Faculty of Aerospace Engineering

Directed Increment Policy Search for Behavior Tree Task Performance Optimization

Crossing the Reality Gap

S.A. Leest

November 20, 2017

TU Delft Delft University of Technology

Challenge the future

Directed Increment Policy Search for Behavior Tree Task Performance Optimization

Crossing the Reality Gap

Master of Science Thesis

For obtaining the degree of Master of Science in Aerospace Engineering
at Delft University of Technology

S.A. Leest

November 20, 2017

Faculty of Aerospace Engineering · Delft University of Technology

Delft University of Technology

Delft University Of Technology
Department Of
Control and Simulation

The undersigned hereby certify that they have read and recommend to the Faculty of Aerospace Engineering for acceptance a thesis entitled **"Directed Increment Policy Search for Behavior Tree Task Performance Optimization"** by **S.A. Leest** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: November 20, 2017

Readers:

dr. Q.P. Chu

dr.ir. E. van Kampen

dr. G.C.H.E. de Croon

ir. K.Y.W. Scheper

dr.ir. E. Mooij

Abstract

Applications of Micro Air Vehicles (MAVs) require these small robotic platforms to operate autonomously. Autonomous behaviors are often designed in a simulation environment such that the simulated MAV can safely interact with the environment at reduced resource costs. Discrepancies between simulation and the real-world environment cause a reality gap. This reality gap results in a performance degradation once simulation-learned robotic behavior policies are transferred to the real-world robotic platform.

In this research we investigate the possibility to cross the reality gap by adapting a simulation-learned behavior policy using Reinforcement Learning (RL) methods. RL algorithms are a promising approach as these methods can find an optimal policy through trial-and-error exploration of the environment, however, often require many data points to converge. By adding a-priori information to the learning process, the number of data points may be reduced.

A-priori information may be added through understanding of the behavior by the designer. As such, this research employs a Behavior Tree (BT) to represent the robotic behavior because BTs facilitate human interpretation of the encoded behavior. A BT decomposes a general behavior in sub-behaviors and consists of a structure and a number of parameters. Assuming the discrepancies between simulation and the real-world are small, we attempt to reduce the influence of the reality gap by adapting the BT parameters only. This objective may be formulated in the following research question:

How can reinforcement learning reduce the influence of the reality gap experienced by a behavior policy represented in a behavior tree by adapting the behavior tree parameters to maintain performance in adapted simulation environments?

To investigate this research question this research adopts the task and behavior tree as described in [Scheper et al., 2016]. Here the DelFly Explorer flapping wing MAV is tasked to escape from a square room by flying through a square window after being initialized at a random location and orientation. The BT formulated for this task combines three sub-behaviors and consists of five parameters.

We consider the expressed behavior of a given BT as an emergence from the coupling between the BT structure, BT parameters and the environment. From this we hypothesize that the functional form of the behavior tree parameters has a single peak amongst a more flat surface.

We validate this hypothesis for the DelFly window fly-through task and demonstrate that the performance curve keeps a similar shape in adapted environment if the discrepancies are small. Using this insight, an efficient RL algorithm named DIPS is proposed.

Directed Increment Policy Search (DIPS) is a model-free episodic policy search algorithm. DIPS leverages the prior knowledge on the performance landscape and the interpretable properties of a BT by reducing the search space and directing the exploration process. This knowledge is added to the learning process through the direction array. This array specifies a direction for every parameter in which the respective policy parameter is incremented during exploration. The resulting hill-climb algorithm overcomes coupling effects by updating the policy parameters proportionally to the experienced performance increase of the exploration policies relative to the current policy.

The effectiveness of the DIPS algorithm is evaluated on three simulated reality gaps. These simulated reality gaps combine adaptations of the room dimensions and maximum elevator deflection of the DelFly. It is demonstrated that DIPS efficiently and effectively improves the performance of the behavior tree policy to an equal performance as obtained the baseline environment.

DIPS is a promising method to cross the reality gap because it facilitates on-line behavior adaptation and is therefore completely independent of simulation. Despite that DIPS is evaluated on one task and for a single BT only, we believe DIPS can generalize to other behavior representation methods and tasks due to the inherent coupling between behavior and environment. To verify this hypothesis, however, we recommend further research in the DIPS algorithm.

Contents

List of Figures

Acronyms

AI	Artificial Intelligence
ANN	Artificial Neural Network
BBR	Behavior-Based Robotics
BT	Behavior Tree
CHDS	Controlled Hybrid Dynamic System
DIPS	Directed Increment Policy Search
DT	Decision Tree
EL	Evolutionary Learning
FFNN	Feed-Forward Neural Network
FL	Fuzzy Logic
FSM	Finite-State Machine
HFSM	Hierarchical Finite-State Machine
HRL	Hierarchical Reinforcement Learning
MAV	Micro Air Vehicle
MDP	Markov Decision Process
ML	Machine Learning
POMDP	Partially-Observable Markov Decision Process
PS	Policy Search
RL	Reinforcement Learning
RRL	Relational Reinforcement Learning
TD	Temporal Difference
UAV	Unmanned Aerial Vehicle

List of Symbols

Greek Symbols

α	Learning rate
β	Policy function mapping states to actions
γ	Disccount factor
π	Reinforcement learning policy
χ	Behavioral representation structure
υ	Behavioral representation variables
θ	Function approximation argument

Roman Symbols

A	Set of all possible actions
a	Specific action out of the set of all possible actions
D	Increment direction array
G_t	The discounted reward at timestep t, also named the *return*
J_t	Cummulative reward at timestep t
$Q(s, a)$	State-action value of a state and action
S	Set of all possible states
s	Specific state out of the set of all possible states
$V(s)$	Value of a state

Chapter 1

Introduction

Many real-world applications require Micro Air Vehicles (MAVs) to operate autonomously (Floreano & Wood, 2015). To meet this requirement, MAVs have to determine how to act in a given state to make progress towards the task objective. This mapping from state to action is referred to as the *behavior policy*. Over time, the behavior policy leads to the expression of a specific *behavior*. With changing operating conditions the robot has to adapt its behavior in order to maintain a high level of task performance. Designing and optimizing autonomous behavior for MAVs poses a large challenge due to real-world uncertainties and limited on-board resources such as sensor quality and computing power.

A special case of behavior optimization originates from the simulation methods used to design and learn robotic behavior. Simulation allows the virtual MAV to safely interact with the environment over many iterations at a reduced resource costs. A simulation environment is, however, unable to capture the real-world complexities, resulting in a *reality gap*. As a result, the robotic behavior can become less efficient or even ineffective once transferred to a real-world environment. Arguably the influence of the reality gap is one of the largest challenges facing the implementation of real-world robotics as all simulation-based optimization methods suffer from this performance degradation Koos et al. (2013); Kober et al. (2013).

Three types of solution categories exist to mitigate the impact of the reality gap: 1) conduct the learning process in the real-world to avoid the reality gap; 2) implement simulator mechanisms to alleviate simulation inaccuracies; and 3) allow the behavior to be evaluated in the real-world during or after learning (Koos et al., 2013; Kober et al., 2013). The first category clearly does not profit from the advantages of simulation environments and therefore is not considered as a feasibly solution for obtaining complex real-world behavior (Gullapalli et al., 1994; Floreano & Mondada, 1998). The second category can either be achieved by increasing simulator fidelity or by removing the influence of known simulation inaccuracies. Increasing simulator fidelity, however, results in high computational requirements and assumes that all physical processes can be modeled exactly. Disguising simulator inaccuracies by adding noise or by abstracting the control inputs to higher-level measurements have been shown to result

in a reduced reality gap (Jakobi et al., 1995; Scheper & Croon, 2017). These methods, however, can deter the learning algorithm from high-performance policies which are sensitive to noise or limit the solution space of possible behaviors. The third approach, also named a robot-in-the-loop approach, allows the behavior learned in simulation to be evaluated in the task environment during or after learning. Most robot-in-the-loop approaches are focused on reducing the simulation inaccuracies. These inaccuracies can be reduced by learning both a behavior policy and by heuristically improving the simulator fidelity (Bongard & Lipson, 2004; Abbeel et al., 2007; Ross & Bagnell, 2012). An underlying assumption here is that the simulators become sufficiently accurate that the behavior can be transferred directly to the real-world environment.

Robot-in-the-loop approaches combine the advantages of both simulation and real-world based learning. Robot-in-the-loop approaches, however, learn behaviors which remain dependent on the simulation environment in which they are formulated. As a result, if the environment changes, the behavior is required to be adapted back in the simulation environment. This shortcoming can be mitigated by adapting the simulation-learned behavior in the real-world environment. Following this concept, a robot-in-the-loop algorithm which adapts a behavior policy in the real-world environment is required to:

1. **Improve the behavior policy performance for the real-world environment** such that the performance of the adapted behavior policy is increased with respect to the initial behavior directly after the transfer.
2. **Be robust to environmental conditions** because the conditions in the real-world environment are dynamic and unpredictable.
3. **Keep the number of evaluations at a minimum** for practical implementations, especially since it is recommended to run the algorithm multiple times to ensure the optimal behavior for the real-world environment is found.

Reinforcement Learning (RL) is a promising method to address these requirements. RL is a type of machine learning where a decision maker learns to formulate an optimal policy through trial-and-error exploration of the environment (R. S. Sutton & Barto, 1998). As a result, RL can be a promising robot-in-the-loop method as it facilitates on-line updates and therefore on-line adaptation. The third requirement for the robot-in-the-loop algorithm, however, gives rise to the following three challenges for RL algorithms:

1. **RL algorithms require many data points to converge** because the agent is required to find the optimal policy though exploration of the task environment.
2. **RL algorithms are dependent on reward function crafting** which convey a continuous indication of performance instead of a discrete or binary reward signal. For high-level behaviors this is a challenge because behavior should be evaluated based on reliability instead of the method through which the behavior performs the task.
3. **RL algorithms are sensitive to hyper-parameters** which govern the learning process; tuning of these parameters increases the number of evaluations.

The objective of this thesis is to formulate a RL algorithm which contributes to the aforementioned thee challenges and meets the three requirements to effectively cross the reality gap. The next section elaborates on the research approach to accomplish this research objective.

1-1 Research Approach

RL algorithms benefit significantly from the addition of prior knowledge to the learning problem. Prior knowledge can be added in the form of an initial policy, by task structuring and by guiding the exploration process (R. S. Sutton & Barto, 1998; Kober et al., 2013). Given the research objective, an initial policy is assumed to be available which captures important elements of the behavior, however, has a reduced performance as it operates in a similar but adapted environment. Task structuring implies decomposing the task into different components. For behaviors this implies decomposing a general behavior in sub-behaviors. This effectively reduces the search space, however, can result in optimization of the individual sub-behaviors instead of the general behavior. Finally, adding knowledge to guide the exploration process results in more efficient exploration. On a behavior level, this requires the designer to understand the behavior and how this behavior should be updated to increase performance in the real-world environment. In term this requires an insight in how behaviors are expressed through the method used to encode the behavior policy. These methods to encode a behavior policy are referred to as a *behavior representation* (Arkin, 1998).

A behavior representation consists of a structure and a set of parameters. Behavior representations are often abstract and difficult to interpret structures such as Artificial Neural Networks (ANNs) and Finite-State Machines (FSMs) (Arkin, 1998). ANNs scale well to large state spaces but are difficult to adjust to a new environment due to their inter-dependencies and black-box nature. FSMs are simple to interpret and adjust, however quickly lose these advantages when applied to large state-actions spaces due to the *state explosion* (Harel, 1987; Valmari, 1998). Recent work in Behavior Trees (BTs) shows a promising alternative structure to represent robotic behavior which allows for easy adjustment (Dromey, 2003; Scheper et al., 2016).

BTs efficiently encode robotic behavior in a tree-like structure which hierarchically decomposes a main task in smaller sub-task, facilitating better understanding and adaptability of the behavior. [Scheper et al., 2016] demonstrates the first application of a BT to represent the behavior of a real-world MAV. In their research a BT is evolved in simulation to control the DelFly, a flapping wing MAV, on a window fly-through task. Afterward the BT parameters were manually adjusted in the real-world to improve task performance.

This research continues on the work of [Scheper et al., 2016] and contributes by formulating a RL algorithm which adapts behavior tree parameters in an automated fashion. Similar as in [Scheper et al., 2016], this research assumes that the discrepancies between simulation and the real-world are small enough such that reality gap can be crossed by only updating the numerical BT parameters. The research approach is visualized in Figure 1-1.

Figure 1-1 displays the high-level approach of employing RL to reduce the influence of the reality gap by segmenting the problem in three domains. The first domain consist of the task and environment. The second domain includes the behavior representation domain, in this case a BT, which directly interacts with the first domain through on-board measurements

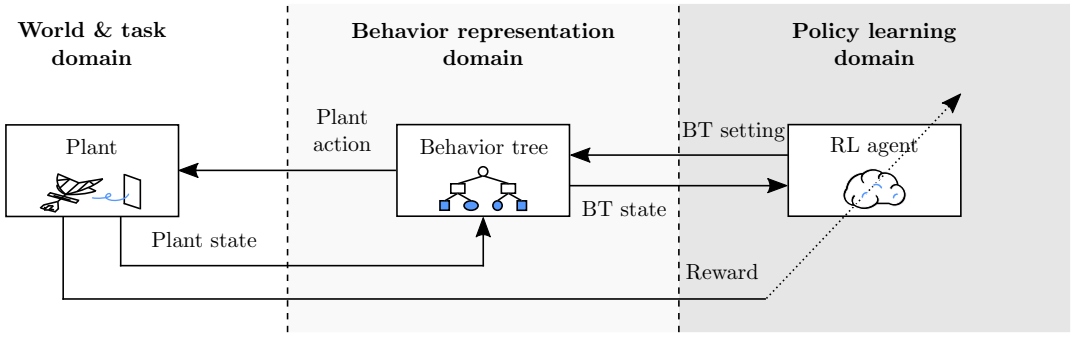


Figure 1-1: Block diagram of proposed approach to cross the reality gap.

of the DelFly. The third domain contains the RL agent. This agent indirectly interacts with the environment through the BT and optimizes the BT parameters to maximize the task performance in the first domain. Following the design philosophy of the DelFly, the RL algorithm only has access to on-board measurements and the current BT parameters to be independent off external measurements, allowing for a wider field of applications (Croon et al., 2016).

To limit the scope, this research will focus on the development of the RL algorithm and test the performance in simulation only. A reality gap may be simulated by adapting the environment used by [Scheper et al., 2016] to evolve the BT for the DelFly window fly-through task.

1-2 Research Questions

The objective of this research is formulated in the following research question:

How can reinforcement learning reduce the influence of the reality gap experienced by a behavior policy represented in a behavior tree by adapting the behavior tree parameters to maintain performance in adapted simulation environments?

To answer this question, the research is split into five sub-questions; these sub-questions collectively contribute to the research question:

1. How can reinforcement learning learn and optimize behavior policies for real-world robotics?
2. What are behavior trees and how can these models be used to represent robotic behavior?
3. How can reinforcement learning be applied to find the optimal numerical behavior tree parameters?
4. Which a-priori knowledge can be added to the learning process to reduce the number of evaluations?
5. How robust is the reinforcement learning algorithm to environmental adaptations and hyper-parameter settings?

1-3 Report Outline

This report consists of four parts. Part I presents the main results in the form of an article and derives the proposed RL algorithm for the research objective. This part includes, but does not explicitly mention, the answers to the five sub-questions. Part II contains a preliminary research conducted to answer the first three sub-questions. The preliminary research consists of a literature study on policy learning methods and behavior representations in Chapters 2 and 3, respectively. A preliminary analysis on how RL can be applied to learn BT parameters is conducted in Chapter 4. Part III includes an extended analysis of the RL algorithm derived in Part I. Finally Part IV discusses the final results, concludes on the research by answering the individual sub-questions and provides recommendations for further research.

Part I contains the main results of this research. Readers which are unfamiliar with RL and behavior policies, however, are advised to start reading at Part II to get acquainted with these concepts before starting with Part I.

Part I

Article

Directed Increment Policy Search for Behavior Tree Task Performance Optimization

S.A. Leest*

Delft University of Technology, Delft, 2629 HS, The Netherlands

Robotic behavior policies learned in simulation suffer from a performance degradation once transferred to a real-world robotic platform. This performance degradation originates from discrepancies between the real-world and simulation environment, referred to as the reality gap. To cross the reality gap, this papers presents a simple reinforcement learning algorithm named Directed Increment Policy Search (DIPS). DIPS is a form of episodic model-free policy search which leverages the interpretable structure and the coupling of the Behavior Tree (BT) parameters to reduce the number of required real-world evaluations. Additionally, DIPS does not require a form of reward function crafting and is robust to hyper-parameter settings. DIPS is evaluated on a simulated model of the DelFly Explorer which is tasked to perform a window fly-through maneuver. It is demonstrated that DIPS efficiently and effectively improves the BT behavior policy performance for three simulated environments with increasingly large reality gaps. We believe DIPS can generalize to other behavior representation methods and tasks due to the inherent coupling between behavior and environment experienced by embodied robots.

Nomenclature

Greek symbols:

δ	Rudder deflection, [rad]
ϵ	Relative initial increment step size
ε	Increment step size
η	Increment increase factor
θ	Policy parameters
π_θ	Parametrized policy
σ	Window response value
μ_D	Mean disparity

Latin symbols:

D	Increment direction array
P	Performance
r	Reward
t	Policy trial or evaluation
T_{min}	Performance increase threshold

I. Introduction

Many real-world applications require Micro Air Vehicles (MAVs) to operate autonomously.[1] To meet this requirement, MAVs have to determine how to act in a given state to make progress towards the task objective. This mapping from state to action is referred to as the *behavior policy*. Over time, the behavior policy leads to the expression of a specific *behavior*. With changing operating conditions the robot has to adapt its behavior in order to maintain a high level of task performance. Designing and optimizing autonomous behavior for MAVs poses a large challenge due to real-world uncertainties and limited on-board resources such as sensor quality and computing power.

A special case of behavior optimization originates from the simulation methods used to design and learn robotic behavior. Simulation allows the virtual MAV to safely interact with the environment over many iterations at a reduced resource costs. A simulation environment is, however, unable to capture the real-world complexities, resulting in a *reality gap*. As a result, the robotic behavior can become less efficient or

*MSc. Student, Department of Control & Simulation, Faculty of Aerospace Engineering, S.A.Leest@student.tudelft.nl

even ineffective once transferred to a real-world environment. Arguably the reality gap is one of the largest challenges facing the implementation of real-world robotics as all simulation-based optimization methods suffer from this performance degradation.[6,8]

Three types of solution categories exist to mitigate the impact of the reality gap: 1) conduct the learning process in the real-world to avoid the reality gap; 2) implement simulator mechanisms to alleviate simulation inaccuracies; and 3) allow the behavior to be evaluated in the real-world during or after learning.[6,8] The first category clearly does not profit from the advantages of simulation environments and therefore is not considered as a feasibly solution for obtaining complex real-world behavior.[14,15] The second category can either be achieved by increasing simulator fidelity or by removing the influence of known simulation inaccuracies. Increasing simulator fidelity, however, results in high computational requirements and assumes that all physical processes can be modeled exactly. Disguising simulator inaccuracies by adding noise or by abstracting the control inputs to higher-level measurements have been shown to result in a reduced reality gap.[11,16] These methods, however, can deter the learning algorithm from high-performance policies which are sensitive to noise or limit the solution space of possible behaviors. The third approach, also named a robot-in-the-loop approach, allows the behavior learned in simulation to be evaluated in the task environment during or after learning. Most robot-in-the-loop approaches are focused on reducing the simulation inaccuracies. These inaccuracies can be reduced by both learning a behavior policy in simulation and by heuristically improving the simulator fidelity through real-world trials.[17,19,21] An underlying assumption here is that the simulators become sufficiently accurate that the behavior can be transferred directly to the real-world environment.

Robot-in-the-loop approaches combine the advantages of both simulation and real-world based learning. Robot-in-the-loop approaches, however, learn behaviors which remain dependent on the simulation environment in which they are formulated. As a result, if the environment changes, the behavior is required to be adapted back in the simulation environment. This shortcoming can be mitigated by adapting the simulation-learned behavior on-line in the real-world environment. This shows parallels with Reinforcement Learning (RL). RL is a type of machine learning where a decision maker learns to select actions by interacting with an environment.[5] As a result, RL can be a promising robot-in-the-loop method as it requires no a-priori knowledge of the environment and facilitates on-line updates.

A RL algorithm which reduces the impact of the reality gap should meet three design criteria: the algorithm is required to increase policy performance, be robust to different environments and should keep the number of evaluations at a minimum. The last requirement gives rise to three challenges for RL algorithms.[6,7] First, RL algorithms often require many data points to converge. Second, RL algorithm are often developed for lower-level behavior which evaluate the actions through crafted continuous reward-functions. For high-level behavior this is more difficult because behavior should be evaluated based on the reliability. Finally, RL algorithms require careful tuning of many hyper-parameter settings.

The research described in this paper contributes to crossing the reality gap by introducing a simple RL algorithm named Directed Increment Policy Search (DIPS). DIPS is a Policy Search (PS) algorithm which leverages the initial policy, coupling of behavior parameters and designers understanding of the expressed behavior to improve on the before-mentioned challenges. To exploit the properties of DIPS, this research employs a Behavior Tree (BT) to represent robotic behavior as this structure facilitates human understanding of the encoded behavior.[12,13]

[Scheper et al., 2016] demonstrates the first application of a BT to represent the behavior of a real-world MAV. BTs efficiently encode robotic behavior in a tree-like structure which hierarchically decomposes a main task in smaller sub-tasks, facilitating better understanding and adaptability of the behavior. In their research a BT is evolved in simulation to control the DelFly -a flapping wing MAV– on a window fly-through task.[3] Afterwards, the BT parameters were manually adjusted in the real-world to improve task performance. This research continues on the work of [Scheper et al., 2016] by automating the behavior policy adjustment process using DIPS. Similar as [Scheper et al., 2016], this research assumes that the discrepancies between simulation and the real-world are small enough such that reality gap can be crossed by only updating the numerical parameters of a BT. Despite that DIPS is evaluated on this specific simulated experiment only, we believe there are strong indications that DIPS generalizes to other tasks or behavior representation methods.

Figure 1 displays the high-level approach of employing DIPS to reduce the influence of the reality gap by segmenting the problem in three domains. The first domain consist of the task and environment, the plant.

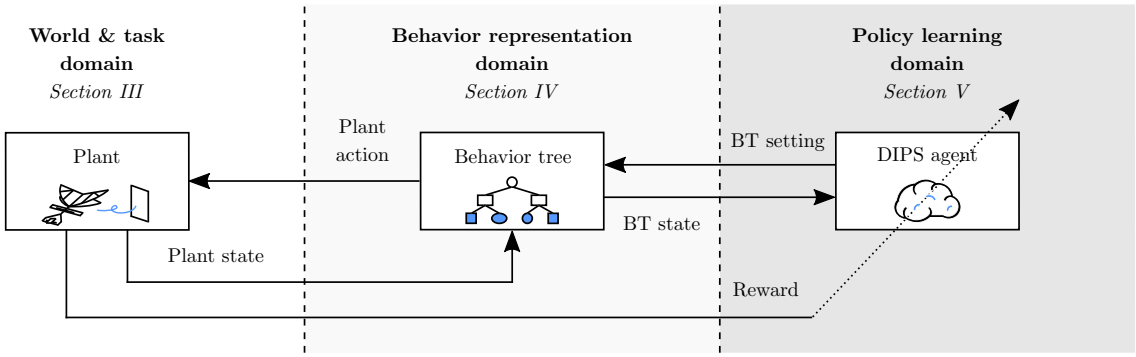


Figure 1: Block diagram of proposed approach to cross the reality gap.

The second domain includes the behavior representation domain, in this case a BT, which directly interacts with the first domain through on-board measurements of the DelFly. The third domain contains the RL agent. This agent indirectly interacts with the real-world environment through the BT and optimizes the BT parameters to maximize the task performance in the first domain. Following the design philosophy of the DelFly, DIPS only has access to on-board measurements and the current BT parameters to be independent off external measurements, allowing for a wider field of applications.[3]

This paper starts with a background on RL and BTs in Section II. Thereafter, Section III specifies the window fly-through task and the used simulation models. Section IV analyzes the BT and investigates the presence of the parameters coupling used to derive the DIPS algorithm in Section V. The experimental set-up including the simulated reality gaps are described in Section VI. DIPS is applied to the simulated reality gaps and the results are presented and analyzed in Section VII. These results and the DIPS algorithm are further discussed in Section VIII and Section IX draws conclusions from the previous sections. Finally, Section X provides recommendations for future research on DIPS.

II. Background

A. Policy Search Reinforcement Learning for Robotics

In RL an agent learns a control policy by interacting with its environment, receiving feedback in the form of a numerical reward after every selected action.[5] Over time, the agent learns which action to select based on the received feedback. Coarsely, RL algorithms can be segmented in *value-based* and *policy-search* methods. Value-based methods estimate the long-term expected reward –the value– for all states and actions. PS Methods directly search for an optimal policy. Given a parametrized policy $a = \pi_\theta(s)$, the goal is to find the combination of policy parameters θ such that action a is optimal in state s.[6,7] The following subsections elaborate on the advantages and taxonomy of PS methods.

1. *Advantages of Policy Search Algorithms*

PS methods have several advantages over value-based methods for this research. First, PS methods allow for a natural integration of a-priori knowledge to the search domain, in this case an initial policy. Second, optimal policies often have fewer variables than value function (approximation) methods, reducing the number of real-word evaluations. Finally, when transferring from simulation to a real-world environment, PS methods do not require any pre-training in simulation to reduce the number of real-world evaluations. Subsequently, we consider PS algorithms the better category ofRL methods for this research.

2. *Taxonomy of Policy Search Algorithms*

PS methods search in the policy parameter-space by iteratively updating the policy parameters to increase the expected reward. Policy parameters are updated according to Eq. 1.

$$\theta_{t+1} = \theta_t + \Delta\theta \tag{1}$$

PS algorithms require sampled experience from the environment to determine the policy update step $\Delta\theta$. In a model-free approach the sampled experience is used directly to update a policy. Model-based approaches use the sampled experience to first learn a model of the environment, after which this model is used to search for policies. Model-based approaches can reduce the number of real-world samples, however, rely on an accurate model of the environment. This research specifically targets the transition from simulation to reality, removing the dependency on (simulation) models, and therefore favors model-free methods. In addition, learning a policy is less computationally expensive and simpler than learning an accurate model, facilitating on-line learning by MAVs with limited on-board resources.

Model-free PS algorithms follow three sequential steps: 1) policy exploration; 2) policy evaluation; and 3) policy update.[7] Policy exploration generates new policies which are evaluated in the evaluation step. The policy update step updates the policy using the received reward from evaluating the exploration policies.

This three-step cycle can be evaluated at every time-step or over an entire episode, referred to as step-based and episode-based learning, respectively. Episode-based learning is preferred for this research as a higher level policy –represented in a BT– is considered. The higher-level behavior policy combines several sub-behaviors. The performance of these sub-behaviors is dependent on both the other sub-behaviors and the environment. Optimizing on a step-bases can therefore lead to the optimization of the individual sub-behaviors instead of the general behavior.

Model-free episodic PS algorithms combine a promising set of features for this research. Model-free episodic PS algorithms can be further subdivided in two categories based on the approach to calculate $\Delta\theta$: using policy gradients or expectation-maximization. Both categories have achieved successful real-world learning but are susceptible to noise, are prone to hyper-parameter settings or require recording of the full trajectory over an episode.[23–25] These limitations are not desired for the objective of this research. Furthermore, none of these methods exploit the (expected) functional form of the parameter performance landscape to minimize the number of evaluations. The objective of this research is to formulate a PS algorithm which does not suffer from these limitations by exploiting the expected performance landscape of the BT parameters. The next subsection elaborates on BTs and describes the expected performance landscape of the BT parameters.

B. Behavior Trees as Behavior Representation

BTs are models to describe a system which can be represented graphically as a directed tree.[12,13] BTs can function as representation of robotic behavior. This section first describes the mechanics of a BT after which the advantages of a BT for this research are listed. The final subsections discuss the emergence of behavior from the BT parameters and hypothesizes about the performance landscape of these parameters.

1. *Behavior Tree Mechanics*

A BT consists of nodes which have a parent-child relation and a shared data and memory source called the *blackboard*. Nodes can be either *executable* or *composite* nodes. Executable nodes interact with the environment through the blackboard and composite nodes determine the flow of evaluation within the BT.

Evaluated nodes return a status depending on the node type and the current state. This research only considers the *success* and *failure* status. Additionally, two types of executable and composite nodes are considered. Executable nodes can be either *action* or *condition* nodes. Action nodes determine the action and return success when the action is saved to the blackboard. Condition nodes return success if the condition specified by the node is met. Executable nodes are specific for the robotic platform and the task.

The return status of composite nodes depends on successful activation of two or more child nodes. This research adopts composite nodes of only two types: *sequence* and *selector*. Sequence nodes succeed if all children return success and selector nodes succeeds if one child returns success. Both types evaluate their children from left to right.

A BT execution sequence is referred to as a *tick*, initiating BT execution at a certain frequency. The tick starts at the root and propagates through the BT hierarchy until the root node receives a success status or all branches have been evaluated. At the end of a tick the action currently on the blackboard is activated.

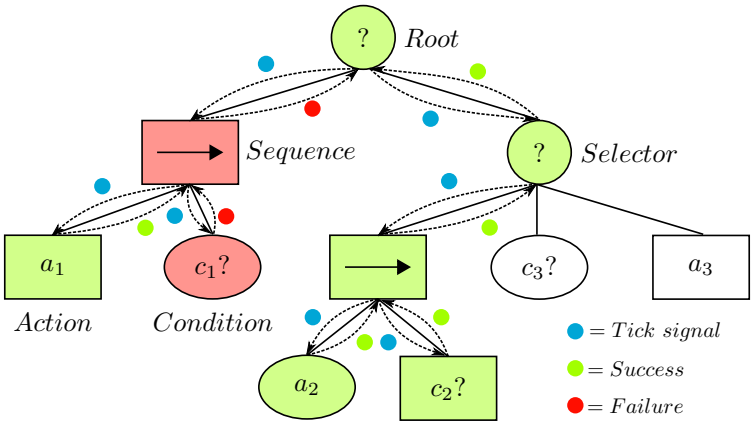


Figure 2: Visual representation of a BT with two branches. Node types are indicated in the figure.

Figure 2 visualizes the tree structure of a BT and demonstrates how the composite nodes control the flow of execution through the tree. The BT consists of two branches. In the current situation, action 2 is performed as this action is last to be saved to the blackboard and the root node receives a success signal from the second branch.

2. Advantages of Behavior Trees as Behavior Representation

A behavior representation functions as a mapping between the state and action-space. Compared to other behavior representation methods, BTs have tree advantages for the objective of this research.

First, BTs are modular and simple to interpret as the tree-structure hierarchically decomposes the main task in smaller sub-tasks. These sub-tasks are stored in different branches of the BT which facilitates understanding of the behavior by the designer. Additionally, this makes BTs simpler to adjust as it results in fewer dependencies than, for instance, artificial neural networks.

Second, BTs are computationally efficient as condition nodes segment and combine states in the state space and action nodes mark a specific action or set of actions in the action space. Subsequently, this direct mapping does not require calculation of a specific action given the current state. Furthermore, the hierarchical evaluation of the branches and control flow nodes minimize the number of nodes to be evaluated.

Finally, BTs are scalable to complex tasks as the tree structure does not follow a state-based formulation. As a result, the hierarchical tree structure suffers less from a *state explosion* with increasing task complexity or number of failure modes compared to Finite-State-Machines.

3. Emergent Behavior Through Node Coupling

BTs express a general behavior by combining sub-behaviors which are encoded both in the tree structure and the parameters of the corresponding nodes. Given a BT structure, the condition node parameters, $\theta_{cond.}$, determine in which states a sub-behavior is activated and the action node parameters, $\theta_{act.}$, regulate the specific action. Akin to other behavior representation methods, successful global behavior of BTs requires synchronization of the sub-behaviors to ensure the correct sub-behavior is activated which performs the optimal action.

In addition to the BT structure and parameters, the behavior depends on the operating environment. If the environment conditions change or are different from the conditions used to formulate the BT, the BT is required to be updated to the new environment such that the BT encodes a high-performance behavior. This is especially true for robotics with limited on-board resources, such as MAVs, which exploit sensory-motor coordination to increase performance. Sensory-motor coordination can be exploited by the robot to influence the sensory input it receives through the actions it selects.[9,11] As such, the optimal BT parameter settings are also coupled to the operating environment.

Following the reasoning of the previous two paragraphs, the expressed behavior of a given BT can be considered as an emergence from the coupling between the BT structure, BT parameters and the environment. This coupling is not unique to BTs, but applies to other behavior representation methods which combine sub-behaviors.[9,10] If the changes to the environment are small and a successful initial BT is available, it is assumed that the BT performance can be improved by tuning of the BT parameters only. Tuning of the parameters implies finding the optimal values of $\theta_{cond.}$ and $\theta_{act.}$ such that a similar behavior as in the original environment is expressed in a comparable but different environment.

Considering the coupling of the parameters we hypothesize that the functional form of the BT parameter performance curve has a distinct peak among a more flat surface. The peak represents the parameter range which results in well-coordinated sub-behaviors. Outside of the peak range the parameter settings express a sub-behavior which no longer contributes to the general behavior, resulting in a low-performance behavior.

As such, for a given representation structure and environment the expressed behavior can be interpreted as the correlations between the BT parameters. We therefore believe the hypothesized functional form of the representation parameters extends to other methods to represent behavior because the expressed behavior remains a result of the correlations between these parameters.

In this research, however, we will validate and confine the hypothesis to BTs only. The hypothesis on the functional form is further analyzed in Section IV for the task described in the next section. Section V thereafter elaborates on how the proposed PS algorithm exploits both the functional form of the performance curve and the interpretability of a BT representation to reduce the number of required evaluations.

III. DelFly Window Fly-Through Task

A. Task Description

In this research the DelFly Explorer is tasked to escape from a rectangular room by flying through a square window after being initialized at a random location and orientation. To achieve this, the DelFly implicitly has to avoid walls, locate the window and perform a fly-through maneuver using on-board sensors only. Given the task complexity, the problem is reduced to directional control only; during flight the altitude and speed remain constant at 1.5 meters and 0.5 meters per second, respectively.

(a) **(b)**

Figure 3: Picture of the DelFly Explorer flapping wing MAV (a) and rendering of the wall, floor and ceiling textures of the simulated square room (b).

The DelFly Explorer is a bio-inspired flapping wing MAV shown in Figure 3a.[3] The 4-gram on-board vision system consists of two cameras used for visual navigation. Both cameras approximately have a 60° by 45° field of view which captures a 128×96 resolution image. The spacial separation of the two on-board cameras creates a stereo image of the room. This stereo image is processed to generate a disparity map using the *LongSeq* algorithm.[4] The disparity map is used to extract features from the environment. The

BT generated by [Scheper et al., 2016] employed two features: 1) the sum of the disparity values and 2) a window response value. We will focus on the two features for now, the next section elaborates on the generated BT.

The disparity sum used by [Scheper et al., 2016] is equal to the summed disparity of all active pixels in the disparity map scaled with an 8-bit scaling factor. Instead of the disparity sum this research employs the average disparity of the active pixels in the disparity map. This feature reduces the influence of environment textures and camera specifications on the measurement.

By applying an integral image algorithm to the disparity map the (low disparity) window can be identified on the (high disparity) wall. The certainty that a window is present in the disparity map is captured by the window response value, where a low value corresponds to a high certainty. For a more detailed description of the DelFly, the on-board vision system, the LongSeq and window response algorithm the reader is referred to [de Croon et al., 2009],[3] [de Wagter et al., 2014][4] and [Scheper et al., 2016].[2]

A simulated square room of $8 \times 8 \times 3$ meter functions as environment for the DelFly. The textured floor, walls and ceiling ensure that the vision algorithms can detect a sufficient amount of features. The 0.8×0.8 meter window is placed in the center of the eastern wall. Behind the eastern wall an mirrored copy of this room is placed to ensure the cameras detect texture when looking through the window. Figure 3b shows a visual rendering of the virtual room and textures.

B. Simulation Model

[Scheper et al., 2016] employed the SmartUAV simulation platform to evolve a BT for the window fly-through task. SmartUAV is a simulation environment, created in-house at the Delft University of Technology, which is primarily used to simulate small and micro air vehicles.[27] Vision-based algorithms can be tested extensively as SmartUAV creates a detailed 3D test environment.

To reduce simulation time, however, a custom simulator was implemented, referred to as FastSim. The simulation model consists of the DelFly dynamics and the vision system. The following section elaborates on how both systems were implemented in FastSim.

1. DelFly Dynamics Implementation

The DelFly dynamics are simulated using the model parameters identified by [Armanini et al., 2016].[26] During simulation the dynamics are calculated at a rate of 100 Hz and are integrated using an Euler integration scheme.

In [Scheper et al., 2016] a simplified model of the DelFly dynamics was implemented due to the lack of accurate models of the DelFly Explorer at that time. With these new dynamics the flight modes are no-longer uncoupled and the system dynamics will behave more like the actual DelFly dynamics. We expect that the influence of the new dynamics on the task performance is limited as the task is confined to directional control only.

2. DelFly Vision System Approximation

The room features are captured by the on-board vision system in terms of the average disparity and the window response value. FastSim approximates both feature values to reduce computation time. The approximation functions were identified empirically from SmartUAV simulations by recording the average disparity and window response value for two flights where the DelFly flew straight at a wall and at the window, respectively. The obtained approximation functions are discussed below.

The disparity is inversely proportional to the distance in a nonlinear manner.[28] As such, the mean disparity is modeled with the distance to the nearest border, d_s in the environment. Equation 2 shows the exponential relation between the average disparity and the distance to an obstacle which is estimated through the SmartUAV simulations.

Because the disparity and distance are related nonlinearly, the field of view is divided in a number of segments. Line-plane intersection methods are implemented to calculate the distance from the DelFly to the closest room border which visible in the center of each segment. The approximated disparity over all segments is averaged to obtain the mean disparity in the entire field of view . By limiting the maximum

disparity at 16, coinciding with the typical settings of the on-board camera, and dividing the field of view in 17x17 equally spaced segments the disparity measurements of FastSim correspond to the results found in SmartUAV.

$$D_{segment} = 7.61 d_s^{-1.17} \tag{2}$$

Similar as the disparity sum, the window response is calculated with an approximation function derived from SmartUAV simulations. A low window response value σ corresponds to a square in the disparity map, indicating the window is nearby. In SmartUAV it was found that the window response increases polynomially with the distance between the DelFly to the window. The polynomial increase is expected to originate from inverse exponential relation between disparity and distance because the window response value is derived as a feature from the disparity map. Additionally, the window response requires the entire window to be in view. As such, the window response value is estimated from the distance to the window, d_w, using eq. 3. It was found that the window response value produces a stochastic signal in SmartUAV due to the feature size, therefore a white noise term is added to eq. 3. The white noise term w is uniformly distributed and has the value between -1 and 1.

$$\sigma = \begin{cases} -4.37 d_w^2 + 41.42 d_w - 25 \pm 2.5w, & \text{if window in frame and } d_w < 5 \text{ m} \\ 70 \pm 2.5w, & \text{if window in frame and } d_w > 5 \text{ m} \\ 90 \pm 10w, & \text{if window not in frame} \end{cases} \tag{3}$$

The purpose of the aforementioned approximation functions is to reduce computation time whilst providing an accurate estimate of the true average disparity and window response value. The functions, however, only approximate the true value. As such, we expect a clear reality gap to be present between the results generated in this research and the results of [Scheper et al., 2016]. Section VI analyzes this reality gap by quantifying the performance of the BT described in the next section.

IV. Behavior Tree Analysis

This section analyzes the BT generated by [Scheper et al., 2016] for the DelFly window fly-through task as described in the previous section. Additionally, this section demonstrates the coupling of the BT parameters as described in Section II. These couplings verify the hypothesis on the functional form of the performance landscape and the working principle of the PS algorithm presented in the next section.

A. Behavior Tree Description

The BT employed for the DelFly window fly-through task was evolved by [Scheper et al., 2016] and is depicted in Figure 4.[2] This BT is also used as behavior representation for this research. In the BT the action nodes set the rudder proportionally to the maximum deflection. The condition nodes analyze the state of the system by means of the average disparity μ_D and the window response value σ from the vision system.

The BT consists of five executable nodes and two composite nodes which combine three sub-behaviors. Nodes corresponding to the same sub-behavior are indicated with similar colors in Figure 4 and Figure 6. The top-left plot of Figure 6 visualizes the path of the DelFly during the window fly-through task. From both figures the following sub-behaviors become evident:

1. Slight right turn when disparity is low.
2. Max right turn when disparity is high.
3. Moderate left turn when disparity is high and window in field of view .

Combined the three sub-behaviors cause the DelFly to express a behavior which is expected to be specific for square rooms. If the DelFly is initiated facing towards a nearby wall, the DelFly will first attempt to avoid the wall and ensure it is oriented towards the room center (second sub-behavior). The DelFly then flies towards the room center (first sub-behavior) around which it circles to find the window (second sub-behavior). Once the on-board camera and algorithms detect the window, the DelFly starts the window approach (third sub-behavior). During approach the DelFly attempts to fly straight by alternating slight

right turns when the window is out-of-sight (first sub-behavior) and moderate left turns when the window is in view of the camera (second sub-behavior). The window approach maneuver is a clear example of sensory-motor coordination. These corrections allow the approach to essentially become a closed feedback loop with the objective to keep the window in sight. If, however, the DelFly loses the window from its field of view and is not yet close enough to look through the window, an evasion maneuver is executed to avoid a potential crash (second sub-behavior). The evasion maneuver brings the DelFly back to the room center from which a new approach may be initiated. If the DelFly gets close enough to look through the window, action 1 is activated until a window-fly through is realized.

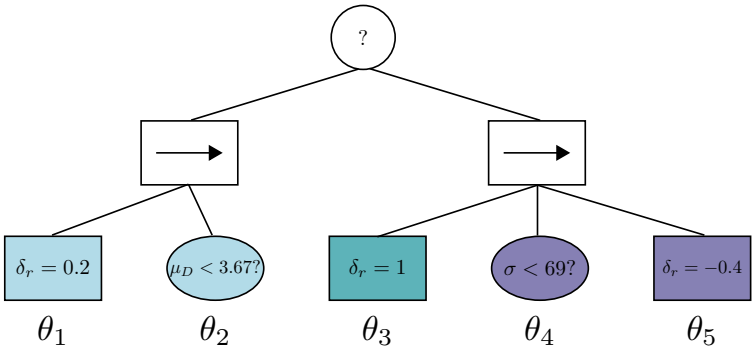


Figure 4: Behavior tree evolved by [Scheper et al., 2016] for the DelFly window fly-through task.[2]

B. Analysis of Emergent Behavior Through Node Coupling

The BT combines five parameters to express the window fly-through behavior; the five parameters are indicated below the corresponding nodes in Figure 4. Coupling of parameters implies that the optimal setting of one node depends on the setting of a different node. These couplings represent a behavior for a given behavior representation structure. In term, success of the behavior depends on the environment. As such, coupling can be quantified and analyzed by evaluating the success rate of the BT in completing the window fly-through task for different parameter setting combinations. To some extend, we believe that all parameters are coupled for a given BT structure and environment. To confirm the coupling, however, his analysis will consider only two combinations of two parameters because this simplifies the analysis whilst still clearly demonstrating the effect. Figure 5 visualizes the coupling of parameters θ_1 & θ_2 and θ_1 & θ_5.

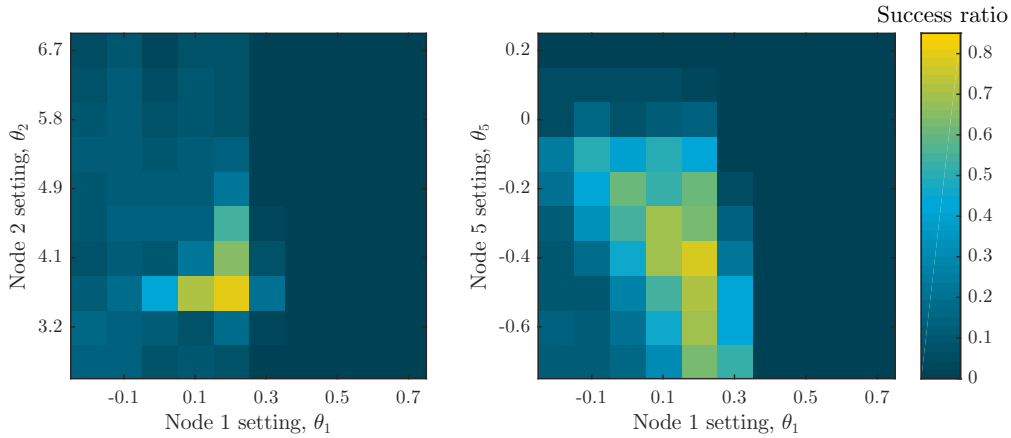


Figure 5: Policy performance for combinations of parameters 1 & 2 (left) and 1 & 5 (right) in the baseline environment.

Parameters θ_1 & θ_2 represent the first action and the first condition node, respectively. These parameters are coupled through the window-search and window-approach maneuver. From Figure 5 left it is evident

that the respective parameters are highly coupled: both nodes only allow for a small range of settings which result in a behavior with a high success-rate. If the threshold for the respective node is reached, the other parameter is unable to recover. Node 1 shows a clear threshold value of 0.2 above which the behavior becomes inefficient as the DelFly will crash into the window-pane. Values below 0 result in the DelFly being unable to return to the room center or even to diverge from the window during approach.

Parameters θ_1 & θ_5 represent the first and third action nodes and are coupled through the approach maneuver. The coupling approximately represents a negative linear relation with a small margin. This relation directly follows from the approach maneuver; the approximate negative relation ensures DelFly to fly as straight as possible. Again, node 1 has a clear threshold. In this case, however, the threshold can be compensated slightly by reducing the parameter value of node 5. A similar threshold is witnessed for θ_5 at a value greater than zero, causing a divergent approach behavior. These thresholds again demonstrate the importance of tuning the parameter settings: the other parameter are limited in their ability to compensate for the other parameter setting once the threshold is reached.

Each combination of parameters represents a behavior. Both parameter combinations in Figure 5 demonstrate that only a limited number of parameter combinations result in a successful behavior for the given BT and task environment. From these couplings it can be seen that the performance landscape indeed has a single peak where the sub-behaviors are coordinated most successfully, verifying the assumption made in Section II. The next section describes the proposed PS algorithm and how it leverages the performance landscape of the BT paramters.

V. Directed Increment Policy Search

DIPS is based on the observation that the performance landscape of BT parameters can be approximated by a flat surface with a single peak. As such, a simple hill-climb procedure may be initiated to tune the parameters to the new environment. The coupling of the parameters, however, results in a changing performance landscape when the policy parameters are updated. Additionally, exploring for every parameter may unnecessarily increase the number of evaluations as the individual parameter settings may have a reduced or increased impact on the final behavior. DIPS aims to mitigate these challenges by performing a hill-climb procedure where exploration is limited through policy increments which are directed towards either increasing, decreasing or static parameter values. DIPS consists of five sequential high-level steps; algorithm 1 shows the pseudo-code for DIPS, elaborating on the five steps discussed below:

1. Evaluate the current policy over t trials to generate a performance baseline.
2. Generate exploration policies for every selected parameter by perturbing the respective policy parameter in a specified direction and evaluate their performances.
3. Update the increment steps by increasing the step-size if the exploration policy has equal performance with respect to the baseline policy, or decrease the step-size if the exploration policy performed poorer.
4. Calculate the performance increase of the exploration policies with respect to the baseline policy. If the absolute value of the performance increase is lower than a threshold value, set the performance increase to zero. Calculate the total performance increase by summing the absolute value of the performance increases of the exploration policies.
5. Update the policy parameters by adding an increment scaled by performance increase of the specific node over the total performance increase to the current parameter value.

Step 1 evaluates the performance of the current policy parameters which functions as baseline for the exploration policies. The performance of a policy, $P(\pi_\theta)$, is defined as the mean reward over t trials. The number of trials is dependent on the reward function and the stochasticity of the environment.

Step 2 explores the parameter space with directed increments. The exploration direction is regulated by the *direction array*: an array of length n containing elements of value -1, 0 or 1 for decreasing, stationary or increasing parameter value, respectively. A direction-array has three main advantages which contribute to

Algorithm 1 Directed Increment Policy Search

function SEARCH($\theta_{initial}, D, \epsilon, \eta$)
  Initialize:
  $\varepsilon(n) \leftarrow \theta_{init}(n) \cdot \epsilon$
  repeat for every episode
    evaluate π_θ over t trials ▷ Step 1
    $P(\pi_\theta) \leftarrow \frac{1}{T} \sum_{t=1}^{T} r_t(\pi_\theta)$
    for all nodes where $D(n) \neq 0$ **do**
      copy current policy: $\theta_{explore} = \theta$
      perturb node n: $\theta_{explore}(n) \leftarrow \theta_{explore}(n) + D(n) \cdot \varepsilon(n)$ ▷ Step 2
      evaluate $\pi_{\theta_{explore}}$ over t trials
      $P\left(\pi_{\theta_{explore}}\right) \leftarrow \frac{1}{T} \sum_{t=1}^{T} r_t\left(\pi_{\theta_{explore}}\right)$
      $\Delta p(n) \leftarrow P\left(\pi_{\theta_{explore}}\right) - P(\pi_\theta)$
      if $-T_{min} < \Delta p(n) < T_{min}$ **then** ▷ Step 3
        $\Delta p(n) \leftarrow 0$
        $\varepsilon(n) \leftarrow \varepsilon(n) \cdot \eta$
      else if $\Delta p(n) \leq -T_{min}$ **then**
        $\varepsilon(n) \leftarrow \varepsilon(n) \cdot \frac{1}{\eta}$
      end if
    end for
    $\Delta P \leftarrow \sum_{n=1}^{N} |\Delta p(n)|$ ▷ Step 4
    $\theta(n) \leftarrow \theta(n) + D(n) \cdot \varepsilon(n) \cdot \frac{\Delta p(n)}{\Delta P}$ ▷ Step 5
  until stopping criteria is met
end function

reducing the number required real-world evaluations. First, the designer can incorporate a-priori knowledge on the task environment and behavior by specifying the update direction in which the policy parameter are expected to increase the behavior performance. Second, the direction array results in a dimensionality reduction as zero elements of the direction array require no exploration. Finally, by confining exploration to a single node in the direction of n^{th} element of D, the reward signal quality is increased as the performance increase is directly correlated to the perturbed node.

Formulation of the direction array has a large influence on the final policy. For simple tasks and small BTs the directions can be formulated using the designers knowledge of the task. Larger or more complex tasks, however, may require the directions to be sampled from the environment for every parameter. These samples can then be translated to a direction by differentiating between a performance increase and decrease. Sampling has the advantage that the dependency on a human designer is removed, however, increases the number of real-world evaluations and is more prone to noise and coupling effects.

Step 3 attempts to improve the initial increment step-size through the observed performance of the exploration policies. The increment step size ϵ regulates the size of the exploration step size and is expressed as a fraction of the original policy parameter $\theta_{initial}$. An increment step size which is too small may result in an increased number of real-world evaluations or may keep the algorithm from improving as the expected performance increase is too little. A large increment step can improve learning speed, but can also cause the algorithm to overshot, reducing algorithm performance. An increment scale factor $\eta > 1.0$ is introduced to solve this problem. If a performance decrease of an exploration policy is experienced, the respective increment is divided by η to reduce the increment size. This assumes that the algorithm overshoots the optimal parameter value when a performance degradation is measured. Likewise, the increment step size is multiplied by η when the performance of the exploration policy is equal to the baseline performance, assuming the current parameter setting is located on the flatter regions of the performance landscape.

Step 4 calculates a performance normalization factor for every parameter. The performance normalization factor is equal to the performance increase of the specific parameter over the summed absolute value performance increase of all exploration policies. A threshold for the performance increase, T_{min}, of all exploration policies is added to reduce the sensitivity to noise. Despite that exploration is directed by the direction

array, the performance normalization factor allows for policy updates in the reversed direction. Reversed policy updates occur when the performance increase of the respective exploration policy is negative. In this case, the performance scaling factor becomes negative, resulting in a policy update with a different sign as specified in D.

Step 5 updates the policy parameters using the direction array, increment step-size and performance normalization factor. The direction array ensures the sign of the update coincides with the exploration policy and the step-size and normalization factor determine the step-size of the update. Normalizing the step-size with the performance normalization factor has three effects. First, it is guaranteed that the step-size does not exceeded the parameter value of the exploration policy. Second, the normalization factor scales the step-size with the relative performance increase of each parameter to reduce the effect of parameter coupling. Third, it allows for policy updates in a different direction as specified as the direction array, as discussed in step 4.

VI. Experimental Set-Up

Three environments with an increasing reality gap are developed to test the DIPS algorithm. This section specifies the employed reward function, describes the adapted simulation environments and evaluates the experienced reality gaps.

A. Objective and Reward function

The goal of the learning agent is to search the parameter-space for a combination of BT parameters which, given the operating environment, result in the most successful window fly-through behavior. As such, the performance of a behavior policy should not be assessed by the expressed behavior, but by the reliability of the behavior in terms of successful task completion.

Many real-world applications of PS algorithms craft (continuous) reward functions using a combination of (external and on-board) measurements. Reward crafting can be very effective for lower-level behavior optimization, however, is believed to be less well suited for high-level behavior applications as the RL algorithm can become prone to optimizing sub-behaviors instead of the general behavior. As such, the reward function used in this research only considers simulation outcome of a given policy, rewarding reliability over the manner in which the task is completed. The reward function is defined as:

$$r\left(\pi_\theta\right) = \begin{cases} 0, & \text{if } t_{sim} > 100 \text{ [sec]} \\ 1, & \text{if crash} \\ 5, & \text{if success} \end{cases} \tag{4}$$

The discrete reward function of Eq. 4 rewards the three possible events which cause an end of simulation. By awarding a lower reward for running out of time the learning agent experiences a higher pressure to attempt a window fly-though maneuver. Note that this reward function assumes that the behavior of the initial policy is oriented more at avoiding walls than approaching the window. Such behavior is desired as this supports safer real-world evaluations. A reward ratio of 1 to 5 is adopted to significantly reward successful behavior over a crash behavior.

B. Simulated Reality-Gap

This research considers three adapted environments with an increasing expected reality gap. All three environments are adapted from the baseline environment described in subsection III. The three environments function as a measure of robustness to discrepancies between simulation and the real-world.

In Environment 1 the DelFly dynamics are changed to allow a maximum rudder deflection of 0.26 radians, causing the DelFly to make sharper turns as the BT actions specify the rudder deflection as a ratio of the maximum rudder deflection. As a result the minimal turn radius decreases from 1.25 to 0.5 meters. We expect a relatively small reality gap in this environment which can be improved by scaling the action node parameters.

Environment 2 simulates the real-world experiment of [Scheper et al., 2016]. This environment combines both the increased maximum rudder setting and the reduced room dimension of 5×5 meters. Here we expect

a medium reality gap as the learning agent has to cope with two significant changes and an exact copy of the baseline behavior is no longer possible.

The third environment incorporates the same changes of the second environment and additionally increases the room width to 7 meters, such that none of the room dimensions matches that of the baseline. Because the room is also no longer of a similar shape, it is unknown if the behavior will still be effective or whether the behavior can be adapted to the new room shape. It is expected that the behavior encoded in the BT is specific for a square room as wall-avoidance behavior has to align the DelFly at a specific location in the room such that the approach angle towards the window is within a small margin.

Table 1 summarizes the environmental parameters of the three test environments and quantifies the reality gap in terms of the success rate. The quantified reality gap is obtained by initializing the DelFly with the original BT parameters 500 times in each of the environments. The quantified reality gap coincides with the expected reality gap, increasing from Environment 1 to 3. Notably, the baseline environment also suffers from a small performance degradation as compared to the 88% found in [Scheper et al., 2016]. This discrepancy is expected to originate from the updated dynamics and vision system approximations made in the FastSim simulator.

Table 1: Environmental variable settings of the baseline and three adapted environments.

Name	δ_{max} $[rad]$	Room dimensions $[m \times m]$	Expected reality gap	Initial success rate [%]
Baseline	0.065	8×8	-	72.4
Environment 1	0.26	8×8	*Small*	3.0
Environment 2	0.26	5×5	*Medium*	2.2
Environment 3	0.26	7×5	*Large*	1.0

C. Simulated Reality-Gap Behavior Analysis

Figure 6 shows three flight path of the DelFly controlled by the BT of Figure 4 for the four simulated environments. From the flight paths it can be observed that the BT expresses the three sub-behaviors in all environments. The coupling between the sub-behaviors and the environments also becomes apparent as the coordination of the sub-behaviors is less effective in the adapted environments.

In Environment 1 the increased maximum rudder deflection causes the DelFly to approach the window at a lower angle than the baseline. This results in an early approach abort as the limited field of view of the camera is unable to keep the entire window in sight until the the majority of the disparity is generated from textures behind the window. This results in a high window-response and disparity, causing the second action to be executed.

The reduced room dimensions of Environment 2 cause the DelFly to experience high disparity throughout the room. Therefore the DelFly searches for the window near its initialization location instead of around the center of the room. This produces unreliable window search behavior and unstructured approaches at many different angles, resulting both in the DelFly flying next to the window or continuously turning around a single point.

The anti-symmetric room of Environment 3 causes the disparity measurements to differ in x and y-direction. The DelFly therefore does not return to the center of the room to search for the window. In Environment 3 the DelFly fails because of the reduced approach angle of Environment 1 and the continuous turning of Environment 2.

D. Application of DIPS to DelFly Window Fly-Through Task

DIPS requires a total of five inputs: an initial policy, ϵ, η, T_{min} and the direction array. The policy parameters listed in Figure 4 function as initial policy for all environments. Parameters ϵ and η influence the performance of the obtained policy, the consistency of finding good policies and the number of required episodes to obtain a high performance policy. The sensitivity of DIPS to parameters ϵ and η is assessed based on these criteria in Section VII. Initially ϵ and η are set at 0.2 and 1.2, respectively.

The elements of D are constructed to facilitate the BT to express similar behavior based on the adaptations made to each of the environments described in Section VI.

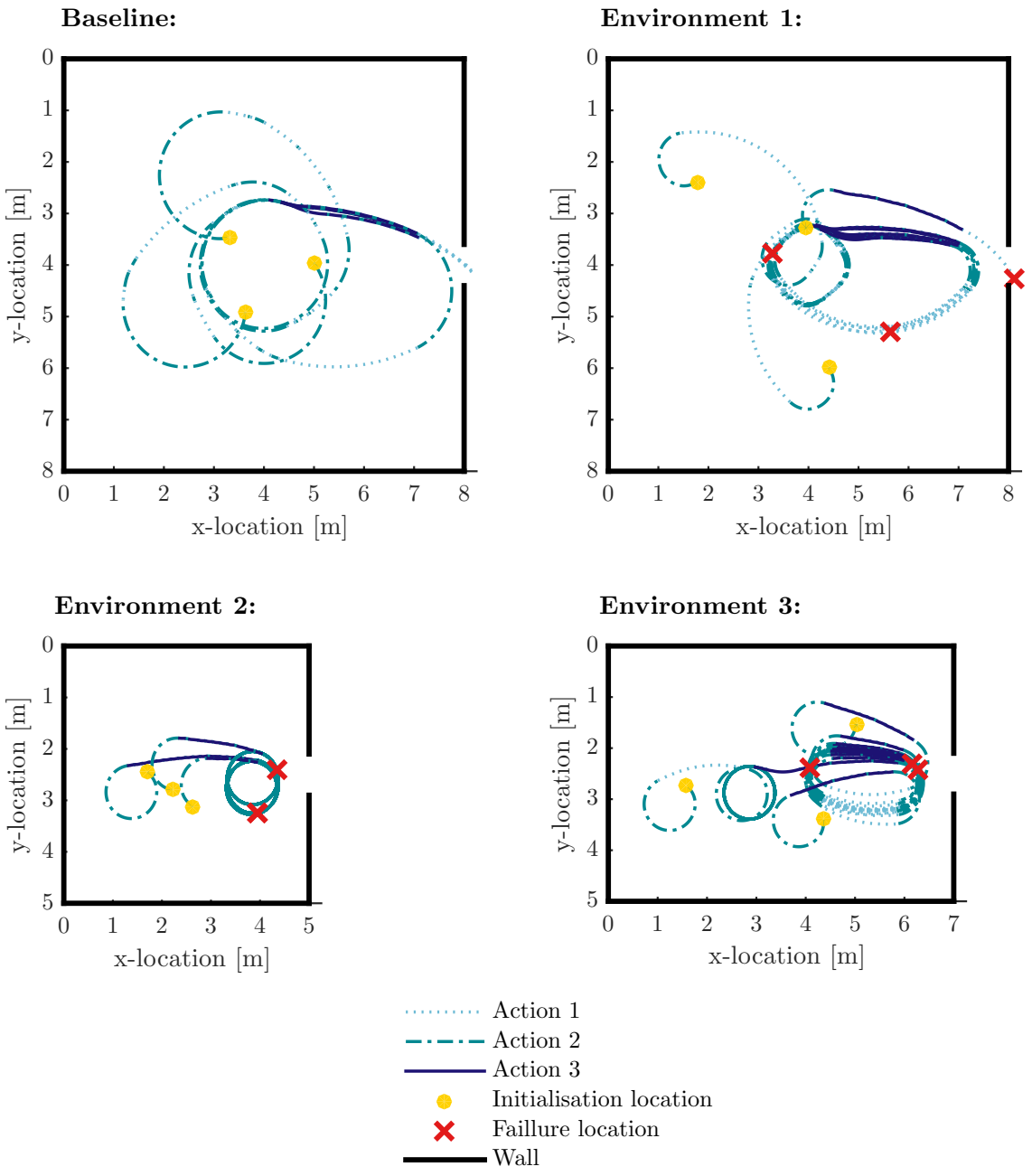


Figure 6: Three example paths of DelFly controlled by the original Behavior Tree settings for the baseline and three adapted environments.

Environment 1 differs from the baseline environment in terms of the maximal elevator deflection. Therefore, only the actions have to be considered during the optimization process. From Table 2 it can be seen that the parameters corresponding to the action nodes are all reduced to account for the increased maximal elevator deflection. Note that the elements of D specify the direction in which the nodes are perturbed; in order to reduce the deflection of the fifth node, the fifth element directs the updates to the positive axis.

For Environment 1 we make the assumption that the increased maximum rudder deflection counters the reduced room dimensions. In the behavior analysis, however, it was concluded that the success of the DelFly was highly dependent on the final part of the window approach, governed by the first action node. Therefore the first action node is the only action node to be adapted in this environment; action 1 will be reduced to support the final part of the approach. The reduced room dimensions cause the DelFLy to experience an increased disparity throughout the room. As such, the disparity value of the second node is required to be increased to facilitate a behavior similar to the baseline. Finally, the threshold of the window response is left unchanged because having the window in view is the limiting factor of the window search behavior for the reduced room size, rather than the distance to the window.

Environment 3 combines the adaptations of Environment 2 with the added increased room width. As such, we expect that the derived elements of D for Environment 2 are also applicable to Environment 3. Despite that the elements of D are equal for both environments, we hypothesize that DIPS will find different settings for the considered parameters due to the extended room width of Environment 3.

Finally, the policy performances are calculated as the mean reward over 5 trials to balance the number of evaluations and accuracy of the policy performance prediction. It was noticed that the success of a policy is highly dependent on the initialization locations, especially when the policy is still low-performing. Two techniques are implemented to increase the signal-to-noise ratio. First, a minimum performance threshold is implemented and fixed at a value of $T_{min} = 0.4$. This value ensures that at least one extra trial must receive a 5 points reward or at least two extra trials must receive a 1 point reward. Additionally, all trials during an episode are evaluated using the same initialization location and orientation. After every episode 5 new initialization locations are randomly generated to facilitate optimization of the general behavior.

Table 2: Direction array for the three adapted environments.

Environment	$D(n)$
Environment 1	[-1 0 -1 0 1]
Environment 2	[-1 1 0 0 0]
Environment 3	[-1 1 0 0 0]

VII. Results

This section presents and discusses the results of applying DIPS to the three adapted environments of the DelFly window fly-through task. Subsection A provides an insight in the learning process by evaluating the parameter settings per episode and the performance of the respective settings. Thereafter the performance landscape of the parameters is analyzed to obtain a better understanding of the learning process. The next subsection elaborates on how the DIPS algorithm improved the task performance by comparing the expressed behavior after learning to the initial behavior seen in Figure 6. Finally, the sensitivity of DIPS to hyper parameters ϵ and η is assessed to conclude on the robustness of DIPS.

A. Learning Process Analysis

Figure 7 displays the parameter settings and performance of the respective settings for all adapted environments over 20 episodes of the learning process. These results are obtained over 100 independent runs using the ϵ and η settings as listed in Table 2. The parameter performance is measured by evaluating the parameter settings over 100 initializations using the node settings of the respective episode. Table 3 summarizes the results of Figure 7. Note that the required number of evaluations in the fourth column is calculated using number of episodes until the policy converged to within 5% of the maximum value of the median performance.

For Environment 1 DIPS improves the success rate of the behavior by 2% relative to the baseline environment. Three observations can be made from the learning process:

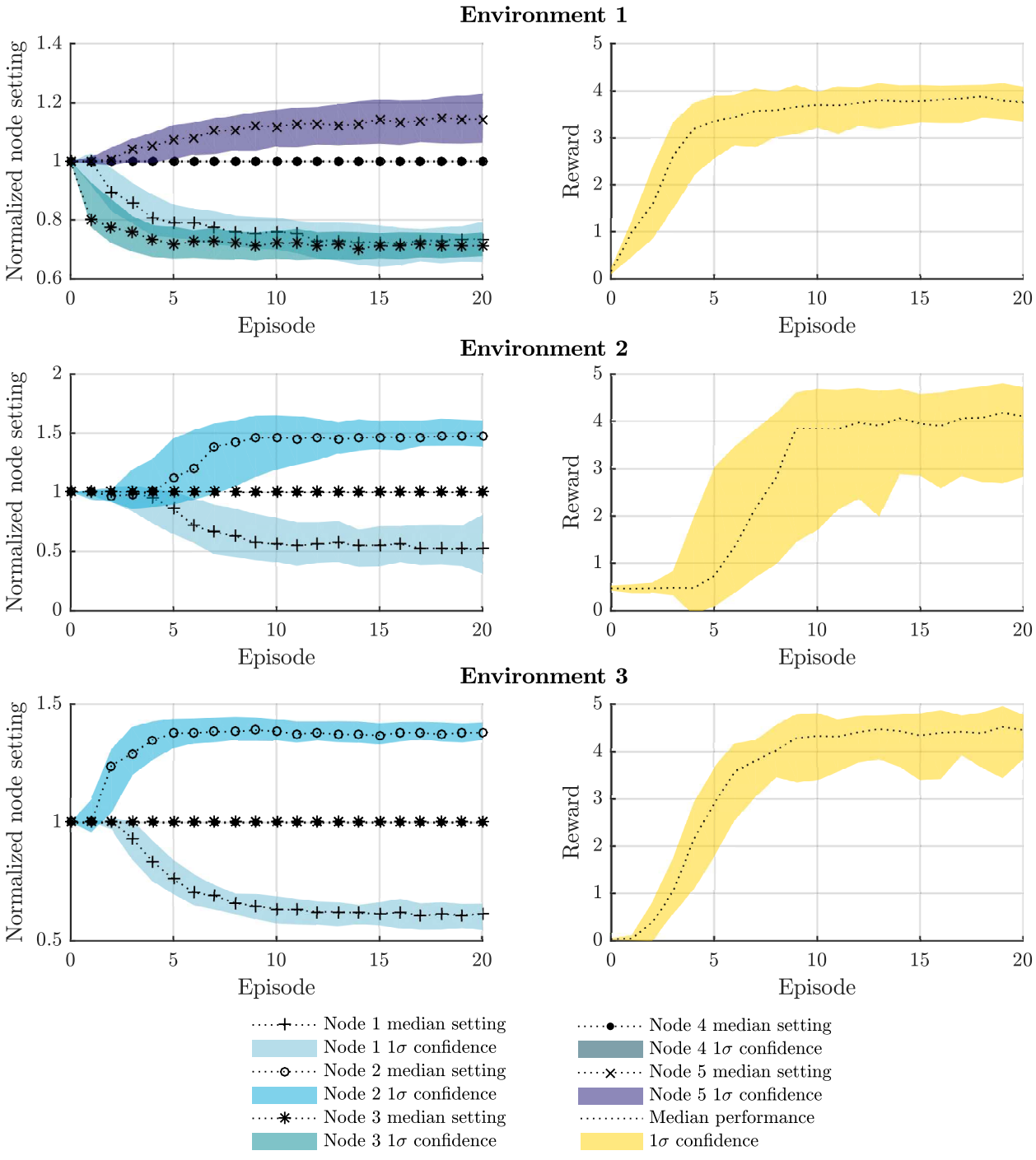


Figure 7: Learning process of Directed Increment Policy Search for the three adjusted environments. The confidence interval of both the parameter settings and the rewards have been calculated over 100 independent runs. The parameter settings are normalized with respect to the initial settings at episode 0; the reward is obtained by evaluating the policy at the corresponding episode over 100 initializations.

Table 3: Learning results of DIPS for the three adapted environments, policy parameters adapted by DIPS indicated in bold.

Name	Initial reward	Final reward	Number of evaluations	Initial success	Final success	$\theta(n)$				
Baseline	3.84	–	-	72.4%	-	[0.20	3.67	1.00	69	− 0.40]
Environment 1	0.15	3.76	180	3.0%	71.8%	[**0.15**	3.67	**0.71**	69	− **0.44**]
Environment 2	0.52	3.91	135	2.2%	75.8%	[**0.10**	**5.33**	1.00	69	− 0.40]
Environment 3	0.05	4.37	135	1.0%	91.2%	[**0.12**	**5.06**	1.00	69	− 0.40]

- Parameter setting of node 5 advances in the reversed direction as specified by the direction array. Reversed updates are allowed by DIPS if the exploration policies of the respective parameter result in a reduced performance with respect to the current policy. These reversed parameter updates demonstrate that the direction array functions mainly as a biased form of exploration; the algorithm can recover if an element in D is specified incorrectly or DIPS overshoots the optimal setting. This effect is further discussed in the next subsection.

- Parameters of nodes 1 and 3 show a clear coupling, where the first node is updated after the third node. In the first episode the parameter setting of node 3 is reduced to 80% of its original value, corresponding to the initial step size of $\epsilon = 0.2$.

- Policy performance increases to 90% of the maximum performance in the first 4 episodes; the last 10% takes an additional 5 episodes. This is typical for RL problems as policy improvements are less pronounced for high-performing policies. Additionally, the influence of the minimum performance threshold T_{min} diminishes because of the improved policy performance in the later episodes. Therefore, the update strategy of DIPS becomes more susceptible to noise and allows for parameter updates in the reversed direction of D, making the optimization more difficult. A higher threshold is expected to mitigate this effect, however, can also reduce learning rate in the early episodes.

Similar as for Environment 1, DIPS improves the success rate of the behavior to 75.4%, up from 2.2% of the initial parameter settings. Specific for Environment 2, the following thee observations can be made from the learning process:

- DIPS requires a median of 4 episodes to realize an update leading to a policy with improved performance. During these initial episodes, the algorithm updates the step-size to increase the probability of generating an exploration policy with increased performance. Because of the large difference between the initial node settings and the final node settings listed in Table 3, it is expected that the initial ϵ setting was too low to observe a performance increase. This hypothesis is investigated in the next subsection.

- The spread of the parameter values is greater in the initial episodes. This spread originates from updating the exploration increment step-size during the initial episodes. The initialization locations encountered by DIPS differ per episode. Because the initialization location has a large influence on the performance of the policy, the effectiveness of a the current step-size is stochastic. As such the parameters updates are stochastic, resulting a parameter values with a higher spread in the initial episodes. This effect is particularly pronounced in this environment because the high spread is amplified by the extended updates of the step-sizes in the initial episodes.

- Parameter value of node 2 is increased only after the value of node 1 is updated due coupling effects as described earlier for Environment 1. Increasing the value of node 2 in this environment only increases the performance when the value of node 1 is reduced because the DelFly must approach the window with a reduced turn rate to avoid hitting the window pane.

- Parameter value of node 2 is decreased in the initial episodes. Similar as in Environment 1, this demonstrates that DIPS can update parameters in a reversed direction as specified in the direction

array. Combined with observation 1, this effect contributes to the increased number of required episodes compared to the other environments.

DIPS increases the policy success rate in Environment 3 to 91.2% from an initial success rate of 2.2%. The following two observations can be made on the learning process:

- DIPS requires a median of 1 episode to update the step-size which results in the first policy update. Compared to Environment 2 this environment requires fewer episodes to start improving the policy. We therefore hypothesize that the peak in the performance landscape is closer to the initial policy for this environment than for Environment 2.

- Nodes 1 and 2 converge to parameter values with a relatively low standard deviation of approximately 0.05. We expect the low standard deviation to be the result of a performance landscape peak over a small range of values. This hypothesis is investigated in the next subsection. Similarly as in Environment 2, both nodes are coupled.

B. Performance Landscape of Node Settings

To obtain an increased understanding of the learning process this section analyzes the performance landscape of the individual nodes for all environments. These performance landscapes are displayed in Figure 8. The performance landscapes of Figure 8 are generated by averaging the obtained reward of the specified node setting over 100 independent evaluations. Every node and environment uses the improved policy found by DIPS as listed in Table 3.

As expected, all nodes coarsely express a one-peak performance landscape. The following observations can be made from Figure 8 which provides additional insights in the learning process:

- Nodes 1 and 2 have the smallest value range which result in a high-performing behavior. This sensitivity corresponds to the understanding of the behavior encoded in the BT as discussed in Section IV. This shows that the dimensionality reduction of the D array can effectively reduce the number by limiting the optimization to the parameters with the largest impact on the behavior performance.

- The individual parameter settings of the policies listed in Table 3 are tuned to the optimal settings, provided the settings of the other parameters. This demonstrates that, despite the couplings, DIPS iteratively locates the performance curve peak of the nodes considered in the optimization process.

- The performance landscape of all nodes is similar-shaped for environments with comparable conditions. As such, the performance landscape of the baseline environment and Environment 1 are similar-shaped; this is also true for Environment 2 and 3. Additionally, the performance peak of node 1 is merely translated under the adapted environments and respective final policy parameter settings. This suggest that, provided the considered environment conditions, the optimal setting of this node is dependent on the maximum rudder deflection only. Remarkably, the performance landscape of Environments 2 and 3 allow the node setting of node 5 to become slightly positive. This is the result of the reduced approach distance in these environments. The reliability of positive settings for node 5, however, is lower due to the reduced performance and increased stochastic in the performance landscape estimate.

- The difference between the baseline parameter settings and the optimal parameter settings for Environment 2 are greater than for the other environments. This raises the question of which environment results in the largest reality gap. Despite that the DelFly was able to initially achieve a higher success rate in Environment 2 than in Environment 3, we argue that the adaptations of Environment 2 result in a larger reality gap because the BT parameters are required to be changed more for this Environment.

Note that the performance landscape of Figure 8 is limited to a range of node settings and considers only the performance landscape of the individual nodes, not the coupling between the nodes. As such, the discussed performance landscape and observations only hold for parameter tuning. Because DIPS explores the parameters individually, the algorithm is limited to tuning only. As such, the performance landscape of Figure 8 is considered representative to the landscape encountered by DIPS. If the entire performance

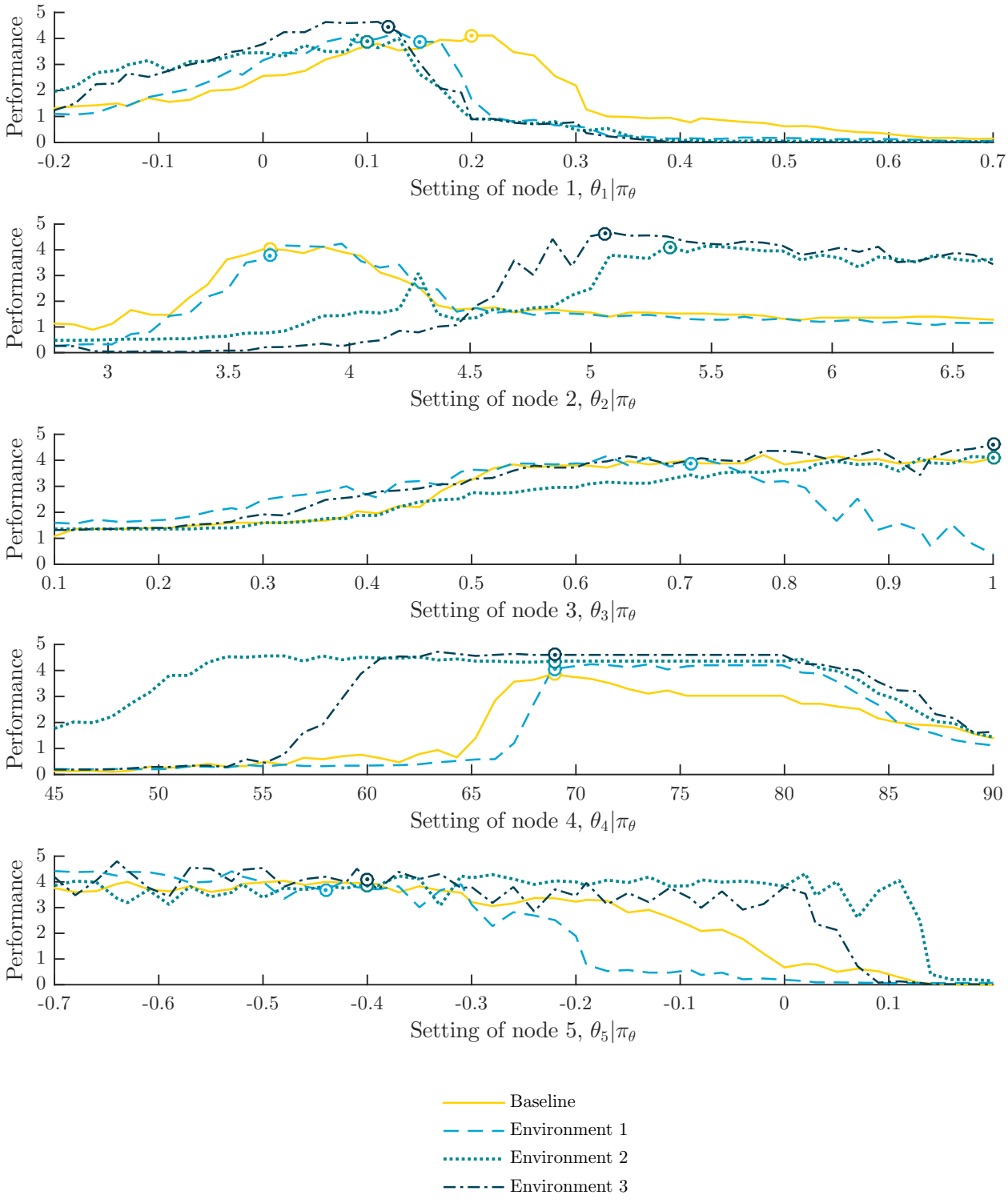


Figure 8: Performance landscape of nodes 1 (top) - 5 (bottom), given the other initial policy parameters, for the baseline and three adapted environments. Optimized parameter settings found by DIPS are indicated with a circle.

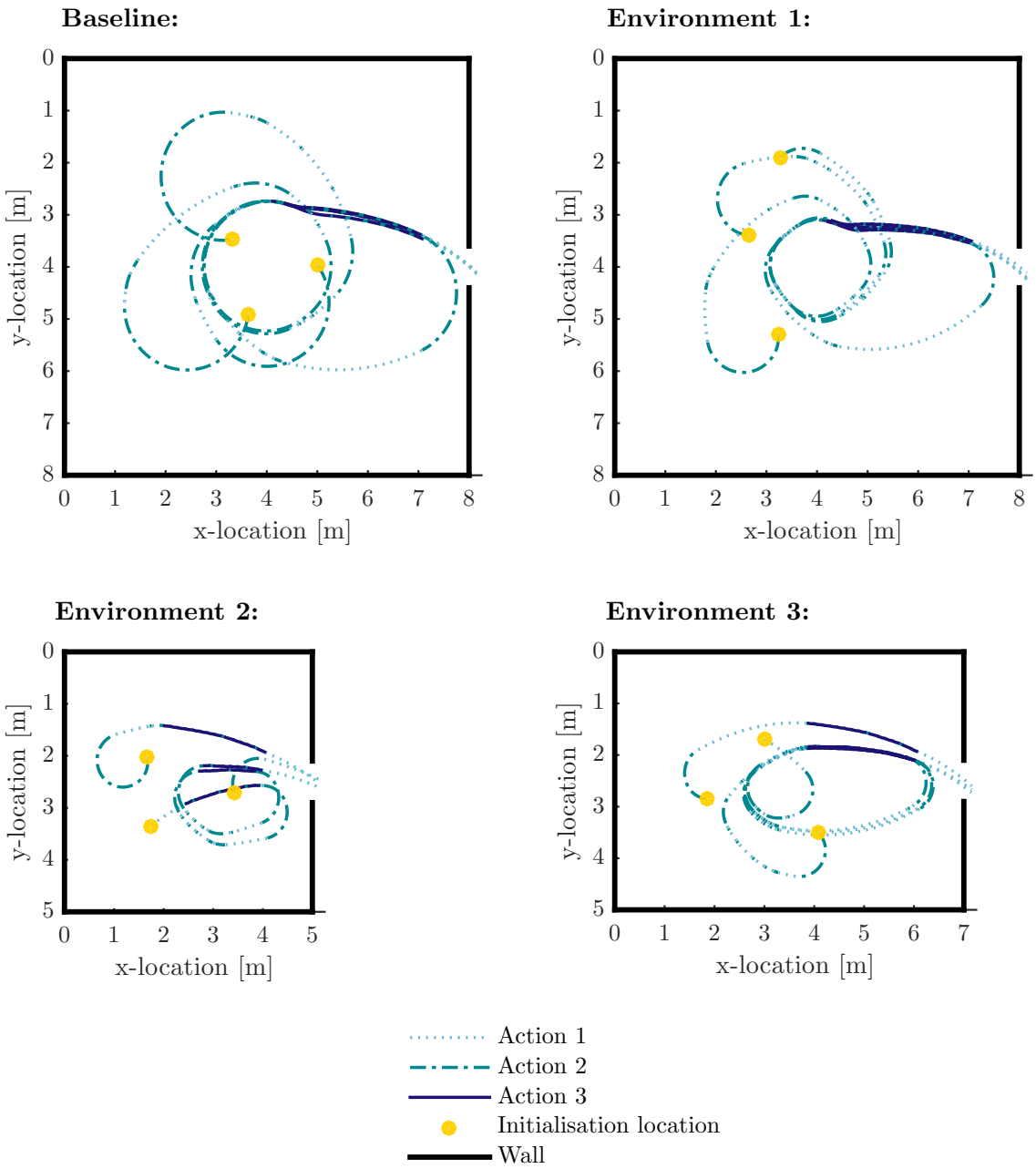


Figure 9: Three example paths of DelFly controlled by the policies found by DIPS for the baseline and three adapted environments.

landscape would be considered, however, multiple peaks are expected to be present. An example of a peak in the higher-dimensional performance landscape would be at the reversed sign action node parameter settings. In this case the DelFly would express the same behavior, however, turn counter clock-wise around the room center instead of clock-wise.

C. Behavior Improvement Analysis

Figure 9 displays three typical paths of the DelFly controlled by the BT of Figure 4 using the BT parameters listed in Table 3 for the baseline and adapted environments. Compared to Figure 6, it can be seen that the policy parameters found by DIPS successfully adapt the behavior to the experimental environments, expressing a behavior similar to the behavior in the baseline environment.

In Environment 1 the DelFly expresses the same three sub-behaviors as in the baseline environment. By reducing the action nodes, the effect of the increased maximum rudder setting is mitigated. Compared to the baseline behavior the DelFly flies through the window at a similar approach angle and location, despite taking sharper turns in general.

By increasing the average disparity condition of node 2 the DelFly is able to express the missed approach behavior in Environment 2. The new behavior now much better resembles the baseline behavior, resulting in a significant performance increase. Because the direction array did not consider optimizing the window response value of node 4, the DelFly approaches the window from a relatively large distance compared to the room width. If the DelFly is initialized near the top half of the room, this results in a direct approach maneuver. If the approach is unsuccessful, the DelFly is able to adjust with the improved missed approach behavior.

Interestingly, the behavior in Environment 3 is able to adapt and resemble the baseline behavior despite that the room now has a rectangular shape. The missed approach behavior is adapted to the square environment by increasing the disparity; the increased maximal rudder deflection allows the DelFly to return to the room center.

D. Parameter Sensitivity

Robustness to hyper-parameter settings was one of the algorithm requirements set out in the introduction. An algorithm that is robust to its hyper-parameters requires less tuning, which in term reduces the number of real-world evaluations and increases the real-world functionality. To measure the robustness of the algorithm, a grid-search is performed for the two hyper-parameters ϵ and η. These parameters were selected because they determine the exploration step-size and therefore directly influence the performance of the algorithm.

Both ϵ and η are evaluated using 10 settings bounded between 0.05 & 0.5 and 1.05 & 1.5, respectively. Twenty-five independent training runs are performed for every combination of ϵ and η. Each hyper-parameter configuration is evaluated on three criteria over all runs: the mean maximum achieved performance, the mean number of episodes to reach maximum performance and the standard deviation at maximum performance. These criteria are a measure for performance, consistency and practical application, respectively. Figure 10 presents the results of the hyper-parameter sensitivity analysis for all three adapted environments.

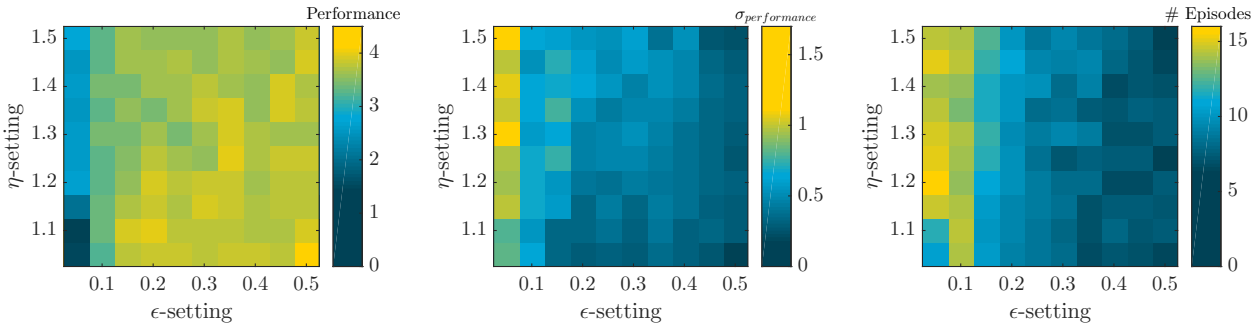
The sensitivity analysis reveals that DIPS is robust to the hyper-parameter settings for the smaller reality gaps of Environment 1 and 3, provided that extremes are avoided. For the larger reality gap experienced in Environment 2, DIPS is more sensitive to the hyper-paramter settings. Generally ϵ values below 0.10 are a hard threshold which result in low performance as the step-size is too small to measure the performance accurately over 5 trials, making the algorithm more susceptible to noise and therefore also reducing the functionality of η.

In Environment 1 DIPS experiences a maximum performance, consistency and convergence rate improvement with increasing values of ϵ. For this environment a small step size suffices as the reality gap is relatively small. Increasing η with ϵ therefore generally results in better settings because this allows DIPS to recover from the large initial step size dictated by large ϵ. This seemingly linear relation can be distinguished from Figure 10.

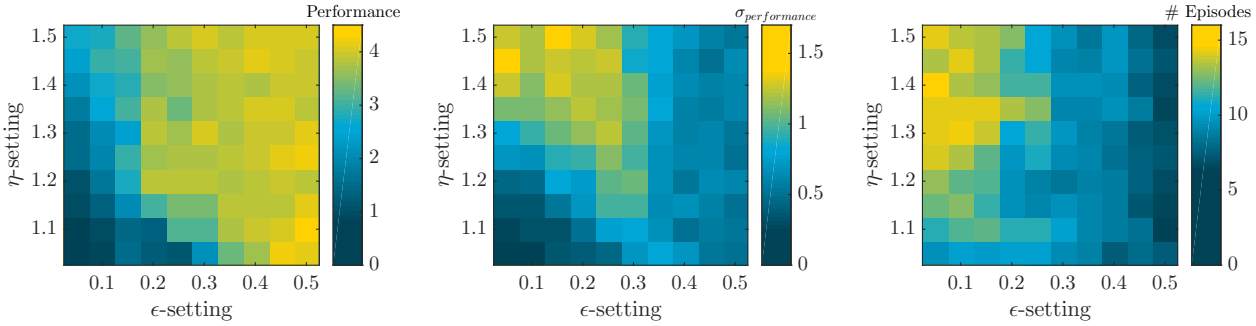
The adaptations to Environment 2 were shown to result in the largest reality gap in terms of the required parameter adaptation. This is also reflected in the sensitivity analysis. We analyze this environment by segmenting the hyper-parameter grid of the performance in three sections. The first section combines low

ϵ and η settings. DIPS is unable to find high-performing policies in this section. As a result, the standard deviation is very low because DIPS consistently produces bad policies. The convergence episode is considered to be random and therefore averages at around 10. The second section approximately combines lower ϵ values with high η values. DIPS can produce high-performing polices using these settings, however, this is highly dependent on the initialization locations encountered during learning. As such, the large spread in policy performances is reflected in the high standard deviation. Because initially ϵ is low, DIPS requires many episodes to obtain a step-size which allows a performance increase to be measured. Finally, the third segment consists of all higher ϵ values. In this section the performance and consistency of DIPS are robust to the hyper-parameters. The number of episodes reduces with increasing ϵ, as expected.

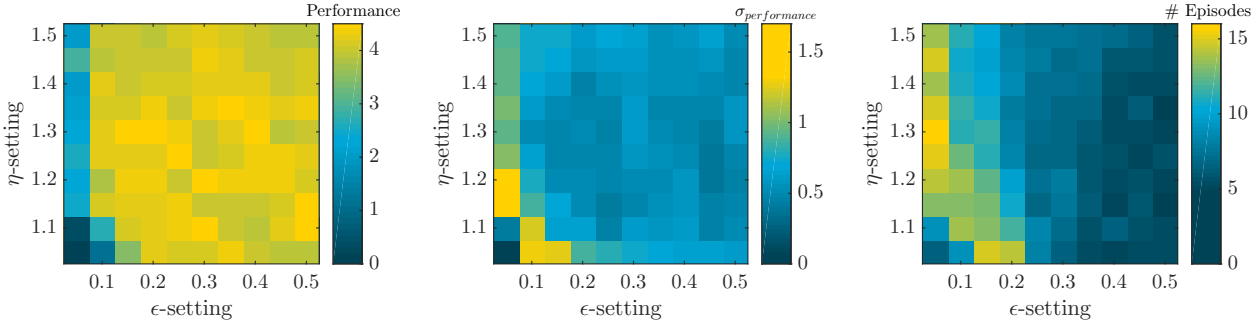
The performance and consistency of the policies found by DIPS in Environment 3 are no longer influenced by most of the hyper-parameter values tested in this analysis. For both measures, however, the threshold for small ϵ and η is increased slightly. In region of small values a similar behavior as in Environment 2 is experienced, but on a reduced scale. The number of episodes again decreases with increasing ϵ; η appears to be of a reduced influence.



(a) Environment 1.



(b) Environment 2.



(c) Environment 3.

Figure 10: Parameter sensitivity analysis of ϵ and η for the three adjusted environments. Sensitivity measured in performance mean(left), standard deviation (center) and mean number of episodes required to achieve maximum performance (right) over 25 independent learning sessions .

VIII. Discussion

In this section we evaluate the results achieved by DIPS by discussing the dependency on the direction array and the applicability DIPS in the proposed form. These discussions form form the basis of the recommendations for future research discussed in Section X.

The direction array is the predominant factor in balancing the number of evaluations and the performance of the final policy; formulating a correct direction array has a large influence on the final result. In this research the policy parameters were allowed to be undefeated in the opposite direction as specified in D if the incremented exploration policy resulted in a performance decrease. We therefore refer to this update strategy as *polytonic*. If the designer can conclude with certainty that elements in the direction array are correct, the update mechanics of DIPS can be updated to increase convergence rate. In this case the policy parameter may only be updated in the direction specified by the elements of D. The resulting *monotonic* DIPS is less susceptible to coupling effects where parameters are updated in the towards the reverse direction as specified by D. Additionally, monotonic DIPS is less influenced by noise for high-performing policies, as discussed in the previous section, increasing convergence speed during the later episodes. Table 4 lists the results of applying DIPS to the three adapted environments using the settings of Table 2. From this table it can be seen that monotonic DIPS reduces the number of required episodes and results in more aggressive parameter updates. Despite that monotonic updates outperform polytonic updates for some environments, we believe polytonic DIPS is the superior algorithm because it allows for greater robustness and can recover from overshooting the optimal parameter setting.

Table 4: Learning results of monotonic DIPS for the three adapted environments.

Name	Initial reward	Final reward	Number of evaluations	Initial success	Final success	$\theta(n)$				
Baseline	3.84	–	-	72.4%	-	[0.20	3.67	1.00	69	− 0.40]
Environment 1	0.15	3.94	120	3.0%	76.4%	[0.13	3.67	0.64	69	− 0.39]
Environment 2	0.52	4.08	75	2.2%	80.2%	[0.06	5.72	1.00	69	− 0.40]
Environment 3	0.05	4.54	75	1.0%	94.8%	[0.11	5.21	1.00	69	− 0.40]

Limiting the search space through the zeros in the direction array is one of the mechanisms of DIPS to reduce the number of required evaluations. Zero-elements in the direction array reduce the parameter search space by neglecting the respective parameters during the exploration and update process. To asses the influence of the zeros Table 5 presents the results of providing DIPS with a naive direction array, containing a one for all policy parameters. From Table 5 it can be seen that a naive direction array results in a reduced performance and increased number of evaluations. The reduced performance originates from the coupling between the policy parameters. Here, especially parameter 4 is updated during the initial episodes as this results in an increased performance due to the reward function assigning a greater reward for crashes than running out of simulation time. In the later episodes the increased value of θ_4 complicates the learning process and results in a lower performance. The increased number of evaluations results from both the reduced amount of zero-elements in D and the coupling effects which result in a reduced performance.

Table 5: Learning results of polytonic DIPS with a naive direction array for the three adapted environments.

Name	Initial reward	Final reward	Number of evaluations	Initial success	Final success	$\theta(n)$				
Baseline	3.84	–	-	72.4%	-	[0.20	3.67	1.00	69	− 0.40]
Env. 1	0.15	3.79	480	3.0%	71.8%	[0.15	3.83	1.00	77	− 0.42]
Env. 2	0.52	3.64	450	2.2%	68.2%	[0.08	5.21	1.00	79	− 0.46]
Env. 3	0.05	4.18	540	1.0%	86.4%	[0.12	5.02	1.00	76	− 0.44]

DIPS mitigates the node coupling by prioritizing nodes which result in the largest policy performance increase. The prioritization is regulated through the performance normalization factor, or step 4 of the DIPS algorithm. Essentially, this corresponds to updating nodes with the steepest gradient first because initially the relative increment step-size is equal for all parameters. For this reason we would classify DIPS as a finite-difference policy gradient method, despite that the algorithm does not directly estimate and exploit the policy gradients.

We believe DIPS is an efficient RL algorithm which significantly improved on the challenges set out in the Introduction. The high feedback signal quality, updates based on sampled performance instead of gradients and robustness to hyper-parameter settings makes DIPS an efficient PS algorithm. The required number of evaluations, however, still exceeds the number required for practical implementations, especially considering that multiple independent runs are advised in a real-world environment.

IX. Conclusion

The reality gap experienced by robotic systems is characterized by small discrepancies between the simulation and real-world environment which cause a simulation-learned behavior policy to become ineffective once transferred to a real-world environment. In this paper a simple PS algorithm has been proposed to cross the reality gap for autonomous MAVs controlled by a BT.

This research showed that, for the selected task and adapted environments, the performance of the general behavior could be improved by adapting the individual sub-behaviors. Adapting sub-behaviors corresponds to adapting the parameters of the respective sub-behavior. It was shown that the coupling of the behavior parameters resulted in a specific order of improvement.

By exploiting the coupling between BT parameters and the interpretable properties of a BT representation, DIPS improved on three challenges faced by RL algorithms: large number of required data points, dependency on reward-function specification and sensitivity to hyper-parameter settings.

Three simulated reality-gaps demonstrated the effectiveness and robustness of DIPS to different types of environmental adaptations and hyper-parameter settings. For all three simulated environments, DIPS found a policy with similar performance as the baseline environment performance. In the third environment, DIPS even found a higher performance policy within 135 evaluations. Note that 135 evaluations coincides with 27 actual data points, as the stochastic nature of the window fly-through task requires 5 evaluations to obtain a reasonable estimate.

Despite that DIPS has been applied to a single task only, we believe there are strong indications that DIPS generalizes to different tasks or behavior representation methods. These indications originate from the inherent coupling between sub-behaviors and environment experienced by embodied agents leveraging sensory-motor-coordination.

In this research DIPS was applied to a problem with the objective to reduce the influence of the reality gap. DIPS may, however, also be employed to update a behavior during operation because DIPS is not dependent on reward function crafting or external measurements and has an on-line nature.

X. Recommendations

This paper demonstrated the effectiveness of DIPS to cross the reality gap and improve on the challenges listed in the introduction. There are, however, many open questions and improvements which can be made to the algorithm. These recommendations can be segmented by proving the generalization properties of DIPS and improving the DIPS procedure.

The most important recommendations are related to the generalizing properties of DIPS to other behavior representation methods and tasks. This research only considered a BT to represent the robotic behavior and conducted a single experiment. We believe that the working principle –coupled nodes resulting in a single-peak performance landscape– extends to other parametric representation methods such as finite-state machines and small artificial neural networks. This also applies for different task environments. A real confirmation, of course, can only be given by testing DIPS in a real-world experiment.

We believe the presented version of DIPS can be improved in terms of convergence rate and performance. Figure 7 demonstrated the coupling of the node settings during the learning process. Possibly these couplings are more behavior-dependent than environment-dependent. As such, these couplings may be learned in simulation before-hand to make the exploration process more efficient and effective. Additionally, a learning

rate parameter can be added to the policy update-step which is initialized at > 1 and decays over the episodes to increase the update step-size. This, however, does add another hyper-parameter to the algorithm. Finally, the current DIPS implementation does not utilize previous experiences in form of data points. Likely convergence rate can be improved by adding a memory system which, for instance, updates and exploits coupling of the parameters or estimates the location of the performance-landscape peak.

References

[1]D. Floreano and R.J Wood, *Science, technology and the future of small autonomous drones*. In *Nature*, volume 521, 2015, pp. 460-466.

[2]K.Y.W. Scheper, S. Tijmons, C.C. de Visser and G.C.H.E. de Croon, *Behavior Trees for Evolutionary Robotics*. In *Artificial Life*, volume 22, 2016, pp. 23-48.

[3]G.C.H.E. de Croon, K.M.E. de Clercq, R. Ruijsink, B. Remes and C. de Wagter, *Design, Aerodynamics, and Vision-Based Control of the DelFly*. In *International Journal of Micro Air Vehicles*, volume 1, Issue 2, 2009, pp. 71 - 97.

[4]C. De Wagter, S. Tijmons, B. D. W. Remes, and G. C. H. E. de Croon, *Autonomous Flight of a 20-gram Flapping Wing MAV with a 4-gram Onboard Stereo Vision System*. In *IEEE International conference on robotics & automation*, 2014, pp. 4982-1987.

[5]R. S. Sutton and A.G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[6]J. Kober, A. Bagnell and J. Peters, *Reinforcement learning in robotics: A survey*. In *The International Journal of Robotics Research*, volume 32, Issue 11, 2013, pp. 1238- 1274.

[7]M.P. Deisenroth, G. Neumann and J.Peters, *A Survey on Policy Search for Robotics*. In *Foundations and Trends in Robotics*, volume 2, 2013, pp. 1-142.

[8]S. Koos, J.B. Mouret and S. Doncieux, *The transferability approach: Crossing the reality gap in evolutionary robotics*. In *IEEE Transactions on Evolutionary Computation*, volume 17, 2013, pp. 122 - 145.

[9]S. Nolfi, *Power and limits of reactive agents*. In *Neurocomputing*, volume 42, 2002, pp. 119-145.

[10]R.C. Arkin, *An Behavior Based Robotics*, Cambridge MA, USA, MIT Press, 1998.

[11]K.Y.W. Scheper and G.C.H.E. de Croon, *Abstraction, Sensory-Motor Coordination, and the Reality Gap in Evolutionary Robotics*. In *Artificial Life*, volume 23, 2, 2017, pp. 124-141.

[12]A.J. Champandard, *Behavior trees for next-gen game AI*. In *Game Developers Conference*, 2007, San Fransisco CA, pp. 1 - 96.

[13]R.G. Dromey *From requirements to design: Formalizing the key steps*. In *IEEE First International Conference on Software Engineering and Formal Methods*, 2003, Brisbane, pp. 2-11.

[14]V. Gullapalli, J.A. Franklin and H. Benbrahim, *Acquiring robot skills via reinforcement learning*. In *IEEE Control Systems*, volume 14, Issue 1, 1994, pp. 12-24.

[15]D. Floreano and F. Mondada, *Evolutionary neurocontrollers for autonomous mobile robots*. In *Neural Networks*, volume 11, Issue 7, 1998, pp. 1461-1478.

[16]N. Jakobi, P. Husbands and I. Harvey, *Noise and the reality gap: The use of simulation in evolutionary robotics*. In *Proceedings of the Third European Conference on Advances in Artificial Life*, 1995, pp. 704-720.

[17]S. Ross and J.A. Bagnell, *Agnostic system identification for model-based reinforcement learning*. In *Proceedings of the 29th Interational Conference on Machine Learning*, 2012, pp. 1703-1710.

[18]F. Saunders, E. Golden, R. White and J. Rife, *Experimental verification of soft-robot gaits evolved using a lumped dynamic model*. In *Robotica*, volume 1, 2006, pp. 1-8.

[19]J. Bongard and H. Lipson, *Once more unto the breach: Co-evolving a robot and its simulator*. In *Artificial Life*, volume 9, 2004, pp. 57-62.

[20]C. Hartland and N. Bredeche, *Evolutionary Robotics, Anticipation and the Reality Gap*. In *IEEE Transactions on Evolutionary Computation*, volume 17, 1, 2013, pp. 122-145.

[21]P. Abbeel, A. Coates, M. Quigley and A. Y. Ng, *An application of reinforcement learning to aerobatic helicopter flight*. In *Advances in Neural Information Processing Systems*, volume 19, 2007.

[22]S. Koos, J. Mouret, S. Doncieux, *The Transferability Approach: Crossing the Reality Gap in Evolutionary Robotics*. In *Proceedings of SAB*, volume 1, 2003, pp. 1640-1645.

[23]N. Kohl and P. Stone, *Policy Gradient Reinforcement Learning for Fast Quadrupedal Locomotion*. In *Proceedings of the IEEE International Conference on Robotics and Automation*, New Orleans LA, 2004, pp. 2619-2624.

[24]J. Peters and S. Schaal, *Reinforcement learning of motor skills with policy gradients*. In *Neural Networks*, volume 4, 2008, pp. 682-697.

[25]J. Kober, E. Oztop and J. Peters, *Reinforcement learning to adjust robot movements to new situations*. In *Proceedings of the 2010 Robotics: Science and Systems Conference*, 2010, pp. 835-853.

[26]S.F. Armanini, C.C. Visser G.C.H.E. de Croon and M. Mulder, *Time-varying model identification of flapping-wing vehicle dynamics using flight data*. In *Journal of Guidance, Control, and Dynamics*, volume 39, number 3, 2016, pp. 526-514.

[27]M. Amelink, M. Mulder and M.M. van Paassen, *Designing for Human-Automation Interaction: Abstraction Sophistication Analysis for UAV Control*. In *International Multi Conference of Engineers and Computer Scientist*, volume 1, 2008, Hong-Kong.

[28]S. Tijmons, G.C.H.E. de Croon, B.D.W. Remes, C. De Wagter and M. Mulder, *Obstacle Avoidance Strategy using Onboard Stereo Vision on a Flapping Wing MAV*. In *IEEE Transactions on Robotics*, volume 33, Issue 4, 2017, pp. 858-874.

Part II

Preliminary Research

Chapter 2

Policy Learning and Optimization in Robotics

This chapter provides a literature review on policy learning and optimization techniques for the field of robotics. In the first section the learning problem is introduced. The next section provides an overview of the three most commonly employed biologically-inspired learning algorithms in robotics: learning by evolution, by imitation and by reinforcement. This section also reasons that RL is the most suitable approach for this research. The last section provides an overview of RL and discusses recent developments relevant for this research.

2-1 The Policy Learning Problem

Arkin (1998) defined robotic learning as a process which produces changes within an agent that over time enable it to perform more effectively within its environment. This implies that learning a behavioral policy is an iterative process; different methods exist that allow the agent to perform learning. Within learning, some techniques also allow for *adaptation*. Adaptation implies that the robot learns by making adjustments to the current behavioral policy to be more attuned to its environment (Arkin, 1998). Learning can take place on many levels; we focus on learning how to act in a given situation, *behavioral learning*.

The *behavioral policy* determines how robots act in a given state. Robots continuously interact with their environment to achieve a goal by performing actions; at any given time the robot is situated in a state which describes its environment. The robot probes the environmental by taking measurements with onboard sensors to receive information about the state to determine which action to perform. For now, let us conciser a *behavioral policy* as a function which maps the state to an action; in the next chapter we will cover the definition of *behavior* more detailed. Visually the behavioral policy is depicted in Figure 2-1. Here the policy function β maps the state-space S to the action-space A.

The mapping function β can be interpreted as a function with a structure χ and a set of variables υ which together determine the output of $\beta(\chi, \upsilon)$. The structure —or representation—

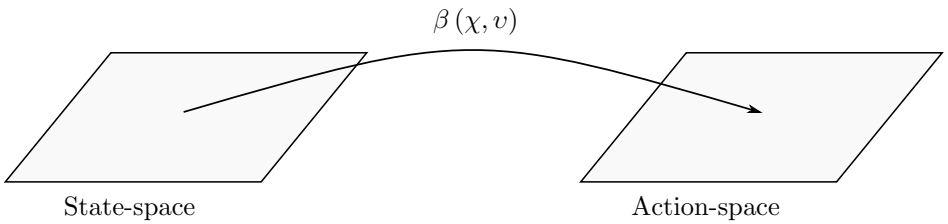


Figure 2-1: Visual representation of a policy function, $\beta(\chi, \upsilon)$, mapping the state-space S to the action-space A.

of $\beta(\chi, \upsilon)$ is discussed in the next chapter. This chapter concerns the optimization of the variables contained within the behavioral policy function. Essentially the goal of policy learning is to find a set of parameters, for a given representation, which allow the robot to optimally perform on its task. Not every learning method, however, is capable to find these parameters for a given structure; some methods can both learn the variables and the structure.

Traditionally, the robotic behavior is manually designed using dynamic models of both the task and the robotic platform. In many instances this is not possible for autonomous robots. Despite that the robotic task can be described at high level, a solution to this problem is often unknown. Therefore it is not possible to obtain capable traceable models for the task environment. In addition, full information on the dynamic environment is often not defined or impossible to obtain. Learning provides a framework to introduce and incorporate new knowledge into a system, generalize from past experience and come up with new, novel, solutions to a problem. Learning is therefore an essential part for (future) autonomous operation (Nelson et al., 2009; Kober et al., 2013; Floreano & Wood, 2015).

2-2 Overview of Policy Learning and Optimization Methods

The previous section introduced the concept of policy learning & adaptation and why these are required for autonomous operation. Learning techniques draw inspiration from biology; from how animals and insect learn and operate. In this review we conciser three types of learning: learning by evolution, learning by imitation and learning by reinforcement. The following sections provide a high-level overview of these methods and reasons why learning by reinforcement is most suited for this research.

2-2-1 Learning by Evolution

Learning by evolution is a metaheuristic optimization method which mimics the process of evolution (Floreano & Mattiussi, 2008). *Evolutionary Robotics* is the field of research which applies evolutionary computation techniques to develop controllers or behavioral polices for autonomous robotics.

Evolutionary algorithms use a population of randomly initialized members to incrementally converge to a (local) optimal solution by recombining members of the population and mutating their offspring. The performance of individual is measured using a *fitness function*. A fitness function is responsible for determining which solutions in the population are more optimal

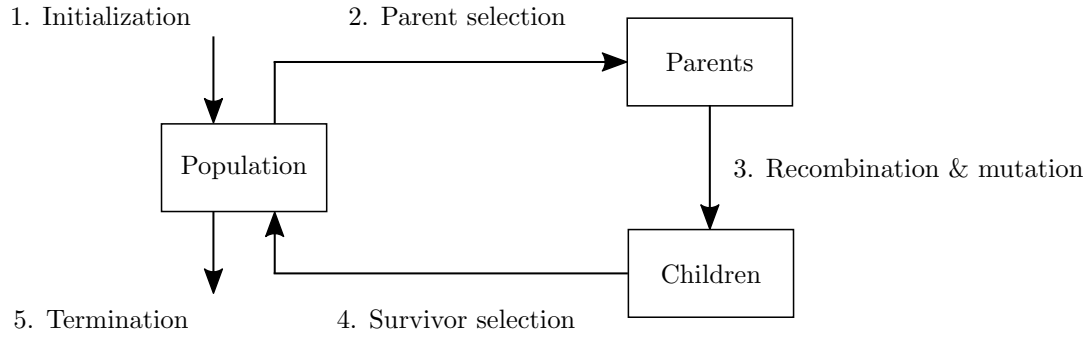


Figure 2-2: Block diagram of the evolutionary algorithm, based on Floreano & Mattiussi (2008).

for the given ask. The evolutionary algorithm can be described by the five steps on the next page; a schematic overview is provided in Figure 2-2. Note that there are numerous methods for every step and that Figure 2-2 serves as a general scheme.

1. **Initialization:** initialize a population with random members.
2. **Parent selection:** evaluate population members and select parents for the next generation. This process often has stochastic nature where members with high fitness have a greater probability of becoming a parent. Members with lower fitness can become parent, albeit with a lower probability, to ensure gene diversity and prevent the process from converging too early to local minima.
3. **Recombination and mutation:** recombine two parents using *crossover* and apply small mutations to form children with a new gene set.
4. **Survivor selection:** evaluate the offspring and select individuals of both parents and children to form a new population. This process is often deterministic where selection can depend on member properties, i.e. age and fitness.
5. **Termination:** select the best individual when a stopping criteria is met, e.g. maximum number of generations is reached or an individual has a minimum fitness.

Steps two, three and four are repeated until the a stopping criteria is met. In evolutionary robotics every member of the population is evaluated on the task. This implies that the controller, encoded in the individual, is trialed either in a real-world environment or in simulation to measure its fitness.

An advantage of evolutionary learning methods is that they are able optimize unrelated parameters and therefore also the structure of the behavioral policy function $\beta(\chi, \upsilon)$ due to their metaheuristic nature; this will be further discussed in the next chapter (Stanley & Miikkulainen, 2002; Scheper et al., 2016). Furthermore, no assumption on the state has to be made; for some problems —especially partially observable Markov Decision Processes— evolutionary learning methods provide better results than other learning algorithms (Croon et al., 2005). Finally, when using an ANN as structure for $\beta(\chi, \upsilon)$, an evolutionary learning approach is very good at avoiding local maxima Yao (1999). A large disadvantage is that evolutionary algorithms cannot adapt the behavior during operation.

2-2-2 Learning by Imitation

Learning by imitation allows a robot to learn from demonstrations or examples provided by a teacher (Argall et al., 2009). An example, or data-point, can be seen as a state-action pair; the data-set used to learn from is a sequence of state-action pairs. The goal of learning from imitation is then to construct a policy which best mimics the the behavioral policy of the teacher. This implies that the behavioral policy of the robot will likely be very similar to that of the teacher. Learning by imitation distinguishes itself from other learning techniques in that the robot does not learn from self-obtained experience.

Learning by imitation can be interpreted as a form of *supervised learning*. Supervised learning is a form of Machine Learning (ML) where the agent is presented with labeled data. In general, learning a behavioral policy through imitation can therefore take two approaches: *classification* or *regression*. Both approaches are well studied in the field of statistics and supervised ML and can make use of a variety optimization methods such as *gradient-descent* and *k-means clustering*, depending on the policy structure χ.

Classification attempts to categorize the states in discrete classes and map these states to an action class. A nice example is presented in Chernova & Veloso (2008) where a multi-robot ball sorting task is learned from demonstration.

Regression approaches map states to continuous actions, generalizing over examples. Regression approaches are often used for low-level learning, such as motor control, and classification methods for coordinating high-level behaviors. In (Pomerleau, 1991), however, an ANN learned to drive a simulated van in different conditions from teacher demonstration.

Learning by imitation is a fast and accurate method to learn a policy. A disadvantage is that the robot needs a teacher to collect data. This implies that the teacher is required to have sufficient knowledge on the task environment and preferred approach; in uncertain environments this is often not the case. Additionally, the policy is only defined near the states encountered and the actions taken, not necessarily in the entire state-action space.

2-2-3 Learning by Reinforcement

Learning by reinforcement is a type of ML where an agent learns which action to select by receiving feedback in the form of a reward signal (R. S. Sutton & Barto, 1998). The agent learns by trial-and-error while automatically exploring the environment. The goal of RL is to find a policy which optimizes the received reward.

To start a RL session, a reward function is defined and the agent is initialized with a random or zero policy. The reward function returns scalar *rewards* based on the quality of the action selected. The agent then selects an action and receives a new state and a reward from the environment. The goal for the agent is to discover relations between states, actions and received rewards. After receiving a reward, or after an episode, the agent updates its policy. Learning continues until an agent receives a minimal score or a certain maximum of episodes have passed. A block-diagram of the RL process is provided in Figure 2-3.

Learning by reinforcement has three main advantages for the field of robotics. First, RL can learn policies through experience without the need of a-priori knowledge on the task environment. Second, the directed and automated exploration and direct policy update allow

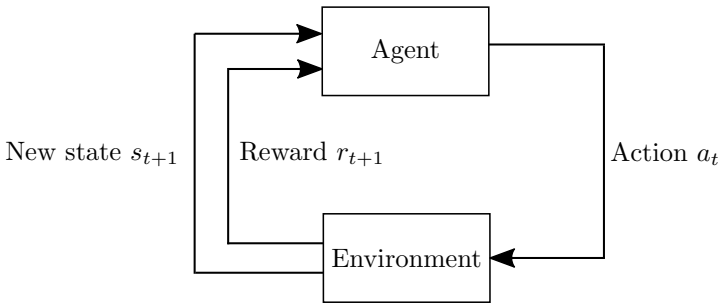


Figure 2-3: Block diagram of the reinforcement learning process.

for on-line learning to adapt a policy to a different environment. Third, RL can generalize from experience by using function approximation techniques, Section 2-3-2. A disadvantage of RL is that the algorithm is sensitive to the large number of hyper parameters. Additionally, the number of variables can scale exponentially with the task complexity.

2-2-4 A case for Reinforcement Learning

The previous sections elaborated on different robotic learning techniques. Given the research motivation to reduce the effect of the reality gap we can set a number of requirements for the learning algorithm.

1. **The algorithm must facilitate on-line learning.** It is essential that the technique can optimize the behavioral policy, $\beta(\chi, \upsilon)$ during operation to account for discrepancies in the simulation model.

2. **The algorithm must not require significant a-priori knowledge.** Generally, it is unknown how the behavioral policy must be adapted to reduce the reality gap. Therefore the algorithm must be able to perform with minimal a-priori knowledge of the task environment and solution.

When comparing the above requirements to the advantages and disadvantages of the different learning techniques, it becomes clear that RL meets the two main requirements. Learning by evolution falls short on on-line adaptation methods and learning by imitation is highly dependent on a-priori knowledge. Therefore it is argued that RL is the best learning method for this research.

2-3 Reinforcement Learning as Policy Learner in Robotics

Section 2-2 discussed several learning methods and concluded that RL is the best learning method for this research. This section elaborates on RL by briefly summarizing the background of RL, presenting a concise overview of RL and discussing relevant recent developments.

2-3-1 Background

RL has been developing by combining two branches of research (R. S. Sutton & Barto, 1998). The first branch concerns learning by trial-and-error which originated from the field of psychology and animal learning. The second branch is the field of optimal control; researches in this branch already developed solutions using *value* and dynamic programming, however did not yet utilize the concept of learning. The combination of both branches produced the field of RL.

RL is now generally seen as a form of ML, being an intermediate version of *supervised-* and *unsupervised learning*. Within this field of research RL has achieved many impressive feats, ranging from investment portfolio management to recently defeating the world-champion in a game of GO, a board game with 10^{170}(!) different board positions (Silver et al., 2016).

The field of robotics differs from traditional, well-studied, RL domains; these differences originate from the real-world operating environment. A robotic task is highly-dimensional, the measurements are noisy and obtaining experience is a tedious and sometimes dangerous process. Despite these challenges RL has achieved many successful robotic applications, however, also has a large number of open questions (Kober et al., 2013). We hope that this research can contribute to the development of the field such that intelligent robots can become a part of our future.

2-3-2 Overview

In this section the basics of RL for robotics is presented to provide an overview of the learning technique and the successful implementations. This section is based on the work of R. S. Sutton & Barto (1998) unless otherwise specified.

Markov Decision Process

A fundamental assumption in classical RL states that the system can be described by a Markov Decision Process (MDP). The system can then be modeled with a set of states S, a set of actions A, a set of transition probabilities T and a set of rewards R. The *Markov Property* states that the next state, s', only depends on the current state s and the selected action a.

By applying the Markov Property to model a system, the previous states and actions do not have to be considered. This drastically reduces the number of variables which have to be taken into account and therefore simplifies the learning process. For robotic applications, however, the Markov Property does not hold as the world is inherently uncertain and a robot is often unable to determine the current state of the system; the state is partially observable. Therefore most robotic applications rely on a Partially-Observable Markov Decision Process (POMDP) to describe their environment. For the rest of this section, we will consider MDP.

Goal of Reinforcement Learning

The goal of RL is to find an optimal policy $\pi(s)$ for the optimization variables in the function $\beta(\chi, \upsilon)$ for mapping states to actions. If the robot is controlled directly by a RL policy, the

policy structure often has a tabular- or function approximation representation, Section 2-3-2. A policy can be seen as a rule for selecting the action in the current state, the *actor*. A policy is considered optimal if it obtains a maximal cumulative reward, J_t, during operation. Because robots operate in uncertain domains, a policy maximizes the expected cumulative reward, Equation 2-1.

$$J_t = E\left[\sum_{t=0}^{T} R_t\right] \tag{2-1}$$

The above equation assumes that a task has a finite time window T and that the number of time steps in this window is known. Many applications, however, are not time constrained. Therefore, the *discounted reward* is often applied to RL problems, Equation 2-2. In this relation G_t is the discounted reward over a predetermined set of time steps T, or the *return*, and γ is the *discount factor* which affects how much the future rewards are taken into account.

$$G_t = E\left[\sum_{t=0}^{T} \gamma^t R_t\right] \tag{2-2}$$

The rewards that an agent receives are formulated in a *reward function*. A reward function is specific to the problem and must capture the goal of the task. The reward function can be formulated according to a MDP for a state $R = R(s)$, a state-action pair or the state transition $R = R(s, a)$ or $R = R(s, a, s')$. Other formulations, such as event-based reward functions, have yielded successful applications in robotics.

The first goal of learning is to obtain an optimal policy. During learning, RL algorithms iteratively improve the policy performance. To obtain an optimal policy, the agent is required to *explore* unseen or unvisited states and actions in order to gain information on the system dynamics and received rewards. Additionally, the agent has to *exploit* the information it has to continue improving upon the current policy. A secondary goal is therefore to obtain the highest possible reward during learning . Balancing both goals is a difficult problem to solve; the *exploration versus exploitation trade-off* has a large influence on the performance of the final policy.

Principle of Optimality

In the previous section a policy was defined as a rule which determines the action for a particular state. If an agent acts according to a fixed policy over many episodes, the agent is expected to receive a certain reward, see Equations 2-1 and 2-2. This expected reward is defined as the *value*. If the agent is initialized in the same state over all episodes, the value of the policy for that state s can be calculated. This is equal to the expected cumulative reward for following policy π when initialized at state s, $V^\pi(s)$. The value is a measure for the quality of the actions, the *critic*.

By assuming the learning problem has the Markov Property, there exists at least one optimal policy, π^*, that maximizes the expected value for all states. All optimal policies share the *Principle of Optimality* and can be captured by the recursive Bellman equation:

$$V^{\pi^*}(s) = \max_a \sum_{s'} P\left(s'|s,a\right)\left[R\left(s'|s,a\right) + V^{\pi^*}(s')\right] \tag{2-3}$$

Intuitively, the above relation formulates the value of the current state as the expected reward for the next time step plus the value of the next state. The $\max_a$ ensures that the best action is selected; Equation 2-3 can also be rewritten without this maximization. This yields $\mathrm{Q}(s,a)$, the expected value for state s when selecting action a, Equation 2-4. An advantage of using $Q(s,a)$ instead of $V(s)$ is that the state-action value contains explicit information about the consequences of the specified action, the *action-value*.

$$\begin{aligned} Q^{\pi^*}(s,a) &= \sum_{s'} P\left(s'|s,a\right)\left[R\left(s'|s,a\right) + V^{\pi^*}(s')\right] \\ &= \sum_{s'} P\left(s'|s,a\right)\left[R\left(s'|s,a\right) + \max_{a'} Q^{\pi^*}(s',a')\right] \end{aligned} \tag{2-4}$$

Many RL techniques attempt to solve the relations of Equation 2-3 and 2-4. These methods are named *Value Function Methods* as they first approximate the values for each state or state-action pair, and then extract the policy the action with the highest value for the given state. These methods are discussed in Section 2-3-2. Other RL techniques, known as *Policy Search Methods*, directly search for policies without consulting state-action values. An overview of Policy Search Methods is provided in Section 2-3-2.

Value Function Methods

Value Function Methods can be segmented into three types: Dynamic Programming Methods, Monte-Carlo Methods and Temporal Difference Methods. This section briefly discusses these three types.

Dynamic Programming Methods iteratively update the current policy and value function to converge to a final policy. There are two approaches to this iteration sequence: policy iteration and value iteration.
In the two-step policy iteration approach, the policy is initialized randomly. The agent then acts according to this policy to find the corresponding value function by visiting every state many times —in theory, an infinite amount is required— and updating the value estimate of the states, Equation 2-3, the *policy evaluation* step. When the value estimate converges to a fixed value, the policy is updated by selecting the action with the highest value for every state, the *policy improvement* step. The final, optimal, policy is obtained when there are no changes to the policy during the policy improvement step. A value iteration approach is very similar to the policy iteration approach. In value iteration, however, the policy is updated directly after every state visit.
Dynamic Programming Methods require the transition probabilities between states to iterate over the state-space. The transition probabilities essentially model the system, therefore these methods are also named *model-based* approaches. The employed model can be learned from data before or, incrementally, during training.

Monte-Carlo Methods attempt to reduce the number of required iterations in the policy evaluation step of dynamic programming methods by sampling from experience. The value, $V(s)$, is estimated through the mean return which is the cumulative return, G_t, per state divided by the number of visits for that state. Since Monte-Carlo Methods learn from experience they do not require transition probabilities or reward functions; they are *model-free*. One disadvantage of Monte Carlo Methods is that they are only applicable to episodic task as they use the return to estimate the value function, Equation 2-2.

Temporal Difference Methods learn directly from experience and are model-free like Monte-Carlo Methods, however, can also be applied to non-episodic tasks. Temporal Difference Methods update *temporal errors* to estimate the value of a state. A temporal error is the difference between the current state value and the new estimated value for the corresponding state. The new estimated value for a state solely depends on the value of the of the next state and the received reward during the transition. Iteratively, the value function is then formulated according to Equation 2-5.

$$V'(s) = V(s) + \alpha \left[R' + \gamma V\left(s'\right) - V\left(s\right)\right] \tag{2-5}$$

Here $V(s)$ is the old estimate for state s, $V(s')$ is the value of the new state after taking action a, $V'(s)$ is the new estimated value for state s and α is the *learning rate*. The learning rate defines the step size for updating the current value towards reducing the temporal error. Equation 2-5 can also be formulated in a state-action value, Equation 2-6.

$$Q'(s,a) = Q(s,a) + \alpha \left[R' + \gamma Q\left(s',a'\right) - Q\left(s,a\right)\right] \tag{2-6}$$

Again, $Q(s,a)$ is the old estimate for the state-action pair s and a, $Q(s',a')$ is the value of the new state-action pair after taking action a in state s and $Q'(s,a)$ s the new estimated value for the current state-action pair.

An agent is said to be learning *on-policy* if it uses the current policy to select the actions. For temporal difference and Monte-Carlo methods an agent can also learn by selecting other actions than their policy dictates, *off-policy* learning. With off-policy learning an agent can learn from observing other agents, re-use experience generated from old policies and learn about the optimal policy while exploring different actions. During off-policy learning, an agent can select an action which does not coincide with their policy, determined by an *exploration function*. Q-learning is a widely used off-policy Temporal Difference Method. To ensure that the agent learns correct Q-values, Equation 2-6 is adapted slightly such that the Q-value of the new state-action pair always represents the Q-value for the optimal action. The Q-learning update rule is provided in Equation 2-7.

The main advantage of TD-Learning is that is only gains experience in relevant states that it visits (as opposed to DP) and it learns quickly by bootstrapping (as opposed to MC).

$$Q'(s,a) = Q(s,a) + \alpha \left[R' + \gamma \max_{a'} Q\left(s',a'\right) - Q\left(s,a\right)\right] \tag{2-7}$$

Value Function Approximation

Until now the value function has been presented in a *look-up table*. Every state s has an entry in $V(s)$ and every state-action pair can be formulated in a $Q(s, a)$-matrix, or *Q-table*. For large MDP, which is typically the case the in robotic applications, there are too many states and actions to store in a table-formulation. Additionally, learning of these large MDPs will be very slow as every state has to be visited a large number of times. A solution to this problem is to *estimate* the the value function using function approximation.

Function approximation methods are based on data samples which are collected by the agent during interaction with the environment; his also allows for generalization from visited states to unvisited states. Value function approximation therefore combines learning from reinforcement with learning from imitation. Coarsely, two types of function approximation methods exist: parametric and non-parametric (Kober et al., 2013). Both methods are discussed briefly below.

Parametric Function Approximation Methods formulate the value or action-value function with a finite set of parameters, θ, that approximates the sample data as close as possible. Generally, the parametric function approximation method can be formulated according to Equation 2-8.

$$\hat{V}(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + ... + \theta_n f_n(s) \tag{2-8}$$

In Equation 2-8, $\hat{V}(s)$ is the approximate value for state s, θ_1 to θ_n is the finite set function approximation arguments and $f_1(s)$ to $f_n(s)$ are the features of the state s. These features can be defined by hand using a-priori knowledge or learned. The most frequently employed parametric function approximation methods are linear combination of features (see above) and Neural Networks. ANNs learn features by extracting information from the state in the hidden layers and linearly combine these features in the output layer; a more detailed description in provided in Section 3-2-1.

Non-Parametric Function Approximation Methods generalize states and actions to unseen states and actions, by discretizing the state-action space. Example methods are *decision trees*, Section 3-2-3, and neighboring cell methods such as tile coding. A problem with Non-Parametric Function Approximation Methods is that they lack scalability in higher dimensional problems.

Policy Search Methods

Section 2-3-2 stated that an optimal policy can be found by both Value Function Methods and Policy Search Methods. The previous two sections focused on first group; this section provides an introduction to the second group of methods.

Policy Search Methods search directly for an optimal policy instead of through value functions. Given a policy $\pi_\theta(s, a)$ with the parameters θ, the goal is to directly find the best combination of parameters θ. Policy Search Methods often optimize locally around a current policy, therefore the goal can be formulated as the iterative process of Equation 2-9.

$$\theta_{i+1} = \theta_i + \Delta\theta_i \tag{2-9}$$

The success of the policy update depends on determining $\Delta\theta$, the *hill-climb* step. In general, methods can be segmented in Gradient Methods and Non-Gradient Methods (Kohl & Stone, 2004; Abbeel et al., 2007; Kober et al., 2013).

Policy Gradient Methods search for a maximum of the policy objective function —which is often the average reward function over a number of time-steps— by moving towards the gradient of the policy objective function. The policy gradient is frequently calculated using finite differences or likelihood ratios which require the policy to be differentiable. Likelihood ratio methods often require an estimate of the action-value function; therefore employs both an actor and a critic. The critic updates the action-value function parameters and the actor updates the policy parameters in the direct suggested by the critic Kohl & Stone (2004); Kormushev et al. (2013).

Non-Gradient Methods do not rely on a policy gradient to formulate their policy update step and are based on expectation-maximization methods. An expectation-maximization algorithm estimates the next policy parameters θ_{i+1} based on the expected return obtained from previous policy parameter settings (Abbeel et al., 2007; Kormushev et al., 2013).

Policy search methods have several advantages for robotics (Kober et al., 2013). First, Policy Search Methods allow for a natural integration of a-priori knowledge to the search domain. Second, optimal policies often have fewer variables than optimal value functions or -approximation methods. Finally policy search methods can often find better policies by local search around the current policy. A large disadvantage is that Policy Search Methods can be more difficult to implement and tune.

Combining Value Function and Policy Search Methods

Value function methods, as discussed in Section 2-3-2, can be interpreted as *critic-only methods*. The policy of critic-only methods is simply constructed by acting greedily with respect to the state value. Similarly, policy search methods are sometimes refereed to as *actor-only methods*. By directly searching for an optimal policy the concept state, or state-action, values have no meaning, from which the actor-only name is derived. Both methods have their benefits as discussed in the preceding paragraphs. A third method to solve MDPs is obtained by combining both methods, so-called *actor-critic methods*.

Actor-critic methods explicitly separate the policy and the value function, or the actor and the critic (R. Barto et al., 1983). Both the actor and the critic often employ a function approximator as representation. The actor determines the action and the critic values the action of the critic. The value of the action is then used to incrementally update the the parameters of the actor towards the gradient of improved performance. The largest benefit of this approach originates from the beneficial convergence properties as compared to critic-only methods using function approximation —critic-only methods using function approximates converge is a limited number of cases (Greensmith et al., 2004). Actor-critic methods, however, are less well suited to highly dynamic environments due to the rapidly changing conditions.

2-3-3 Recent developments

This section highlights some of the relevant recent advances in the field of RL applied to robotics, specifically we will look at the following subjects:

1. **Hierarchical reinforcement learning**. These developments discuss how RL can be applied to learn high-level sequential tasks consisting of sub-tasks. Since Behavior-Based Robotics (BBR) also employ sub-tasks, these method can possibly complement each other.

2. **On-line policy updating methods**. These developments discuss how RL can be employed to update the policy in a real-world environment. The goal of these researches is closely related to the goal of this research, possibly allowing us to draw inspiration from their methods.

3. **Relational reinforcement learning**. This form of RL encodes the learned policy in the form of a discrete structure. Although this form of RL is relatively new, it could potentially directly learn a simple behavior representation method such as discussed in the next chapter.

Hierarchical Reinforcement Learning

Robots operate in a real-world domain with a large —possibly infinite— state-action space; often the objective or task can be decomposed into components or sub-tasks. Hierarchical Reinforcement Learning (HRL) introduces a form of *state-abstraction* which reduces the size of the state-action space by only considering the relevant states and actions for the individual sub-tasks. Generally, there are two types of HRL: *meta-actions* and *options* (R. Sutton et al., 1999; Dietterich, 2000; A. G. Barto & Mahadevan, 2003). A meta-actions is a high-level action which activates a series of lower-level actions. Options are *temporally-extended* actions which extend multiple time steps.

Both types of HRL consists of hierarchically structured sub-tasks with different learning spaces which interact with each other, essentially trading exponential growth for linear growth of the state-action space. Additionally, these approaches allow the learning process to be guided by adding a-priori knowledge into the learning process by defining the options or sub-tasks (Dietterich, 2000). This is, however, not required as HRL can also learn which lower-level actions are part of the meta-actions or options.

In Hart & Grupen (2011) a meta-action approach to HRL is employed to learn bi-manual manipulation tasks, similar as how a *Hierarchy of Abstract Machines* segments the policy space in policies for sub-tasks (Parr & Russell, 1998). Their approach assembles these policies for the sub-tasks in a finite state machine-like transition diagram. It is concluded that the sub-policies can be assembled hierarchically to support incremental and cumulative learning of the robotic task.

An interesting application of HRL using an options framework is presented in Konidaris et al. (2012), where a skill tree is learned from human demonstrations and optimized using HRL. At the core of their method is an options framework which segments a chain of actions from a demonstration directory. These chains are later merged to form a skill tree. They successfully implemented their method on a robotic platform which had to enter a room and push on a certain panel. The obtained skill tree provided an overview and abstraction of the different segments in the robotic task.

On-line Policy Updating Methods

On-line policy updating is an essential part of implementing a RL policy on a robotic platform. On-line policy updating belongs to a robot-in-the-loop approach to reducing the effect of the reality gap, as discussed in the Introduction. A problem with on-line learning is the safety of the robot and its environment. During on-line learning the policy obtained in simulation is first tested on a robotic platform; this brings two possible risks. First, it is unknown how the policy will behave on the robot, as the robotic platform always differs from the simulated platform. Second, during learning the robot has to explore, taking new actions, of which the outcome is uncertain. This section highlights some approaches to tacking these problems.

In Molenkamp et al. (2017) a RL policy is learned to select controllers for a quadrotor performing aggressive maneuvers. The RL policy was stored in a Q-table. Their method consisted of a two-step approach: first learn a policy in simulation and then adapt the policy on-line. The second step did not take place on an actual drone but on a simulated drone with increased weight. They showed that the success of on-line learning is highly dependent on the quality of the simulation model and that successful on-line learning required mostly exploitation of the off-line policy while learning at a much lower rate. The simplicity of their approach is very appealing, to use a Q-matrix instead of an action-value approximation method.

A different approach is presented in Schonebaum et al. (2017). In their research they investigated the effect of using human demonstrations for exploration. The RL policy is stored in a type of function approximation called a *basis function*. They proposed an algorithm, named On-line Least Squares Policy Iteration, which combines on-line and off-line Least Squares Policy Iteration methods by initializing the on-line policy iteration algorithm with the policy found by the off-line policy iteration algorithm (Lagoudakis & Parr, 2003; Busoniu et al., 2010). The Least Squares Policy Iteration method can be interpreted as a form of batch-learning; when transferring the policy to the on-line algorithms the off-line data points are also transferred such that the new algorithm does not have to start without any data. The proposed algorithm was able to quickly learn from human demonstration samples to initialize a policy on-line. The policy derived from the demonstrations adapted to environmental changes and was improved on-line. This research demonstrated that human demonstrations can improve the performance of RL learning and safe exploration and that the algorithm is able to adapt to a changing environment. Furthermore their approach was model-free, providing many benefits for difficult to model UAVs such as the DelFly (Croon et al., 2012). A limitation in their approach was that the best performance was achieved when the human demonstrator has a good understanding of the task.

Mannucci et al. (2017) are the first to explore a HRL approach to on-line safe RL. In this article it is shown that a hierarchical structure, consisting of cascaded *managers*, can increase safety during exploration in a real-world environment by leveraging a-priori knowledge of the model and off-line learning of the lower-level tasks. Therefore the lower-level tasks have less uncertainty about the real-world environment. The managerial hierarchy leverages the simplicity of these lower-level tasks to facilitate safe on-line learning as the real-world uncertainty is concentrated in the higher managerial levels. On-line learning successfully learned a high performing navigation task while avoiding obstacles.

Relational Reinforcement Learning

Relational Reinforcement Learning (RRL) is a form of RL which discretizes the state-action space by grouping states and actions with similar properties and mapping them to a Q-value (Dzeroski et al., 2001). The relational representation enables the re-use of experience on related problems or situations. The Q-values are often represented by logical regression structures such as decision trees.

RRL consists of a two-step approach. In the first step a representation of the value function is learned in the form of a logical regression tree, called a Q-tree (Dzeroski et al., 2001). This Q-tree replaces the Q-table in classical RL. The second step uses the Q-tree to learn a classification tree, called a P-tree, which maps actions as optimal or sub-optimal. The P-tree representation is usually more compact than a Q-table, as it does not consider different levels of sub-optimality. Therefore, the second step incrementally constructs a tree with first-order logic to analyze the current state, mapping the state to an action in the tree leafs.

Taylor & Stone (2009) point out that RRL methods can be useful for *policy transfer problems*. The goal of transfer learning problems is to use experience of one task to improve learning in a related, but slightly different task. This problem type is related to the motivation of this research as transfer learning problems often consider different task-environments.

In Martinez et al. (2015) RRL is employed to learn from demonstrations with the goal to require as few RL episodes and human demonstrations as possible. They conclude that the generalization capabilities of RRL significantly reduced the number of exploration actions required, therefore speeding up the learning process. Possibly we can draw inspiration from their method to reduce the required on-line exploration episodes.

Chapter 3

Representation Methods for Intelligent Robotic Behavior

Chapter 2-1 introduced the formal definition of behavior as the (S, A, β) triple and reviewed several methods to optimize the variables υ in the mapping function $\beta(\chi, \upsilon)$. This chapter describes the different methods to represent the mapping function $\beta(\chi, \upsilon)$. First the definition of intelligent robotic behavior is refined to better agree with the research objective. Section 3-2 provides an overview of the different behavior representation methods and reasons that a BT is the most suited framework for this research. Finally Section 3-2-5 aims to provide a concise overview of BTs and their applications in robotics.

3-1 Definition of Representation Methods for Intelligent Robotic Behavior

The title of this chapter is composed out of three components, namely *representation methods*, *intelligence* and *robotic behavior*. The goal of this section is to provide a definition of all three components which will be employed throughout this research. We define the terms in reverse order as this provides the most natural flow of the definitions.

3-1-1 What is Robotic Behavior?

Robotic actions can be interpreted as interactions with the environment. Figure 3-1 shows a block diagram of the control sequence for action selection. Robotic- and software agents are embodied within their environment if the environment is dynamic, can be changed by the actions of the agent and can be perceived by the onboard sensors (Brooks, 1986; Anderson, 2003; Pfeifer et al., 2007). Robots are therefore always embodied in their environment. From an embodied viewpoint behavior emerges from the interaction between agent and environment. Following this viewpoint, we define robotic behavior as the sequence of actions selected by an agent to achieve a certain goal.

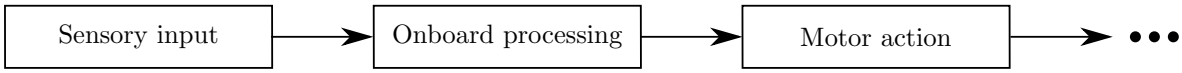


Figure 3-1: Block diagram of sensory-motor coordination; triple dots indicate initialization of new control sequence.

If an agent is embodied, its actions directly effect the state of the environment. The sensory input received by the onboard sensors is therefore influenced by the selected actions. (Nolfi, 2002) entitled this effect as the *sensory-motor coordination*. In his research he showed how the sensory-motor coordination process can be exploited to prevent sensor aliasing, to simplify problems and to leverage emerging behaviors.

Coarsely, robotic behavior can be separated in two types of controllers: *reactive-* and *pro-active controllers* (Brooks, 1991; Mataric, 1998; Nolfi, 2002). Reactive controllers select an action without an internal state and are solely dependent on the sensory input. Pro-active controllers compute the best action dependent on an internal state and the sensory input. As a result, reactive controllers always select the same action when provided with the same sensory input. On the contrary, pro-active controllers can act differently to equivalent sensory inputs if the internal state differs. An internal state can be viewed as the necessary information about the environment to allow the agent to achieve its goal. Pro-active controllers can often express more complex appearing behavior but require more computation and memory. On the other hand, reactive controllers have quick reaction times and have solved complex tasks on numerous occasions (Nolfi, 2002).

3-1-2 When is Behavior Intelligent?

The definition of robotic intelligence has been subjected to a large amount of debate without a clear winner (Brooks, 1991; Winston, 1992; Russel & Subramanian, 1995). One common factor in all arguments is that the agent is required to be central, sensing and acting by itself, such that it operates autonomously. Following the definition of Brooks (1991), in this research robotic intelligence is considered as an emergence from the interactions between simple individual sub-behaviors. This translates to considering an agent intelligent if it can act rationally with the available resources. This coincides with the definition of *bounded-optimality* in Artificial Intelligence (AI) formulated by Russel & Subramanian (1995).

Within the field of robotics, BBR concerns the development of intelligent robotic behavior by combining individual sub-behaviors (Brooks, 1991). In contrast to other behavior design approaches, such as classical AI, sub-behaviors generally do not require a world model. This makes a BBR approach particularly useful for small robotic platforms as sub-behaviors require less computation, lowering the on-board computational requires and reducing computation time. As the application of this research mainly considers small Unmanned Aerial Vehicles (UAVs), we will keep to the BBR philosophy when considering intelligent robotic behavior.

3-1-3 What is a Behavior Representation?

Behavior representation methods are control architectures which coordinate the activation of different actions or sub-behaviors. A behavior representation method supplies structure, im-

poses constraints on the way robots are controlled and functions as a framework for implementation. In a BBR approach the robot controller consists of a collaboration of sub-behaviors, each which achieves or maintains a specific goal (Brooks, 1991; Arkin, 1998). These sub-behaviors can be explicitly defined or be emergent from the representational complexity such as ANNs. Behavior representations activate sub-behaviors as a response to a *stimulus*, which can be either internal or external, and outputs an action such as depicted in Figure 3-1. The sequence and priority in which different sub-behaviors are activated is a key issue which every behavioral representation method solves in its own way.

When considering sub-behaviors, the behavior representation can function as a coordination system. The most famous coordination system is Brooks *subsumption architecture* (Brooks, 1991). The subsumption architecture hierarchically orders the sub-behavior; behaviors of a higher order can subsume lower order behaviors. Working principles of other popular behavior coordination or behavior representation methods are generally based on a voting system, behavior fusion or action-selection coordination. The next section elaborates on these coordination types by providing an overview of the most frequently employed behavior representation methods.

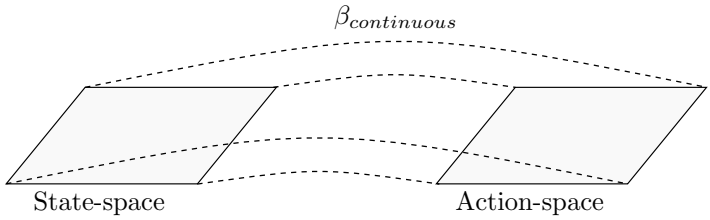
3-2 Overview of Behavioral Representation Methods in Robotics

Intelligent robotic behavior can be represented by a variety of methods. As stated previously, most traditional representation methods are based on the subsumption architecture approach proposed by Brooks (1986). This approach employs (reactive) modules to build layers of competence, each which expresses a different behavior —or sub-behaviors—, every layer is often constructed using FSMs, Section 3-2-4. Currently, most autonomous embodied agents are encoded using more complicated biologically inspired structures such as ANNs. These methods differ significantly, therefore a distinction is made in this report between two main approaches: *continuous* and *discretized* behavioral representation.

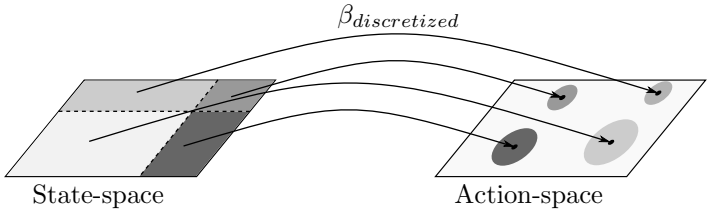
Continuous representation methods map all states in S to an action a in A; such that every state s often —but not necessarily— has a unique action. Discretized representation methods divide the state-space into discrete segments —often by condition-type statements— and map each segment to a specific action (in case of action coordination) or action-set (in case of behavior-coordination) in A. Discretized methods therefore generalize from the state segments and only consider a subset of actions in A. A visual representation of both continuous- and discretized behavior representation methods is provided in Figures 3-2a and 3-2b. This section provides an overview of the most frequently employed representation methods for embodied agents and robotics.

3-2-1 Continuous: Artificial Neural Network

ANNs are biological-inspired computational models to process information (Floreano & Mattiussi, 2008). The interconnected structure of an ANN is derived from biological nervous-systems. Every node in the ANN structure is referred to as a *neuron*; the *synapses* connect the neurons to form a network. An ANN consist of three types of neurons: input neurons, hidden neurons and output neurons. Figure 3-3 displays a simple network structure.



(a) Continuous behavior representation methods: every state maps to an action.



(b) Discretized behavior representation methods: states are clustered and map to an action or action set.

Figure 3-2: Visual representation of Continuous- and Discretized behavior representation mapping functions β. Please note that this is a simplified visualization and is intended to aid in understanding the segmentation.

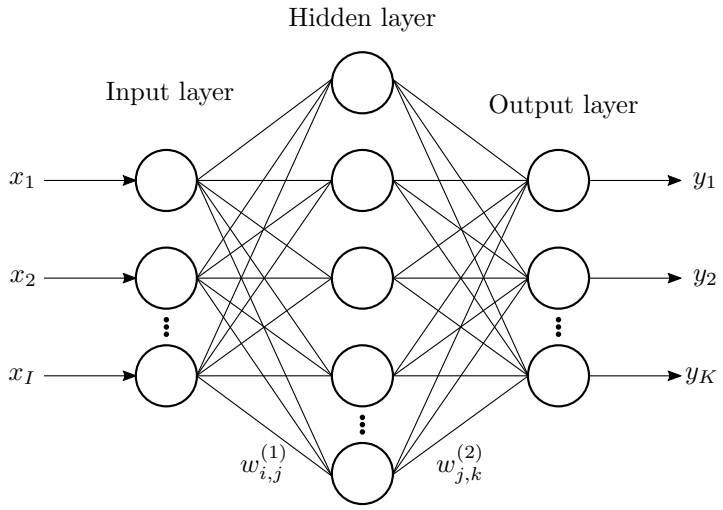


Figure 3-3: Visual representation of an ANN structure consisting of I input neurons, J hidden neurons and K output neurons.

An ANN processes information by combining the output of the neurons according to the network structure. Each neuron has a single input and output. An *activation function* transforms the neuron input to the neuron output. In general, the input of a neuron consists of the weighed sum of outputs from the neurons in the previous layer. The weights depend on the synapses connecting the neurons and must be learned, see Chapter 2. For the first layer, the input is simply the input data; the output of the ANN is the output generated by the neurons in the final layer.

ANNs can be used as a behavior representation method to map states to actions. The current states, or a transformation of the states, is then supplied as input and the outputs of the ANN are combined to an action. Intuitively, the hidden layer of a simple Feed-Forward Neural Network (FFNN), as seen in Figure 3-3, can be interpreted as a process of extracting features from the state. The output layer then combines these features to find the best action, similar as in Section 2-3-2. Employing this approach for BBR yields the advantages that the ANN encodes both the different sub-behaviors and selection of the best sub-behavior for the current state.

Learning an ANN is the process of finding a set of parameters which lead to a network with high performance. The output of an ANN is fully specified by the number of neurons per layer, the number of layers, the weights in the synapses connecting these layers and the activation functions in the neurons. The activation functions are often defined by the designer, the other parameters are learned. ANNs can be learned by all three methods as described in Sections 2-2-1 to 2-2-3.

An Evolutionary Learning (EL) approach to ANN optimization is called *neuroevolution*. Neuroevolution can be employed to both tune the network weights and other parameters and to define the network structure. The latter is named NeuroEvolution by Augmenting Topologies (Stanley & Miikkulainen, 2002). Employing EL for ANN learning yields two advantages, in addition to the advantages listed in Section 2-2. First, EL allows for optimization of multiple parameter types (Beer & Gallagher, 1992). Second, EL is effective in avoiding local optima when applied to ANNs (Yao, 1999). Both advantages are demonstrated in Croon et al. (2013). Here a continuous time recurrent neural network is evolved; this pro-active network requires additional parameters —such as a time constant. Their research shows that ANNs can successfully represent different sub-behaviors and that the behavior encoded in an ANN can be reproduced using a Finite State Machine, see Section 3-2-4.

When learning an ANN with RL the ANN functions as a function approximation of the value function, Section 2-3-2. During on-policy operation, the agent will select the action with the highest value. ANN are well suited for value function approximation since the networks extract their own features. ANNs as global function approximators are difficult to tune. Riedmiller et al. (2009) proposes two key aspects to achieve good performance with ANNs. First, as both policy iteration and weight updates are an iterative process, performance can be improved by re-iterating on past experience, named *experience replay* (Lin, 1992; Wawrzyski, 2009). Second, as frequently visited states will influence the value of other states —causing divergence—, the network values must be restraining by an absorption state for which the value is always zero (Kober et al., 2013). While employing both techniques, Riedmiller et al. (2009) successfully encoded different sub-behaviors in multi-layer ANN on a robotic soccer application.

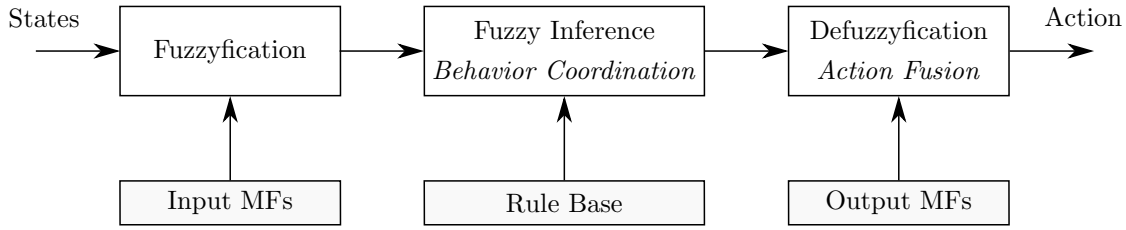


Figure 3-4: Block diagram of a fuzzy logic behavior representation.

3-2-2 Continuous: Fuzzy Logic

Fuzzy Logic (FL) is a technique to deal with uncertainties. In traditional- or Boolean logic, values can either be a zero or a one. With FL, values can take on any value between zero or one. As a real-world environment is inherently uncertain, FL offers many solutions to the field of robotics.

In behavior-based robotics FL can be employed for both behavior coordination and action selection, resulting in a continuous representation of both states and actions (Safiotti, 1997; Seraji & Howard, 2002; Vadakkepat et al., 2006). For behavior coordination FL rules asses the current state from the onboard measurements to activate the suitable sub-behaviors. Action selection is based on fusion between actions of the activated sub-behaviors for the current environmental condition. A nice feature of action fusion is that the agents will can follow a smooth path, which is a combination of different sub-behaviors.

A FL behavior representation consists of three steps, depicted in Figure 3-4. In the first step the state, or measurements, are *fuzzyfied* using input *membership functions*. A membership function is a simple function which maps the state —or features of the state— to a value between zero and one, indicating the *membership degree* of state s for a certain condition. The second step, *Fuzzy Inference*, evaluates the environmental conditions based on the membership degree and selects the required sub-behaviors based on a *Rule Base*. The Rule base can be interpreted as the decision maker to coordinate the sub-behaviors. Finally, the actions of the activated sub-behaviors are merged in the *defuzzyfication* step. This step employs similar membership functions as the first step to combine the actions of the sub-behaviors. A FL representation is therefore fully determined by the in- and out-put membership functions and the rule base.

Membership functions and rule bases are functions; finding the parameters to represent these functions can be seen as a learning problem. In Rusu & Petriu (2003) an ANN is trained to represent both the input membership function and the rule base. Their work shows that a neuro-fuzzy controller can coordinate the different sub-behaviors of a robot on a navigation task. The imitation approach Rusu & Petriu (2003) employed, however, limits the practical applications of their approach.
Evolutionary learning and RL have also proven to effectively coordinate robotic sub-behaviors (Yung & Ye, 1999; Vadakkepat et al., 2006; Mucientes et al., 2007). The evolutionary approach of Vadakkepat et al. (2006) to learning simple membership functions proved very effective. In their research, they successfully implemented a fuzzy representation to coordinate the behavior of a robot-soccer team. This shows that a fuzzy representation can even coordinate the behaviors of a multi-robotic system.

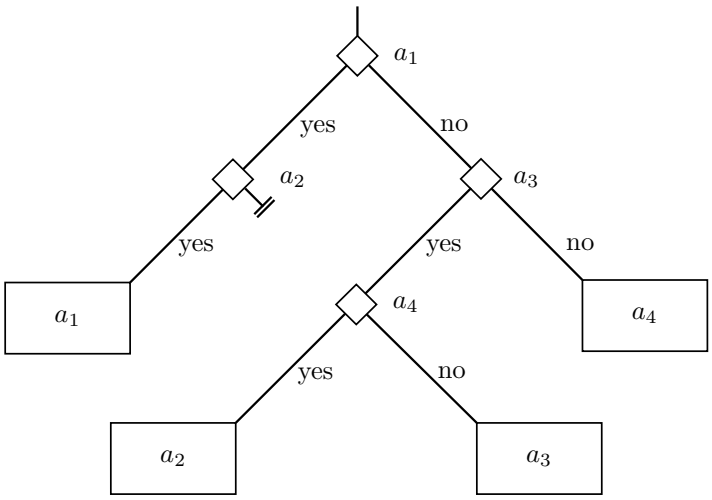


Figure 3-5: Block diagram of a binary decision tree, based on Millington & Funge (2009).

3-2-3 Discrete: Decision Tree

Decision Trees (DTs) are compact representations of functions which map features to decisions by a sequence of simple choices (Millington & Funge, 2009). A DT has a tree-like structure, where every branch split represents a different path. Path selection depends on a choice. The decision process is initialized at the top node, the *root*, and propagates downward until a *leaf node* is reached. Leaf nodes represent actions or sub-behaviors, as depicted in Figure 3-5. Hence, the choices made during the tree evaluation determine which leaf node is activated and therefore which behavior is expressed.

DTs simplify the decision making process by splitting a large problem into a series of simple choices, typically by means of single Boolean conditions. This makes a decision tree fast to evaluate and relatively simple to design and understand. Furthermore, by re-using the behavior nodes —always located at the leafs— DTs can be modular. For larger trees with many decisions, however, the downward-propagating tree structure requires the operator to take more decisions that required. Additionally, as the tree becomes larger for more complex problems, the readability quickly diminishes.

DTs for behavior representation are most often designed by hand in the gaming industry, although learning by imitation has also extensively been explored (Quinlan, 1986; Winston, 1992; Millington & Funge, 2009). In Smith & Nguyen (2007), however, a DT with conditions based on fuzzy logic successfully coordinated the autonomous cooperation of UAV team. Here it must be noted that the DT only coordinated the assistance behavior, a small part of the entire UAV team behavior.

3-2-4 Discrete: Finite State Machines

FSMs are often considered the building blocks of sub-behaviors and achieved many successes in the field of robotics and autonoma theory (Brooks, 1986). A FSM can, however, also be employed directly to encode robotic behavior. FSMs represent behavior in a *state-transition diagram* consisting of a finite set of states which are connected through *transitions*. A state

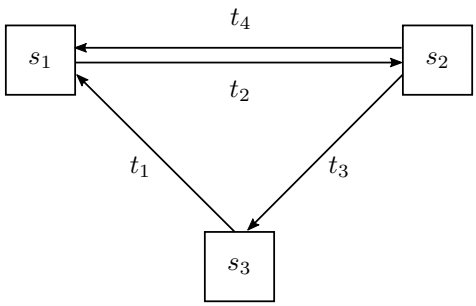


Figure 3-6: Block diagram of a finite state machine of three states s and four transitions t.

can be interpreted as an activity which the robot is currently expressing, leading to a certain behavior. Transitions are conditions statements which cause the agent to switch states if the condition is met. Therefore, an agent implicitly keeps track of the current state and acts to environmental conditions dependent on the current state.

For robotics, FSMs have the advantage that humans can interpret the agent behavior (Croon et al., 2013). This feature can be employed for error-checking, validation and implementation of robotic behavior. As FSMs employ a state-based representation, however, the number of nodes and transitions scales exponentially with the number of states the agent can occupy. Valmari (1998) named this phenomenon *state explosion*.

HFSM are a methods to mitigate the state explosion problem. The behavior of FSMs is often a result of emergence through the interactions of states and transitions. In Hierarchical Finite-State Machines (HFSMs), states are combined to form sub-behaviors —a BBR approach. Despite that this method drastically increases the readability of the FSM it still requires the designer to specify all possible transitions per state. Real-world robotic applications often have a large possible set of states and sub-behaviors, reducing the applicability of HFSMs.

FSMs have, however, a wide variety of successful real-world applications. Often, the FSMs have been designed by hand or copied from other behavior representation methods by careful observation (Millington & Funge, 2009; Croon et al., 2013; Tijmons et al., 2013). Other applications constructed FSMs through learning techniques. In Grollman & Jenkins (2010) it is attempted to learn a the set of states and transitions to control a robotic soccer dog through imitation. The authors only consider a simple linear regression approach and conclude that it is not yet possible to learn FSMs through imitation as the problem becomes an ill-posed regression. An evolutionary approach as proposed in Chivilikhin & Ulyantsev (2013) did manage to learn a successful FSM.

3-2-5 Discrete: Behavior Trees

BTs represent behavior by organizing actions and conditions in a hierarchical tree-structure (Dromey, 2003; Millington & Funge, 2009). The BT tree-structure consists of nodes which have a parent-child relation; nodes can be either a control flow node or an executable node. A BT starts evaluation at the *root* and propagates downward, from left to right, until one child branch is able to successfully activate. Condition nodes return *success* if the condition is met and action nodes return success when the action is completed. In addition to action- and condition nodes, BTs consist of *composite* nodes. Composite nodes are control flow nodes

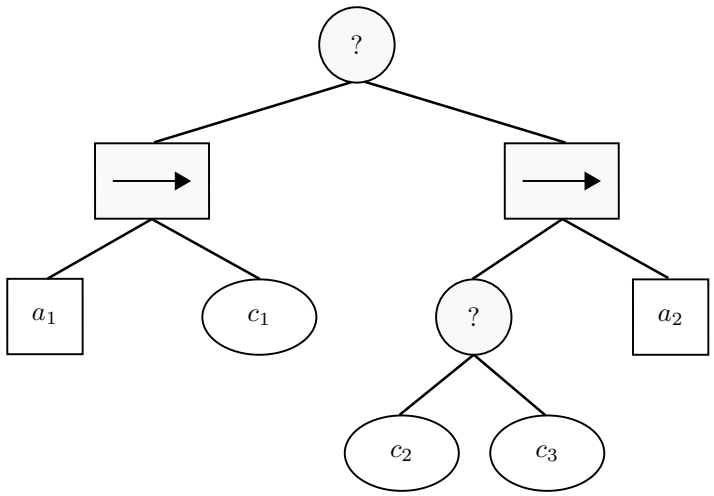


Figure 3-7: Visual representation of a BT. Gray tinted nodes represent composite nodes; an arrow indicates a sequence node and a question mark represents a selector.

which successful activations depend on the activation of two or more child behaviors. The most frequently employed composite nodes are *selector* and *sequence* nodes. Selector nodes succeeds if one child returns success; sequence nodes succeed if all children return success. An example BT is displayed in Figure 3-7.

BTs are goal-oriented behavior representation methods which recursively decompose that goal into sub-tasks. This segmentation makes the sub-behaviors in different branches very modular and simple to interpret. Additionally, this allows for implementation of different sub-behaviors for similar tasks, as the BT branches are hierarchically evaluated, creating robustness. Finally the use of composite nodes makes a BT computationally efficient as not all conditions have to be tested before acting. A BT acts on a set of predefined sub-behaviors which each considers a single or set of states. These states can be live sensory input or constructed from earlier actions or sensory patterns. This allows the designer flexibility with respect to designing a reactive or proactive BT controller.

The visual structure and modularity enable BTs to be designed by hand (Millington & Funge, 2009; Ogren, 2012; Abiyeva et al., 2016). Recently, several implementations of BTs on robotic platforms or embedded software agents have been formulated through evolutionary learning (Lim et al., 2010; Perez et al., 2011; Scheper et al., 2016). In an EL approach the BT is parametrized and optimized through several EL mutation methods ranging from small mutations to leaf nodes to replacing composite nodes with random trees to ensure gene diversity. Since all branches in a BT contribute to a single goal, the previously mentioned methods employed a fitness function which only rewarded the end goal of the agent.

In Dey & Child (2013) a RL approach to optimizing BTs is suggested. In their research the conditional nodes in a BT were replaced with *Q-Condition nodes*. These nodes formed a simple look-up table containing all high-utility states, from which it was possible to select a particular value for the corresponding conditional node. This article is discussed in detail in Section 3-3-3.

3-2-6 A Case for Discretized Behavioral Representation and Behavior Trees

In the previous sections we provided an overview of different behavioral representation methods. Every method has achieved successful robotic implementations; every method has a specific set of advantages and disadvantages. Given the research motivation provided in Chapter 1, we are looking for a representation method that has the following properties:

1. **Interpretable & adaptable**. Given that a simulation environment is unable to capture the real-world complexities, the simulation result is required to be adapted to the real-world environment.
2. **Computationally efficient**. Due to the limited size of MAVs the behavior representation method must be computationally efficient to collaborate with the on-board processing unit; a more efficient method may be activated at a higher frequency.
3. **Scalable & modular**. Although this research focuses on a task with a limited complexity, a scalable & modular representation could allow for the encoding of increasingly complex behavior, facilitating future research.

As a first consideration, let us examine the continuous versus a discretized behavior representations. Continuous representations approach the behavior representation through functions. These functions can express smooth and effective behavior. The sub-behaviors are, however, highly coupled through the function variables; this does not facilitate simple interpretation and adaptation of the sub-behaviors. The sub-behaviors are different to distinguish when looking at the representation method, however can be identified through observation of the agent in its environment. As adaptability is the decisive property, we will discard the continuous behavior representations and continue considering the discretized methods.

Discretized methods form behavior by discretely combining sub-behaviors This discretized segmentation of sub-behaviors enables adaptation of action or sub-behavior coordination. Within discretized methods, BTs appear to combine the following positive properties of the other methods:

- The structured and task-oriented decision making process of DTs, but with more efficient evaluation due to the implementation of composite nodes, incorporation of relations between nodes in different branches and the possibility for parallel evaluation of branches.
- The graphical flow-chart representation from FSMs, without suffering from the *state explosion* problem as the tree does not follow a state-based formulation.
- The decomposition of large branches in modules from HFSMs, without requiring to formulate the transitions between the sub-behaviors as the transitions are encoded in the tree structure.

Furthermore, the successful learning applications of BTs, discussed in Section 3-2-5, demonstrate that the structure and numerical values in the tree can be optimized for a certain task. Therefore it is argued that a BT representation is the most suitable method for this research.

3-3 Behavior Trees as Representation of Robotic Behavior

The previous section deduced that a BT are the best behavior representation method for this research. This section provides a more in-depth documentation and formulation of the BT concept. In Section 3-3-1 the background of BTs is summarized after which a detailed overview of the BT notation and mechanics is provided. The final section discusses relevant recent advances on BTs.

3-3-1 Background

BTs were originally designed by Dromey (2003) as a suggestion to design software models in a more direct, repeatable and constructive from a set of functional requirements. Computer game designers became the first adapters of the BT to design and represent the behavior of intelligent non-player characters (Isla, 2015).

Robotic and non-player characters are different, but also share the key feature that they operate autonomously as they can both be considered to be embodied —to some extentt. This similarity caused the field of behavioral robotics to start researching the potential of BTs. Most research into has been theoretical, although BTs have successfully been used as behavior representation method on real robots (Bagnell et al., 2012; Marzinotto et al., 2014; Scheper et al., 2016; Abiyeva et al., 2016).

Research on BTs is still limited; different terminology and definitions are being used. Throughout this research we will use the definitions from Ogren (2012) and Marzinotto et al. (2014), which are produced by the same research group in KTH Royal Institute of Technology in Stockholm. This group has been most active at researching behavior trees for robotics and embodied agents.

3-3-2 Overview

Section 3-2-5 briefly discussed the concept of BTs. BTs are This section will elaborate on the behavior tree notation, such as different types of nodes, and the mechanics of evaluation based on the definitions of Ogren (2012) and Marzinotto et al. (2014).

Behavior Tree Notation

Formally, a BT is a *directed acyclic graph* with a number of nodes connected through edges in a rooted tree structure. The tree structure enables a compact representation and natural sequence to evaluate the nodes. In addition to the tree structure, the type of nodes influence which nodes are evaluated; in this section we will provide a more complete overview of the node types.

Section 3-2-3 defined the concept of action-, condition- and composite nodes. Action- and condition nodes are child-less nodes which can be found at the bottom of the tree, *leaf nodes*, and perform computations. Composite nodes always have two or more child-nodes and influence which nodes are evaluated, *branch nodes*. Leaf nodes need to be designed for the specific task but can be re-used if a similar action is required whereas composite nodes

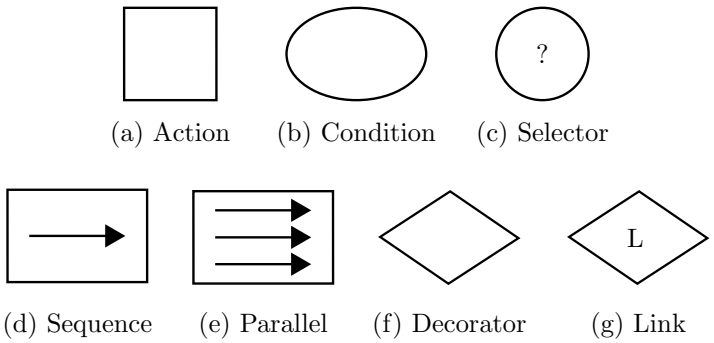


Figure 3-8: Visual representation the most frequently used behavior tree nodes

are usually standardized nodes. A visual representation of the most frequently used behavior tree nodes is provided in Figure 3-8.

An agent controlled by a BT acts with its environment through action nodes, action nodes therefore change the state of the agent. This implies that action nodes can have many different implementation types, ranging from performing sub-tasks or setting different variables in a controller of the agent. Despite the wide possibilities of implementation types, action nodes are always task-oriented. Action nodes return *success* if their action is successfully completed and *failure* if the task could not be completed. In some cases an action node can return *running* if the task is still being activated.

Like action nodes, condition nodes are leaf nodes which perform computation. The condition node often consists of a pre-defined condition which is tested for the current state. To test the condition, the condition node extracts information from the blackboard and then performs the required computation. If the condition is met the action node returns *success* and *failure* otherwise. Condition nodes are often followed by action nodes, such that an action is executed dependent on evaluation of the preceding condition node.

Composite nodes come in a wider variety than leaf nodes. Sequence and selector nodes are the composite nodes which are most frequently implemented in BT and have already been described in Section 3-2-5. Here we discuss three other types of composite nodes: *parallel*nodes, *decorator* odes and *link* nodes.

Parallel nodes are similar to sequence nodes, however, they evaluate all their children in parallel to potentially reduce computation time. The BT designer can specify the requirements for the return of a parallel such as a minimal number of *successes* received from its children. Decorator nodes are the exception of composite nodes and can only have a single child node. A decorator node can change the return or ticking frequency of a child node. This enables decorator nodes to invert or force a specific number of executions for the child node. Finally, the link node commands the execution of a different BT. The return of a link node depends on the return of the activated BT. Link nodes make BTs very modular as they allow re-use of other BTs in their own BT.

Behavior Tree Mechanics

Nodes which are evaluated in a BT return a status depending on the node type and the input state. The status can be *success*, *failure* or *running*. The *running* status can be useful in

cases when tasks take a multiple time-steps to execute, such as commanding the agent to move to a certain location. In this case the agent returns *running* while moving and *success* when it has reached the desired location. The

Some BTs nodes require a data source to both access information about their environment and to communicate with the agent on which action to take. The *blackboard* concept is such a data source which is globally accessible by the BT. Nodes, depended on their type, can read or write on the blackboard.

A BT execution sequence is referred to as a *tick*, this initiates the BT execution at a certain frequency. The tick starts at the root and propagates through the hierarchy of the BT until the root node receives a *success* status or all branches have been evaluated. At the end of a tick the action currently on the blackboard is activated. This execution procedure highlights another difference between BTs and DT. In DTs the tree evaluation propagates downward only. Composite nodes in BTs allow the evaluation to travel back up the tree structure; this in term facilitates to express behavior with fewer repetitions and therefore lower the computational time and number of required nodes.

3-3-3 Recent Developments

Since the introduction, BTs have already been applied on a variety of different applications ranging from UAVs to medical robots. This section highlights some of the relevant recent advances, specifically we will look at the following subjects:

1. **Theoretical properties of BTs**. These developments demonstrate the relation between BTs and established behavior representation methods. This possibly allows us to employ or draw inspiration from techniques developed for the established methods in our research.
2. **Reinforcement learning for BTs**. These developments discuss how RL has been combined with BTs, possibly allowing us to benefit from these previous applications.
3. **Theoretical UAV mission management by BTs**. These developments demonstrate how BTs can be used in the field of aerospace engineering, specifically for mission management on UAVs.
4. **Robotic implementations of BTs**. These developments demonstrate that BTs can be implemented on robotic platforms and how these implementations can be improved.

Theoretical Properties of Behavior Trees

The previous section deduced that a BT possesses the most favorable properties for this research. Originally designed for gaming applications, an accurate and compact mathematical framework for BTs was not required. For robotic applications, with real-world continuous-time dynamics, these foundations are required to provide a better understanding of the tree mechanics. Recently, two papers worked on developing such a foundation.

Marzinotto et al. (2014) proposed a compact and accurate framework to describe BTs. Additionally, they argue that any Controlled Hybrid Dynamic System (CHDS) representation

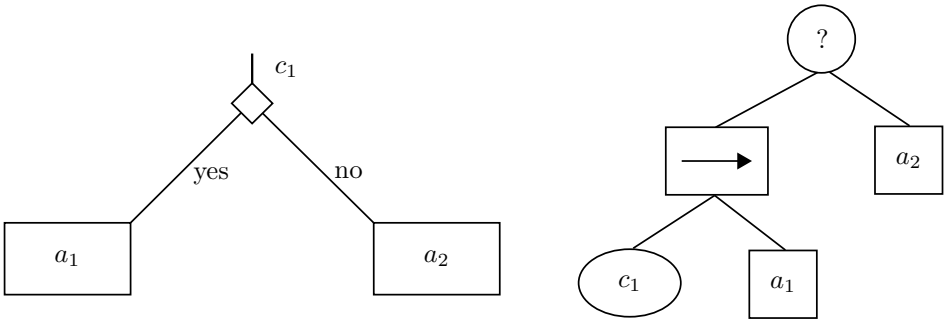


Figure 3-9: Visualization of how a DT (left) can be represented by a BT (right), from Colledanchise & gren (2016).

can be represented by a BT consisting only of selector-, sequence-, action-, and condition nodes —under assumption that an infinite tick frequency and instant tree evaluation. A CHDS expresses both discrete- and continuous behavior, such as FSMs where each state can represent a variable in a differential equation. Despite that their proof and implementation only considers open-loop behavior, we think this demonstrates the potential for BTs as BTs can be implemented with more types of nodes and have a more modular structure than other CHDS systems.

More recently, in Colledanchise & gren (2016) it is demonstrated that BTs generalize both Brooks Subsumption Architecture and DTs. These generalizations are interesting as these control architectures are fairly different and are well understood. This could increase the number of methods to generate, improve and validate BTs. The field of ML has developed several methods to learn DTs as classifier. Potentially, we think that BTs eventually can be learned using techniques developed for DTs. A visual representation of how a DT can be represented by a BT is depicted in Figure 3-9.

Reinforcement Learning for Behavior Trees

In an effort to assist game AI designers to find optimal conditions for activating certain tasks, Dey & Child (2013) combined RL with BTs. Their method consisted of a three step approach. First a BT was designed to perform satisfactory on the task. Then they identified all conditions and actions which they included in the manually designed BT. The second step applied RL to the same problem, using the BT conditions as states and the BT actions as actions. This resulted in a state-action space of 1024 by 8. By applying RL they obtained a Q-value for each of the entries in the state-action space. They then analyzed the Q-table and extracted all states with positive Q-values per action. These states were then stored in a different Q-table; such that each of the eight possible actions had a list with states that resulted in positive Q-values. In the third step they introduced a Q-condition node, which essentially looks at the Q-table for the respective action to see if the current state (which is the set of conditions) is present in the table. If the current state is present, the Q-condition node is activated. This step also included re-arranging the main branches in the original BT by the maximum Q-value in each of the eight Q-condition nodes. The rearranged tree outperformed both the RL Q-table and the original BT.

This research demonstrates that RL and BTs can complement each other, even when both

methods are generated separately. A limitation in their approach is that they do not learn directly on the BT structure. It is expected that when learning directly on a BT the algorithm can exploit synergies between the branches.

A different approach to combining BTs and RL is provided by Pontes Pereira & Engel (2015). In their research they proposed a framework that uses *Reinforcement Learning nodes* as part of Behavior Trees to address the problem of adding learning capabilities in constrained agents. An advantage of their approach is that the agent can learn in real-time, in contrast to the previously discussed combination of BT and RL. Albeit successful, the experiment they conducted to validate the framework was very limited in complexity. Additionally, their framework required the designer to manually formulate which possible actions the agent can learn, possibly limiting this approach to be extended to continuous space.

Theoretical UAV Mission Management

Traditionally UAV missions are coordinated using a FSM–like representation, including states such as *TAKE-OFF* or *GOTO-WP*, in software packages such as Paparazzi (Community, n.d.). Ogren (2012) argued that a BT framework can provide many advantages in terms of modularity, re-useability and readability for UAV mission plans. Combined, these features can increase the mission complexity while decreasing the effort to design a mission planning system.

Klöckner (2013) agreed with the advantages of employing BTs as UAV management systems proposed by Ogren (2012). Klöckner (2013) pointed out, however, that BTs require deactivation of the current action or behavior in case an earlier —more important— node is triggered. This is especially relevant when sub-tasks have large resource requirements, such as an observation task relies on operating a camera which uses on-board energy and computation resources. Drawing inspiration from FSMs, he proposes to add deactivation transitions between tasks. Although Klöckner (2013) points out a valid problem, adding extra relations between tasks reduces many advantages of BTs.

Verification is a crucial step in the design of UAV mission plans. In Klöckner (2015) a method using descriptive logic to verify the interface between the BT and the world is proposed. These logic statements, transforming environmental conditions to BT inputs, can be used to verify safe operation based on a set of logic rules. This is a nice addition as BTs already facilitate validation, and with this addition the conditions can be validated independently.

Robotic Implementations of BTs

For this research, the robotic implementation of BTs is most relevant. BTs have found various real-world applications. The first real-world robotic application was found in Bagnell et al. (2012). Here a BT was employed to coordinate behaviors of a robotic grasping task. The employed BTs was user-defined and demonstrated modularity by successfully re-using branches to grasp differently-shaped objects.

A more complex application of BTs was achieved in Abiyeva et al. (2016) by controlling the decision making and obstacle avoidance behavior of a robotic soccer team. They employed both simple, single-layered, and complex, multi-layered, BTs and increased the flexibility of

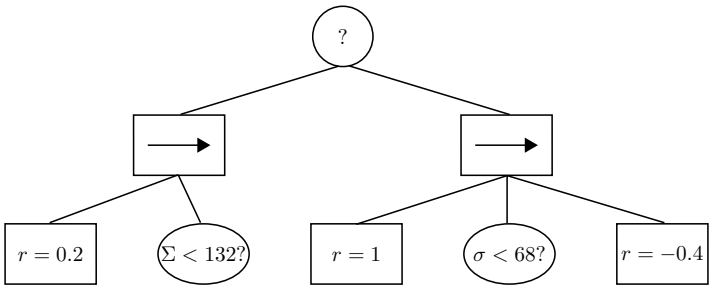


Figure 3-10: Visual representation of the BT employed in Scheper et al. (2016). Square r is the rudder setting of the Delfly, σ represents the window response value and Σ is the sum of disparity.

decision making by employing FL. The application of FLs with 25 rules, however, makes interpreting the BT much more difficult.

In Scheper et al. (2016), a BT representation of MAV behavior reduced the effect of the reality gap due to the simple to interpret and adjust properties of a BT. In their research, a BT is evolved in simulation to encode the behavior of an MAV on a fly-through-window task. The evolved BT is visualized in Figure 3-10. Their implementation uses two features generated from the onboard cameras. The first feature is a measure of depth used to estimate distance, called *disparity*. The second feature is generated with integral images to recognize the pattern of the window, the *window response value*. After optimization in simulation the numerical values in the BT were manually adjusted to account for simulation inaccuracies. The simulation inaccuracies originated from different wall texture, room dimensions and plant turning rate. The adjusted BT outperformed the tree generated in simulation, reducing the effect of the reality gap.

The success of Scheper et al. (2016) research in BTs demonstrate both that evolved BTs can successfully be implemented on robotic platforms and that the effect of simulation bias can be mitigated by representing complex behavior in a BT structure. The adjustment process, however, had to be carried out manually. This requires the operator to have significant knowledge of the task environment and platform and lacks a generic solution. An automated solution to the adjustment process could address these problems and potentially improve performance as well as providing a framework to scale to larger BTs on more complex tasks in different environments.

Chapter 4

Preliminary Analysis

This chapter covers the preliminary analysis which was conducted based on the research definition of the previous chapter. The goal of the preliminary analysis is to obtain an understanding of how a RL algorithm can learn a policy for the numerical BT values. As such, this analysis contributes to the first research question.

Section 4-1 describes the task setting and BT employed for the preliminary analysis. The next section elaborates on the methodology to learn the numerical BT values, after which the results are presented in Section 4-3. Finally the conclusion and the implications of these conclusions on the preliminary analysis on this research are discussed in Section 4-4.

4-1 Task Description

To achieve the goal of the preliminary analysis we designed a simple discrete navigation task based on Janssen et al. (2016). This task was selected for the preliminary analysis as (1) Janssen et al. (2016) already formulated a BT for this task, (2) the discrete setting provides a simplified initial environment for experimenting with indirect RL and (3) the agent is required to express different sub-behaviors for successful completion. Point (3) ensures that the RL algorithm must successfully learn all sub-behaviors and provides an interesting insight in the process of learning sub-behaviors.

The task description consists of three parts: the task goal, the task environment and a description of the BT employed to represent the controller for the task goal. The goal, environment and BT for this analysis are based on Janssen et al. (2016).

In this analysis a UAV is tasked to collect trash while navigating through a room and avoiding obstacles. The UAV has 150 time steps to obtain as many points as possible. For every trash piece that is cleared the UAV receives 20 points; when hitting a wall the agent is allowed to continue but gets 10 points deduced. Furthermore the agent gets a reward of -1 point for every turn and zero reward for moving forward.

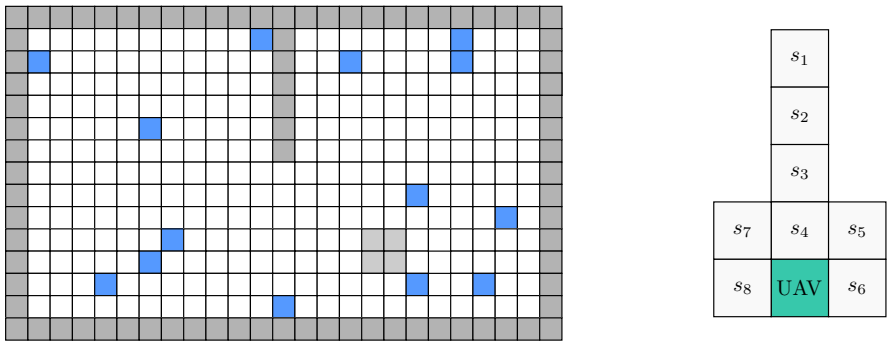


Figure 4-1: Visual representation of (1) grid world and (2) the MAV sensor positions.

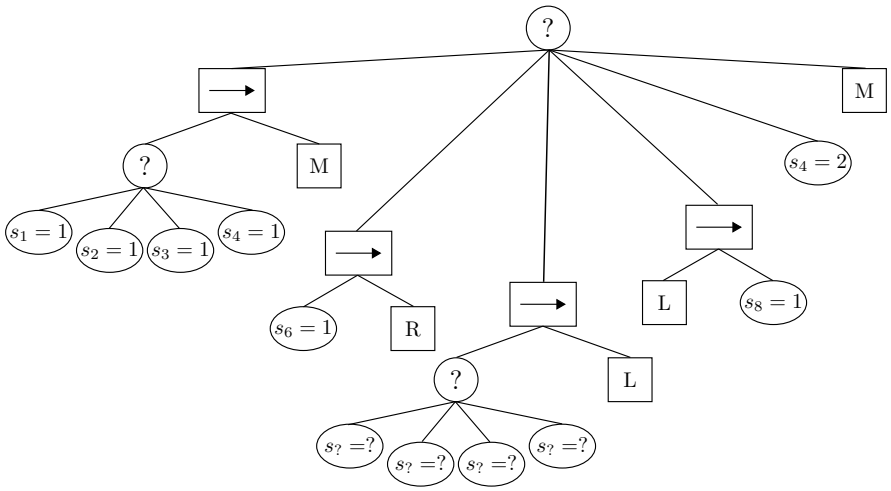


Figure 4-2: Visual representation of the behavior tree found by Janssen et al. (2016)

The room is a discrete grid of 25 by 15 cells; Figure 4-1 provides an illustration of the room. In Figure 4-1 the white cells indicate free space and the gray cells represent walls or obstacles. The blue cells contain trash which the UAV can collect by flying over the particular cells. Every time a trash cell is cleared, a new trash cell appears on a random empty cell; the trash cells are always initialized on the locations indicated in Figure 4-1. The UAV is equipped with eight sensors which allow it to perceive the environment. The sensor locations relative to the UAV are visualized in Figure 4-1. A sensors returns the value 1 when it detects trash, the value 2 for a wall and the value 3 when the corresponding cell is empty. This sensor configurations makes the problem partially observable, as the agent only has information on the contents of the cells which are covered by the onboard sensors.

The BT consists of six branches which combine five sub-behaviors to formulate the behavior of the agent. Condition nodes in the BT check the return status of the UAV sensor measurements. Action nodes command the UAV to either move forward by one cell or to turn 90 degrees towards the respective direction. The first sub-behavior is represented in the first branch and encodes trash collecting behavior when detecting trash with the frontal sensors. The second sub-behavior concerns turning towards trash on the right of the UAV, after which the first sub-behavior is activated to collect the trash piece. The sub-behavior of the third

branch is unknown as the BT depicted in Janssen et al. (2016) contained some graphical errors. If possible, learning the correct values for this branch provides an interesting insight in the actual BT. The fourth branch encodes a similar sub-behavior as the second branch, but for trash on the left of the UAV. The firth branch collaborates with the fourth branch to avoid walls when directly in front of the UAV. The final branch represents wandering behavior to encounter trash; this sub-behavior is activated if the conditions for all previous sub-behaviors have not been met.

The objective of the preliminary analysis is to learn a RL policy which sets the numerical values in the BT depicted in Figure 4-2 on the previously described task. To simplify the problem we will only consider the condition nodes for this analysis and leave the action nodes unchanged. The condition nodes alone, however, already poses a large challenge as there are 24^{11} possible combinations of condition node indexes and thresholds.

4-2 Methodology

Given the task described in the previous section, this section elaborates on the applied methodology. The methodology explains which RL technique is used, the framework though which the agent learns and which methods are considered to facilitate the learning process, namely eligibility traces and state-based exploration and learning rate techniques.

4-2-1 Q-Learning as Reinforcement Learning Method

Section 2-3-2 discussed the three main methods of value function learning. Temporal Difference (TD) learning is considered the best learning technique for the preliminary analysis –and possibly for the research objective– for three main reasons:

- TD learning can learn on-line after every time-step. This allows for fast value function and policy updates during operation of the agent.
- TD algorithms can learn from incomplete sequences in continuous environment. Despite that the current task has a fixed duration of 150 time steps, the real-world task for the research objective may not.
- TD methods are model-free; no knowledge of the MDP transitions or rewards is required.

Q-learning is selected as TD algorithm as this learning technique can also learn off-policy, ensuring the agent learns about the optimal policy while following an exploration policy.

4-2-2 Description of Q-Learning Framework

Implementing Q-learning requires a framework which relates the agent and the environment and defines the states and actions. Two frameworks were considered: a meta-learning and world-learning framework. A description of the working principle of both frameworks is provided on the next page.

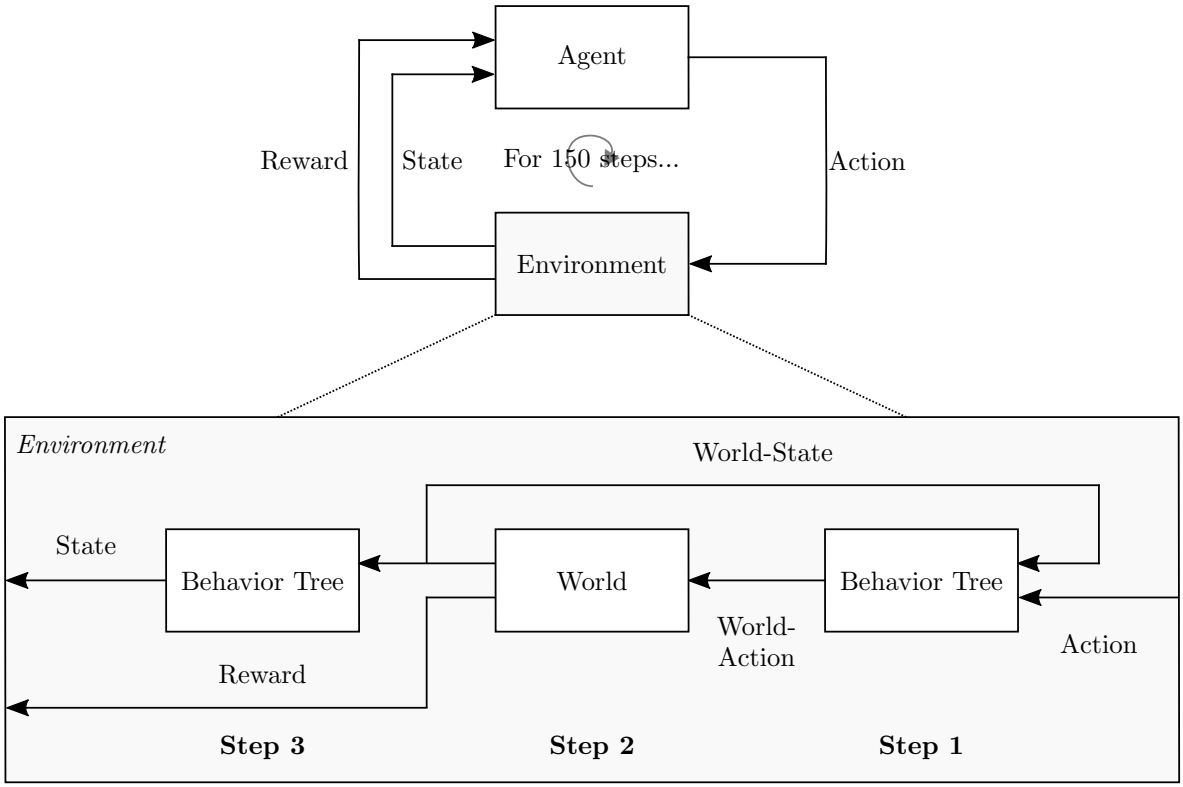


Figure 4-3: Block diagram of the world-learning framework.

- **Meta-learning framework:** agent has full knowledge on the current node values inside the BT and can change the value of one node once per game.
- **World-learning framework:** agent only has the knowledge of the currently activated node and can change the value of the activated node once per game tick.

From the above description it becomes clear that the trade-off between both methods is observability of the numerical BT values and update frequency of these values. In this section we focus on the world-learning framework as this framework was best suited for the large number of condition nodes in this specific tree, leveraging the abstraction of low-observabilityty. Figure 4-3 displays a block diagram of the world-learning framework.

The world-learning framework employs 11 states, one for every condition node in the BT. The agent receives the index of the activated condition node as the state —which can be a scalar ranging from 1 to 11. As an action, the agent selects a combination of the sensor index and sensor threshold for the corresponding state. This results in a total of 24 actions since there are eight onboard sensors which can have three thresholds. The total state-action space therefore has a size of $11 \times 24 = 264$ state-action pairs. Since the world-learning framework only requires the state index —and not the settings of the other condition nodes— the observability of the learning process diminishes significantly; this may result in reduced learning performance.

From Figure 4-3 it can be seen that world-learning distinguishes state, world-state, action and world-action. Previously, the state and action were defined. The world-state and world-action

are the state and the action which the UAV selects in the world, respectively. The world-state consists of the sensor readings; the world-action can be either to more forward or to turn left or right. With these definitions we can describe environment of the world-learning framework according to the following three step process:

1. Apply the action to update the current BT and find the world-action from the current world-state and the updated BT.
2. Command the UAV to take the world-action and retrieve the new world-state and observe the corresponding reward.
3. Find the activated condition node in the updated BT for the new world-state; the index of the activated node is sent to the agent as the new state.

4-2-3 Eligibility Traces

Eligibility traces keep track of which states have been visited and assign credit to these states when a reward is received. The Q-update rule with eligibility traces —using replacing traces— is shown in Equation 4-1 (R. S. Sutton & Barto, 1998).

$$\begin{aligned} Q(s_{t+1}, a_{t+1}) &= Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a_{t+1}} Q\left(s_{t+1}, a_{t+1}\right) - Q\left(s_t, a_t\right) \right] Z_t(s, a) \\ Z_t(s, a) &= \begin{cases} \gamma\lambda Z_{t-1}(s, a), & \text{if } s \neq s_t \wedge a \neq a_t \\ 1, & \text{if } s = s_t \wedge a = a_t \end{cases} \end{aligned} \tag{4-1}$$

In Equation 4-1 the $Z_t(s, a)$ matrix is initially zero and resets to zero when an exploration action is selected. Intuitively, the $Z_t(s, a)$ matrix can be interpreted as a method to keep track of which state-action pairs have recently been visited. This matrix is then employed to determine the influence the recently visited state-action pairs on the TD update. The term $\gamma\lambda$ determines how much the eligibility trace decays per time-step.

For the world-learning framework, it is hypothesized that eligibility traces facilitate the learning process as they reward sub-behaviors which do not directly receive a reward such as wandering or turning towards trash. Additionally, eligibility traces are required to learn wall avoidance behavior as the last action node of the BT from Figure 4-2 is activated when neither of the conditions are met. Therefore the agent does not receive a state for every world state which makes the agent unable to always update the Q-table. This results in the agent being unable to learn from crashes. High eligibility traces cause the agent to avoid walls by positive reinforcement since avoiding walls results in more trash collected in the long run.

4-2-4 Learning Rate and Exploration Function Implementations

RL algorithms are often sensitive to the hyper-parameter settings (R. S. Sutton & Barto, 1998). As this experiment —in our knowledge— is the first attempt to learn the numerical

BT values, we implemented different methods to set two of the hyper-parameters. These parameters are the learning rate and the exploration of the RL algorithm. In the results section of this chapter we compare the performance of the different learning rate and exploration functions.

The learning rate was calculated using two approaches: episode-based and state-based. The first approach employs a generic learning rate, whereas the second approach calculates a learning rate specific for the state-action pair.

1. The first approach computes the learning rate based on the episode according to Equation 4-2. Here α_0 is the initial learning rate, e is the current episode and E is the total number of episodes.

$$\alpha_{episode-based} = \alpha_0 \left(1 - \frac{e}{E}\right) \tag{4-2}$$

2. The second approach calculates the learning rate based on the number of visits per state-action pair, $N(s, a)$, as represented in Equation 4-3.

$$\alpha_{state-based} = \frac{\alpha_0}{N(s, a) + 1} \tag{4-3}$$

Exploration is implemented using two types of exploration functions: ϵ-greedy and a form of UCB-1 exploration (Auer et al., 2002). Just as for the learning rate one exploration function is generic for all state-action pairs, and one is specific.

1. ϵ-greedy selects the action corresponding to the current policy with a probability of $1-\epsilon$ a random action with a probability of ϵ. An expression for ϵ is provided in Equation 4-4; here σ determines the slope of ϵ

$$\epsilon = \epsilon_0 \exp\left(\frac{-e}{\sigma}\right) \tag{4-4}$$

2. The second approach adds an exploration bonus to the Q-values, and then selects the action with the highest exploration value, $Q_{explore}$. This approach results in a more guided exploration, preferring exploration of unseen state-action pairs. In Equation 4-5 k is the exploration bonus.

$$Q_{explore}(s, a) = Q(s, a) + \frac{k}{N(s, a)} \tag{4-5}$$

To aid the exploration of the RL algorithm some node values are mutated after every episode. This form of exploration is meant to prevent the RL algorithm from ending in local minima. At the start of each episode the agent is initialized in the world and the BT conditions are set according to the current policy with a number of mutations. The BT conditions are initialized random; the number of mutations starts at 11 and exponentially decreases to 1 after approximately 50% of the episodes.

4-3 Results of Preliminary Analysis

This section discusses the obtained results by applying the techniques discussed in the previous section to the task of Section 4-1. In the first sub-section the results of the Q-learning algorithm are discussed and the second sub-section analyzes the behavior of the agent during learning.

4-3-1 Q-Learning Results

The results of the Q-learning are provided in Table 4-1. From this table it becomes clear that the the state-based learning rate- and exploration function has the most consistent performance; therefore we will focus on the results of this method for the rest of this chapter. For these results $\gamma = 0.8$, $\lambda = 0.9$, and $\sigma = \frac{1000}{6}$; the other settings are listed in Table 4-1. These values were heuristically determined to provide the best results; the same amount of time per method was spent to tune the variables. It was found that the performance of these methods was very sensitive to the settings of these hyper-parameters.

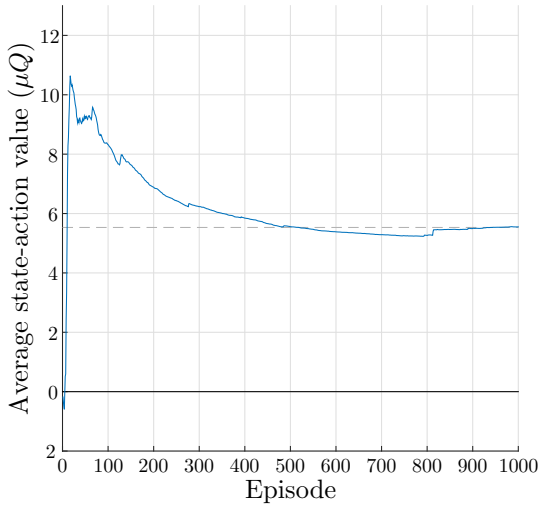
Table 4-1: Performance of the learning algorithm for different learning rate- and exploration functions over 100 independent runs.

Learning rate Function	Exploration Function	Average score	% Converged to optimum	Exploration setting	Learning rate
$\alpha_{episodebased}$	ϵ-greedy	-461.7	22	$\epsilon_0 = 0.3$	$\alpha_0 = 0.5$
	$Q_{explore}(s,a)$	189.6	64	$k = 30$	$\alpha_0 = 0.1$
$\alpha_{state-based}$	ϵ-greedy	15.9	34	$\epsilon_0 = 0.5$	$\alpha_0 = 1$
	$Q_{explore}(s,a)$	206.4	83	$k = 30$	$\alpha_0 = 1$

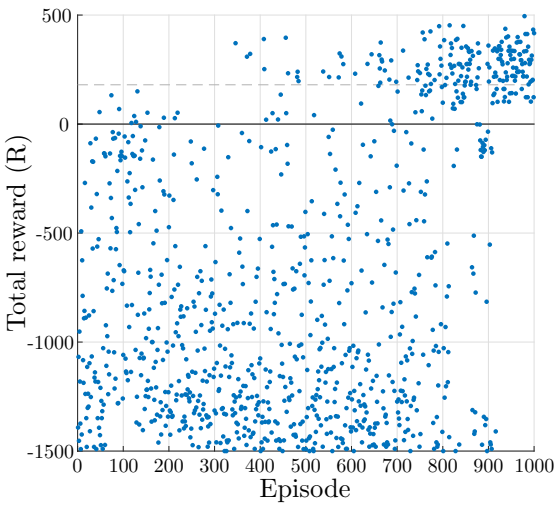
The RL algorithms in Table 4-1 were considered converged when they received a similar score as obtained by Janssen et al. (2016). Janssen et al. (2016) reported that the genetically optimized BT of their research scored 266 points on average in an episode. Given the stochastic elements of trash spawn locations and initial agent position we will consider the RL policy converged if it achieves a score of at least 256 points. This definition removes two important uncertainties. First of all, we hypothesize that the influence of the random events in the task environment (as described above) is reduced by considering the algorithm converged at ten points below Janssen et al. (2016) average score. Second, it is unknown if the conditions in the BT of Janssen et al. (2016) is indeed the optimal combination. By only setting a minimal score to the convergence criteria, possible better combinations are also allowed. This is especially relevant as the structure of the tree allows for multiple optimal solutions. These solutions combine different condition indexes and threshold but express the same behavior, resulting in an equal average score.

Figure 4-4 displays the results of a typical Q-learning run using the state-based exploration- and exploration function approach. The most profound observations from Figure 4-4 and possible causes of these observations are listed below.

- **The average Q-value peaks at the start and ends positive-valued.** Initially the agent learns all positive Q-values, obtaining a high average Q-value at early episodes.



(a) Average state-action value per episode (blue line) and the average action-value over the last 100 episodes (gray-dashed).



(b) Reward during learning per episode (blue dots) and the average reward over last 100 episodes (gray-dashed).

Figure 4-4: Results of a typical Q-learning session.

To explain this phenomenon a more in-depth analysis of the agents behavior during the learning process is required. The next section provides this analysis and explains the high initial Q-values. Additionally, the average Q-value converges to a positive value. This is unexpected as there are only a limited number of condition combinations which result in a behavior achieving positive rewards. It is therefore expected that the Q-value estimates are sub-optimal for most states. Suboptimal Q-value estimates are a typical consequence for applying Q-learning to a POMDP (Kaelbling et al., 1996). Ultimately, this can lead to sub-optimal policies if the Q-value estimates vary too much from the optimal Q-values.

- **The average Q-value displays a gradual convergence rate.** The slope of the average Q-value decreases with the number of episodes. This behavior originates from a reduced learning rate as the learning rate determines the update step size of the Q-values. The learning rate decreases with the number of episodes as the number of state visits increases, leading to a decreased learning rate according to equation 4-2.

- **The average Q-value sharply increases at specific episodes.** At episodes $\approx 60, 120, 270$ & 820 the average Q-value suddenly increases. These spikes are caused by discovering a new, improved, policies for individual nodes.

- **The total reward expresses a random behavior with a sudden increase at episode ≈ 820.** As a result of exploration during the learning process the agent can follow a sub-optimal policy or even follow different policies during a single episode, leading to a seemingly random total reward. At episode 820 the agent starts following the optimal policy, reducing the number of exploration actions. As the agent explores using the described state-based exploration function, the agent can suddenly stop exploring

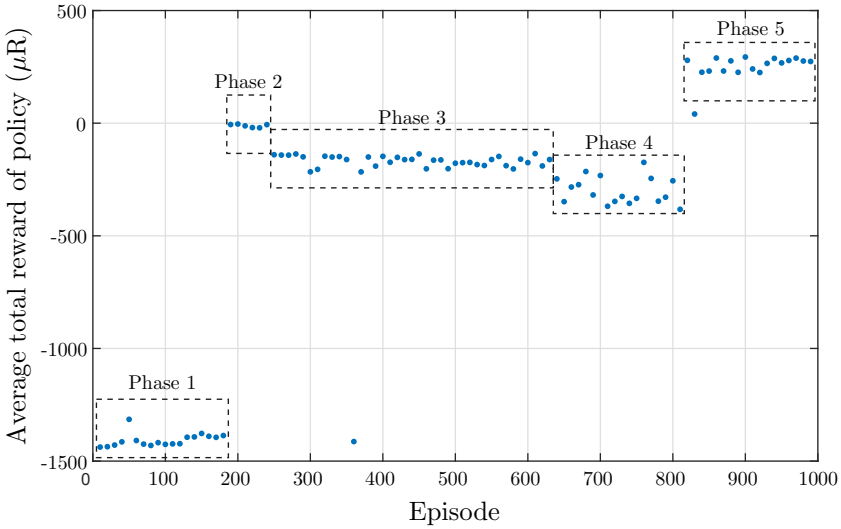


Figure 4-5: Average reward over 100 independent runs of the agent policy per 10 episodes.

when the exploration bonus for most states becomes small due to the high number of state visits. This explains the sudden increase in total reward at episode 820.

- **The total reward continues to display high variance at later episodes, despite that the policy seems to be converged.** The random trash spawn locations cause the agent to encounter a varying amount of trash. Therefore the variance of the total reward can be high despite that the agent follows an optimal policy.

4-3-2 Analysis of Behavior During Learning Process

In the previous section it was shown that both a state-based exploration function and learning rate resulted in the best learning performance for this task. This section analyzes how the agent learned the policy and which sub-behaviors the agent expressed during learning. The goal of this analysis is to obtain a deeper understanding of the learning process, such that we can tune the hyper parameters and develop techniques to facilitate the learning in future applications of indirect RL.

To conduct this analysis a copy of the agent was saved every 10 episodes during learning. We then evaluated the policies of these agents in the world; the agent would not explore and only act according to the policy. As the task environment is stochastic, the reward of every agent was averaged over 100 independent simulations. A typical resulting reward per episode is depicted in Figure 4-5.

Figure 4-5 clearly shows that the agent expresses 5 different behaviors during the learning process; every sub-behavior results in a different score. The two outliers are a result of exploration actions. The policy recorded to generated Figure 4-5 only considers the policy at the end of an episode. Therefore, if the agent selected an exploration action at a later time step, the recorded final policy of the episode can express different behavior. Below the five phases during learning are described.

1. **Phase 1 - trash collecting behavior:** Agent moves forward and learns to use frontal sensors in nodes of the first branch to detect trash cells.
2. **Phase 2 - wall following behavior:** Agent traverses along walls and learns to use frontal sensors in nodes of branch 3 to detect wall cells.
3. **Phase 3 - turn to trash behavior:** Agent spins around its axis and learns to use nodes in branch 2 and 3 to locate trash to the side of the UAV.
4. **Phase 4 - improved wall avoidance behavior:** Agent walks through the world and collects trash, crashing in specific situations.
5. **Phase 5 - final behavior:** Agent expresses all sub-behaviors and successfully completes the task for every episode while improving the Q-value estimates.

It is reasoned that the hierarchical learning of the sub-behaviors originates from the random initialization of the condition nodes. The agent starts off without a strategy and encounters trash based on coincidence. Initially, the number of state visits is low, resulting in a higher learning rate and fast learning. These high Q-updates force the agent to always move forward, setting the condition nodes such that either the first or the last action node is always activated —these nodes lead to the highest reward of +20. The policy is then simply updated in the order which provides most cumulative reward, as sub-behaviors leading to higher rewards profit from larger Q-updates.

We believe the state-based exploration and learning rate function contributes to the phased learning of the agent behavior. As previously discussed, the agent learns sub-behaviors by positive reinforcement through eligibility traces; some of the sub-behaviors are dependent on other sub-behaviors. As a result, the agent has to carefully balance exploration and exploitation since exploratory actions break the eligibility traces, however, are required to find new sub-behaviors. Therefore the agent is required to explore only in behaviors which are still unknown, while exploiting the already learned behavior; this is exactly what the state-based exploration function facilitates. The state-based learning rate allows the agent to quickly adapt to new behavior once explored and naturally copes with the tree hierarchy as later branches are visited less frequently during learning.

4-4 Conclusion

The objective of the preliminary analysis was to obtain an insight in how a RL algorithm can learn a policy for the numerical values in a BT. In this analysis we showed that RL can indeed learn a policy for the numerical BT values. More importantly, some important findings indicate that further research is required to develop a generic method for this objective. These findings are listed below.

1. The methodology has to be updated to also learn negative Q-values. Given the research objective, the agent is required to be aware which actions result in (very) bad rewards to aid the on-line learning process.

2. State-based learning rate and exploration functions resulted in the best performance for this task. It must be further investigated if these techniques also improve the learning performance on other BTs.

3. The learning performance was very sensitive to the hyper-parameter settings. Further optimization of these settings is advised to aid both offline- and on-line learning.

4. The current optimization only focused on learning the condition nodes; the dynamics between learning condition- and action nodes is not treated in this analysis. The combination of learning both node types must be further researched as both have to be optimized in a real-world environment.

5. The current learning method is only applicable to a discrete state-action space. This method has to be adapted to function either with function approximation techniques or the algorithm has to search for the optimal parameters in discrete steps. Our preference goes to the second option as this simplifies the RL process and allows for state-based learning rate- and exploration functions. Further research by learning a policy for the BT of Scheper et al. (2016) must conclude on which option will be selected.

6. In this experiment the tree nodes were learned from a random initialization. For the research objective, however, it is assumed that the BT values are already near their optimal values. This may simplify the learning process.

Part III

Additional Results

Chapter 5

Extended Parameter Coupling Analysis

Part I of this report presented the coupling of nodes 1 & 2 and 1 & 5 for the baseline behavior and demonstrated that these parameter combinations resulted in a performance landscape with a single peak. In this section the coupling of every combination of two parameters is presented for all environments. This section functions as additional validation of the one-peak performance landscape validation.

Similar as for Part I, the couplings are generated by performing a grid search for the considered policy parameters. Two values for the policy parameters, which are not varied during the grid search, are employed: (1) initial policy settings; and (2) the improved parameter settings found by DIPS as presented in Part I. These two different policy parameter settings visualize the performance landscape of the (coupled) parameters during the first episode and at the optimized settings found by DIPS, respectively. For Environment 1, however, both settings of the policy parameters are equal.

The performance of the parameter setting pair is determined by initializing the DelFly at 100 random locations; for every environment the initialization locations are the same for all plots. The resulting couplings are displayed in Figures 5-1-5-7. From these figures, the following main observations can be made:

- **Performance landscape of all combinations of two policy parameters has a single peak in all considered environments.** This one-peak performance landscape verifies the assumption made in Part I. Additionally, it can be observed that the peak has a relatively large setting allowance once the policy parameters are near their optimal setting, which was also observed in the performance landscape analysis of Part I. Further it can be seen that the combination of two parameter settings is often limited by the lowest performance of the two considered parameters. This further demonstrates the important working principle of DIPS: a policy performance increase must be observable by incrementing one of the policy parameters. These observations, however, are only valid for the considered parameters and their specific ranges. A higher-dimensional grid search may reveal multiple peaks if the considered parameter ranges are increased at, for

example, the opposed sign of all action parameters. This second peak would correspond to the same behavior, but with opposite actions.

- **Direction array elements of were selected properly for the three experimental environments.** By analyzing both the behavior encoded in the BT and the adaptations to the environment a direction array could successfully be formulated to improve the behavior. This becomes especially apparent by analyzing the performance landscape of the initial policy settings of Figures 5-1–5-4. For Environment 2 and 3 the performance appears to be especially sensitive to combinations of parameter 1 and 2. This also applies for Environment 1, with the addition of parameters 1 and 3. Possibly the behavior performance in Environment 1 could have been increased by also including θ_2 in the optimization process. Finally, due to reward function formulation, parameters 3 and 4 appear to initially result in the largest performance increase. Reducing parameter 3 or increasing parameter 4 increased the probability of crashing; initially this results in a performance increase. This demonstrates the dependency of RL algorithms on formulating a good reward function, even for this research where reward function crafting has been kept at a minimum.
- **Performance of the baseline environment can be increased.** Part I assumed that the initial policy would result in an optimal behavior in the baseline environment. Figure 5-1, however, shows that by reducing the settings of θ_1 and θ_3 the behavior performance is increased. This different node settings are expected to originate from the reality gap between the SmartUAV and FastSim simulator.
- **The performance landscape of the baseline environment and Environment 1 are similar-shaped**. Approximately, the shape of these performance landscapes are translated and scaled. The scaling effect is expected to be the result of three factors. First, the baseline policy has not been optimized by DIPS. Second, because of computational limitations the success rate of the parameter combinations was calculated using a limited number of trials. Third, an increased performance in Environment 1 may indicate that a the increased maximum rudder deflection of the DelFly allows for a performance increases because the DelFly is able to perform steeper turns. Additionally, initialization locations are different for both environments. As such, one environment may be evaluated at initialization locations which results in a increased success rate.
- **The performance landscape of the baseline environment and Environment 1 are similar-shaped.** Similar as for the baseline environment and Environment 1, the scaled performance peak is partly the result of different initialization locations. In Part I, however, it was shown that the final policy of Environment 3 has an increased success rate over Environment 2. This again proposes that the encoded behavior can be more effective in Environment 3 than Environment 2.

The similar shape of the parameter performance curves suggests that the performance curve of one environment can be approximated by translating the known curve of a slightly different environment. This may be leveraged in future research to encode additional a-priori knowledge in the policy search algorithm. Note that the analysis of this chapter considered coupling of two policy parameters only. Possibly a higher-dimensional grid search allows for an increased insight in the true functional form of the policy parameters. Due to computational limitations, however, we were limited to a two-dimensional analysis.

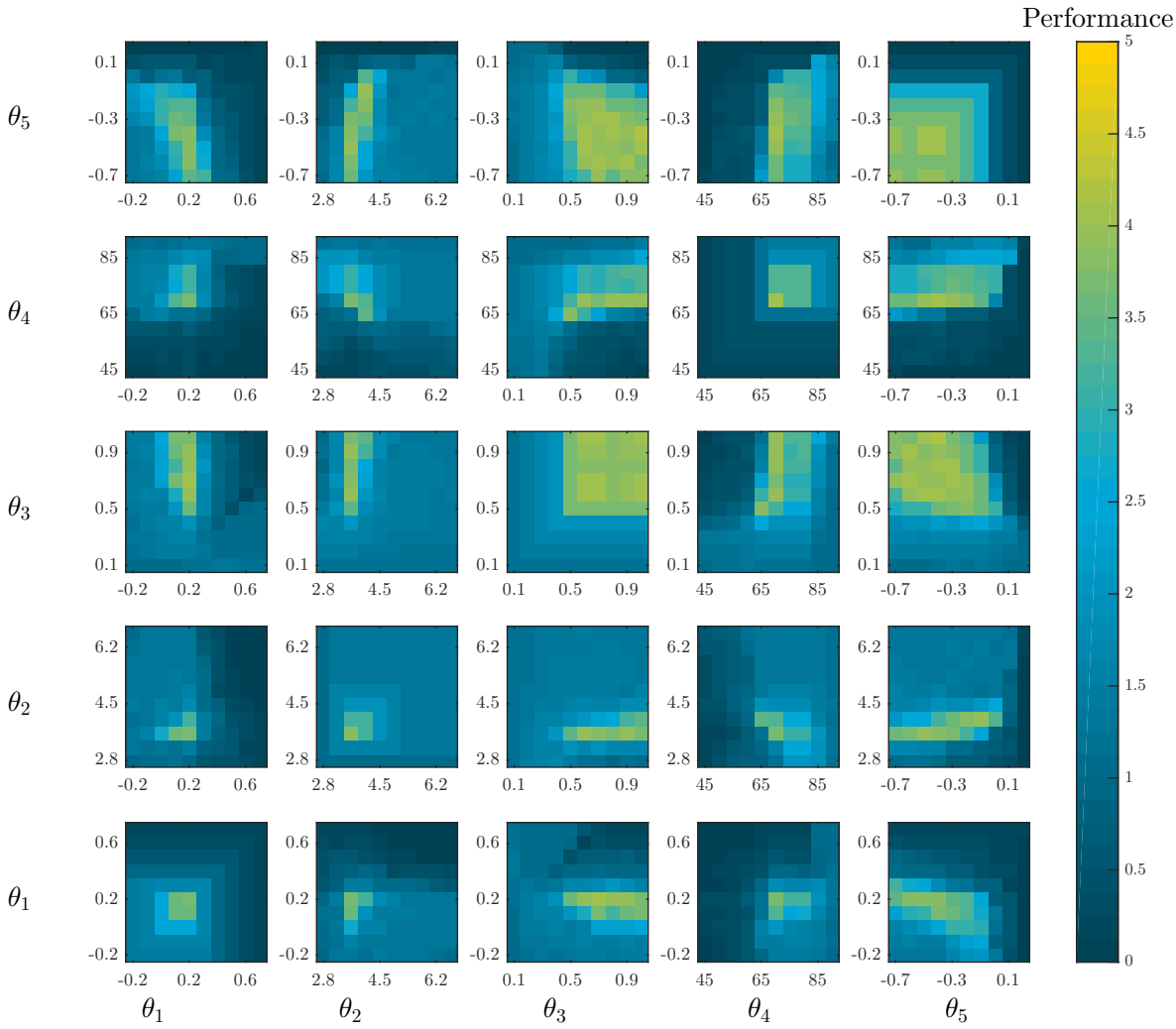


Figure 5-1: Coupling of behavior tree parameters around the optimized policy for the baseline environment.

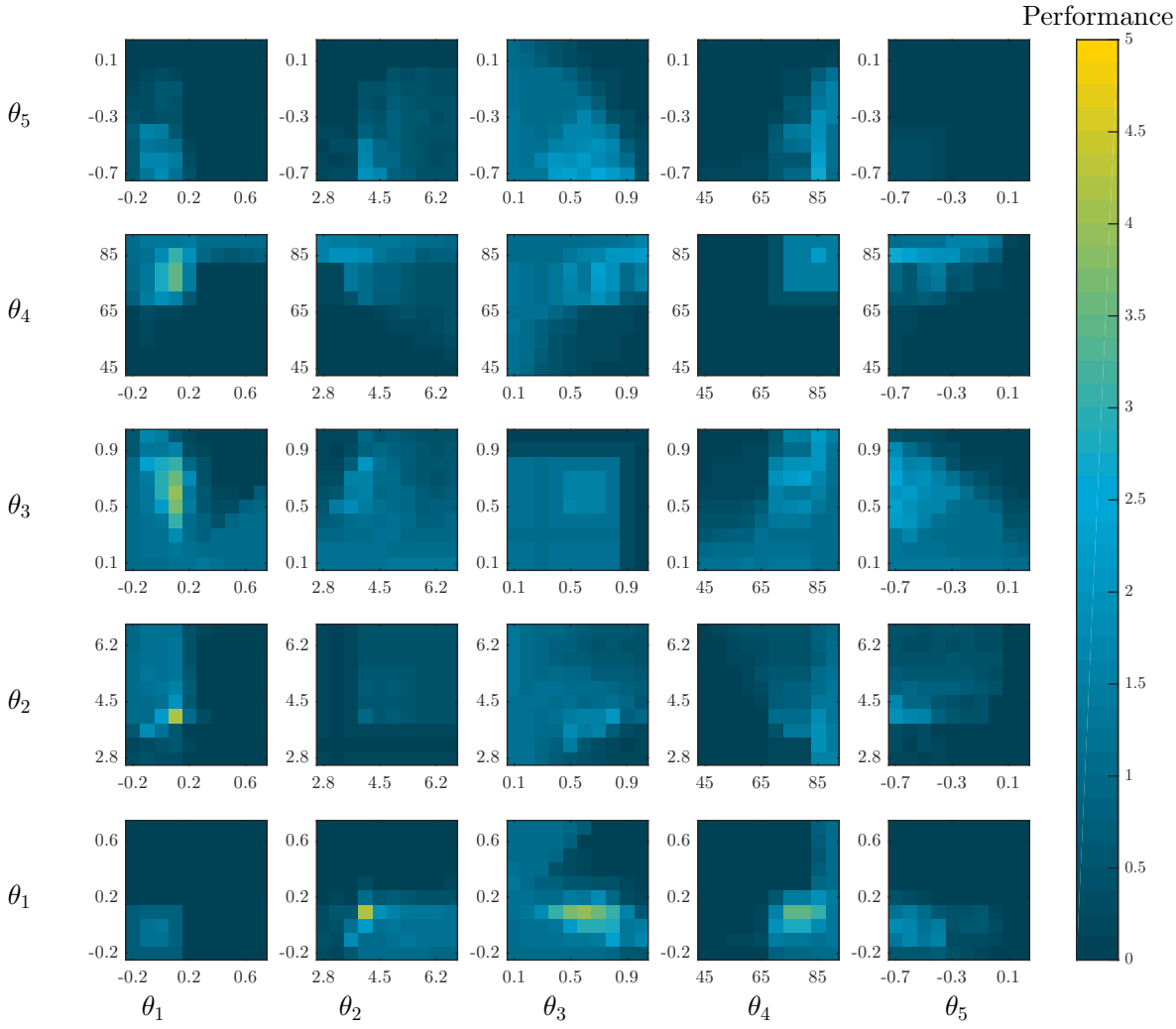


Figure 5-2: Coupling of behavior tree parameters around the initial policy for Environment 1.

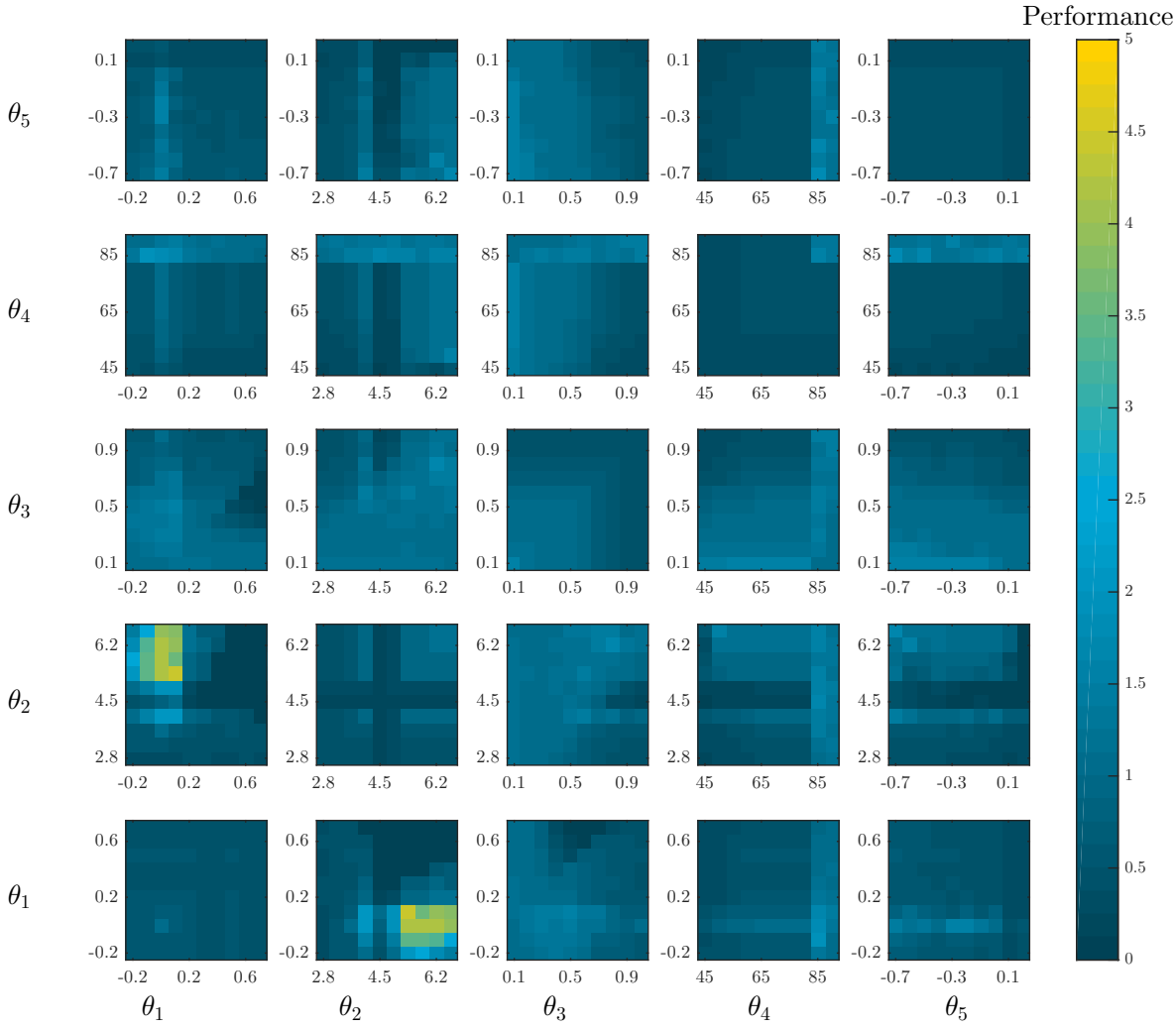


Figure 5-3: Coupling of behavior tree parameters around the initial policy for Environment 2.

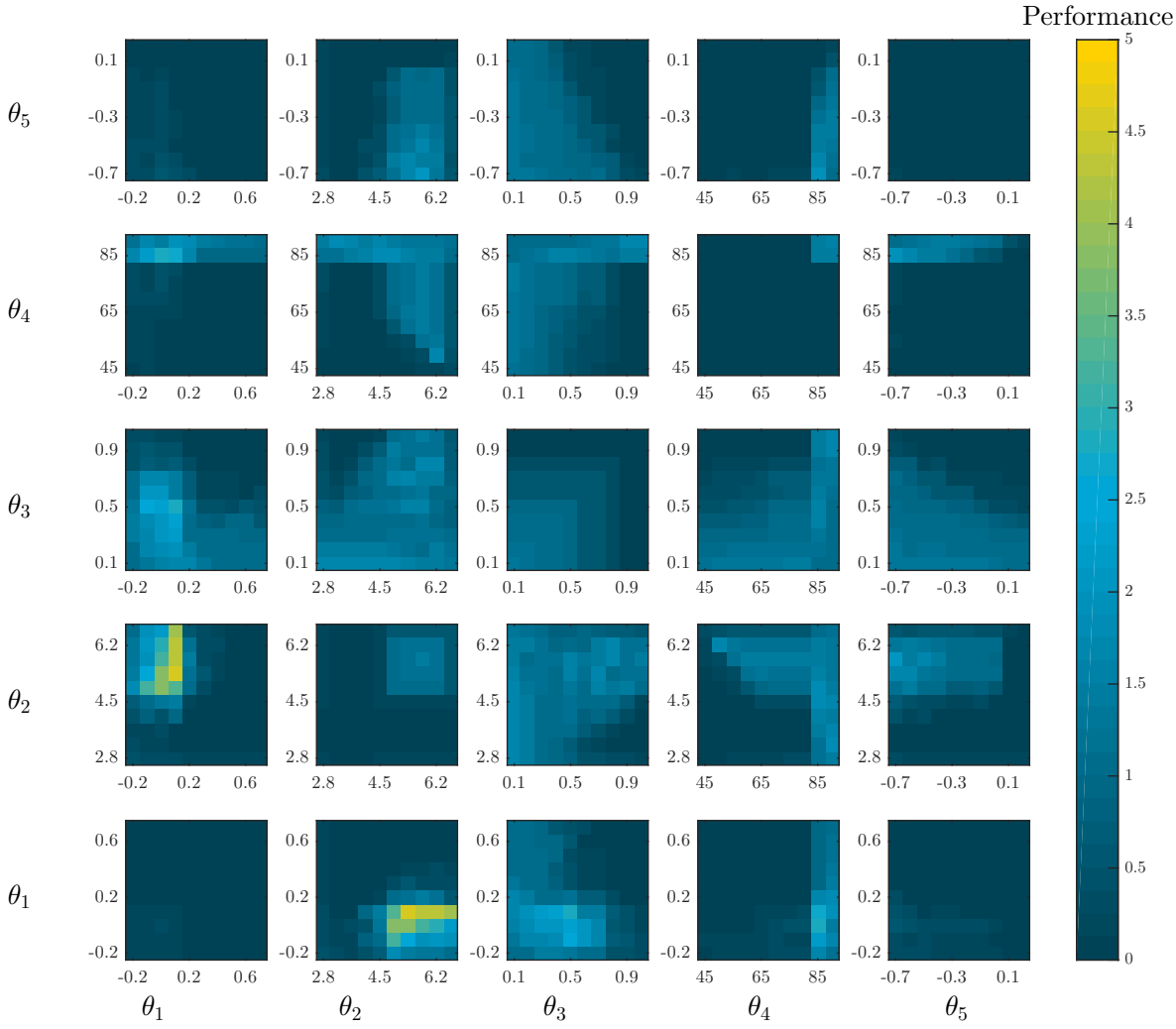


Figure 5-4: Coupling of behavior tree parameters around the initial policy for Environment 3.

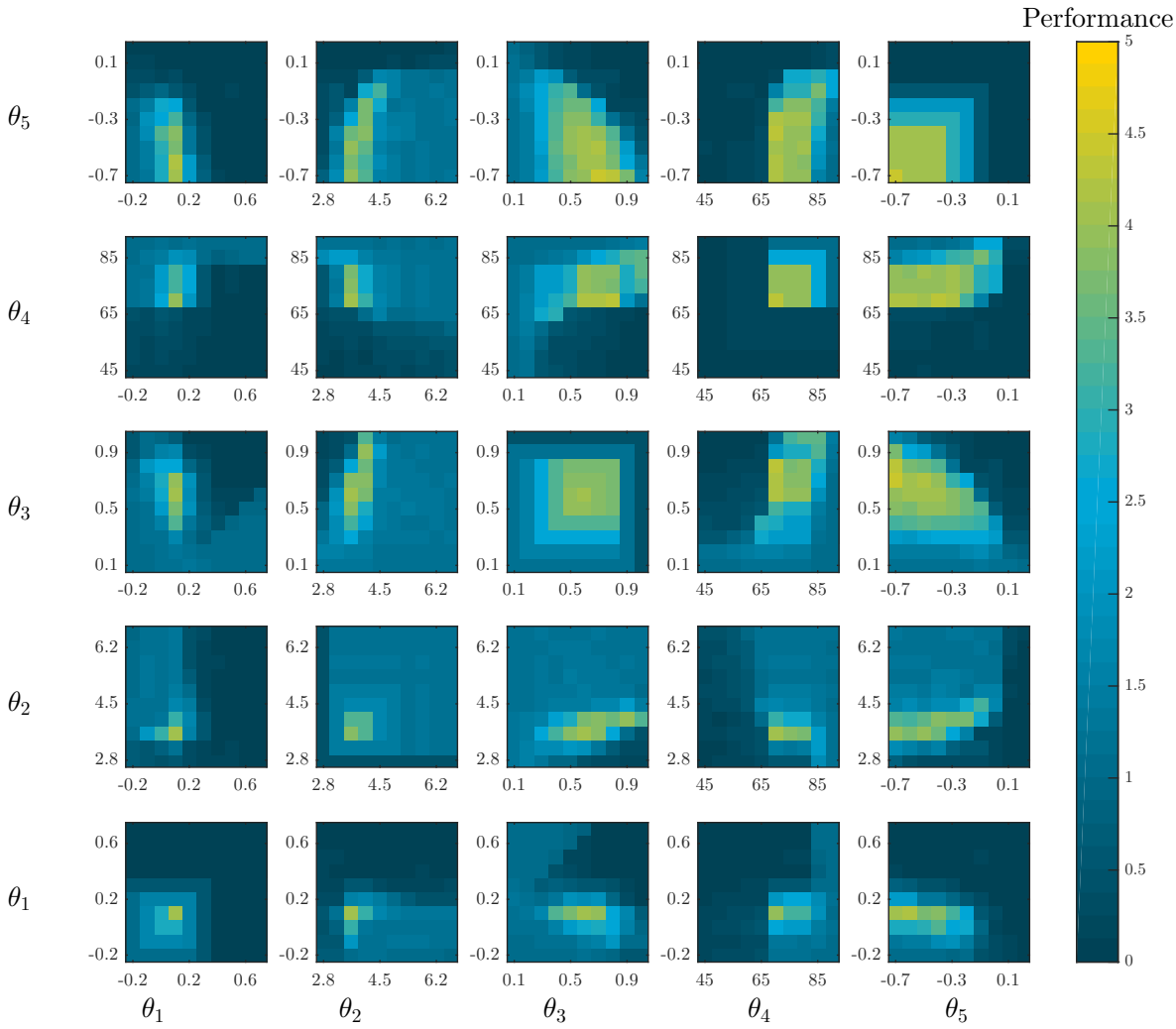


Figure 5-5: Coupling of behavior tree parameters around the optimized policy for Environment 1.

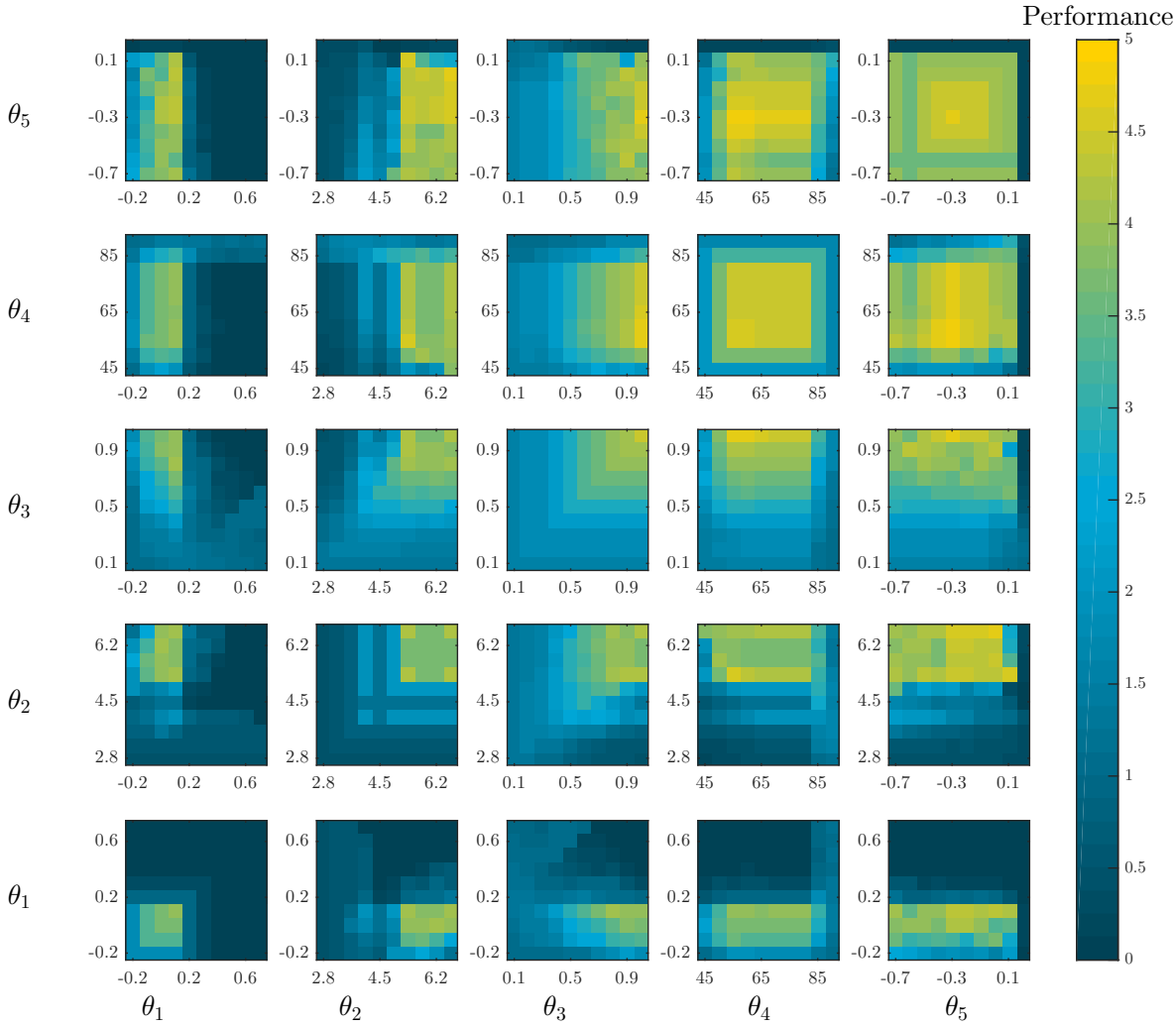


Figure 5-6: Coupling of behavior tree parameters around the optimized policy for Environment 2.

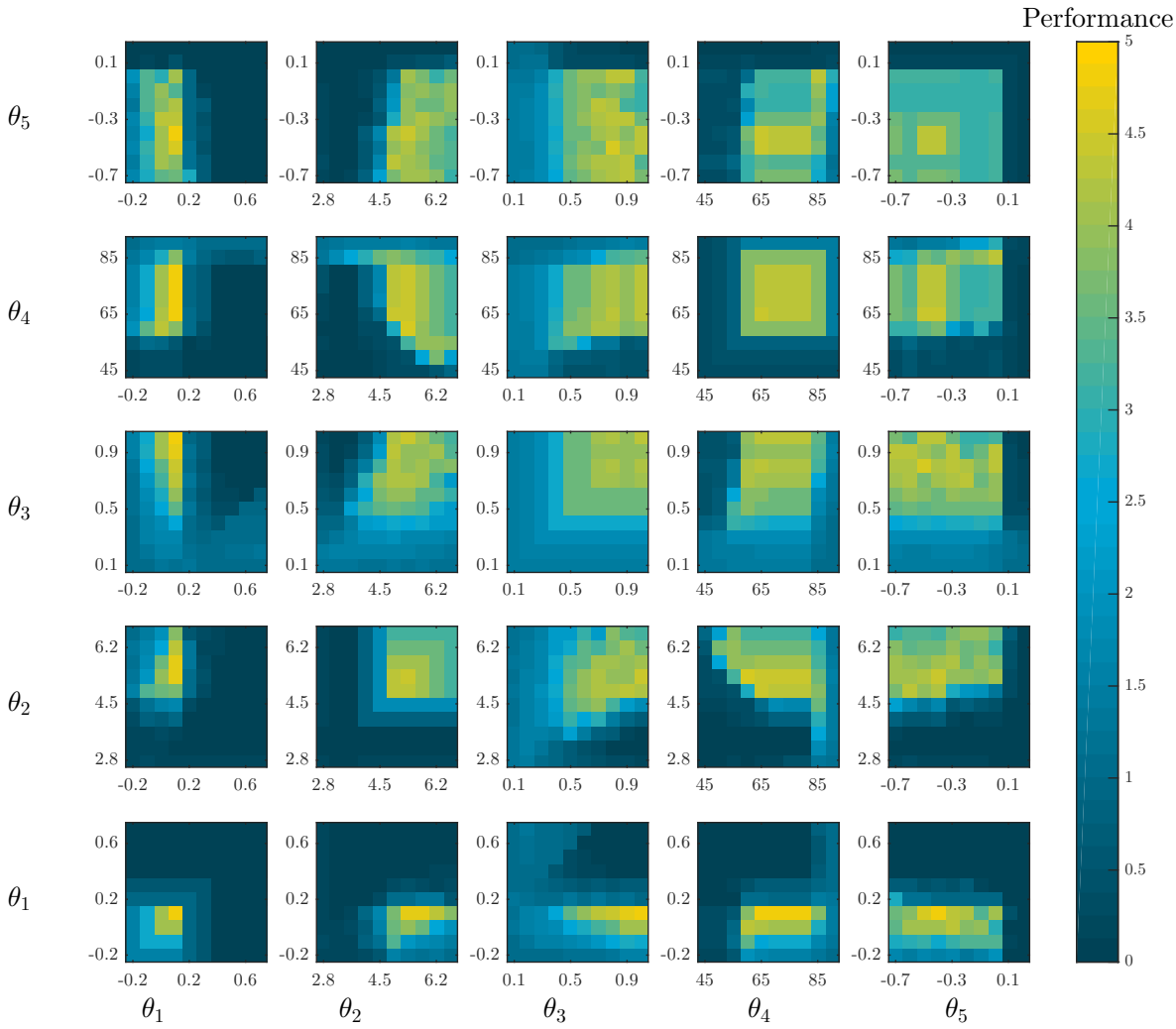


Figure 5-7: Coupling of behavior tree parameters around the optimized policy for Environment 3.

Chapter 6

Extended Analysis of the DIPS Algorithm

Part I of this report provided an overview of DIPS and analyzed how the algorithm improved the policy for all three adapted environments. It was shown that the update strategy and the direction array influence the algorithm performance in terms of final policy success rate, consistency and required number of evaluations. This Chapter provides an extended analysis of DIPS by assessing: (1) an alternative policy update strategy for DIPS; (2) automated formulation of the direction array through sampling; and (3) the result of employing a naive direction array during the learning process. All three analyses employ the same task environments and DIPS settings as described in Part I, unless otherwise specified.

6-1 Alternative Policy Update Strategies for DIPS

DIPS updates the policy parameters proportionally to the performance increase of the exploration policy for the respective node. The performance increase of by the exploration policy with respect to the baseline is normalized by the absolute value of the total relative performance increase for all nodes. As a result, the policy parameters can be updated towards the direction specified by the direction array, but also in the reversed direction if the performance increase of the exploration policy is negative. Because the sign of the policy updates can be different than the sign of the respective element in the direction array, this update approach is referred to as *polytonic*. On the contrary, a *monotonic* update strategy only allows policy updates in the direction specified by the elements of D. This section presents and compares the results of mono- and polytonic updates and assess the influence of a performance increase threshold for both update strategies. First the function of a performance threshold is discussed after which the polytonic and monotonic update strategies are presented and compared.

6-1-1 Performance Increase Threshold

Experiments are influenced by noise. For this task noise represents the randomness of the task environment which influences the performance of a policy. In this task the initialization location is found to be the largest contributor to the noise. As such, it can occur that for at least one of the t trials of the exploration policy performs worse than the baseline policy, despite that the parameter is incremented towards the direction specified in D. The reduced performance originates from the coupling between the nodes and the random initialization location.

To reduce the influence of noise, a threshold is added. The threshold attempts to increase the signal-to-noise ratio of the exploration policy reward signals. Exploration policies are evaluated over t trials; the agent receives t rewards r_t. The performance of the exploration policy is obtained by summing all trial rewards. Subsequently, the performance increase is calculated by subtracting the performance of the exploration policy by the performance of the current policy. If the absolute value of the performance increase is greater than the value specified by the threshold T_{min}, the signal-to-noise ratio of the feedback signal is considered high enough to be used for policy updates. Addition of a threshold is expected to reduce the dependency on the initial conditions of the trials and therefore increase the consistency of the DIPS algorithm, however, also requires the specification of an extra hyper-parameter.

6-1-2 Monotonic Parameter Updates

The initial version of DIPS developed in this research updated the policy paramters monotonically. Monotonic updates implies that the policy parameters are only updated towards the direction specified by the increment direction array D. This section provides a description of monotonic DIPS, discusses the advantages and disadvantages and presents the learning results of applying monotonic DIPS with and without performance increase threshold to the three adapted environments.

Description of Monotonic DIPS

Monotonic parameter updates are achieved by adapting steps 3 and 4 of the algorithm descriptive in Part I. From the pseudo-code of Algorithm 6-1 it can be seen that monotonic DIPS realizes the monotonic updates by only considering the exploration policies with a positive performance increase to construct the performance normalization factor P. As a result, only the policy parameters corresponding to the exploration policies with a positive performance increase are updated. Especially during the episodes where a high-performance policy is obtained this is expected to reduce the noise sensitivity and therefore increase convergence rate.

Advantages and Disadvantages

Monotonic DIPS ensures that the policy parameters are only updated in the direction specified by the direction array elements. As such, the main advantage is the speed up of the learning process as the parameter coupling effects are neglected. For example, if the performance increase of a parameter is initially unobservable and may only be experienced when a

Algorithm 6-1: DIPS algorithm following a monotonic parameter update scheme.

function SEARCH($\theta_{initial}, D, \epsilon, \eta$) $\vdots$
  repeat for every episode
    Evaluate performance of current policy ▷ Step 1
    for all nodes N **do**
      Generate exploration policy ▷ Step 2
      Evaluate exploration policy t times to obtain $r_n(t)$
    end for
    Calculate performance difference w.r.t. current policy: $\Delta p(n)$ ▷ Step 3
    if $0 \leq \Delta p(n) < T_{min}$ **then**
      Assume difference is caused by noise: $\Delta p(n) = 0$
      Increase exploration step-size
    else if $\Delta p(n) < 0$ **then**
      Neglect exploration policy result $\Delta p(n) = 0$
      Decrease exploration step-size
    end if
    Calculate normalization factor: $P = \sum_{n=1}^{N} \Delta p(n)$ ▷ Step 4
    Update parameters: $\theta(n) = \theta(n) + D(n) \cdot \varepsilon(n) \cdot \frac{\Delta p(n)}{\Delta P}$ ▷ Step 5
  until stopping criteria is met
end function

different parameter is updated, monotonic DIPS does not update the respective parameter if a performance decrease is observed. This can result in a stable learning process but requires the designer to be certain of the direction array.

Monotonic updates, however, also have three main disadvantages. First, the algorithm has no mechanism to recover when it overshoots the optimal parameter setting. If the algorithm has overshoot the optimal parameter setting it may only improve performance by adapting a different parameter to to compensate for the overshoot. Second, when no threshold is added the algorithm will leverage noise to update the policy parameters. As a result the algorithm can continue updating the policy parameters whilst the updates do not result in a policy improvement. These unreliable updates can cause the algorithm to diverge. Finally, monotonic DIPS is fully dependent on the correct specification of the direction array elements. Incorrect specification can result in no or erroneous parameter updates and monotonic DIPS has limited mechanics to compensate for this.

Results

To assess the influence of the performance increases threshold and evaluate the impact of monotonic parameter updates, monotonic DIPS is applied to the same environments and algorithm settings as described in Part I. The resulting learning process is visualized in Figures 6-2 and 6-3. From these figures the following observations can be made:

- **Addition of a performance threshold generally increases the number of evaluations and the spread of both the parameter settings and performance**. The

performance increase threshold prevents the monotonic DIPS to leverage the noise of the initialization locations during the early episodes. Therefore, DIPS requires an increased number of episodes to increase the increment step size and initialize the policy update process. In term, this results in an increased uncertainty and therefore higher spread in both the parameter settings and performance.

- **The policy performance decreases after a maximum performance has been achieved.** This performance degradation is the result of overshooting the optimal value. Overshooting originates from the continued policy parameter updates of monotonic DIPS as a result of the randomized initialization locations. This noise allows the exploration policies to be more effective than the baseline policy for some initializations; because the updates are monotonic, the algorithm is unable to compensate for the overshoot. The current implementation of the performance increase threshold is unable to stabilize the learning process because of the increased policy rewards obtained during the later episodes.

6-1-3 Polytonic Parameter Updates

Part I provided an elaborate discussion of polytonic DIPS with a performance increase threshold. For completeness the five steps of the DIPS algorithm are provided in Algorithm 6-4. The remainder of this section discusses the advantages and disadvantages of polytonic parameter updates and presents the results of the learning process for polytonic updates with and without a performance increase threshold.

Advantages and Disadvantages

Polytonic DIPS has two main advantages over monotonic parameter updates. First, polytonic policy updates reduce the influence and dependency on specifying a correct D array. Exploration policies are generated based on the designers understanding of the behavior and expected direction in which the policy parameters have to be updated in order to improve a behavior. In this case the direction array D adds a bias to exploration. If this bias is incorrect the policy parameters can be updated in the correct direction. Incorrect direction array elements, however, are expected to result in more episodes until the policy converges because a policy performance decrease is more difficult to observe than a performance increase when the initial policy also is of low quality. Second, polytonic updates allow the algorithm to recover from overshooting the optimal parameters settings, following the same reasoning as the first advantage.

A disadvantage of polytonic updates is that they are expected to be more sensitive to noise. Combined, the sensitivity to noise and the reduced influence of D, allow polytonic updates to incorrectly update the policy due to noise. The addition of T_{min} can compensate for this, however, the influence of this threshold diminishes when higher-performance policies are obtained. Also, this effect is enhanced by reducing the increment step-size if a performance decrease is observed in Step 4. In this case the step-size decreases rapidly for the nodes with a reversed update direction, increasing the number of required episodes.

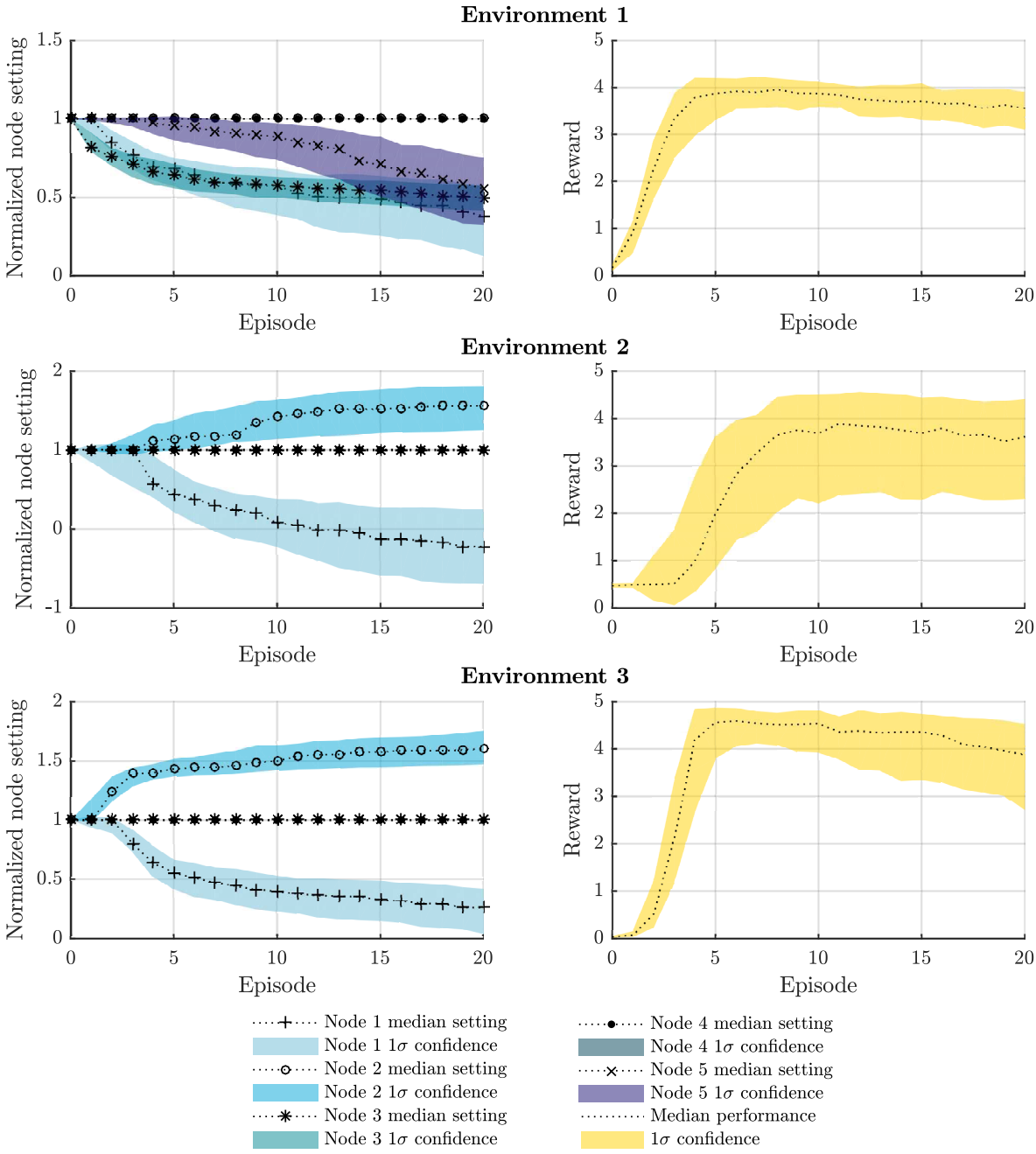


Figure 6-2: Learning process of Directed Increment Policy Search using the monotonic update strategy without a performance increase threshold for the three adjusted environments. The standard deviation and median parameter setting and performance have been calculated over 60 independent runs. The parameter settings are normalized with respect to the initial settings at episode 0; the reward is obtained by evaluating the policy at the corresponding episode over 100 initializations.

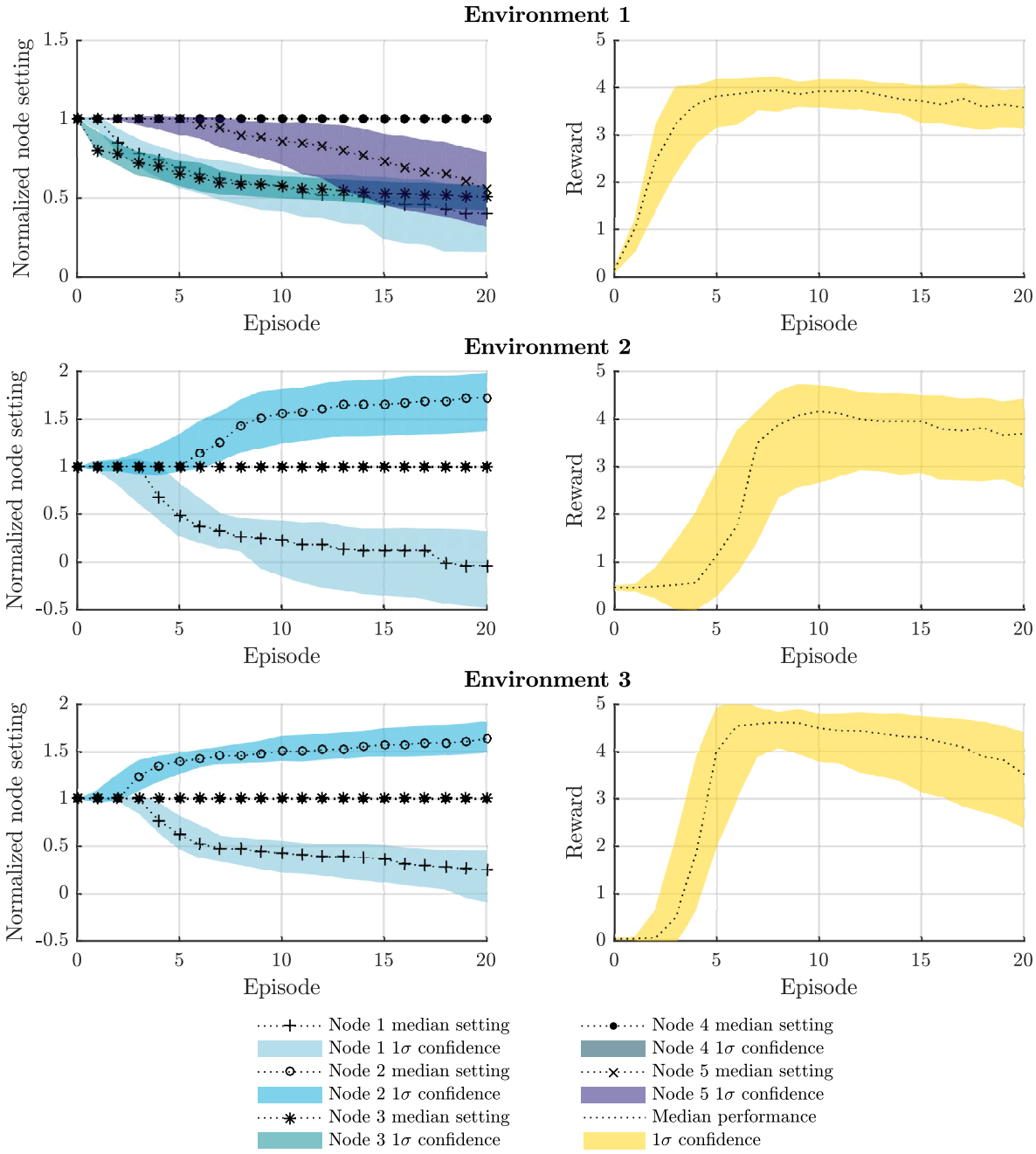


Figure 6-3: Learning process of Directed Increment Policy Search using the monotonic update strategy with a threshold of $T_{min} = 0.4$ for the three adjusted environments. The standard deviation and median parameter setting and performance have been calculated over 60 independent runs. The parameter settings are normalized with respect to the initial settings at episode 0; the reward is obtained by evaluating the policy at the corresponding episode over 100 initializations.

Algorithm 6-4: DIPS algorithm following a polytonic parameter update scheme.

function SEARCH($\theta_{initial}, D, \epsilon, \eta$) $\vdots$
  repeat for every episode
    Evaluate performance of current policy ▷ Step 1
    for all nodes N **do**
      Generate exploration policy ▷ Step 2
      Evaluate exploration policy t times to obtain $r_n(t)$
    end for
    Calculate performance difference w.r.t. current policy: $\Delta p(n)$ ▷ Step 3
    Update step size of increments
    if $-T_{min} < \Delta p(n) < T_{min}$ **then**
      Assume difference is caused by noise: $\Delta p(n) = 0$
      Increase exploration step-size
    else if $\Delta p(n) \leq -T_{min}$ **then**
      Decrease exploration step-size
    end if
    Calculate normalization factor: $P = \sum_{n=1}^{N} |\Delta p(n)|$ ▷ Step 4
    Update parameters: $\theta(n) = \theta(n) + D(n) \cdot \varepsilon(n) \cdot \frac{\Delta p(n)}{\Delta P}$ ▷ Step 5
  until stopping criteria is met
end function

Learning Process Results

Figures 6-5 and 6-6 display the learning process of applying polytonic DIPS without threshold and with threshold, respectively, to the three adapted environments using the DIPS settings of Part I. By comparing both figures, the following observations can be made:

- **Addition of a threshold generally reduces the spread of both the parameter values and the performance of the policy during all episodes.** The increased consistency of the algorithm originates from the reduced vulnerability to noise. Therefore the algorithm has an increased certainty that the update results in a higher-performing policy. This is especially true for the early episodes; the reduced standard devition in the later episodes is expected to originate from the reduced spread in the early episodes.
- **The median performance of the final policy is also higher for polytonic DIPS with threshold as compared to polytonic DIPS without threshold for all environments.** This is especially true for Environment 2. When applying polytonic DIPS without threshold to Environment 2, the algorithm appears to mainly update the parameters based on noise. Only a slight performance increase over is witnessed over 20 episodes. Polytonic DIPS requires more episodes for Environment 2 than the other environments because the optimal parameters for Environment 2 are located at a larger distance from the initial values.

From these observation we conclude that the addition of a (small) threshold to polytonic DIPS results in a more consistent learning process and higher-performing final policy. Combined

with the sensitivity anlysis of Part I, the required number of episodes until the policy converges appears to be affected more by ϵ and η, except for Environment 2. This is the result of the reduced effectiveness of a threshold at high-performing policies. As such, the convergence rate remains similar for both tested cases.

6-1-4 Comparing Poly and Monotonic DIPS

Table 6-1 lists the learning process results in terms of policy performance, performance standard deviation and number of episodes for monotonic and polytonic DIPS and both threshold settings. These metrics provide an estimate of the performance, consistency and practical applicability of both algorithms. From the table and previous figures the following conclusions may be made:

- **Monotonic and polytonic parameter updates achieve an equal final policy performance.** The performances of the final policies is approximately equal for all three adapted environments. Polytonic DIPS without performance increase threshold for Environment 2 is the only outlier for the performance metric. The reason for this reduced performance was already discussed in the previous section. The discrepancies in the performances of the other environments are expected to be a result of the limited number of samples used to evaluate the policy performance.
- **Monotonic parameter updates reduce the number of convergence episodes.** The increased learning rate is the result of the reduced influence of parameter coupling and, for the case where $T_{min} = 0$, leveraging of the noise to update the policy parameters.
- **Polytonic DIPS facilitates a more robust learning process.** This increased robustness is expressed through two observations. First, the performance of the policy remains constant during the episodes after an optimum is found. Because polytonic DIPS is able to update the policy parameters in the opposite direction as specified in D, the algorithm is able to correct for unreliable parameter updates during the later episodes as the result of noise experienced. Second, for Environment 1 polytonic DIPS updates θ_5 in a reversed direction as specified in D, demonstrating the reduced dependency on the correctness of the direction array elements.
- **Performance threshold has a reversed effect for monotonic and polytonic DIPS.** For monotonic DIPS the addition of a performance threshold reduces the consistency and convergence rate of the algorithm. Conversely, polytonic DIPS benefits from the threshold in terms of policy performance, consistency and number of required episodes. From this it can be concluded that the performance of the monotonic DIPS algorithm is dependent on leveraging the noise of the learning process. This is undesirable as the algorithm will be less dependent on the actual signal from the environment, reducing reliability.

Polytonic updates can be regarded as a more general form of the DIPS algorithm form than monotonic updates. We believe polytonic DIPS is the superior algorithm because it allows for greater robustness, despite that monotonic updates outperform polytonic updates for some environments. Following these conclusions the next sections will only consider polytonic DIPS.

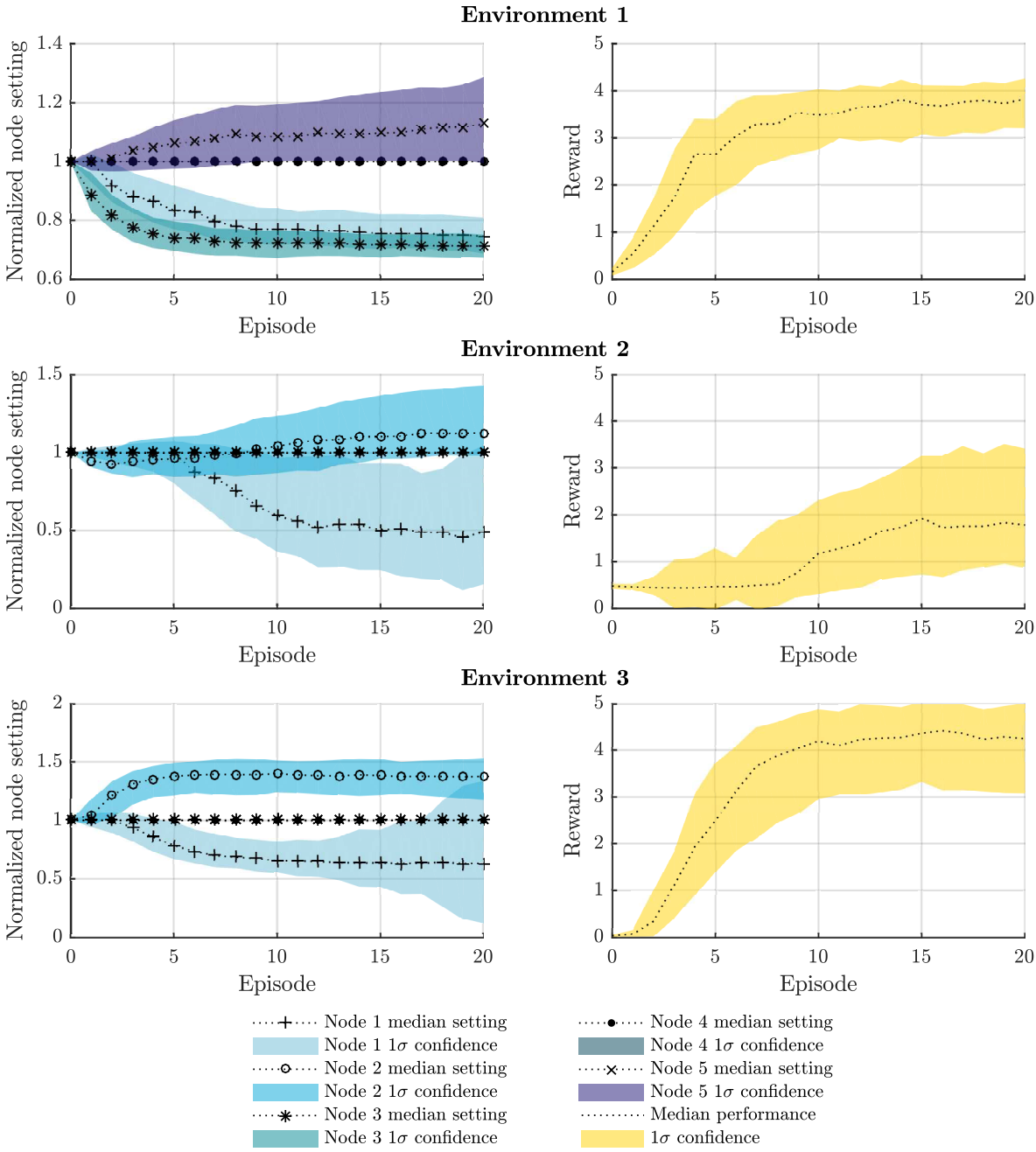


Figure 6-5: Learning process of Directed Increment Policy Search using the polytonic update strategy without a threshold for the three adjusted environments. The standard deviation and median parameter setting and performance have been calculated over 60 independent runs. The parameter settings are normalized with respect to the initial settings at episode 0; the reward is obtained by evaluating the policy at the corresponding episode over 100 initializations.

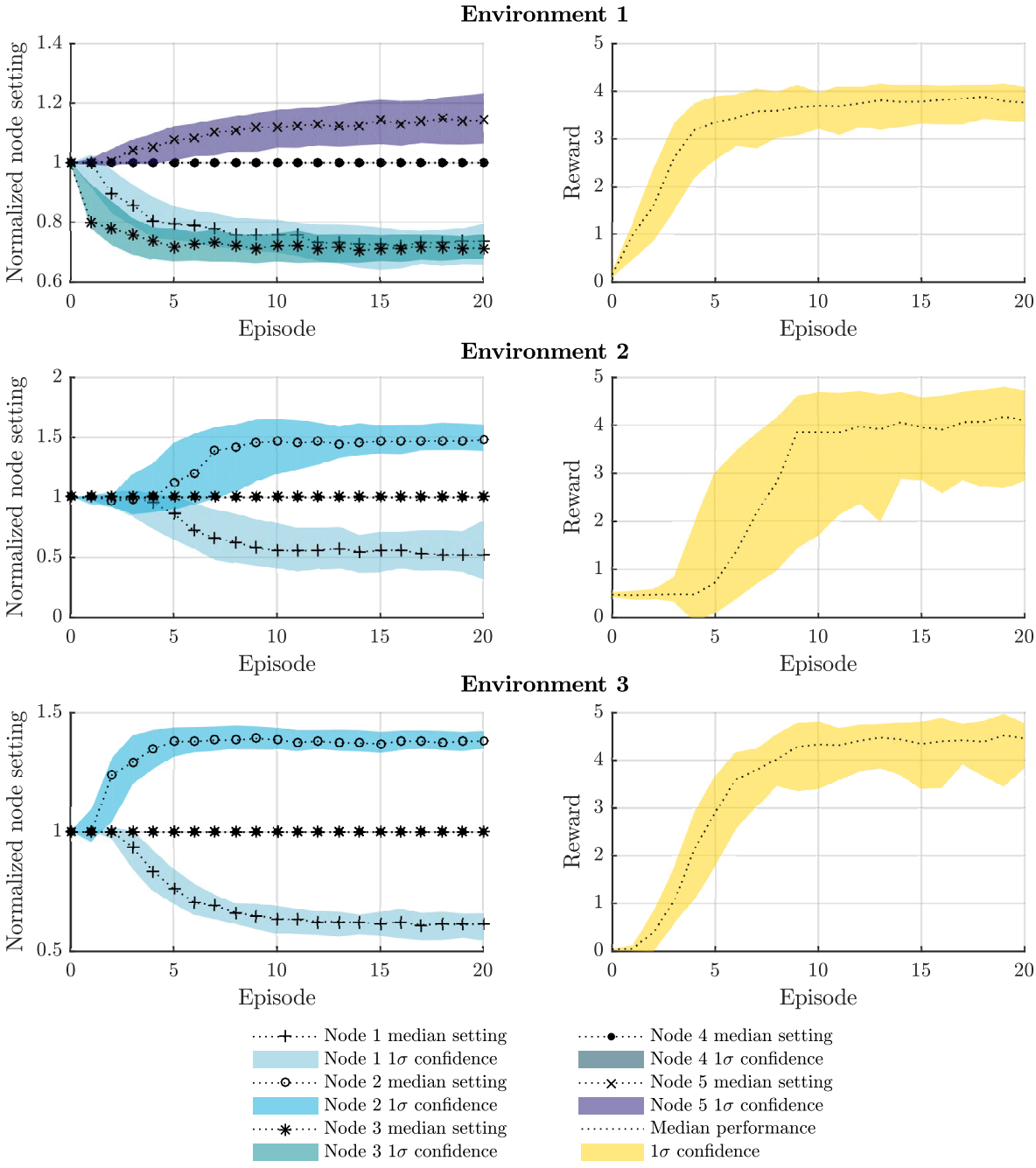


Figure 6-6: Learning process of Directed Increment Policy Search using the polytonic update strategy with a threshold of $T_{min} = 0.4$ for the three adjusted environments. The standard deviation and median parameter setting and performance have been calculated over 60 independent runs. The parameter settings are normalized with respect to the initial settings at episode 0; the reward is obtained by evaluating the policy at the corresponding episode over 100 initializations.

Table 6-1: Learning process results of monotonic and polytonic DIPS compared for the three adapted environments.

Environment	Metric	Polytonic		Monotonic	
		$T_{min} = 0$	$T_{min} = 0.4$	$T_{min} = 0$	$T_{min} = 0.4$
Environment 1	P	3.72	3.76	3.79	3.81
	σ_P	0.49	0.52	0.62	0.51
	# episodes	220	180	**80**	100
Environment 2	P	1.92	3.91	3.76	4.08
	σ_P	1.27	1.57	**1.09**	**1.08**
	# episodes	225	135	135	135
Environment 3	P	4.2	4.37	4.56	4.54
	σ_P	0.96	0.72	**0.53**	0.99
	# episodes	150	135	**75**	90

6-2 Direction Array Sampling

This research employed a BT to represent the robotic behavior. Because BTs facilitate human interpretation of the encoded behavior, the direction array could be constructed using the knowledge of the behavior and the changes to the environments. Manually formulating a direction array, however, has two main limitations. First, the designer is required to understand the behavior. Other behavior representation methods, such as ANNs, are much less suited for human interpretation. Also, the interpretable structure of BTs reduces if the task becomes more complex or the tree size increases. Second, the direction array elements have a large influence on the learning process; this is especially true for the zeros in D. If the direction array contains a zero for an important parameter, DIPS may be unable to increase the policy performance.

Sampling the elements of D may therefore be a promising solution to this problem. By sampling the individual policy parameters the direction array may be formulated in an automated fashion and is no longer dependent on the designer. Compared to the current DIPS implementation sampling results in a more generalized form of the DIPS algorithm. This section proposes a simple version of DIPS using sampling and presents the results of applying this algorithm to the three adapted environments.

6-2-1 Proposed Algorithm

The goal of this analysis is to demonstrate that sampling can be a viable method to construct the elements of the direction array. As such, a simple algorithm is implemented in polytonic DIPS; this algorithm is presented in Algorithm 6-7. The *Sample* function is called at the start of the learning process and after every three episodes. Re-sampling after a number of episodes is required to mitigate the coupling effects: the effect of adapting a policy parameter may only become apparent after a different parameter is updated. Furthermore, the *Sample* function is contained in a while-loop which ensures that sampling continues until at least one direction array element is nonzero.

Algorithm 6-7 is called between lines 3 and 4 of Algorithm 6-4 as it requires the performance

Algorithm 6-7: Proposed algorithm for direction array element sampling.

function SAMPLE(n, P)
    Generate 2 policies for parameter n: positively and negatively incremented ▷ Step I
    Evaluate the 2 sampling policy t times to obtain p_n^+ and p_n^- ▷ Step II
    if $p_n^+ - P \geq T_{min}$ **then** ▷ Step III
        Set $D_n = 1$
    else if $p_n^- - P \geq T_{min}$ **then**
        Set $D_n = -1$
    else
        Set $D_n = 0$
        Update increment step-size of parameter n ▷ Step IV
    end ifreturn D_n
end function

of the current policy. The algorithm requires similar inputs as polytonic DIPS, with the addition of the parameter index and the current policy performance. As such, the algorithm sequentially samples the direction array element for all policy parameters n in four steps.

Step I generates two exploration policies: a policy where parameter n is incremented positively and one where parameter n is incremented negatively by ε. The direction array elements are formulated based on experienced policy performance increases only because a performance increase contains higher quality information compared to a policy performance decrease. This is different compared to the policy update step of polytonic DIPS, where a policy par mater may be updated based on a performance decrease. Sampling therefore requires two exploration policies to find the greatest policy increase. Step II evaluates the two policies with either a positively or negatively incremented parameter n over t trials to find the performance of both policies. The direction array elements are assigned in Step III based on the performance of the incremented policies. A performance increase threshold is employed here to increase the signal-to-noise ratio. Step IV assigns a zero to the respective element in the direction array if no performance increase is achieved by either of the two incremented policies. Additionally, this step updates the increment step-size of the respective parameter. Here the increment step-size is increased by a factor η if the sampling policies achieve an equal performance as the baseline policy and the step-size is decreased if the exploration policies result in a reduced policy performance.

6-2-2 Learning Process Results

Figure 6-8 presents the learning process results of applying polytonic DIPS with performance increase threshold and direction array sampling to the three adapted environments. In the current application the direction array elements are re-sampled every three episodes. From Figure 6-8 and Table 6-2 the following observations can be made:

- **Direction array sampling increases the policy performance without any dependency on a human designer.** Direction array sampling is able to increase the policy performance for all environments. Compared to the polytonic DIPS without direction array sampling the performance of the final policy is reduced. The reduced

Table 6-2: Learning results of DIPS for the three adapted environments using Polytonic DIPS with direction array sampling.

Name	Initial reward	Final reward	Number of evaluations	Initial success	Final success	$\theta(n)$				
Baseline	3.8	–	-	72.4%	-	[0.20	3.67	1.00	69	− 0.40]
Env. 1	0.2	3.5	560	3.0%	64.8%	[0.14	3.74	0.72	69	− 0.43]
Env. 2	0.5	3.3	762	2.2%	61.6%	[0.12	5.05	0.78	69	− 0.44]
Env. 3	0.1	3.6	422	1.0%	67.4%	[0.12	4.77	0.81	69	− 0.42]

performance is expected to originate from the parameter couplings. For example, in all adapted environments the setting of node 3 is reduced initially to compensate for the increased maximum rudder deflection. The resulting behavior allows the DelFly to approach the window. Initially this results in a performance increase because the reward for crashing is greater than for running out of time. In later episodes, however, the reduced setting of this node reduces the effectiveness of the wall avoidance behavior. It can be seen that DIPS increases the parameter setting of this node after other nodes have been updated to compensate for the reduces effectiveness of this sub-behavior. In the final episode the setting remains lower than for the initial setting, suggesting that DIPS is not yet converged or is stuck in a local minimum.

- **Direction array sampling reduces the reliability of the learning process.** The increased spread of the parameter settings, policy performance and number of episodes indicates that the learning process is less reliable compared to polytonic DIPS without sampling. The increased spread originates from of the increased uncertainty of the increased number of episodes required for sampling of the direction array elements.

- **Direction array sampling requires an increased number of evaluations.** For the current task and algorithm settings sampling requires an additional 55 evaluations every 3 episodes. In addition to these added evaluations DIPS with sampling has requires more episodes to converge than DIPS without sampling. This increased number of episodes especially applies for Environments 1 and 2 and is the result of the increased uncertainty. For the first episode Environments 2 and 3 require three sampling runs to formulate an initial direction array.

These observations demonstrate that DIPS can be applied to improve a policy without any dependency on a pre-specified direction array, providing a generalized form of the DIPS algorithm. The current implementation of direction array sampling may be considered as a simple and naive formulation. Many improvements may increase the reliability and convergence rate of polytonic DIPS with direction array sampling. For example, the previous direction array may be employed to bias the formulation of a new direction array or a form of correlated direction array sampling may reduce the need to evaluate every policy parameter individually with both positive and negative increments.

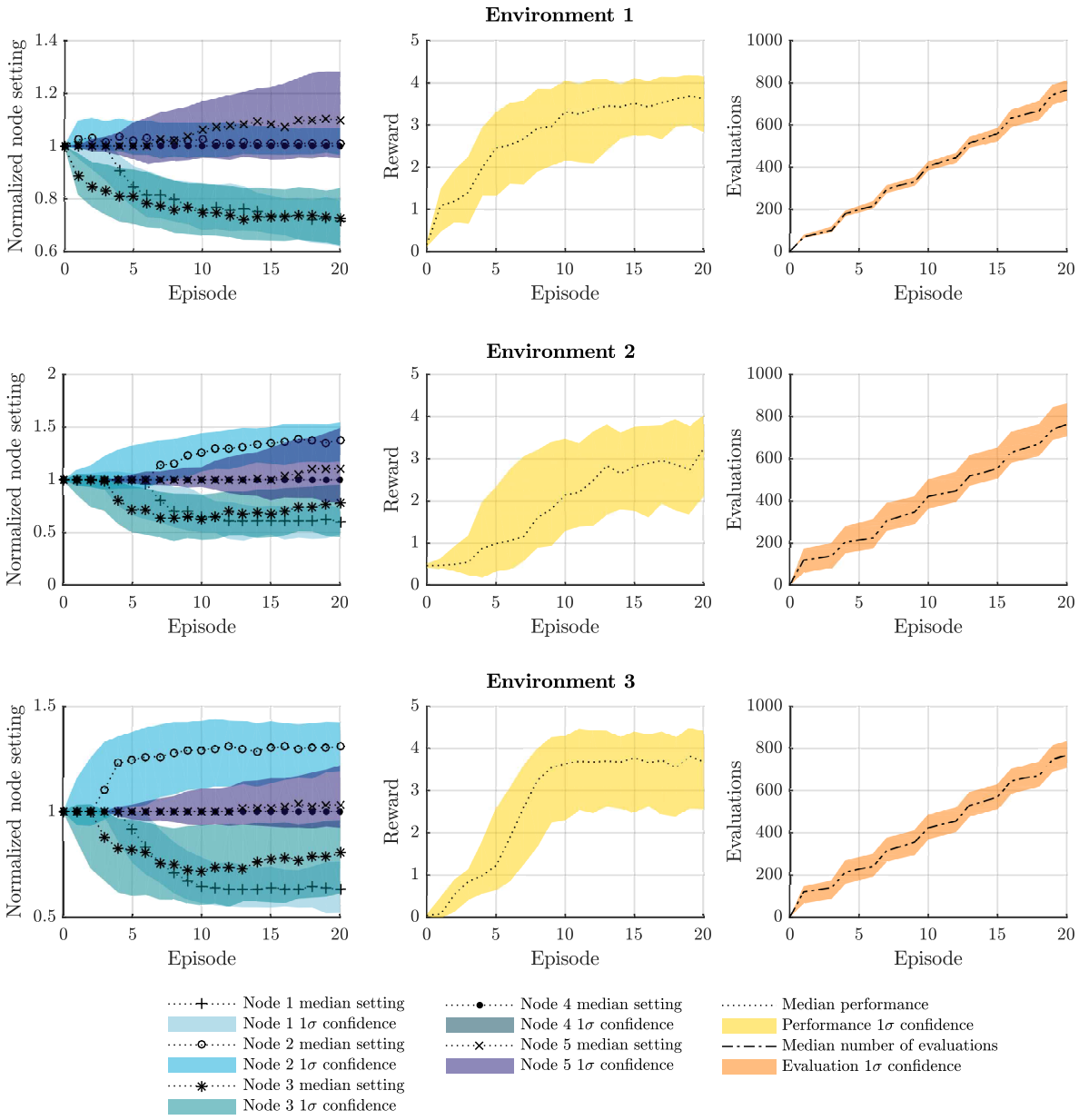


Figure 6-8: Learning process of Directed Increment Policy Search using the polytonic update strategy with a threshold of $T_{min} = 0.4$ for the three adjusted environments. The standard deviation and median parameter setting and performance have been calculated over 60 independent runs. The parameter settings are normalized with respect to the initial settings at episode 0; the reward is obtained by evaluating the policy at the corresponding episode over 100 initializations.

Table 6-3: Learning results of DIPS for the three adapted environments using Polytonic DIPS with a naive direction array.

Name	Initial reward	Final reward	Number of evaluations	Initial success	Final success	$\theta(n)$				
Baseline	3.8	–	-	72.4%	-	[0.20	3.67	1.00	69	− 0.40]
Env. 1	0.2	3.8	480	3.0%	71.8%	[0.15	3.83	1.00	77	− 0.42]
Env. 2	0.5	3.6	450	2.2%	68.2%	[0.08	5.21	1.00	79	− 0.46]
Env. 3	0.1	4.2	540	1.0%	86.4%	[0.12	5.02	1.00	76	− 0.44]

6-3 Naive Direction Array Formulation

The previous section demonstrated that the direction array can be formulated through interaction with the environment. Sampling removes the need of specifying a direction array, however, resulted in a learning process with reduced performance. Pre-specifying a direction array has been shown in Part I to improve the performance of the learning process, however, is dependent on the formulation of a correction direction array. This section provides an alternative approach: a naive direction array. Here, a naive direction array consists of ones only. Figure 6-9 presents the learning process of polytonic DIPS with performance increase threshold using a naive direction array for the three adapted environments. Comparing the results of this figure to the results of Figure 6-6 and Table 6-3 the following observations can be made:

- **A naive direction array is able to increase policy performance.** Employing a naive direction array results in a final policy of equal performance as compared to polytonic DIPS with a direction array specified through the designers understanding of the behavior and environment. This demonstrates the robustness of polytonic DIPS. Compared to the pre-specified direction array of Part I, however, the learning process differs slightly. For this direction array the third parameter may not be updated as this parameter is already at its maximal value. As such, exploration policies of this parameter do not yield additional information. This impairment is mitigated by parameter 4. Parameter 4 initializes the learning process in a similar fashion as parameter 3 in the previous section. Again, this parameter is reduced during the later episodes in Environment 2 and 3. For Environment 1, however, parameter 4 remains constant to cope with the inability to reduce parameter 3 for this naive direction array.

- **Naive sampling reduces the consistency of the learning process.** The increased spread of the parameter setting and policy performance results in a reduced consistency. The increased spread originates from the increased number of parameters considered in the optimization process.

- **Naive sampling increases the number of evaluations**. This observation also follows from the increased number of parameters considered by DIPS. Additionally, required number of episodes is increased as DIPS has to recover from initial parameter updates which limit the performance in later episodes, as the result of coupling.

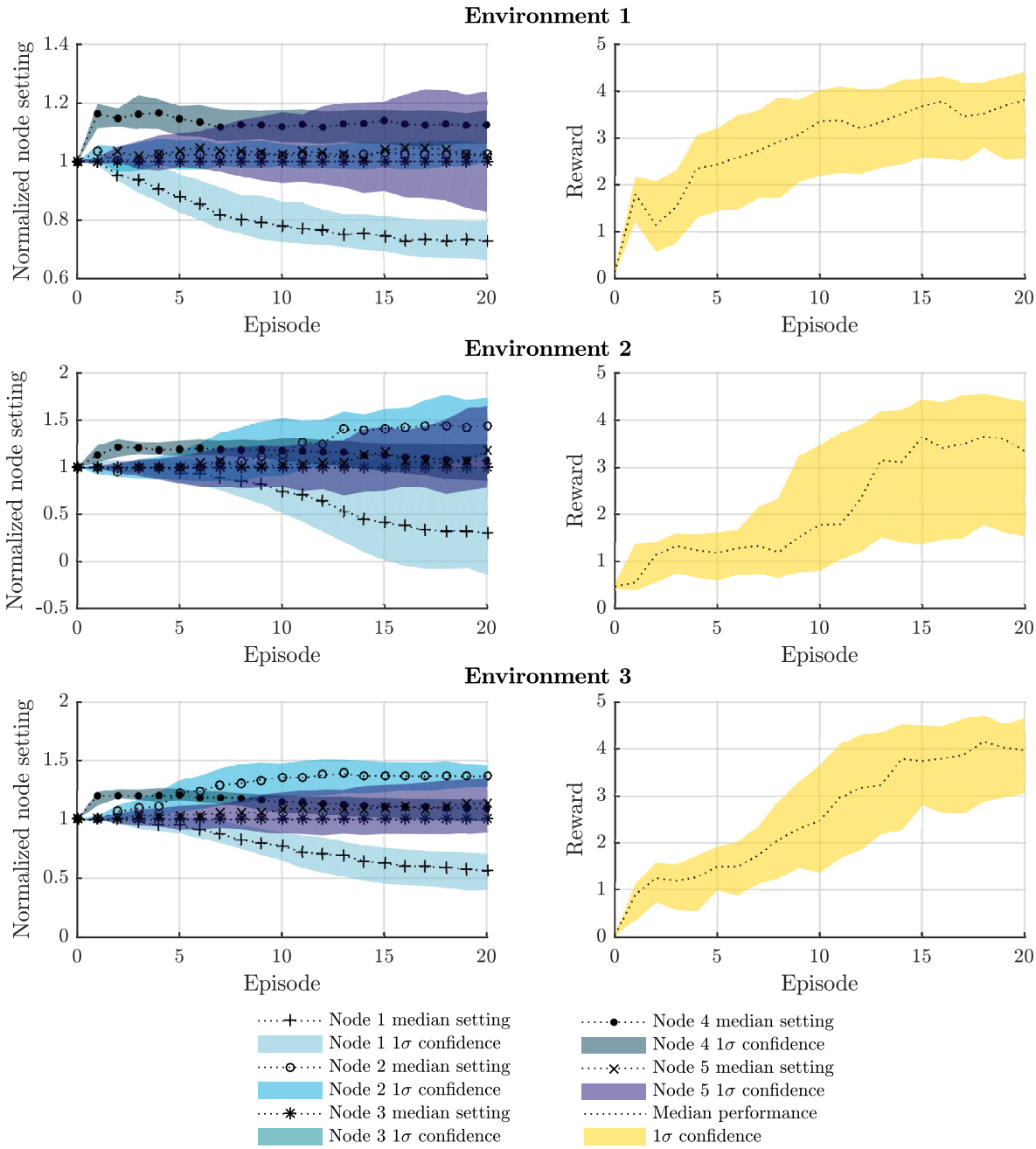


Figure 6-9: Learning process of Directed Increment Policy Search using the polytonic update strategy with a threshold of $T_{min} = 0.4$ for the three adjusted environments. The standard deviation and median parameter setting and performance have been calculated over 60 independent runs. The parameter settings are normalized with respect to the initial settings at episode 0; the reward is obtained by evaluating the policy at the corresponding episode over 100 initializations.

Chapter 7

Additional Environments

Part I and the previous two chapters demonstrated the effectiveness of DIPS in updating the policy parameters to increase performance in three adapted environments. This chapter evaluates the effectiveness of DIPS by evaluation three alternative large reality gaps to find a limit to which DIPS can adapt the policy parameters to express a high-performing behavior. This analysis employs polytonic DIPS with direction array formulation through analysis of the adaptations to the environment.

7-1 Description of Additional Adapted Environments

This analysis considers three additional adapted environments with large reality gaps. All three environments are adapted from the baseline environment described in Part I. The additional environments, named Environments 4 through 6, function as additional measure of robustness to discrepancies between simulation and the real-world.

In Environment 4 the DelFly dynamics are adapted to allow a maximum rudder deflection of 0.26 radians, similar as in Environments 1-3. Additionally, the flight speed of the DelFly is doubled to 1.0 m/s. The increased deflection causes the turn radius to decrease, whereas the increased flight speed increases the turn radius. The higher flight speed also reduces the time to react to objects, in this case walls, in the environment. Environment 5 is similar to Environment 1, however, the room dimensions are increased instead to 10×10 meters. Combined with the increased maximum rudder deflection the larger room is expected to result in a large performance degradation. Environment 6 adopts the same adaptations as Environment 3, however, with a reversed room aspect-ratio. Table 7-1 summarizes the adaptations of Environments 4-6.

7-2 Behavior Analysis and Direction Array Formulation

Figure 7-1 shows three flight path of the DelFly for the baseline and addition three adapted simulation environments. From the flight paths it can be observed that the coupling between the sub-behaviors of the initial policy and the environments is less effective in the adapted

Table 7-1: Environmental variable settings of the baseline and three additional adapted environments.

Name	δ_{max} $[rad]$	Speed $[m/s]$	Room dimensions $[m \times m]$	Expected Reality Gap
Baseline	0.065	0.5	8×8	-
Environment 4	0.26	1.0	8×8	*Large*
Environment 5	0.26	0.5	10×10	*Large*
Environment 6	0.26	0.5	5×7	*Large*

Table 7-2: Parameter settings of DIPS for the three adapted environments.

Name	ϵ	η	T_{min}	$D(n)$
Environment 4	0.2	1.2	0.4	[1 -1 0 0 -1]
Environment 5	0.2	1.2	0.4	[-1 -1 -1 1 1]
Environment 6	0.2	1.2	0.4	[-1 1 0 0 0]

environments. This section briefly discusses the expressed behavior and constructs the direction array elements based on the behavior analysis. The resulting direction arrays for the three adapted environments are listed in Table 7-2.

In Environment 4 the window-search and approach behavior remain similar as for the baseline environment. The wall avoidance behavior, however, appears to be ineffective due to the increased flight speed for the given disparity value. To compensate for these changes the action nodes are increased to increase the responsiveness of the actions. Parameter 3, however, may not be increased further as it already operates at maximum setting. To account for the inability to adapt parameter 3, the disparity is reduced such that the wall avoidance behavior is initiated at a greater distance from the wall.

The increased room dimensions and increased rudder deflection of Environment 5 cause the DelFly to experience a lower disparity throughout the room. As a result, the DelFly circles the room center unable to locate the window due to the low window-response value of parameter 4. To correct for this, the window-response value is increased to activate the window approach behavior at the greater distance from the window. Additionally, the disparity is decreased to reduce the radius around the room center where the DelFly searches for the room. Finally, the action parameters are reduced to allow for a turn radius which is proportionally equal to that of the baseline environment of all actions.

The anti-symmetric room of Environment 6 causes the disparity measurements to differ in x and y-direction. The DelFly therefore does not return to the room center to search for the window. To compensate for these adaptations the same direction array as for Environment 3 is employed. Despite of the difference in aspect ratio, this direction array is expected to result in a behavior with improved performance. Here the disparity parameter is increased to account for the reduced room dimensions and the action node setting is reduce to facilitate a straight path during the window approach sub-behavior.

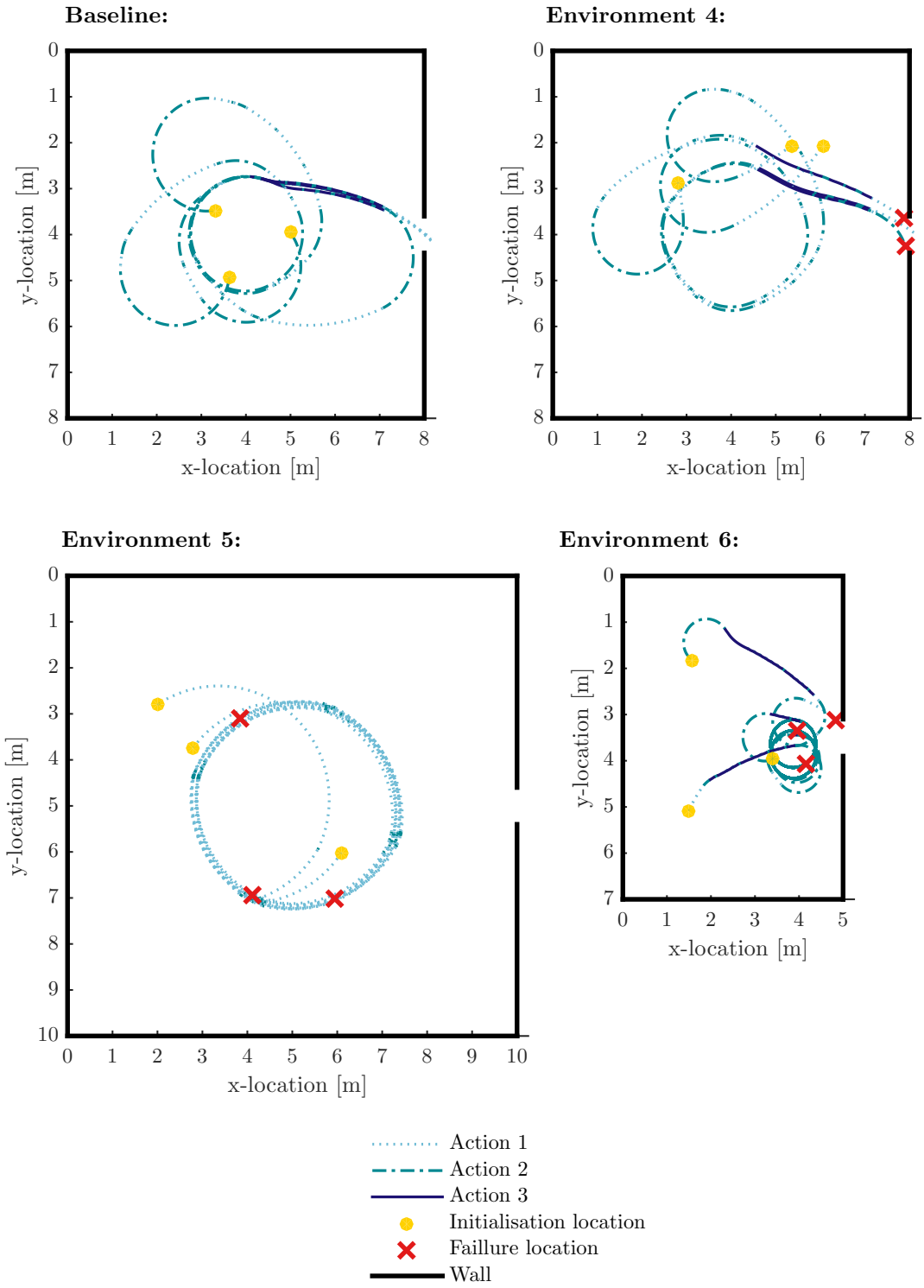


Figure 7-1: Three example paths of DelFly controlled by the original Behavior Tree settings for the baseline environment and Environments 4–6

Table 7-3: Learning results of DIPS for adapted environments 4–6.

Name	Initial reward	Final reward	Number of evaluations	Initial success	Final success	$\theta(n)$				
Baseline	3.8	–	-	72.4%	-	[0.20	3.67	1.00	69	− 0.40]
Env. 4	2.0	1.7	400	23.6%	18.8%	[0.20	3.85	0.97	69	− 0.36]
Env. 5	0.1	2.4	600	0.2%	45.2%	[0.18	3.44	0.92	76	− 0.40]
Env. 6	0.7	3.6	300	4.8%	67.8%	[0.12	4.77	1.00	69	− 0.40]

7-3 Learning Process Results

Figure 7-2 and Table 7-3 display the learning process and final results of applying polytonic DIPS with the settings of Table 7-2 to the additional adapted environments. Figure 7-3 visualizes three example paths of the DelFly in the adapted environments using the optimized policy parameters. From these results the following observations can be made:

- **DIPS is unable to increase policy performance for Environment 4.** During the learning process the policy performance remains constant, given the specified direction array and DIPS settings. This indicates that the reality gap is too large that behavior adaptation through policy parameters optimization only may not be applicable for this environment. It can be seen that the expressed behavior remains similar for the initial and final policy parameters. Possible a different behavior represented by an alternative behavior representation structure is required to achieve a high performance.
- **DIPS is able to increase policy performance for Environment 5.** DIPS manages to improve the success rate of the behavior from 0.2 to 45.2%. The final behavior resembles the behavior expressed in the baseline environment. This is achieved mainly by adapting the window-response parameter to facilitate window approaches. In this environment, however, the DelFly passes through the window at an increased angle. This makes the behavior less robust and is expected to result in a reduced behavior policy as compared to the baseline behavior.
- **DIPS is able to increase policy performance for Environment 6.** DIPS successfully adapts the policy parameters to achieve a similar performance as observed in the baseline environment. The final behavior now expresses a missed approach behavior which is unique in shape compared to the other environments. The reduced value of parameter 1 both facilitates the DelFly to find the window center when initialized at the top or bottom of the room and a straight window-approach behavior.

These results demonstrate that the ability to improve a behavior performance is limited by the type of the adaptations to the environment. The behavior encoded in the employed BT proved to be especially sensitive to the flight speed of the DelFly. For the other adapted environments DIPS improved the policy performance, however, required an increased number of episodes. Additionally, the learning process appears not to be converged yet at episode 20. Possibly DIPS could have resulted in an increased behavior performance if different settings for ϵ and η were employed or if the number of episodes was increased.

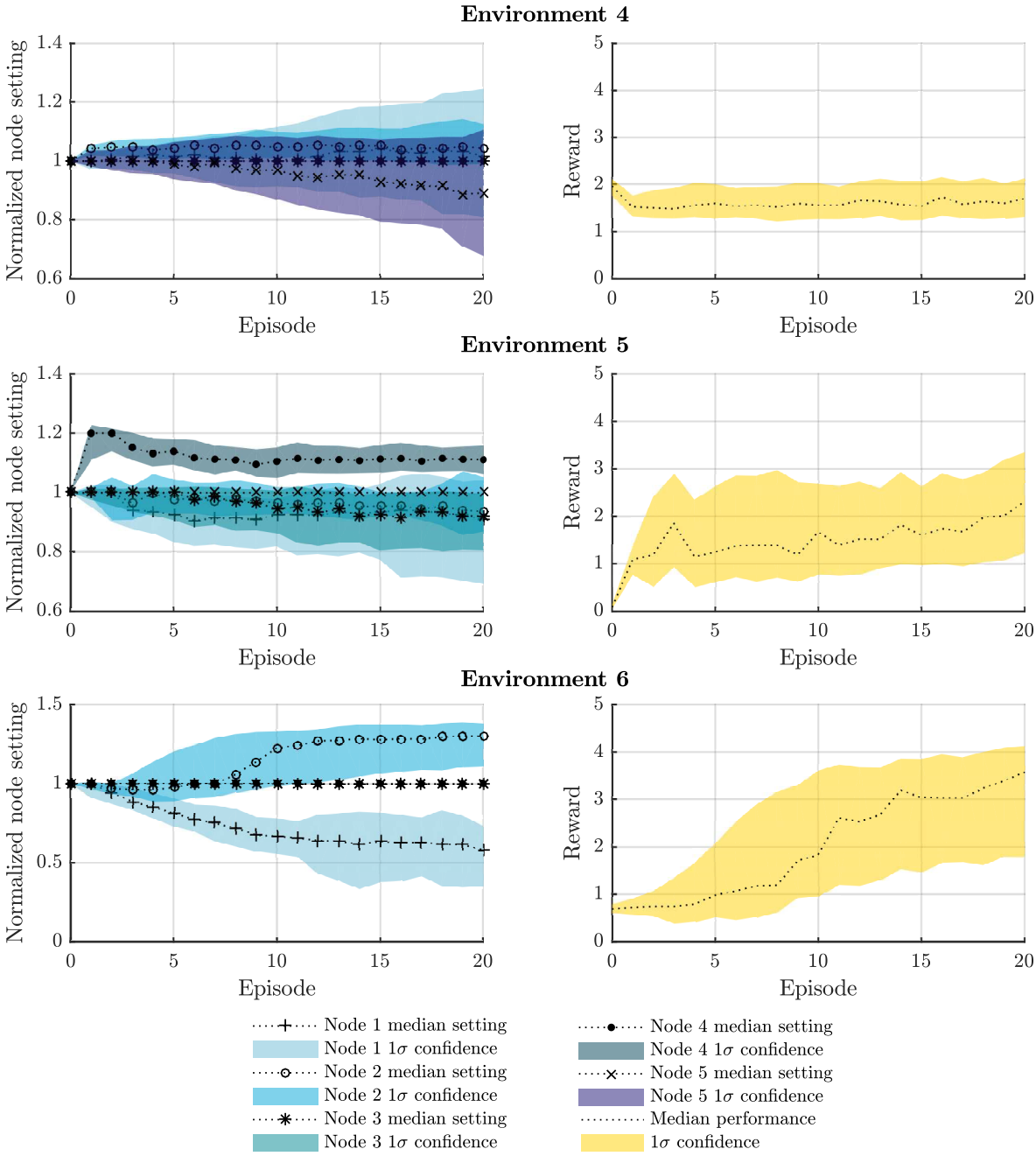


Figure 7-2: Learning process of Directed Increment Policy Search using the monotonic update strategy with a threshold of $T_{min} = 0.4$ for the three adjusted environments. The standard deviation and median parameter setting and performance have been calculated over 60 independent runs. The parameter settings are normalized with respect to the initial settings at episode 0; the reward is obtained by evaluating the policy at the corresponding episode over 100 initializations.

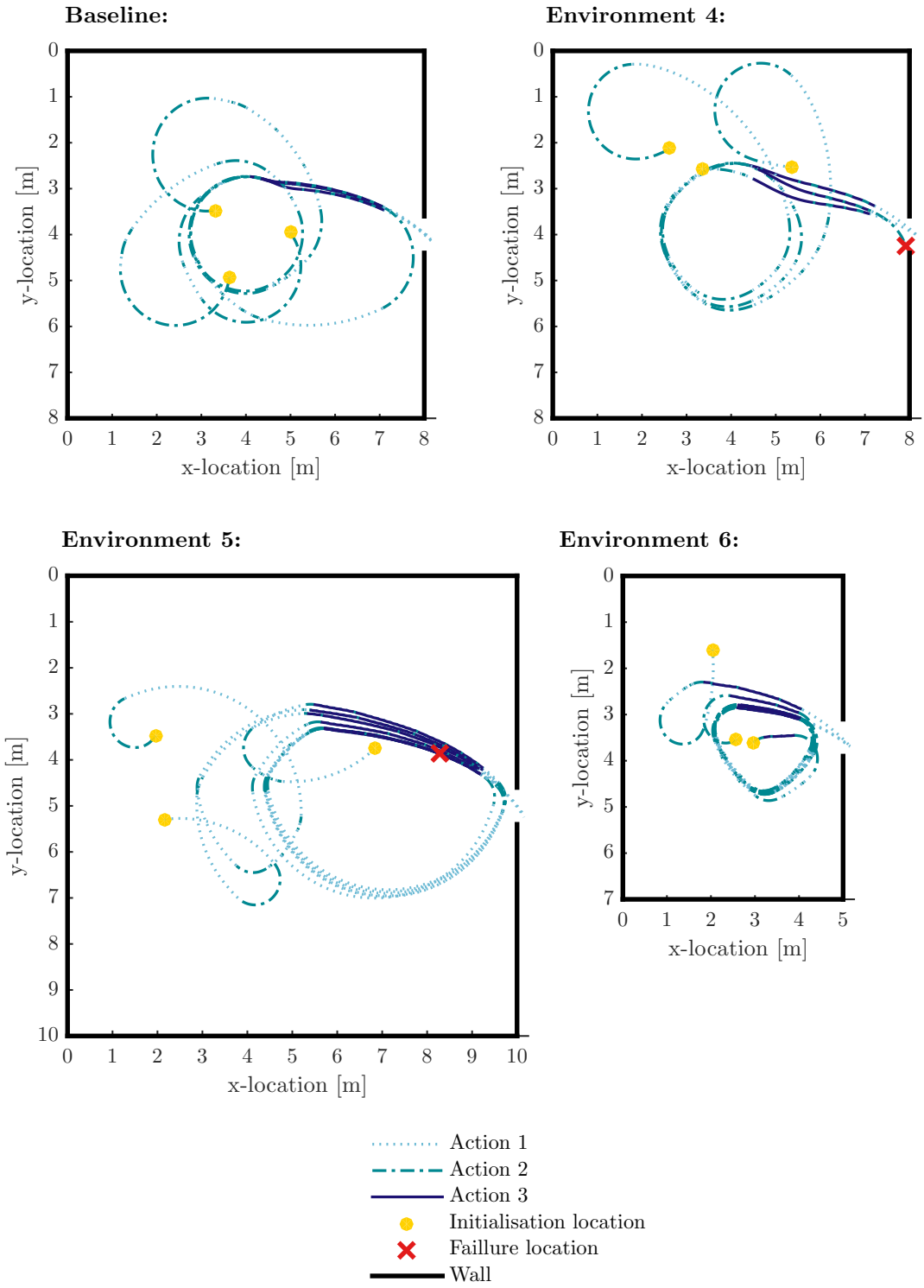


Figure 7-3: Three example paths of DelFly controlled by the policies found by DIPS for the baseline and Environments 4–6.

Part IV

Wrap-Up

Chapter 8

Discussion

The objective of this research was to reduce the influence reality gap of robotic behavior policies by applying RL techniques to update the behavior policy parameters to maintain performance in a similar but adapted simulation environment. In this chapter we discuss the proposed DIPS algorithm and interpret the most important results which were obtained throughout this research: 1) the coupling of the parameters; 2) dependency on the direction array; 3) quantifying the impact of the reality gap; 4) classification and efficiency of DIPS; and 5) the general applicability of DIPS. These discussions form the basis of the recommendations listed in Chapter 10.

Coupling of Behavior Representation Parameters
Chapter 3 discussed different methods to encode robotic behavior. A behavior representation method consists of a structure and a set of parameters. This research focused on the behavior representation parameters. Provided a representation structure and an operating environment, combinations of the behavior representation parameters results in the expression of behaviors with varying performance. Given a limited range of possible parameter values around the initial settings, we expect a single-peak performance landscape originating from the coupling between these parameters.

Parts I and III proposed the DIPS algorithm as policy search method for the DelFly window fly-through task and verified the hypothesized performance landscape and node couplings. Additionally, Chapter 4 demonstrated that these couplings also exist in a discrete parameter search space. These couplings result in a phased learning behavior with a fixed performance; a behavior with increased performance may only be realized when the coordination of sub-behavior improves.

Both experiments demonstrated parameter coupling despite the different domains and applied learning methods. We believe this is a strong indication that these couplings also exist for other tasks and behavior representation methods. Essentially behavior is the result of couplings; if the task changes, the effectiveness of the parameter combinations changes. Similarly, a behavior consists of sub-behaviors stored in a behavior representation. Irrespective of the representation structure, sub-behaviors are required to execute the correct action

(set) in the correct state such task progress is realized. These properties are independent of the task nature and the behavior representation. We therefore hypothesize that the node couplings and subsequent performance landscape, provided initial parameter ranges, also exists for other tasks and behavior representations methods. As a result, this property may also be leveraged to improve and optimize robotic behavior in a wider field of applications.

Dependency on Direction Array

DIPS exploits the coupling of the BT parameters to reduce the number of evaluations. The direction array governs this process by limiting the search space and directing exploration. It was shown that the benefit of these properties, however, is related to the quality of the specified direction array. Here we discuss the the expected influence of learning with incorrect direction array elements and the applicability of applying sampling to formulate the direction array elements.

Limiting the search space through the zeros in the direction array was one mechanism of DIPS to reduce the number of required evaluations. Zero-elements in the direction array reduce the parameter search space by neglecting the respective parameters during the exploration and update process. As such, two cases can be considered in this discussion: 1) over-constraining the search space and 2) under-constraining the search space.

When over-constraining the search space the DIPS algorithm is unable to find high-performing policies because the direction array contains zero elements for parameters with a large influence on the behavior performance. In this case DIPS may still improve upon the performance of the initial policy, however, is likely unable to achieve a similar performance as for the original environment.

Under-constraining the search space implies that the direction array contains non-zero elements for parameters with a small or negligible influence on the behavior performance. In Chapter 6, it was shown that a naive direction array reduced the performance of the final policy and increased the number of required evaluations. The experienced performance reduction was the result of parameter coupling. The magnitude of this effect is expected to be dependent on the reward function. The number of required evaluations scales linearly with the number of non-zero elements in D and therefore increases when the direction array is under-constrained. To reduce the effect of under-constraining the learning process a condition may be implemented which sets the corresponding element in D to zero if the respective parameter step-size is reduced by η to a specified fraction of the parameter value. A possible consequence, however, is that an element can incorrectly be set to zero due to coupling effects.

The sign of the direction array elements direct the exploration process. Hence, the exploration policies result in biased sampling of the parameter directions. Assuming a Gaussian-shaped single peak performance landscape and a low performance initial policy, the bias has two effects on the learning process. First, biased sampling increases the probability of generating exploration policies with increased performance because the magnitude of the performance increase is greater than the magnitude of the performance decrease. Second the relative magnitude of the policy parameter update is increased if multiple exploration policies experience a performance increase as the magnitude of the performance increase is greater. As a result, biased exploration increases the learning rate of the process if the direction array elements have the correct sign. Conversely, the learning rate is reduced if the sign of the exploration policies is incorrect. For incorrect signs this effect is magnified by η which reduces the step-size. Despite the reduced learning rate it was shown that polytonic DIPS can recover from

incorrect sign specification (for node 5 in Environment 1).

From the previous two paragraphs and the results found in this report it may be concluded that the direction array has a large effect on the performance of the final policy, the consistency of the learning process and the number of required episodes. Chapter 6 demonstrated that the dependency of the D array diminished by sampling and re-sampling the elements in D from the environment. Sampling, however, increased the number of required episodes and reduced the final policy performance and consistency of the learning process. For complex behaviors or behavior representations which are difficult to interpret it may not be possible to formulate a direction array by hand; requiring sampling to formulate D. This proposed simple sampling algorithm, however, mainly functioned as a proof of concept; we believe that for real-world applications the proposed algorithm requires to be improved improvement to reduce the number of required evaluations.

Quantifying the influence Reality Gap

In Part I the success rate was initially used to quantify the impact of the reality gap as this provides a direct indication of the behavior policy reliability. Later the average reward over t trials, referred to as the performance, was employed such as to provide the learning process with additional information on the behavior quality. In addition to the success rate, we would like to discuss two other methods of quantifying the influence of the reality gap: 1) the relative required parameter update; and 2) the spread in the policy performance. Both methods are briefly discussed below.

The relative required parameter update indicates by how much the behavior representation parameters are required to be updated. In this research it was shown that the success rate of the initial policy for Environment 2 was (slightly) greater than for Environment 3. Environment 2, however, required a larger relative parameter update as compared to Environment 3 in order to increase performance. We believe a larger relative parameter update corresponds to a more difficult optimization process because we would expect more required episodes which in term results in an extended learning process. An extended learning process is more susceptible to noise. Following this method Environment 2 is the environment with the largest reality gap.

Spread in performance provides a direct measurement of the adaptation process robustness and therefore reliability. An increased spread of the policy performances can originate from a greater required parameter update step but also from the influence of the coupling effects. Indirectly, the spread is an indication of how suitable the BT structure is for the adapted environment. A small spread here indicates that the encoded behavior can effectively be updated by optimization of the BT parameters only. As such, this method is an extension to the first method. Following this method Environment 2 is again the environment with the largest reality gap.

Both proposed methods are unable to quantify the impact of the reality gap before the optimization process, however, they are can be a measure for the difficulty of adapting the initial policy to the respective environmental adaptations. These measures can therefore be a useful tool for assessing and improving the mechanics of any algorithm which updates a behavior on-line.

Classification and Efficiency of DIPS

Policy Search (PS) methods are a category of RL algorithms which directly search for optimal policy parameters. PS algorithms sample experience from the environment to determine the policy parameter update step. DIPS can be categorized as a model-free episodic policy search algorithm which determines the step-size of the parameter updates based on the experienced performance increase. This corresponds to updating the parameters with the steepest gradient first, assuming the increment step-sizes approximately have an equal relative magnitude. For this reason we would classify DIPS as a finite-difference policy gradient method, despite that the algorithm does not directly estimate and exploit the policy gradients.

Let us again consider the three requirements for an effective algorithm to reduce the reality gap, the algorithm is required to: 1) improve policy performance; 2) be robust to adaptations to the environment; and 3) keep the number of evaluations at a minimum. For PS and RL algorithms in general the third requirement forms a large challenge. Other PS methods approach this challenge by learning models of the environment or implementing crafted continuous reward functions. In this research we deliberately averted from these methods as we want to follow the autonomous flight design philosophy of the DelFly, favoring the use of onboard measurements only and algorithms with low computational complexity. These added constraints dismiss model-based PS methods and the recording of full trajectories for crafting reward function; the task of learning a high-level behavior favors episodic policy updates.

Coarsely, model-free episodic policy search algorithms update the policy parameters based on two methods: 1) policy gradients; and 2) expectation maximization. Traditional policy gradient methods are most effective for continuous reward functions or when full trajectories can are recorded such that the gradient can be estimated using pseudo-inverse. For this task and added limitations, however, we believe reliable gradient estimates are difficult to gather due to the discrete reward function. Expectation maximization algorithms learn and adjust higher-level distribution over the policy parameters. These algorithms, however, are expected to suffer from the discrete reward function, coupling between policy parameters and dependency on initialization location of the DelFly. As a result, both methods are expected to require an increased amount of evaluations to either obtain an accurate estimate of the policy performance or to cope with the node couplings. Therefore, we believe DIPS is an efficient PS algorithm for the DelFly window fly-through task.

General Applicability of DIPS

We believe DIPS is an efficient RL algorithm which significantly improved on the challenges set out in the Introduction. The high feedback signal quality, updates based on sampled performance instead of gradients and robustness to hyper-parameter settings makes DIPS an efficient PS algorithm. The required number of evaluations, however, still exceeds the number required for practical implementations –especially considering that multiple independent runs are advised in a real-world environment. We therefore recommend further research in developing the algorithm; these recommendations are listed in Chapter 10.

In addition to reducing the reality gap, we believe the DIPS algorithm can be an improvement over other algorithms to problems which meet the following criteria: 1) the discrepancy between environments are small; 2) an initial policy is available; and 3) the designer has an idea in which direction the parameters have to be updated. The first requirement limits the search space and forms an important basis for the one-peak performance landscape. This also

accounts for the second requirement as this provides an initial parameter range to limit the number of possible solutions. The final requirement increases the learning rate and leverages the one-peak performance landscape.
Combining these requirements, the DIPS algorithm may also be applied to real-world robotics which operate in a gradually changing environment such as lightning conditions or wear and tear. These gradual changes are expected to result in small discrepancies. In these type of problems, the goal has to be properly defined such that rewards can be formulated and used for optimization. As such the MAV can, based on probability, experiment with an exploration behavior policy during operation to check if this results in an increased performance.

Chapter 9

Conclusion

The reality gap experienced by robotic systems is characterized by small discrepancies between the simulation and real-world environment which cause a simulation-learned behavior policy to become ineffective once transferred to a real-world environment. In this research a simple reinforcement learning algorithm named DIPS has been proposed to reduce the reality gap for autonomous MAVs controlled by a behavior tree. This section concludes on the five research questions formulated in the Introduction:

1. In reinforcement learning an agent learns a control policy by interacting with its environment, receiving feedback in the form of a numerical reward after every selected action. Coarsely reinforcement learning algorithms can be segmented in value-based and policy search methods. Value-based methods estimate the long-term expected reward for all states and actions. Policy search methods directly search for an optimal policy.

2. Behavior trees are models to describe a system which can be represented graphically as a directed tree, consisting of a structure and a set of parameters. Behavior trees function as a representation method of robotic behavior by hierarchically combining sub-behaviors, resulting in a scalable and interpretable structure. The behavior expressed by the behavior tree is considered as an emergence from the coupling between the behavior tree structure, parameters and operating environment.

3. Adapting the behavior tree parameters is equivalent to a high-dimensional parameter search problem. Within the reinforcement learning algorithms model-free episodic policy search algorithms combine a promising set of features to adapt the behavior tree parameters. This form of policy search algorithms does not rely on learning complex models of the environment, optimizes the general behavior and facilitates on-line learning.

4. The performance of model-free episodic policy search algorithms can be improved by adding a-priori knowledge to the learning process. This research proposed an algorithm

which leverages the interpretable structure of behavior trees and two forms of a-priori knowledge: 1) an initial policy; and 2) the hypothesized function form of the performance landscape. Provided an initial policy, it was shown that the performance landscape of the behavior tree parameters has a distinct peak amongst a more flat surface. This functional form is the result of the coupling of the behavior tree parameters; the peak represents the parameter range which results in well-coordinated sub-behaviors. Outside of the peak range the parameter settings express a sub-behavior which no longer contributes to the general behavior, resulting in a low-performance general behavior. DIPS leverages this functional form through a direction array. This direction array is specified by the designer through his or her understanding of the behavior encoded in the behavior tree. The direction array reduces the number of evaluations by reducing the search space, directing the exploration process and increasing the quality of the reward signal.

5. Three simulated reality gaps demonstrated the effectiveness and robustness of DIPS to different types of environmental adaptations and hyper-parameter settings. For all three simulated environments, DIPS found a policy with similar performance as the baseline environment performance. In the third environment, DIPS even found a higher performance policy within 135 evaluations. Note that 135 evaluations coincides with 27 actual data points, as the stochastic nature of the window fly-through task requires 5 evaluations to obtain a reasonable estimate.

Despite that DIPS has been applied to a single task only, we believe there are strong indications that DIPS generalizes to different tasks or behavior representation methods. These indications originate from the inherent coupling between sub-behaviors and environment experienced by embodied agents leveraging sensory-motor-coordination.

In this research DIPS was applied to a problem with the objective to reduce the reality gap. DIPS may, however, also be employed to update a behavior during operation because DIPS is not dependent on reward function crafting or external measurements and has an on-line nature.

Chapter 10

Recommendations

This report demonstrated the effectiveness of DIPS to reduce the influence of the reality gap and improve on the challenges listed in the introduction. There are, however, many open questions and improvements which can be made to the algorithm. This chapter formulates the open questions in the form of recommendations. These recommendations are related to researching the generalization properties of the DIPS algorithm, improving the current version of DIPS and finding alternative methods to leverage the parameter couplings.

Generalization Properties of the DIPS Algorithm
The most important recommendations are related to the generalizing properties of DIPS. Below we discuss and recommend three methods to verify the generalizing properties by experimenting with: 1) different behavior policy representations; 2) different tasks; and 3) real-world experiments.

This research only considered a BT to represent the robotic behavior and conducted a single experiment. An important mechanism is the single-peak performance landscape which is the result of the coupled behavior parameters. We believe that this working principle extends to other parametric behavior representation methods such as finite-state machines and small artificial neural networks. For finite-state-machines it is believed that the direction array can be formulated by the designers understanding of the behavior, however, may require an increased number of optimization parameters. For an artificial neural network this is expected to be more difficult –but not impossible. Sampling the direction array elements may be a method to formulate the direction array for artificial neural networks. Both hypotheses are recommended for further research. Here it would be interesting to see how the couplings behave if the number of optimization parameters is increased.

Because general behaviors often consist of sub-behaviors we expect that parameter couplings are also present in behaviors for other tasks than adopted in this research. The DelFly window fly-through task combined three sub-behaviors. We expect that the coupling effect can become even more pronounced if a general behavior consists of an increased amount of sub-behaviors. This hypothesis can be verified by evaluating a more complex task. Increasing

the task complexity, however, results in a more difficult optimization process as number of couplings scales exponentially with the number of parameters.

A real confirmation of the DIPS can, of course, only be proved by testing the algorithm in a real-world experiment. In this research reality gaps were simulated. This implies that we had total control of the environment; this control could be leveraged by formulating a direction array specific for the independent environment variables. For a real-world environment this is different. In a real-world environment many, known and unknown, independent variables influence the performance of a behavior policy.

To test the generalization properties of DIPS we recommend to first experiment with a different task whilst employing a BT to represent the robotic behavior. This facilitates the understanding of the behavior and allows to test the feasibility and performance of DIPS for different and more complex tasks. A more complex task can result in a larger behavior tree; it will be interesting to see if the couplings still hold for an increased tree size and if the search space reduction property of DIPS is still applicable.

Improving the Current DIPS Algorithm

The current version of DIPS is kept as simple as possible to reduce the number of hyper-parameters. We expect, however, that by adding a number of extensions the performance may be increased. Below we list three possible additions.

RL algorithms often employ a learning rate to determine to which extend new information overrides old information. The learning rate may be added to the policy parameter update step. Addition of a learning rate can influence learning process stability and convergence rate. In this research a learning rate of $\alpha = 1$ has been used. If the learning rate is increased however, the algorithm will update the policy parameters more aggressively. In term, this can increase convergence speed but can also reduce stability due to the coupling effects. By letting the learning rate decay over the number of episodes the learning process is expected to become more stable. Following the conclusions from the preliminary research, a learning rate specific for every parameter can possibly combine the advantages of stability and convergence speed.

In addition to a variable learning rate, the hyper-parameters η, ϵ and T_{min} can be made variable to each of the respective parameters. Currently these parameters are determined heuristically through the knowledge of the designer and have been shown to mainly influence the number of required episodes. A large improvement to DIPS can be made if these parameters can be determined in a systematic way. It is, however, unsure if this can be realized as hyper-parameter selection currently considered one of the more difficult problems faced by RL algorithms.

RL methods are known to require many data points to converge. The current implementation of DIPS updates the parameters based on the sampled experience of the most recent episode only. To increase the available information a system may be added which stores the performance increases of the exploration policies over the past episodes, as a form of experience replay (Silver et al., 2016). We expect that this can especially improve convergence rate of the later episodes because by then most coupling effects have diminished and the the polytonic updates are mostly governed by noise.

Research in Leveraging of Coupling Effects

A key contribution of this research is the insight in the performance landscape of the behavior representation parameters. This insight is currently only employed by the direction array. Model-free episodic PS algorithms follow a three-step cycle: 1) policy exploration; 2) policy evaluation; and 3) policy update. Below we discuss how the parameter couplings may improve the algorithm performance by leveraging this knowledge in the exploration and policy update steps. Both recommended approaches rely on the insight that the parameter couplings keep a similar shape if the environmental adaptations are small.

Exploration is currently governed by the direction array. During every episode DIPS naively explores the policies generated by the non-zero elements in D. A more sophisticated form of exploration may be implemented by correlating different parameters based on the couplings found in the original environment. As such, the exploration step may no longer rely on evaluating an equal amount of exploration policies as policy parameters. Possibly this can result in a linear decrease of number of evaluations.

The concept of coupled exploration can be taken a step further by updating a policy parameters without conducting any exploration of the respective parameter. The update step of one parameter can then be related to the performance increase of an exploration policy of a different, but strongly coupled, policy parameter. Again, this can result in a reduction of required number of evaluations, however, can also destabilize the learning process and heavily relies on the assumption that the parameters couplings remain similar-shaped in similar environments.

The previous two approaches are focused on the DIPS algorithm. It is, however, also possible to use the information on the performance landscape and parameter couplings through a different algorithm. Chapter 5 demonstrated that the policy parameter performance landscapes remain similar-shaped when the adaptations to the environment are small. As such, the performance landscapes may be parametrized. A different learning algorithm can then attempt to learn the parameters of the parametrized functional form for the adapted environment, starting with the parameters for the initial environment.

Bibliography

Abbeel, P., Coates, A., Quigley, M., & Ng, A. Y. (2007). An application of reinforcement learning to aerobatic helicopter flight. *Advances in Neural Information Processing Systems*, *19*, 1–8.

Abiyeva, R., Gnsela, I., Akkayaa, N., Aytaca, E., Cagman, A., & Abizadaa, S. (2016). Robot soccer control using behaviour trees and fuzzy logic. *Procedia Computer Science*, *102*, 477-484.

Anderson, M. (2003). Embodied cognition: A field guide. *Artificial Intelligence*, *149*(1), 91–130.

Argall, B., Chernova, S., Veloso, M., & Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, *57*(4), 469 - 483.

Arkin, R. C. (1998). *An behavior-based robotics* (1st ed.). Cambridge, MA, USA: MIT Press.

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, *47*(2-3), 235–256.

Bagnell, J. A., Cavalcanti, F., Cui, L., Galluzzo, T., Hebert, M., Kazemi, M., et al. (2012). An integrated system for autonomous robotics manipulation. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2955-2962.

Barto, A. G., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, *13*(1-2), 41–77.

Barto, R., Sutton, R., & Anderson, C. (1983). Neuron-like elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, *13*, 835–846.

Beer, R., & Gallagher, J. (1992). Evolving dynamic neural networks for adaptive behavior. *Adaptive Behavior*, *1*(1), 91–122.

Bongard, J., & Lipson, H. (2004). Once more unto the breach: Co-evolving a robot and its simulator. *Artificial life*, *9*, 57–62.

Brooks, R. (1986). A robust layered control system for a mobile robot. *Journal on Robotics and Automation*, *2*(1), 14–23.

Brooks, R. (1991). Intelligence without representation. *Artificial Intelligence*, *47*(1), 139–159.

Busoniu, L., Ernst, D., De Schutter, B., & Rabuska, R. (2010). Online least-squares policy iteration for reinforcement learning control. *Proceedings of the 2010 American Control Conference*.

Chernova, S., & Veloso, M. (2008). Teaching multi-robot coordination using demonstration of communication and state sharing. *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems,*.

Chivilikhin, D., & Ulyantsev, V. (2013). Learning finite-state machines: Conserving fitness function evaluations by marking used transitions. *International Conference on Machine Learning and Applications*.

Colledanchise, M., & gren, P. (2016). How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees. *Transactions on Robotics*, *PP*(99), 1–18.

Community, P. (n.d.). *Paparazzi mission planning.* http://wiki.paparazziuav.org/wiki/Mission. (Accessed: 01-03-2017)

Croon, G. de, Connor, L., Nicol, C., & Izzo, D. (2013). Evolutionary robotics approach to odor source localization. *Neurocomputing*, *121*(2), 481–497.

Croon, G. de, Dartel, M. van, & Postma, E. (2005). Evolutionary learning outperforms reinforcement learning on non-markovian tasks. *Learning and Memory WOrkshop of European Conference on Artificial Life*.

Croon, G. de, Groen, M., Wagter, C. de, Remes, B., Ruijsink, R., & Oudheusden, B. van. (2012). Design, aerodynamics and autonomy of the delfly. *Bioinspiration and Biomimetics*, *7*(2).

Croon, G. de, Percin, M., Remes, B., Ruijsink, R., & De Wagter, C. (2016). *The delfly* (1st ed.). Amsterdam, Netherlands: Springer Netherlands.

Dey, R., & Child, C. (2013). Ql-bt: Enhancing behaviour tree design and implementation with q-learning. *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, 1-8.

Dietterich, T. (2000). Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, *13*(1), 227-303.

Dromey, R. G. (2003). From requirements to design: formalizing the key steps. *Proceedings of first International Conference on Software Engineering and Formal Methods*, 2-11.

Dzeroski, S., Raedt, L. de, , & Driessens, K. (2001). Relational reinforcement learning. *Machine Learning*, *43*(1–2), 5–52.

Floreano, D., & Mattiussi, C. (2008). *Bio-inspired artificial intelligence: Theories, methods, and technologies*. Cambridge, Massachusetts: The MIT Press.

Floreano, D., & Mondada, F. (1998). Evolutionary neurocontrollers for autonomous mobile robots. *Neural Networks*, *11*(7-8), 1461–1478.

Floreano, D., & Wood, R. J. (2015). Science, technology and the future of small autonomous drones. *Nature*, *521*(7553), 460–466.

Greensmith, E., Barlett, P., & Baxter, J. (2004). Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, *5*, 1471–1530.

Grollman, D., & Jenkins, O. (2010). Can we learn finite state machine robot controllers from interactive demonstration? *From Motor Learning to Interaction Learning in Robots*, *261*(1), 407–430.

Gullapalli, V., Franklin, J. A., & Benbrahim, H. (1994). Acquiring robot skills via reinforcement learning. *IEEE Control Systems*, *14*(1), 13-24.

Harel, D. (1987). Statecharts: a visual complex systems. *Science of Computer Programming*, *8*, 231–274.

Hart, S., & Grupen, R. (2011). Learning generalizable control programs. *IEEE Transactions on Autonomous Mental Development*, *3*(3), 216–231.

Isla, D. (2015). Handling complexity in the halo2 ai. *Game developers conference*.

Jakobi, N., Husbands, P., & Harvey, I. (1995). Noise and the reality gap: The use of simulation in evolutionary robotics. *Proceedings of the Third European Conference on Advances in Artificial Life*, 704–720.

Janssen, Y., Scheper, K., Kampen, E. van, & Croon, G. de. (2016). Reinforcement learning policy approximation by behavior trees. *Unpublished thesis at the Delft University of Technology*.

Kaelbling, L., Littman, M., & Moore, A. (1996). Reinforcement learning: A survey. *Journal of Artifcial Intelligence Research*, *4*(1-2), 237–285.

Klöckner, A. (2013). Behavior trees for uav mission management. *Informatik angepasst an Mensch, Organisation und Umwelt*, *220*, 57–68.

Klöckner, A. (2015). Behavior tees with stateful tasks. *Advances in Aerospace Guidance, Navigation and Control - Selected Papers of the Third CEAS Specialist Conference on Guidance, Navigation and Control Held in Toulouse*, 509–5019.

Kober, J., Bagnell, J. A., & Peters, J. (2013). Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, *32*(11), 1238-1274.

Kohl, N., & Stone, P. (2004). Policy gradient reinforcement learning for fast quadrupedal locomotion. *Proceedings of the IEEE International Conference on Robotics and Automation*, *3*, 2619–2624.

Konidaris, G., Kuindersma, S., Grupen, R., & Barto, A. (2012). Robot learning from demonstration by constructing skill trees. *International Journal of Robotics Research*, *31*(3), 360-375.

Koos, S., Mouret, J. B., & Doncieux, S. (2013). The transferability approach: Crossing the reality gap in evolutionary robotics. *IEEE Transactions on Evolutionary Computation*, *17*, 122–145.

Kormushev, P., Calinon, S., & Caldwell, D. (2013). Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, *2*(1), 122–148.

Lagoudakis, M., & Parr, R. (2003). Least-squares policy iteration. *Journal of Machine Learning Research*, *4*(1), 107–1149.

Lim, C., Baumgarten, R., & Colton, S. (2010). Evolving behaviour trees for the commercial game defcon. *European Conference on the Applications of Evolutionary Computation*, *PP*(99), 100-110.

Lin, L. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, *8*(3), 293–321.

Mannucci, T., Kampen, E. van, Visser, C. de, & Chu, Q. (2017). Hierarchically structured controllers for safe uav reinforcement learning applications. *AIAA Information Systems-AIAA Infotech Aerospace*.

Martinez, D., G., A., & Torras, C. (2015). Relational reinforcement learning with guided demonstrations. *Artificial Intelligence*, *21*, 216–231.

Marzinotto, A., Colledanchise, M., Smith, C., & gren, P. (2014). Towards a unified behavior trees framework for robot control. *IEEE Robotics and Automation Society*, 5420-5427.

Mataric, M. (1998). Behavior-based robotics as a tool for synthesis of artificial behavior and analysis of natural behavior. *Trends in Cognitive Sciences*, *2*(3), 82–86.

Millington, I., & Funge, J. (2009). *Artificial intelligence for games (2nd edition)*. Boston, MA, USA: Morgan Kaufmann.

Molenkamp, D., Van Kampen, E., Visser, C. de, & Chu, Q. (2017). Intelligent controller selection for aggressive quadrotor manoeuvring. *AIAA Information Systems-AIAA Infotech Aerospace*.

Mucientes, M., Moreno, D., & Barro, S. (2007). Design of a fuzzy controller in mobile robotics using genetic algorithms. *Applied Soft Computing*, *7*(1), 540–546.

Nelson, A. L., Barlow, G. J., & Doitsidis, L. (2009). Fitness functions in evolutionary robotics: A survey and analysis. *Robotics and Autonomous Systems*, *57*(4), 345 - 370.

Nolfi, S. (2002). Power and limits of reactive agents. *Neurocomputing*, *42*, 119–145.

Ogren, P. (2012). Increasing modularity of uav control systems using computer game behavior trees. *AIAA Guidance, Navigation, and Control Conference*(August), 1–8.

Parr, R., & Russell, S. (1998). Reinforcement learning with hierarchies of machines. In *Proceedings of the 1997 conference on advances in neural information processing systems 10* (pp. 1043–1049). Cambridge, MA, USA: MIT Press. Available from `http://dl.acm.org/citation.cfm?id=302528.302894`

Perez, D., Nicolau, M., ONeill, M., & Brabazon, A. (2011). Evolutionary behavior tree approaches for navigating platform games. *IEEE Transactions on computational intelligence and AI in games*, *PP*(99), 123-132.

Pfeifer, R., Lungarella, M., & Iida, F. (2007). Self-organization, embodiment, and biologically inspired robotics. *Science*, *318*(5853), 1088–1093.

Pomerleau, A. (1991). Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, *3*(1), 88–97.

Pontes Pereira, R. de, & Engel, P. M. (2015). A framework for constrained and adaptive behavior-based agents. *CoRR*, *abs/1506.02312*. Available from `http://arxiv.org/abs/1506.02312`

Quinlan, J. (1986). Induction of decision trees. *Artificial Intelligence*, *1*(1), 81–106.

Riedmiller, M., Gabel, T., Hafner, R., & Lange, S. (2009). Reinforcement learning for robot soccer. *Autonomous Robots*, *27*(1), 55–74.

Ross, S., & Bagnell, J. A. (2012). Agnostic system identification for model-based reinforcement learning. *Proceedings of the 29th Interational Conference on Machine Learning*, 1703–1710.

Russel, J., & Subramanian, D. (1995). Provably bounded-optimal agents. *Journal of Artificial Intelligence Research*, *2*.

Rusu, P., & Petriu, M. (2003). Behavior-based neuro-fuzzy controller for mobile robot navigation. *Transaction on Instrumentation and Measurement*, *52*(4), 1335–1341.

Safiotti, A. (1997). Fuzzy logic in autonomous robotics: Behavior coordination. *Proceedings of the Sixth IEEE International Conference on Fuzzy Systems*, *1*(1), 146–161.

Scheper, K. Y. W., & Croon, G. C. H. E. de. (2017). Abstraction, sensory-motor coordination, and the reality gap in evolutionary robotics. *Artificial Life*, *23*(2), 124-141.

Scheper, K. Y. W., Tijmons, S., Visser, C. C. de, & Croon, G. C. H. E. de. (2016). Behavior trees for evolutionary robotics. *Artificial Life*, *22*(1), 23-48.

Schonebaum, G., Junell, J., & Kampen, E. van. (2017). Human demonstrations for fast and safe exploration in reinforcemetn learning. *AiAA Information Systems-AIAA Infotech Aerospace*.

Seraji, H., & Howard, A. (2002). Design of a fuzzy controller in mobile robotics using genetic algorithms. *IEEE Transactions on Robotics and Automation*, *18*(3), 308–22.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Driessche, G. van den, et al. (2016, January). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489.

Smith, J., & Nguyen, T. (2007). Fuzzy decision trees for planning and autonomous control of a coordinated team of uavs. *Signal Processing, Sensor Fusion, and Target Recognition*, *16*(1).

Stanley, K., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, *10*(2), 99-127.

Sutton, R., Precup, D., & Singh, S. (1999). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*(1), 181-211.

Sutton, R. S., & Barto, A. G. (1998). *Introduction to reinforcement learning* (1st ed.). Cambridge, MA, USA: MIT Press.

Taylor, M. E., & Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, *10*, 1633–1685.

Tijmons, S., Croon de, G., Remes, B., De Wagter, C., Ruijsink, E. J., R. annd van Kampen, et al. (2013). Off-board processing of stereo vision images for obstacle avoidance on a flapping wing mav. *Advances in Aerospace Guidance, Navigation and Control*, *16*(1), 463–482.

Vadakkepat, P., Peng, X., Boon Kiat, Q., & Tong Heng, L. (2006). Evolution of fuzzy behaviors for multi-robotic systems. *Robotics and Autonomous Systems*, *55*(1), 146–161.

Valmari, A. (1998). *The state explosion problem.* Berlin, Heidelberg: Springer.

Wawrzyski, P. (2009). Real-time reinforcement learning by sequential actorcritics and experience replay. *Neural Networks*, *22*(10), 1484 - 1497.

Winston, P. H. (1992). *Artificial intelligence (3rd ed.).* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, *87*(10), 1423–1447.

Yung, N., & Ye, C. (1999). Design of a fuzzy controller in mobile robotics using genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, *29*(2), 315–323.