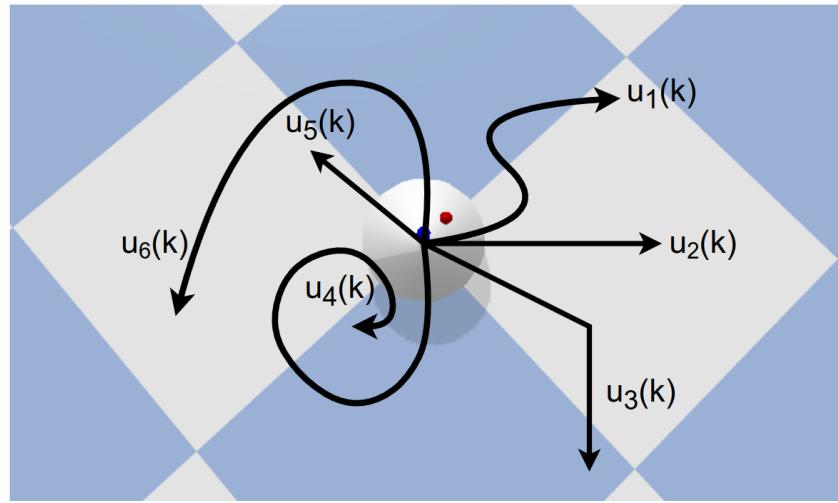
An example of a system model is the drive model for the robot. The model estimates the robot's trajectory when input is sent to the robot. For a specified range of inputs that can be sent to the robot, a system model can predict the possible future states that a robot can reach for a small number of timesteps. 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:

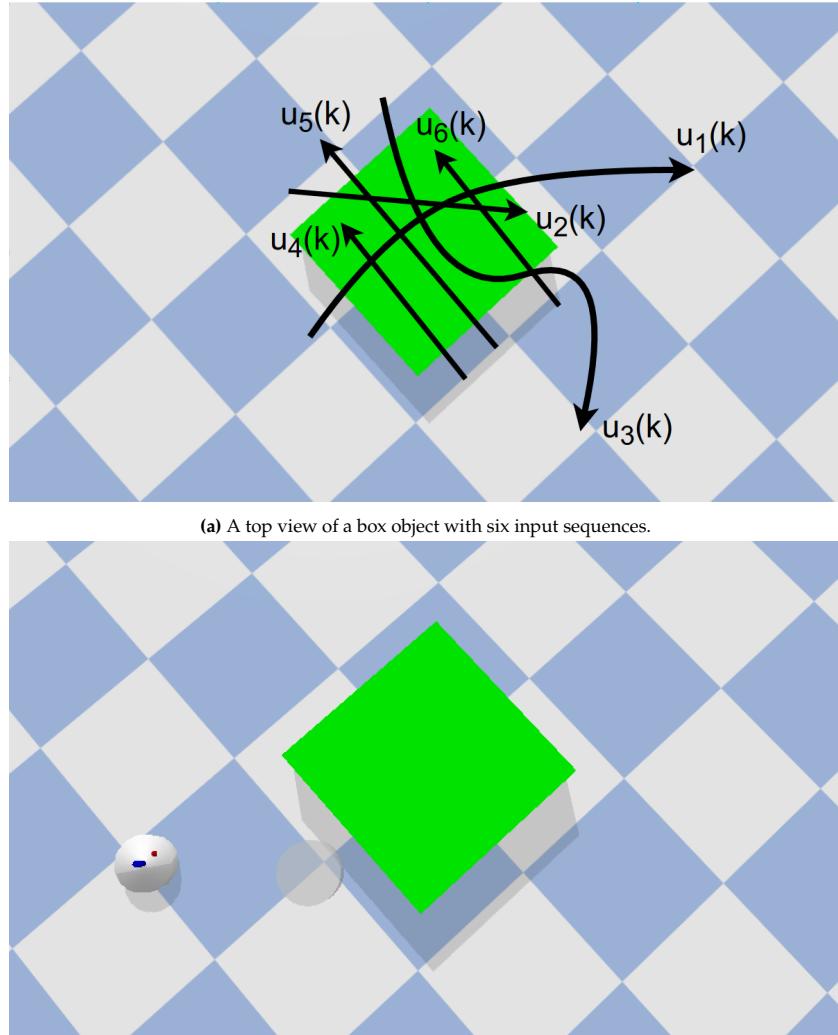
$$\begin{aligned} \text{IO data set} &= [(u_1(k), y_1(k)), \dots, (u_g(k), y_g(k))] \\ &= [(u_1(1), \dots, u_1(a), y_1(1), \dots, y_1(a)), \dots, (u_g(1), \dots, u_g(a), y_g(1), \dots, y_g(b))] \end{aligned}$$

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 sequences are sent toward the robot as input to collect the output response of the robot.

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.



**Figure 2.2:** To collect IO data sequence, the robot must first drive toward the initial pose at the start of an arrow. The corresponding input is sent toward the robot, and the robot and the object's output response is collected.

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 excited 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 timewise 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?”.

Related to system identification is *system classification*, a symbolic model that indicates the affordance 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 4.2.3.

## 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 leverage the prediction system models provide to perform actions. 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. 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 [28]. The best feasible input is found every time step by optimizing the objective function whilst 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. System models for driving the robot can be estimated with Linear Time-Invariant (LTI) models without compromising on model accuracy, making MPC controllers a suitable candidate for driving actions. A more elaborate description of MPC control can be found in Appendix B.1.

**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, [24]. 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 Appendix B.2.

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 to which Section 4.4.3 is dedicated.

## 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 [44]. If a path exists, the double tree optimised Rapidly-exploring Random Tree (RRT\*) planning algorithm is responsible for finding a path from starting point to a target point in configuration space whilst respecting applicable constraints [5].

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

Global Planning:	Planning for a task by focussing on one single subtask at a time.
Local Planning:	Validating if two closeby configurations that are maximal <i>step_size</i> 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.

**Table 2.1:** Terminology related to planning, terms are presented in the left column with corresponding description in the right column.

### 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 nonholonomic constraints. By neglecting nonholonomic constraints, the path estimator is magnitudes faster than the planner; the planner is responsible for checking the nonholonomic constraints.

**Discretizing 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 have 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  $\theta$  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 configuration space for a circular object is defined as:

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

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

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-, obstacle-, 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 obstacle 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 obstacle space will evaluate as obstacle space since obstacle space is more important than the unknown space.

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

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

Defined for:

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

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

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

Defined for:

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

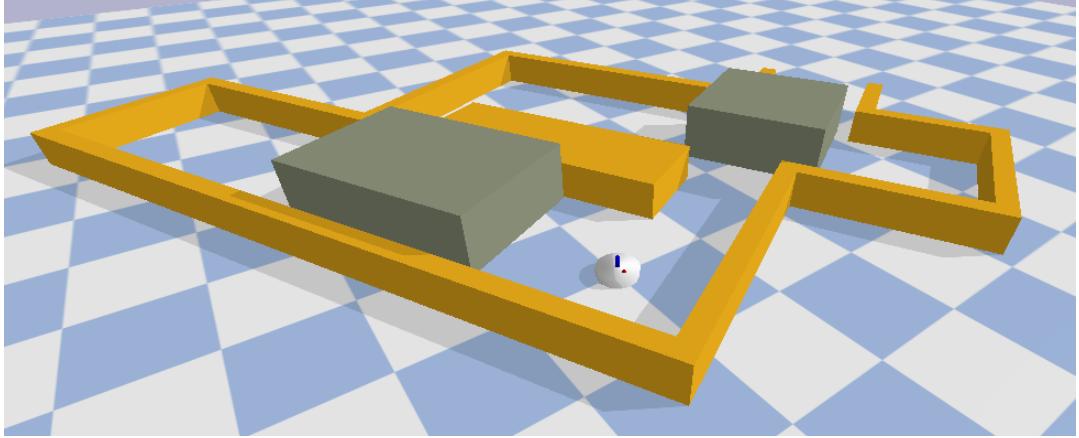
The configuration space for a rectangular object is defined as:

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

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

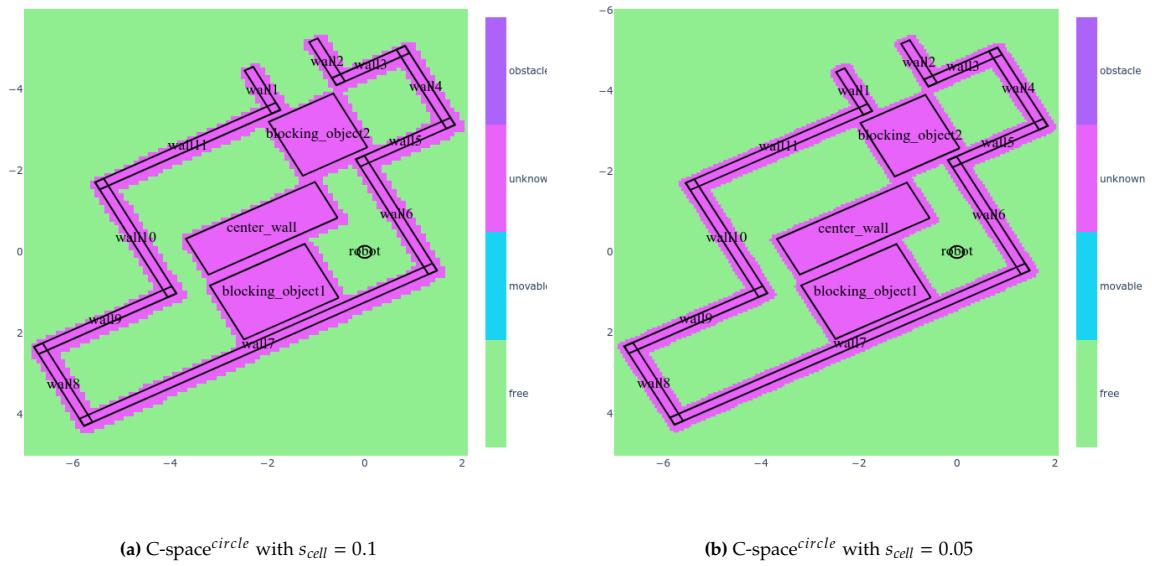
Similar to  $f_{\text{chart\_to\_idx}}(x, y)$  and  $f_{\text{idx\_to\_chart}}(i, j)$  the mapping functions  $\text{chart\_to\_idx}(x, y, \theta)$  and  $f_{\text{idx\_to\_pose}}(i, j, k)$  exist and map between the  $(x, y, \theta)$  pose and the  $(i, j, k)$  indices.

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



**Figure 2.3:** Environment with the point robot, unmoving yellow walls and movable brown boxes. The robot classifies all objects as unknown since it is unaware of the objects' classes.

The point robot has a cylindrical shape; thus, a 2-dimensional configuration space is created and can be seen in Figure 2.4.



**Figure 2.4:** The configuration space for the point robot with different cell sizes. The robot environment corresponds to the environment presented in Figure 2.3.

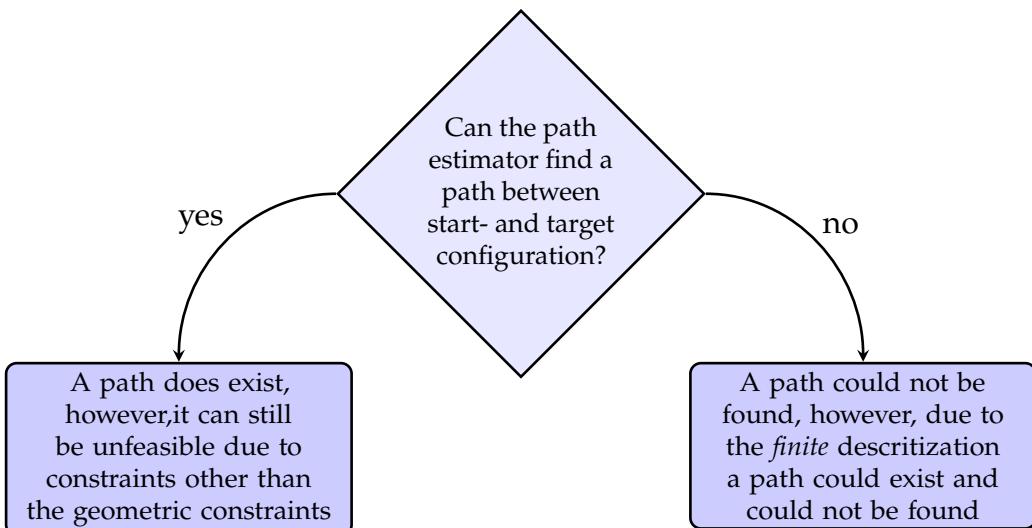
The resolution of the configuration space at Figure 2.4b is higher compared to Figure 2.4a because of a smaller grid size. A high resolution is better at detecting paths through small corridors and tight corners, but it comes at the cost of a more extended creation and search time.

**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 obstacle 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 [9] on the discretized 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 nonholonomic 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 [44]. 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 centre 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 [44]. The following flowchart neatly presents why the existence of paths is an *estimation* rather than a guarantee.



**Figure 2.5:** Flowchart displaying the drawbacks when checking if there exist a path in configurations space between a start- and target configuration. These drawbacks are the reason why path existence can only be *estimated* rather than guaranteed.

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. Motion 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 obstacles 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.” [21]”*

The path planner is defined with the following tuple:

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

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:

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

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:

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

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 analysed in which the path planner adds a single node to the connectivity graph. The variables and functions are neatly grouped after discussing the exmaple in Table 2.2.

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, untill the stopping criteria is reached with the *NotReachStop* function.

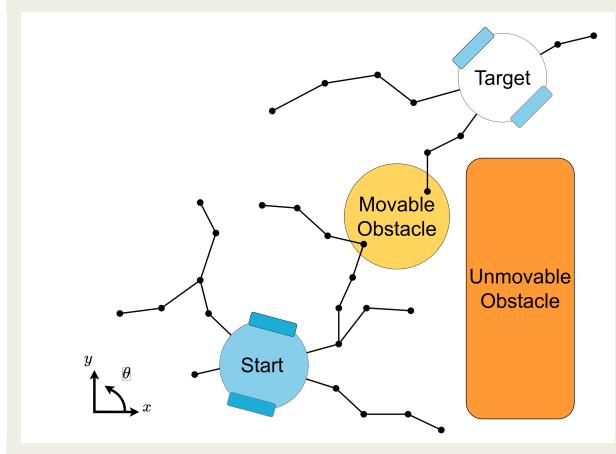
**Algorithm 1** Pseudocode for double tree RRT\* algorithm. The pseudocode is split into three parts, Algorithms 2 to 4 that correspond to the blue, yellow and green colored sections.

```

1:  $V_{MP} \leftarrow x_{init}$ 
2: while NotReachStop do
   ▶ Create, check and project a new random node
   ▶ Find and connect new node to parent node
   ▶ Check if the newly added node can lower cost for nearby nodes
3: end while
```

The example that adds a single node starts in Figure 2.6. In this example, one node is added to the starting connectivity tree. Adding this node involves many 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 to the start or target connectivity tree. The algorithm explores configuration space by growing these connectivity trees.



**Figure 2.6:** Snapshot of the configuration space during a search from start to the target configuration.

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  $\text{Sample}_{random}$  creates a random node in free- or unknown space. This random node is projected to the closest existing node using two functions. First, the  $\text{Nearest}(x, V)$  returns the nearest nodes from  $x$  in  $V$ . Second, the  $\text{Project}(x, x')$  projects  $x$  toward  $x'$ . Newly added configurations are added structurally, guaranteeing an optimal path is found with infinite sampling [5]. Where optimality is defined as the path with the lowest possible cost. The projected random node can reside in free- or obstacle space, ensuring that the node is in free space is validated with the  $\text{CollisionCheck}(x)$  function that returns true if  $x$  is in free space.

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

---

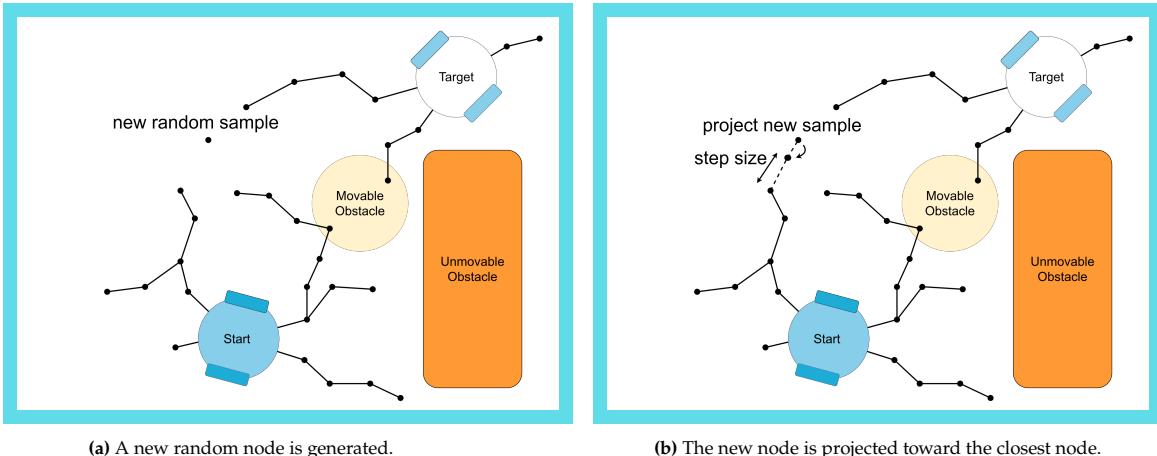
```

1:  $Cost_{min} \leftarrow +\infty$ 
2:  $v_{rand} \leftarrow Sample_{random}$ 
3:  $v_{nearest} \leftarrow Nearest(v_{rand}, V_{MP})$ 
4:  $v_{temp} \leftarrow Project(v_{rand}, v_{nearest})$ 
5: if CollisionCheck( $v_{temp}$ ) then
6:    $v_{new} = v_{temp}$ 
7: else
8:   Continue
9: end if

```

---

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 2.7b 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 obstacles 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 obstacle space. The second tuning parameter is the *search size*, which is a subspace around the newly noded node (see, Figure 2.9).



**Figure 2.7:** Create, find and connect new node 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 totalcost is sought, defined as:

$$totalCost = CostFromInit(v_{near}) + Distance(v_{near}, v_{new})$$

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.

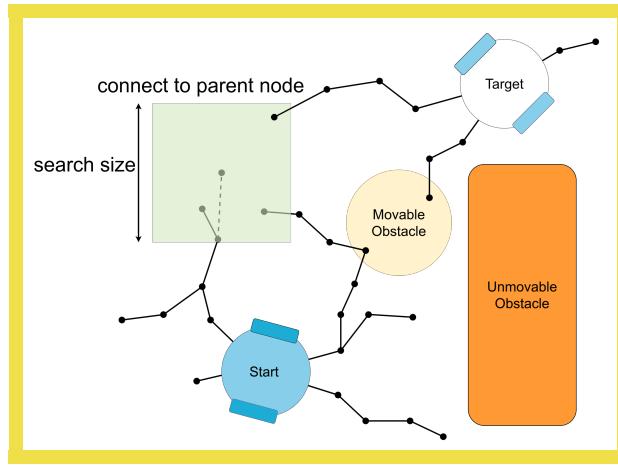
---

```

1:  $X_{near} \leftarrow \text{NearestSet}(v_{new}, V_{MP})$ 
2: for  $v_{near} \in X_{near}$  do
3:    $Cost_{temp} \leftarrow CostFromInit(v_{near}) + Distance(v_{near}, v_{new}) + ObjectCost(v_{near}, v_{new})$ 
4:   if  $Cost_{temp} < Cost_{min}$  then
5:      $Cost_{min} \leftarrow v_{temp}$ 
6:      $v_{minCost} \leftarrow v_{near}$ 
7:   end if
8: end for
9: if  $Cost_{min} == \infty$  then
10:   Continue
11: else
12:    $V_{MP}.add(v_{new})$ 
13:    $E.add(v_{minCost}, v_{new})$ 
14: end if

```

---



**Figure 2.8:** The new node is connected to the node in search space that results in the lowest cost.

The new node is connected to a parent node as can be seen in Figure 2.9. 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 in the same tree or not. 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 totalCost for that node. The rewire procedure changes the parent node by removing and adding an edge, rewiring can be visually seen in Figure 2.9a. The nodes in the set of nearest nodes that are in the other three,

and checks if the closeby node is in the same tree as the newly connected node, or in the other three with the

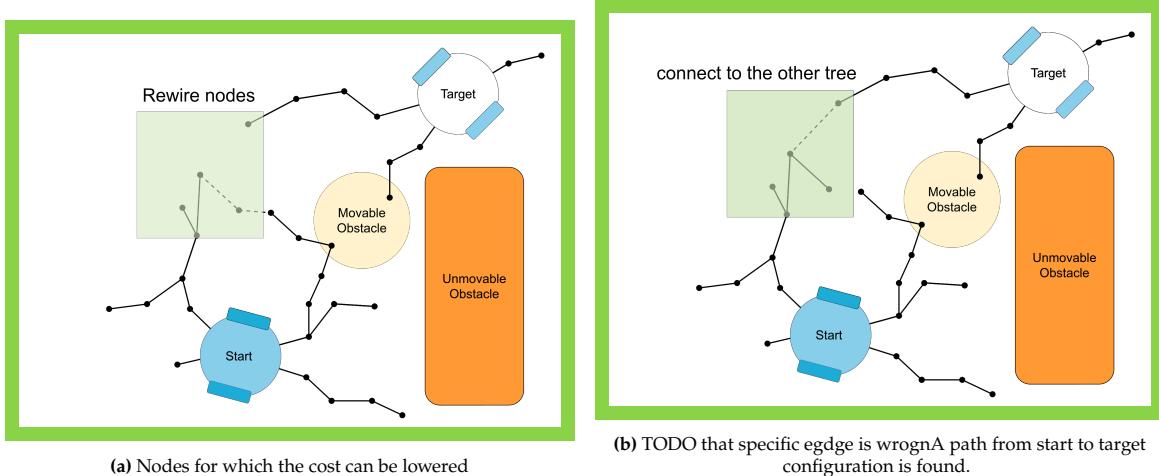
Increasing the search size improves the choice of the parent node and improves cost due to rewiring, but it also exponentially increases computation time.

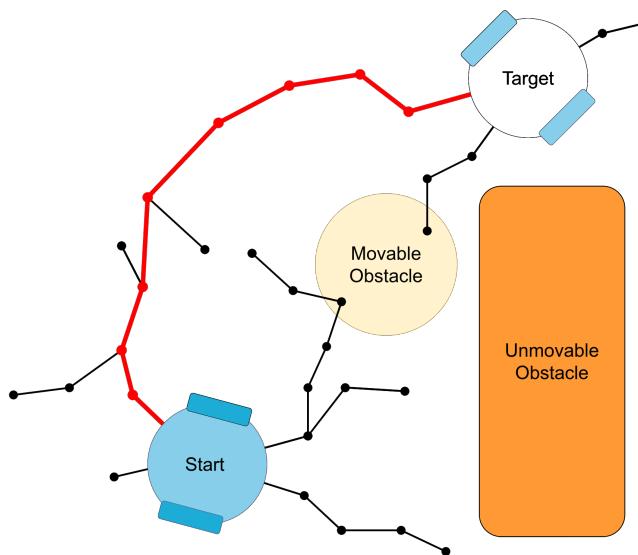
**Algorithm 4** Pseudocode to check if the newly added node can lower cost for nearby nodes.

```

1:  $Cost_{path} \leftarrow +\infty$ 
2: for  $v_{near} \in X_{near}$  do
3:   if  $InSameTree(v_{near}, v_{new})$  then
4:      $Cost_{temp} \leftarrow CostFromInit(v_{new}) + distance(v_{new}, v_{near}) + ObjectCost(v_{new}, v_{near})$ 
5:     if  $Cost_{temp} < CostFromInit(v_{near})$  then
6:        $E.rewire(v_{near}, v_{new})$ 
7:     end if
8:   else                                     ▷ Add lowest cost path to the list of paths
9:      $Cost_{temp} \leftarrow CostFromInit(v_{new}) + distance(v_{new}, v_{near})$ 
10:     $+CostFromInit(v_{near}) + ObjectCost(v_{new}, v_{near})$ 
11:    if  $Cost_{temp} < Cost_{path}$  then
12:       $Cost_{pathMin} \leftarrow v_{temp}$ 
13:       $v_{pathMin} \leftarrow v_{near}$ 
14:    end if
15:   if  $Cost_{pathMin} == \infty$  then
16:     Continue
17:   else
18:      $P.addPath(v_{new}, v_{pathMin}, Cost_{pathMin})$ 
19:   end if
20: end for

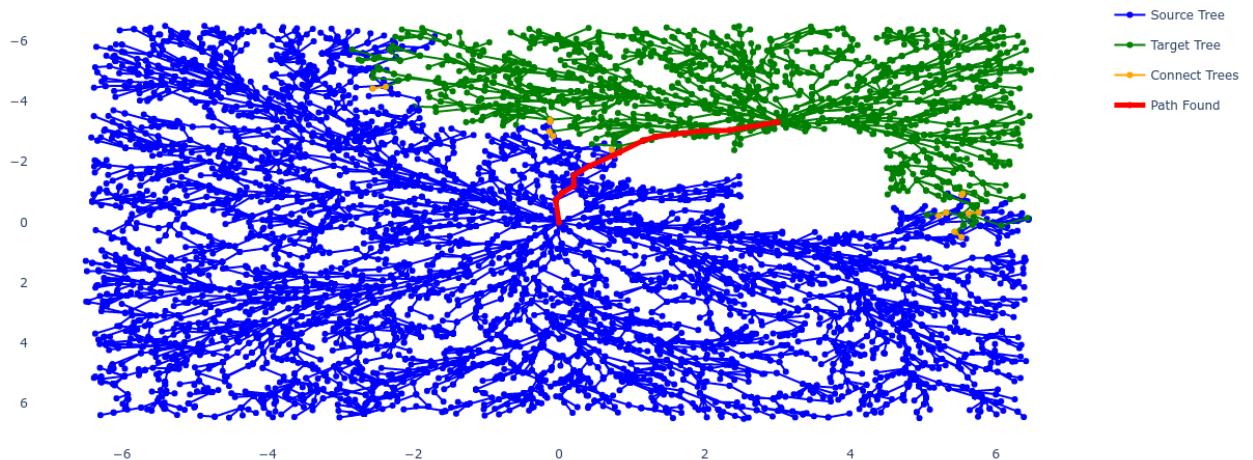
```

**Figure 2.9:** Check if the newly added node can lower cost for nearby nodes.



(a) Schematic Motion planner that found a path from start to target node marked in red.

Connectivity Trees



(b) A visualization of the implemented RRT\* algorithm as the stopping criteria were reached for an environment with two unmovable objects. The start connectivity tree is shown in blue, the target connectivity tree in green, connecting the start- to target connectivity en yellow and the lowest cost path in red.

**Figure 2.10:** Comparing schematic example to a visualization of the implemented algorithm.

$x$ :	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-, movable- or unknown space
$ObjectCost(x', x)$ :	Returns a fixed additional cost if $x$ enters movable- or unknown space from $x'$ , otherwise returns 0
$Distance(x, x')$ :	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

Table 2.2: Functions used by the Algorithm 1.

## 2.4. Monitoring Metrics

In the next chapter, the proposed framework is discussed, for this section, it is important to know that the proposed framework contains the halgorithm which is responsible for finding action sequences, motion/manipulation planning, and execution of drive and push actions. During the execution time of a drive or push action, the hypothesis algorithm is unable to perform any other action. This blocking behavior has some implications, mainly a controller can steer the system to a state from which it cannot independently reach the target state, as a result, it will never halt. For example, a controller tries to drive the robot toward a target state but there is an unmovable obstacle in the way. Another example is the controller is closed-loop unstable and never reaches its target state. Both examples do not occur in well defined simulation environments, because of the *closed-world assumption* defined in Section 1.2.2. In the real world, an unexpected blocking obstacle or unstable controller is more likely to occur.

Detecting controller faults is a large robotic topic [17], 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. The first monitoring metric is Prediction Error (PE), the second monitoring metric is Tracking Error (TE). Definitions of the monitoring metrics are summarised in Table 4.3, it also provides insight in which monitoring metric would catch what faulty behavior. The PE and TE are 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 actual configuration.

The Prediction Error can be described as:

Every time step a prediction one step into the future is made with the use of the system model and system input. Then the system input is applied to the system and the actual configuration is measured. The difference between predicted and actual configuration is defined as:

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

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

The Tracking Error can be described as:

A path consists of a list of configurations, a controller tracks the path by steering the system to the upcoming configuration in the path when reached the configuration is updated to the next configuration in the path. The difference between the current configuration and the current configuration the controller tries to steer toward is the TE.

$c_{target}$  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.

Prediction Error (PE)	During executing 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. Single collisions 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 error during execution and the edge failed.
-----------------------	--

Corrado: This relies on many heuristics that have not been explained properly or even enough to be reproduced

**Table 2.3:** Monitor metrics used to monitor if a fault occurred during the execution of an edge

# 3

## Planning in four subspaces

make sure nodes are in this thing, not samples

This chapter presents an existing motion planner [5] that is extended to incorporate movable and unknown spaces next to the conventional free and obstacle spaces. The modification incentivizes the planner to find a path in free space but can pass through unknown or movable space as a last resort. The robot should first remove the blocking objects if a planned path crosses an unknown or movable subspace.

Finding a path between the start and target configuration for pushing applications whilst avoiding collisions is referred to as *manipulation planning*. This and the upcoming subsection present two new sample-based planners (one for drive and one for push applications); they are based upon an existing double tree RRT\* planner [5]. The existing planner plans in free and obstacle space; a modification extends the planner to incorporate movable and unknown space. First, this section presents the new motion planning algorithm, and the next section, ?? dedicates itself to manipulation planning. The new planning algorithms are sample-based, which can be described as.

Generally, the configuration space consists of 2 subspaces, free- and obstacle space. The configuration space in this thesis consists of 4 subspaces: free, obstacle, unknown and movable space. A dedicated path planning algorithm has been developed to solve planning problems for such a configuration space with four subspaces. The newly developed path planner extends the existing double tree RRT\* algorithm [5].

The goal of the motion planner is to find a path between the start and target configuration that results in the lowest totalPathCost, defined as:

$$\text{TotalPathCost} = \text{PathCost} + \text{MovableSpaceCost} + \text{UnknownSpaceCost}$$

The MovableSpaceCost and UnknownSpaceCost correspond to a fixed addition cost for a configuration in the path that crosses through movable or unknown subspace, respectively. 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 movable space, MovableSpaceCost will be 0, equivalent to unknown space and UnknownSpaceCost. Optimizing the path for the lowest cost incentivizes the motion planning algorithm to find a path around unknown or movable objects.

The third and fourth tuning parameters are the fixed costs for crossing through movable or unknown space, Figure 3.1 clearly shows the effect of varying such costs.

### Tuning Parameters

Corrado: Did not see any algorithm so far. It's hard to map these parameters to something I did not see yet jk

The algorithm has four tuning parameters that can be tweaked; first, the *step size*, the maximal normalized distance between connected samples in the connectivity trees, see Figure 2.7b

**Corrado:** Put more of simple images as you go instead of a gigantic one later on

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.

**Corrado:** Additionally, .. not Not a sentence

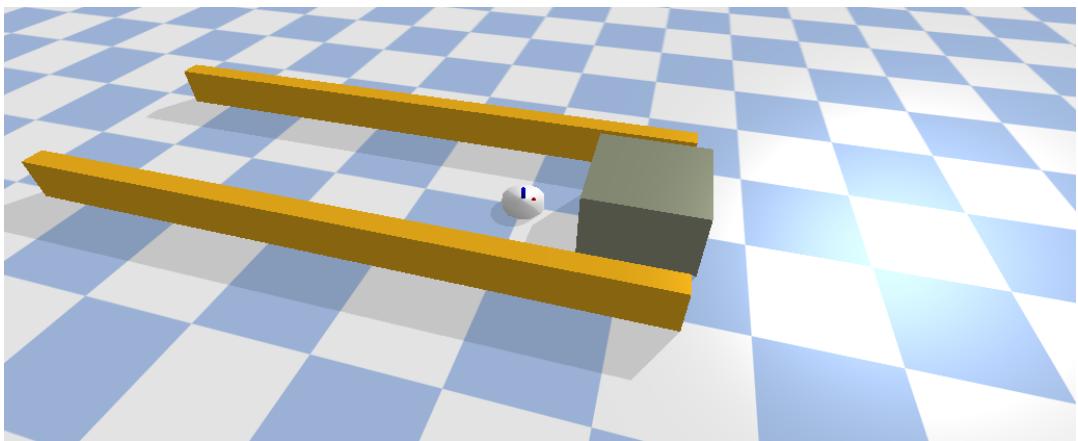
Additionally, the path has an increased chance of collision with obstacles because, for two connected configurations in a path, the individual configurations can both lie in free space. The space between configurations is not checked, and an obstacle could be in between the configurations, especially when cutting corners around obstacles. The second tuning parameter is the *search size*, which is a subspace around the newly sampled sample (see, ??). In this subspace, a parent node is sought, and rewiring occurs. A parent node connects a new node with an edge to a connectivity tree. After connecting the new node to its parent node, rewiring occurs, changing the parent node by removing and adding an edge. If that results in a lower cost for that node, rewiring can be visually seen in Figure 2.9a. Increasing the search size improves the choice of the parent node and improves cost due to rewiring, but it also exponentially increases computation time. The third and fourth tuning parameters are the fixed costs for crossing through movable or unknown space, Figure 3.1 clearly shows the effect of varying such costs.

The Pseudocode of the proposed algorithm is provided in ???. The variables and /Rfunctions used are elaborated upon in the following Table 2.2.

**Corrado:** above check, take what is needed only

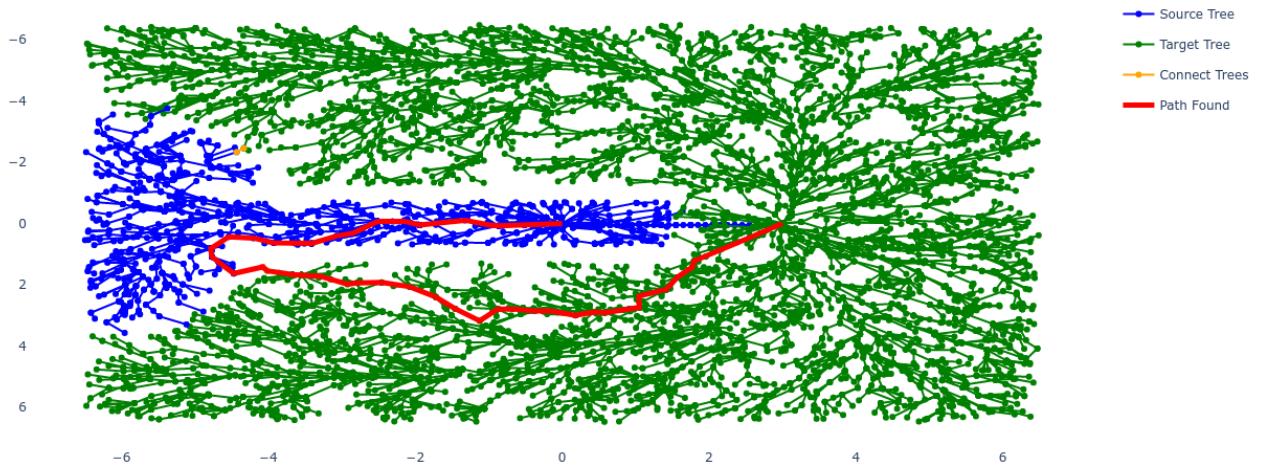
The added fixed cost for a path crossing through a movable or unknown object motivates the motion planner to find the shortest path around objects but prefers moving an object over making a large detour. Tuning the additional fixed cost for a path crossing through movable or unknown space balances the robot's decision between how long 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, which is why the additional cost to remove an unknown object is higher than that of a movable object.

**Corrado:** So what's your approach? Is there a systematic way or approach you used to tune this?



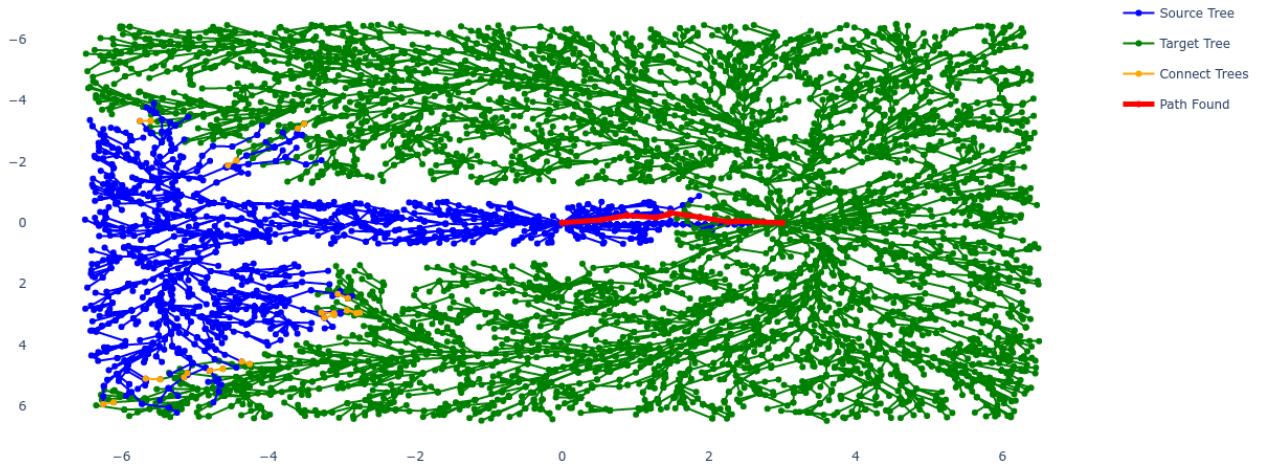
(a) Robot environment with the point robot, two yellow unmovable walls and an unknown brown box.

Connectivity Trees



(b) planned path around the brown box and yellow obstacles, with unknownSpaceCost = 2.

Connectivity Trees



(c) planned path going through the brown box, unknownSpaceCost = 0.5.

**Figure 3.1:** The robot tasked with driving toward the other side of the brown box.

Corrado: a top view would be nice of that above thingy... easy to map to the ones just below

The

Corrado: It's not a property of your proposed method per se but rather one of double rat star right?

proposed motion planning algorithm searches the configuration space from the start to target connectivity trees. Exploring faster than the single tree RRT\* algorithm

Corrado: Do you have a reference for this?

because two trees grow and explore faster than a single tree. The proposed algorithm rewrites nodes, lowering the cost for existing paths and convert to the optimal lowest-cost path with infinite sampling. System constraints are ensured by the *ReachabilityCheck*

Corrado: ReachabilityCheck, is n'to explained at all

that validates if a node is reachable from another node using system models.

The proposed algorithm yields feasible paths (according to the system model used to check reachability) that respect the system constraints. The algorithm prevents planning a path through blocking objects except when no other option is available or a large detour can be prevented. There are no performance tests taken on the modified motion planner other than visual inspection.

Corrado: So why should we believe the things you said so far about your proposed algorithm?

For performance tests and comparison with equivalent sample-based state-of-the-art motion planners, see [5]. Now that motion planning is discussed for drive actions, manipulation will be discussed for push actions.

display which sections are changed compared to previous algorithm

**Algorithm 5** TOOD: 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 colored sections.

```

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

```

# 4

## The Hypothesis and Knowledge Graph

*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 the composite configuration space for action sequences to complete a specified task. Section 4.2.1 is dedicated to the halgorithm and the hgraph. The halgorithm additionally collects new knowledge of the environment. Gathered new environment knowledge is stored in the knowledge graph, and used for suggesting actions from experience, defined in Section 4.4.1.*

### 4.1. Overview of the proposed framework

Figure 4.1 presents a schematic overview of the interconnection of the knowledge-, hypothesis graph and the robot environment. A more in-depth analysis of the hypothesis graph is given in Section 4.2 and for the knowledge graph in Section 4.4.

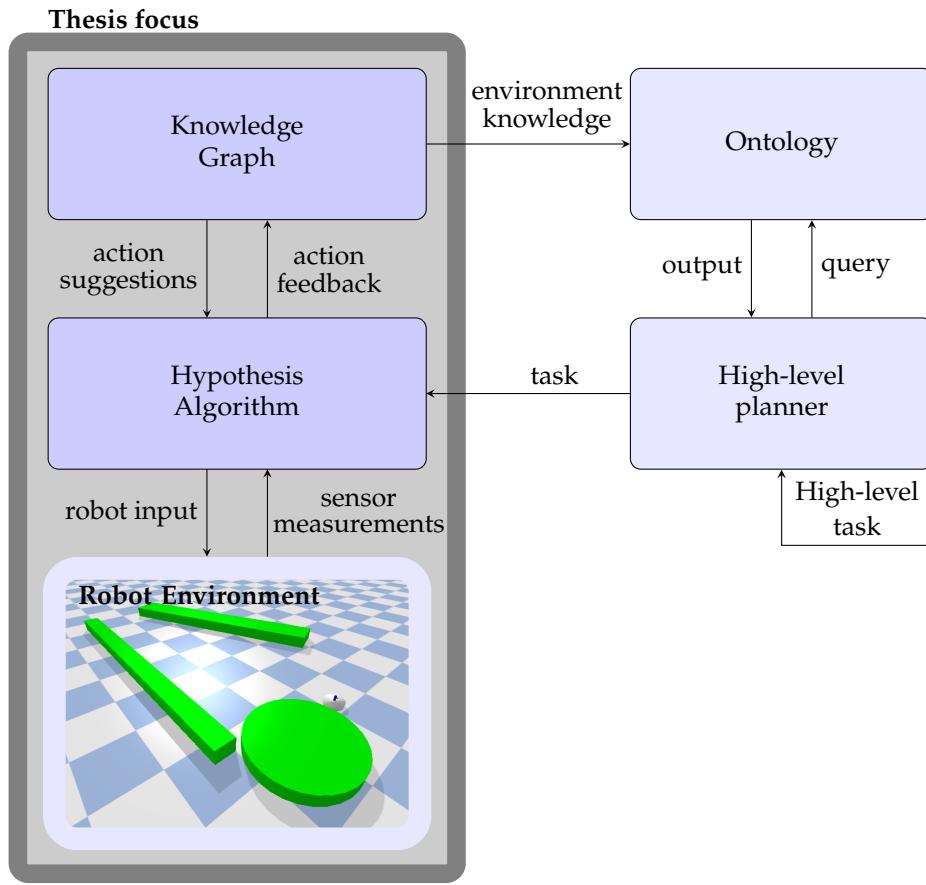


Figure 4.1: Flowchart representation of the proposed framework.

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 4.2.3. 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 4.2.3, 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 of the hgraph

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  $x, y \in \mathbb{R}$ ,  $\theta \in [0, 2\pi]$

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**,  $V_{id}^{obj} = \langle status, obj(k) \rangle$ .

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 two categories because of very different goals, *system identification edges* that have as a goal to collect IO data and generate a system model, and *action edges* that steer a system toward a target configuration. The edges are formally defined as:

A **identification edge**,

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

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**,

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

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.

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:  $V_H = \{V_H_i^{obj}\}$ ,  $E_H \in \{e_{(i,j)} | E_{Hi}, E_{Hj} \in \{V_H^{obj}\}, i \neq j\}$ .

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

**Status, Types and Lifetime of edges** 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).

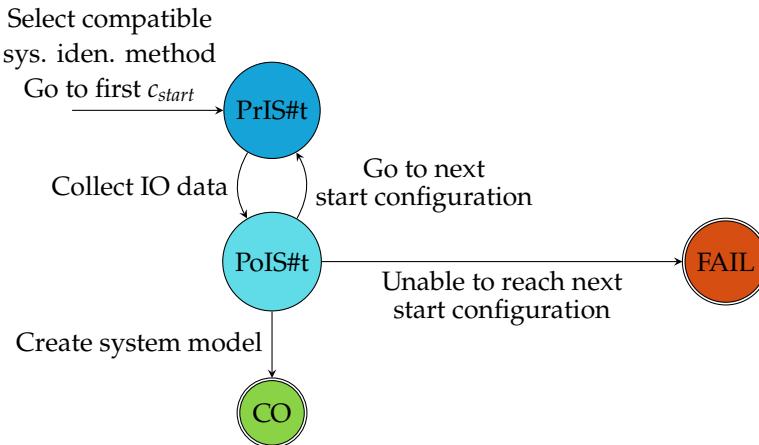


Figure 4.2: FSM displaying the status of an identification edge

PRE INPUT SEQUENCE    Go to target configuration to apply the input sequence.  
number t (PrIS#t):

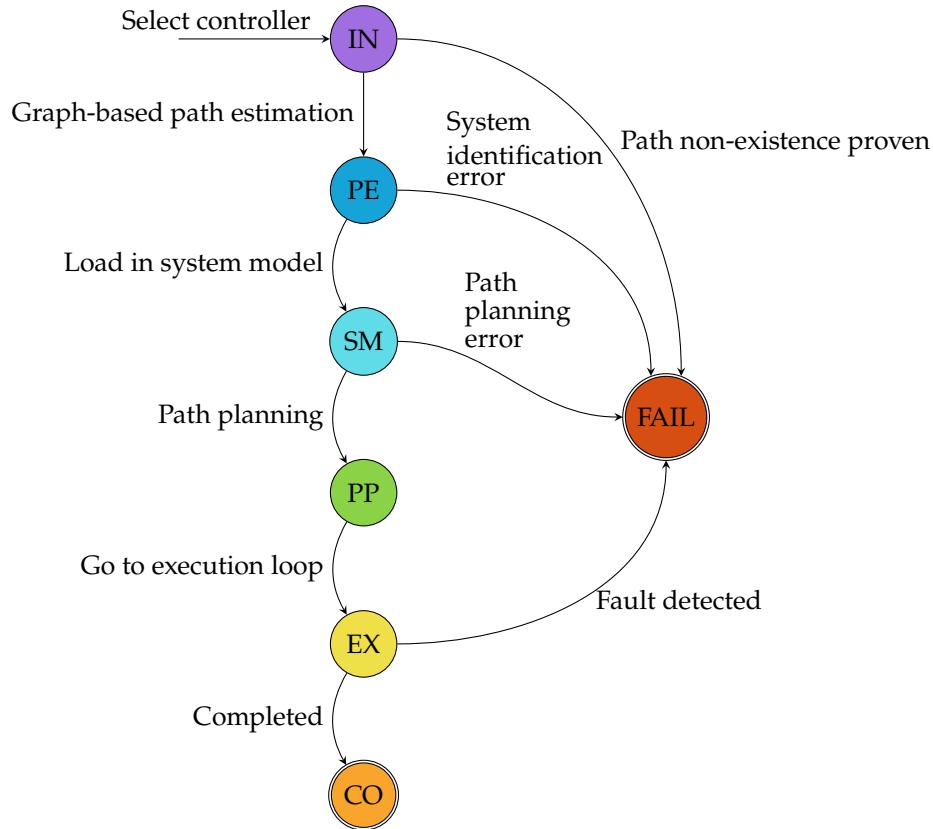
POST INPUT SEQUENCE    Collect the output sequence.  
number t (PoIS#t):

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:** FSM 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.

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.

### 4.2.2. Examples

Before displaying example hgraph's, a legend is presented below.

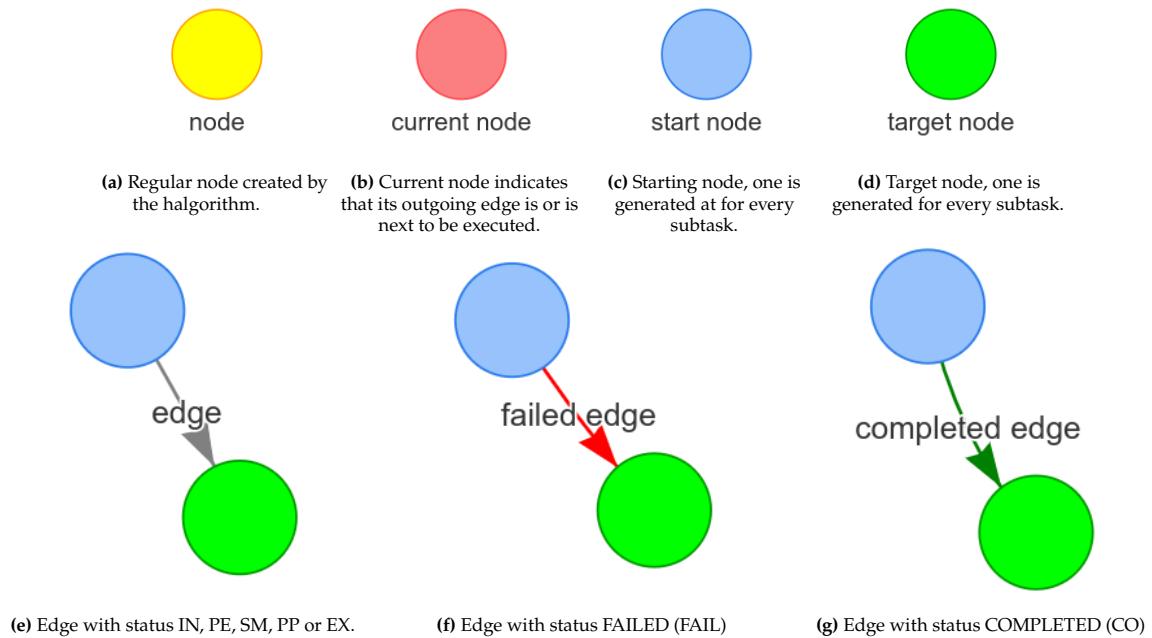


Figure 4.4: Legend for hgraph's nodes and edges

This chapter has terminology that is conveniently grouped in the following table.

Task:	Tuple of objects and target configurations. $\text{task} = S = \langle Obj_{task}, C_{targets} \rangle$
Subtask:	A single object and a single target configuration. $\text{subtask} = s = \langle obj_{subtask}, c_{target} \rangle$
Object Class	Classification assigned to an object. $\text{OBJ\_CLASS} = \text{Unknown} \vee \text{Obstacle} \vee \text{Movable}$
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.
	$\text{NODE\_STATUS} = \text{Initialised} \vee \text{Completed} \vee \text{Failed}$
Node:	A node in the hgraph represents an object in a configuration with reachability indicated with a node status. $\text{node} = v = \langle \text{status}, obj, c \rangle$
Edge Status:	Status of an edge,  $\text{EDGE\_STATUS} = \text{Initialised} \vee \text{PathExists} \vee \text{SystemModel} \vee \text{PathPlanned} \vee \text{Executing} \vee \text{Completed} \vee \text{Failed}$
Edge:	elaborate information on the edge statuses can be found in Figure 4.3.
Hypothesis:	Edge connecting a node to another node in the hgraph or kgraph. $\text{edge} = e = \langle \text{status}, id_{from}, id_{to}, \text{verb}, \text{controller}, \text{dynamic model}, \text{path} \rangle$
Hypothesis Algorithm:	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. $\text{hypothesis} = h = [e_1, e_2, e_3, \dots, e_m], \quad m > 0$
Hypothesis Graph:	Graph-based algorithm that searches for hypothesis in the hgraph to complete subtasks, eventually completing a task.
Knowledge 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. $\text{hgraph} = G^{hypothesis} = \langle V_H, E_H \rangle$
	Collection of nodes and edges. The kgraph acts as a knowledge base and can be queried for an action suggestion. $\text{kgraph} = G^{knowledge} = \langle V_K, E_K \rangle$

**Table 4.1:** Terminology of terms used

add non-failed status

### 4.2.3. 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 is discussed in the proposed halgorithm. Stepwise, the terminology is elaborated with a generated and executed hypothesis for a pushing task. Then two more examples are provided that involve a blocked path and failure of edges. Finally, the pseudocode can be presented, supported by a proposed halgorithm flowchart.

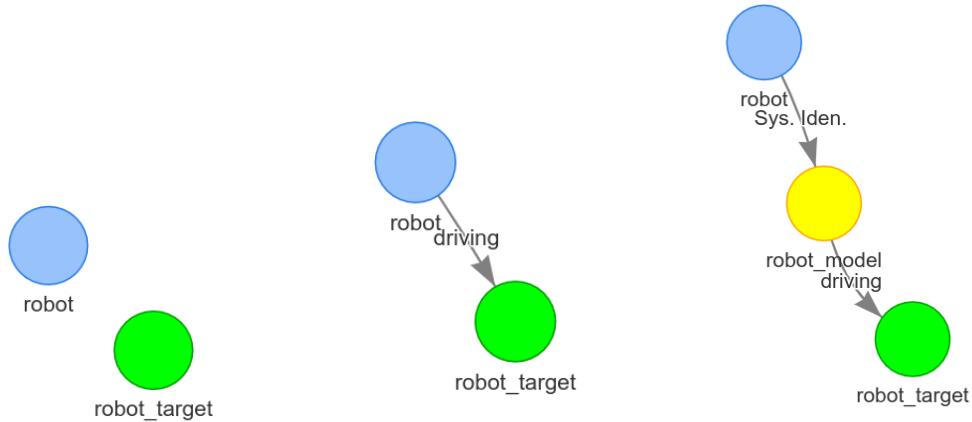
explain the single access point, and rewrite

there remain two subjects to discuss, the tuning parameters and the blocklist.

corrado: what tuning parameters? it's the first time in the text talk about tuning parameters

when all tuning parameters are set, the hgraph is initialized, and a task is provided; there is only a single access point toward the hgraph. a function *respond(observation)* that provides the halgorithm with sensor measurements of the environment with the argument *observation*. the function *respond()* returns control input for the robot. in this thesis, the sensor measurements are the configuration of 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. 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. the rightmost subfigure adds an extra node, the *robot\_model* node and an extra edge, the *sys. iden* edge, the purpose is to generate a system model that describes robot driving and can be used to initialize the controller that resides in the *driving* edge.



**Figure 4.5:** the hgraph in multiple stages when the halgorithm searches for an hypothesis to a drive task.

Now that an hypothesis is created, 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 elaborating upon the search- and execution loop.

**initialisation 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.

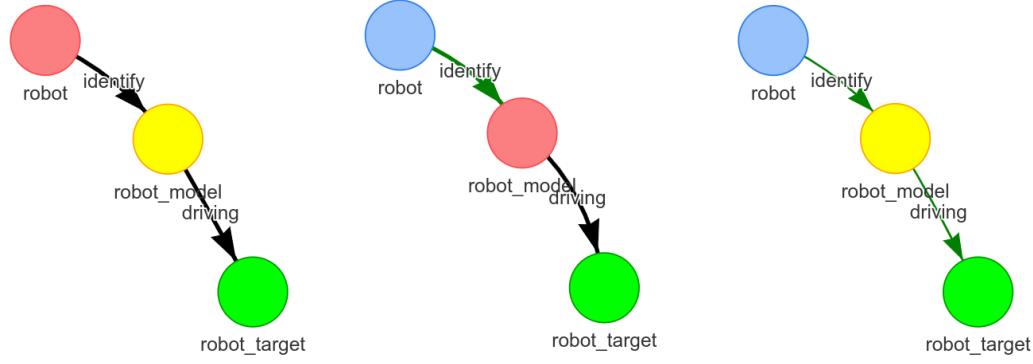


Figure 4.6: executing the hypothesis found in figure 4.5.

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.

**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. later in this chapter a flowchart is discussed, in figure 4.13, here the two main loops can be identified, see figure 4.7.

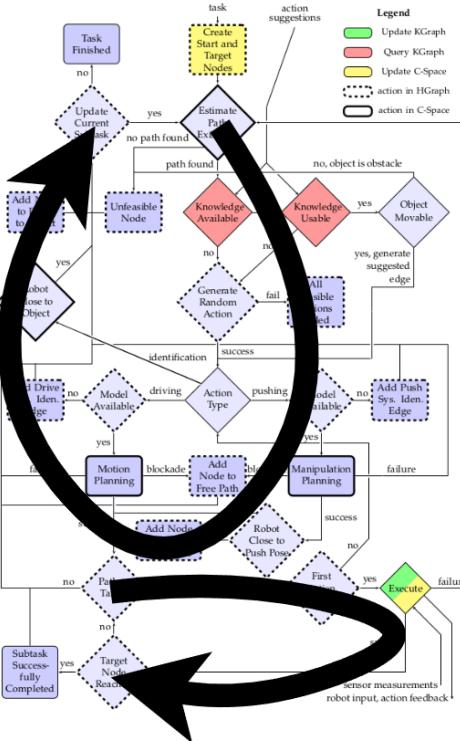


Figure 4.7: the search (above) and execution (below) loop, that make up the two main parts of the proposed halgorithm. the full flowchart is presented in figure 4.13

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.

**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 *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 *robot* node must be connected to the *box* node. these nodes are both starting nodes. The first challenge is to find the two nodes to connect from an unfinished target node.

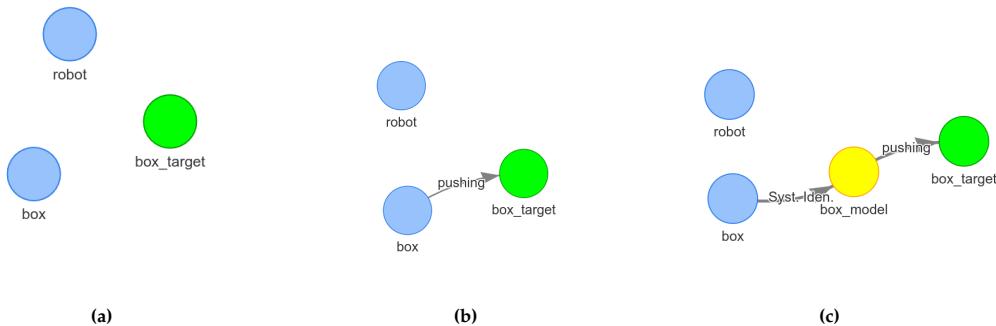
The halgorithm relies on a backward search technique. The backward search technique can be described as *start the search at a goal state and work backwards until the initial state is encountered* [21]. A motivation for a backward search over a forward search is that it might be the case that the branching 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<sub>target</sub>)* function takes the a target node *v<sub>target</sub>* that corresponds to a unfinised 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<sub>box\_target</sub>)* function retuns the *v<sub>box\_target</sub>* node, in figure 4.8b the *gobackward(v<sub>box\_target</sub>)* returns the *v<sub>box</sub>* node.

The *gobackward(v<sub>target</sub>)* 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<sub>target</sub>))* returns node *v<sub>box</sub>*. the nodes that the halgorithm desires to connect are renamed to prevent long function names:

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

$$v_{from} = findcorrespondingnode(gobackward(v_{target}))$$



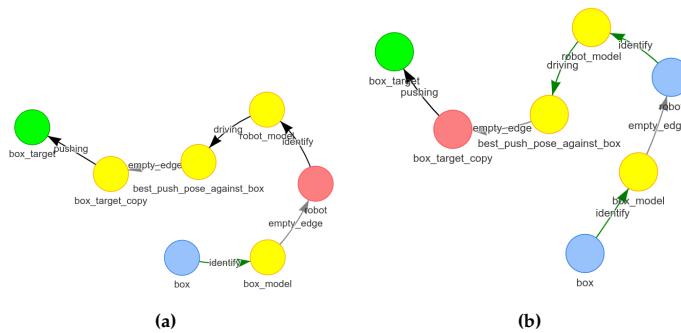
**Figure 4.8:** initialize start and target nodes and the start of a created hypothesis to complete a pushing task.

**creating edges** the `connectwithedge(v1, v2)` 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 *emptyedge* 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 INITIALIZE.

Push action edges require more than only initializing and preparing for execution. The robot must first drive toward a push position against or close to the object to push. By default, creating a push action edge will spawn a  $v_{best\_push\_position}$  node connected with an emptyEdge. The  $best\_push\_position$  depends on the object's planned path. Thus, a path is planned, and then the best push position is determined. Figure 4.9 shows the  $v_{best\_push\_position\_against\_box}$  node that is connected to the  $v_{copy\_box\_target}$ .

make something to test class of objects

**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.



**Figure 4.9:** The hypothesis for a pushing task becomes valid. The hypothesis contains an edge that sends actions to the robot, and thus the hypothesis is executed.

**Preparing edges for Execution** Initialized identification edges are immediately ready to send input toward the robot to collect IO data. 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, indicated in Figure 4.3. After initialization, the action edge must perform path estimation, load in a system model, perform 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 depending on its status. Identification edges are ready for execution when they bear the INITIALIZED status, and action edges are ready when they bear the PATH PLANNED status. The *MakeReady(e)* function takes an edge and takes action depending on its status presented in the following table.

Action edge status	action taken by <i>MakeReady</i> 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. 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 4.14.

**Table 4.2:** Steps were taken to prepare an action edge for execution. An action edge increments its status as indicated in Figure 4.3. The path estimator and path planner are elaborated in Section 2.3 and chapter 3 respectively.

**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. As discussed in the upcoming paragraph, the robot will detect that the object deviated too much from the path. The robot then drives toward the object’s opposite side to again, push it over the target position. 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, the halgorithm remains in the execute loop, or the next edge is not yet ready. The halgorithm goes from the execution loop toward the search loop to prepare the next edge for execution.

updated figures below

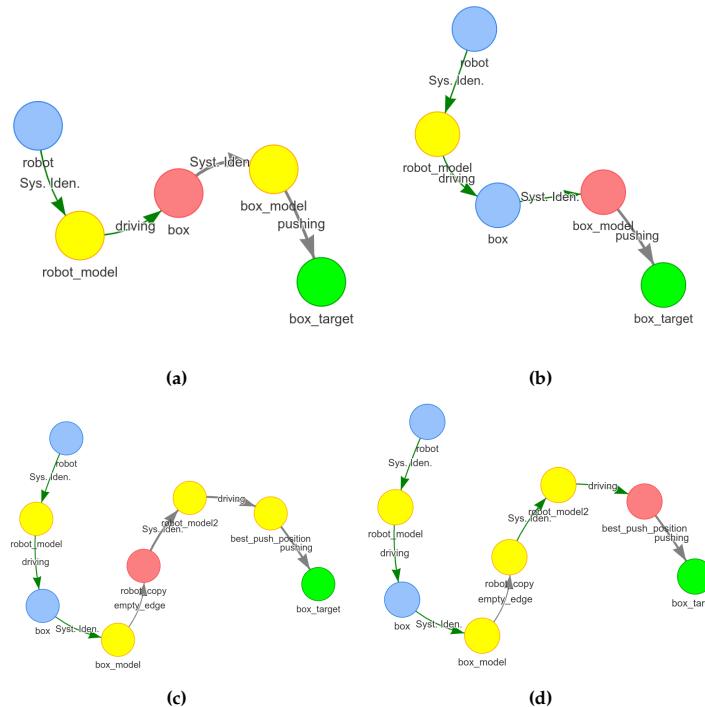


Figure 4.10: todo

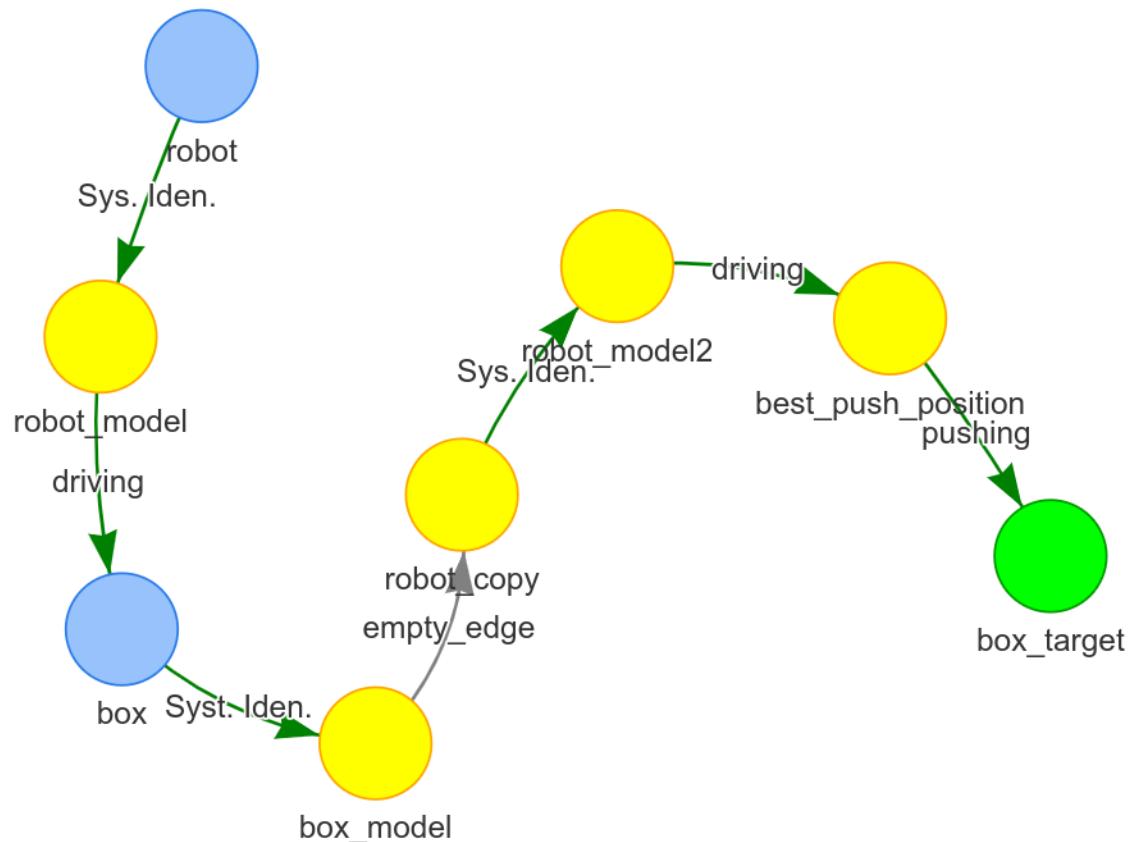


Figure 4.11: TODOhgraph for pushing the green box to the target configuration

## Successfull Completing Hypotheses and Edges

### Fault Detection

This entire section should be merged with the text belwo

## 4.3. Monitoring Metrics

In the next chapter, the proposed framework is discussed, for this section, it is important to know that the proposed framework contains the halgorithm which is responsible for finding action sequences, motion/manipulation planning, and execution of drive and push actions. During the execution time of a drive or push action, the hypothesis algorithm is unable to perform any other action. This blocking behavior has some implications, mainly a controller can steer the system to a state from which it cannot independently reach the target state, as a result, it will never halt. For example, a controller tries to drive the robot toward a target state but there is an unmovable obstacle in the way. Another example is the controller is closed-loop unstable and never reaches its target state. Both examples do not occur in well defined simulation environments, because of the *closed-world assumption* defined in Section 1.2.2. In the real world, an unexpected blocking obstacle or unstable controller is more likely to occur.

Detecting controller faults is a large robotic topic [17], 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. The first monitoring metric is PE, the second monitoring metric is TE. Definitions of the monitoring metrics are summarised in Table 4.3, it also provides insight in which monitoring metric would catch what faulty behavior. The PE and TE are 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 actual configuration.

The Prediction Error can be described as:

Every time step a prediction one step into the future is made with the use of the system model and system input. Then the system input is applied to the system and the actual configuration is measured. The difference between predicted and actual configuration is defined as:

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

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

The Tracking Error can be described as:

A path consists of a list of configurations, a controller tracks the path by steering the system to the upcoming configuration in the path when reached the configuration is updated to the next configuration in the path. The difference between the current configuration and the current configuration the controller tries to steer toward is the TE.

$c_{target}$  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.

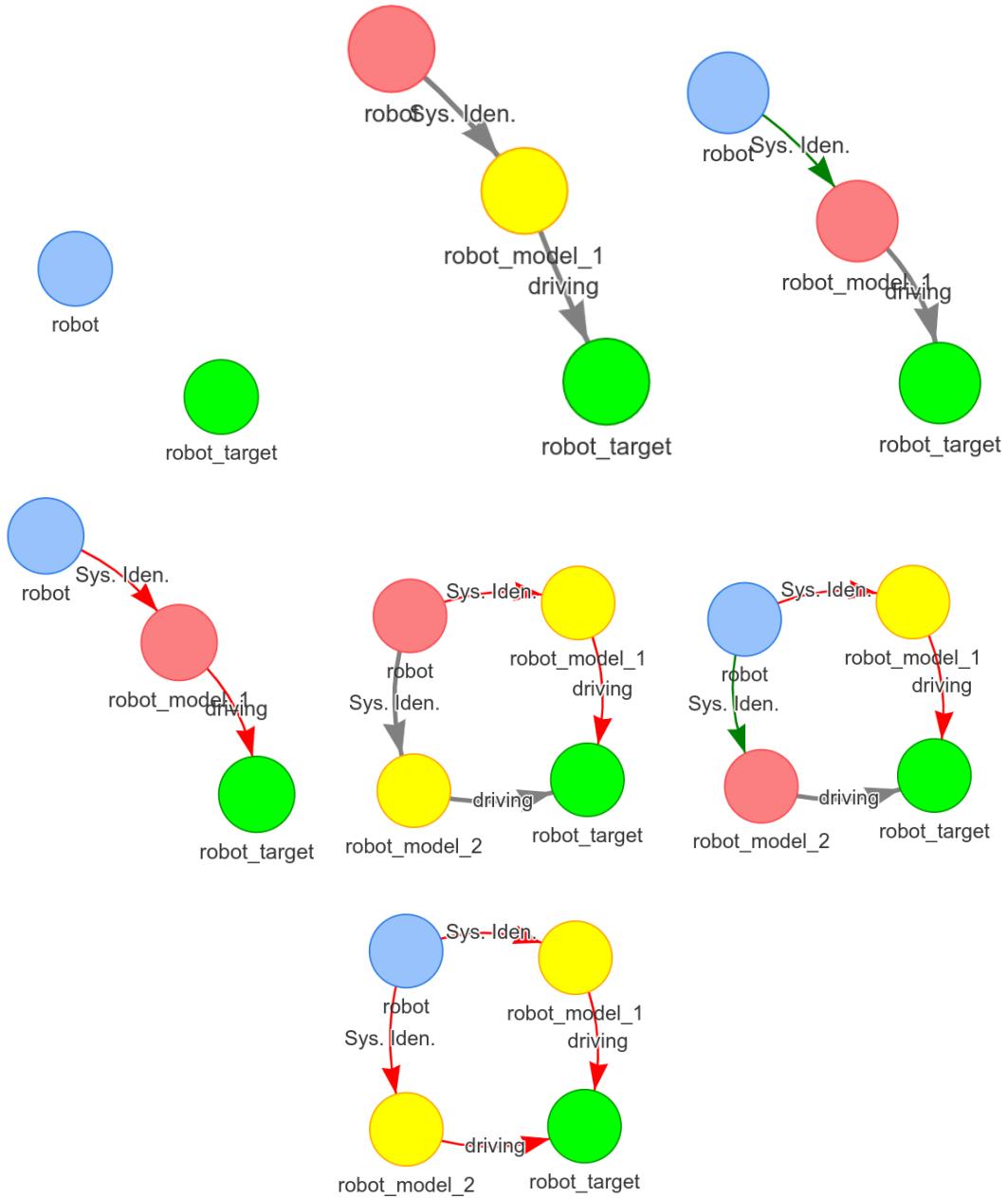
Prediction Error (PE)	During executing a sudden high PE indicates unexpected behavior occurs, such as when the robot has driven into an object which it was not expecting. A high PE, which persists indicates that the robot is continuously blocked. Single collisions are allowed, but when the PE exceeds a pre-defined threshold and persists over a pre-defined time, the hgraph concludes that there was an error during execution and the edge failed.
Tracking Error (TE)	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.

**Table 4.3:** Monitor metrics used to monitor if a fault occurred during the execution of an edge

During the execution of edges, the progress is monitored every timestep by the *FaultDetected(e)* function. A fault is concluded when the Prediction Error (PE) or the Tracking Error (TE) crosses a predefined threshold. 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 predefined thresholds are split for drive and push actions because driving actions, on average, have much lower PE and TE. For drive action edges, if 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. Only a TE is used for push actions, 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.

**The Blocklist** The blocklist prevents the regeneration of failed edges in the hgraph. The infinite loop of creating an edge that fails only to be regenerated is prevented. The blocklist keeps a list of edge parameterization with the node identifier; 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 4.12 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 4.12:** Executing two hypotheses that both failed to complete because a fault was detected. Both 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 4.12, 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.

<i>SubTaskNotFinished(s):</i>	Return False if the subtask $s$ is completed or it is concluded to be unfeasible
<i>IsConnected(<math>v_1, v_2</math>):</i>	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
<i>ReadyForExecution(<math>e</math>):</i>	Return True if the edge $e$ is ready to execute
<i>TargetNotReached(<math>e</math>):</i>	Return True if edge $e$ has not reached its target configuration
<i>FaultDetected(<math>e</math>):</i>	Return True if a fault has been detected during execution of edge $e$
<i>HandleFault(<math>e</math>):</i>	Update edge $e$ status to FAILED and remove edge from hypothesis
<i>SteerTowardTarget(<math>ob</math>):</i>	Update controller with observation $ob$ and compute response that steers the system to target configuration
<i>ReadyForExecution(<math>e</math>):</i>	Check if edge $e$ has the PATH PLANNED status and contains all components to control the system
<i>incrementEdge:</i>	Mark current edge as completed, set next edge in $h$ as current edge
<i>MakeReady(<math>e</math>):</i>	Perform actions to make the edge $e$ ready for execution
<i>goBackward(<math>v</math>):</i>	Find the source node that points toward $v$ through a number of non-failed edges
<i>findCorrespondingNode(<math>v</math>):</i>	Find the node containing the same object as $v$
<i>connectWithEdge(<math>e_1, e_2</math>):</i>	Randomly generate edge between nodes $v_1$ and $v_2$ or use kgraph to suggest an edge

**Table 4.4:** The functions used by the halgorithm.

**Algorithm 6** Pseudocode for the proposed hypothesis algorithm.

---

```

1: for  $s \in S$  do
2:   while  $\text{SubTaskNotFinished}(s)$  do                                ▷ Search Loop
3:     if  $G^{\text{hypothesis}}.\text{isConnected}(s.\text{start}, s.\text{target})$  then
4:       if  $h.\text{currentEdge}.\text{readyForExecution}$  then
5:         while  $\text{TargetNotReached}(h.\text{currentEdge})$  do                      ▷ Execution Loop
6:           if  $\text{FaultDetected}(h.\text{currentEdge})$  then
7:              $\text{HandleFault}(h.\text{currentEdge})$ 
8:             break
9:           end if
10:           $h.\text{currentEdge}.\text{steerTowardTarget}(ob)$ 
11:          if  $\text{TargetReached}(h.\text{currentEdge})$  then
12:            if  $\text{readyForExecution}(h.\text{currentEdge})$  then
13:               $h.\text{incrementEdge}$ 
14:            else
15:              break
16:            end if
17:          end if
18:        end while
19:      else
20:         $\text{makeReady}(h.\text{currentEdge})$ 
21:      end if
22:    else
23:       $v_{\text{localtarget}} \leftarrow G^{\text{hypothesis}}.\text{goBackward}(v.\text{target})$ 
24:       $v_{\text{localstart}} \leftarrow G^{\text{hypothesis}}.\text{findCorrespondingNode}(v_{\text{localtarget}})$ 
25:       $G.\text{connectWithEdge}(v_{\text{localstart}}, v_{\text{localtarget}})$ 
26:    end if
27:  end while
28: end for

```

---

A flowchart of the halgorithm is presented in Figure 4.13. Compared to the pseudocode presented above, the flowchart provides more detail, especially in the elaborate description accompanying the flowchart in Table 4.5. 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.

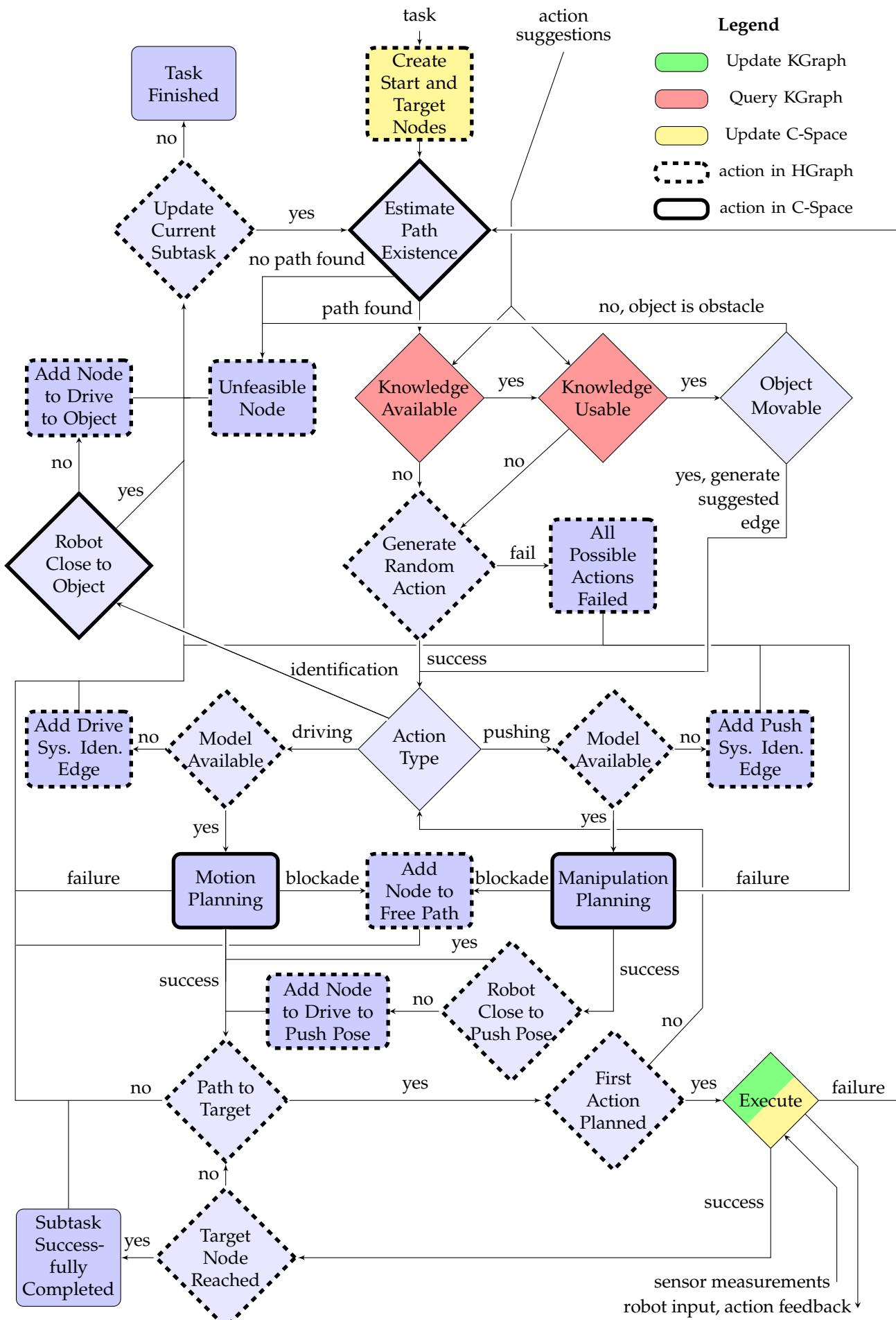


Figure 4.13: Flowchart displaying the hypothesis graph's workflow.

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 <i>current_start_node</i> and <i>current_target_node</i> are updated. When all subtasks have been addressed, conclude task is finished.
Estimate Path Existence	Check if a path exists between <i>current_start_node</i> and <i>current_target_node</i> whilst assuming that the object is holonomic.
Add Node to Drive to Object	Add a node before the <i>current_target_node</i> .
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 <i>current_target_node</i> to <i>current_target_node</i>
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 <i>current_target_node</i> , update <i>current_target_node</i> 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 <i>current_edge</i> , 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 <i>current_edge</i> , 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 position.
Path to Target	Is there a path from robot to target node in the hgraph, then set the first edge to <i>current_edge</i> otherwise update subtask.
First Action Planned	Check if motion/manipulation planning was performed.
Execute	Execute the <i>current_edge</i> , 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.

**Table 4.5:** Elaborate information on actions taken by blocks in ??.

**Encountering a Blocked Path** During propagation of an action edge’s status, motion or manipulation planning occurs. If an object blocks the path, planning will detect it, and the halgorithm tries to free it. In the following example, the halgorithm detects a blocking object and frees the path by pushing the blocking object to a new configuration, and can be visualized in Figure 4.14.

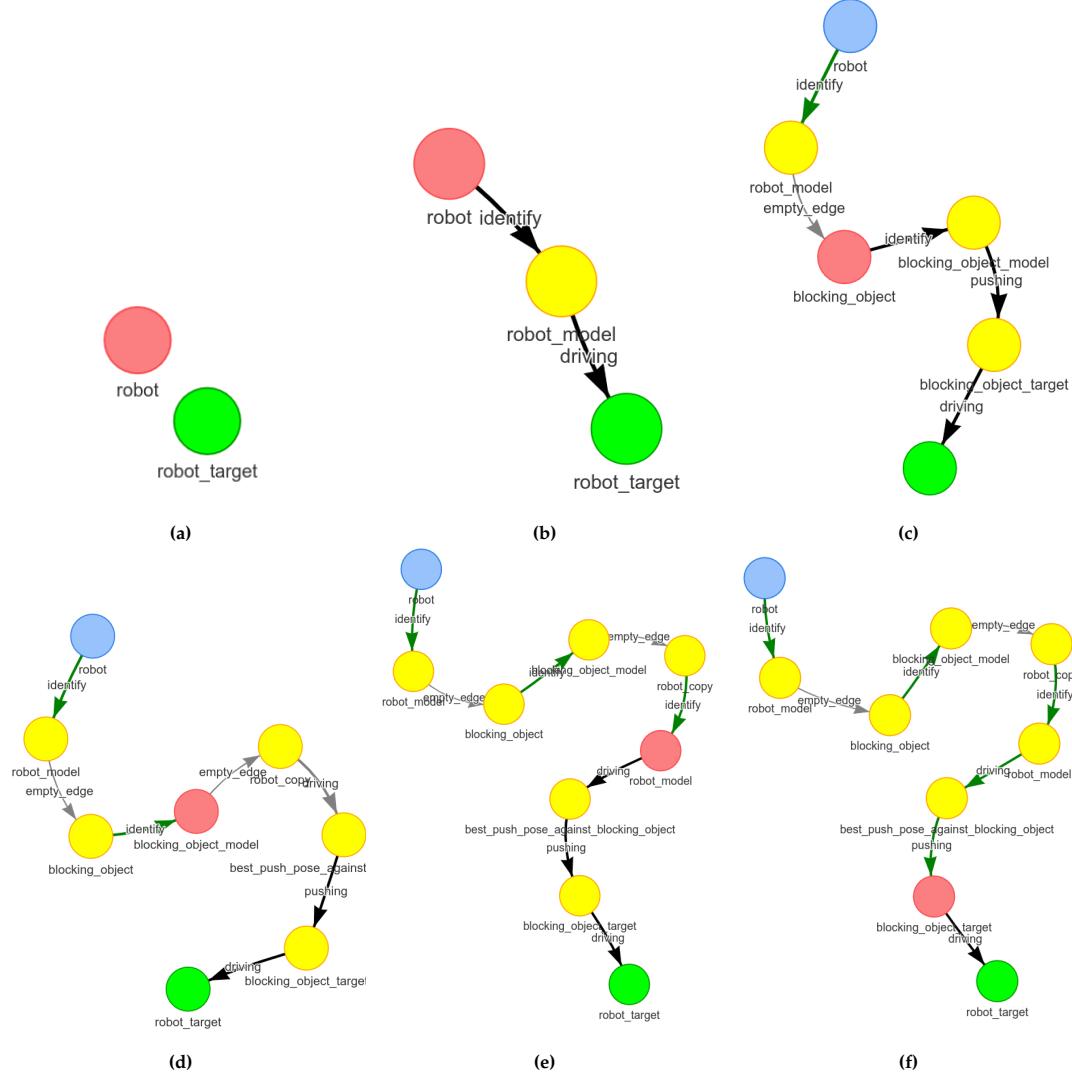


Figure 4.14: hgraph for driving to target configuration and encountering a blocked path

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 parameterizations (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 positions even if reaching such a position requires objects to be moved first. The proposed algorithm learns to classify objects by updating the object’s

class from unknown to movable or obstacle. The halgorithm can push objects to target positions by identifying a system model and then pushing the object toward its target position. However, the system model that system identification yields is of short use because it is only given to the corresponding action edge. The executed edges will be reviewed and stored in a knowledge base named the knowledge graph (kgraph) in the next section. The ackgraph is filled with action reviews such that it can suggest a parameterization based on the stored action reviews.

was there in here somehow an indication that classifies objects? unknown to movable unmovable

## 4.4. Knowledge Graph

The hgraph, discussed in the previous section, spans a lifetime over a single task. Learned system models created for future tasks are not stored for hgraph. Storing learned environment knowledge is the kgraph's responsibility. Another responsibility of the kgraph is to make an ordering in the stored environment knowledge. The ordering is made with a proposed success factor. This metric combines multiple metrics such as prediction error, tracking error and the success-fail ratio of an edge parameterization (controller and system model).

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 does not adhere to any standard that may apply to standardized knowledge bases.

### 4.4.1. Definition

Before defining the kgraph, the definition of the success factor, edges, center- and side nodes are defined where the kgraph depends upon.

Formally the **success factor** =  $\alpha$ :

$$\alpha = \begin{cases} 0.1^{\epsilon_{avg}} & \text{if edge does not yet exist in kgraph} \\ \alpha + 0.1 * (1 - \alpha) & \text{if success_factor already exist in kgraph} \\ & \text{and edge was successfully completed} \\ \alpha - 0.1 * \alpha & \text{if success_factor already exist in kgraph} \\ & \text{and edge failed} \end{cases}$$

**Success Factor** The responsibility of the kgraph is to store object class information and to collect edge feedback, to then suggest edge parameterization based on the collected feedback. Estimating which parameterization would be the best candidate is an entire field of research. In this thesis, a simple metric, the success factor, has been chosen based on the average prediction error and the number of times an edge succeeded or failed. From the point of the kgraph there is little information to work with; feedback must be created with only information on prediction error, tracking error and whether a fault was detected. Then action suggestions must be made based on collected feedback and an object that should change the start configuration to a target configuration (connecting two nodes in the hgraph). A simple success factor thus already incorporates most of the available metrics.

An edge describes its parameterization and how that parameterization compares to other edges in the kgraph.

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

Where (System Model, Controller) together is referred to as edge parameterization.

An center node's task is to represent an object and store its class which can be Movable or Unmovable.

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

Where  $id$  an identifier for the center node,  $obj_{id}$  an identifier linked to an object,  $\text{OBJ\_CLASS}$  the classification of that object.

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

Formally, a **side node**,  $v_{id}^{side} = \langle id \rangle$

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  $V_K = \{v^{center}, v^{side}\}$ ,  $E_K \in \{e_{(i,j)} | i \in E_{Kids}^{center}, j \in E_{Kids}^{side}\}$ .

Where  $E_{Kids}^{center}$  and  $E_{Kids}^{side}$  are the identifiers of the set of center edges and side edges respectively.

The kgraph has three essential functions. The *add\_object* function adds object information to the kgraph, which is important for adding unmovable obstacles that the robot cannot manipulate. The *add\_review* function is used when an edge is successfully or unsuccessfully completed, and the corresponding node in 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.

#### 4.4.2. Example

An example kgraph can be visualized in Figure 4.15, where parameterization of edges is displayed, and the object on which that kgraph holds information is displayed as an image. For clarification, the connected left part with the image of the point robot on the center node has three outgoing edges that describe robot driving. The connected part on the right with an image of the point robot and the green box on the center node has two outgoing edges that describe the robot pushing against the green box.

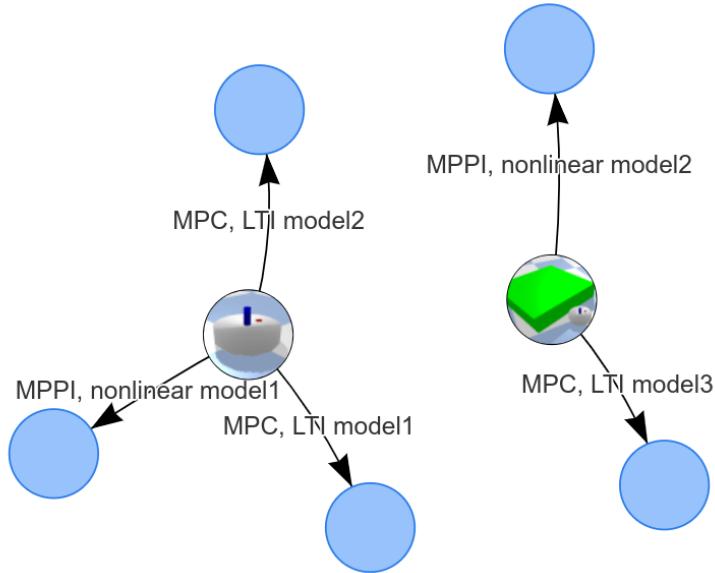


Figure 4.15: kgraph with three edges for robot driving and two edges for pushing the green box.

The edges in the figure above display only the edge parameterization but store more information, mainly the success factor. The blue nodes serve a small purpose, ensuring edges can point to a node. The blue nodes could fulfil a larger purpose, describing which actuators the edge can control. For

example, a mobile robot with a robot arm attached can have a set of controllers that only drive the base, a set of controllers that only steer the robot arm and a set that controls both the base and robot arm. In such cases, the blue nodes describe which part of the robot can be actuated. The controllers considered in this thesis control every robot actuator, resulting in the blue nodes serving such a small purpose.

#### 4.4.3. Edge Metrics

Corrado: rephrase good and bad to sound like a professional cunt

The kgraph keeps an ordered list of ‘good’ and ‘bad’ edge arguments (controller and system model). ‘Good’ and ‘bad’ are defined by edge metrics; these metrics are created after the completion of an edge, regardless of whether the edge was completed or failed. An indication is given on why specific metrics matter in Table 4.6.

Prediction Error (PE)	To better compare prediction errors the PE is summarized and average PE. The average PE indicates an accurate system model but can give misleading results since PE is also an indicator of unexpected collisions. Prediction error should thus only be used if there are no collisions detected. The average PE has more flaws since outliers mostly determine the average. For the largest part, some unfortunate outliers in the PE might determine the average PE. The average PE will thus not be used because it is not robust enough.
ratio num_successfully completed edges and num_total edges	Over time, the kgraph can recommend the same edge arguments multiple times. Logging the ratio of succeeding edges vs total edges builds an evident portfolio. Still, this metric has to be taken with a grain of salt because edges with equal edge arguments perform similar actions e.g. pushing an object through a wide corridor is compared to pushing the same object through a narrow corridor. One could say “comparing apples with pears”
	Corrado: But then you would have task-specific metrics right? Meaning a knowledge graph for when you do pushes in open space and one for small corridors? I don't know if this would scale. To make sense of the metric this metric should be task specific though
the final position and displacement error	The quality of the result is measured in the final position and displacement error. The importance should thus be stressed when ordering edge arguments.
planning time	With system identification, path estimation, motion or manipulation planning, the planning time can vary in orders of magnitude between simple or more complex approaches.
run time	Also known as execution time, would be a quality indicator if start- and target states were equal. Edges are recommended to solve similar tasks where the path length between the start and target state differs. Thus planning time is not of any use to rank edges.
completion time = run time + planning time	With the same argumentation as run time, completion time is not used to rank edges.

Table 4.6: Edge metrics used to rank control methods from ‘good’ to ‘bad’

Corrado: So why do we have all these metrics if you do not consider them? Shouldn't you normalize the metrics such that they are at least comparable with each other in similar yet different tasks?

The kgraph fulfils two goals. It stores information on whether an object can be manipulated and stores edge parameterizations from the highest success factor to the lowest success factor per object. Information if objects can be manipulated to prevent the halgorithm from trying to push unmovable objects.

# 5

## Results

*This chapter presents various robot environments and tasks that challenge the proposed framework. The test results provide evidence that supports the claims in the conclusion Chapter 6. This chapter starts by introducing **method metrics** in Section 5.1 that measure task performance. Then the proposed framework solves driving and pushing tasks in randomized environments in Section 5.2. The chapter finishes with a comparison with the state-of-the-art in Section 5.3.*

**The Simulation Environment** Testing in a simulation environment has been done using the URDF Gym Environment [35], a 100% python environment build upon the PyBullet library [7]. 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.

### 5.1. Proposed Method Metrics

The proposed frameworks task performance is measured in metrics named the method metrics, these include the PE, search-, execute- and total time to complete a task. Note that two other metrics have been defined; the monitoring metrics in Section 4.3, and the edge metrics in Section 4.4.3. The results are interesting, but most interesting is the progression of the method 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 with corresponding argumentation on the relevance of the metric.

Prediction Error [meter]	The PE is calculated every time step, then the PE is augmented into a single list. That list is then later analysed by calculating for example the mean and standard deviation. The PE is high when unexpected behaviour occurs, unexpected behaviour can be expected when entering a new environment. When the PE lowers that would indicate the robot encounters less unexpected behaviour, indicating the robot is learning.
search time [sec]	The time the proposed framework spends searching for hypotheses. The search time is very dependent on the environment and the number of subtask in the task. The search time can be used for comparison if both tasks are equal.
execution time [sec]	The time the proposed framework spends executing hypotheses. Just as the search time, conclusions can be drawn when comparing the execution time between similar tasks.
task time	search time + execution time.

**Table 5.1:** Method metrics used to measure task performance by the proposed robot framework, and motivation why the metric is relevant.

The simulation environment provides many different robots, of which one robot is selected to perform tests, the point robot, which is 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 drive controller, and a MPPI push controller that were introduced in Section 2.2. Three system models are used, a 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.

<i>lti-drive-model</i>	A second order LTI model that can be used by both the MPC and the MPPI drive controller. The next robot configuration is based on the current configuration and robot system input in $x$ and $y$ direction.
<i>nonlinear-push-model-1</i>	A nonlinear model describing the next object configuration in $x$ and $y$ direction and the next robot configuration in $x$ and $y$ direction. The next object and robot configuration are based on the current configurations and robot system input in $x$ and $y$ direction.
<i>nonlinear-push-model-2</i>	A nonlinear model describing the next object configuration in $x$ , $y$ direction and the orientation $\theta$ , and the robot configuration in $x$ and $y$ direction based on the current configurations of the robot, the object and on the robot inputs in $x$ and $y$ direction.

**Table 5.2:** Available drive and push system models.

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

$$x_{lti\text{-}drive\text{-}model}(k+1) = \begin{bmatrix} x_{robot}(k+1) \\ y_{robot}(k+1) \end{bmatrix} = \begin{bmatrix} x_{robot}(k) + DTu_x(k) \\ y_{robot}(k) + DTu_y(k) \end{bmatrix} \quad (5.1)$$

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

$$x_{nonlinear\text{-}push\text{-}model\text{-}1}(k+1) = \begin{bmatrix} x_{robot}(k+1) \\ y_{robot}(k+1) \\ x_{obj}(k+1) \\ y_{obj}(k+1) \end{bmatrix} = \begin{bmatrix} x_{robot}(k+1) + DTu_x(k) \\ y_{robot}(k+1) + DTu_y(k) \\ x_{obj}(k+1) + \frac{1}{2}DTu_x(k) \\ y_{obj}(k+1) + \frac{1}{2}DTu_y(k) \end{bmatrix} \quad (5.2)$$

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

$$x_{nonlinear\text{-}push\text{-}model\text{-}2}(k+1) = \begin{bmatrix} x_{robot}(k+1) \\ y_{robot}(k+1) \\ x_{obj}(k+1) \\ y_{obj}(k+1) \\ \theta_{obj}(k+1) \end{bmatrix} = \begin{bmatrix} x_{robot}(k) + DTu_x(k) \\ y_{robot}(k) + DTu_y(k) \\ x_{obj}(k) + DT \sin(\theta_{obj}(k))(1 - |\frac{2st}{H}|)vp \\ y_{obj}(k) + DT \cos(\theta_{obj}(k))(1 - |\frac{2st}{H}|)vp \\ \theta_{obj}(k) + \frac{2*DT*vp*st}{H} \end{bmatrix} \quad (5.3)$$

Where  $st$  indicates the distance from the contact point between the robot and pushed object, perpendicular to the line that coincides with the object and the robots center of mass. A positive  $st$  indicates the object will rotate anticlockwise, a negative  $st$  indicates a clockwise rotation. The width of an object is defined as  $H$  and is set to 2 meter,  $vp$  is the velocity of the robot perpendicular to the object defined as

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

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 combination of controller and system model in the available set of controllers and system models.

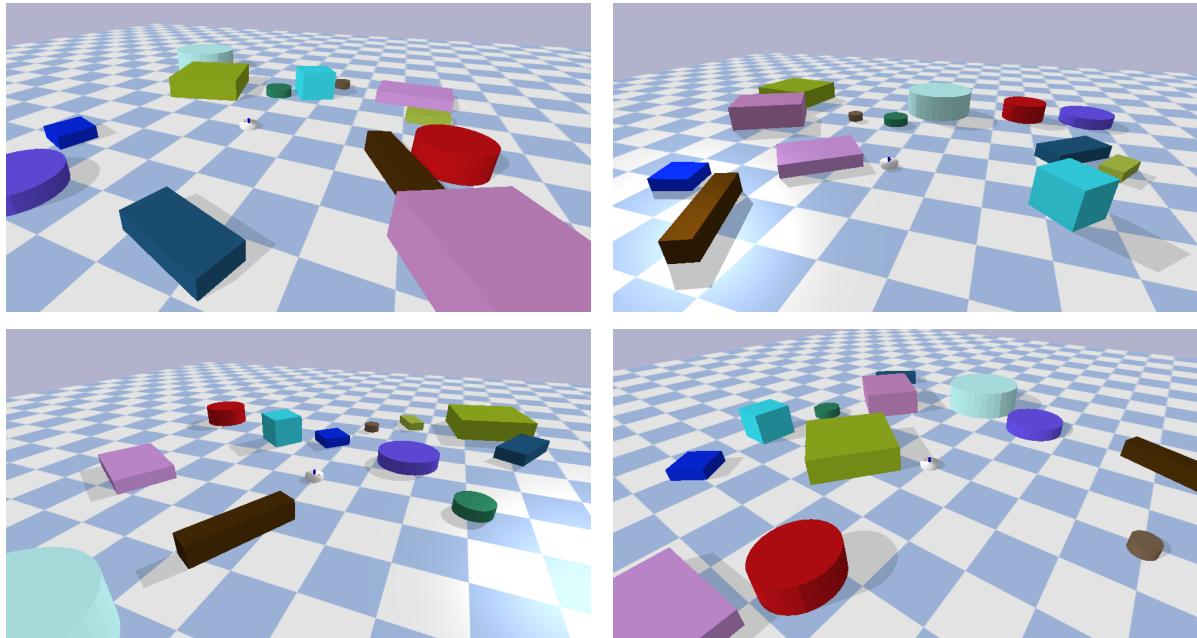
## 5.2. Randomization

In the randomized environment the task and the environment are initialised by randomization, after tasks completion, the environment is reshuffled. A reshuffled gives the objects in environment and task a new initial pose and resets the robot position. By solving a task, reshuffling the environment and solving another reshuffled task, the robot gains experience that is stored in the kgraph. A set of multiple tasks where initially the kgraph is empty, is named a *run*, and at the end of a run the kgraph is filled with the gained experience. Table 5.3 presents a set of parameters that initializes the random environment. Two type of tasks solved by the robot, an driving task, where the robot must drive toward multiple random target poses, and a pushing task, where the robot must push an movable object toward a random target pose. 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 parameterisations, and once without suggestions, by randomly selecting an available edge parameterization.

The <i>size of the grid</i>	length and width of the ground plane in $x$ and $y$ direction.
The <i>minimal and maximal size of objects</i>	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 <i>maximal weight</i>	which is uniformly distributed for the environment objects, minimal weight is set by default to 1 gram.
The <i>number of unmovable objects</i>	Specify the amount of unmovable objects.
The <i>number of movable objects</i>	Specify the amount of movable objects.
The <i>number of subtasks in a task</i>	Specify the amount of subtasks in a task.

**Table 5.3:** The tuning parameters that must be specified to create a random environment

The ratio between the number of cylinders and boxes is determined by randomization. For every new object generated there is a 50% chance it becomes a box and 50% chance it becomes a cylinder. The environment can be *reshuffled*, which can be visualised in Figure 5.1. Reshuffling the environment changes 3 aspects of the environment, first, the robots position is reset to the origin Origin. Second, every object is set to a new initial position whilst their properties remain unchanged. Thirdly, objects in the task receive new target poses. The reshuffle functionality can be visualised in Figure 5.1. By solving a similar task in an reshuffled random environment multiple times, the robot gains experience and task execution can be investigated. With the investigation trends in task execution is monitored, to see improvement whilst gaining experience.



**Figure 5.1:** A random environment initialised by tuning parameters presented in Table 5.4. After initialisation the environment is reshuffled three times.

### 5.2.1. A Driving Task

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

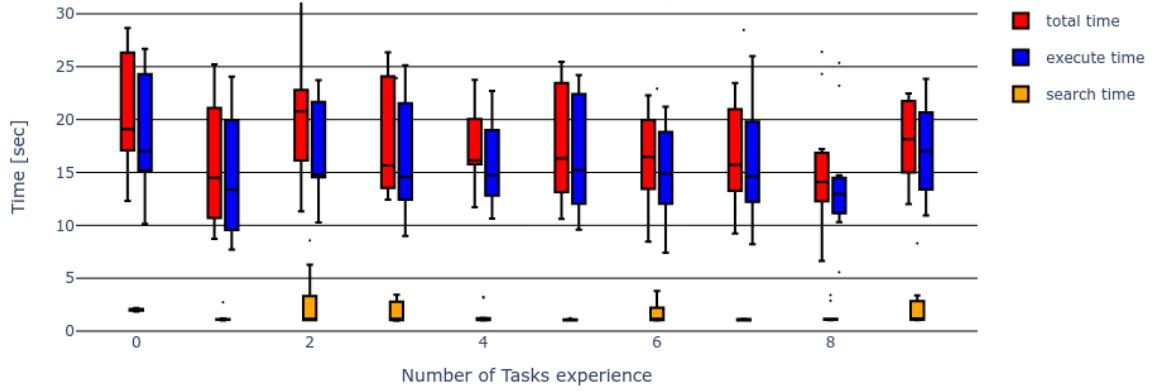
grid size	$x=12 \text{ m}, y=12 \text{ m}$
object size	$\min\_length = 0.2\text{m}, \max\_length = 2\text{m}$
object weight	$\max\_weight = 1000\text{g} = 1\text{kg}$
number of objects	$\text{num\_unmovable\_obj} = 3, \text{num\_movable\_obj} = 5$
number of tested runs	$\text{num\_runs} = 10$
number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

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

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 unmoving obstacles that reward the robot learning such objects are unmoving (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 randomised environments. The number of subtasks is set to 3, a low number of drive subtasks that can be completed in under 2 minutes.

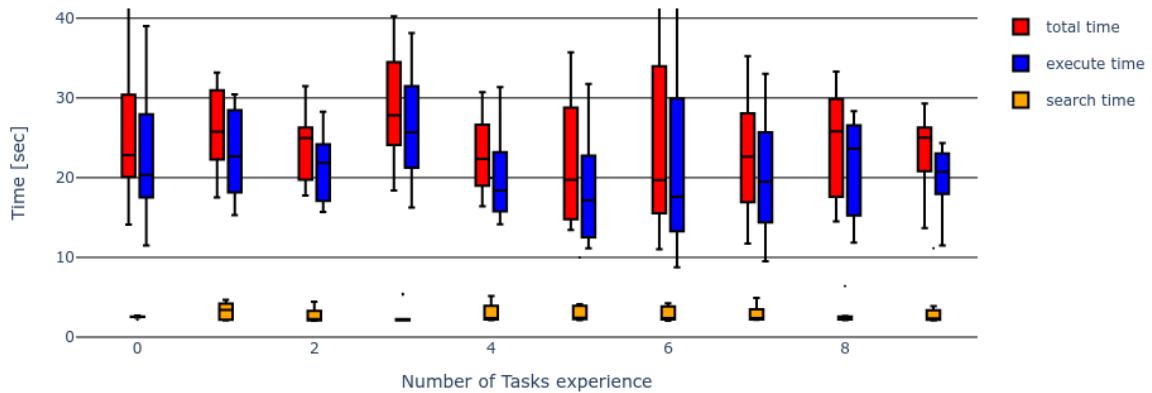
Lastly, a number of tuning parameters must be set for the halgorithm. These are the maximal robot speed, set to  $1 \text{ m/s}$ , the *cell size* for the path estimator set to 0.1 meter, the action planner takes four tuning parameters; the *step size* set to 0.2 meter, the *search size* set to 0.35 meter, an *known obstacle space cost* set to 2 meter and an *unknown obstacle space cost* set to 3.5 meter.

All parameters are set, results can be analysed, the following two figures show the execution-, search- and total times over the ten runs. First, a boxplot displaying task execution whilst using kgraph action suggestions in Figure 5.2. Second, a boxplot that displays task execution without using kgraph action suggestions in Figure 5.3.



**Figure 5.2:** 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. A run contains ten tasks, starting the run with an empty kgraph that collects action feedback as the robot gains experience. The task contains three subtasks, in other words, the robot must drive to three target poses in order to complete a task. The vertical axis displays a boxplot of the search-, execution- and total time over ten runs, where the sum of seach- and execution time equals total time.

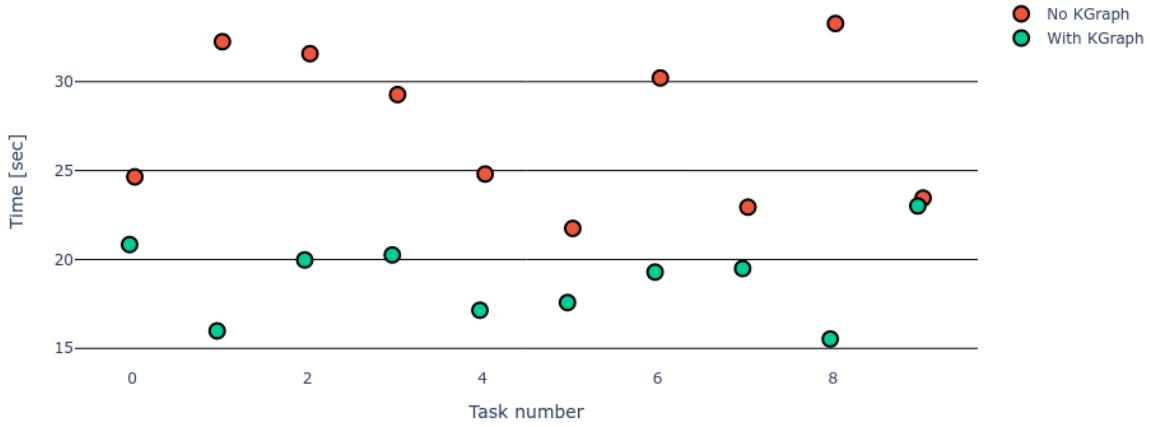
The above figure displays results where the halgorithm sends action feedback and received action suggestions from the kgraph, the figure below displays the results for solving the same tasks in the same random environment without using kgraph suggestions. Instead a random parameterization is selected for every action edge. Ensuring that the random environments are repeatable is accomplished by fixing the seed. The fixed seed ensures that the randomly generated environments can be created multiple times, once to solve with kgraph suggestions and once to solve without help from the kgraph.



**Figure 5.3:** Search-, execution- and total time to complete a drive task **without** using 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 task contains three subtasks, in other words, the robot must drive to three target poses in order to complete a task. The vertical axis displays a boxplot of the search-, execution- and total time over ten runs, where the sum of seach- and execution time equals total time.

The results in both Figures 5.2 and 5.3 show no clear trend. However overall Figure 5.2 shows significant improvement over Figure 5.3 which becomes better visible if only the means are compared in

Figure 5.4.



**Figure 5.4:** 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 is leveraged indicated by “with kgraph” and once with random selection indicated by “no kgraph”.

Two driving edge parameterisations are available, the MPC and the MPPI parameterisation that both use lti-drive-model. The kgraph prefers the MPC parameterization over de 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 5.2, it was already converged at the first task.

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

**Table 5.5:** Selecting the (MPC, *lti-drive-model*) parameterization versus selecting the (MPPI, *lti-drive-model* parameterization for drive actions.

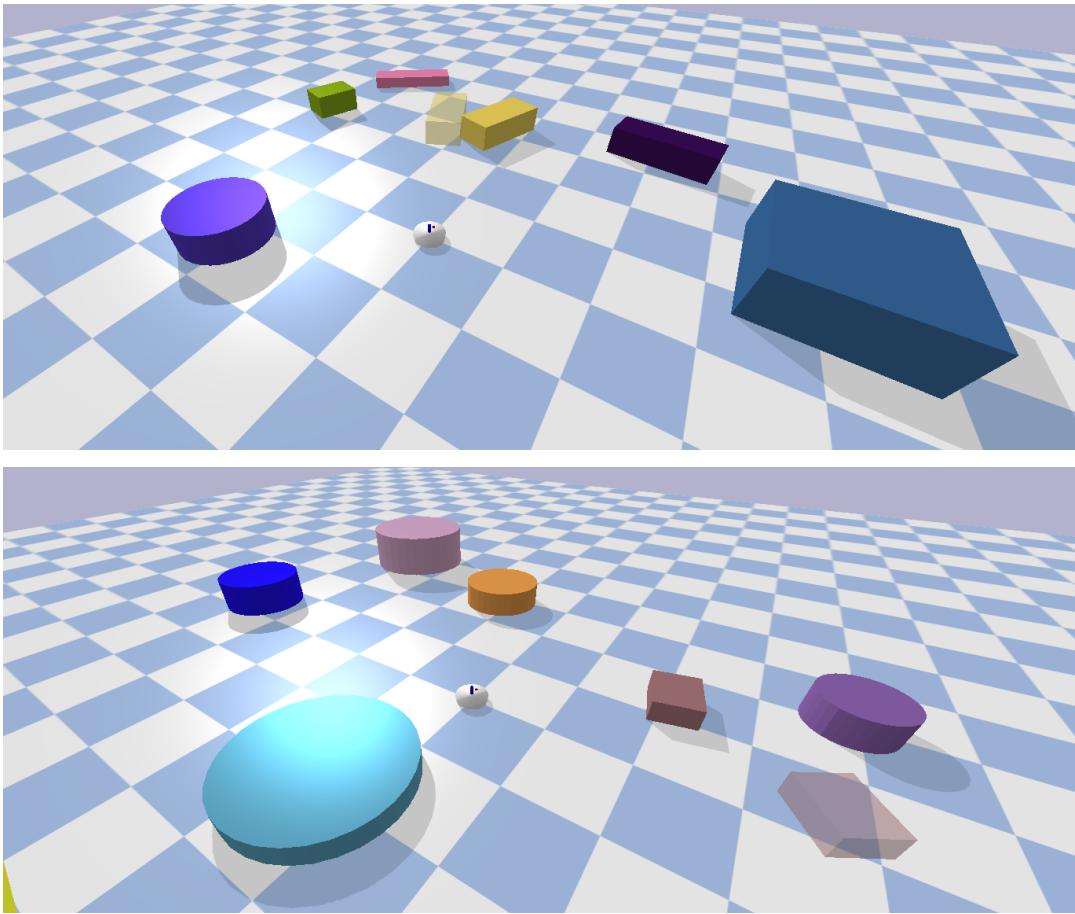
The halgorithm successfully completes 300 subtasks (3 subtasks per task, 10 tasks per run, 10 runs are completed) twice. Once while stroing edge feedback in the kgraph that suggest edge parameterization, and once without kgraph. In Table 5.5 the number of MPC and MPPI parameterizations should be more 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.

### 5.2.2. 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 an object toward its specified target pose. The tuning parameters that make up the random environment for the push task can be visualised in Table 5.6.

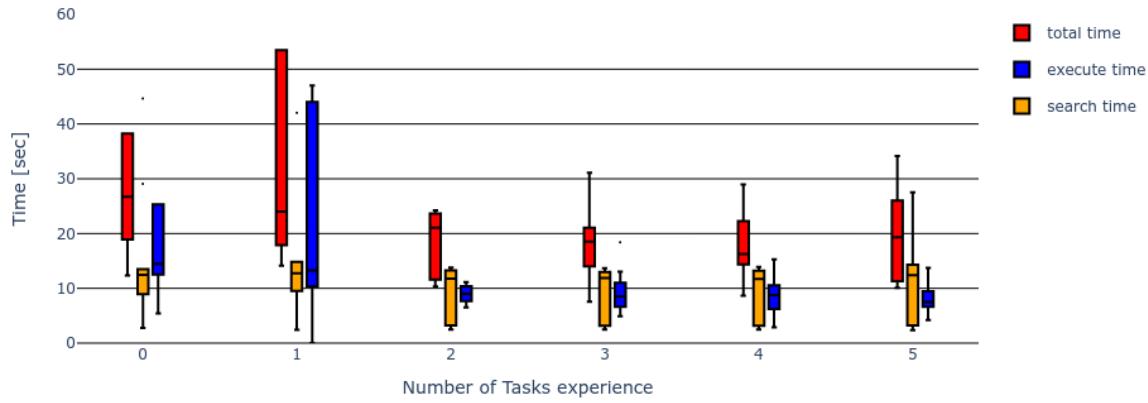
grid size	$x=12 \text{ m}, \quad y=12 \text{ m}$
object size	$\min\_length = 0.2m, \quad \max\_length = 2m$
object weight	$\max\_weight = 1000g = 1kg$
number of objects	$\text{num\_unmovable\_obj} = 3, \quad \text{num\_movable\_obj} = 5$
number of tested runs	$\text{num\_runs} = 10$
number of tasks in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

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

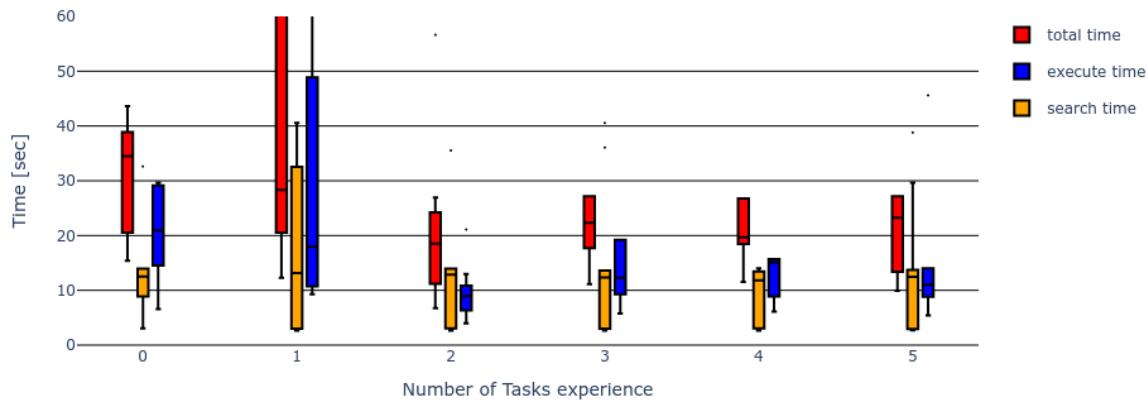


**Figure 5.5:** Two random environments where the task contains a subtask displayed by a target ghost pose.

All tuning parameters are set up, now the results are presented, first the pushing task is completed using kgraph suggestions. The search-, execute- and totaltime for task completion is presented in Figure 5.6. Then the same tasks are completed without help of the kgraph suggestions in Figure 5.3.

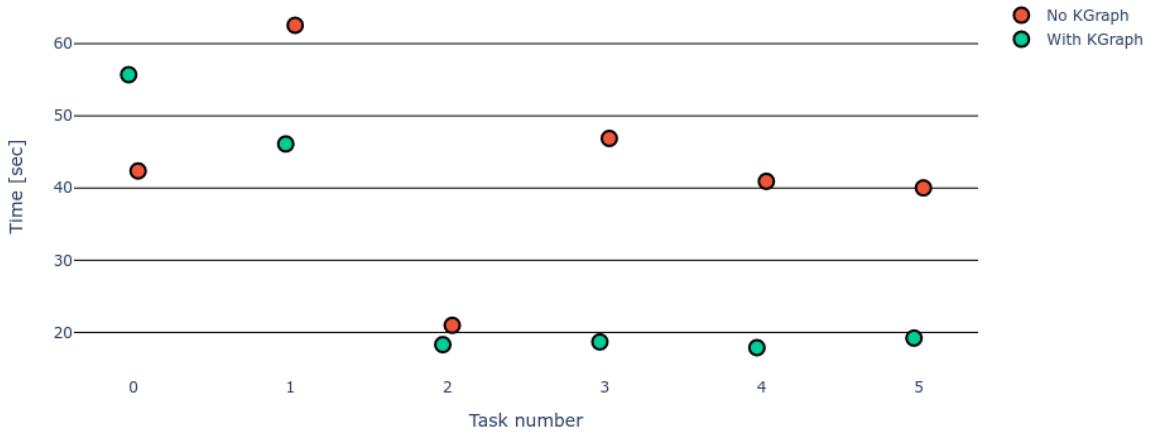


**Figure 5.6:** Search-, execution- and total time to complete a pushing task **with** kgraph action suggestions. The horizontal axis indicates the number of task experience in a run. A run contains ten tasks, starting the run with an empty kgraph that collects action feedback as the robot gains experience. The vertical axis displays a boxplot of the search-, execution- and total time over ten runs, where the sum of seach- and execution time equals total time.



**Figure 5.7:** 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 seach- and execution time equals total time.

Both Figure 5.6 and Figure 5.7 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 5.8.

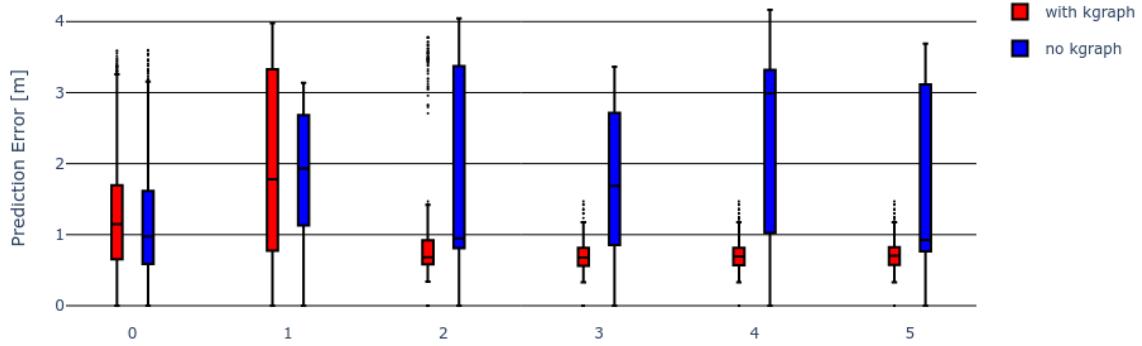


**Figure 5.8:** Comparing average total time to complete a task out of then runs for a pushing task. For the edge parameterizations once with the use of kgraph action suggestions is leveraged indicated by “with kgraph” and once with random selection indicated by “no kgraph”.

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

**Table 5.7:** Influence of kgraph suggestions in selecting the (*MPPI, nonlinear-push-model-1*) parameterization versus selecting the (*MPPI, nonlinear-push-model-2*) parameterization for push actions.

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 5.9.



**Figure 5.9:** Box plot of the Prediction Error, a comparison between kgraph action suggestions and randomly selection.

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

### 5.3. Comparison with State-of-the-Art

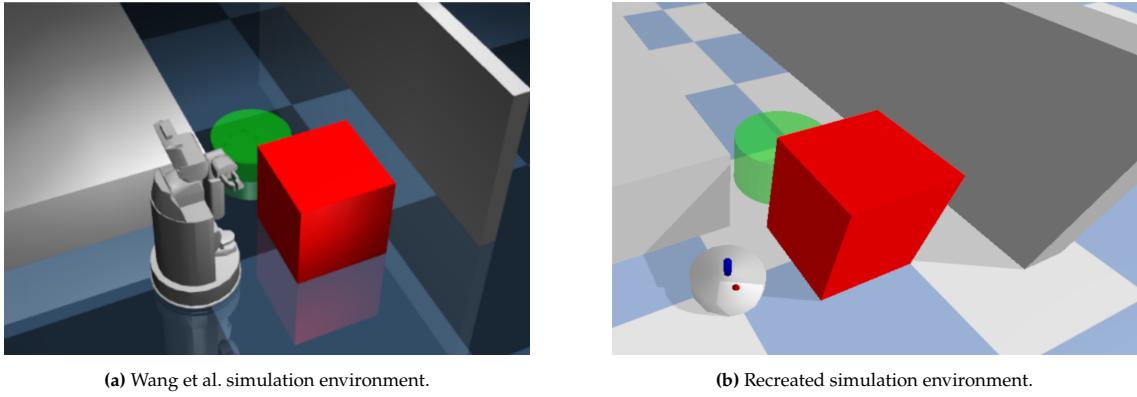
In the introduction Table 1.1 was presented. That table presents state-of-the-art methods and the subset of the three main topics that they include (the three topics; learning system models, the NAMO problem and nonprehesile pushing). Now that table presented again in Table 5.8, an additional column 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.

Author	Citation	Learns object dynamics	NAMO		Specify object target poses		method metric
			prehesile	nonprehesile	prehesile	nonprehesile	
Ellis et al.	[12]	✓	✗	✓	✗	✗	success rate
Sabbagh Novin et al.	[32]	✓	✓	✗	✓	✗	success rate, execution time prediction error, final position error
Scholz et al.	[33]	✓	✓	✗	✗	✗	runtime, planning time, number of replannings number of calls to update model
Vega-Brown and Roy	[40]	✗	✓	✗	✓	✗	computation time
Wang et al.	[42]	✓	✗	✓	✗	✗	computation and execution time
Groote	Propose Framework	✗/✓	✗	✓	✗	✓	

**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.

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 [42]. Wang et al. proposed framework takes an affordance approach compared with a contact-implicit motion planning algorithm [42]. A test is performed in a real-world environment, where the robot drives 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 the chair stands. The implicit representation of the real-world environment and the mimicked robot environments can be seen in Figure 5.10.



**Figure 5.10:** Two similar environments and tasks, the robots are tasked to drive toward a target position 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 affordance of an object and time to optimize the trajectory. The recreated or mimicked environment is displayed in Figure 5.10b and is solved ten times, every time starting without environmental knowledge and thus an empty kgraph. The avarage search, execute- and total time over these then executions is also displayed in Table 5.9.

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

**Table 5.9:** Average execute-, search and total times for a repeated task solved three times. The task involves learning object dynamcis and the NAMO problem and can be visualise in Figure 5.10.

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.

# 6

## Conclusions

This thesis aims to combine three topics in robotics; learning object dynamics, the NAMO problem and nonprehensile pushing, into a single robot framework. The overwhelming research in the individual topics stand in 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.

This 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?* How the learned objects' system models improve global task planning 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 almost instantly 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 thirth improvement. 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. The second improvement 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 bybassed 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 **X**/**✓**, 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 replannings 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 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.

## 6.1. Future work

### 6.1.1. System identification module

The thesis provides an module that collects IO data for the robot, and for pushing an object from multiple sides. It has been fully intergrated 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 categoreis; data-driven model, hybrid models and analytic models. To convert IO data to a dynamical system model for robot driving or the robot pushing is for system identification methods.

Possible extensions to the proposed framework and improvements on the proposed framework are grouped in this section.

proposed planner with local checks

system identificaiton instead of analytic models

### 6.1.2. 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 [20] 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.

# Glossary

## List of Acronyms

<b>RRT</b>	Rapidly-exploring Random Tree	
<b>RRT*</b>	optimised Rapidly-exploring Random Tree	13
<b>PRM</b>	Probabilistic Road Map	
<b>PRM*</b>	optimised Probabilistic Road Map	
<b>MPPI</b>	Model Predictive Path Integral	iii
<b>MPC</b>	Model Predictive Control	iii
<b>NAMO</b>	Navigation Among Movable Objects	i
<b>NP-hard</b>	non-deterministic polynomial-time hard	3
<b>NP</b>	non-deterministic polynomial-time	83
<b>FSM</b>	Finite State Machine	33
<b>halgorithm</b>	hypothesis algorithm	iv
<b>hgraph</b>	hypothesis graph	ii
<b>kgraph</b>	knowledge graph	iv
<b>IO</b>	Input-Output	iv
<b>PE</b>	Prediction Error	24
<b>TE</b>	Tracking Error	24
<b>LTI</b>	Linear Time-Invariant	13
<b>DOF</b>	Degrees Of Freedom	4
<b>MDP</b>	Markov Decision Process	5

# 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] 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).
- [4] 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).
- [5] 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).
- [6] 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).
- [7] Erwin Coumans and Yunfei Bai. *PyBullet, a Python Module for Physics Simulation for Games, Robotics and Machine Learning*. 2016–2021. URL: <http://pybullet.org>.
- [8] Misha Denil et al. *Learning to Perform Physics Experiments via Deep Reinforcement Learning*. Aug. 17, 2017. arXiv: 1611.01843 [physics, stat]. URL: <http://arxiv.org/abs/1611.01843> (visited on 05/19/2022). preprint.
- [9] 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).
- [10] *Disjunctive Programming*. New York, NY: Springer Berlin Heidelberg, 2018. ISBN: 978-3-030-00147-6.
- [11] 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).
- [12] 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](https://www.researchgate.net/publication/362092621_Navigation_Among_Movable_Obstacles_with_Object_Localization_using_Photorealistic_Simulation).
- [13] 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).

- [14] 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).
- [15] 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).
- [16] 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).
- [17] 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).
- [18] 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).
- [19] Marek Kopicki et al. "Learning Modular and Transferable Forward Models of the Motions of Push Manipulated Objects". In: *Autonomous Robots* 41.5 (June 2017), pp. 1061–1082. issn: 0929-5593, 1573-7527. doi: 10.1007/s10514-016-9571-3. url: <http://link.springer.com/10.1007/s10514-016-9571-3> (visited on 03/15/2022).
- [20] Athanasios Krontiris and Kostas Bekris. "Dealing with Difficult Instances of Object Rearrangement". In: July 2015. doi: 10.15607/RSS.2015.XI.045.
- [21] Steven Michael LaValle. *Planning Algorithms*. Cambridge ; New York: Cambridge University Press, 2006. 826 pp. isbn: 978-0-521-86205-9.
- [22] Tekin Meriçli, Manuela Veloso, and H. Levent Akin. "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).
- [23] Tiago P. Nascimento, Carlos E. T. Dórea, and Luiz Marcos G. Gonçalves. "Nonholonomic Mobile Robots' Trajectory Tracking Model Predictive Control: A Survey". In: *Robotica* 36.5 (May 2018), pp. 676–696. issn: 0263-5747, 1469-8668. doi: 10.1017/S0263574717000637. url: [https://www.cambridge.org/core/product/identifier/S0263574717000637/type/journal\\_article](https://www.cambridge.org/core/product/identifier/S0263574717000637/type/journal_article) (visited on 04/30/2022).
- [24] neuromorphic tutorial. *LTC21 Tutorial MPPI*. June 7, 2021. url: [https://www.youtube.com/watch?v=19QLyMuQ\\_BE](https://www.youtube.com/watch?v=19QLyMuQ_BE) (visited on 05/31/2022).
- [25] 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).
- [26] Manoj Pokharel. "Computational Complexity Theory(P,NP,NP-Complete and NP-Hard Problems)". In: (June 2020).
- [27] Philip Polack et al. "The Kinematic Bicycle Model: A Consistent Model for Planning Feasible Trajectories for Autonomous Vehicles?" In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017 IEEE Intelligent Vehicles Symposium (IV). Los Angeles, CA, USA: IEEE, June 2017, pp. 812–818. isbn: 978-1-5090-4804-5. doi: 10.1109/IVS.2017.7995816. url: <http://ieeexplore.ieee.org/document/7995816/> (visited on 05/03/2023).

- [28] 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.
- [29] 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).
- [30] Nicholas Roy. "Hierarchy, Abstractions and Geometry". May 6, 2021. URL: <https://www.youtube.com/watch?v=mP-uK1PFqfI> (visited on 05/03/2023).
- [31] 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).
- [32] 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).
- [33] 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).
- [34] 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).
- [35] Max Spahn. *Urdf-Environment*. Version 1.2.0. Aug. 8, 2022. URL: [https://github.com/maxspahn/gym\\_envs\\_urdf](https://github.com/maxspahn/gym_envs_urdf).
- [36] 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).
- [37] 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).
- [38] 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).
- [39] 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](http://link.springer.com/10.1007/978-3-642-00312-7_37) (visited on 06/24/2022).
- [40] 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](http://link.springer.com/10.1007/978-3-030-43089-4_34) (visited on 06/23/2022).

- [41] 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.
- [42] 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).
- [43] 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).
- [44] 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](http://link.springer.com/10.1007/978-3-540-68405-3_17) (visited on 06/16/2022).

# 7

## Appendix

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

## 1. System Identification Methods

what is belwo must be rewritteen, it's only the appendix to realx

Let's first clarify the use of system models. The goal of system models is to capture the true dynamics of the system it describes. For instance, consider the following examples: when a robot arm reaches for some product to pick, a system model can estimate how the robot arm reacts before actually sending the input. Or consider a sphere which has received a push, the behaviour after the push is different from a cube which received an equal push, thus their models must be different also. A system model combined with the current state of the system and potential system input can estimate the state a system will be in as a result of the system input. An example of system input is wheel velocity  $\omega_l$  and  $\omega_r$  in ???. State information and the effect of system input are crucial information for robot control design, and for controllers to operate effectively, robot controllers will be discussed in the next section.

Models investigated in this literature are split into 2 other categories, single-body models and multi-body models. Such a distinction is made because both models have quite different dynamical properties, let's first define both. A **single-body model** is a system model which estimates the true dynamics of an object, which is assumed to be connected for all times. An example is a robot arm existing of multiple parts which are connected by joints (assuming that a part of the robot arm does not break off). The robot arm is a single-body and any model estimating only the true dynamics of only the robot arm belongs to the set of single-body models. **Multi-body models** are system models which estimate the true dynamics which involve multiple objects. These multiple objects can be connected or disconnected. Multi-body models include any combination of single-/multi- body models with other single-/multi-body models. For example, a robot arm grasping a box with a gripper. For the period the box is grasped by the robot gripper, the single-body model estimating the true dynamics of the robot arm can be augmented such that it estimates the pose of the box. Such an augmented model belongs to the set of multi-body models.

The true dynamics of a single-body contain some nonlinear parts such as slip and friction, however in general nonlinear dynamics are not dominating such that a single-body system can be estimated with LTI models. Dynamics which allow simplification to a linear model dominate compared to the nonlinear dynamics of the true dynamics, small accumulating errors as a result of for example slip can be accounted for. Stable control using LTI models is possible because the true dynamcis can be estimated by an LTI system model. Opposed to single-body system, multi-body systems are dominated by nonlinear dynamics, so much that a simplification to a linear system erases most true dynamics. Such a compromise for multi-body models is undesired since it would lead to a model mismatch between the model and the true dynamics, motivating the separation of single- and multi-body models.

?? is not approximating the true dynamics accurately, because pushing a cube using this differential equation does not capture any speed in the direction of the  $t$ -axis at point  $p$ . There is no friction of the object with the ground, let alone a distinction between static and dynamic friction which both affect the dynamics considerably. Simplifying the true dynamics results in model mismatch, too much simplification would eventually lead to a nonsense model, incapable of modelling the system behaviour accurately and leading to stability issues when used by a controller. ?? simplifies too much and is thus an ineffective modelling method. However, for the purpose of an example ?? is sufficient. To improve modelling the true dynamics of a robot pushing a box, more details of the true dynamics should be captured. [bauza\_data-efficient\_2018] created an analytical model for push manipulation involving Coulomb friction, force and friction constraints, resulting in a model modelled accurate enough to successfully track a reference signal. [bauza\_data-efficient\_2018] performed a close inspection of the object to push. Assuming prior knowledge about the object to encounter is unrealistic as opposed to robot dynamics. Now the two single-body models will be combined as a multi-body model, a schematic 2D diagram is displayed in ??.

Augmenting the differential ?? with ?? creates a multi-body model:

with  $s_t$  now dependent on both the location and geometric properties of the robot and the object, defined as:

$$s_t = \sqrt{(x^o(t) - x^r(t) - \frac{H+W}{2} \cos(\theta^o(t)))^2 + (y^o(t) - y^r(t) - \frac{H+W}{2} \sin(\theta^o(t)))^2}$$

The multi-body model in ?? displays how the first derivative of state variables can be calculated based on the input  $u^r(t) = [\omega_l \quad \omega_r]$ , the system constants  $H, r_l, r_r, W$  and the state variables. The multi-body model estimates the true dynamics of the robot and the box. The robot and cube object are touching at point  $p$ , when the objects become disjoint the multi-object model is not a valid representation of the true dynamics any more.

This concludes the example of multiple single-body models into a multi-body model. In the example, we saw one way of analytically modelling a robot and an cube object. There is however a vast literature of different methods which could been applied to model ?? [23], [**bauza\_data-efficient\_2018**], [**stuber\_feature-based\_2018**], [**stuber\_lets\_2020**]. An overview of modelling methods reviewed is conveniently condensed into ?? . Now the distinction between single-body models and multi-body models is clear, and the advantages and disadvantages per class of models are discussed.

### .1.1. 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], [**stuber\_feature-based\_2018**].

Nevertheless, the work in [**bauza\_data-efficient\_2018**] 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.

### .1.2. Data-driven models

Among recent studies, data-driven models shown an uptrend in popularity [**mericli\_push-manipulation\_2015**], [**bauza\_data-efficient\_2018**], [**stuber\_feature-based\_2018**], [**stuber\_lets\_2020**]. Fully data-driven methods don't model any structure of the system it describes, or use a generalised model which applies to all. 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, or the data is not rich enough then the model will not be accurate. For example, [**bauza\_data-efficient\_2018**] 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.

A field of research which has become very popular in recent years is artificial intelligence. In for example, estimating physical parameters from data. This is what [8] has shown with the use of deep reinforcement learning. By learning physical parameters artificial intelligence can help tackle the main issue with analytical approaches, estimating physical parameters. Instead of explicitly estimating physical parameters, another approach is learning a dynamics model directly [**stuber\_lets\_2020**]. [**cong\_self-adapting\_2020**] showed a powerful self-adapting push controller based on a *long short-term memory* modelling approach. A powerful advantage is online adaptation, which lowers a time-consuming train set to a warm start of approximately 5 test pushes with an object to push.

Alternatively, a set of robot-object test pushes can be used to fit a underlying distribution. Where

test pushes and their effect on the object's position and velocity are taken as a train set, then a distribution is fitted mapping the train set's input to its output. Such a fitted distribution is called a *contact model*. Recent literature shows a *Gaussian distribution* is a popular distribution used by [mericli\_push-manipulation\_2015] and [stuber\_feature-based\_2018]. [stuber\_feature-based\_2018] includes, additionally to a robot-object model (fitted from test pushes) an object-environment model, which considers object surface features. Object-environment information is added by modelling the surfaces of an object. All objects have surfaces, from a simple cube with 6 equal surfaces to more complex shapes. For the object-environment contact model, a central aspect is the probabilistic modelling of surface features, which describes every surface as a 3D position, 3D orientation and 2D local surface descriptor that encodes local curvatures. The data stored for surface features come from 3D point clouds created with a depth camera. An advantage is that the object environment model can be constructed based on 3D point cloud, and no interaction with the object is required. After having learned the contact model (object-environment model), a robot-object motion model is learned for which test pushes are required [stuber\_feature-based\_2018].

Contact models used for push manipulation make use of robot-object contact or additionally use object-environment contact. With enough rich data contact models outperform analytical and hybrid approaches. To tackle the amount of data required a more hybrid approach is developed. Which separates agent-object contact from object-environment contact. To generate a new model, two things are required, first an object-environment contact model of the object to model, and second, a sufficiently learned agent-object contact model. The latter does not necessarily have to be created from the object to model, existing agent-object contact models, combined with transfer learning can be sufficient [19].

The nonlinear effects which dominate multi-object system resides in IO data, because data-driven approaches models are not assuming any structure which could limit capturing nonlinear effects, the data-driven approaches are a worthy method for estimating true dynamics of in particular multi-body systems. 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.

### .1.3. 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) [stuber\_lets\_2020]. 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:

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

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 example parameterisation for analytical model example ??.

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 [34].

Single-bodies can and should be modelled as hybrid models, hybrid models allow a workable model which can be created from only the prior structural knowledge of the system. After some off or online system identification detailed parameters can be found, as effect the model will converge toward the true dynamics. While data-driven methods outperforms hybrid methods such methods take long

to properly train. Multi-body models are dominated by nonlinear dynamics, to fully capture such nonlinear dynamics, data-driven methods can and should model multi-body model, even if this means collecting a large train set.

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.

# A

## 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 [26]. 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 positions, the conclusion can be drawn that this NAMO problem is NP-hard.

# B

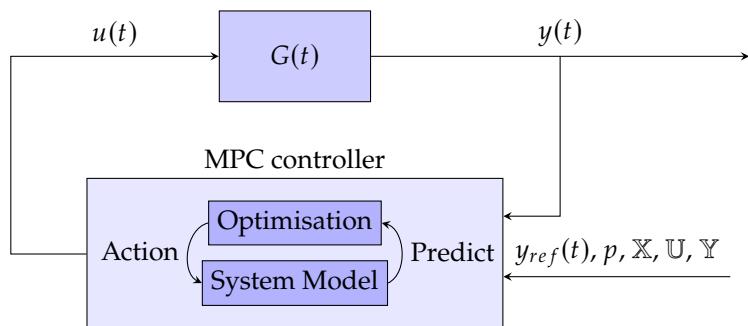
## Control Methods

### B.1. 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 [28]. A dynamical model can be presented in various forms, let's consider a familiar differential equation.

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

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 B.1, 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 [28]. 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 B.1.



**Figure B.1:** 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  $\mathbb{X}, \mathbb{U}, \mathbb{Y}$

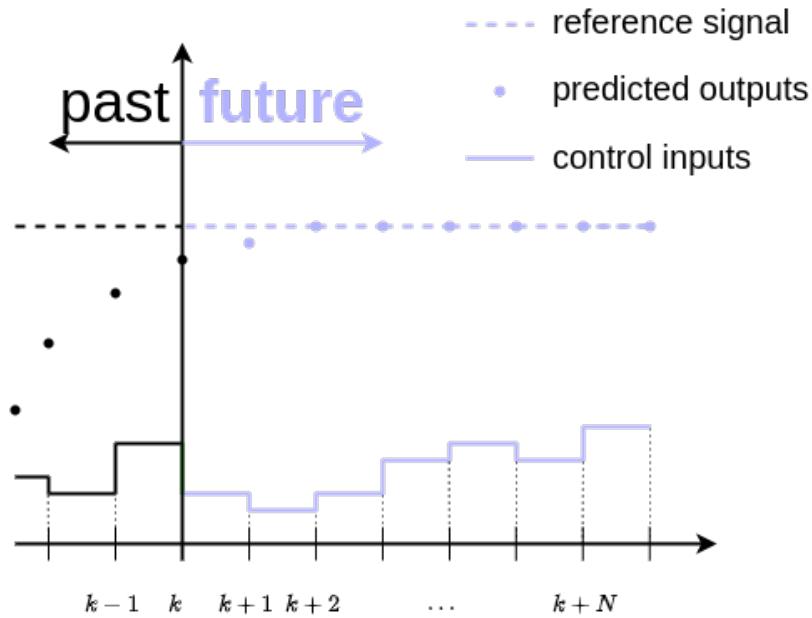
Some states of the system might be inside an obstacle region, such a region is undesirable for the robot to be in or go toward. The robot states are allowed in free space, which is all space minus the

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  $\mathbf{u}_N(k) = (u_k, u_{k+1}, \dots, u_{k+N})$ . 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 (B.1) gives the optimal input which steers the system toward the output reference signal while at the same time respecting the constraints.

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

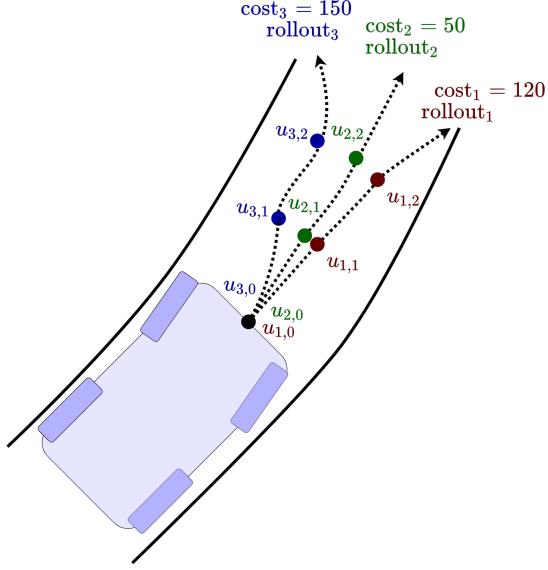
Figure B.2 displays the predicted output converging toward the constant output reference. After solving the minimisation problem, equation (B.1), the optimal input sequence is obtained  $\mathbf{u}_N^*$  (given that the constraints are respected for such input), from which only the first input is executed 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. Note that figures B.1 and B.2 is an example MPC controller, which hardly scratched the surface of MPC, there are many variations and additions such as deterministic and stochastic MPC, stage and terminal cost, distributed MPC, etc. which [28] visits extensively.



**Figure B.2:** A discrete MPC scheme tracking a constant reference signal.  $k$  indicates the discrete time step,  $N$  the control horizon

## B.2. MPPI Control

Introduced by [43] MPPI control arose. Which was followed by MPPI control combined with various system models, identification methods [1, 6, 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, [24]. 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 B.3.



**Figure B.3:** 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 [24].

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<sub>1</sub> and rollout<sub>3</sub> compared to rollout<sub>2</sub>. As a result, the input send to the system as a weighted sum of the rollouts is mostly determined by rollout<sub>2</sub>. The weighted sum determining the input is displayed in equation (B.2), from [24].

$$u(k+1) = u(k) + \frac{\sum_i w_i \delta u_i}{\sum_i w_i} \quad (\text{B.2})$$

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