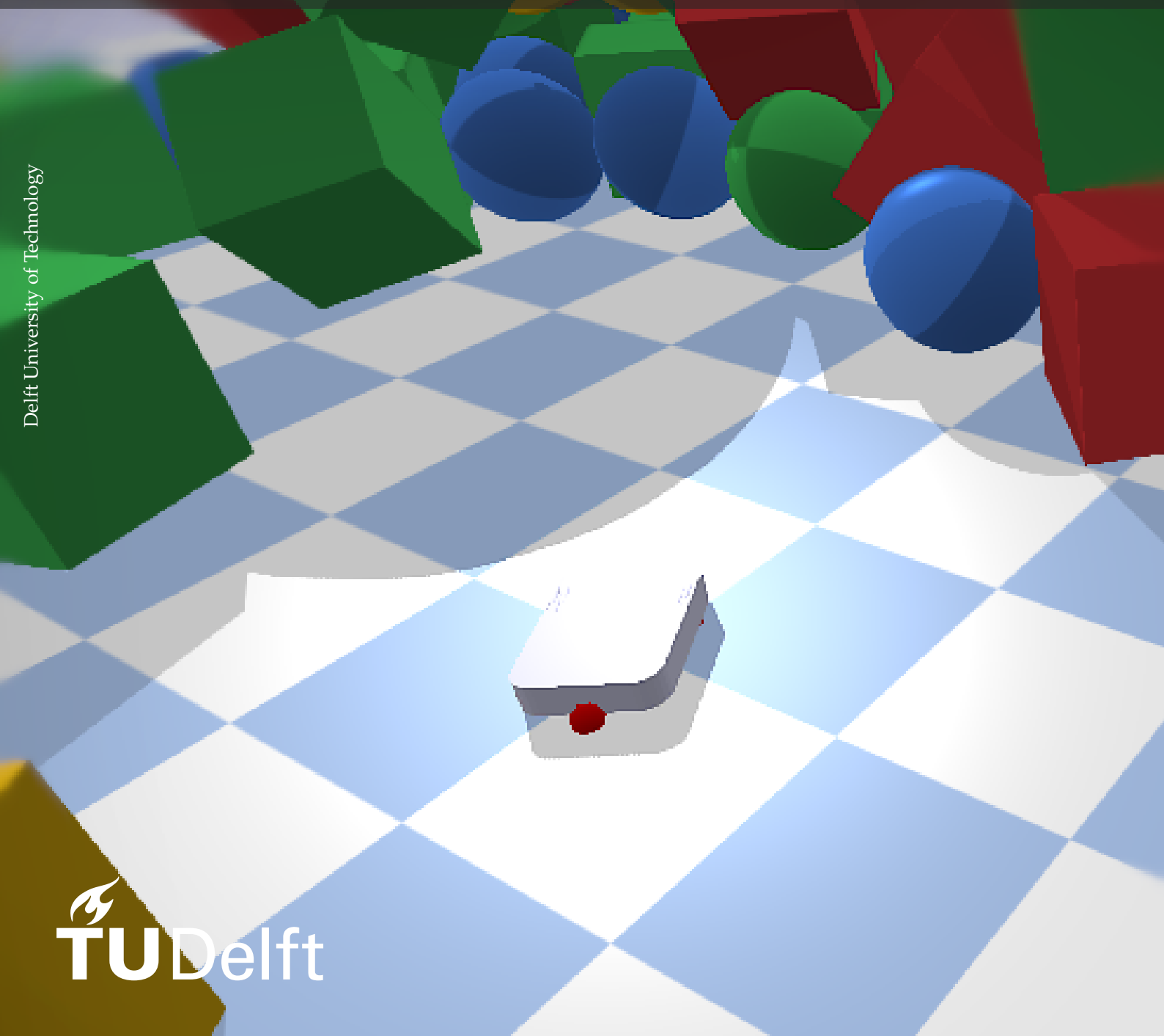
A graph-based search approach for planning and learning

An application to planar pushing and navigation tasks

SC52045: System & Control Thesis Report

G.S. Groote



Delft University of Technology

TU Delft

page intentionally left blank.

A graph-based search approach for planning and learning

An application to planar pushing and navigation tasks

by

G.S. Groote

Student Name	Student Number
Gijs S. Groote	4483987

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

Cover: Simulation environment used during the thesis [26].

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

TU Delft

Delft Center for Systems and Control

THE AI FOR RETAIL LAB DELFT

Cognitive Robotics

Abstract

In the field of robotics, much attention has been given to the research topics *learning object dynamics*, *Navigation Among Movable Objects (NAMO)* and *nonprehensile pushing*. However, combining these 3 research topics into one robot framework is insufficiently investigated. The main goal of this thesis is to combine these 3 research topics. A subgoal is to test the effect of learning object dynamics on task execution. A task consists of relocating a subset of the objects in a robot environment and a task can be split into individual subtasks. Finding an action sequence that completes a subtask requires a search in the joint configuration space, a space that emerges when the configuration space of the robot is augmented with the configuration spaces of objects in the environment.
Upon entering a new environment, objects are classified as "unknown" and their status can be updated to "movable" or "obstacle". A newly proposed planning algorithm incorporates a search in unknown and movable space in addition to the regular free and obstacle space. The proposed method consists of the **hypothesis algorithm**, the **hypothesis graph** and the **knowledge graph**. The hypothesis algorithm searches the joint configuration space and creates a hypothesis graph, together producing non-deterministic action sequences called a hypothesis. A hypothesis could complete a subtask, during a search for an hypothesis failures can emerge and during execution, faults can be detected. Faults and failures discard the current hypothesis and restart a search for a new hypothesis. A proposed knowledge graph stores feedback on actions in the form of a controller, system model and a reviewing method. Using this reviewing method the knowledge graph is able to make action suggestions to the hypothesis algorithm.
The results show that task execution improves because firstly, the robot gains experience and learns which objects can be manipulated, and secondly the robot finds the best strategy on how to manipulate which object. The proposed method shows comparable results to the state-of-the-art methods, whilst the proposed method combines all three topics and the multiple state-of-the-art methods are all specialized and combine only 2 out of the 3 research topics.

G.S. Groote
Delft, April 2023

Contents

List of Figures

List of Tables

Symbols

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

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

1 Introduction

This chapter narrows the broad field of robotics down to a largely unsolved problem. For that problem, the state-of-the-art methods are presented, and their shortcomings are highlighted. Section 1.1 presents the main and sub research questions that address the current gap in research. A largely unsolved problem 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, it remains a hard problem to navigate and act in new, unseen environments. It can be motivated that this is due to many challenges that the robot has to overcome. In this thesis, such challenges are categorised into 3 topics, namely: **learning object dynamics**, **NAMO** and **nonprehensile push manipulation**. The main goal of this thesis is to combine these 3 topics, a secondary goal is to investigate how these topics can strengthen each other over time. Learning object dynamics ables the robot to manipulate unforeseen objects, NAMO allows the robot to move around in an environment even if the robot's target location is blocked by an object, and nonprehensile pushing allows the robot to change the environment. Combining these 3 topics covers any task that involves relocating objects by pushing, which is a wide variety of tasks. Examples are clearing debris in construction sites or war zones or cleaning through pushing trash in one spot. Learning abilities allow robots to operate in completely new environments and improve robots to adapt to environmental changes. Crucial when the robot can encounter many different objects and unforeseen environments. Examples are exploration or rescue missions in collapsed buildings, but also in more everyday robot applications. An unfamiliar environment can emerge from a familiar environment due to some unforeseen change that the robot is not aware of. An example is a leakage that changes the friction coefficient between the floor and everything standing on it. Another example are supermarkets, due to the presence of people in the supermarket the environment changes, providing a slightly new environment for the robots that operate in them. Nonprehensile pushing is a form of manipulation that is 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, and pushing is a manipulation action many robots should leverage.

Research into approaches tackling the 3 topics just described can be split into two categories. The bulk falls into the category of hierarchical approaches [10, 14, 24, 31, 33]. The remainder falls in category locally optimal approaches [18, 22, 23]. Both approaches are elaborated upon later in this chapter. First, a number of problems are highlighted.

Joint-Configuration space The combination of robots performing NAMO *and* pushing tasks introduces the first problems. Imagine a robot that can drive and push objects around in its environment, the robot is then tasked to relocate several objects. A solution to such a task consists of a number of drive and a number of push actions, where every drive or push action acquires a path from the start to the target location. Finding a path is known as a *motion* or *manipulation planning problem* and is planned in configuration space. Configuration space can be described as an n-dimensional space related to a single

object, where n is the number of degrees of freedom for that single object. The workspace obstacles are mapped to configuration space obstacles that make up 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 *joint configuration space* emerges when the robot's configuration space is augmented with the configuration space of every object. For example, if the configuration space for both robot and objects consist of position x, y and orientation θ around the z axis (thus $n = 3$) then the joint configuration space is $3m$-dimensional, where m is the number of objects in the environment including the robot. Thus the dimensionality of the joint configuration space grows linearly with the number of objects in the robot environment, also known as the *curse of dimensionality*.

Challenges Let's revisit the robot tasked with relocating several objects. For a solution, a path is sought from the current configuration of the environment, a point in the joint configuration space to a target point in the joint configuration space where all objects are at their target position. Conventional motion planners cannot efficiently find a path because of the enormity of the joint configuration space. The enormity can be described by the following analysis. Drive actions put the robot in a new location, and push actions put an object at a new location, both influencing the configuration spaces of all other an object in the environment. Thus future planning is influenced by the actions taken now, resulting in an explosion in the number of possibilities. Another analysis that describes the hugeness of the joint configuration space are unspecified target positions. During the relocation of objects, other objects might be present in the environment. No target location is specified for such objects, whilst they could be essential to relocate in order to put the objects with target positions at their target positions. Consider a blocked corridor. The blocking object needs to be pushed to free the path but the target location of the blocking object is unspecified, as long as the robot can drive through the corridor unhindered. The target point in joint configuration space is thus not unique.

Finding an optimal solution to a NAMO and pushing task requires a search in the joint configuration space. Apart from the fact that the joint configuration space grows ridiculously fast, there is another problem. The joint configuration space is *piecewise-analytic*, which is explained below. Drive and push actions are translated to the joint configuration space as subspaces. A certain subspace of the joint configuration space is assigned to robot driving and another subspace is assigned to robot pushing. These different subspaces in joint configuration space are called different modes of dynamics. In the driving mode of dynamics, a set of driving constraints must be respected, and the pushing mode has a set of push constraints that must be respected. Such constraints originate from multiple sources, such as the robot being nonholonomic, the robots and objects properties (e.g. geometry, weight distribution), friction coefficient between objects. Multiple modes of dynamics introduce a discontinuity in the constraints, hence the joint configuration space is a piecewise-analytic. Motion planners have great difficulty crossing the boundary from one mode of dynamics to another mode of dynamcis [31].

Appendix A contains an explanation on complexity classes which may be helpful to better understand this paragraph. As mentioned before finding an optimal solution to a NAMO and pushing task requires a search in joint configuration space. Finding an optimal solution falls in category of non-deterministic polynomial-time hard (NP-hard) problems, motivation is provided with the following simplification. If the search for an optimal solution in joint configuration space is simplified by completely removing relocating objects to new positions from the task, a purely NAMO problem is what remains. If the problem is simplified even further, by assuming that every object is an unmovable obstacle, the problem falls in the category of NP-hard problems because a reduction exists from the piono mover's problem which is known to be NP-hard [21]. That a simplified version is NP-hard indicates how difficult it is to find an optimal path in joint configuration space.

The last problem that is introduced is the uncertainty of actions in unknown environments. Planning an action sequence with limited or no environmental knowledge inevitably leads to unfeasible action sequences, such as pushing unmovable obstacles. Updating the environmental knowledge and replanning the action sequence is the cure to the uncertainty introduced by a lack of environmental knowledge. Trying to complete unfeasible action sequences is time and resources lost. Additionally, it

can lead to the task itself becoming unfeasible. For example, a pushing robot ends up pushing an object into a dead end due to an action sequence planned with limited environment knowledge. Now that the object is in a stuck position 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. To find such an action sequence a path from the start configuration to a desired target configuration in the joint configuration space is sought, where all specified objects are at their specified target position. The emerging challenges are the enormity of the joint configuration space, the different modes of dynamics that make the joint configuration space piecewise analytic, and lastly the uncertainty introduced by the lack of environmental knowledge.

Now the state-of-the-art methods are discussed that can be categorized into two categories, namely locally optimal and hierarchical approaches, first locally optimal approaches are discussed.

Locally Optimal Approaches As has been indicated in the previous paragraph, finding a path in the joint configuration space cannot computationally be found in reasonable time (orders of magnitude slower than real-time, with no guarantees if no path exists). Only by leveraging simplifications applied to the joint configuration space, a search can be performed, such as considering a heavily simplified probabilistic environment [30], considering a single manipulation action [4], discretization [22] or a heuristic function combined with a time horizon [22]. 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. [23]. She presents an optimal motion planner that avoids obstacles in the workspace and respects kinematic and dynamic constraints of a robot arm [22]. Examples of the motion planner are provided using a 3- and 4- Degrees Of Freedom (DOF) planar robot arm. Sampling in the joint configuration space is simplified using discretization (by disjunctive programming) of the joint configuration space and by using a receding horizon. The disjunctive programming concept is applied for converting 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 which represent the entire path. After discretization the joint configuration space remains huge. 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 [22] is then converted toward path planning for a nonholonomic mobile robot with a gripper [18]. 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 [24]. An MPC controller then tracks the path and compensates for modeling errors. A key 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 [23]. She has presented a manipulation planning framework focused on moving legged objects in which the robot has to choose between which leg to push or pull. The framework can operate in real-time, and the local optimality has been shown. From the 3 topics that 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. Prehensile manipulation is considered easier in comparison with nonprehensile manipulation because

it is harder to disconnect a gripped object.

Hierarchical Approaches The second class of approaches to finding a path in joint configuration space is classified as hierarchical approaches [10, 14, 24, 31, 33] 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 which includes several atomic actions and their sequencing. Whilst a low-level controller acts to accomplish a single action that acts in a single mode of dynamics (e.g. drive toward object, push object), by sending input signals toward the robot actuators. 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 maximal for a single action.

The most relevant hierarchical approach presenteded by Scholz et al. [24]. 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 free space regions can be connected by manipulating the object that separates these 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 a number of drive and object manipulation actions to eventually drive the robot toward a target position. 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. The method's effectiveness in learning and driving toward a target location has been shown by an implementation on a real robot. From the 3 topics that this thesis focuses on, Scholz et al. includes learning and the NAMO problem, missing only push manipulation toward target locations. By not including manipulation of 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 the manipulation of an object does not affect the feasibility of the global path, hence the simplification.

Both local optimal and hierarchical approaches have been discussed, both having their advantages and disadvantages. Local optimal approaches can theoretically converge to a global optimal plan. To avoid the curse of dimensionality simplifications must be used to sample the joint configuration space in order to be computationally feasible. Such simplifications determine the quality of solutions found. Hierarchical structures generally provide solutions which are computationally efficient 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 [31].

The most relevant work for local optimal [23] and hierarchical [24] approaches are discussed. They both [23, 24] have in common that they both combine 2 of the 3 topics (learning, the NAMO problem, object manipulation toward target positions). Note both relevant works focus on prehensile manipulation, whilst this thesis focuses on nonprehensile push manipulation. Individually a considerable amount of research is done on these 3 topics (NAMO [5, 9, 10, 12, 15, 33], nonprehensile push manipulation [2, 3, 16, 27, 28, 29], learning object dynamics [6, 25]). Combining two topics received little attention by 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.

Author	Citation	Learns object dynamics	NAMO	Specify object target positions	Object Manipulation
Ellis et al.	[10]	✓	✓	✗	pushing
Sabbagh Novin et al.	[23]	✓	✗	✓	grasp-push grasp-pull
Scholz et al.	[24]	✓	✓	✗	graph-push grasp-pull
Vega-Brown and Roy	[31]	✗	✓	✓	gripping
Wang et al.	[33]	✓	✓	✗	pushing
Groote	proposed solution	✗/✓	✓	✓	pushing

Table 1.1: Overview of 3 topics in recent literature and their object manipulation, where *grasp-push* and *grasp-pull* refer to prehensile push and pull manipulation, *gripped* refers to fully gripping and lifting objects for manipulation, *pushing* refers to nonprehensile push manipulation. The proposed method shows ✗/✓for learning system dynamics because it proposes system identification to generate a system model, however for the implementation an hardcoded system model is used.

The main contribution of this thesis is to combine all three topics. These topics are learning object dynamics, the NAMO problem and nonprehensile push manipulation. The proposed method combines these 3 topics with the *hypothesis algorithm*. The algorithm builds a graph-based structure with nodes and edges, named the *hypothesis graph*. Planning directly in the joint configuration space is avoided. 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 [14]. Learned object dynamics are stored in a knowledge base called the *knowledge graph*. The halgorithm, hgraph and kgraph are introduced in Chapter 3.

1.1. Research Question

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

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. Essentially the first research subquestion asks "how does the proposed method work?", which allows to explain the proposed method. The second research subquestion asks: "How does it compare to state-of-the-art methods?", allowing to compare the proposed methods with existing state-of-the-art methods.

Research subquestion:

1. Can the proposed method combine learning and planning for push en drive applications with a technique known as backward search?
2. How do learning system models and remembering interactions compare to only learning system models? And, how does the proposed method compare against the state-of-the-art?

Answering the research subquestions provides a solid base to answer the main research question. The main research question is aimed to test robot abilities in a new environment and tracking improvement in that new environment. Some questions which come up are: Will the robot prefer specific strategies for certain objects? How much improvement will a robot make with some experience? Will the robot

converge to a preferred strategy for an object, and will it converge to the same strategy again if its memory is wiped. At the end of this thesis, answers will be given to such questions.

1.2. Problem Description

To answer the research questions, tests will be 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. The robots to test should be flat robots that have a low center of gravity, which lowers the chance of tipping over. A 3-dimensional environment is selected. Mostly the 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).

Let's start with defining the environment. Let the tuple $\langle$Origin, 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, $Obj = (ob_1, ob_2, ob_3, \ldots, ob_i)$ with $i \geq 1$, an object is a 3-dimensional body with shape and uniformly distributed mass. 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 behavior of objects such as gravity, interaction with the ground plane or interaction with other objects. In recent literature, the motion equations are also named the true dynamics.

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 $pos_x, pos_y \in \mathbb{R}$, $\quad pos_\theta \in [0, 2\pi)$, $\quad k$ indicates the time step and can be removed for simplicity if the configuration remains constant for all k. *id* is short for identifier and indicates the object to which this configuration belongs.

1.2.1. Task Specification

The research questions want to investigate the effect of learning system models and monitor the effect of learned knowledge over time. Thus the robot needs an incentive to learn object properties, and interactions with the objects in its environment, otherwise it would simply remain standing still in its initial location. Therefore the robot is asked to complete a task. A task is defined as a subset of all objects with associated target configurations

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

where $Ob_{task} = (ob_1, ob_2, ob_3, \ldots, ob_k) \subset Ob$, $C_{target} = (c_1, c_2, c_3, \ldots c_k)$ and $k > 0$.

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

1.2.2. Assumptions

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

> ***Closed-World Assumption:*** *Objects are manipulated, directly or indirectly only by the robot. Objects cannot be manipulated by influences from outside the environment.*
>
> ***Perfect Object Sensor Assumption:*** *the robot has full access to the poses and geometry of all objects in the environment at all times.*
>
> ***Tasks are Commutative Assumption:*** *Tasks consist of multiple objects with specified target positions. The order in which objects are pushed toward their target position is commutative.*
>
> ***Objects do not tip over Assumption:*** *Movable objects slide if pushed.*

The assumptions taken serve to simplify the problem of task completion. Note that in Section 5.1 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.

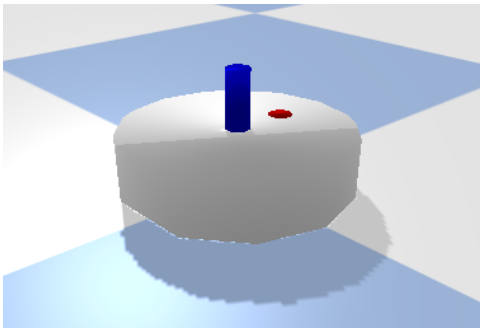
Assumptions might have certain implications, which are listed below. The **closed-world assumption** implies that objects untouched by the robot and with zero velocity component remain at the same position. Completed subtasks are therefore assumed to be completed for all times after completion time.

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 [32].

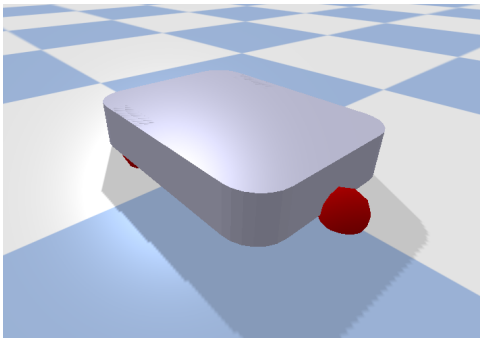
Certain tasks are only feasible if performed in a certain order (e.g. the Tower of Hanoi). The **tasks are commutative assumption** allows focusing only on a single subtask since it does not affect the completion or feasibility of other subtasks.

The **objects do not tip over assumption** ensures that objects do not tip over and suddenly have vastly different dynamics. In practice, objects will not be higher than the minimum width of the object, and spheres are excluded from the environment. This experimental strategy seemed sufficient whilst running experiments but does not guarantee that objects do not tip over.

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.

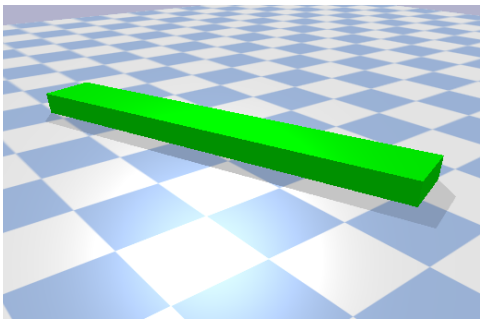


(a) The holonomic point robot the 2 velocity inputs drive the robot in x and in y direction

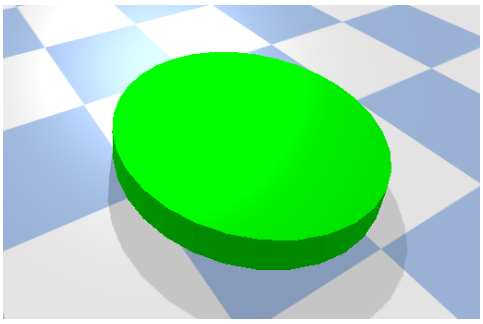


(b) The nonholonomic boxer robot, the first velocity input drives the robot forward or backward the second rotates the robot

Figure 1.1: Robots used for testing the proposed method



(a) A box object



(b) A cylinder object

Figure 1.2: Various objects in the robot environment

For complete environments with accompanying tasks, see Chapter 4.

1.3. Report Structure

The proposed method heavily relies on a number of methods and functions. These methods and functions are conveniently grouped in Chapter 2. Then the proposed method is presented and discussed in Chapter 3. The proposed method is followed by a chapter committed to testing the proposed method, presented in Chapter 4. The last chapter dedicates itself to drawing conclusions on tests and answering the research questions in Chapter 5.

2

Required Background

This chapter presents the components that the proposed method relies upon, the required background. These components are: system identification, control methods, planning and finally fault detection. Every component plays a crucial role for the proposed method and their performance influences the proposed methods overall performance. System identification is responsible for converting Input-Output (IO) data into a system model, Section 2.1. System is required by control methods in order to create stable control and to track a reference signal. Control methods are presented in Section 2.2. Planning is a core component of the proposed method and is responsible for finding a path in configuration space. Such a path acts as reference signal for the controller and planning detects objects that are blocking the path, and must be removed first. The planning component is split into path estimation in Section 2.3.1, motion planning in Section 2.3.2 and manipulation planning in Section 2.3.3. Lastly fault detection halts persisting faulty behavior and will be elaborated upon in Section 2.4.

2.1. System Identification

Understanding the world is captured by models, *system models* that estimate the behavior of real-world systems. For example what trajectory does an object take after receiving a push? The trajectory is dependent on the properties of the object (e.g. mass, geometry), and interaction with its surroundings (e.g sliding friction). These properties and interactions are captured by a system model. Just as applicable constraints that are captured by a system model, see both robots in Figure 1.1, the holonomic point robot in Figure 1.1a can drive without constraints, and the boxer robot in Figure 1.1b can drive forward, backwards and can rotate. A system model for robot driving for the boxer robot should thus capture that it is not able to drive directly sideways.

This thesis encounters only 2 different robots but many different objects. Since every object can be different in type, weight and dimensions. System identification methods are required to capture the variety of objects that the robot can encounter.

2.2. Control Methods

This section elaborates why control is required and which control methods are best suitable for various control applications. During this thesis, the effect of the robot interacting with objects is captured by system models. In addition to predicting output with system models, control methods leverage the prediction that system models provide to perform actions, in this thesis drive and push actions. A first requirement for a controller is that it should yield a stable closed-loop control because that guarantees converging toward a set point. Secondary requirements are then desired such as a low prediction error, low tracking error and low final placement error, see Section 2.4. The 2 implemented control methods are discussed below.

Model Predictive Control 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 [20]. The best feasible input is found every time step by optimising the objective function whilst respecting constraints. Tuning is accomplished by modifying the weight matrices in the objective function, or by modifying the constraint set. Solving an objective function to find the best feasible input generally yields robust control. It is however required that the system model is Linear Time-Invariant (LTI). System models for driving the robot can be estimated with 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.

Model Predictive Path Integral Control 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 several "rollouts" for a specific time horizon, [17]. 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 penalized undesired states and rewards 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 compatible with nonlinear system models. Whilst drive applications can be accurately estimated by linear models, 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.

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 best 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, the performance is measured with various metrics to which Section 3.2.3 is dedicated.

2.3. Planning

This section explains planning for a single drive or push action, planning for a single action is referred to as *local planning*. Local planning consists of 2 steps, firstly path estimation, and secondly motion or manipulation planning. Path estimation checks if a path can be found from start to target configuration whilst neglecting nonholonomic constraints. The path estimator can detect non-existent paths and filters out a number of unfeasible drive or push actions. For the remainder of actions, motion or manipulation planning is responsible for finding a path from starting point to a target point in configuration space. A path is a list of successive configurations beginning with the start and ending with the target configuration. During planning, constraints are respected by checking the reachability of the configurations added to the path with a system model. The planner has to account for multiple subspaces in configuration space (free, obstacle, movable and unknown space), an existing motion planner [5] is extended to incorporate movable and unknown space next to free and obstacle space. The modification incentivises the planner to find a path in free space but is allowed to pass through unknown or movable space if necessary. The robot should first take care of the blocking objects if a planned path crosses unknown or movable subspace.

2.3.1. Estimating Path Existence

In this subsection motivation and explanation for estimating path existence is presented. We can describe the path estimation algorithm as: *The idea is to discretize the configuration space with a finite discretization. The emerged cells act as nodes in the graph, cells are connected through edges to nearby cells. Graph-based planners start from the cell containing the starting pose and search for the cell containing the target pose whilst avoiding cells which lie in obstacle space.*

Discritizing the Configuration Space For general geometric shapes, a configuration space can be constructed and discretized. During this thesis, an implementation of configuration space is made for cylinders and cuboids. Configuration space for objects with unknown shapes can be estimated by constructing the configuration space of a cylinder or cuboid. 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, 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 cuboids become rectangles. The definition of configuration space is presented below for a circular object, such as the point robot without loss of generality.

Configuration space is represented by a grid of cells. 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:

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

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

Where $c_{(i,j)}$ in matrix C-space represents to which subspace the cell at indices (i, j) belongs. The 4 subspaces considered in this thesis are free-, obstacle-, movable- and unknown space and are 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 as follows: 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 has a higher order of importance than the unknown space.

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

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

and is defined for:

$$(x, y) \in [-\frac{x_{grid}}{2}, \frac{x_{grid}}{2}] \times [-\frac{y_{grid}}{2}, \frac{y_{grid}}{2}]$$

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

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

and is defined for:

$$(i, j) \in [0, \frac{x_{grid}}{s_{cell}}] \times [0, \frac{y_{grid}}{s_{cell}}]$$

For rectangular objects, the orientation of the object becomes important because the orientation together with the (x, y) coordinates determines in which subspace the object resides. The dimension for rectangular objects is thus 3 whilst circular objects have a 2-dimensional configuration space.

The configuration space for a rectangular object is defined as:

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

with $0 \leq i < i_{\max} = \frac{x_{grid}}{s_{cell}}, \quad 0 \leq j < j_{\max} = \frac{y_{grid}}{s_{cell}}, \quad 0 \leq k < k_{\max} = 2\pi.$

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

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

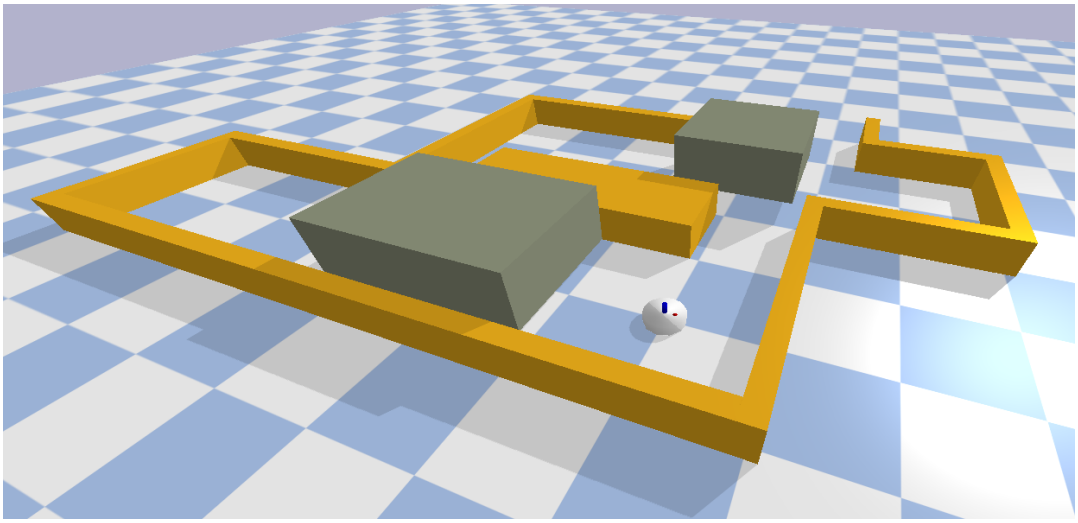
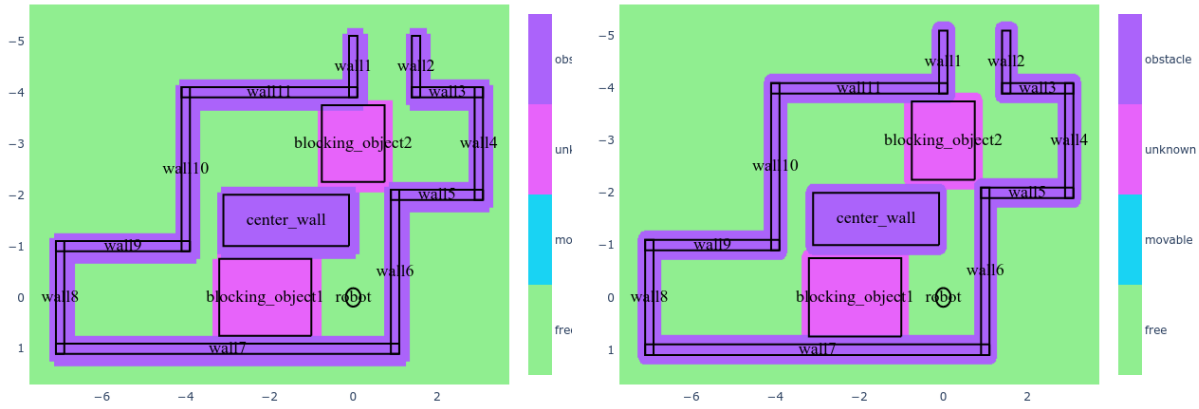


Figure 2.1: Environment with the point robot, unmovable yellow walls and movable brown boxes.

The point robot has a cylindrical shape thus a 2-dimensional configuration space is created and can be seen in Figure 2.2.



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

(b) C-space^{circle} with $s_{cell} = 0.01$

Figure 2.2: The configuration space for the point robot with different cell sizes. The robot environment corresponds to the environment presented in Figure 2.1.

The resolution of the configuration space at Figure 2.2b is higher compared to Figure 2.2a because of the smaller grid size. The difference in resolution is especially visible at the corners of the walls. A high resolution is better at detecting paths through small corridors and tight corners, but it comes at the cost of a longer creation and search time.

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

A path in configuration space is detected using the implemented $f_{shortest\_path}(c_{start}, c_{target}, detect\_blocking\_objects)$ function. This function returns the shortest path from the c_{start} to the c_{target}. *detect_blocking_objects* is a boolean flag that if *True* returns a list blocking objects that are classified as movable or unknown. If no path can be found, the $f_{shortest\_path}$ function raises an error.

The *shortest_path* function uses the Dijkstra algorithm [8] on the discretizing configuration space to find a shortest path. This path can be converted to provide an initial number of samples for motion or manipulation planner, also referred to as a "warm start".

Unfeasible solutions and an undecidable problem The path estimation algorithm does not take system constraints into account. It is thus possible that the path estimation algorithm finds a list of neighboring cells from the start to the target configuration and concludes that there exists a path. 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 motion or manipulation 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 [35]. This is due to the chosen cell size during discretizing the configuration space. An example is a corridor that has exactly the width of the robot, the robot does fit exactly through this corridor. Detecting such a path requires a number of neighboring cells that lie exactly in the center line of the corridor. Only with a cell size going to zero, and the number of cells going to infinity such a path is guaranteed to be detected. Path non-existence on the other hand is more easily to proof, because the path estimation algorithm provides an upper bound on existing paths and a lower bound on non-existing paths [35].

The path existence algorithm can detect non-existence paths. Thus checking path existence before motion or manipulation planning filters out several non-existent paths that would otherwise waste time and resources. Even if in exceptional cases the path estimation algorithm can yield unfeasible paths and can fail to detect existing paths. Checking path existence before motion or manipulation planning filters a number of non-exist paths and is additionally motivated by two reasons. First, path estimation is orders of magnitude faster compared to motion or manipulation planning. Secondly, the path estimation algorithm can provide a number of initial samples to the motion or manipulation 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. 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, where the first point is the start configuration and the last point is the target configuration. How far the successive points can lie apart is a tuning parameter of the planner. Seeking a path from start to target inside a configuration space whilst avoiding obstacles for the robot to track is referred to as *motion planning*. 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 optimised Rapidly-exploring Random 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, Section 2.3.3 dedicates itself to manipulation planning. The new planning algorithms are sample-based, which 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 motion planning algorithm considers as a "black box." [15]"

Generally, the configuration space motion planners plan consists of 2 subspaces, free and obstacle space. The configuration space in this thesis consists of 4 subspaces, namely free, obstacle, unknown and movable space. To solve motion planning problems for such a configuration space a dedicated motion planning algorithm has been developed that extends the existing algorithm extends the existing double tree RRT* algorithm [5]. The motion planner consists of.

$$V_{MP} : \text{A set of nodes}$$
$$E_{MP} : \text{A set of edges}$$
$$P : \text{A set of paths}$$

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. When the start connectivity tree is close enough (inside the search size of a newly added sample) to the target connectivity tree a path from start to target is found. 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 PathCost corresponds to the sum of euclidean distances between the successive configurations in the path, the MovableSpaceCost and UnknownSpaceCost correspond to a fixed addition cost for a configuration in the path crosses through movable or unknown subspace respectively. Here 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. By optimizing the path for the lowest cost the motion planning algorithm is

incentivized to find a path around unknown or movable objects. However, if there is no path in free space only, the algorithm plans through movable or unknown space. Newly sampled configurations are added in a structural manner that guarantees an optimal path is found with infinite sampling [5]. Where optimality is defined as the path with the lowest possible cost.

Tuning Parameters The algorithm has 4 tuning parameters that can be tweaked, first, the *step size*, the maximal normalized distance between connected samples in the connectivity trees, see Figure 2.3c for a visual example. Choosing a high step size will increases the 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. 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, Figure 2.3d). In this subspace, a parent node is sought and rewiring occurs. A new node is connected with an edge to a connectivity tree through a parent node. After connecting the new node to its parent node, rewiring occurs which is 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 2.3e. Increasing the search size improves the choice of 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 2.5 clearly shows the effect of varying such costs.

Pseudocode of the proposed algorithm is provided in Algorithm 1. The functions used are elaborated upon in the following Table 2.1.

x:	A node representing a point in configuration space
x_{init}:	Creates a start and target node
NotReachStop:	True if the stopping criteria is not reached
$Sample_{random}$:	Creates a random sample 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 sample x and x'
$CostToInit(x)$:	Find the total cost from x to the initial node
$ReachabilityCheck(x, x')$:	Return true if a system model with initial state x is able to reach state x' for a range of allowed input, otherwise, return false.
$InSameTree(x, x')$:	Returns true if both x and x' are in the same tree, otherwise return false

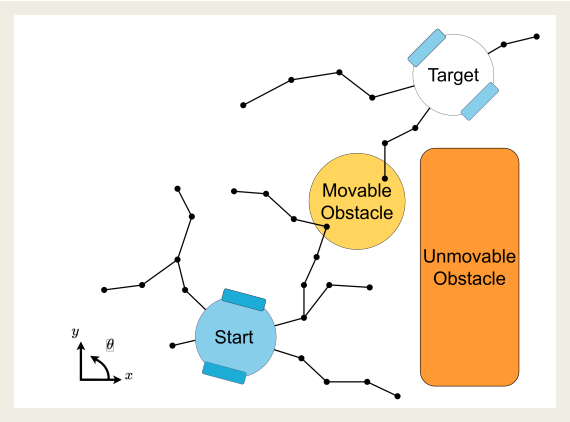
Table 2.1: Functions used by the Algorithm 1

Notice that in Algorithm 1 the colored sections correspond to the surrounding colored border in the subfigures of Figure 2.3.

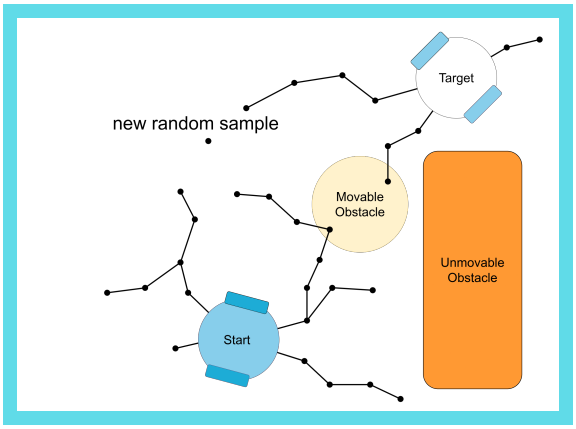
Algorithm 1 Pseudocode for modified double tree RRT* algorithm

$V_{MP} \leftarrow x_{init}$
while *NotReachStop* **do**
  $Cost_{min} \leftarrow +\infty$ ▷ Create, check and project a new random sample
  $v_{rand} \leftarrow Sample_{random}$
  $v_{nearest} \leftarrow Nearest(v_{rand}, V_{MP})$
  $v_{temp} \leftarrow Project(v_{rand}, v_{nearest})$
  if $CollisionCheck(v_{temp})$ **then**
    $v_{new} = v_{temp}$
  else
    Continue
  end if
  $X_{near} \leftarrow NearestSet(v_{new}, V_{MP})$ ▷ Find and connect new node to parent node
  for $v_{near} \in X_{near}$ **do**
    $Cost_{temp} \leftarrow CostFromInit(v_{near}) + Distance(v_{near}, v_{new}) + ObjectCost(v_{near}, v_{new})$
    if $Cost_{temp} < Cost_{min}$ **then**
      if $ReachabilityCheck(v_{new}, v_{near})$ **then**
        $Cost_{min} \leftarrow v_{temp}$
        $v_{minCost} \leftarrow v_{near}$
      end if
    end if
  end for
  if $Cost_{min} == \infty$ **then**
    Continue
  else
    $V_{MP}.add(v_{new})$
    $E.add(v_{minCost}, v_{new})$
  end if
  $Cost_{path} \leftarrow +\infty$ ▷ Check if the newly added node can lower cost for nearby nodes
  for $v_{near} \in X_{near}$ **do**
    if $InSameTree(v_{near}, v_{new})$ **then**
      $Cost_{temp} \leftarrow CostFromInit(v_{new}) + distance(v_{new}, v_{near}) + ObjectCost(v_{new}, v_{near})$
      if $Cost_{temp} < CostFromInit(v_{near})$ **then**
        if $ReachabilityCheck(v_{new}, v_{near})$ **then**
          $E.rewire(v_{near}, v_{new})$
        end if
      end if
    else ▷ Add lowest cost path to the list of paths
      $Cost_{temp} \leftarrow CostFromInit(v_{new}) + distance(v_{new}, v_{near})$
        $+CostFromInit(v_{near}) + ObjectCost(v_{new}, v_{near})$
      if $Cost_{temp} < Cost_{path}$ **then**
        if $ReachabilityCheck(v_{new}, v_{near})$ **then**
          $Cost_{pathMin} \leftarrow v_{temp}$
          $v_{pathMin} \leftarrow v_{near}$
        end if
      end if
    end if
    if $Cost_{pathMin} == \infty$ **then**
      Continue
    else
      $P.addPath(v_{new}, v_{pathMin}, Cost_{pathMin})$
    end if
  end for
end while

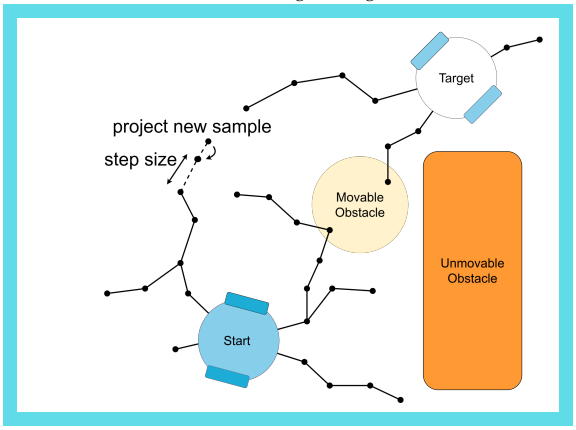
Now an example is provided of the proposed algorithm, see Figure 2.3. In this example, one single sample is added to the starting connectivity tree. Adding this sample involves many steps, which are generating a new random sample, projecting the sample to the nearest node, rewiring nearby samples and connecting the start to the target tree.



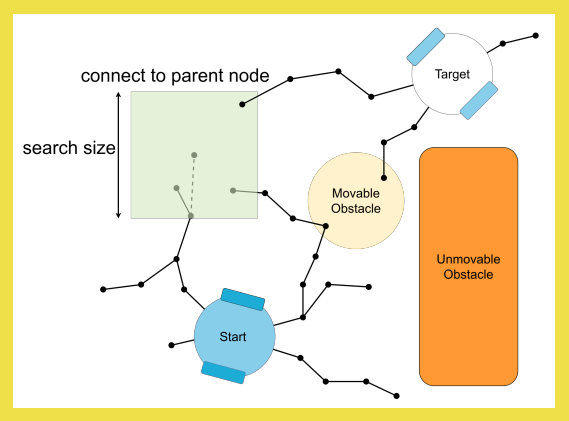
(a) Snapshot of the configuration space during a search from start to the target configuration.



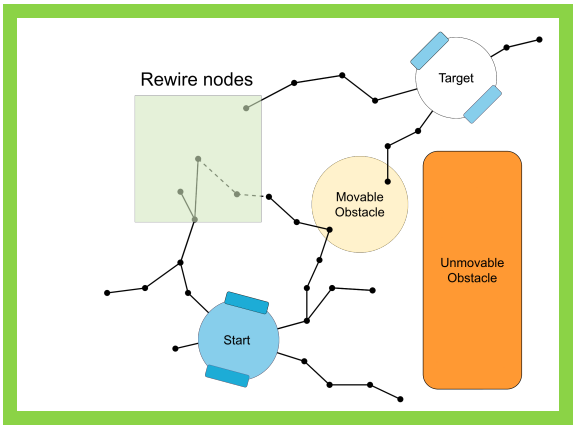
(b) A new random sample is generated.



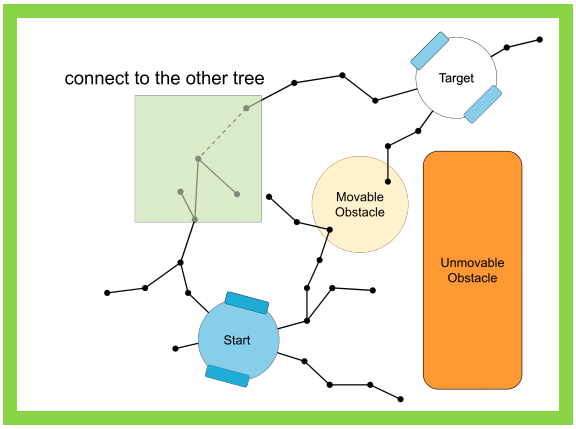
(c) The new sample is projected toward the closest sample.



(d) The new sample is connected to the node in search space that results in the lowest cost.

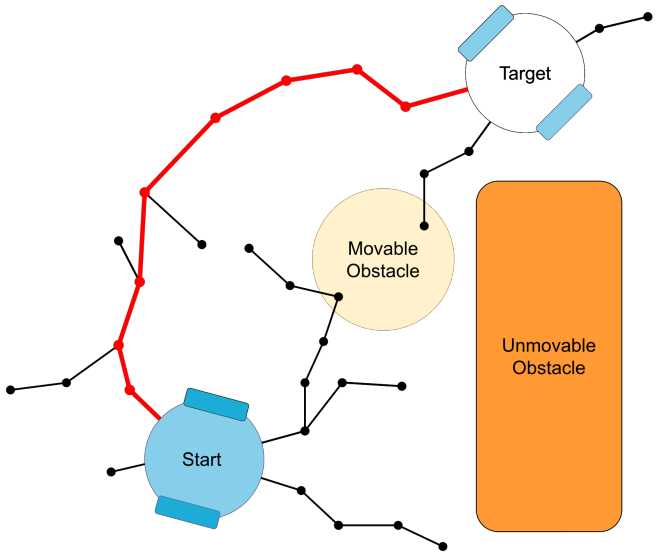


(e) Nodes for which the cost can be lowered from the new sample are rewired.

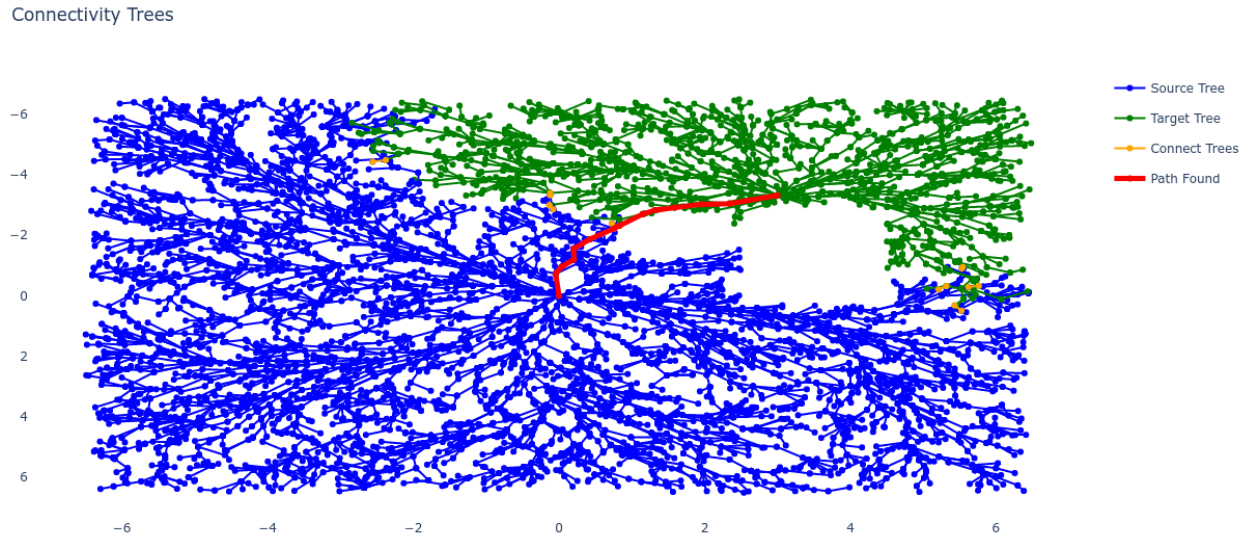


(f) A path from start to target configuration is found.

Figure 2.3: Visualization of the Double tree RRT* motion planner that adds a single sample to the connectivity graph. The color of the box surrounding subfigures corresponds to the colored sections in Algorithm 1. 3-dimensional configuration space displayed as 2-dimensional configuration space (x and y are visible, θ is not visible).



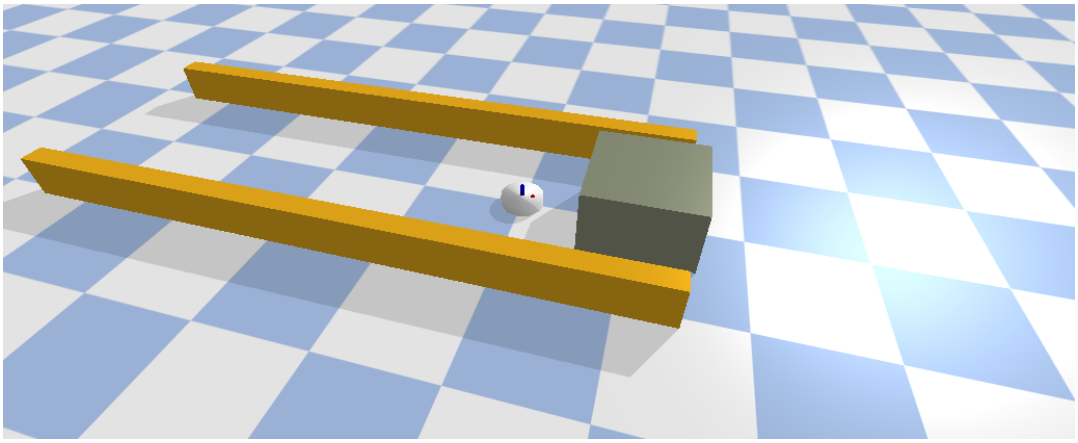
(a) Motion planner found a path found marked in red.



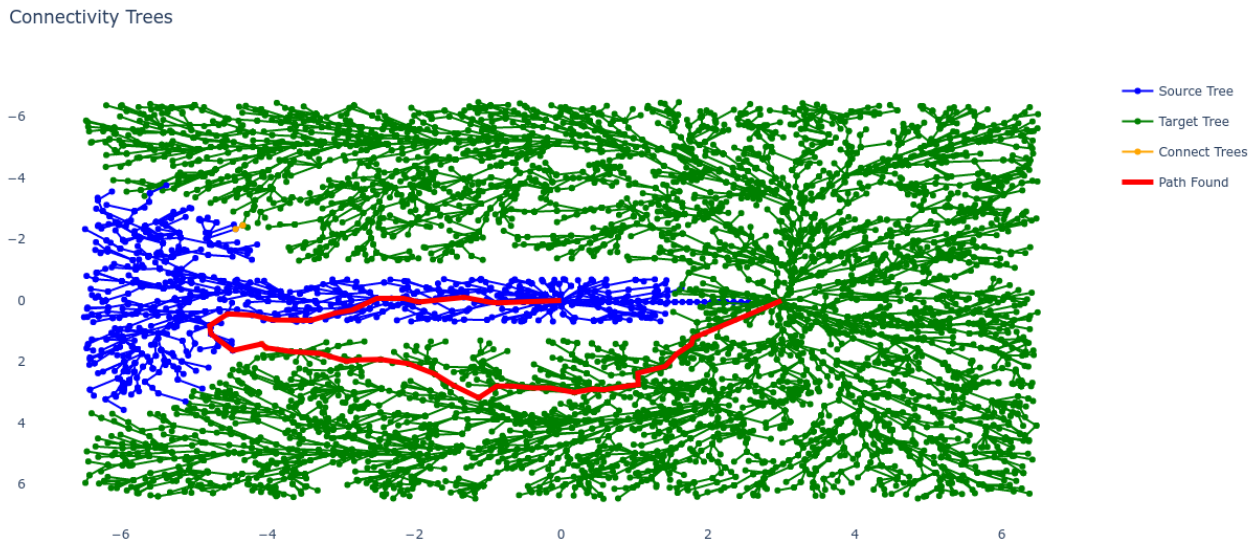
(b) A visualization of the implemented RRT* algorithm when the stopping criteria was reached.

Figure 2.4: Comparing schematic example to a visualization of the real algorithm.

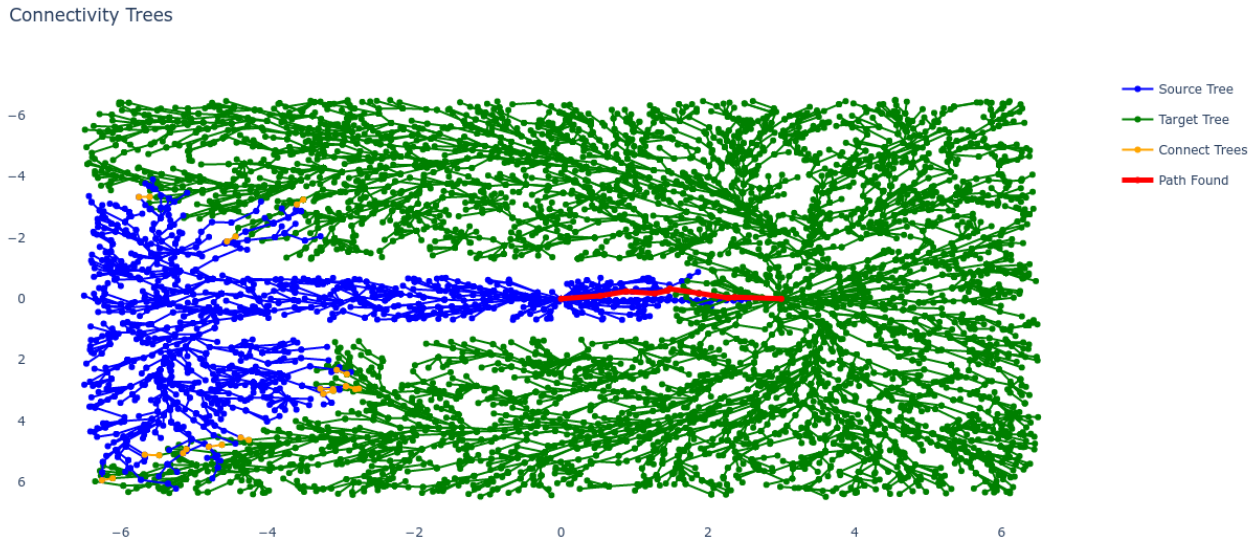
The added fixed cost for a path crossing through a movable or unknown object motivates the motion planner to find the shortest path around objects but prefers moving an object over making a large detour. Tuning the additional fixed cost for a path crossing through movable or unknown space balances the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



(a) Robot environment with the point robot, 2 yellow unmovable walls and an unknown brown box.



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



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

Figure 2.5: The robot tasked with driving toward the other side of the brown box.

The proposed motion planning algorithm searches the configuration space from the start connectivity tree and the target connectivity tree. Exploring faster compared to the single tree RRT* algorithm because two trees grow and explore faster than a single tree. The proposed algorithm rewires nodes, resulting in lowering the cost for existing paths, and converting to the optimal lowest-cost path with infinite sampling. System constraints are ensured by the $ReachabilityCheck$ 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 it prevents a large detour. There are no performance tests taken on the modified motion planner other than visual inspection. 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.

2.3.3. Manipulation Planning

With a push, two objects are primarily involved, the pushed object and the robot. Generally, and in this thesis, the pushed object's configuration is more important than the robots configuration. The robot is only a means to push the object toward the target configuration. At which final configuration the robot itself end up is of lesser importance. As long as during the push, the robot does not collide with objects other than the pushed object, and constraints on the robot must be respected.

To plan a path that respects the constraints, the robots configuration is generated for every newly added sample in the manipulation planning algorithm. The *ReachabilityCheck()* (see Table 2.1 and line 33 in Algorithm 1) generates the robot configuration to validate if a new sample is reachable from an existing sample. This additional configuration is stored to create only feasible paths that respect the applied constraints. When the stopping criteria is reached and a shortest path is found, the generated robot configurations are discarded. Figure 2.6 displays a visual example of the procedure.

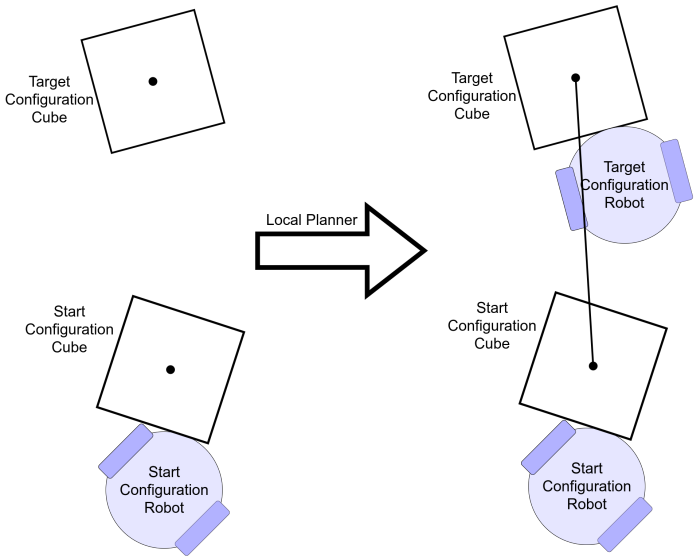


Figure 2.6: Generating a new robot configuration whilst adding a sample to the connectivity tree during manipulation planning.

2.4. Monitoring Metrics

In the next chapter, the proposed method is discussed, for this section, it is important to know that the proposed method 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 should not occur, because of the Section 1.2.2, but in the real world, an unexpected blocking obstacle or unstable controller is more likely to occur.

Detecting controller faults is a large robotic topic [11] 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. Prediction Error (PE) where the predicted state is compared with the measured state of the system. Definitions of PE is provided below, and Table 2.2 provides insight in which monitoring metric would catch what faulty behavior.

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

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

The PE 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 the PE.

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

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

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.

Table 2.2: Monitor metrics used to monitor if a fault occurred during the execution of an edge

3

The Hypothesis and Knowledge Graph

*This chapter is dedicated to introducing and defining the proposed method, the proposed method consist of the hypothesis algorithm, the hypothesis graph and the knowledge graph. The **hypothesis algorithm** acts on the **hypothesis graph** is responsible for searching the joint configuration space for action sequences to complete a specified task, Section 3.1.1 is dedicated to the halgorithm and the hgraph. The halgorithm additionally is responsible for collecting new knowledge of the environment. Gathered new environment knowledge is stored in the **knowledge graph**, defined in Section 3.2.1. Collected knowledge is rated and ordered from "good" to "bad" experiences using various metrics. Such an ordering within collected knowledge makes the kgraph a knowledge base that can be queried for action suggestions. Figure 3.1 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 3.1, and for the knowledge graph in Section 3.2.*

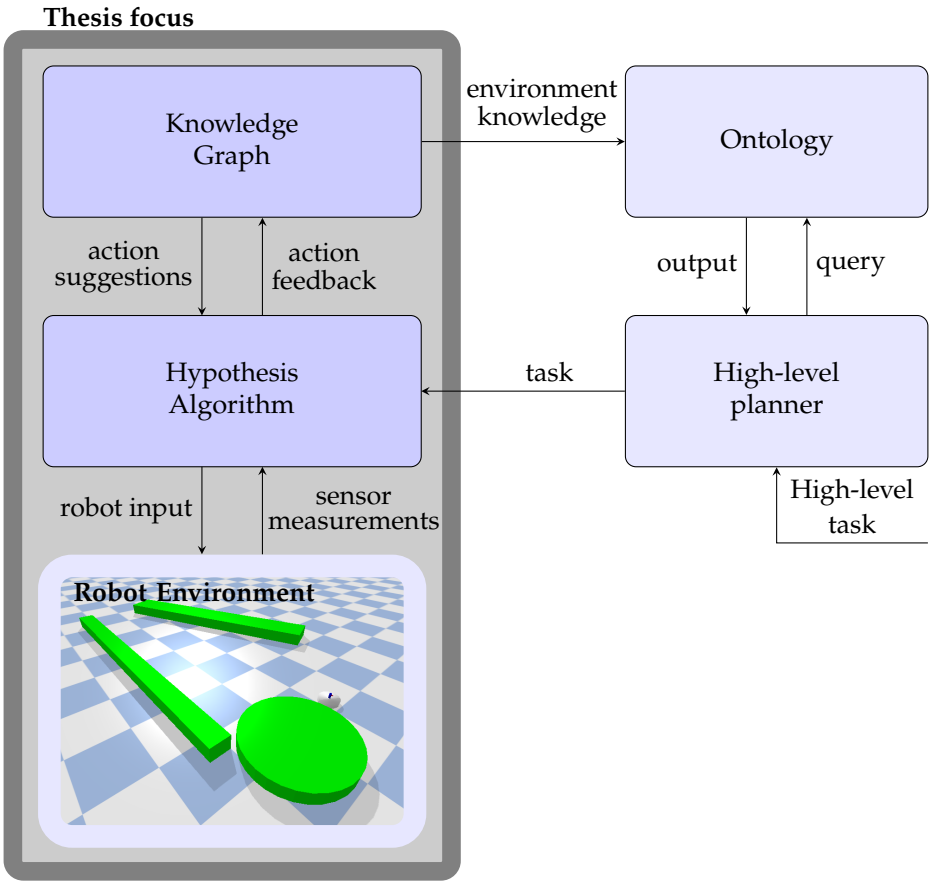


Figure 3.1: Flowchart representation of the proposed method.

As the above figure shows, 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.

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

Task:	Tuple of objects and target configurations. task = $S = \langle Obj_{task}, C_{targets} \rangle$
Subtask:	A single object, and a single target configuration. subtask = $s = \langle obj_{subtask}, c_{target} \rangle$
Object Class	Classification assigned to an object. OBJ_CLASS = Unknown ∨ Obstacle ∨ 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. NODE_STATUS = Initialised ∨ Completed ∨ Failed
Node:	A node in the hgraph, represents an object in a configuration with reachability indicated with a node status. node = $v = \langle$status, $obj, c\rangle$
Edge Status:	Status of a edge, elaborate information on the statuses can be found in Figure 3.3. EDGE_STATUS = Initialised ∨ PathExists ∨ SystemModel∨ PathPlanned ∨ Executing ∨ Completed ∨ Failed
Edge:	Edge connecting a node to another node in the hgraph or kgraph. edge = e = $\langle$status, id_{from}, id_{to}, verb, controller, dynamic model, path$\rangle$
Hypothesis:	Sequence of successive edges in the hgraph, an idea to put a object at it's target configuration. If executed and successfully completed, a subtask is completed. hypothesis = $h = [\, e_1, e_2, e_3, \ldots e_m \,], \quad m > 0$
Hypothesis Algorithm:	Graph based algorithm that searches for hypothesis in the hgraph to complete subtasks eventually completing a task.
Hypothesis Graph:	Collection of nodes and edges. For every subtask a start and target node exist in the hgraph, the halgorithm searches for a path through nodes and edges to connect start to target node. hgraph = $G^{hypothesis} = \langle V_H, E_H \rangle$
Knowledge Graph:	Collection of nodes and edges. The kgraph acts as a knowledge base and can be queried for an action suggestion. kgraph = $G^{knowledge} = \langle V_K, E_K \rangle$

Table 3.1: Terminology of terms used

3.1. Hypothesis Graph

The hypothesis graph (hgraph) consists of a set of nodes and edges. Where the node correspond to an object at a configuration, and the edges correspond to actions. As a whole the hgraph represents a

search (or planned search) in the joint configuration space. The halgorithm creates and updates nodes and edges in the hgraph and is discussed in Section 3.1.2. A search in the joint 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 created specifically 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 halgorithm with the hgraph have a familiar structure compared to some recent literature [10, 33]. An important distinction is that the proposed method in this thesis aims to combine the 3 topics: learning object dynamics, solving NAMO problems and nonprehensile pushing. Recent literature is able to only combine one or two topics of the three.

In the upcoming section the hgraph is defined and discussed in Section 3.1.1. The halgorithm is then discussed and in Section 3.1.2, where an explanation is provided on how the halgorithm searches for a solution in the joint configuration space. The section is concluded with an extensive example.

3.1.1. Definition

Before defining the hgraph, some definitions are defined on which the hgraph depends. First, recall the **configuration** defined in the Section 1.2.

Formally a **configuration**, $c_{id}(k)$ is a tuple of $\langle x(k), y(k), \theta(k) \rangle$
where $x, y \in \mathbb{R}, \quad \theta \in [0, 2\pi)$

An object holds the information about an objects configuration and shape of the object.
Formally, a **object**, $obj_{id}(k) = \langle c(k), shape \rangle$

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

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

An edge describes the details of how a node transitions to another node in the hgraph. In the robot environment, an edge represents a change of configuration of an object. Edges are split into 2 categories, system identification edges that have as goal to collect IO data and generate a system model, and action edges that steer a system toward a target configuration. The different goals make action and system identification edges very different, which is why the distinction was make. At the same time the edges both represent a change in the environment as a result of the robot driving, pushing or collecting IO data. The edges are formally defined as:

A **identification edge**,

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

With id_{from} and id_{to} indicating the node identifier of the node in the hgraph where the edge start from and point towards respectively, identification method indicates the used method that converts IO data to a system model, the controller contains the control method used for driving the robot during the collection of IO data and input contains multiple sequences of input to send toward the system. For an identification edge it is important that it is compatible with the controller residing in the corresponding action edge.

A **action edge**,

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

With id_{from} and id_{to} indicating the node identifier of the node in the hgraph where the edge start from and point towards respectively, verb 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 the nodes and edges have been defined, the hgraph can be defined.

Formally, a **hypothesis graph**, $G^{hypothesis} = \langle V_H, E_H \rangle$
comprising $V_H = \{V_{H_i}^{obj}\}, \quad E_H \in \{e_{(i,j)} | E_{Hi}, E_{Hj} \in \{V_H{}^{obj}\}, i \neq j\}$.

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

Status, Types and Lifetime of edges System identification and tracking a path are so very different, the edges are split into two categories, identification edges and action edges. An identification edge, which is responsible for sending an input sequence to the system and recording 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 dynamical model which is augmented with a corresponding controller is closed-loop stable. The status of an identification edge can be visualized in the following Finite State Machine (FSM).

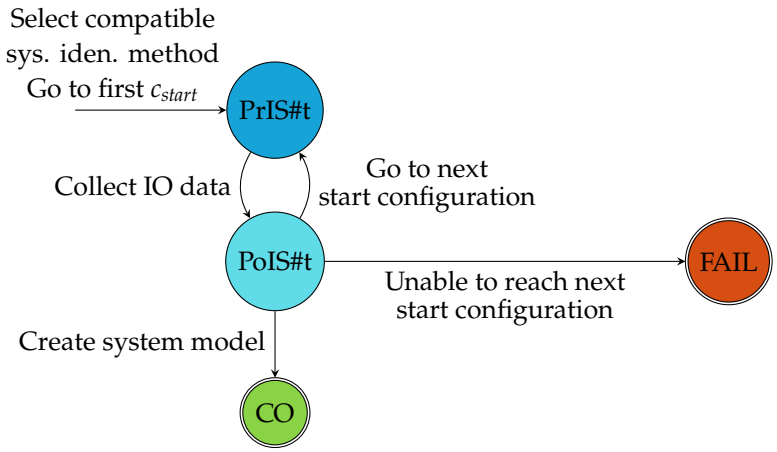


Figure 3.2: FSM displaying the status of an identification edge

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

An identification edge corresponds to an action edge, because its goal is to generate an system model to then hand over its corresponding action edge. The system identification method is selected after the

action edge is selected such that the system identification method yields a system model compatible with the controller that resides in the action edge. Two types of system models generated, system models that describe the driving behavior of the robot, and system models that describe the push behavior of the robot and an object. Data collection for a driving model can be collected by sending input to the robot assuming that the robot has enough free space around it such that is does not collide with an object during data collection. To collect IO data for a pushing model the robot first drives to a starting configuration next to the object. After the first input sequence is over, the robot drives to the next start position next to the object. After several input sequences the robot has gathered enough IO data to generate a system model for pushing.

An action edge, contains a drive or push action. From the point of initialization an action edge starts collecting all necessary ingredients such that is can be executed (track a path). These ingredients are estimating path existence, being given a system model and searching for a path. To collect such ingredients the edge performs path estimation, is given a system model, performs motion or manipulation planning. Then finally, the edge is ready to be executed and send input toward the robot, an FSM of the action edge's status can be visualized in Figure 3.3.

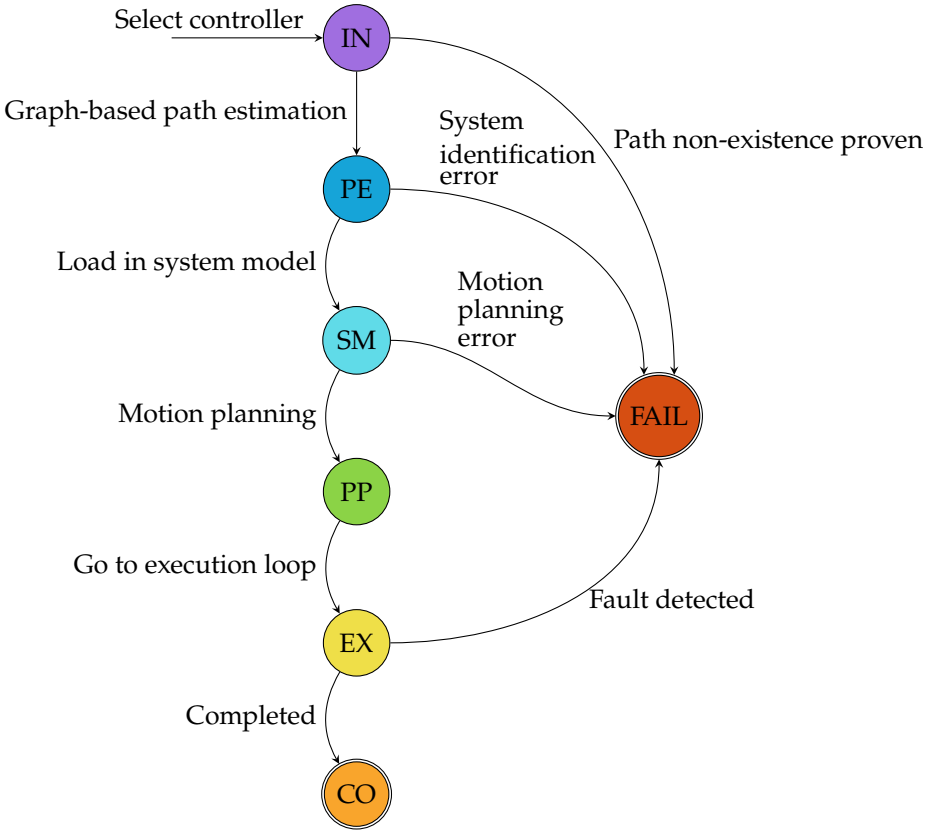


Figure 3.3: FSM displaying the status of an action edge

INITIALIZED (IN):	The edge is created with a source and target node which are present in the hgraph. A choice of controller is made.
PATH EXISTS (PE):	A graph-based search is performed to validate if the target configuration is reachable assuming that 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 is currently receiving 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 3.3 shows that many steps must successfully be completed before the edge can be executed. Before executing edges, edges must be initialized, which is where next section is dedicated to.

3.1.2. Hypothesis Algorithm

This section present and discuss the newly proposed halgorithm, this section will provide a mathematical description of the proposed halgorithm in the form of pseudo code. This section will discuss the search and execution loop that reside in the halgorithm and this section will finalize with 4 examples. First, let's look into the halgorithm pseudocode.

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 backward until the initial state is encountered [15].* A motivation for a backward search over a forward search is that, it might be the case that the branching factor is large when starting from the initial state. In such cases, it might be more efficient to use a backward search.

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

Table 3.2: Functions used by the Algorithm 1

Algorithm 2 Pseudocode for the proposed hypothesis algorithm.

```
for s ∈ S do
    while SubTaskNotFinished(s) do                                  ▷ Search Loop
        if G^hypothesis.isConnected(s.start, s.target) then
            if h.currentEdge.readyForExecution then
                while TargetNotReached(h.currentEdge) do            ▷ Execution Loop
                    if FaultDetected(h.currentEdge) then
                        HandleFault(h.currentEdge)
                        break
                    end if
                    h.currentEdge.steerTowardTarget(ob)
                    if TargetReached(h.currentEdge) then
                        if readyForExecution(h.currentEdge) then
                            h.incrementEdge
                        else
                            break
                        end if
                    end if
                end while
            else
                makeReady(h.currentEdge)
            end if
        else
            v_localtarget ← G^hypothesis.goBackward(v.target)
            v_localstart ← G^hypothesis.findCorrespondingNode(v_localtarget)
            G.connectWithEdge(v_localstart, v_localtarget)
        end if
    end while
end for
```

During a backward search, edges are added pointing toward the target node (or to nodes that point toward the target node). Trying to connect the robot node through a list of successive directed edges to a target node. If such a path has been found in the hgraph, a hypothesis has been found and the robot can start executing edges.

A flowchart of the halgorithm is presented in Figure 3.4. Compared to the mathematical description of the halgorithm the flowchart provides more detail, including an elaborate description for every block in the flowchart (see Table 3.3). The flowchart includes path estimation, planning and the behavior when failure occurs. A connection point to the kgraph and robot environment are included. The blocks in the flowchart indicate which action they take and where, such as the configuration space, the kgraph or the hgraph. With the flowchart is straightforward to see how the halgorithm connects to the status of edges, with the mathematical description of the halgorithm that is harder so see. Compared to the flowchart the mathematical description is a abstracted version, leaving many details out that are related to the robot in this thesis. An abstracted mathematical description is simpler and encompasses a broader field of robots. So could the mathematical description also be applied to another robot such as a movable robot with robot arm and gripper. The flowchart encompasses to many details to be applied after such an change in robot hardware.

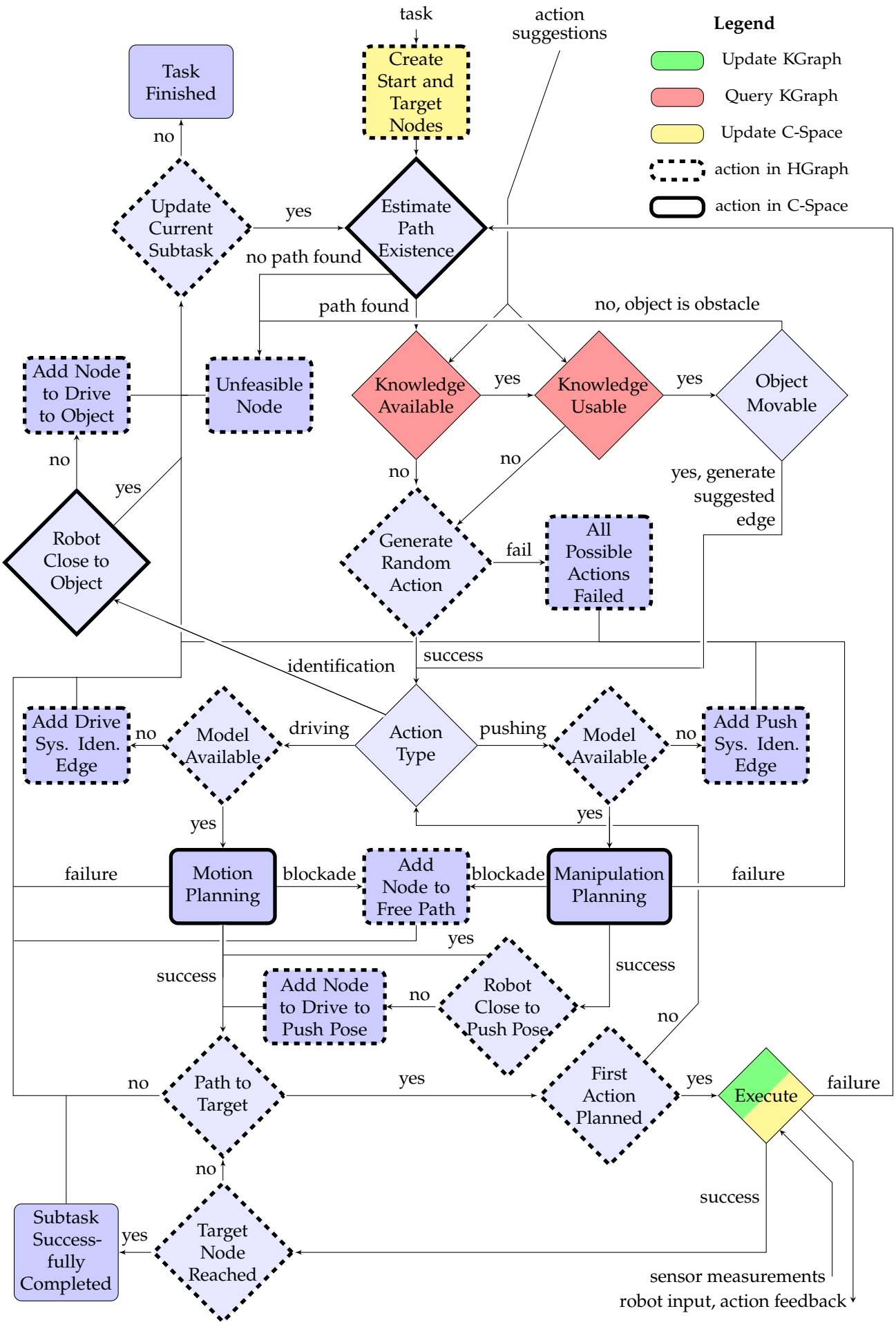


Figure 3.4: Flowchart displaying the hypothesis graph's workflow.

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

Table 3.3: Elaborate information on actions taken by blocks in Figure 3.4.

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 in put for the robot. In this theses, the sensor measurements are the configuration of objects in the environment. Recall that the perfect-sensor assumption, assumption 1.2.2 that makes access to the exact configuration of every object possible.

The Blacklist An failed edge is labeled as failed, then is could be regenerated again. Entering an infinite loop of being generated, failing, being labeled as failed and being generated again. Such behavior is undesirable and is prevented by the blacklist. The blacklist prevents certain edge parameterization to be generated to reach a specific node in the hgraph. When two nodes are connected with an action edge, the possible parameterizations are filtered. Thus any parameterization that is on the blacklist for this specific node (to which the action edge would point toward) cannot be created again for the lifetime of the hgraph. An example where the blacklist can be seen in action is Figure 3.11.

3.1.3. The Search and the Execution Loop

In Figure 3.4 two main loops can be identified, see Figure 3.5. These loops are the search loop, and the execution loop.

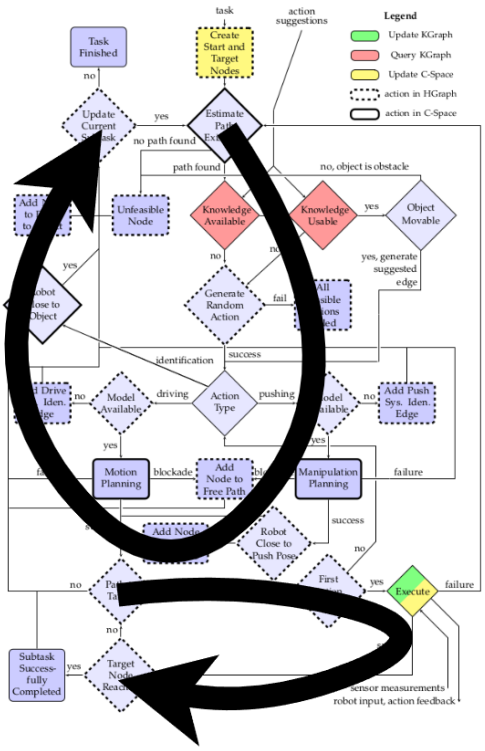


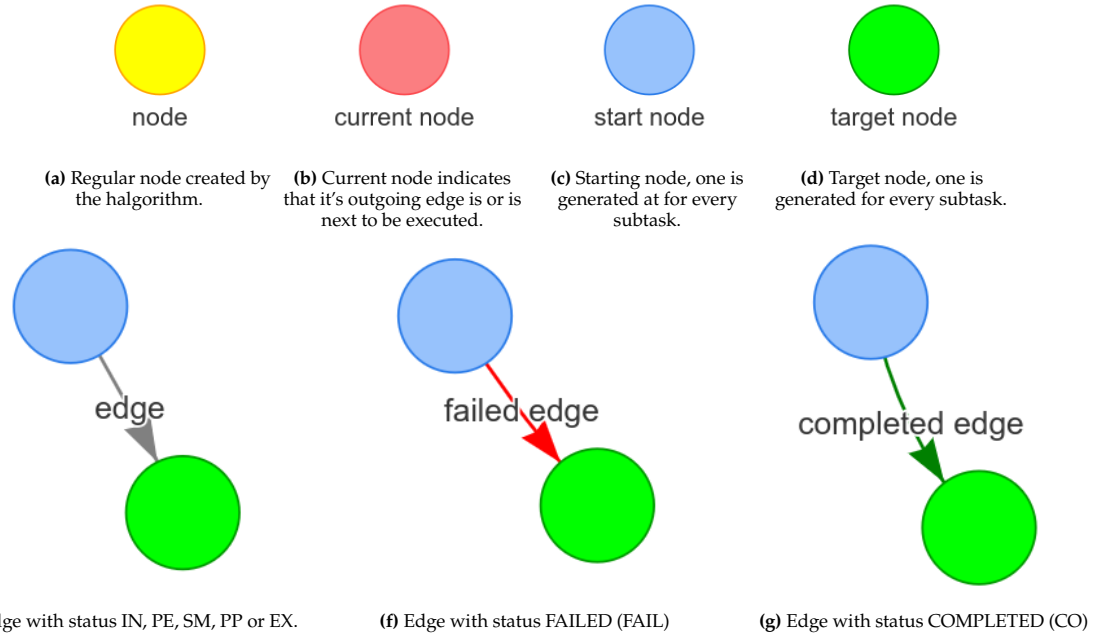
Figure 3.5: The search (above) and execution (below) loop.

Whilst the halgorithm resides in the search loop, hypotheses are formed. Forming a hypothesis generates nodes, edges, and progressing their status as described in Figures 3.2 and 3.3. 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, thus at any point in time, the halgorithm resides in a single block within Figure 3.4. 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. Assumption 1.2.2 guarantees that the robot environment does not change causing existing

hypotheses to be outdated.

3.1.4. Examples

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



(a) Regular node created by the halgorithm.

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

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

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

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

(f) Edge with status FAILED (FAIL)

(g) Edge with status COMPLETED (CO)

Figure 3.6: Legend for hgraph's nodes an edges

Driving and Pushing Four examples are presented, starting with a driving task in Figure 3.7, then a pushing task in Figure 3.9.

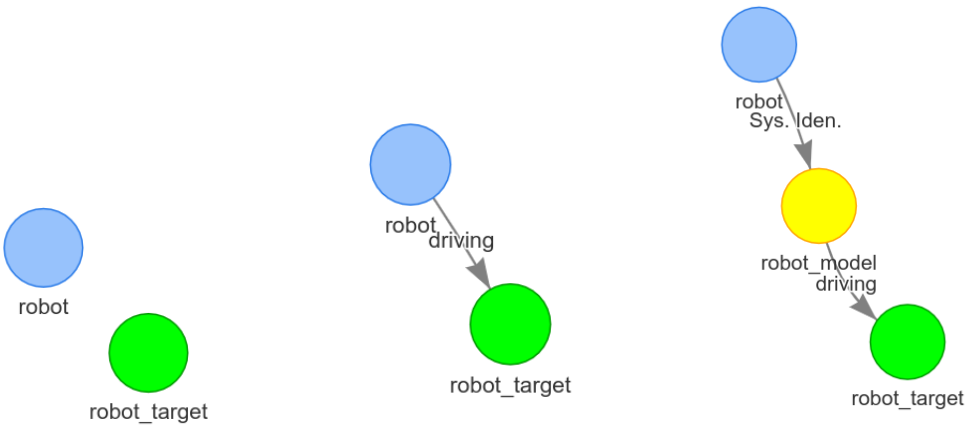


Figure 3.7: hgraph generated by the halgorithm to drive the robot to a target configuration

The robot does not have a system model of itself, thus first system identification must be performed before it can drive to the specified target configuration. The kgraph that will be discussed in Section 3.2.1 can suggest an action that includes a system model. In that case, system identification is not needed. The following figure displays successfully executing the hypothesis found in Figure 3.7.

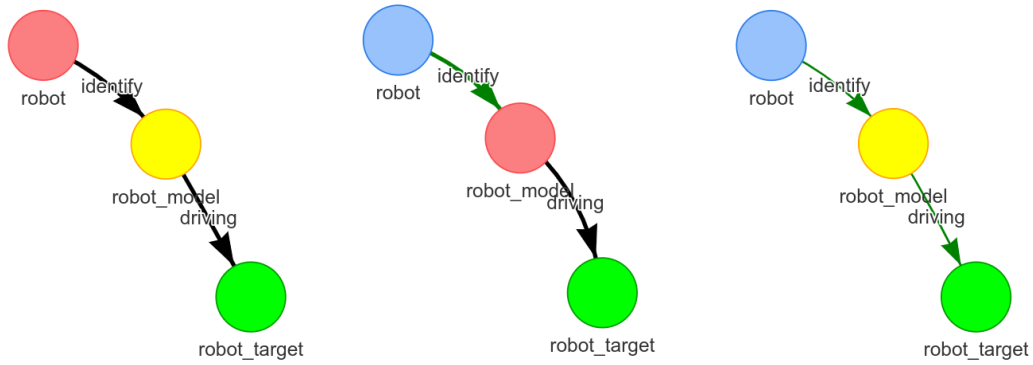


Figure 3.8: Executing the hypothesis found in Figure 3.7.

Upcoming figure will display the hypothesis generated to push an object to a target position. Both generating a hypothesis and executing the hypothesis are intertwined, this is because certain information should first be collected from the environment before the full hypothesis can be generated. An example is the *best_push_position* that can be found in Figures 3.9g to 3.9i. The *best_push_position* can be found after manipulation planning for the pushing edge is completed. For motion planning a system model is required, thus the corresponding system identification edge should be completed before manipulation planning can start, and than the *best_push_position* can be determined.

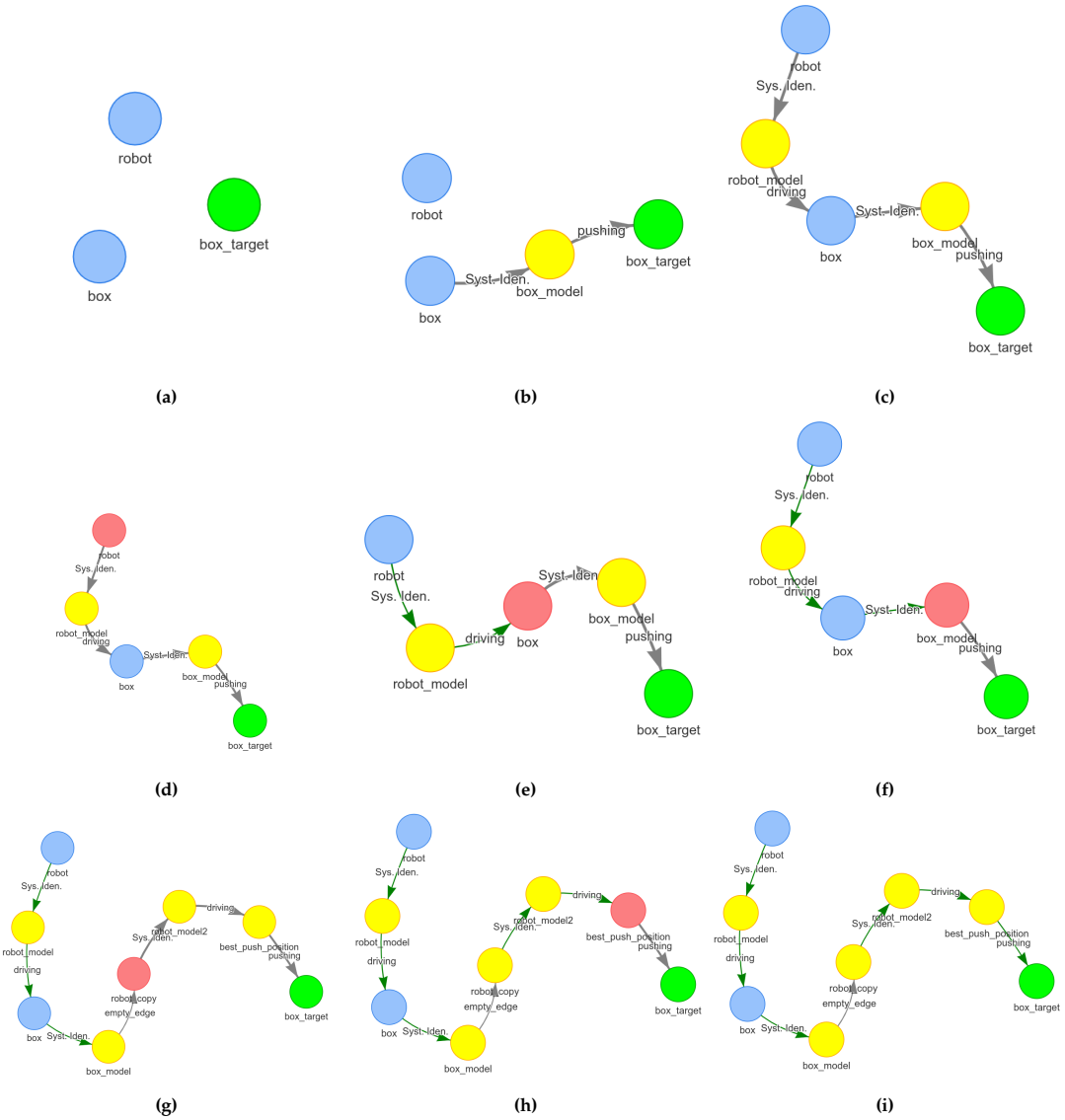


Figure 3.9: hgraph for pushing the green box to the target configuration

Especially in Figures 3.9b and 3.9c the backward search is clearly visible, the halgorithm searches from target node to the robot node. Figure 3.9 is extensive because every necessary steps is included whilst some could be skipped. First, identifying a system model for robot driving twice, if the system model created in edge Sys. Iden. pointing toward node robot_model is reused, then the edge Sys. Iden. pointing toward robot_model_1 would be unnecessary. Second, if system models would already be available for driving and pushing, no single system identification edge would be required. A *empty_edge* can be seen in Figures 3.9g to 3.9i, the empty_edge serves to connect a node to another node (box_model to robot_copy in Figure 3.9). The empty_edge can be traversed without execution, holds no controller, system model or status.

Encountering a Blocked Path During propagation of an action edge's status, motion or manipulation planning occurs. If an object is blocking the path, planning will detect it and the halgorithm tries to free the path. In the next 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 3.10.

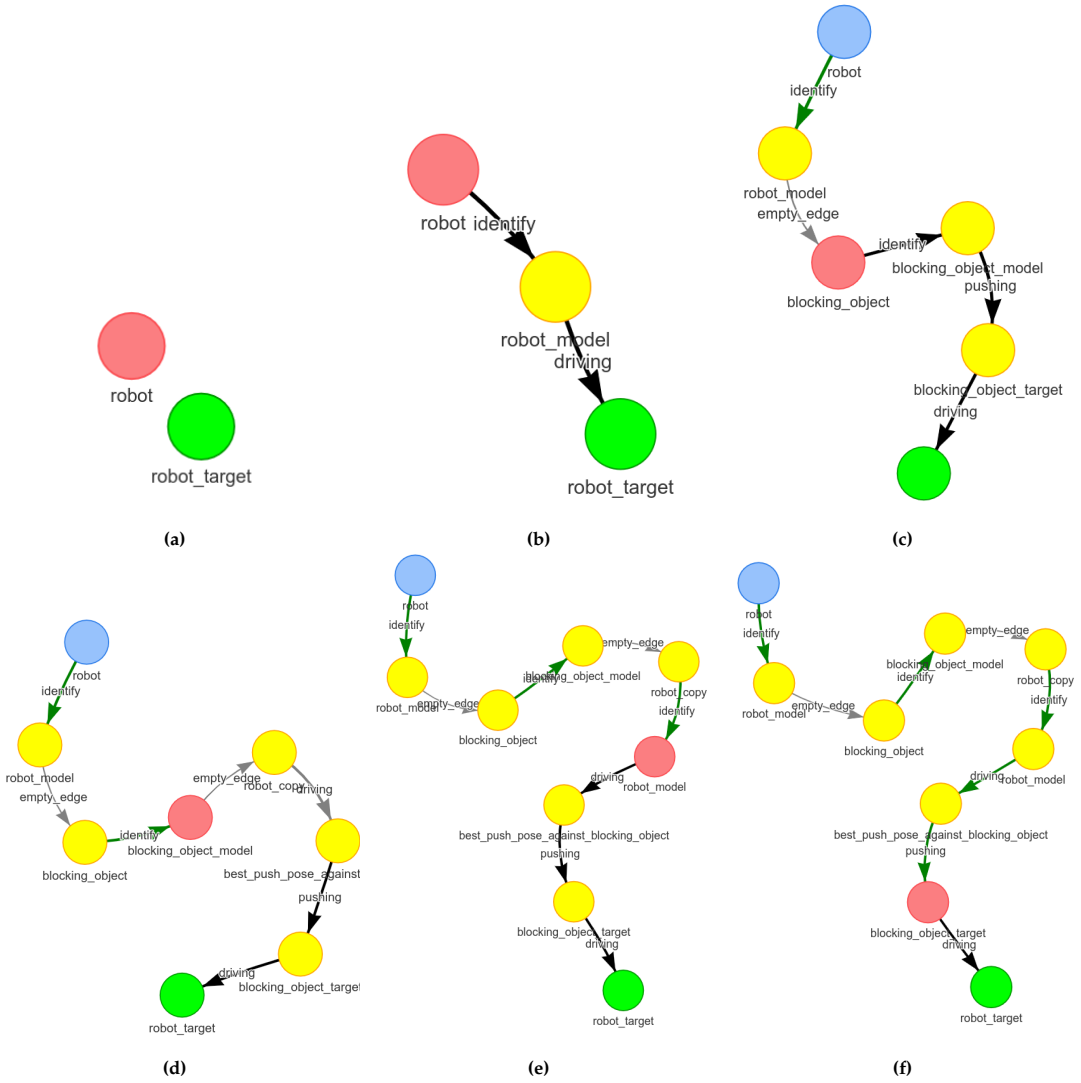


Figure 3.10: hgraph for driving to target configuration and encountering a blocked path

Encountering Failure In the last example, the first hypothesis fails to complete and the halgorithm tries to generate a new hypothesis that also fails to complete. Several faults and failures are modeled, the halgorithm response to faults and failure is the same. If during the propagation of an edge's status any kind of failure arises, the failed edge and corresponding edges are marked as failed. Equally during execution, if a fault is detected, the execution halts and the edge and corresponding edges are marked as "failed", the procedure can be seen in Figure 3.11.

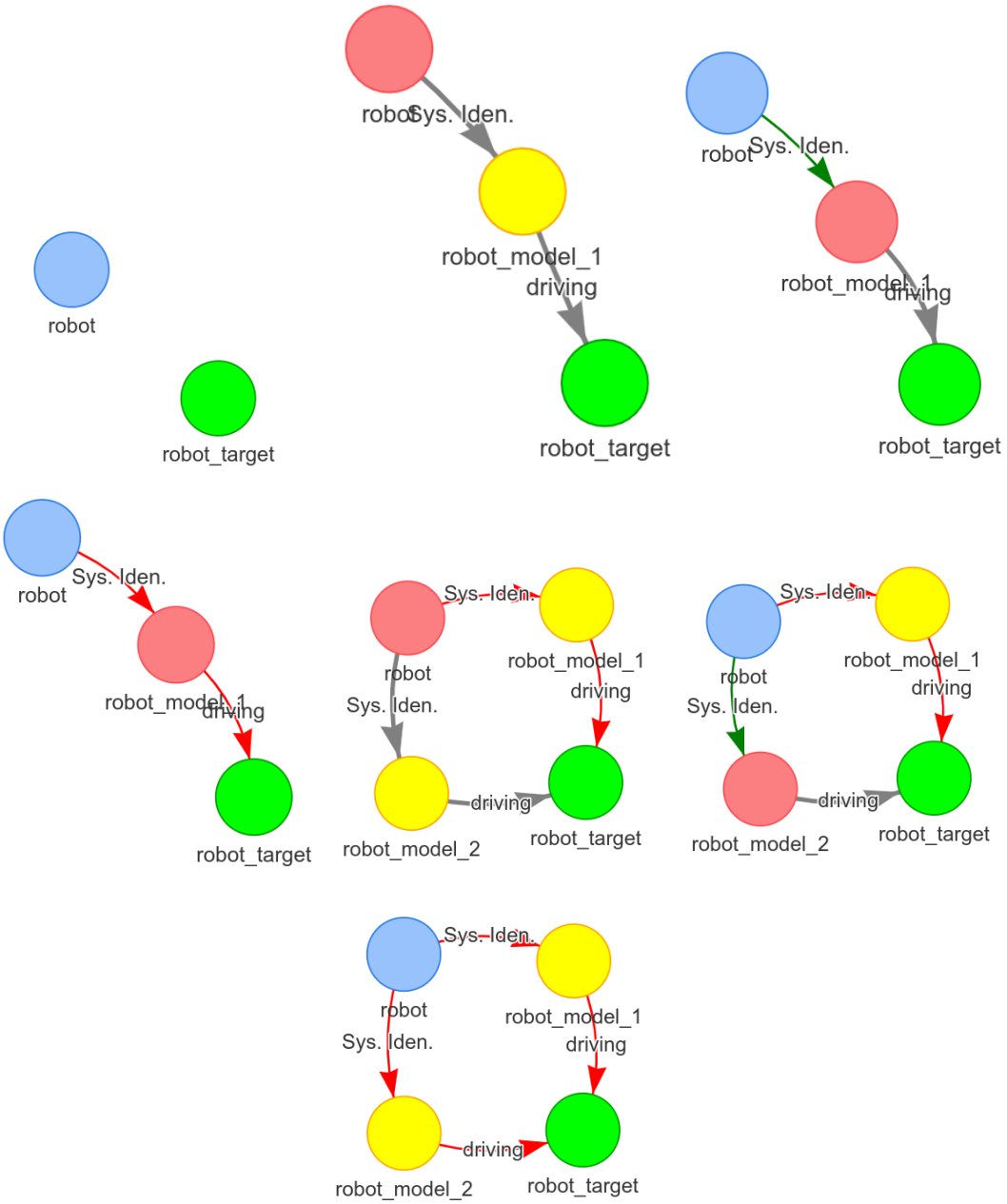


Figure 3.11: Executing two hypothesis, both failing to complete because a fault of failure emerged.

In Figure 3.11 only two parameterizations of drive controller and system model were available. Thus after two failed hypothesis the halgorithm concludes it cannot complete this task.

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 an fault or failure occurred.

The limited number of possible edge parameterization (every combination of a system identification method with a compatible control method) guarantees that the robot tries to complete a subtask, but concludes that it is unable to complete a subtask if all possible edges have failed.

Recall the 3 topics (learning object dynamics, the NAMO problem and nonprehensile push manipu-

lation) that this thesis proposes to combine. 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 halgorithm can push objects to target positions by first identifying a system model and then pushing the object toward its target position. The proposed algorithm learns to classify objects by updating objects class from unknown to movable or obstacle. The system model that system identification yields is however of short use, it is only given to the corresponding action edge. In the next section, the edges that are executed will be reviewed and stored in a knowledge base. The knowledge base will suggest a parameterization for an edge when faced with similar nodes to connect.

3.2. Knowledge Graph

The hgraph discussed in previous section has a lifetime that spans over a single task, learned system models are not stored for hgraph that are created for future tasks. 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, a metric that combines multiple metrics such as prediction error, tracking error and the success-fail ratio of a 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 methods build 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.

3.2.1. Definition

Before defining the kgraph, some definitions are defined on which the kgraph depends, such as the success factor, center and side nodes and edges.

Formally the **success factor**, $\alpha = \begin{cases} \left(0.5 - \epsilon_{\text{avg}}\right)^{1-\frac{N_{\text{success}}}{N_{\text{fails}}+N_{\text{success}}}} & \text{if } \epsilon_{\text{avg}} < 0.4 \\ 0.1^{1-\frac{N_{\text{success}}}{N_{\text{success}}+N_{\text{fails}}}} & \text{if } \epsilon_{\text{avg}} \geq 0.4 \end{cases}$

The success factor is discussed in upcoming paragraph.

An edge describes the parameterization that it holds, and how that parameterization compares to other edges in the kgraph.

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

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

An center node is linked to an object that is its main task.

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, OBJ_CLASS the classification of that object.

A side node represents nothing, it is a placeholder for the edge to point toward.

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

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

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

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

Now the kgraph is defined, lets investigate an example.

Success Factor The responsibility of the kgraph is to collect and suggest edge parameterizations. Estimating which parameterization would be the best candidate is an entire field of research on its own. So could action suggestions incorporate uncertainty, specific parameterizations can be suggested with the goal of collecting feedback on them. Or even suggest a parameterization based on the feedback on other parameterizations [13]. 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 weather there was a fault detected. Then action suggestions must be made based on collected feedback and an object that should change start configuration to a target configurations (connecting 2 nodes in the hgraph). A simple success factor thus already incorporates most of the available metrics.

success_factor =

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

The kgraph has 3 important functions. The *add_object* function adds object information to the kgraph, important for adding unmovable obstacles that cannot be manipulated by the robot. The *add_review* function is used when an edge successfully completed or failed, 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.

3.2.2. Example

An example kgraph can be visualized in Figure 3.12, the parameterization of edges is displayed and the object that the edge controls as image. For clarification, the connected left part with image of the point robot on the center node has 3 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 2 outgoing edges that describe robot pushing against the green box.

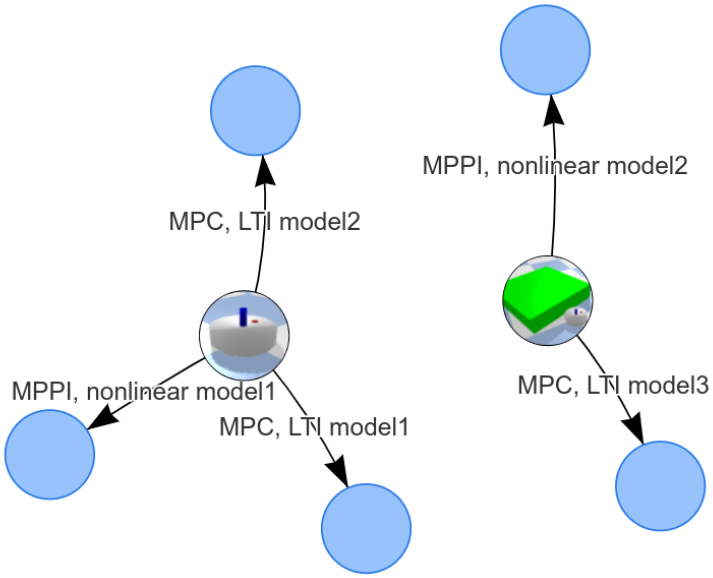


Figure 3.12: kgraph with 3 edges on robot driving, and 2 edges for pushing the green box.

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, making sure edges can point to a node. The blue nodes could fulfill a larger purpose, that is describing which actuators the edge can control. For example, a mobile robot with 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 can of the robot can be actuated. The controllers considered in this thesis control every actuator of the robot, resulting in the blue nodes serving such a small purpose.

3.2.3. Edge Metrics

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 successfully completed or failed. An indication is given on why certain metrics matter in Table 3.4.

Prediction Error (PE)	To better compare prediction errors the PE is summarized and average PE. The average PE is an indicator of 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 comes with more flaws since the average is mostly determined by outliers, some unfortunate outliers in the PE might for the largest part determine the average PE. The average PE will thus not be used because it is not robust enough.
ratio num_succesfully 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".
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. Planning time mainly serves to rank the slowest planners low, whilst not influencing the rank of fast and average planners.
run time	Also known as execution time would be a quality indicator if start and target states would be equal. Edges are recommended to solve similar tasks, where the path length between the start and target state is different. 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 of any use to rank edges.

Table 3.4: Edge metrics used to rank control methods from 'good' to 'bad'

The kgraph fulfills two goals. It stores information whether an object an be manipulated, and stores an ordered list from most successful control and system model combination to least successful per object. Information if objects can be manipulated prevent hypothesis from trying to push unmovable objects. A list of most successful control and system model combinations will suggest the best possible combination to fulfill the drive of pushing action.

4

Results

This chapter presents the test results that test and challenge the proposed method. The randomization tests, which additionally test the influence of the kgraph and test taken including and excluding the kgraph. The test results provide evidence that supports the effectiveness of the proposed method. This chapter starts by introducing ***method metrics*** *that indicate how a task has been executed by the proposed method in Section 4.1. A comparison is made with the state-of-the-art methods in Section 4.3.*

The Simulation Environment Testing in a simulation environment has been done using the URDF Gym Environment [26], a 100% python environment build upon the PyBullet library [7]. The code created during the thesis can be found on GitLab and GitHub. Experiments ran on standard TU Delft laptop: HP ZBook Studio x360 G5, running OS: Ubuntu 22.04.1 LTS x86_64, CPU: Intel i7-8750H (12) @ 4.100GHz, GPU: NVIDIA Quadro P1000 Mobile.The simulation environment provides many different

robots, 2 simple robots are selected to perform tests, they are displayed in Figure 1.1, and various objects are displayed in Figure 1.2.

4.1. Proposed Method Metrics

The results are measured in method metrics, not to be confused with the monitoring metrics in Section 2.4 or the edge metrics in Section 3.2.3. The results are interesting, but most interesting is the progression of the method metrics over time. As will be shown, the effect of learning can be measured by investigating and tracking the method metrics over time. Furthermore, the method metrics will be used to compare the proposed method to relative state-of-the-art papers. First, the method metrics are presented in Table 4.1 with corresponding argumentation on the relevance of the metric.

Total Average Prediction Error	The total average PE is created by averaging over every hypothesis' average PE in a hgraph. Since the PE is high when unexpected behaviour occurs, seeing the total average PE lower would indicate the robot encounters less unexpected behaviour, indicating the robot is learning.
The ratio between the number of hypotheses and the number of tasks	Expected is that whilst learning system models, the hypothesis created will be more effective. Thus the ratio between the total number of hypotheses and the total number of tasks is expected to lower with new knowledge.
The ratio between the number of successful and the number of total edges in kgraph	When the kgraph improves recommending a controller and system model, the ratio between successful edges and total edges is expected to increase because, with better recommendations, more edges will be completed.
task completion time = run time + planning time	If equal tasks are given multiple times, the total task completion time should drop pretty drastically. Multiple factors help to lower the task completion time, firstly system identification has to be performed only once, and there is no need to lose time on redoing system identification. Secondly, the hgraph is expected to improve generated hypothesis, or better said, the same mistake should not be made multiple times, resulting in fewer failing hypotheses and lowering task completion time.

Table 4.1: Proposed method metrics used to compare the proposed method with the state-of-the-art methods.

Three system models are used for testing. The goal of this thesis is not to find optimal control, or to model the environment very accurately. The goal is to select the best combination of controller and system model in the available set of controllers and system models. Only a short textual description of the available system models is provided below.

lti-drive-model	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 system input in x and y direction.
nonlinear-push-model-1	A nonlinear model describing the next object configuration in x and y direction and the orientation θ based on the current configurations of the robot, the object and on the robot inputs in x and y direction.
nonlinear-push-model-2	A nonlinear model describing the next object configuration in x and y, and the object 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 4.2: Available drive and push system models used for testing.

4.2. Randomization

In the randomized environment, the robot is tasked to push a number of objects to a target location. The previous 3 benchmark tests are designed around the proposed method, meaning that these tests highlight the strong points of the halgorithm but prevent the testing of the weak points of the proposed methods. Such bias can lead to false positives, which are undesired. The randomized environment provides a more unbiased result. The random environment does not deliver completely unbiased results because several parameters must be chosen to create and set up the random environment, which are.

The *size of the grid*	in x and y direction.
The *minimal and maximal size of objects*	A box will have sides with a length that lie in the specified range from minimal to maximal length. Cylinders will have a diameter and height that is within the specified range, additionally, cylinders are not higher than the radius of the cylinder to prevent cylinders from tipping over.
The *maximal weight*	which is uniformly distributed for the environment objects, minimal weight is set by default to 0.
The *number of unmovable and movable objects*	For every new object generated there is a 50% chance it becomes a box and 50% chance it becomes a cylinder, letting randomization determine the ratio between boxes and cylinders.
The *number of subtasks in a task*	Which must be greater or equal to the number of movable objects. The task has feasible targets that lie in free space, and the path estimator discussed in Section 2.3.1 is used to determine the target configuration for objects. By creating configuration space for an object the target configuration is chosen to be in free space, and then path estimation is used to determine if an object can reach the target configuration from its initial configuration.

If the task is completed, a *reshuffle* function reshuffles the objects in the environment, that is the object remain the same objects, but the objects receive a new initial position, and the subtask target configurations are renewed. The reshuffled environments can be seen in Figure 4.2.

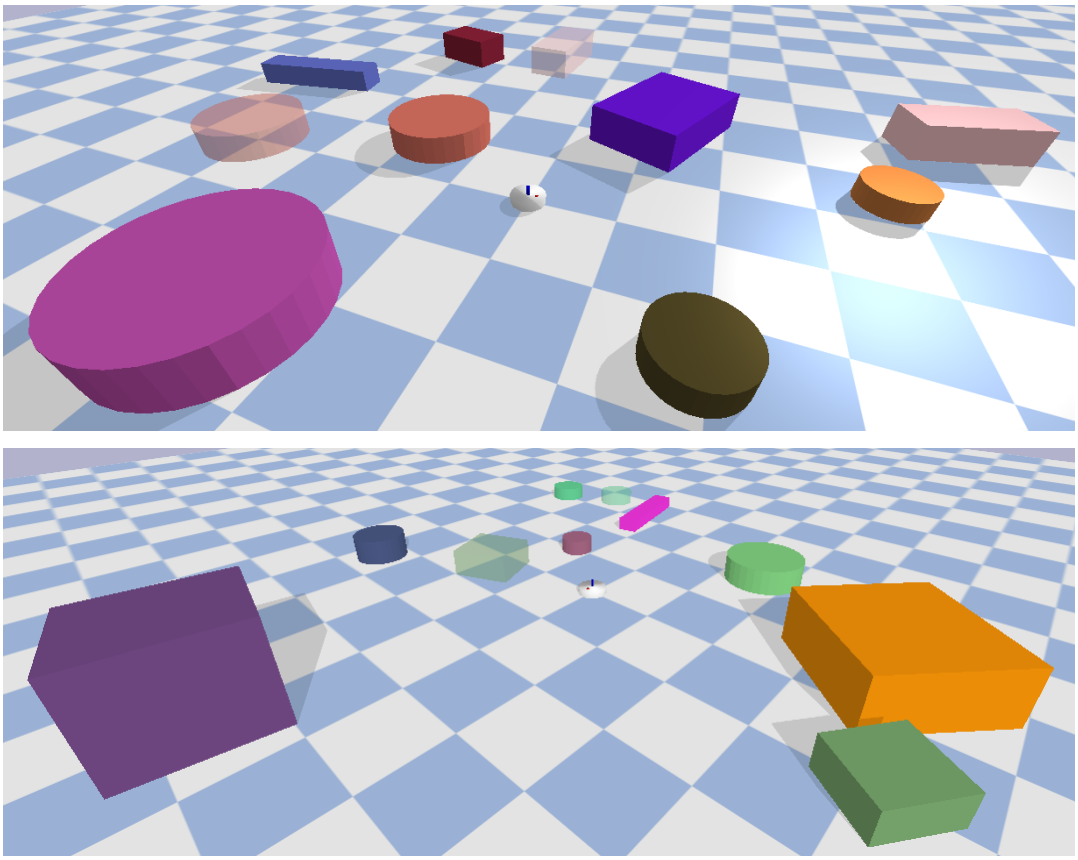


Figure 4.1: Two random environments with 2 target ghost configurations for the task containing 2 subtasks.

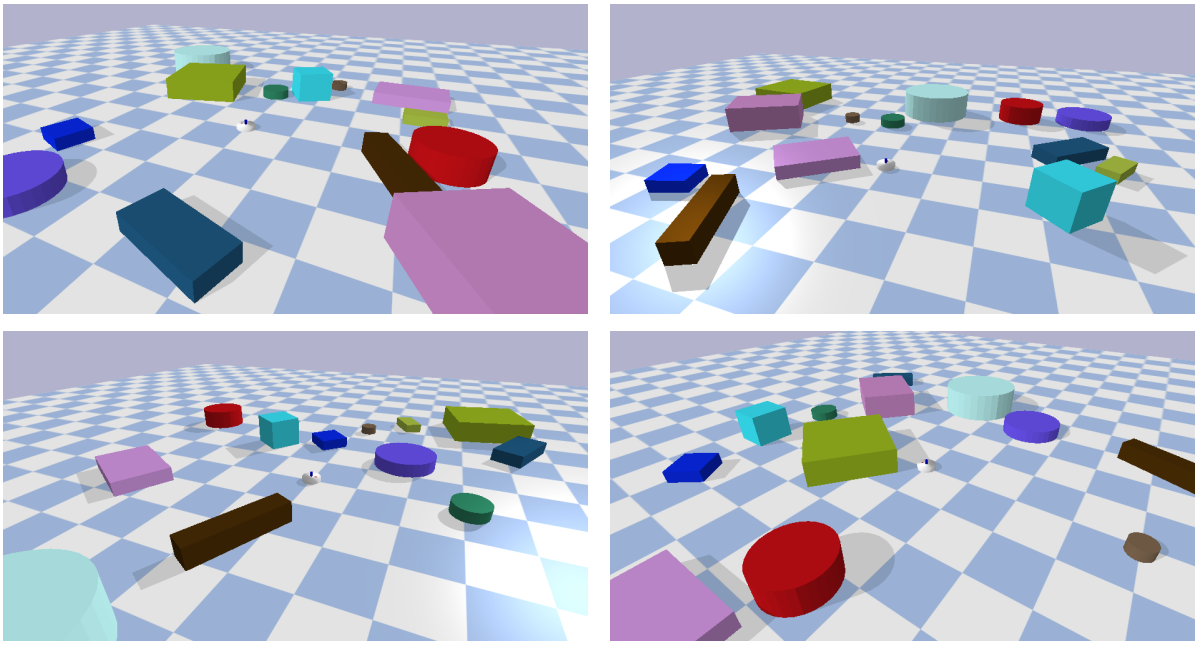


Figure 4.2: A random environment that is reshuffled 4 times. Target ghost configurations are not shown.

The halgorithm will be tested with both driving and pushing tasks. Starting with driving tasks for which the parameters are set to be.

grid size	$x = 12, \quad y = 12$
object size	$min\_length = 0.2, \quad max\_length = 2$
object weight	$max\_weight = 1000$
number of objects	$num\_unmovable\_obj = 3, \quad num\_movable\_obj = 5$
number of subtasks	$num\_subtasks = 3$

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

The following figure shows a driving task containing 3 subtasks for the robot, which translates into driving to three target configurations. The randomly generated task in the random environment is reshuffled and then solved 10 times. The sequence of tasks is performed with the kgraph suggestions.

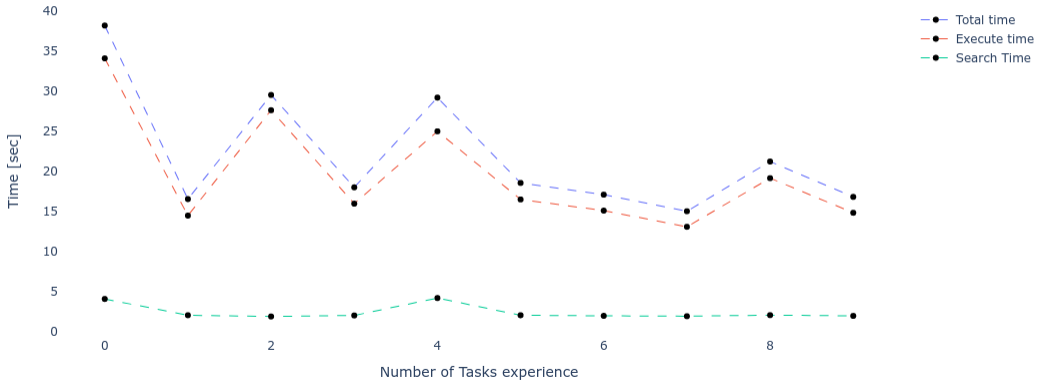


Figure 4.3: Robot tasked to drive toward 3 target configurations, the environment is reshuffled after completing a task.

What is not shown in the above figure is that the kgraph over times prefers the MPC controller over the MPPI controller (both using the *lti-model-1*) because of its lower average prediction error. The MPC controller is a bit faster compared to the MPPI controller thus the execution time decreases over time. The execution time is strongly correlated with the path length because driving to a target further away takes longer. It is thus only useful to investigate the trend that Figure 4.3 and not individual data points.

Then the same environments and tasks are solved 10 times without the kgraph suggestions. The randomly generated tasks and environments are repeatable 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.

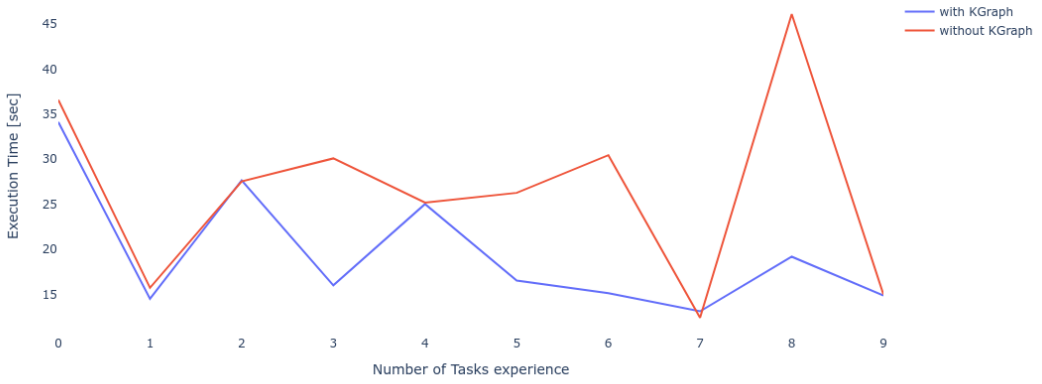


Figure 4.4: Execution times for driving toward target configurations with and without kgraph.

The figure above shows similar running times at the first few solved tasks. During these first tasks the kgraph is filled with edge feedback, but the kgraph has not gained enough feedback to suggest action parameterizations, this period can be referred to as the learning phase. After this learning phase, the kgraph suggests the MPC controller over the MPPI controller, because of its relatively higher *success factor*. The action suggestions improve overall execution time, note that selecting random controllers can result in equal execution time. In these cases, (4, 7 or 9 tasks in experience in figure 4.4) the randomly selected controllers all were MPC controllers.

Now the halgorithm is tasked with pushing a single object toward a target configuration. Except for the number of subtasks all parameters that determine the random environment will remain as previously set for the random driving task.

Figure 4.5: Average Prediction Error and Standard Deviation for action edges during a pushing task.

The MPPI controller with *nonlinear-push-model-1* has an average PE between 1.5 and 2 and can be seen in the figure above as the most left data point in the prediction error plot. The same controller with *nonlinear-push-model-2* has an average lower PE between 0.4 and 0.6 and can be seen in the following data points (1, 2, 3 and 4). The *success factor* is calculated based on the average prediction error so it is expected that the kgraph suggests the controller with the lower prediction error. It is expected that the PE goes down, which is exactly what Figure 4.5 shows. The halgorithm tests both edge parameterisations to conclude that the MPPI controller and *nonlinear-push-model-2* is the better of the two.

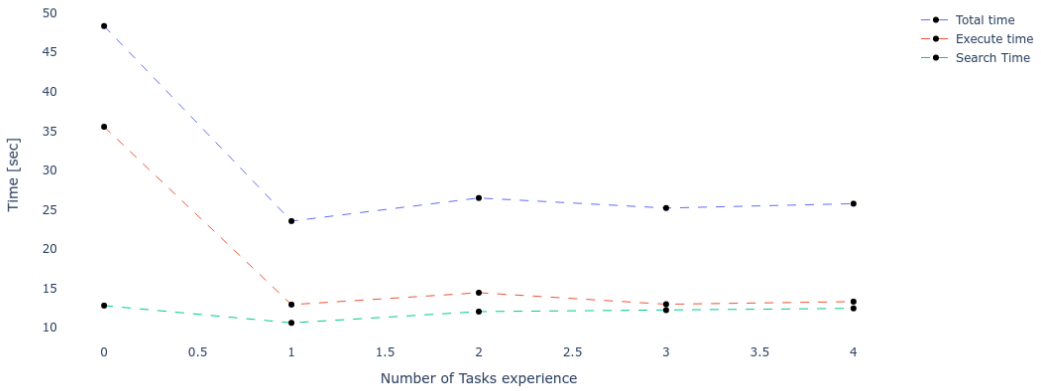


Figure 4.6: Planning times for pushing an object toward the target a configuration.

The MPPI controller with *nonlinear-push-model-2* does not only have a lower PE but is also faster compared to the MPPI controller combined with *nonlinear-push-model-1*. Execution time for driving toward the object and pushing that object to its target location takes around 15 seconds, with a search time to find that hypothesis being between 10 to 15 seconds. The result is that task execution time lowers, bear in mind that this result has to be taken with a grain of salt because the execution time is

correlated with the path length (how far the object is from the robot, and how far should the object be pushed). It is better to look at the figure below that displays the execution time of both solving the same task with help of the kgraph and without its help.

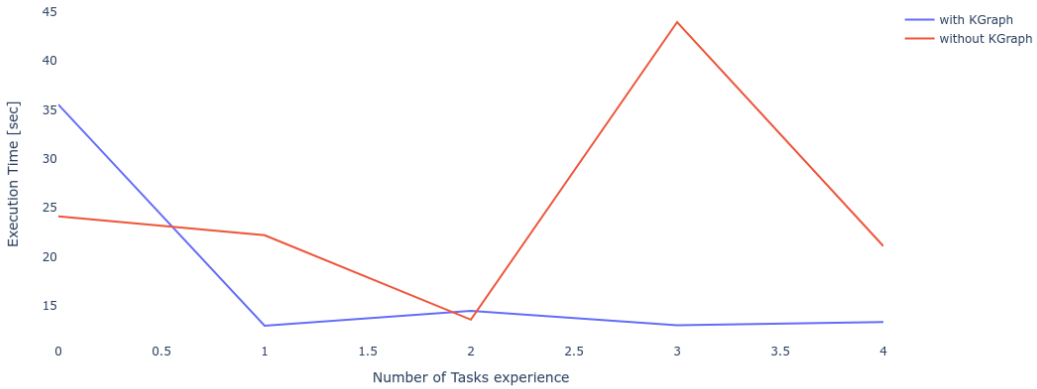


Figure 4.7: Execution times for pushing an object to target configurations with and without kgraph.

The kgraph suggestion work in favour of the execution time as can be seen in the Figure 4.7. Noticable is that a random selection by pure change selects the best choice of the controller as can be seen in 2 tasks of experience in the figure above.

4.3. Comparison with State-of-the-Art

In the introduction Table 1.1 was presented. A table that mainly shows which subset of the 3 main topics are included in which scientific paper. The 3 topics are learning system models, the NAMO problem and nonprehesile pushing. The table is presented again with the addition of the main testing method that is used by the state-of-the-art paper.

Author	Citation	Learns object dynamics	NAMO	Specify object target positions	Object Manipulation	test metric
Ellis et al.	[10]	✓	✓	✗	pushing	success rate
Sabbagh Novin et al.	[23]	✓	✗	✓	grasp-push grasp-pull	success rate, execution time prediction error, final position error
Scholz et al.	[24]	✓	✓	✗	graph-push grasp-pull	runtime, planning time, number of calls to planner number of calls to update model
Vega-Brown and Roy	[31]	✗	✓	✓	gripping	computation time
Wang et al.	[33]	✓	✓	✗	pushing	computation and execution time
Groote	proposed solution	✗/✓	✓	✓	pushing	see Table 4.1

Table 4.3: Overview of recent state-of-the-art papers that include a subset of the 3 topics (learning system models, NAMO, and nonprehensile pushing). The *grasp-push* and *grasp-pull* refer to prehensile push and pull manipulation, *gripped* refers to fully gripping and lifting objects for manipulation, *pushing* refers to nonprehensile push manipulation. The test metric indicates the testing method used by the paper, where the underlined metric is used to compare against the proposed method.

Comparing Success Rate Ellis et al. claims to push with a success rate of a 100% [10]. The test environment that Ellis et al. has used is very similar to the random pushing task discussed in Section 4.2. For both the random driving task and the random pushing task the success rate is 100%. Meaning that every driving or pushing subtask was successfully completed. It is even so that for the result of the random push task (displayed in Figures 4.5 to 4.7) the number of hypotheses is equal to the number of subtasks. Meaning that every subtask is completed by the first hypothesis generated. Both similar experiments from Ellis et al. and this thesis have a 100% success rate for similar , allowing us to conclude that the results are comparable.

Comparing Computation and Execution Time Three state-of-the-art papers that use computation and execution time as the testing metrics are compared to the proposed method. To accomplish a fair comparison the environments that have been used in the citations are rebuilt in the pybullet software. The results are then directly compared to conclude that the proposed method is as good as or even better in terms of computation time and execution time for similar tasks.

Wang et al. drives toward a target configuration that is blocked by a chair. The task is performed 3 times where the robot may remember its interactions with the chair from previous runs. The task is successfully completed all three times with a total time of 200 seconds for the first run to 180 seconds for the third run. Execution time increased from 70 to 85 seconds during these runs. Even with increasing execution times, the solution is found faster because the experience from previous runs could be used that lowered search time [33]. To compare against these results the environment is recreated in the pybullet software. The proposed method was run only once, during this run it classified a wall object as unmovable, and the pushable box object with dimensions equal to that of the chair was classified as movable and moved out of the way. Overall execution time was 45 seconds whilst computation time took 65 seconds. The proposed method successfully completed this similar task whilst having a lower

computation and execution time. This allows concluding that the proposed method improves upon the paper from Wang et al. with a decent margin.

Vega-Brown and Roy has a task that pushes two boxes into a goal region. It takes around 300 seconds to return a hypothesis [31]. Both pushing tasks can be compared with two separate pushing tasks from the random push environment. By doubling the time to push a single object to its target configuration, $2 \cdot 30 = 60$ seconds both tasks could be compared. However, the point of Vega-Brown and Roy paper is that a global minimal task is sought. The proposed method randomly selects a subtask and tries to complete it to then move to the next randomly selected subtask. A global minimum is thus not sought and both papers can therefore not be compared properly.

Sabbagh Novin et al. grasp-pushes and grasp-pulls a walker object to two new target configurations [23]. The task is mimicked by pushing a box object of equal dimensions to the same target configurations. Compared to Sabbagh Novin et al. average of 125 and 160 seconds to complete task 1 and task 2 the proposed method takes an average of only 27 and 33 seconds respectively (10 seconds computation time) to complete similar tasks. The decrease in total time allows to conclude that the proposed methods improve upon the method proposed by Sabbagh Novin et al.

Comparing Prediction Error Sabbagh Novin et al. displays the final position errors for a selection of objects which range from 0.05 to 0.6 meters. A success threshold is set to 0.1 meters, which determines that the object is at its target location [23]. The success threshold in this thesis is set to a staggering 0.9 meters, thus it is concluded an object has reached its target position, whilst it is still 0.9 meters from its target configurations. This thesis does not try to reach optimal control, it tries to select the best controller in the available set of controllers. If the same controller from Sabbagh Novin et al. would be available, then similar final position errors would be obtained. It can be concluded that the prediction error of the proposed method is worse compared to this state-of-the-art method. The reason is that the proposed method focuses on improving the control selection and action sequences over time, and not on lowering final prediction errors.

5
Conclusions

This chapter answers the main and sub research questions. By referring to previously drawn conclusions and insights evidence is provided that supports the answers to the main and sub research questions.

First, the two research subquestions are answered. Recall first research subquestion:

"Can the proposed method combine learning and planning for push en drive applications with a technique known as backward search?"

Chapter 3 is dedicated to presenting the proposed method. It has been shown that the proposed method that consists of the halgorithm, the hgraph and the kgraph work together can combine learning and planning with the technique of backward search. The proposed method searches in the joint-configuration space. A search directly in joint configuration space would result in the curse of dimensionality and is prevented by search only in a single mode of dynamics. Where a search is performed for a drive or push action. 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. The backward search can be seen in action in Figure 3.9. Newly gained environmental knowledge is stored in the kgraph, which firstly holds information on objects and if they can be manipulated, 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.

Recalling the second research subquestion, subquestion second research subquestion:

"Can the proposed method combine learning and planning for push en drive applications? Can the proposed method complete tasks, and how does it compare against the state-of-the-art?"

The proposed method was able to perform as well, better or worse compared to the state-of-the-art, depending on which metric is taken into account. The proposed method was tested against existing methods which only combine a subset of the 3 main topics that the proposed method combines. The results discussed in Chapter 4 are compared to the state-of-the-art papers in Section 4.3. Five state-of-the-art papers are used to compare against the proposed method. In the comparison success rate, planning time, execution time and several calls to the planner are used to compare. From the results, it can be concluded that the proposed method learns relatively fast (prediction error comparison with [33]). The proposed method can complete all of its assigned subtasks, giving it a comparable success rate to the state-of-the-art (success rate comparison with [10]). This thesis does not have small prediction errors or low final position errors, it can be seen that the prediction errors are worse compared to the state-of-the-art (final position errors comparison with [23]).

The proposed method improves upon the state-of-the-art exactly in its field of focus, improving task execution over time. It has been shown to learn faster compared to the state-of-the-art. Especially when

the same or similar task is given to solve multiple times. Execution and planning times are equal or improved compared to the state-of-the-art. Lastly, prediction errors and final position errors are mainly worse compared to the state-of-the-art, which can be improved by better-designed controllers and more accurate system models.

Recall the main research question.

"How do objects' system models learn by a nonprehensile manipulation robot during task execution improve global task planning?"

The main research question can be answered in multiple ways. Mostly because object system models and environmental knowledge improves task execution in multiple ways. A classification of an object that is movable or unmovable improves affordance. Probabilistic action sequences that involve pushing unmovable objects cannot occur when object classification is present. Some (not all) system constraints that are captured by the system model, find in Algorithm 1, line 33 the *reachablilityCheck* where the learned system model prevents the planner from creating an unfeasible path. Thus System models improve paths found by the motion or manipulation planner, this conclusion remains unproven since the evidence is not provided. Lastly, system models in combination with a controller (and edge parameterization) improve task execution in many metrics. For which evidence is proved for metrics prediction error and execution time in Chapter 4.

5.1. Future work

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

5.1.1. 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 method must be much more robust. The proposed method 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 method 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 method 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 [14] 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 method 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

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] 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).

[9] 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).

[10] 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.

[11] 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).

[12] 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).

[13] 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).

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

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

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

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

[18] 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).

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

[20] 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.

[21] 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).

[22] 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).

[23] 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).

[24] 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).

[25] 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).

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

[27] 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).

[28] 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).

[29] 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).

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

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

[32] 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.

[33] 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).

[34] 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).

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

6

Appendix

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

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 [19]. 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 [20]. A dynamical model can be presented in various forms, let's consider a familiar differential equation.

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

In which $x \in \mathbb{R}^n$ is the state, $u \in \mathbb{R}^m$ is the input, $y \in \mathbb{R}^p$ is the output, and $t \in \mathbb{R}$ is time. The initial condition specifies the value of the state x at $t = t_0$, and a solution to the differential equation for time greater than t_0, $t \in \mathbb{R}_{\geq 0}$ is sought. If little knowledge about the internal structure of a system is available, it may be convenient to take another approach where the state is suppressed, no internal structure about the system is known and the focus lies only on the manipulable inputs and measurable outputs. As shown in figure 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 [20]. 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.

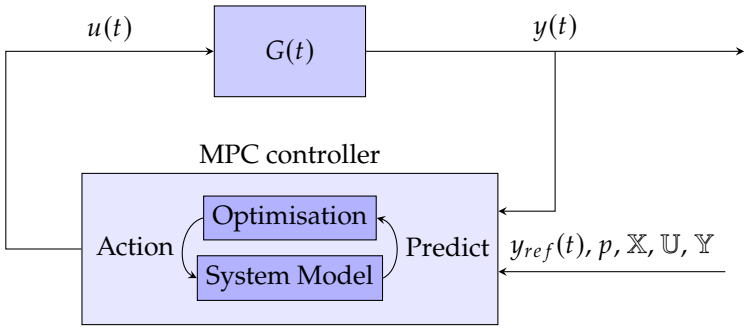


Figure B.1: System $G(t)$ with input $u(t)$, output $y(t)$ and MPC controller with input $y(t)$, reference signal $y_{ref}(t)$, parameterisation p and constraint sets $\mathbb{X}$, $\mathbb{U}$, $\mathbb{Y}$

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 $\mathbf{u}_N(k) = (u_k, u_{k+1}, \ldots, u_{k+N})$. The objective function takes the reference signal as an argument together with the initial state and the control input for the control horizon. The objective function then creates a weighted sum of some heuristic function. States and inputs resulting in outputs far from the reference signal are penalised more by the heuristic function than outputs closer to the reference signal. Because the objective function is a Lyapunov function, it has the property that, it has a global minimum for the optimal input $\mathbf{u}_N^*$. If the system output reaches the reference signal y_{ref}, x_{ref} then u_{ref} will be mapped to the output reference signal as such $y_{ref} = h(x_{ref}, u_{ref})$. As a result solving the minimisation problem displayed in equation (B.1) gives the optimal input which steers the system toward the output reference signal while at the same time respecting the constraints.

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

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 [20] visits extensively.

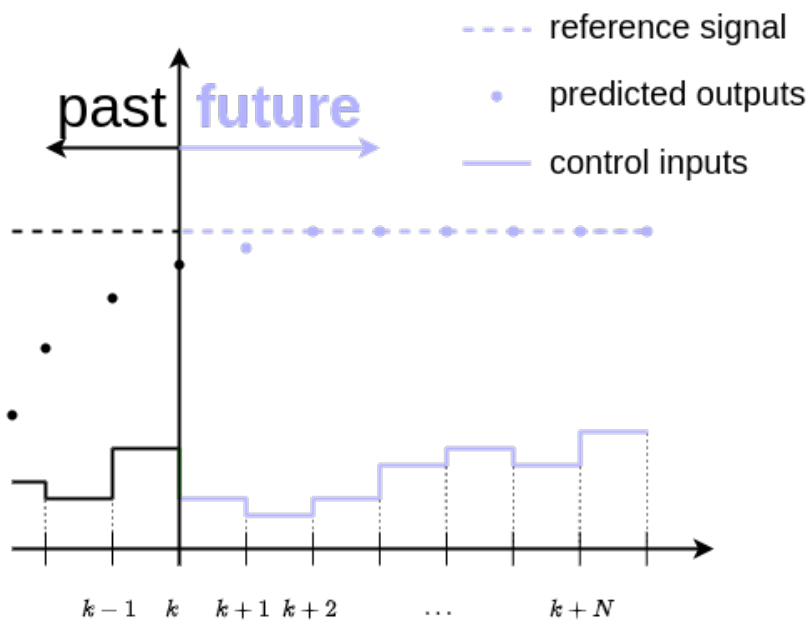


Figure B.2: A discrete MPC scheme tracking a constant reference signal. k indicates the discrete time step, N the control horizon

B.2. MPPI Control

Introduced by [34] 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, [17]. 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.

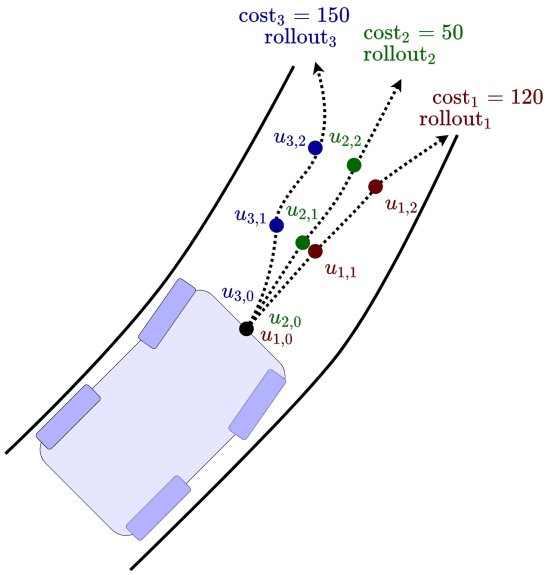


Figure B.3: MPPI controlled race car using a control horizon of 3 time steps, with 3 rollouts all having their respected inputs as $u_{i,j}$ where i is the rollout index and j indicates the time step [17].

Here 3 rollouts are displayed, The objective function is designed to keep the car driving on the center of the road by penalising rollouts which are further away from the center of the road relatively more. resulting in a high cost for rollout$_1$ and rollout$_3$ compared to rollout$_2$. As a result, the input send to the system as a weighted sum of the rollouts is mostly determined by rollout$_2$. The weighted sum determining the input is displayed in equation (B.2), from [17].

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

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