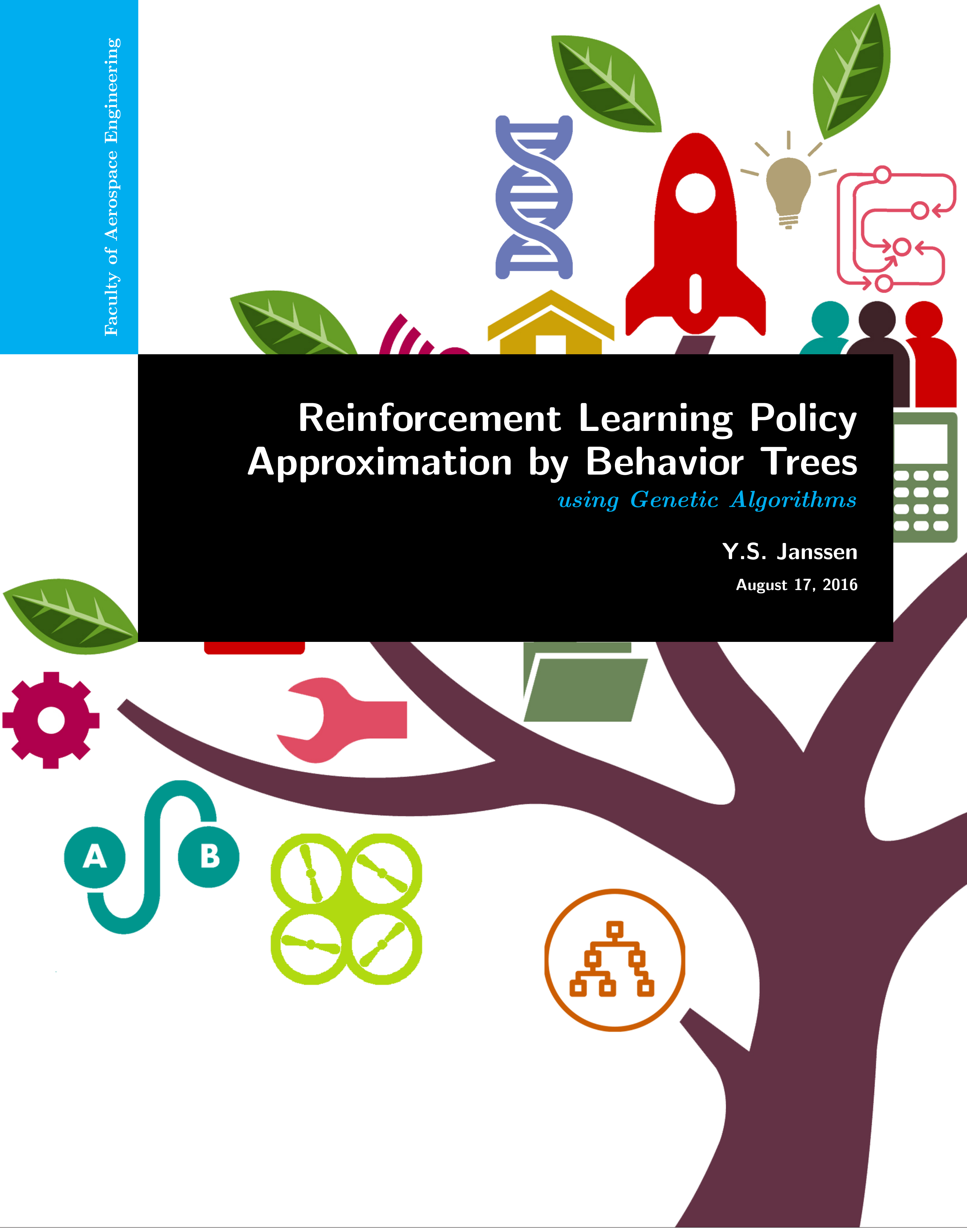
Faculty of Aerospace Engineering



Reinforcement Learning Policy Approximation by Behavior Trees

using Genetic Algorithms

Y.S. Janssen

August 17, 2016

TU Delft Delft University of Technology

Reinforcement Learning Policy Approximation by Behavior Trees

using Genetic Algorithms

Master of Science Thesis

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

Y.S. Janssen

August 17, 2016

Faculty of Aerospace Engineering · Delft University of Technology

TU Delft

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 **"Reinforcement Learning Policy Approximation by Behavior Trees"** by **Y.S. Janssen** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: August 17, 2016

Readers:

dr. Q.P. Chu

ir. K.Y.W. Scheper

dr.ir. E. van Kampen

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

ir. R. Noomen

Acronyms

AE	Aerospace Engineering
AI	Artificial Intelligence
ANN	Artificial Neural Network
BT	Behavior Tree
C&S	Control & Simulation
DAG	Directed Acyclic Graph
DP	Dynamic Programming
DTMC	Discrete Time Markov Chain
DUT	Delft University of Technology
EA	Evolutionary Algorithms
EL	Evolutionary Learning
EO	Evolutionary Optimization
FSM	Finite State Machine
GA	Genetic Algorithms
GNC	Guidance Navigation and Control
GPS	Global Positioning System
GUI	Graphical User Interface
HAM	Hierarchical Abstract Machine
HRL	Hierarchical Reinforcement Learning
MAV	Micro Aerial Vehicle
MDP	Markov Decision Process
MSc	Master of Science
NPC	Non-Playable Character
POMDP	Partially Observable Markov Decision Process
QL-BT	Q-learning Behavior Tree
RL	Reinforcement Learning
TD	Temporal-Difference

UAV	Unmanned Aerial Vehicle
UML	Unified Modelling Language
VV	Verification & Validation

Preface

This thesis includes my research at the Control & Simulation (C&S) Division of Aerospace Engineering (AE) at Delft University of Technology (DUT), and is in partial fulfillment of the Master of Science (MSc) in AE degree.

I would like to express my gratitude to my daily supervisor Kirk Scheper for his feedback throughout my research during meetings or random coffee breaks. My thanks also goes to my fellow students in the graduation room for their advice on the C++language, discussions and group lunches. Finally I would like to thank my family, friends and my girlfriend Claudia for giving me the motivation that kept me going.

Yannick Sebastian Janssen
August 16, 2016
Delft

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Guidance Navigation and Control (GNC) of Unmanned Aerial Vehicles (UAVs) is an active research area (Yamasaki et al., 2007). The design of effective control behavior to guidance tasks for small robotic platforms is a major challenge. So far the main focus has been on making the UAVs perform the tasks *correctly*, designing the controllers, rather than on the high level control of doing the correct *tasks*, designing the control architecture.

However, with increasing capabilities of UAV systems, this high level control becomes more important. Some tasks have to be done in the correct order, while the execution of others depends on certain conditions and a given order of priority, such as collision avoidance and way-point following (Ögren, 2012).

The next challenge arises when UAVs are operated in unknown indoor environments without external navigation possibilities, such as Global Positioning System (GPS). This drives the development of autonomous GNC for a UAV in environments without external localization and without prior knowledge of the environment. A promising method to design GNC for an unknown environment is found in Reinforcement Learning (RL). .

RL algorithms can solve nonlinear, stochastic optimal control problems without using a system model (Busoniu et al., 2010). In a RL problem the agent must learn behavior through trial-and-error interactions with a dynamic environment. "The simplest RL algorithms make use of the common sense idea that if an action is followed by a satisfactory state of affairs, or an improvement in the state of affairs, then the tendency to produce that action is strengthened, i.e., reinforced. This is the principle articulated by Thorndike in his famous Law of Effect (Thorndike, 1911.)." (Barto & Dietterich, 2004). The goal of a RL controller is to find an optimal control policy that maximizes the cumulative long-term reward as a function of the state and possibly of the control action.

Traditionally the learning of a RL controller is stored in lookup tables, *Q-tables*, which store the values of state-action pairs. Based on the agent's policy, which relies on the state-action values, the controller selects a control action. These state-action values cannot always be stored due to the large number of state-action pairs that need to be stored for large state-space environments. In those situations approximations are used to approximate the state-action value Nedić & Bertsekas (2003). Typically, an Artificial Neural Network (ANN) is used to approximate state-action values, the so-called Q(s,a)-values, in a RL problem Watkins & Dayan (1992) He & Jagannathan (2007).

Present day controllers for safety critical GNC applications, like for example UAV guidance, undergo extensive tests to qualify their operation. Although analysis of Q-tables is possible, the difficulty to provide insight in its behavior causes practical issues to Verification & Validation (VV) of the controller, especially for large Q-tables Bishop (1994). Furthermore, Q-tables do not lend themselves well to manual adaptation. This limits the deployment and wider use of RL in guidance applications and creates the desire to look for a framework that is able to give clear insight in RL developed guidance behavior which is also easy to verify.

An Q-table for a simple 3 x 5 grid world problem is shown in Table 1-1. The problem is to navigate from a start state to a goal state with four possible actions, move up, move down, move left and move right. This simple problem results in a the Q-table from which behavior logic is difficult to analyze and from which it is difficult to make adjustments to this logic.

Table 1-1: Conventional way to store a RL policy using a Q-table. Shows a part of the Q-table for a simple 3 x 5 grid world problem with 4 actions.

		Action			
		Up	Down	Left	Right
State	[1,0]	8.73	-99.78	8.99	9.60
	[1,1]	6.47	-93.85	9.15	9.70
	[1,2]	9.30	-89.16	9.21	9.80
	[1,3]	9.34	-25.07	9.27	9.90
	[0,0]	-0.24	9.41	-0.22	-0.23
	[0,1]	-0.20	-0.18	-0.18	8.97
	[0,2]	-0.10	9.70	-0.14	0.47
	[0,3]	4.47	9.76	-0.08	-0.08
	[0,4]	-0.05	-0.05	-0.08	-0.05

A recently developed method to describe control behavior is the Behavior Tree (BT) framework. Initially developed as a method to formally define system design requirements, the BT framework was adapted by the computer gaming industry to control Non-Playable Characters (NPCs) (Lindsay, 2010). BTs contain behavior made up of a hierarchical network of actions . The rooted tree structure of the BT make the encapsulated behavior readily intelligible for users. It provides clear insight to the designer. This BT framework was quickly adopted by game designers who were looking to control for their Artificial Intelligence (AI) in characters. Where in comparison with conventional Finite State Machines (FSMs), BTs feature better scalability with the growing numbers of possible capabilities to be integrated. Developers implemented this BT framework in games such as *Halo2* and *Spore* (Isla, 2005).

BTs applied to GNC tasks have been proven to be manually adjustable after simulations when used on board a real world UAV Scheper et al. (2016). BTs used for guidance give insight in the decision process and shows under which condition an action is carried out. This provides insight into potentially wrong and dangerous decisions of the controller and can be verified by the designer.

A BT orders a global task into smaller subtask. This hierarchical structure of the BT provides a level of abstraction. A designer can use this abstraction to identify subtasks in the RL policy

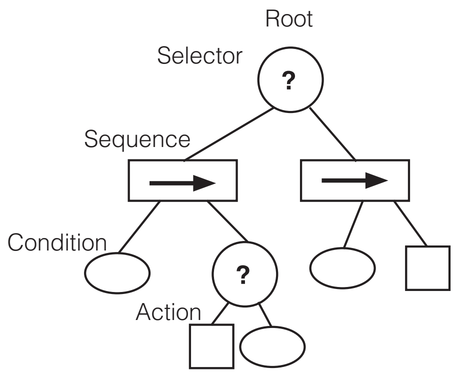


Figure 1-1: Graphical depiction of a Behavior Tree framework. The different node types are indicated.

and structure them in order of priority. Plus, the hierarchy and rules of the BT framework make it possible for designers to verify these subtasks.

Where conventional RL policies are stored in a Q-table, or approximated with ANNs, the hierarchical structure of the BT provides a level of abstraction. This gives the designer the ability to ignore aspects of the current state of the agent that are irrelevant to its current decision, and are therefore easier to interpret. The BT will have the ability to show relevant features that are important to make decisions which will increase the understanding of the developed behavior.

This paper research investigates a method to evolve a BT using Genetic Algorithms (GA) to approximate a learned RL policy. GAs are adaptive heuristic search algorithms based on the evolutionary ideas of natural selection and genetics. A population of BTs is used to produce a BT to evolve a best individual, where the fitness of each individual is measured by a user-defined, problem specific, objective function. The more generic this fitness function is formulated, the more it allows for a easy implementation in a wider range of problems.

However, simulation environments require computationally expensive processing and a disadvantage of using GAs is computation time Scheper et al. (2016). Instead of using the simulation environment to evolve BTs with GAs, a Discrete Time Markov Chain (DTMC) is constructed. The DTMC does not require the simulation of the simulator physics, which speeds up the approximation. Using the DTMC, called 'move through' the DTMC, allows for the analysis of the action identification over time without the simulation environment. By involving probabilities, the outcome of each move through the DTMC will vary from time to time and this variation show the essential variability of each BT in the population.

The method is applied to a RL controller on a UAV for a guidance task in a unknown environment. The RL controller, which uses the Q-learning algorithm, learns a optimum policy after which this policy is approximated with a BT using GA. A successful approximation results the an evolved BT that will aid the user in understanding the developed behavior of the RL algorithm. Selecting the same control actions but in a reusable, adjustable and understandable framework. This BT will also provide the possibility for simple verification of the developed behavior of the controller.

1-1 Research Question

The main goal of the thesis is captured in the research question *How can the policy of a Reinforcement Learning controller be made more intelligible by an automatically generated Behavior Tree?*

This research question is answered by answering the following sub-questions.

- How much learning episodes need to run before a RL policy is converged?
- How to formulate the fitness function for the GA?
- How to evaluate the success of the policy approximation with a BT?

1-2 Thesis Layout

The first part of this thesis will cover the research background, problem identification, methodology, implementation and results of this work in a standalone scientific paper format in Part I.

Part II contains the research background and preliminary results obtained during this thesis. This part presents a more detailed background for this work. For readers unfamiliar with the report structure of a C&S thesis, I advice to start reading with Part II before reading the scientific paper.

Part III contains additional results not covered in the scientific paper.

Part I

Paper

Reinforcement Learning Policy Approximation by Behavior Trees using Genetic Algorithms

Y.S Janssen, K.Y.W. Scheper, E. van Kampen, G.C.H.E. de Croon

***Abstract*—Traditionally a Reinforcement Learning (RL) policy is stored in a lookup table. From such a table it is difficult to observe the behavioral logic or manually adjust this logic post-learning is difficult. This paper shows how behavioral logic of a RL controller is presented in an insightful manner and can be adjusted using the Behavior Tree (BT) framework. It shows a method to approximate an RL policy using a Genetic Algorithms (GA) for BTs for a guidance task carried out by an UAV navigating in an unknown environment. The method shows how Discrete Time Markov Chains (DTMCs) can be used to increase optimization speed. The execution of the RL controller that interacts with the environment is mapped to a DTMC, which results in a representation of the one-step transition matrix. The GA evaluates BTs by transitioning through this DTMC instead of the simulation environment. Without the need to simulate a computationally expensive environment, the optimization is performed faster. The method is demonstrated on a UAV simulation using a Q-learning algorithm to learn a guidance task. The guidance task consists of an avoidance behavior and goal-seek behavior. The evolved BT, containing 6 nodes, successfully identifies 66% of the correct actions of the RL policy for all states in the Q-table. When this BT is run in the simulation environment it results in a success rate of 93%. After adaptation by the experimenter the success rate for all states in the Q-table increases to 86% and in simulation to 96%. For the verification of the avoidance behavior with the evolved BT only 3 nodes need to be verified, compared to 1448 state-action pairs of the Q-table.**

***Index Terms*—Behavior Tree, Reinforcement Learning, Genetic Programming, Q-learning**

I. Introduction

Guidance Navigation and Control (GNC) of Unmanned Aerial Vehicles (UAVs) is an active research area [1]. Designing effective control behaviors to guide tasks for small robotic platforms is a major challenge. So far the main focus has been on making the UAVs perform the tasks *correctly*, designing the controllers, rather than on the high level control of doing the correct *tasks*, designing the control architecture.

With increasing capabilities of UAV systems, this high level control becomes more important. Some tasks have to be done in the correct order, while the execution of others depends on certain conditions and a given order of priority, such as collision avoidance and way-point following [2].

The next challenge arises when UAVs are operated in unknown indoor environments without external navigation possibilities, such as Global Positioning System (GPS). This drives the development of autonomous GNC for a UAV in environments without external localization and without prior knowledge of the environment. A promising method to design GNC for an unknown environment is found in Reinforcement Learning (RL).

A RL controller learns to control a task by interacting with its environment and its immediate performance is measured by a reward [3]. The goal is to find an optimal control policy that maximizes the cumulative long-term reward as a function of the state and possibly of the control action.

Traditionally the learning of a RL controller is stored in lookup tables, *Q-tables*, which store state-action values. Based on the agent's policy, which relies on the state-action values, the controller selects a control action. These state-action values cannot always be stored due to the large number of state-action pairs that need to be stored for large state-space environments. In those situations approximations are used to approximate the state-action value [4]. Typically, an Artificial Neural Network (ANN) is used in RL to approximate state-action values, the so-called Q(s,a)-values [5] [6].

Present day controllers for safety critical GNC applications, like UAV guidance, undergo extensive tests to qualify their operation. Although analysis of Q-tables is possible, the difficulty to provide insight in its behavior causes practical issues to Verification & Validation (V&V) of the controller, especially for large Q-tables [7]. Furthermore, Q-tables do not lend themselves well to manual adaptation. This limits the deployment and wider use of RL in guidance applications and creates the desire to look for a framework that is able to give clear insight in RL developed guidance behavior which is also easy to verify.

A recently developed method to describe GNC tasks is the Behavior Tree (BT) framework [2]. Initially developed as a method to formally define system design requirements, the BT framework was quickly adapted by the computer gaming industry to control Non-Playable Characters (NPCs) [8, 9]. BTs contain behavior made up of a hierarchical network of nodes. The hierarchical rooted tree structure of the BT makes the encapsulated behavior readily intelligible for users. The framework provides readability, maintainability, scalability and re-usability. It presents which control action is executed under which condition.

BTs applied to GNC tasks have been proven to be manually adjustable after simulations when used on board a real world UAV [10]. BTs used for guidance give insight in the decision process and shows under which condition an action is carried out. This provides insight into potentially wrong and dangerous decisions of the controller and can be verified by the designer.

A BT orders a global task into smaller subtask. This

All authors are with the Faculty of Aerospace Engineering, Delft University of Technology, Netherlands. E-mail: ysjanssen@gmail.com

hierarchical structure of the BT provides a level of abstraction. A designer can use this abstraction to identify subtasks in the RL policy and structure them in order of priority. Plus, the hierarchy and rules of the BT framework make it possible for designers to verify these subtasks.

This paper proposes a method to evolve a BT using Genetic Algorithm (GA) to approximate a learned RL policy. GAs are adaptive heuristic search algorithms based on the evolutionary ideas of natural selection and genetics. A population of BTs is used to produce a BT to evolve a best individual, where the fitness of each individual is measured by a user-defined, problem specific, objective function. The more generic this fitness function is formulated, the more it allows for a easy implementation in a wider range of problems.

Simulation environments require computationally expensive processing and a disadvantage of using GAs is computation time [10]. Instead of using the simulation environment to evolve BTs with GAs, a Discrete Time Markov Chain (DTMC) is constructed. The DTMC does not require the simulation of the simulator physics, which speeds up the approximation. Using the DTMC, called 'move through' the DTMC, allows for the analysis of the action identification over time without the simulation environment. By involving probabilities, the outcome of each move through the DTMC will vary from time to time and this variation show the essential variability of each BT in the population.

This research is the first investigation into policy approximation with the BT framework. This paper aims to express a developed RL policy, for an UAV that navigates in an unknown environment using a Q-learning algorithm, in a comprehensive manner using the BT framework. If the policy of a developed RL controller can be approximated using a BT, it will aid the user to understand this developed behavior and allow easy adjustment.

First the problem background is discussed in detail. Then the BT framework is explained. Subsequently the combination of the GA with BT is explained. Followed by an explanation of the contribution of using the DTMC. Later the success of the method is showed using RL navigation task for a UAV trash collector simulation.

II. Problem Statement Background

RL is originated from animal behavior research and their interactions with the environment. If an action is followed by a satisfactory outcome, the tendency to repeat that action is strengthened or *reinforced* [11]. Differing from the traditional supervised learning, there is no desired behavior or training examples employed with a RL schemes. Learning is done from *experience*, meaning from a sequence of states, actions and rewards generated by the agent interacting with its environment. A graphical representation of the learning cycle is presented in Figure 1.

A. Markov Decision Processes

RL is based on the formalism of a Markov Decision Process (MDP). This discrete-time and countable state and action formalism provides a simple framework in which to study the basics of RL.

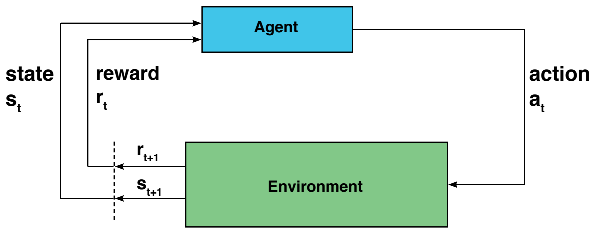


Fig. 1. General reinforcement learning structure. The agent outputs an action a_t to the environment. The environment responds by presenting the agent a new state s_{t+1} and a reward r_{t+1}.

In this framework, a learning *agent* interacts with an *environment* at a discrete timescale, $t = 0, 1, 2, ..., T_m$. On each time step, t, the agent perceives a state of the environment s_t, where $s_t \in S$. In response to each action, a_t, the environment produces a numerical reward one time step later, r_{t+1}, and a next state, s_{t+1}.

Assumed is that S and A are finite and that the environment is completely characterized by on-step state transition probabilities $P(s, a|s')$, and one-step expected reward, r, for all $s, s' \in S$ and $a \in A$.

The formalism assumes the *Markov property*. The Markov property assumes that if the present state, s, is known, then all additional knowledge of the events in the past are irrelevant to select the optimal next action. The current state with transition probability $P(s, a|s')$ unambiguously describes the environment. This is also known as the memoryless property of a stochastic process.

B. Q-learning

One of the most important breakthrough in RL was the development of a control algorithm known as *Q-learning* [3]. Q-learning considers the learning environment as in an MDP and performs value iteration to find the optimal policy. It maintains a value of expected total current and future reward, denoted by Q, for each pair (state, action) in a *table*. For each action in a state, a reward will be given and the Q-value is updated by the following rule defined by Equation (1).

$$\begin{aligned} Q_{t+1}(s,a) = (1-\alpha_t)Q_t(s,a) + \\ \alpha\{r_t + \gamma \underset{a' \in A_{s'}}{max} Q_t(s',a')\} \end{aligned} \tag{1}$$

where α is the learning-rate parameter, γ is the discounted reward factor and r is the reward. All other state-action pairs remain unchanged during the update in Equation (1). In this case, the learned state-action function, **Q**, directly approximates Q^*, the optimal action value function, independent of the policy being followed.

α determines the extent of override of newly acquired knowledge. An α of 0 will not result in any update of the state-action value, while a factor of 1 will result in an update with only using the most recent information.

The discount factor, γ, determines the importance of future rewards. A γ of 0 will result in an agent that only considers short term reward, while a γ of 1 will focus more on long term reward.

When the Q-learning algorithm is used in the MDP formalism and all pairs continue to be updated, the solution to Equation (1) has been shown to converge with probability 1 to Q^* [5].

The agent action-selection procedure, a *policy*, is specified by a mapping from states to probabilities of taking each action: $\pi : S \times A \rightarrow [0, 1]$. Thus, a policy is the mapping from perceived states of the environment to actions to be taken when in those states. An agent that follows a *greedy* policy selects the action with the highest state-action value for every state.

C. Policy storing using Q-table

Traditionally Q-learning keeps a table of all the state-action values. Table I shows an example of the conventional structure, *a Q-table*, in which Q-learning stores its approximations of the state-action pairs. This Q-table does not provide insight in the solution strategy for designers. The designer needs to interpret all the Q-values in the table and link state transitions before a behavior becomes clear, if even possible.

TABLE I
CONVENTIONAL WAY TO STORE A RL POLICY WITH A Q-TABLE. SHOWS A PART OF THE POLICY FOR A 3 X 5 GRID WORLD PROBLEM WITH 4 ACTIONS

		Action			
		Up	Down	Left	Right
State	[1,0]	8.73	-99.78	8.99	9.60
	[1,1]	6.47	-93.85	9.15	9.70
	[1,2]	9.30	-89.16	9.21	9.80
	[1,3]	9.34	-25.07	9.27	9.90
	[0,0]	-0.24	9.41	-0.22	-0.23
	[0,1]	-0.20	-0.18	-0.18	8.97
	[0,2]	-0.10	9.70	-0.14	0.47
	[0,3]	4.47	9.76	-0.08	-0.08
	[0,4]	-0.05	-0.05	-0.08	-0.05

In many cases of practical interest, there are far more states than could possible be entries in a table. In highly-dimensional discrete environments, an exponential explosion of the number of state-action values occurs, the so called *curse of dimensionality*. In these cases the value functions must be approximated using function approximation methods, such as ANNs. However, ANN or other function approximation techniques are not a substitute for understanding the behavior logic. An additionally, the structure of a ANN is highly interconnected which makes them difficult to troubleshoot when they do not work as expected.

So for (large) Q-tables or function approximation methods that store state-action values, a way to present the behavioral logic of a RL controller is lacking. The goal of this paper is to approximate the policy of a optimized RL controller such that it provides the designer with workable tools to identify, adapt and verify (sub)tasks of the learned behavior of a RL controller.

III. BEHAVIOR TREE

Dromey [8] developed a system design method to address the problem of systematically translate large, complex requirement documents into a structured model of the system. This method is called *Behavior Engineering*. The method is centered around a notation to express system behavior called Behavior Trees.

The BT notation got the attention of computer game designers when they searched for a behavior management system for NPCs. They needed a framework to control the modularity, reusability and complexity of the NPC. The BT, although in a different form as was anticipated by Dromey [8], proved to be suited to capture this behavior [12].

Ögren [2] is the first to argue that the modularity, reusability and complexity of UAVs GNC systems might be improve by the BT architecture. Ögren [2] states that this is mainly due to the fact that BTs make the transitions implicit in the tree structure. The implicit transitions substantially increase modularity, which in turn makes design and re-design much simpler. This makes the BT framework an interesting framework to approximate the RL policy for UAV guidance.

A. Syntax and Semantics

BTs are formed by hierarchically organizing behavior subtrees, which consist of nodes. They can also be defined as Directed Acyclic Graphs (DAGs). DAGs consist of a number of nodes that are connected with directed edges. The connections have a direction, to go from node A to node B is not the same as from node B to A. And the structure is acyclic, to move from one node and follow the edges, will ensure that no node is encountered for the second time. The outgoing node is called the parent and the incoming node is the child.

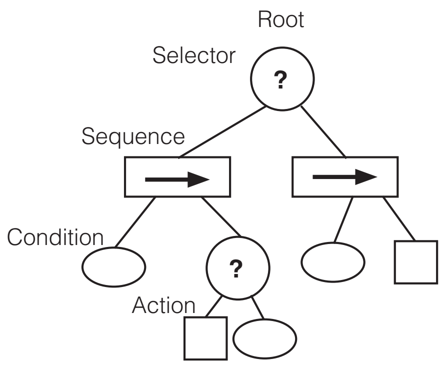


Fig. 2. Graphical depiction of a Behavior Tree with highlighted node types.

The structure of the BT framework provides a hierarchical way of organizing behaviors. It is represented as a rooted tree structure which is evaluated from left to right. The BT is made up of several types of nodes, however all nodes share a core functionality. This core functionality is that a node informs its parent node of its status. This return status is generally either *Success* or *Failure*.

Basic BTs are made up of three kinds of nodes: *Conditions*, *Actions* and *Composites* [9]. Nodes that have no children are called leaf nodes. The leaf nodes can be either Conditions or Actions whilst the branches of the BT consist of Composite nodes. Conditions test a property of the environment. A Condition node returns Success if the condition is met and

Failure otherwise. The agent acts on its environment through Action nodes.

Although more types of Composite nodes can be used [9], this research only considers Sequences and Selectors. The root node of a BT is typically a Selector node.

Selector nodes will return Success to their parent if one of its children return success and will not process any further children. If a Selector node receives Failure from its first child, it will try its second child until it evaluated all its child nodes. The Selector node fails if all child nodes return Failure. Opposite, a Sequence will only return success if all children succeed. If one of its children fails, a Sequence will not process any further children and returns Failure to its parent.

Leaf nodes are problem specific and are developed individually, but can be reused in the tree where required. Composite nodes are versatile and are not problem specific. The shared commonality make it possible to develop a part in the BT without knowledge of other parts. This results in a modular and reusable framework. A sample BT with highlights the different nodes can be seen in Figure 2.

B. *Execution of a BT*

The Composite nodes determine how the BT is executed. Simple combinations of Composite nodes are used to develop complex behavior.

Every time a control action is needed, the tree is executed, referred to as *ticked*. A tick starts from the root node and evaluates down from left to right. An execution is complete when the root node of the tree returns either Success, or when all of its branches are evaluated and return Failure.

The behavior nodes are accompanied by a *Blackboard* which is developed to share information with the BT. During the evaluation of the tree, the tree sets a temporary action to be executed onto the Blackboard until the root node is finished evaluating its children. This allows for actions to be overwritten until the root node returns Success or Failure. The last action that is set on the Blackboard is the action that is available to the agent.

C. *Use BT to Approximate a RL Policy*

The BT framework provides several advantages when used to approximate a developed RL policy. The framework provides readability, maintainability, scalability and re-usability. When the BT framework is used it is clear which control action is selected for which state.

An important advantage is that a BT orders a global task into smaller subtask. This hierarchical structure of the BT provides a level of abstraction compared to the representation all state-action values in a table. A designer can use this abstraction to identify subtasks in the RL policy and structure them in order of priority. This provides the designer to make easy adjustments to the policy.

A second important advantage is that the hierarchical structure also allows to group state-action pairs. This results in a BT that can identify the same action with a few nodes compared to the RL policy that all the states in a large table. This BT will have the ability to show the relevant features the agent decides on and allows the designer to ignore the aspects in the Q-table that are irrelevant for its optimal policy. If a BT can be constructed with only a few nodes that is able to identify the same actions the RL policy identifies based on a Q-table, verification of the task becomes simpler. The tree only requires checking the Condition nodes compared to all state-action pairs in the Q-table.

IV. Genetic Algorithms for Behavior Trees

To evaluate many different BT design on their success of approximating the RL policy, GAs are used. The Genetic Algorithm is a probabilistic search algorithm that iteratively transforms a set (called a population) of mathematical objects into a new population of offspring objects [13]. It mimics nature's mechanisms of evolution like *selection* and *mutation*.

The GA for BTs is adapted from Scheper et al. [10], customized for this research. Scheper et al. demonstrated successfully how to develop robotic control for a single task that uses a population of BTs.

BTs are considered as members of a species, where the fitness of each individual BT is measured according to a specific fitness function. For each generation, the population consists of a set of BTs that are under evaluation.

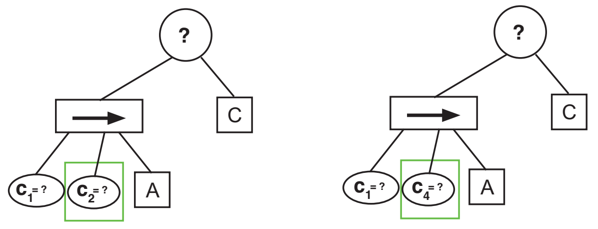
Successful BTs, *parents*, are selected to share their genes, which are, represented by the nodes of a BT. The GAs recombines information of the successful BTs by means of evolution methods, such as *crossover* and *mutation*, to produce new BTs, *children*.

This introduces a variation of BTs which results in exploration of the search space that tends towards the best policy [14].

Genetic Operators

Initialization An initial population of M individuals is generated by means of a grow method. This results in an initial population of BTs with a diversity of genetic material.

Selection Tournament Selection is used. To select a parent tree, first a subgroup of the population is selected, sorted in order of their fitness. Then the best individual from that subgroup is selected.



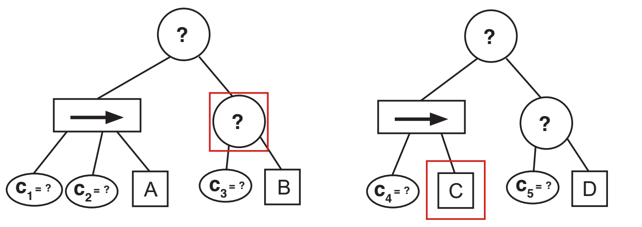
(a) Behavior Tree *before* mutation. (b) Behavior Tree *after* mutation.

Fig. 3. Example Mutation. Node indicated with green box is selected for micro mutation.

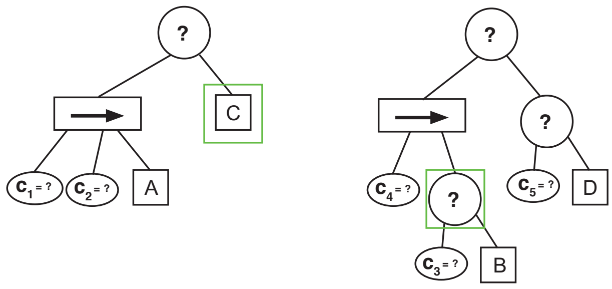
Mutation Two mutation methods are implemented, micro- and macro-mutation. Micro-mutation only affects leaf nodes shown in Figure 3. It reinitializes the node with new operating parameters. Macro-mutation, also named Headless Chicken

Crossover, replaces a selected node by a randomly generated BT, taken into account the maximum tree depth. The probability of mutation is given by the mutation rate P_m. Once mutation is applied, the probability that macro-mutation is applied is given by the Headless Chicken Crossover rate, P_{hcc}.

Crossover A part of the new population is formed by crossover. Crossover recombines a randomly selected node from two parents to produce *two* children. The percentage of the population formed by crossover is defined by the Crossover Rate P_c. The node is selected at random, independent of type or location in the BT. Figure 4 shows the procedure, with Figure 4a presenting the parent BTs and Figure 4b presenting the resulting children BTs.



(a) Parent trees with selected nodes for crossover highlighted in red



(b) Children trees with selected nodes highlighted in green

Fig. 4. Example Crossover.

Stopping rule The evolutionary process repeats itself until the stopping criteria are met. For a GAs, it is typical to use a maximum number of generations. Placing a limit helps to avoid unnecessary long computational time due to the large number of BTs that is considered per generation.

Fitness Function

The fitness function rates the performance of the individuals in the population. This is the objective function that the GA optimizes for. This function is important for the results that can be expected from the optimization.

BTs are considered as members of a population, where the fitness of each individual BT in the population is measured according to Algorithm 1. The main performance metric to score the individuals in the generation of BTs is the number of times the tree identifies the same action as the RL policy. The BT is set as *acting policy* to step through the DTMC, so its fitness will be a representation on how successful the BT identifies the right action compared to the RL policy on its own run through the DTMC.

Algorithm 1: Fitness function used for Genetic Algorithm. For every step through the DTMC for the currently evaluated BT the fitness is increased by 1 if it identifies the same action as the RL policy identifies, otherwise its fitness remains unchanged.

```
if $action_{bt}$ == $action_{Q_{learning}}$ then
    Fitness + 1
else
    Fitness + 0
end if
```

When a tree successfully identifies more than 90% of the actions, an additional fitness function is added to evaluate the BTs. This fitness is related to the size of the BT. A BT with less nodes can be verified more quickly, this results in a tree which scores better when the number of nodes is less. A tree is given maximum fitness score if the number of nodes in the tree is 20 or less. This leads to an optimization that reduces the size of the tree, but stops reducing the size at 20 nodes.

V. Map environment in a Discrete Time Markov Chain

A disadvantage of applying GAs is the long computation time [15]. To speed up the optimization time, the environment is mapped into a DTMC. Compared to rendering a simulation environment the computational requirements of a Markov model is modest.

The DTMC captures the transition probability of the agent in the environment. It allows for an optimization to be performed *without* the simulation environment.

For every action a an $|S| \times |S|$ matrix $P(s, a|s')$ is constructed, as shown by in Equation (2). Each element in the matrix represents the probability of the transition from a particular state (represented by row *i* of the matrix) to the next state (representing the column *j* of the matrix).

$$P(s,a|s') = \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1j} \\ p_{21} & p_{22} & \cdots & p_{2j} \\ \vdots & \vdots & \ddots & \vdots \\ p_{i1} & p_{i2} & \cdots & p_{ij} \end{pmatrix} \qquad (2)$$

In the transition probability matrix all probabilities are non-negative, with the sum of each row equal to 1.

An example of a DTMC with 8 states and the corresponding transition probabilities are shown in Figure 5. The execution of a task is represented as a sequence of probabilistic events.

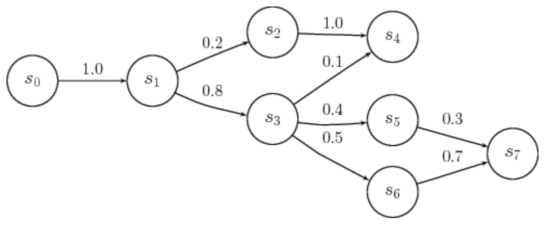


Fig. 5. Discrete Time Markov Chain with 8 states and corresponding transition probabilities.

The Markov chain provides the probability transitions from one state to any other state without the simulation environment. At each time step, the BT identifies an action given the state and presents it to the transition matrix, which determines the state to which the system moves by generating a random number. This computation is repeated for a fixed number of steps through the DTMC.

The action selection of the tree is compared to the action selection of the RL policy in the Markov chain. A fitness function is constructed using this comparison which enables to analyze the success of BTs compared the the RL action selection over time by moving through the DTMC in stead of the simulation environment.

In addition to speed up computation time, the states that are encountered in the Markov chain are states with high probability to transition to. When the BT steps through the DTMC, these states are encountered more often are more likely to be approximated correctly. This way the Markov chain also acts as a guide in the Q-table for the optimization process.

The transition matrix is constructed using the frequencies of state transitions by repetitively stepping through the environment during the development of the RL controller. This captures the stochasticity, the physics of the environment and how the agent transitions through the environment. The DTMC abstracts away the physics of the simulator that requires computationally expensive processing and reduces computation time for the optimization.

VI. Trash collecting UAV Test-Case

First an overview of the method is presented before the task is specified. The method is tested on a simulation of a UAV. The guidance task is learned using the Q-learning algorithm to develop a policy that maximizes its reward per run in the simulation. This developed RL policy is then approximated using GAs with the BT framework.

A. Method

First 1) a RL controller is developed. This controller will learn a control policy for the guidance task based on a Q-learning algorithm. When the RL controller has finished learning learning, its policy will be analyzed to see how well the guidance task is performed. This analysis is then used in the evaluation of the approximation.

Second 2) during the development of the RL controller the environment is mapped into a DTMC. The DTMC is constructed during every episode of the UAV in the environment. To gather as much knowledge on the state transitions as possible.

Then 3) optimize a BT to approximate the RL policy using GAs in the DTMC. GAs allows to quickly evaluate many different BT designs in the DTMC and to find the BT that approximates the RL the best.

At last, 4) results of the optimization and the evaluation of the evolved BT are presented. This BT is compared to the RL policy in the simulation environment to evaluate in the success of the method.

B. Task

The task for the UAV is to collect trash and navigate in a room without hitting walls. The task can be split up in two parts, collision avoidance and search trash. The UAV is equipped with limited amount of sensors which can sense the presence of a wall, trash or free space. This sensor configuration results in a partially observable environment where different physical locations could have identical sensor input. This results in a challenging task for the UAV to learn a policy perform well on the two tasks.

C. Environment

Figure 6 shows the environment for the UAV. The walls are fixed and do not change during learning or after.

When the UAV collects trash, as new piece will appear at a random location in the room. This keeps the probability to encounter a piece of trash for the UAV equal throughout all episodes. There are always 14 pieces of trash in the environment. The initial distribution of trash is fixed and shown with the UAV in the environment in Figure 6.

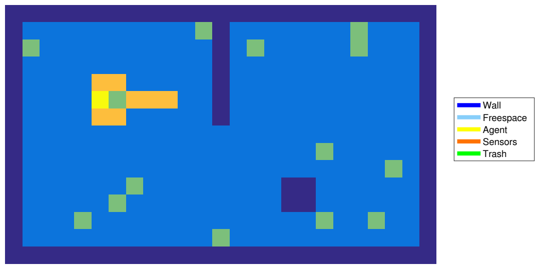


Fig. 6. Environment that shows the UAV and initial trash distribution.

D. State Representation

The sensor pattern of the UAV is shown in Figure 7. The UAV is equipped with 8 senses that can sens the contents of 8 cells around the UAV (*s0* to *s7*) in Figure 7. Each senses the presence of a *wall [0], free space [1] or trash [3]*. The i^{th} element corresponds to the observation of the s_i^{th} sensor. Figure 7 shows two UAV sensor configurations. Left shows that the UAV senses trash on *s6*. Right shows an UAV that senses trash at *s1* and a wall on its right on *s4* and *s5*.

This state representation allows the UAV to be put into other unknown environments. The UAV will still be able navigate around since the states are relative to the UAV.

The state-space of the UAV with the 8 sensors and in 3 different observations possibilities result in $3^8 = 6561$ unique states. However, the UAV can not sense anything behind a wall. So not all 6561 states can be observed. In addition to this sensor constraint, the number of states it can encounter is also limited by the dimension and geometry of the room, e.g. it will never senses two walls on either of its sides in Figure 6.

As mentioned in the Section VI-B the limited local information available to the UAV, the state of the UAV can only partially observe the environment around it. The current state of the environment cannot be fully be determined by the local state information of the UAV.

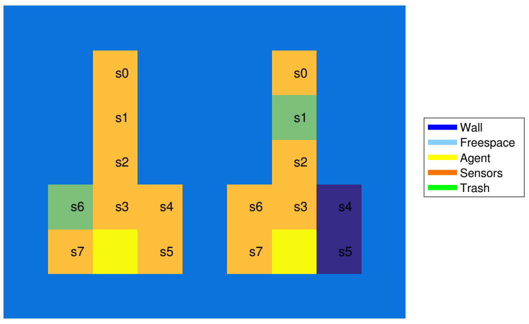


Fig. 7. Two UAV sensor configurations. Left: A UAV observes trash in *s6*. Right: A UAV observes trash at *s1* and observes a wall on its right on *s4* and *s5*.

1) Design RL controller:

The UAV can choose from three actions: *move forward, turn left, turn right*. To stimulate to move through the environment a negative reward of -1 is given when it turns left or right and no reward when it moves forward. A negative reward of -10 is given when the UAV collides with an obstacle. When the UAV collects trash it receives a reward of +20. This reward design encourages the UAV to move trough the environment, search trash and not hit obstacles.

The task starts each episode in a fixed initial state, $s_0 \in S$. The UAV maneuvers for 150 steps, following a policy, try to maximize its total reward for each episode.

All the learning parameters for the UAV are shown in Table II.

TABLE II
PARAMETER SETTINGS FOR THE RL CONTROLLER.

Parameter	Value
Number of episodes	1000k
Number of steps per episodes	150
Learning rate α	0.5
Discount factor γ	0.8
Exploration factor ϵ_0	0.5
Reward move forward	0
Reward turn	-1
Reward hit wall	-10
Reward collect trash	+20

The Q-update is shown in Algorithm 2. The two key steps in Algorithm 2 for the Q-learning are select an action and update of the Q-value. There are many ways to choose actions based on the current Q-value estimates. For the trade-off between explorations and exploitation an $\epsilon - greedy$ policy is implemented, a variation on the normal greedy selection. In both cases the UAV identifies the best action based on the state-action value. But $\epsilon - greedy$ selects the greedy action, the action with the highest state-action value, 1 - ϵ, of the time, where $\epsilon \in [0, 1)$. The policy selects one of the non-greedy actions a fraction ϵ of the time.

To ensure that during the end of the episodes the UAV exploits its knowledge instead of still explores the environment, a decay function for ϵ is implemented. The ϵ decay function is given in Equation (3). It calculates ϵ_k for each episode k.

$$\epsilon_k = \frac{\epsilon_0 * (n_d + 1)}{n_d * e_k} \tag{3}$$

where n_d is the number of total episodes divided by 100 and e_k the current episode. The trend of the decay of epsilon from episode 1 until 50000 is visualized in Figure 8.

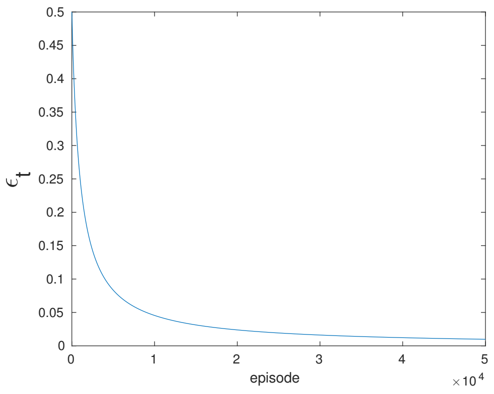


Fig. 8. ϵ-decay shown from episode 1 to 50000 with ϵ_0 = 0.5.

This policy becomes more greedy over time such that during the last episode during learning only 0.004% of the time the non-greedy action is selected.

Algorithm 2: Pseudo code Q-learning algorithm and includes transition matrix, $P(s, a|s')$, update.

```
1. Set parameters $\gamma$ & $\alpha$ & $\epsilon_0$
2. Initialize $Q - matrix$
for each episode do
    Initialize $s_0$;
    for $t \leq t_{max}$ do
        observe current state $s$
        select and perform action $a$
        observe subsequent state $s'$
        receive immediate reward $r$
        update $Q(s, a)$ according to:
        $Q(s, a) =$
        $(1 - \alpha_t)Q(s, a) + \alpha\{r + \gamma \max_{a' \in A} Q(s', a')\}$
        update $P(s'|a, s)$
        set $s' = s$
    end
end
```

Learning results

The total reward for the first 500 episodes of learning is visualized in Figure 9.

Figure 9 indicates the learning of the UAV on its guidance task during the first 500 episodes. It can be seen that for the first 50 episode the UAV collects a lower average reward then for the episodes after.

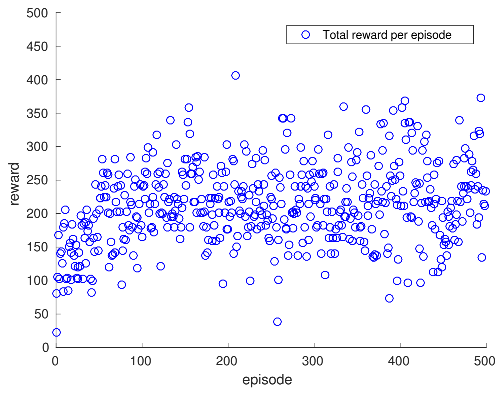


Fig. 9. Progression of total reward after each episode.

To evaluate the learned behavior in more detail the task is broken down to *three* behaviors: `collision avoidance`, `collect trash` and `search trash`. These three behaviors will also be used to evaluate the approximation.

`Collision avoidance`

Due to the state representation for the UAV, the UAV does not have to learn the complete environment to acquire knowledge early on. It learns to identify walls quickly on. Figure 10 indicate if the UAV observes a wall right in front and moves forward for the first 100 episodes with the ϵ greedy action selection.

For a run with 10000 episodes that uses a *greedy* policy for 10000 episodes, it takes the UAV 5598 episodes to not maneuver into a walls. Since in reality maneuver into the wall will increase the change to break the UAV, the collision avoidance behavior is seen as critical.

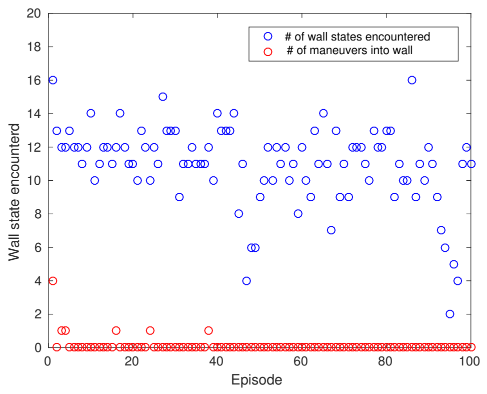


Fig. 10. Progression of the UAV learns to avoid the wall. Number of times states the that the UAV encountered a wall right in front of it is shown in blue. The number of times the UAV maneuvers forward in those states is shown in red.

A motion example of the UAV maneuver too a wall is shown in Figure 11. The UAV observes the wall in front, no pieces of trash are sensed on the other sensors and selects to turn to the left. After which the UAV continues to follow the wall. To turn away from the wall will result in two negative rewards of -1. This results in the UAV that follows the wall until new information on the environment is sensed.

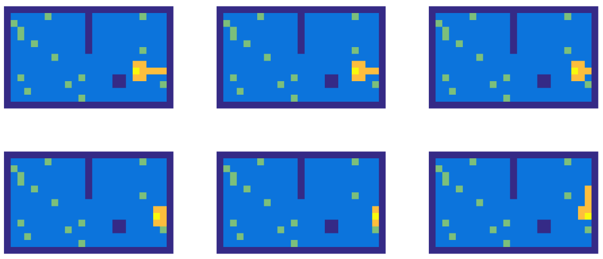


Fig. 11. A motion example of the UAV that shows a counter clockwise maneuver through the environment when it encounters a corner and not other sensor input is available. Dark blue indicates walls, light blue indicates free space, green indicates trash, yellow indicates the UAV's position. Read from left to right, from top to bottom.

`Collect trash`

Figure 12 shows a motion example of the UAV maneuver to collect trash when observed on one of its sensors. The UAV collect trash on either side of the UAV.

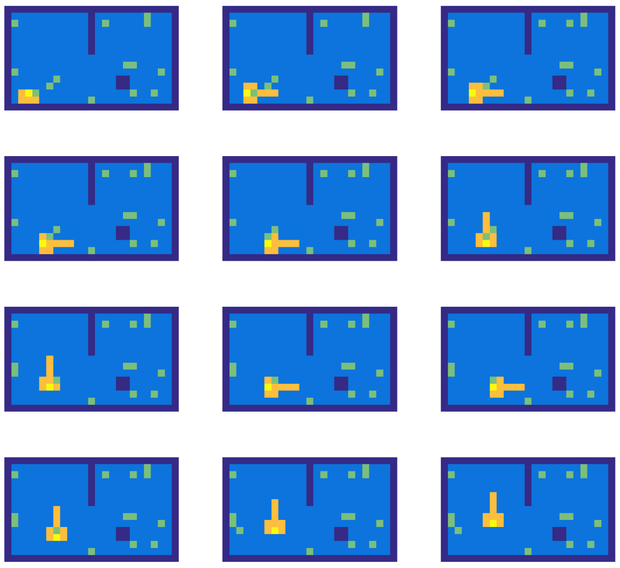


Fig. 12. A motion example of the UAV trained with Q-learning that collects trash as soon as it is sensed by one of its sensors. Dark blue indicate walls, light blue indicate free space, green indicate trash, yellow indicate the UAV's position. Read from left to right, from top to bottom.

The second observation is the choice between two pieces of trash. When the UAV observes trash in front and on its left or right, it decides to collect the piece of trash in front of the UAV. This makes sense since a turn results in a negative reward of -1 and move forward is not penalized. This behavior is presented in Figure 13.

`Search trash`

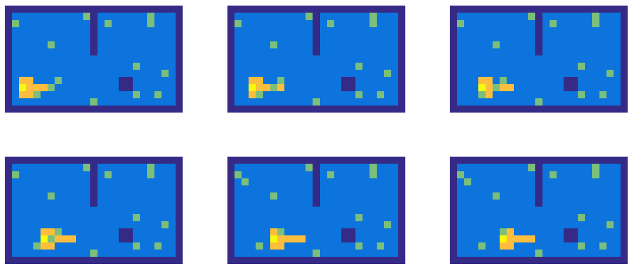


Fig. 13. A motion example of the UAV that shows to collect trash in front instead of right next to it. Dark blue indicate walls, light blue indicate free space, green indicate trash, yellow indicate the UAV's position.

Due to the partially observable state representation of the UAV, the UAV did *not* develop behavior to search for trash. Since a new piece of trash appears on a random location when the UAV collects one, the UAV depends on random chance to encounter it. This results in a fluctuation in the total reward per episode, as shown in Figure 14 for the first 10000 episodes.

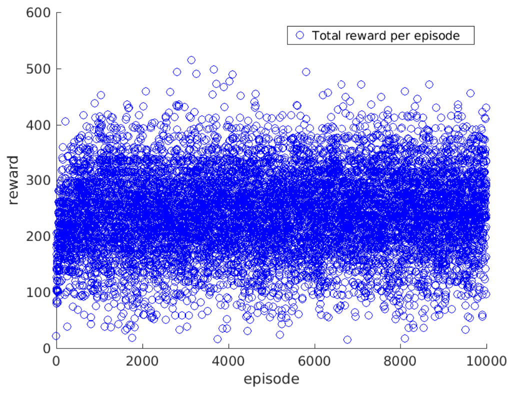


Fig. 14. Total reward for the first 100000 of the UAV that maneuvers through the environment.

Due to this stochasticity of encounter trash, the state-action values do not converge. The Q-update for every step during the last episode is shown in Figure 15. It shows large Q-updates for this last episode. These large updates indicate the UAV still learns a lot during the last episode. The states were this learning occur are investigated in more detail.

An example of two states with large Q-updates is presented in Figure 16. It shows the states of when the UAV observes nothing on its sensors just before finding a piece of trash in one of its sensors. An observe trash state has a high state-action value, but the UAV can transition to this state in more than one way for this example. These states will keep being updated on when either action is chosen. For both positions a different action is the best action which ends up in a high state-action value state.

Without the possibility to search for trash, the UAV develops a control policy to *increase it chance* to encounter trash. In Figure 17 this behavior is shown. The UAV uses the fact that walls provide the UAV with information on the environment

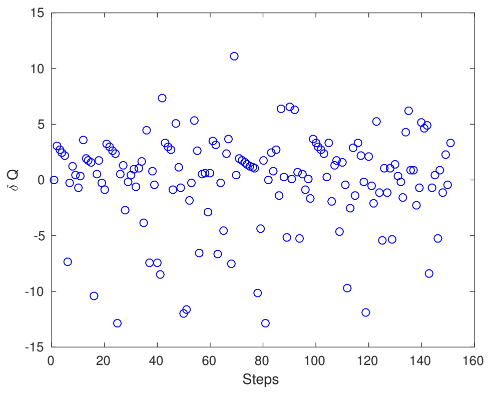


Fig. 15. The delta of the $Q_{t+1}(s,a)$ with respect to $Q_t(s,a)$ for every time step during the last episode. This indicates the stochasticity to encounter trash in the environment.

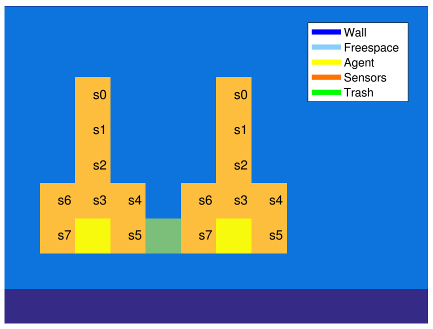


Fig. 16. Same state for the UAV, but different optimal action. This results in large Q-updates.

when it approaches a corner. When the UAV follows a wall *and* approaches a corner, the UAV turns before it reaches the corner. This results in sensors *s4* and *s5* to be free to observe the environment, instead of keep observe the wall.

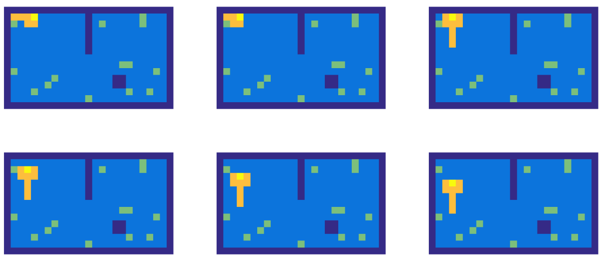


Fig. 17. A motion example of the UAV that shows a counter clockwise maneuver through the environment. When the UAV encounters a corner it turns before the corner to increase observability of the environment. Dark blue indicate walls, light blue indicate free space, green indicates trash, yellow indicates the UAV's position. Read from left to right, from top to bottom.

To summarize, it is observed that the collision avoidance and collect trash are deterministic. These task are executed successfully early on. The non-deterministic part, search trash, is not learned. Since the UAV can only partially observe the environment due to the UAVs limited amount of sensors, the

UAV does not have enough information to search for trash. To maximize its reward the RL learns control actions to increase its possibility to encounter trash and acts if trash is observed.

As a hash-table is used to store the Q-values, only states visited are stored and updated. At the end of all episodes, the UAV encountered 1041 states in the final table. This resulted in $1041 \times 3 = 3123$ state-action values which are used to identify the actions on the UAV. Verify all the state-action value pairs will take time, if even possible.

2) *Map learning into DTMC model:*

Due to the discrete nature of the simulation, the Markov chain is setup with the same states that the RL Q-learning algorithm uses in the Q-table. The number of unique states of the UAV encountered is 1041, this results in a transition matrix P of 1041×1041 per action a.

The frequency from state s, execute action a and end up in state s' is updated and stored in $P(s, a|s')$ every time an action is performed. This update is incorporated in Equation (1) after the Q-update. Due to the large number of states and discrete nature of the environment, many states are not reachable from other states, this results in a sparse matrix P.

3) *Approximating the RL policy with Genetic Algorithms using the DTMC:*

Due to the transition probabilities in the DTMC, a run from a same initial position can transition through the DTMC differently. The main performance is the fitness described in Algorithm 1. To generalize the performance of each BT in the population, each BT is run k times through the DTMC to determine its average fitness.

All runs start of with the same initial position, the same state the UAV stared in the simulation. An action a in s is executed if $P(s, a|s')$ is non-empty. If an action is selected from a state which did not occur during learning, i.e. $P(s, a|s')$ is empty, the BT gets a random state returned to continue its run through the Markov chain. This transition will have no extra effect on the fitness of the BT, other than the formulation in Algorithm 1. The run through the DTMC continuous until the maximum number of steps, t_{max}, through the DTMC is reached.

When a tree successfully identifies more than 90% of the actions, an additional fitness function is added to evaluate the BTs. This evaluates the BTs on the number of nodes where less nodes yield in higher fitness.

All the parameters used for the GA are shown in Table III.

TABLE III
PARAMETER VALUES FOR THE GENETIC ALGORITHM.

Parameter	Value
Policy executed in DTMC	BT
Max number of generation (G)	400
Population size (M)	100
Tournament selection size (s)	8%
Crossover rate (P_c)	50%
Mutation rate (P_m)	30%
Headless-Chick Crossover rate (P_{hcc})	10%
Maximum tree depth (D_d)	3
Maximum no. of children (D_c)	4
No. of simulation runs per generation (k)	5
No. of steps through DTMC (t)	100

Genetic Algorithm Results

The parameter which dictates the evaluation of the optimization is the mean fitness of the generation of BTs. Figure 18 show the population mean fitness and the mean fitness of the best individual in each generation. It can be seen that it takes 145 generations best score of the best individual to evolve. The mean fitness also improves gradually and then settles at around 0.7 a the normalized fitness axis.

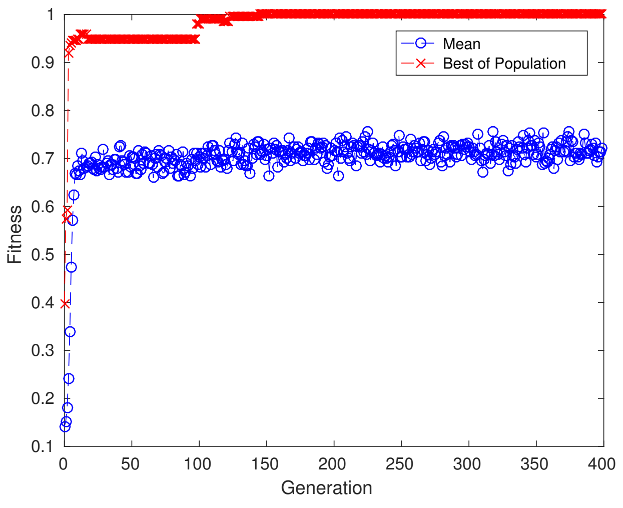


Fig. 18. Progression of the fitness score of the best individual and the mean of the population throughout the genetic optimization. The highest fitness of the best scoring individual is evolved after 145 generations.

Figure 19 indicates the tree size of the best individual and the tree mean size of the population. As shown in Figure 19 the average tree size begins around 55 nodes. Then drops immediately to around 20 to steadily increase to around 38 nodes. From there the number of nodes continue to reduce and eventually drops to around 20 again. The best individual BT after 150 generations has 21 nodes.

If the nodes from the best individual that have no effect on the final behavior are removed, called *pruning*, the final BT results in a BT with 6 nodes.

The optimized BT from the optimization after pruning is presented in Figure 20. The condition statements are indicated with the sensor s_i and the check for a wall (0), free of obstacles (1) or trash (3). *s3* checking for a wall in a condition node is presented as *s3 = 0*. The actions are indicated with *L* for turning left, *R* for turning right and *M* for moving forward.

4) *Optimized BT results & performance evaluation:*

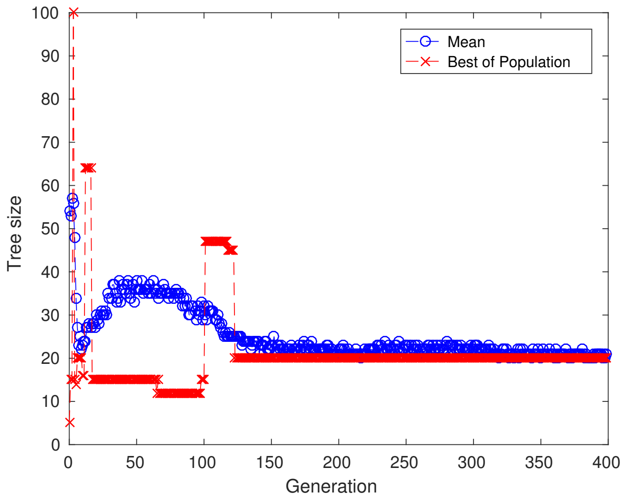


Fig. 19. Progression of the number of nodes of the best individual of the population and the mean of the population throughout the genetic optimization.

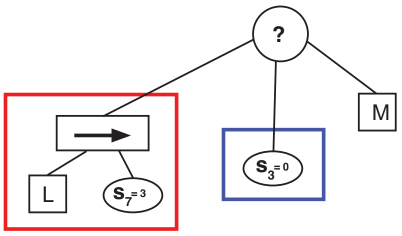


Fig. 20. Graphical depiction of genetically optimized BT. Blue box indicate wall avoidance behavior. Red box indicate trash collect behavior on the left of the UAV.

For the performance evaluation the results of the evolved BT are compared to the results of policy from the RL. In addition to the evolved tree the results of a manually adjusted BT are shown. An overview of the results is presented in Table IV.

TABLE IV
TABLE SHOWING SCORES OF THE DIFFERENT POLICIES THE UAV USED DURING A RUN IN THE SIMULATOR.

	Q-learning	BT	BT adjusted
Action-selection for all states	100	66 %	86 %
Action-selection in simulation	100	93 %	96 %
Average reward in simulation	283	248	266
Representation	3123-entries	6 nodes	20 nodes

For every state in the Q-table, the evolved BT is ticked to see which action it identifies. That action is compared to RL policy. The evolved BT is able to successfully identify the right action for 66% of all states in the Q-table.

Then the action selection of the evolved BT is compared with the RL policy during a run in simulation environment. The actions executed in the simulation environment are the actions selected by the evolved BT. In the simulation environment the evolved BT identifies successfully the right action for 93% of the time.

The evolved BT consists of 6 nodes, compared to the complete Q-table with 3123 entries. To compare the state-action pairs to the number of nodes is not straight forward. One node can check a condition in multiple states in the Q-table. Depending on the condition node, sensor and value, the number of states it checks vary. This makes it a difficult comparison. If the transition from state-action pair to node scale one-to-one, a reduction is size of 189 is obtained in the representation of the policy.

Although the evolved BT successfully identified the correct action in 93% of the states encountered in a simulation run, a closer look is taken into the behavior to the three characteristic subtask that are identified for the RL policy.

Collision avoidance

The highlighted blue box around the nodes in Figure 20 indicates the subtask collision avoidances. If a wall sensed by *s3*, the tree will not be evaluated further and follow the rules of the BT framework, the action that is set is *turn left*. Due to the rules of the framework, no further evaluation of the tree is necessary to verify this behavior. It will never select the action *move forward*.

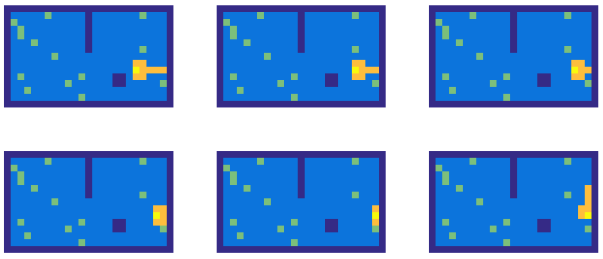


Fig. 21. A motion example of the UAV that shows a counter clockwise maneuvering through the environment when it encounters a corner and not other sensor input is available when follow the BT policy. Dark blue indicates walls, light blue indicates free space, green indicates trash, yellow indicates the UAV's position. Read from left to right, from top to bottom.

The characteristic behavior that explores the room counter clockwise follows from turning left when facing a wall. The motion example of the evolved BT given in Figure 21 also shows this.

Collect trash

The second characteristic subtask is collect trash when it is observed. Figure 22 shows that if a piece of trash is sensed on its left sensor the UAV collects it. This is also verified by a look at the evolved BT in Figure 20. This behavior is indicated with the red box.

UAV only reacts to trash sensed on left sensors. This behavior is verified with Figure 20. It shows that the BT does not include nodes to act on trash on its right sensors. A motion example of the UAV that follows the evolved BT action selection and ignoring the trash is presented in Figure 23. In Figure 23d and Figure 23e it observes trash on *s4* and then

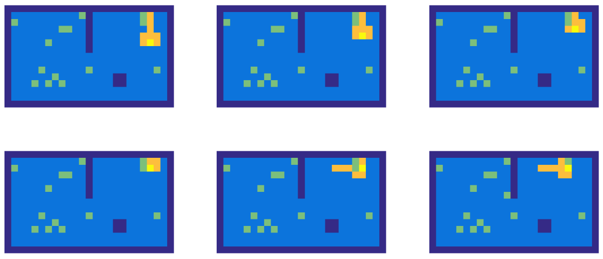


Fig. 22. A motion example of the UAV with optimized BT. UAV encounters trash and a wall and react to trash. Dark blue indicate walls, light blue indicates free space, green indicates trash, yellow indicates the UAV's position. Read from left to right, from top to bottom.

on *s5*. To not collect trash when it is observed on the right of the UAV results in a lower total reward.

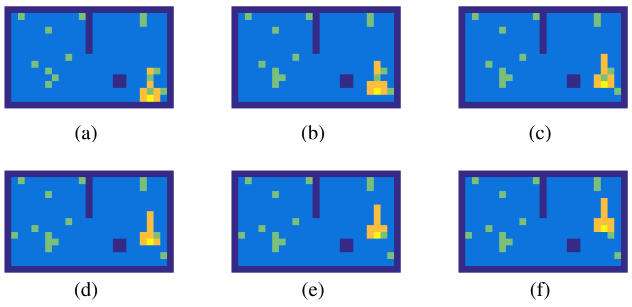


Fig. 23. UAV ignores trash on its right when it uses the evolved BT as policy. The UAV encounters trash and does not collect the trash at Figure 23d and Figure 23e.

When the UAV observes trash left or right *and* in front, the evolved tree decides to collect trash on its left. Select the action to move forward, as demonstrated in Figure 13, is not captured. Inspect the evolved tree in Figure 20 also shows that this behavior is not present.

Search trash

The third subtask, search for trash, is not learned by the UAV, it learned to increase its chance to encounter trash. The evolved tree does not include this behavior. Figure 24 show no nodes for when the UAV already follows a wall to turn before the corner. This leads to that *s4* and *s5* will still observe the wall. Since the chance to encounter trash decrease it is more likely to score a lower total reward.

To summarize, the behaviors are not present in the evolved BT: collect trash on right, collect trash in front first and increase the chance of encounter trash. If Figure 17 and Figure 24 are compared, a clear difference in behavior of the UAV is observed.

However, an advantage of the BT framework is the ease to make adjustments to the tree and add behaviors. The optimized BT is extended with 3 nodes to collect trash on its right. 7 nodes are added to collect trash in front first. 4 nodes are added to capture the learned strategy to turn before a corner. Figure 25 shows the adjusted BT.

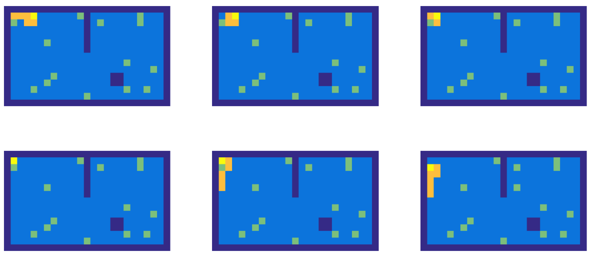


Fig. 24. A motion example of the UAV with optimized BT with same start position as UAV in Figure 17 with evolved BT as policy. The UAV follows wall until the corner and keep follow the wall. Dark blue indicate walls, light blue indicate free space, green indicate trash, yellow indicates the UAV's position. Read from left to right, from top to bottom.

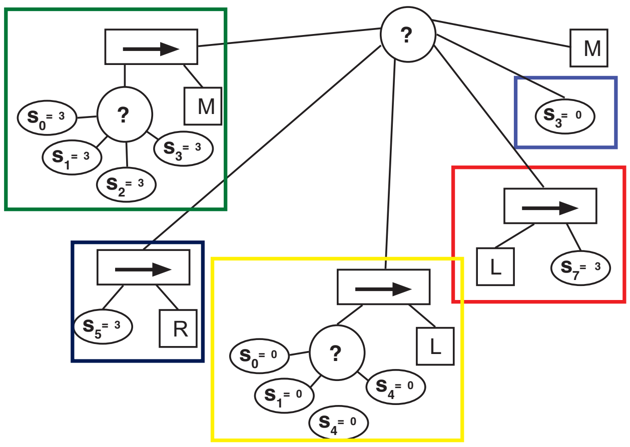


Fig. 25. Graphical depiction of genetically optimized BT with extended characteristic subtasks. Blue box indicates wall avoidance behavior. Red box indicates trash collect behavior on the left of the UAV. The green box indicates collect trash in front first. The dark blue box indicates collect trash on the right of the UAV. The yellow box indicates the increase trash encounter behavior.

To extend the evolved tree with the subtasks resulted in significantly better action-selection for all states in the Q-table. Increasing successful action identification for every time the tree is ticked with the states in the Q-table to 86%. The extension resulted in small increase the correct action identification when the adjusted BT is run in the simulation. The manually adjusted BT successfully identified the correct action for 96% of the states encountered.

The adjusted BT is now able to collect trash on its right, which yield in a higher average reward of 266 compared to the 248 for the non-adjusted BT.

VII. Discussion

Easy verification

From the 1041 states in the Q-table, for 488 states *s3* observes a wall, meaning a wall in front of the UAV. To verify the collision-avoidance behavior of the RL policy, the state-action values for all 488 states have to be checked. This results in 1464 state-action values for which that the state-action value for *move forward* must be lower than the state-action values for *turn right* or *turn left*. Compared to Figure 20, the node in the blue box gives the designer the ability to verify that the UAV will select the action turn left if a wall is sensed on *s3*.

Compared to the 1464 state-action values of the Q-table, the evolved BT provides the designer with 4 nodes to verify whether the UAV can avoid collisions. This indicates that a higher level of abstraction is obtained if a BT is used as the action-selection policy. This higher level of abstraction can lead to a better understanding of developed behavior.

Reinforcement Learning task

The task for the UAV to collect trash in an unknown environment and avoid obstacles results in three different subtasks; `collision avoidance`, `collect trash` and `search trash`. The RL policy successfully learns to control actions to avoid collisions and to collect trash. These two parts are deterministic and can be learned by the UAV.

But, due to the partial observability of the UAV, the UAV does not successful learn to search for trash. The UAV only learns control actions to increase its possibility to encounter trash. This is either due to the environment being complicated or the UAV's ability to sense the environment is too limited. The environment is based on a simple room, so future research will use a UAV that is able to get more information about its environment.

The aim of research is to show that the BT framework can be a useful tool to provide insight in a RL policy and make adjustments. This is partial successful and provided more insight in the policy structure of the RL controller. It is observed that the characteristic policy behavior for a RL agent is strongly coupled to successful action identification for states with low probability to transition to. It suggests that the success of the approximation of a RL policy depends on action identification for these states. Based on this observation future investigation should perform a sensitivity analysis identify and involve these states in the approximation.

This research focuses on learn the state-action values and storing them in a table, a *tabular* case. However, this method is *not* reliant on a tabular form for the state-action values. Approximations of a more compact parameterized function to represent the state-action values can also be used to provide the state-action value.

DTMC

For the states with a high probability to transition to the correct action is successfully identified by the evolved BT. A disadvantage of optimizing in the DTMC the lack of transitions to states with a low probability to transition to. The RL policy learns a counter clock wise wall follow behavior, thus keeps the wall on its right. This results that states with trash sensed left of the UAV are more likely to occur in the DTMC. As such, the probabilities of transition to a state with a piece of trash on the right of the UAV are low in the DTMC compared to the transition probabilities to states with trash on its left sensors.

The observed behavior to free up its sensors is also not approximated with the BT. The state for which this action needs to be identified also has a low probability to transition too. The approximation did not successful identify this behavior, but is viewed as characteristic behavior of the RL controller. Future research will investigate to involve low probability states better in the optimization without increasing computation time significantly.

Another interesting direction for future research is to investigate the active policy that is followed in the DTMC. The BT is used to step through the DTMC in this research, but an argument can be made to transition through the DTMC based on the RL policy. If an action selected by the evolved BT is used, then the UAV will end up in a different state compared to when the RL policy is followed. Subsequently, both policies will follow an other sequence through the DTMC during the rest of the evaluation of the BT.

When the BT is selected as active policy, states that are more likely to be actually encountered by the tree are evaluated. However, for wrong action identification, the tree moves to a state that is is less likely to be visited by the RL policy. It is interesting to see that the optimization has difficulty to evolve a BT that captures the characteristics of a RL controller. When the RL policy is followed, the states that are important to the RL policy might be encountered more often. This might improve the optimization result.

A disadvantage when the RL policy is selected as active policy is that the evaluation is most likely done in a subset of the Q-table and DTMC. If the collision avoidance behavior is missed, the RL action selection will move on from this state and this allows the rest of the selected actions of a BT to obtain high fitness. This BT, with high fitness, will however not perform well in simulation and would get stuck when facing a wall.

Future work will also investigate mapping continuous state-space problems into a DTMC. In addition, the active policy in the DTMC and how it relates to the optimization is interesting to investigate to understand for future work. And further investigation into the optimizing parameters for the GAs used in this paper can be performed.

Genetic Algorithm

Without task specific knowledge implemented in the fitness function, the converged solution is not steered by the human designers. This allows for easy implementation for a wide range of problems. However, to capture characteristic behavior of a RL policy, a tailored fitness function might increase the success of the optimization. Future research will investigate specifying a fitness function to better capture the characteristic behavior of the RL policy.

VIII. Conclusion

This paper presents a method to automatically evolve a BTs using GAs to approximate a developed RL policy, normally captured in a difficult to interpret Q-table. The evolved BT provides a framework in which potential wrong and dangerous decisions of the controller can be identified. The graphical depiction of the evolved BT allows for easy verification for the collision avoidance subtask with only 4 nodes, compared to the verification of 1464 state-action values in the Q-table. This suggests that a BT provides easy verification of subtasks for the designer of RL learned behavior.

The evolved BT successfully identifies the right action for 66% of all the states in the Q-table. When the evolved BT selects the actions on the UAV in the simulation environment, it successfully identifies the right action for 93% of the encountered states. This suggests that approximation of the complete Q-table of a RL policy is not necessary to identify the right actions for a run in the simulation.

The evolved BT is extended with the characteristic subtasks that are not evolved during the optimization. This increases successful action identification to 86% for all states in the Q-table. When the extended BT is run on the UAV in the simulation environment, the tree identifies the right action for 96% of all states encountered in the simulation.

The approximation is performed using GAs for BTs in a DTMC. This DTMC is constructed during the learning of the guidance task by the UAV and allows for evaluation of BT designs without the need of a simulation environment.

By omitting task specific knowledge in the fitness function, the converged solution is not steered by the human designers. This allows for a easy implementation in a wider range of problems. The fitness function only increased the fitness of a BT if the same action was identified by the RL policy.

Concluding, the method presented indicates that the BT framework provides the designer with workable tools to identify, adapt and verify subtasks presenting the learned behavior of a RL controller.

References

[1] T. Yamasaki, H. Sakaida, K. Enomoto, H. Takano, and Y. Baba, “Robust trajectory-tracking method for UAV guidance using proportional navigation,” in *2007 International Conference on Control, Automation and Systems*. Institute of Electrical & Electronics Engineers (IEEE), 2007.

[2] P. Ögren, “Increasing modularity of UAV control systems using computer game behavior trees,” in *AIAA Guidance, Navigation, and Control Conference*. American Institute of Aeronautics and Astronautics (AIAA), Aug 2012.

[3] R. Sutton and A. Barto, “Reinforcement learning: An introduction,” *IEEE Trans. Neural Netw.*, vol. 9, no. 5, p. 1054, Sep 1998.

[4] A. Nedić and D. P. Bertsekas, *Discrete Event Dynamic Systems*, vol. 13, no. 1/2, pp. 79–110, 2003.

[5] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, May 1992.

[6] P. He and S. Jagannathan, “Reinforcement learning neural-network-based controller for nonlinear discrete-time systems with input constraints,” *IEEE Trans. Syst., Man, Cybern. B*, vol. 37, no. 2, pp. 425–436, apr 2007.

[7] C. Bishop, “Novelty detection and neural network validation,” *IEE Proceedings - Vision, Image, and Signal Processing*, vol. 141, no. 4, p. 217, 1994.

[8] R. Dromey, “From requirements to design: formalizing the key steps,” in *First International Conference onSoftware Engineering and Formal Methods, 2003.Proceedings.* Institute of Electrical & Electronics Engineers (IEEE), 2003.

[9] A. J. Champandard, “Behavior trees for next-gen game ai,” https://aigamedev.com/insider/presentations/behavior-trees/, 2007, (Accessed on 11/03/2016).

[10] K. Y. W. Scheper, S. Tijmons, C. C. de Visser, and G. C. H. E. de Croon, “Behaviour trees for evolutionary robotics,” *Artificial Life*, vol. 22, pp. 23–48, 2016.

[11] E. L. Thorndike, *Animal intelligence; experimental studies, by Edward L. Thorndike.* New York,The Macmillan company,, 1911.

[12] D. Isla, “Handling Complexity in the Halo 2 AI,” in *GDC 2005 Proceeding*, 2005.

[13] J. Koza, “Genetic programming as a means for programming computers by natural selection,” *Statistics and Computing*, vol. 4, no. 2, jun 1994.

[14] M. Mitchell, *An introduction to genetic algorithms*, ser. Complex adaptive systems. Cambridge (Mass.): MIT press, 1996, a Bradford book.

[15] V. Ciesielski and P. Scerri, “Real time genetic scheduling of aircraft landing times,” in *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360).* Institute of Electrical & Electronics Engineers (IEEE).

Part II

Literature Study

Chapter 2

Reinforcement Learning

This chapter provides an overview of the RL framework. The majority of the information presented is adopted from R. Sutton & Barto (1998). The advantages and limitations are discussed. Followed with the recent advances in RL.

2-1 Reinforcement Learning the Basics

RL is an active area of machine learning research that also recieves attention from the fields of decision theory and control engineering. To clarify: "The simplest RL algorithms make use of the common sense idea that if an action is followed by a satisfactory state of affairs, or an improvement in the state of affairs, then the tendency to produce that action is strengthened, i.e., reinforced. This is the principle articulated by Thorndike in his famous Law of Effect (Thorndike, 1911.)." (Barto & Dietterich, 2004).

Lets consider a simple scenario from Barto & Dietterich (2004) to explain the concept.

Consider your holiday in the South of France. You see mobile phone users that wander around the camping field to obtain good reception since coverage is poor. Move around and monitor its signal strength indicator. They do this until they found a place with a good signal, or the best signal under the circumstances. Here, the information they receive do not directly tell them what to do to obtain a good reception. Nor is each reading telling them in which direction they should move. Each reading simply allows them to evaluate how good current situation is.

In RL the *agent* receives, at each time step t, a representation of the *state* s_t of the *environment* and performs an *action* a_t. The selection of the action is done according to a *policy*. This changes the state of the environment to a new state s_{t+1}. The environment responds and gives the agent a *reward* r_{t+1}, according to a reward function. The reward function is based on how good the action was to that particular state. The general RL structure is given in Figure 2-1.

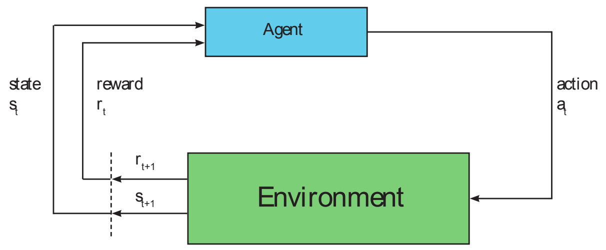


Figure 2-1: General Reinforcement Learning structure. The agent performs action a_t in the environment. The environment responds by presenting the agent a new state s_{t+1} and a reward r_{t+1}. Adapted from (R. Sutton & Barto, 1998).

The **state** state $s_t \in S$, where S is the set of possible sates, is defined by a signal from the environment to the agent, this represents properties of the environment to the agent.

The **action** $a_t \in A$, where A is the set of actions available, is the output of the agent. The action taken influences the state of the environment and the selection of an action is don following a certain policy.

A **policy** $\pi_t(s, a)$ maps state s to an action a. It defines the agent's behavior at a given time. It is written as $\pi_t(s, a) = P(a_t = a|s_t = s)$. There are different types of policies. An optimal policy, denoted by π^*, is the policy that corresponds to the greatest received return.

The **environment** is defined as everything outside the agent. It receives an action *from* the agent and outputs a new state and reward *to* the agent.

The decision maker is called the **agent** in the RL framework. It influences the state of the environment, performs actions and receives rewards based on the state transitions. The agent is not told which actions to take, but must discover which actions yield the most reward by experience.

The agent receives a **reward** from the environment, based on how good or bad an action in a particular state was.

The goal of the agent is to maximize **return**, a function of rewards, over a trajectory generated by a policy. A reward function defines the goal in a RL problem. It maps each perceived state or state-action pair of the environment to a single number, a reward, indicating the desirability of that state. A RL agent's objective is to maximize the total reward it receives in the long run. The simplest case is just the sum of the rewards: $R = r_t + r_{t+1} + r_{t+2} + ... + r_T$, where T is the final time step.

Maximization of the return is done by adapting the policy that the agent is following until an optimal policy is found. The value of the final state is always zero. This way of calculate the return is only possible in application where a final time step N can be defined.

An additional concept that is needed is that of *discounting*. According to this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized. In particular, it chooses a_t to maximize the expected discounted return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2-1}$$

where γ is the discount factor, $\gamma \in [0, 1)$.

For maximizing return, the simplest action selection rule is to select the action with highest estimated action reward, this is called the **greedy** action. Always choose the action with the highest value is done following the greedy policy.

A simple alternative is to behave greedy most of the time, but every once in a while, say with small probability ϵ, is ϵ-greedy. For both policies the agent identifies the best action based on the highest value. But $\epsilon - greedy$ selects the greedy action 1 - ϵ, of the time, where $\epsilon \in [0, 1)$. The ϵ-greedy policy selects one of the non-greedy actions a fraction ϵ of the time.

2-1-1 Markov and Partially Observable Decision Processes

A state signal that succeeds to retain all relevant information is said to be Markov, or the have the Markov property. It can be referred to as "independence of path" property because all that matters is captured in the current state signal; its meaning is independent of the "path" or history of the signals that have led up to it.

The study of RL agents is greatly facilitated with this convenient mathematical formalism. This formalism, known as Markov Decision Processs (MDPs) in RL, is well established. It assumes a simplifying condition on the agent which is, however, largely compensated by the gained ease of analysis. MDPs are important in RL because decisions and values are assumed to be functions only of the current state. Although RL is by no means restricted to MDPs , this discrete-time, countable state and action formalism provides the simplest framework in which to study basic algorithms and their properties.

A finite MDP models the following type of problem. At each stage in a sequence, an agent observes a system's state s, contained in a finite set S, and executes an action a selected from a finite, non-empty set, A_s, of admissible actions. The agent receives an immediate reward having expected value $R(s, a)$, and the state at the next stage is *s'* with probability $P(s, a|s')$, s, $s' \in S$, $a \in A$, together form what RL researchers often call the one-step model of action a.

These quantities, $P(s, a|s')$ and $R(s, a|s')$, completely specify the most important aspects of the dynamics of a finite MDP . The description of MDPs assumes that the environment is fully observable. With this assumption, the agent always knows which state it is in. This, combined with the Markov assumption for the transition model, means that the optimal policy depends only on the current state.

When the environment is only partially observable, the situation is much less clear. The agent does not necessarily know the state it is in, so it cannot execute the action $\pi(a)$ recommended for that state. Furthermore, the utility of a state a and the optimal action in a depend not just on s, but also on *how much the agent knows* when it is in s. For these reasons, Partially Observable Markov Decision Processs (POMDPs) are usually viewed as much more difficult than ordinary MDPs . We cannot avoid POMDPs, however, due to sensor limitations and noise in the real world.

2-1-2 Value Function

Almost all RL algorithms are based on estimating *value functions*. As mentioned previously, functions of states that estimate how good it is for the agent to be in a given state. The notion of "how good " is defined in terms of future rewards that can be expected. Accordingly, value functions are defined with respect to particular policies. There are two types of functions: the *state function* and the *state-action value function*.

The value function,$V^\pi(s_t)$, is the expected return when starting in s_t and following policy π, and is V^π the value function corresponding to policy π. The objective is to find an optimal policy, π^*.

$$V^\pi(s) = E_\pi\{R_t|s_t = s\} = E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s\} \tag{2-2}$$

where E_π denotes the expected value given that the agent follows policy π.

In the same way, the state-action value function, Q^π is the *action-value function for policy* π, see Equation (2-3).

$$Q^\pi(s,a) = E_\pi\{R_t|s_t = s, a_t = a\} = E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a\} \tag{2-3}$$

The optimal action-value function, Q^*, assigns to each admissible state-action pair s,a the expected infinite-horizon discounted return for executing a and s and thereafter following an optimal policy. The difference is represented graphical in Figure 2-2.

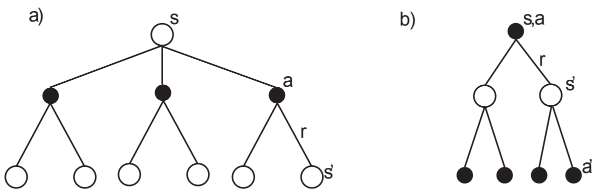


Figure 2-2: a) Action value. b) State-action value.

Bellman equation A fundamental property of these functions that they satisfy particular recursive relationships. So, when the environment can be described as an MDP, it is possible to analytically calculate the optimal policy with this recursive relationship. This recursive relationship is the *bellman equation* given in Equation (2-4).

$$
\begin{aligned}
V^{\pi}(s) &= E_{\pi}\{R_t | s_t = s\} \\
&= E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{r+k+1} | s_t = s\} \\
&= E_{\pi}\{r_{t+1} + \sum_{k=0}^{\infty} \gamma^k r_{r+k+2} | s_t = s\} \\
&= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_{t+1} = s'\}] \\
&= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V^{\pi}(s')]
\end{aligned}
\tag{2-4}
$$

Equation (2-4) is the *Bellman equation for* V^{π}. The Bellman equation averages over all the possibilities, weighting each by its probability of occurring. It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way.

Analogous equations exist for Q^{π} as shown in Equation (2-5).

$$
Q^{\pi}(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V^{\pi}(s')] \tag{2-5}
$$

2-1-3 Dynamic Programming

Dynamic Programming (DP) is a standard method that provides an optimal stationary policy for the stochastic MDP problem. Dynamic Programming (DP) algorithms exploit the fact that value functions satisfy the recursive relation of the *Bellman equation.* It actually encompasses a large collection of techniques, all of them based on a simple optimality principle and on *three* basic theorems. It can be divided into 1) value iteration 2) policy iteration 3) policy search.

Policy iteration algorithms evaluate policies by constructing their value functions (instead of the optimal value function), and use these value functions to find new, improved policies.

$$
\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} .... \xrightarrow{I} \pi^* \xrightarrow{E} V^* \tag{2-6}
$$

Value iteration algorithms search for the optimal value function, which consists of the maximal returns from every state or from every state-action pair. The optimal value function is used to compute an optimal policy.

$$
V_{k+1}(s) = \underset{a}{max} E\{r_{t+1} + \gamma V_k(s_t + 1) | s_t = s, a_t = a\} \tag{2-7}
$$

Policy search algorithms use optimization techniques to directly search for the optimal policy. Policy iteration consists of two simultaneous, interacting processes, one making the value function consistent with the current policy, called policy evaluation, and the other making the policy greedy with respect to the current value function, policy improvement. The term generalized policy iteration is used to refer to the general idea of letting policy evaluation and policy improvement processes interact.

Policy iteration, value iteration and generalized policy iteration can be seen as framework for RL learning techniques. Within each of the three subclasses of RL algorithms, two categories can be further distinguished, namely off-line and on-line algorithms. Off-line RL algorithms use data collected in advance, whereas on-line RL algorithms learn a solution by interacting with the process in real time. A graphical overview is given in Figure 2-3.

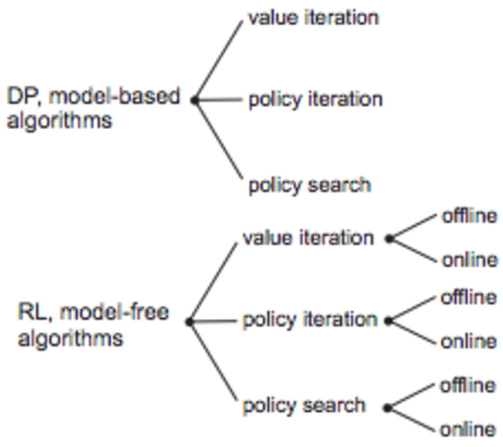


Figure 2-3: DP structure relating to RL. Adapted from (R. Sutton & Barto, 1998).

Limitation relating to Dynamic Programming DP methods operate in sweeps through the state set, performing a full backup operation on each state. Each backup updates the value of one state based on the values of all possible successor states and their probabilities of occurring. Therefore, DP requires a complete and accurate model of the environment.

2-1-4 From Dynamic Programming to Monte Carlo

The algorithms for exact value iteration require the storage of distinct return estimates for every state or state-action pair. When some of the state variables have large or infinite number of possible values (e.g., continuous), exact storage is not possible, and the value functions is approximated. Subsequently, large or continuous action spaces make the representation of Q-functions challenging.

Clearly, the expectation cannot be computed exactly, and must be estimated from a finite number of samples, e.g., by using Monte Carlo methods. Monte Carlo methods are ways of solving the RL problem based on averaging sample returns. To ensure that well-defined returns are available, we define Monte Carlo methods only for episodic tasks. That is, we assume that the experience can be divided into episodes, and that all episodes eventually terminate no matter what actions are selected. It is only upon the completion of an episode that value estimates and policies are changed.

Monte Carlo methods are thus incremental in an episode-by-episode sense, not in a step-by-step sense. The term ‘Monte Carlo’ is sometimes used more broadly for any estimation method whose operation involves a significant random component. Here it is used for methods based on averaging complete returns (as opposed to methods that learn from partial returns, considered in the next chapter).

Despite the differences between Monte Carlo and DP methods, the most important ideas carry over from DP to the Monte Carlo case. Not only the same value functions are computed, but they interact to attain optimality in essentially the same way. By considering Monte Carlo methods for learning the state-value function $V^{\pi}(s)$ for a given policy. An obvious way to estimate it from experience, then, is simply to average the returns observed after visits to that state. As more returns are observed, the average should converge to the expected value. This idea underlies all Monte Carlo methods.

The Monte Carlo methods are essentially the same as presented for state values as they are for the action value estimation $Q^{\pi}(s, a)$.

Limitations relating Monte Carlo Method Maintaining sufficient exploration is an issue in Monte Carlo control methods. It is not enough just to select the actions currently estimated to be best.

2-1-5 Temporal Difference Learning

As discussed, solving MDP problems through DP requires a model of the process to be controlled, the value iteration operators explicitly use the transition probabilities. Monte Carlo methods, on the other hand, use a simulation model that does not require calculation of transition probabilities, but does require complete runs before updates can be performed.

Temporal-Difference (TD) learning is a combination of Monte Carlo ideas and DP ideas. TD resembles a Monte Carlo method because it learns by sampling the environment according to some policy, and is related to dynamic programming techniques as it approximates its current estimate based on previously learned estimates. For on-line purposes, the Monte Carlo Methods suffers form a drawback: $\mathrm{V}_{\pi}(x_0)$ can only be updated after $\mathrm{V}_{\pi}(x_0; m)$ is calculated from a complete simulation run. The TD method provides an elegant solution to this problem by forcing updates immediately after visits to new states.

With TD learning the value function is updated *during* the episode. With Monte Carlo methods we have to wait until the episode is finished before we start updating the value and policy.

$$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)] \tag{2-8}$$

TD learning samples the expected value in 2-8 *and* it used the current estimate V_t instead of the true V^{π}. Thus the TD methods combine the sampling of Monte Carlo with the *bootstrapping* of of DP. That is, they update one estimate based on another estimate. Many RL methods perform bootstrapping, even those that do not require a complete and accurate model of the environment.

Q-learning One of the most important breakthroughs in RL was the development of an off-policy TD control algorithm know as *Q-learning* (Watkins & Dayan, 1992). The most simple form, *one-step Q-learning*, is defined by

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2-9}$$

The learned action-value function, Q, directly approximates Q^*, the optimal action-value function, independent of the policy being followed.

SARSA SARSA is similar to Q-learning except that the maximum action-value for the next state on the right hand side of Equation (2-9) is replaced by the action-value of the actual next state-action pair

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2-10}$$

This rule uses the element of the events $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, given rise to the name SARSA.

Actor-Critic Method The Actor-Critic method is a TD method that has a separate memory structure to explicitly represent the policy independent of the value function. It is an on-policy algorithm, like SARSA.

In contrast to Q-learning and SARSA, Actor-Critic methods keep track of two functions; a Critic that evaluates states and an Actor that maps states to a preference value for each action. After an experience (s_t, a_t, r_t, s_{t+1}), Actor-Critic methods make a temporal difference update to the Critic's value function V. After each action selection, the critic evaluates the new state to determine whether things have gone better or worse than expected. The general structure of this Actor-Critic method is presented in Figure 2-4.

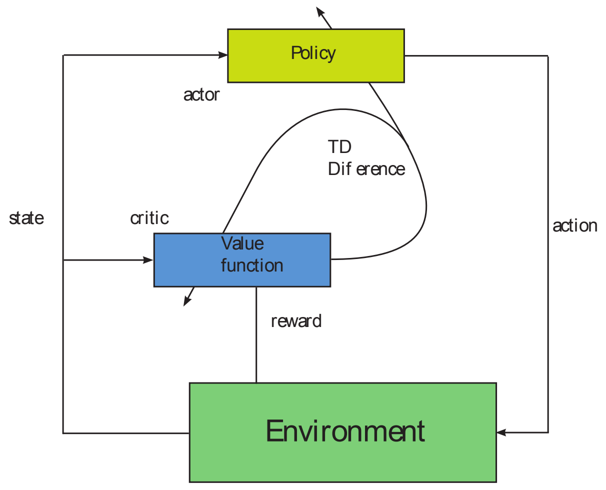


Figure 2-4: General Reinforcement Learning Structure. Adopted from (R. Sutton & Barto, 1998).

Function Approximation So far the estimates of value functions is represented as a table with one entry for each state or for each state-action pair. This is a particularly clear and instructive way to represent the learning, but is limited to tasks with small numbers of states and actions. The problem is not just the memory needed for large tables, but the time and data needed to accurately fill them.

Function approximation is an instance of supervised learning, the primary topic studied in machine learning, artificial neural networks, pattern recognition, and statistical curve fitting. Function approximation is already widely applied to approximate the state-action values in a RL problem. One achievement is the combination of ANNs and RL in machine learning research. This combination is implemented as a backgammon player by Tesauro (1994).

Qiao et al. (2009) presented an automation learning and navigation strategy based on dynamical structure neural network and RL. Results showed that the robot can learn the correct action and finish the navigation task without external guidance. In their proposal, ANNs were used as function approximator for the RL policy.

2-1-6 Advantages

RL enables a agent to autonomously discover an optimal behavior through trial-and-error interactions with its environment. Instead of explicitly detailing the solution to a problem, in RL the designer of a control task provides feedback in terms of a reward function that measures the one-step performance of the agent. It is an efficient method to control in unknown environment. It can learn how to control after its environment has changed beyond the scope of the original controller.

2-1-7 Limitations

When Bellman explored optimal control in discrete high-dimensional spaces, he faced an exponential explosion of states and actions for which he used the term *curse of dimensionality.* As the number of dimensions grows, exponentially more data and computation are needed to cover the complete stateaction space.

While often simpler than specifying the behavior itself, in practice, it can be surprisingly difficult to define a good reward function RL. For challenging tasks, it is often complex and time-consuming for designers to precisely specify rewards.

Traditionally the learning of a RL controller is stored in lookup tables, *Q-tables.* Present day controllers for safety critical applications, like for example UAV guidance, undergo extensive tests to qualify their operation. Although analysis of Q-tables is possible, the difficulty to provide insight in its behavior causes practical issues to VV of such a controller, especially for large Q-tables Bishop (1994).

Furthermore, Q-tables do not lend themselves well to manual adaptation. This limits the deployment and wider use of RL in control applications.

To apply RL in robotics, safe exploration becomes a key issue of the learning process. Repairing a robot system is a non-negligible effort associated with cost, physical labor and long waiting periods. And as the dynamics of a robot can change due to many external factors ranging from temperature to wear, the learning process may never fully converge.

2-2 Recent Advances

Recently, combinations with RL have been applied to many problems. Yoshikawa et al. (2008) introduced a RL algorithm with a hierarchical evolutionary mechanism to evolve adaptive action value tables. The algorithm evolves several Q-Learning parameters, such as state discretization data, learning rate α, discount factor γ and searching rate (non deterministic action selection).

2-2-1 Hierarchical Reinforcement Learning

'Flat' RL works well but on small problems. However for complex problems, scaling the problem gets very large very quickly, the so called curse of dimensionality.

Recent attempts to combat the curse of dimensionality have turned to principled ways of exploiting temporal abstraction, where decisions are not required at each step, but rather invoke the execution of temporally-extended activities which follow their own policies until termination. This leads naturally to hierarchical control architectures and associated learning algorithms, Hierarchical Reinforcement Learning (HRL).

(Dietterich, 2000) presented a new approach to HRL. Decomposing MDPs into a hierarchy of smaller MDPs and decomposing the value function, known as MAXQ. The MAXQ hierarchy proves formal results on its representational power and establishes five conditions for the safe use of state abstractions. However it assumes that the programmer can identify useful subgoals en define subtasks.

Yan et al. (2010) stated that a well designed, heuristic, reward function along with HRL can decrease the number of impractical acts of exploration which in turn allows the agent to interact easily and quickly with the environment. Their results show that their method can overcome the huge state space of the environment. However, if the subtask has more than one sub-goal, their HRL algorithm can only converge to the recursive optimal.

Also video games provide a rich testbed to apply RL for artificial intelligence methods. For example, HRL has already been applied to computer games, (Xiaoqin et al., 2009).

Barto & Mahadevan (2003) and Al-emran (2015) have performed good surveys on the current HRL knowledge. There are several powerful HRL models such as Hierarchical Abstract Machines (HAMs) (Parr, 1998), Options (R. S. Sutton et al., 1999), and ALisp (Andre & Russell, 2001). These models provide a general framework for scaling RL to problems with large state spaces by using the task or action structure to restrict the space of policies.

Limitations of HRL One of the limitations of HRL is how to define components. When these components are defined, it is easy to find sub-optimal solutions. And with the predefined hierarchical structure, it is not possible to escape these local maxima.

And policies are restricted to sequential combinations of activities, predefined by the hierarchy implemented by the designer.

2-2-2 Continuous Reinforcement Learning

The progress so far has been mostly constrained to the discrete formulation of RL problems. But some work on Q-learning for continuous time systems has been investigated by Palanisamy et al. (2015), although without any convergence and stability guarantees.

The authors (Palanisamy et al., 2015) propose a Q-learning approach to solve the continuous-time infinite-horizon optimal control problem by writing the Q-function with respect to the state, the control input and the derivatives of the control input and without having knowledge of the system dynamics.

Doya (2000) has presented RL algorithms to deal with continuous-time continuous-state control tasks without explicit quantization of state and time. He uses the continuous TD(λ)-learning with the actor-critic method to learn the control command sequence. He performed a experiment with a pendulum swing-up task with limited torque. The swing-up task was accomplished with a number of trails several times fewer than by the conventional discrete Actor-Critic.

A continuous Q-learning method was presented in (Millán et al., 2002) by using an incremental topology preserving map to partition the input space and the incorporation of bias to initialize the learning process. The resulting continuous action is an average of the discrete actions of the winning unit weighted by their Q-values. More interesting, the author also showed the experimental results in robotics indicating that the continuous Q-learning method works better than the standard discrete action version of Q-learning in terms of both asymptotic performance and learning speed. This continuous Q-learning method still focus on single-agent systems.

This approach, although using a finite set of target actions, deals with this problem by selecting real-valued actions obtained by interpolation of the available discrete actions on the basis of their utility values. Despite of this capability, the learning performance of these algorithms relies on strong assumptions about the shape of the value function that are not always satisfied in highly non-linear control problems (Lazaric et al., 2007).

However, there is still a need to develop methods for high-dimensional function approximation and for global exploration.

Chapter 3

Behavior Trees

This chapter presents an introduction to the BT framework. The basics of the framework are explained and the advantages of are discussed. Then the recent advances on the research on BTs is presented.

3-1 Behavior Tree

Digital games and robotics share a common goal: to develop intelligent artificial agents to interact with the environment and interact with other agents, whether real or virtual. To develop intelligent agents, it is necessary to build a vast repertoire of behaviors. Behaviors involve control of various actions over a period of time.

Traditionally, user-defined autonomous behaviors are described with FSMs. Where each state has transition conditions and the execution logic contained in it. FSMs are computationally efficient, since all changing conditions are inside the current state and no other state run simultaneously.

However, FSMs have limitations. To add or remove a behavior, it is necessary to change the conditions of all other states that can transition to the new or old state. This also makes it impractical to reuse FSMs. The graphical readability is lost for FSMs with many states.

In 2005, the BT was introduced in the gaming industry to address the problem of control of NPC in games, in a more efficient way compared to FSMs, and since then, BTs have been replacing FSMs in various segments, including robotics.

The essence of a BT is expressed as *a graphical modeling language primarily used in software engineering which facilitates the representation of organizational elements in a large-scale system.* This chapter discusses its emergence and how it finds it way into the control of Micro Aerial Vehicles (MAVs).

3-1-1 The Emergence

Dromey (2003) developed a new system design method to address the problem of systematically *translate* large, complex requirement documents into a *structured model* of the system. Over the years he used different names and ultimately settled for **Behavior Engineering**. The method is centered around a notation to express system behavior with Behavior Trees.

The BT presents an intuitive, stepwise process to go from functional requirement to a design. Each requirement translates into its own, small BT, and each node in the tree is tagged with the number of the requirement from which it was translated, it allows traceability back to the original informal requirements.

Then the requirements are progressively integrated into a whole-system tree, to find syntactically matching constructs. This process will reveal *inconsistencies*, *redundancies*, *incompleteness*, and *ambiguities*. The constructed tree serves as the basis for discussion between developer and client for validation purposes, using the traceability tags on each node to help understand the problem space and clarify system and software requirements.

The BT notation got the attention of computer game designers when they searched for a behavior management system for NPCs. They needed a framework to control the modularity, re-usability and complexity of the NPCs. The BT, although in a different form as was anticipated by Dromey, proved to be suited to capture this behavior (Isla, 2005).

In computer games, the control architecture of automated opponents if often designed with FSMs. But unlike a FSMs, a BT is a tree of nodes that are hierarchical structured to control the flow of decisions of an AI entity. At the extents of the tree, the leaves, are the actual commands that control the AI entity. Their can be various types of leaf nodes to control the AI to reach the sequences of commands best suited to the situation.

BTs provide a hierarchical way to organize behaviors in a descending order of complexity; broad behavioral tasks are at the top of the tree, and are broken down into several sub-tasks. The trees can be deep, with nodes called sub-trees which perform particular functions, which allows the developer to create libraries of behaviors that can be chained together to provide very convincing AI behavior.

This development can be used and reused, a major advantages over FSMs. Start by forming a basic behavior, then create new branches to deal with alternate methods of achieving goals, with branches ordered by priority, allowing for the AI to have fall back tactics should a particular behavior fail. This property and and re-usability have made BTs very popular in industry and are found in games such as *Halo 2* and *Spore*.

3-1-2 Semantics

BTs provide a hierarchical way of organizing behaviors represented as a rooted tree structure that is evaluate from left to right. The outgoing node is called the *parent* and the incoming node is the *child*. The BT is made up of several types of nodes, however all nodes share a core functionality. This core functionality is that they inform their parent node with their status. This return status is generally either *Success* of *Failure*. They inform their parent node that their operation was a Success or a Failure.

Basic BTs are made up of three kinds of nodes: Conditions, Actions and Composites (Champandard, 2007). The leaf nodes can be either Conditions or Actions whilst the branches of the BT consist of Composite nodes. Conditions test a property of the environment returning and the agent acts on its environment through Action nodes.

Composite A composite node determines how the BT is executed. It is a node that can have one or more children. They will process their children in either a Sequence or Selector way. At some stage will consider their processing complete and pass either Success or Failure to their parent, often determined by the success or failure of the child nodes. This thesis research only considers Sequences and Selectors although many others are used in practice.

Sequence The Sequence can be seen as an AND operator. A Sequence will visit each child in order, from left to right. If any child fails a Sequence will immediately return Failure to the parent. If the last child in the Sequence succeeds, then the Sequence will return Success to its parent.

It is important to make clear that the node types in BT have quite a wide range of applications. The most obvious usage of sequences is to define a Sequence of tasks that must be completed in entirety, and where Failure of one means further processing of that Sequence of tasks becomes redundant.

Selector Where a Sequence is an AND, a Selector is analogous with an OR statement. It will return Success if any of its children succeed and not process any further children. It will process the first child, and if it fails will process the second, and if that fails will process the third, until a Success is reached, at which point it will instantly return success. It will fail if all children fail. This means a Selector is analogous with an OR gate, and as a conditional statement can be used to check multiple conditions to see if any one of them is true.

A example BT highlighting the graphical representation of the different nodes can be seen in Figure 3-1.

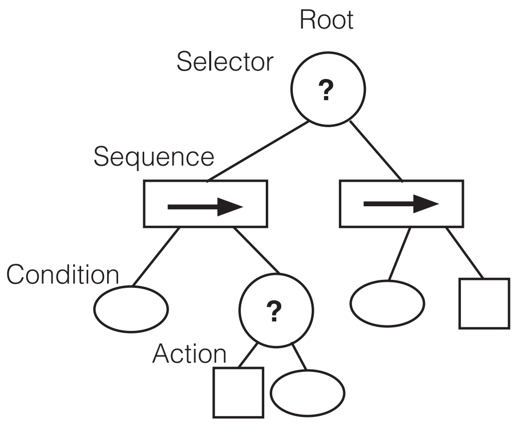


Figure 3-1: Graphical depiction of a Behavior Tree framework. The different node types are indicated.

Decorator A decorator node, a sub category of a composite node, can have a child node. Unlike a composite node, they can specifically only have a single child. Their function is either to transform the result they receive from their child node's status, to terminate the child, or repeat processing of the child, depending on the type of decorator node.

Leaf These are the lowest level node type, and are incapable of having any children. Leafs are the most powerful of node types, as these will be defined and implemented to perform specific tests or actions required to make the tree actually act in the world. An example of this would be a move action. A walk leaf node would make a system move to a specific point on the map, and return Success or Failure depending on the result.

The main power comes from their ability to choose from multiple different courses of action, in order of priority from most favorable to least favorable, and to return Success if it managed to succeed at any course of action. The implications can, for example in the gaming industry, be used to very quickly develop pretty sophisticated AI behaviors through the use of Selectors.

Leaf nodes are developed individually to perform specific tasks but can be reused in the tree. Composite nodes can also be reused in the current tree, but they are typically not platform dependent and can be reused in any BT. All the nodes in the BT share the same core functionality, so that combination of these nodes is possible without knowledge of any other part of the BT making BTs modular and reusable.

For this research *all* action nodes return Success when the node is evaluated.

3-1-3 Execution

The root node is typically a Selector node. Figure 3-2 shows a BT example adapted from (Millington & Funge, 2009), were the goal is to move into the room. This example illustrates the modularity of the BT, showing a character trying to move into a room in three different ways.

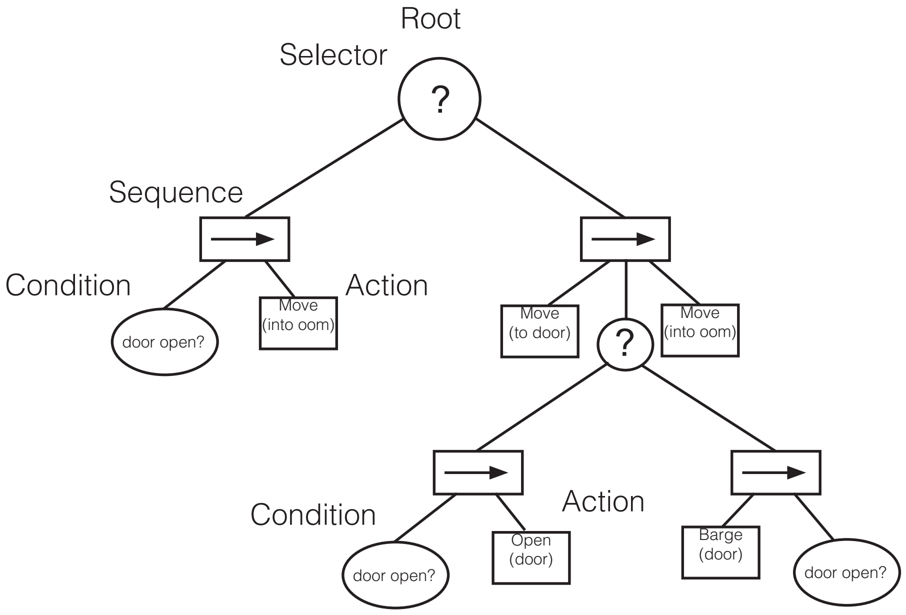


Figure 3-2: An example BT defined for behavior to open a door. Adapted from Millington & Funge (2009).

When the tree is activated, the root node tries its first child. The first child is a Sequence for moving through an open door. The Sequence starts with a condition which checks if the door is open. If the door is open, the character moves through the door and the Sequence returns Success to its parent. The top Selector receives Success from its first child, so no need to process the others and will return Success.

If, however, the door is closed, the first child of the Selector will fail for its condition task. The sequences is stopped immediately. The top Selector receives this Failure, but can select its second child to move to the door.

Its second sequences starts with a task of move to the door. Its second child is a Selector with two sequences. Both sequences, if successful, will result in opening the door. First it checks if the door is unlocked and if so opens the door. The sequences will continue to its third child and will move into the room.

If the door cannot be opened, the character will try to barge the door. If this this fails, the Sequence will return fail to the Selector and the character has no more ways to try to open the door. It will return a Failure to its parent Sequence, and this Sequence then also fails. This will fail the move into room behavior. This shows the modularity of the BT, since at this point more room-entering behavior could be implemented to the existing tree. Adding checks to search for windows to smash through for example.

3-1-4 Advantages

The strength of a BT comes from their ability to create complex tasks composed of simple tasks, without worrying how the simple tasks are implemented.

The BT evaluates its branches defined by the structure of the tree. It can simply integrate multiple methods to achieve the same goal. If one branch with behavior is not successful, it moves to the next branch, different behavior but yielding the same outcome. This results in goal based behavior and contingency planning.

This is different compared to a FSM. A FSM maps sets of state-actions. When certain conditions are met, the object changes to another state. The designer has to link all the states of the agent. A disadvantage of FSMs is that this representation of all state-action sets can become complex for complex behavior. Large FSM with many states and transitions can be difficult to manage and maintain, resulting in state-explosion.

To reduce this state-explosion, hierarchy was introduced. This reduces the representational complexity of the FSM, but it is not a very reusable framework. It still requires *explicit linking* from one state to another. This implies that reusing the same particular states in another area of the behavior still need to be explicitly described. The advantage of a BT is that it describes its behavior in the structure of the tree instead of state transitions. This makes the BT modular and reusable resulting in a scalable design framework.

A BT does not require a model of the environment. Because the framework does not consider the effects of actions, but only observers the current state. This model free approach is simple and relatively computationally efficient to evaluate. It also restrains state explosion, a phenomenon which affects FSM. This makes small robotic platforms, where computational power is low, a well suited platform for using BTs.

(Nolfi, 2002) work shows that agents with a form of reactive architecture can exhibit complex behaviors without requiring any internal state and demonstrates that this is due to their ability to coordinate perception and action. This type of planning has very low computational requirements and can be run very quickly to operate in highly dynamic environments.

And finally, the ease of human understanding and readability make BTs less error prone and very popular in , i.a the game developer community.

3-1-5 Limitations

BTs become reasonably clunky when representing the kind of task that need to respond to external event-interrupting conditions. When behaviors need to react to changing conditions and has a behavior switch strategies, trees become large. Notice that this can still be implemented with BTs, but its size increases (Millington & Funge, 2009).

BTs work great if a system needs to switch to between different tasks and corresponding controllers, based on Success or Failure of certain actions or conditions. However, BTs make it more difficult to think and design in terms of states since they usually represent a hierarchical structure.

Discrete events are not explicitly implemented in the structural framework of a BT as is the case with an event driven FSM. This problem can be addressed by include memory, to store events for the BT to handle some tick later. Or by building a hybrid system with multiple BTs and state machines to determine which BT is running.

3-2 Recent Advances

Ögren (2012) was the first to argue that the modularity, re-usability and complexity of UAVs guidance and control systems might be improved by using a BT architecture. He states that this is mainly due to the fact that BTs make the transitions implicit in the tree structure. The implicit transitions substantially increase modularity, which in turn makes design and re-design much simpler. Section 3-2 shows the proposed BT for combat behavior.

This approach is different than standard currently used in the aerospace industry where FSM are commonly used to describe UAV behavior. Some of the most popular UAV software packages such as Paparazzi and ArduPilot both use forms of FSM to implement their mission management Scheper et al. (2016).

Colledanchise & Ogren (2014) showed how these key properties can be traced back to the ideas of subsumption and sequential compositions of robot behaviors. They have provided a theoretical description of how properties such as robustness and safety are preserved in modular compositions of BTs.

Klöckner (2013) built on the work of Ögren (2012) and describes in his paper the advantages of BTs for use in mission management. As compared to the state-of-the-art solution using finite state machines, BTs offer increased scalability due to their tree structure and their simple, standardized interface. The paper points out research towards their deployment to on-line use.

Klöckner (2013) presented a mission management system based on BTs. It uses the description logic variant ALC(D) as interface to the world. An implementation of the system is

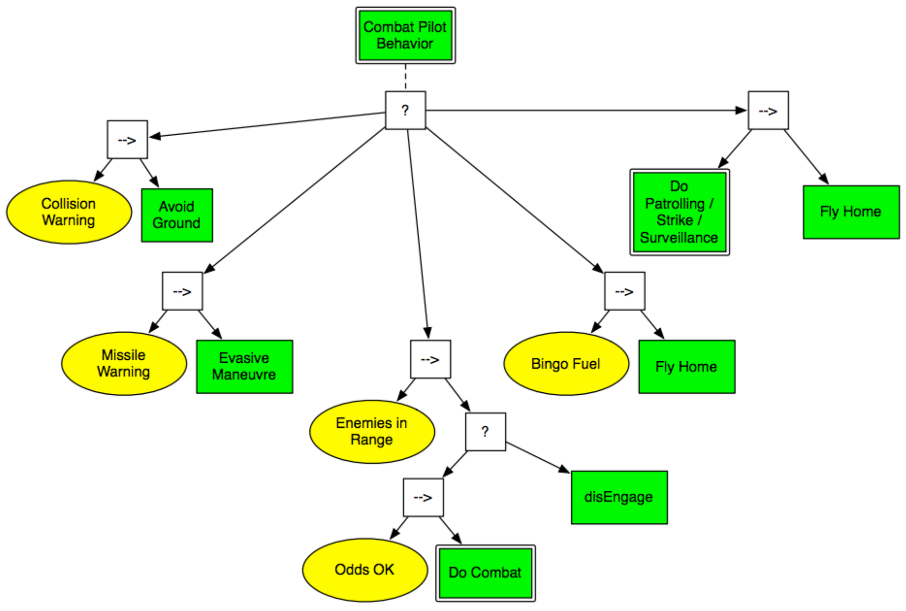


Figure 3-3: BT defined for combat behavior for an UAV. Adapted from (Ögren, 2012).

shown with a test case. The test case used in this section employs the safety layer of an UAV. The UAV is meant to return to its base, if the battery state-of-charge reaches a critical level. This condition will be queried through an abstracted logical interface.

3-2-1 Optimizing BT

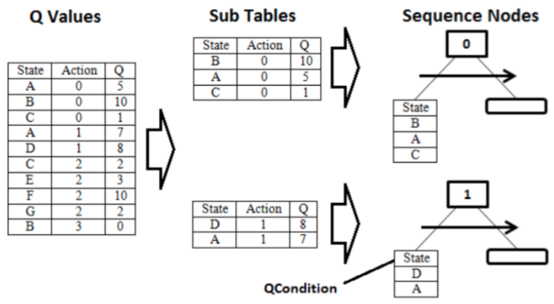
Recently, there have been works to improve BT design with the use of several learning techniques, for example, Q-learning (Dey & Child, 2013) and Evolutionary Learning (EL) approaches (Scheper et al., 2016).

(Dey & Child, 2013) applied off policy Q-learning to BTs. They performed their research for a predator-prey game scenario and showed that a tree resulted from Q-learning Behavior Tree (QL-BT) performs on a par with the original BT or outperforms it in all areas. Create a Q-condition node that assists them in identify the most appropriate moment to execute each branch of AI logic.

Their method is an algorithm that begins with a BT as input. They analyze the tree to find the deepest Sequence nodes. These nodes are then identified as actions for the RL approach. These actions are used in an off-line Q-learning phase to generate a Q-value table.

The table is then divided into sub-tables by action and the highest valued states for the action are extracted into the, what they call, Q-Condition nodes within the BT. The condition nodes in the input BT are then replaced with these Q-Condition nodes. Finally, the BTs topology is reorganized each node child is sorted by their maximum Q-value, which provides AI designers with a more optimized permutation of the BT. The Q-condition node is shown in Figure 3-4.

This representation does however not use the hierarchy architecture advantage and readability the BT framework has. The BT suggested by (Dey & Child, 2013) still requires a lot of insight in the decision process to fully understand under which conditions actions are executed.



State	Action	Q
A	0	5
B	0	10
C	0	1
A	1	7
D	1	8
C	2	2
E	2	3
F	2	10
G	2	2
B	3	0

State	Action	Q
B	0	10
A	0	5
C	0	1

State	Action	Q
D	1	8
A	1	7

State
B
A
C

State
D
A

Figure 3-4: Q-condition node. Adapted from (Dey & Child, 2013).

Insight in the decision process is particularly an advantage of the BT framework that is not used in their work.

An other drawback of the QL-BT algorithm is its reliance on correct Q-values. The validity of the Q-values relies on an appropriate reward function and an effective learning phase to provide accurate utility estimates for each state-action pair. Values can be improved by a longer training period because it is a preprocessing step. The Q-condition node does not provide the designer with easy verification possibility to check the Q-value. Even though the above-mentioned works motivates our work, the missed opportunity to use the BT framework to its full potential causes us to move in an other direction for combining RL and BTs.

de Pontes Pereira & EngeMartinsl (2015) proposed a framework to use RL in behavior-based agents, which provides adaptiveness to physical or virtual agents while still respect the constraints modeled by the expert. Based on BTs, they proposed the creation of a new type of Composite and Action node, called Learning Node, in which we embed a Q-Learning algorithm to perform a local learning, without effect how other nodes work.

Scheper et al. (2016) applied EL to BTs for a real-world robotic, to an MAV, application. They showed the first real-world robotic application of evolved BTs. An initial population of BTs is created and genetic operators are used to evolve a BT. This genetically optimized BT performed better than human designed BTs. In addition, their BT provide the possibility to tuned manually when the controller transferred to the real world robotic platform. They conclude that the increased intelligibility of the BT framework does give a designer increased understanding of the automatically developed behavior and a possibility to adjust this behavior.

However, evolutionary computing is computationally intensive due to the large number of simulations required to evaluate the performance of the population of individuals. When the performance of the genetically evolved BTs is analyzed, every BT is run in simulation. The basic disadvantage of GA is its unguided mutation. The mutation operator in GA functions like add a randomly generated number to a parameter of an individual of the population.

Perez et al. (2011) have also applied evolutionary computing to BTs with the use of a genetic approach based on grammatical evolution. This work was applied to simulated games where the game world is inherently discrete and certain.

Chapter 4

Preliminary research

During the preliminary analysis RL agent is designed to choose which RL algorithm to use, either Q-learning or SARSA. Then the policy for one of the RL algorithms is manually approximated with a BT. This BT is evaluated on the ability to verify its behavior and manual adaption capabilities.

4-1 Windy Grid World

A RL agent with a guidance task in a discrete windy grid world is implemented. The RL algorithms in this simulation are Q-learning and SARSA. Both methods have small difference as discussed in Section 2-1-5 and this small example will help chose one of the algorithms to be used in the rest of this research.

Figure 4-1 shows a standard grid world, with start state and a goal state. In addition to the start and goal state, a crosswind that blows north through the middle of the grid is present. The agent has to move from the start state to the goal state in the least amount of steps. The agents actions are standard: up, down, right, and left.

In wind states, the resultant next states after an action is shifted upward by the 'wind'. The upward shift is depended on the strength of the wind which varies from column to column. The strength of the wind is given below each column, in number of cells shifted upward. For example, if you are one cell to the right of the goal, then the action left takes you to the cell just above the goal.

The state the agent observers is its x- and y-position and learns to achieve its goal environment using the SARSA algorithm in Equation (4-2) and the Q-learning algorithm in Equation (4-1).

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{4-1}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{4-2}$$

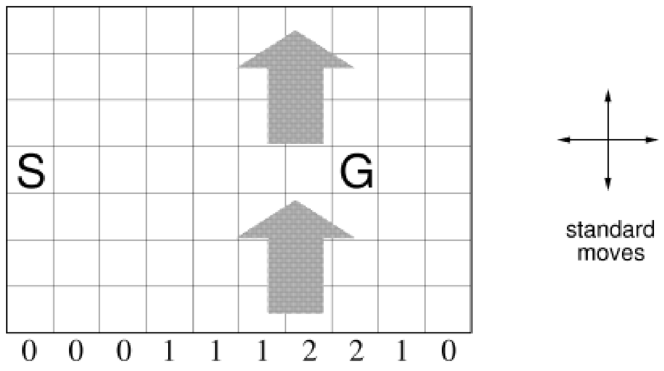


Figure 4-1: Windy grid world. Adapted from R. Sutton & Barto (1998)

The parameters for the learning algorithms are presented in Table 4-1. For the trade-off between explorations and exploitation an $\epsilon - greedy$ policy is implemented, a variation on the normal greedy selection. In both cases the agent identifies the best action based on the state-action value. But $\epsilon - greedy$ selects the greedy action, the action with the highest state-action value, 1 - ϵ, of the time, where $\epsilon \in [0, 1)$. The policy selects one of the non-greedy actions a fraction ϵ of the time.

Both algorithms use a decay ϵ-greedy policy. This policy starts with an $\epsilon = 0.5$, and linearly reduces every episode until 0 for the last episode.

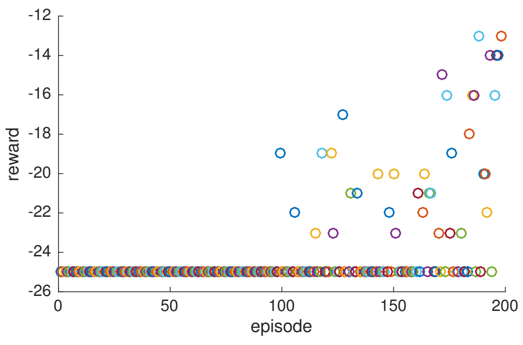
Table 4-1: Parameter settings for the RL controller.

Parameter	Value Q-learning	Value SARSA
Number of episodes	200	200
Max number of steps per episodes	25	25
Learning rate α	0.3	0.3
Discount factor γ	0.8	0.9
Exploration factor ϵ_0	0.5	0.5
Reward move (up,down,left,right)	-1	-1
Reward hit border of grid wold	0	0
Reward goal state	+10	+10
Converged at episode	198	269

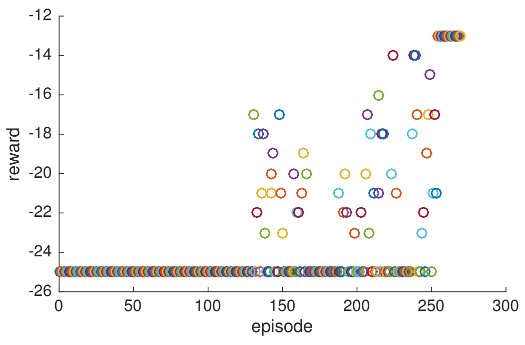
The results of the agent trying to maximize its reward for both algorithms are presented in Figure 4-2.

For both algorithms the agent explores the state-space during its first trails. Due to the high epsilon, the agent selects more often an exploration action compared to an exploitation action. To cut of unnecessary long episodes when the agent does not find the goal state, the episode is terminated when a reward of -25 is collected. The Q-learning algorithm finds the goal first, around episode 100. From that moments, the algorithm learn and move to the goal more often. Resulting in a maximum reward of -13, the highest possible reward for each episode for this task.

Both algorithms are considered converged if δ-Q is less than 1×10^{-8}. Since very small updates indicate that the algorithms do not learn anymore. The progression of this learning

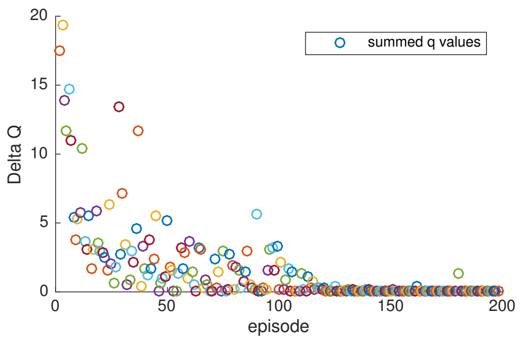


(a) Collected Reward per Episode for Q-learning Algorithm

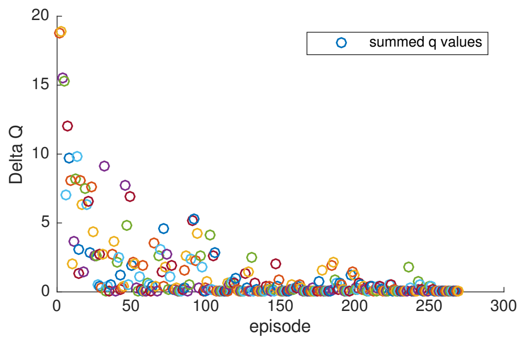


(b) Collected Reward per Episode for SARSA Algorithm

Figure 4-2: Reward per episode for RL controller using Q-learning or SARSA.



(a) Summed Q-value of Q-learning



(b) Summed Q-value of SARSA

Figure 4-3: Delta Q per episode.

for the Q-learning algorithm and SARSA algorithm are shown in Figure 4-3a and Figure 4-3b respectively.

RL-selection

The test case of the windy-grid world does not provide clear results to select either algorithms. The Q-learning results indicate a faster convergence for the windy grid problem.

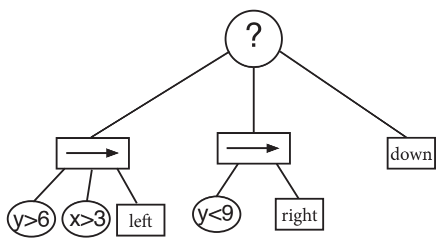
However, research on RL and BT is been applied to computer game environments (Dey & Child, 2013) where the state is fully known and have deterministic outcomes. Classical RL approaches often consider a grid-based representation with discrete states and actions, often referred to as a grid-world (Kober et al., 2013). And, as discussed in Section 2-1-5, it is the most basic learning algorithm (Yoshikawa et al., 2008). When the state space is sufficiently explored, Q-learning has proven to always converge to the optimal policy (Watkins & Dayan, 1992).

Based on previous work, the Q-learning algorithm is selected to be used for further research during this thesis. Since the focus of this thesis work moved towards approximating a developed optimum RL policy with a BT, HRL is not considered. The learned behavior to be approximated with a BT should have no constraints imposed on in advance. BTs already provide a hierarchical representation and its design should not already be limited to a predefined HRL structure.

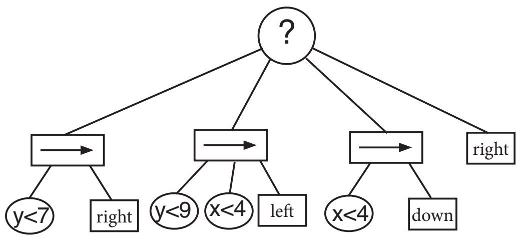
4-2 Policy Representation using BTs

When the Q-learning RL controller converged, two BTs are designed manually based on the state-action values in the Q-table. The designed BTs are presented in Figure 4-4a and Figure 4-4b. Figure 4-4a represent the policy using 3 branches and 9 nodes and Figure 4-4b uses 4 branches and 12 nodes.

The elliptical nodes represent the condition nodes in the tree. These condition nodes contain a variable, *x or y*, that the node checks with a limit value. The action nodes are the rectangle nodes. These nodes contain the action that is executed by the node, either *up, down, left or right*.



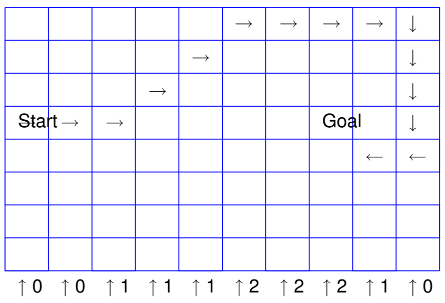
(a) BT 1 containing 9 nodes



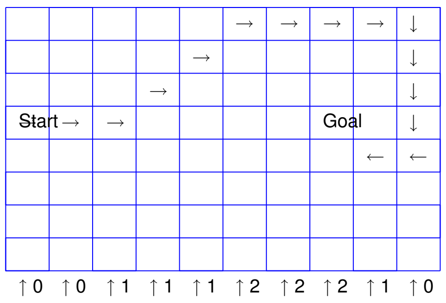
(b) BT 2 containing 12 nodes

Figure 4-4: Manual BT designed based on state-action values.

The agent that uses the either one of the BTs follows the same optimal path in the grid world as the RL policy did as can be seen in Figure 4-5.



(a) Path traveled for BT 1



(b) Path traveled for BT 2

Figure 4-5: Resulted path for BT designs. Both BT identify the same actions for all states in the grid world. This results in the same path through the grid world for both trees.

For this grid-world problem, the Q-learning policy is stored in a Q-table containing *320 state-action pairs*. These are contained in a Q-table. To adjust the action selection for the Q-table involves change these state-action pairs. This lead to practical issues to be able to see the resulted behavior.

Both BTs presented in Figure 4-4 identify the same action for all states in the grid world with only 9 or 12 nodes. As BTs are meant to increase the user understanding of the solution strategy, for BT with less nodes less information needs to be considered. Although both BTs result in the same action identification, a lower amount of nodes is considered better.

The manually design BT presents the same action identification is a framework containing 9 nodes in Figure 4-4a. To manually adjust this tree is as simple by changing the nodes. The tree can also easily be extended, resulting for example in Figure 4-4b. The BT also allows to extend its behavior for implementation in other environments.

Although comparing the size of a Q-table with the number of nodes in a BT is difficult, presenting a RL policy with a BT indicates the have promising advantages. If the the number of nodes can be compared one-to-one with state-action pairs, a reduction in size of a factor of 35 is achieved.

The manually designed BT also allows for easy verification of the developed behavior, verifying 9 nodes compared to 320 state-action pairs.

For further research a more challenging RL task will be set up. The evolved RL policy will be approximated with a BT. To manually design these BT will lead to the same practical issues as verifying the Q-table. So further research will investigate the possibilities to automatically generate a BT that approximates the developed RL policy.

Part III

Additional Results

Chapter 5

Additional Results

To automatically generate BT to approximate a developed RL policy, GAs are used. The GA optimize for a specific fitness function. The formulation of the fitness function used in Part I is the result of previous experiences of multiple fitness functions.

5-1 Fitness Functions

This section presents multiple fitness functions that were used for the approximation of the RL policy. Although these fitness functions did not evolve the most successful BT during this research, which is presented in the scientific paper, it did provide extra understanding of the RL policy structure and how the DTMC is used in the optimization.

5-1-1 Accumulated State-Action Value Fitness Function

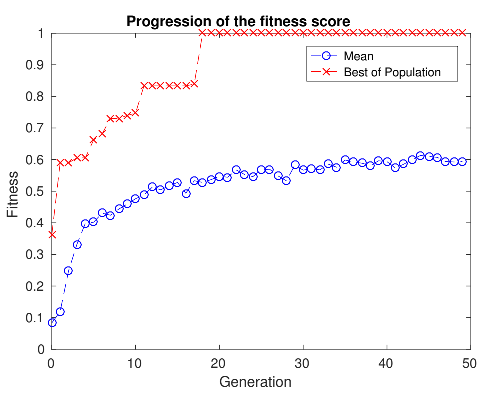
A RL agent's objective is to maximize the total reward each episode, the greedy policy does this by select the action with the highest state-action value. For this reason the formulation of the first fitness function for the GA is to evaluate BTs based on the the accumulated state-action value for a walk through the DTMC.

The fitness function is implemented according to Algorithm 1. To observe the progress of the optimization, first a run is performed for 50 generations.

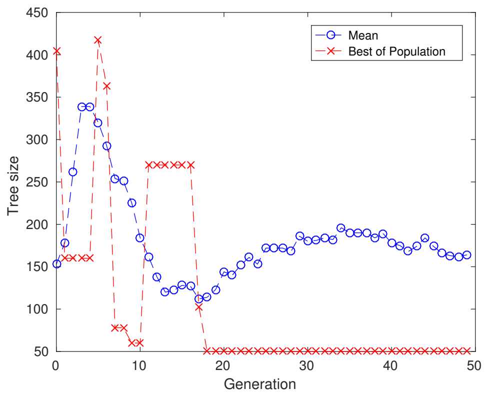
```
  Fitness = Fitness + Q-table(s,a)
```

Algorithm 1: Fitness function used for Genetic Optimization. The fitness is the accumulated Q-score during the run through the DTMC.

50 Generations
Every tree in the population is run 5 times through the DTMC and mean score of the individual is used for the evaluation of the tree. The results of the optimization using GAs are shown in Figure 5-1. The normalized fitness score of the best individual and the mean of the population throughout the genetic optimization is shown in Table 5-1. This optimization did not include a second fitness function to reduce the size of the tree. This resulted in a population with more nodes compared to the results presented in Part I. Progression of the number of nodes of the best individual of the population and the mean of the population throughout the genetic optimization is shown in Figure 5-1b.



(a) Progression of the fitness score of the best individual and the mean of the population throughout the genetic optimization using the accumulative Q-score for fitness for a run with 50 generations



(b) Progression of tree size of the best individual and the mean of the population throughout the genetic optimization using the accumulative Q-score for fitness for a run with 50 generations

Figure 5-1: Results Genetic Optimization for 50 generations.

This fitness function shows good initial results for 50 generations. The results of the evolved BT is shown in Table 5-1. Compared to the RL policy, the optimized tree accumulates a higher state-action score compared to a walk through the DTMC following the RL policy.

The evolved BTs are evaluated with the same metric used in the scientific paper with the score of the fitness function. The action identification of the tree is compared to the action identification of the RL policy for all states in the Q-table. Then the tree is run in the simulation environment and the actions identified of the states that are encountered are compared to the RL policy.

Although the reward of each episode for the RL controller fluctuates, the reward collected by the evolved BT is presented. However, this score does not directly represent the success of the approximation as discussed in Part I. Finally the representation of the Q-table is compared with the tree.

When the evolved BT is run in the simulation environment for 150 steps, wall avoidance behavior is observed. This wall avoiding behavior is shown in a motion example in Figure 5-2.

The result of the optimization indicate that the optimization might need more generations to evolve a BT with the other subtasks of the RL policy.

Table 5-1: Results optimized BT with GA for 50 generations using the accumulated Q-values as fitness function.

State	Q-learning	Optimized BT with Q-values
Accumulated Q-score	3237	6324
Action-selection for all states	50%	89%
Action-selection for simulation	100 %	86%
Average reward	283	48
Representation	3123 entries	91 nodes

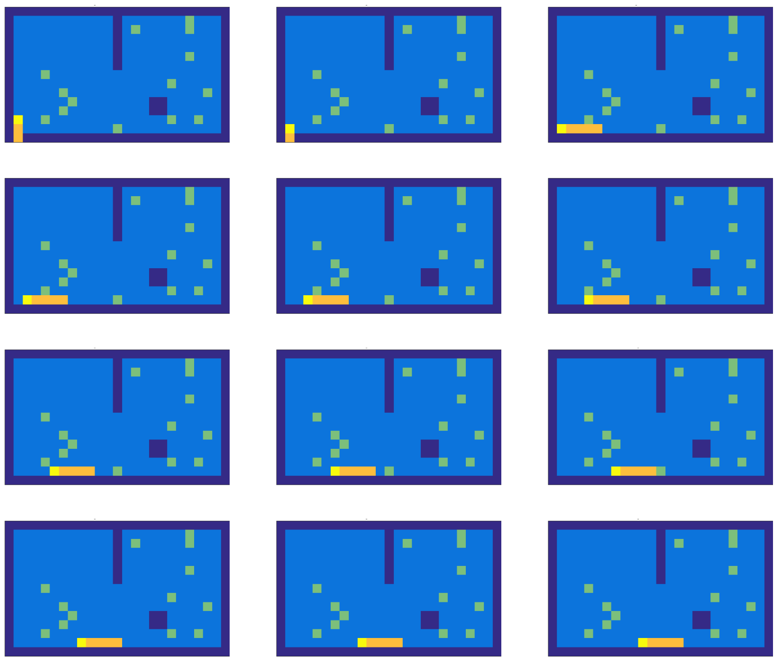
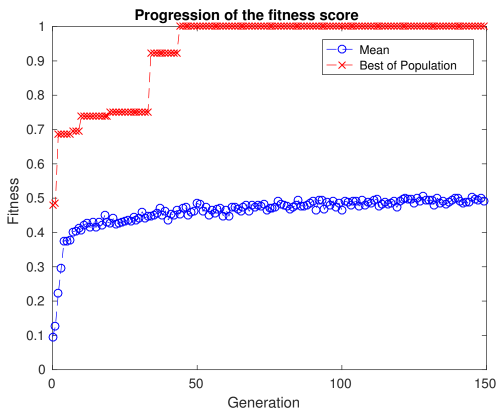


Figure 5-2: A motion example of the UAV using the BT that is the evolved for 50 generations. This BT avoids walls and not reacts to trash. Dark blue indicates walls, light blue indicates free space, green indicates trash, yellow indicates the UAV's position.

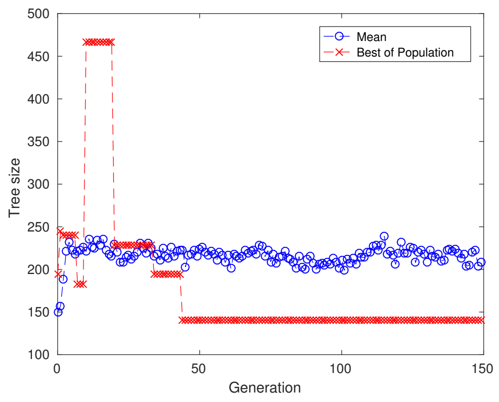
150 Generations
So, the optimization is run for 150 generations using the same fitness function, as presented in Algorithm 1. The approximation results with 150 generations are shown in Figure 5-3. Again the normalized fitness score of the best individual and the mean of the population throughout the genetic optimization is shown in Table 5-2. And the progression of the number of nodes of the best individual of the population and the mean of the population throughout the genetic optimization is shown in Figure 5-1b.

An overview of the result of the optimization of the evolved BT for 150 generations is shown in Table 5-1. Notice the high accumulated state-action value for the optimized tree, more than *tree times* more than the RL policy collects for a run through the DTMC.

When the evolved BT is run in the simulation environment, unexpected behavior is observed. A motion example shows this behavior and is presented in Figure 5-4. It shows the UAV turns to observe trash and then selects the action to turn again. This behavior ensures the



(a) Progression of the fitness score of the best individual and the mean of the population throughout the genetic optimization using the accumulative Q-score for fitness for a run with 150 generations



(b) Progression of tree size of the best individual and the mean of the population throughout the genetic optimization using the accumulative Q-score for fitness for a run with 150 generations

Figure 5-3: Results Genetic Optimization for 150 generations.

Table 5-2: Results optimized BT with GA for 150 generations using the accumulated Q-values as fitness function.

State	Q-learning	Optimized BT with Q-values
Accumulated Q-score	3237	10124
Action-selection for all states	100 %	52 %
Action-selection for simulation	100 %	50%
Average reward	298	-150
Representation	3123 entries	90 nodes

UAV to keep sensing a piece of trash on one of its sensors. The UAV does not move forward to collect it.

This results however, leads to a very low reward. Although the number of successful action identifications compared to the RL policy is 50%, the reward is obviously not 50% of the reward collected by the RL policy in a simulation run.

These results provides insight in how the DTMC relates to the simulation environment. The lack of actual reward as is provides by the simulation environment is missing in the DTMC compared to the simulation.

In the DTMC, when noting is observed, transition to a state where trash is senses have low probability. If trash is sensed in front and the UAV collects it, the UAV moves from a state with high state-action values to a state with low the state-action values, observing nothing. The state-action values of observing nothing, observing trash in front and observing trans on its right are presented in Table 5-3.

The states in Table 5-3 indicate the sensed variable on *sensor0, sensor1* , *sensor2* , *sensor3* , *sensor4* , *sensor5* , *sensor6* , *sensor7*. Where the observations are *wall (0), free of obstacles (1), trash(3)*. The state representation of the UAV is presented in Figure 5-5.

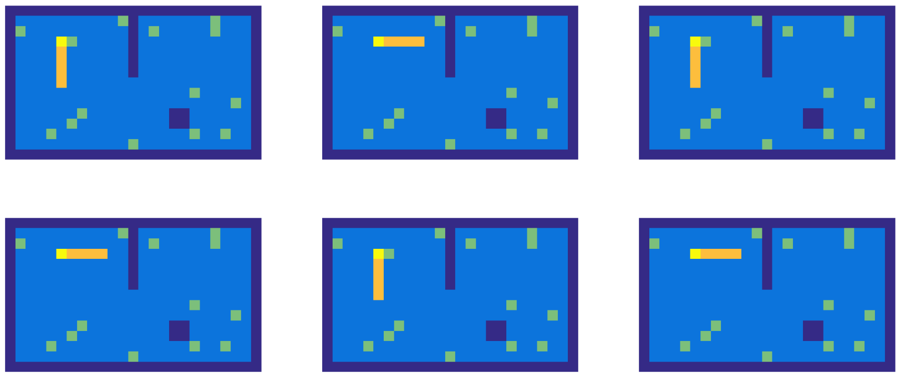


Figure 5-4: A motion example of the UAV that observes trash instead of collect it. Only front facing sensors are indicated. Read from left to right, top to bottom.

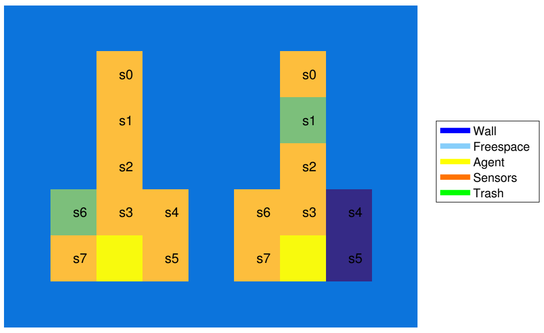


Figure 5-5: Two UAV sensor configurations. Left: A UAV observes trash in *s6*. Right: A UAV observes trash at *s1* and observes a wall on its right on *s4* and *s5*.

To turn and still sense trash on its sensors, the UAV is most likely to transition too observe trans on its right sensor, state (1,1,1,1,1,3,1,1). This state has higher state-action values compared to the state (1,1,1,1,1,1,1,1). Using this fitness function results an evolved tree that observes trash instead of collect it.

Table 5-3: State-action pairs for 3 states with 3 actions.

State	Move	Turn Left	Turn Right
(1,1,1,1,1,1,1,1)	7.6659	0.728183	0.580431
(1,1,1,3,1,1,1,1)	14.301	4.88565	6.2666
(1,1,1,1,1,3,1,1)	1.28739	4.72524	20.7678

A transition from state (1,1,1,3,1,1,1,1), observing trash in front, and move forward is investigated to see how it effects the optimization behavior. The transition probabilities from moving forward from state (1,1,1,3,1,1,1,1) to the possible other states for the UAV are presented in Table 5-4.

When the UAV takes the action to collect trash, the UAV most likely transitions to a state where it observes only free space (1,1,1,1,1,1,1,1). For this state, the state with the highest state-action value is moving forward. Moving forward from state (1,1,1,1,1,1,1,1) has a high probability to end up in the same state, 78%.

Subsequently, if the tree BT selects the action to collect trash, it leads to transition to a state with low state-action value and has a high probability remain in that state.

Table 5-4: Transition probabilities for moving forward when observing trash in front.

State	Transition	Probability
s0,s1,s2,s3,s4,s5,s6,s7	-	-
0 ,0 ,0 ,1 ,1 ,1 ,1 ,1	298	0.7 %
1 ,1 ,1 ,1 ,1 ,1 ,1 ,1	32683	78.0 %
1 ,1 ,1 ,1 ,3 ,1 ,1 ,1	527	1.0 %
0 ,1 ,1 ,1 ,1 ,1 ,1 ,1	6258	15 %
1 ,1 ,1 ,1 ,1 ,1 ,3 ,1	1296	3.0 %
1 ,1 ,1 ,1 ,0 ,1 ,1 ,1	299	0.7 %
0 ,1 ,1 ,1 ,1 ,1 ,3 ,1	31	0.0 %
1 ,1 ,1 ,1 ,1 ,1 ,0 ,1	235	0.6 %
0 ,1 ,1 ,1 ,3 ,1 ,1 ,1	37	0.0 %

In addition, the results in Table 5-2 indicate that successful action identification of 50% compared to the RL policy does not lead to a reward of 50% of the RL policy. It even resulted in a negative reward. This indicates that some information stored in the Q-table is more important than other information.

The accumulated state-action value does not capture which information is important in the fitness function. This indicates that approximating the RL policy is more complex than initially was assumed. However, these results provided insight into a DTMC and how it relates to the simulation environment. It led to change the fitness function. A new fitness function is set up that uses the actual action identification in the next section.

5-1-2 Problem Specific Fitness Function

The BT must identify the same action as the action identified by the RL policy. The new fitness function compares the action that is identified by the tree and directly compares it to the action that the RL policy identifies.

Using the knowledge from Section 5-1-1, problem specific information is added to the fitness function. To steer the optimization to successful action identification in important states, the fitness function is tailored for these important states.

Important states are the states that involves trash or a wall. Higher fitness is given to the tree that identifies the correct action in states where trash is observed. If trash is observed on *s3*, *s4* or *s5* and the the tree identifies the correct action, its fitness increases with 8, compared to a decrease with 4 of the identification is unsuccessful.

The fitness of the BT gets extra penalized if the sensor *s3* observes a wall and the wrong action is identified. This action is most likely to cause a collision. For this reason the fitness is increase with 1, and decreased with 2. This results in the fitness function formulation shown in Algorithm 2. The fitness function steers the GAs to converge on a population that selects the right action when trash or a wall is observed.

```
if $s_3 = 3$ or $s_4 = 3$ or $s_5 = 3$ then
  if $a_{bt} = a_{rl}$ then
    Fitness = Fitness + 8;
  else
    Fitness = Fitness - 4;
  end if
else if $s_0 = 0$ then
  if $a_{bt} = a_{rl}$ then
    Fitness = Fitness + 1;
  else
    Fitness = Fitness - 2;
  end if
else
  if $a_{bt} = a_{rl}$ then
    Fitness = Fitness + 1;
  end if
end if
```

Algorithm 2: Fitness function used for Genetic Optimization. Increase/decrease fitness for the BT when trash is observed at s_3, s_4 or s_3. Worse fitness when s_3 observers a wall and the wrong action is selected.

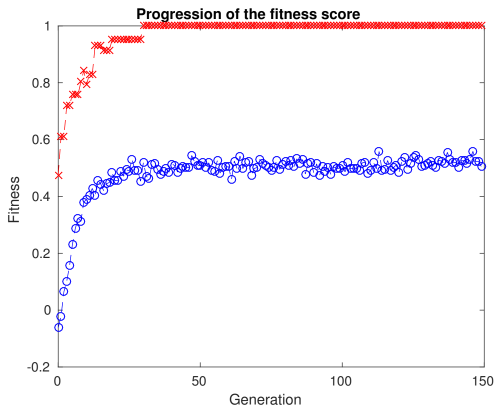
When the this fitness increases above the 90% of the maximum reached fitness score of a previous population, an additional fitness function is added. This fitness function evaluates the BTs size. A BT with less nodes is easier to interpret. This results that a tree with less nodes is scored with high fitness, and vice versa. Maximum fitness is scored for trees with 20 nodes or less. This leads to an optimization to reduce the size of the tree too a number of 20 nodes.

The parameter values for the GA that are used for the optimization are presented in Table 5-5.

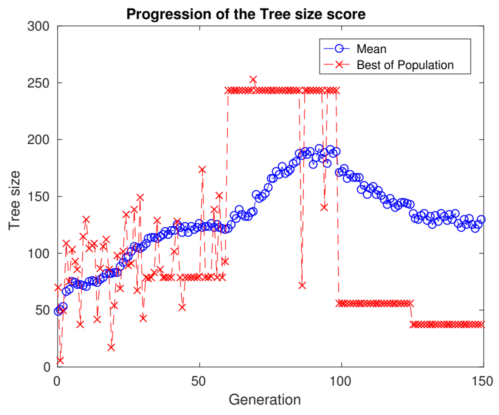
Table 5-5: Parameter values for the optimization using Genetic Algorithms.

Parameter	Value
No. of steps through DTMC	100
Max number of generation (G)	150
Population size (M)	100
Tournament selection size (s)	8%
Crossover rate (P_c)	50%
Mutation rate (P_m)	30%
Headless-Chick Crossover rate (P$_{hcc}$)	10%
Maximum tree depth (D_d)	3
Maximum no. of children (D_c)	4
No. of simulation runs per generation (k)	5

Figure 5-6a show the population mean fitness and the mean fitness of the best individual in each generation. The mean fitness improves gradually and then settles at around the 0.5 of the normalize fitness axis. This implies that the genetic diversity in the pool is sufficient.



(a) Progression of the fitness score of the best individual and the mean of the population throughout the genetic optimization. The highest fitness of the best scoring individual is evolved after 145 generations.



(b) Progression of the number of nodes of the best individual of the population and the mean of the population throughout the genetic optimization.

Figure 5-6: Genetic Optimization results.

Figure 5-6b shows the tree size of the best individual and the tree size of the population mean. Figure 5-6b shows that the average tree size began at about 75 nodes and then slowly increases until 200 nodes and from there continues to reduce in size and eventually drops below 150. The best individual BT after 150 generations had 38 nodes. By removing the nodes that have no effect on the final behavior, called *pruning*, the final BT resulted in a BT with 17 nodes.

The best evolved BT of the population after 150 generations and after pruning is presented in Figure 5-7. The condition statements are indicated with the sensor s_i and the check [wall (0), free (1), trash (3)]. And the action are indicated with L for turn left, R for turn right and M for move forward.

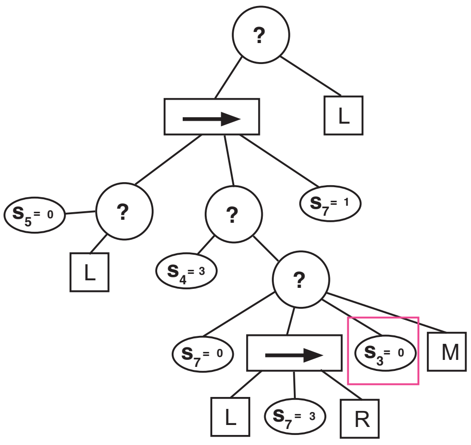


Figure 5-7: Graphical depiction of genetically optimized BT.

The results of the evolved BT are shown in Table 5-6. The optimized tree is evaluated on three subtask: `collision avoidance`, `collect trash` and `search trash`.

Table 5-6: Results optimized BT with GA for 150 generations using problem specific fitness function.

State	Q-learning	Optimized BT
Action-selection for all states	100 %	34 %
Action-selection for simulation	100 %	93%
Average reward	298	159
Representation	3123 entries	17 nodes

Collision avoidance
The RL algorithm learned to avoid collisions by turning left. This action results in counter clockwise traveling through the room. This characteristic behavior is also demonstrated when the UAV uses the evolved BT.

The UAV with the evolved BT as action identification method is shown in Figure 5-8. The agent observes the wall in front, no pieces of trash and selects to turn to the left. After which the agent continues to follow the wall.

The collision avoidance behavior can now be verified using Figure 5-7. The red box indicating the check for a wall in front, together with the rules of the BT framework show that the agent will never move forward. The 17 nodes of the BT make this possible. This compared to identifying all wall states in the Q-table of the RL policy and then check the state-action values.

Collect trash
When the UAV observes trash on its left and then collect the trash by moving forward. A

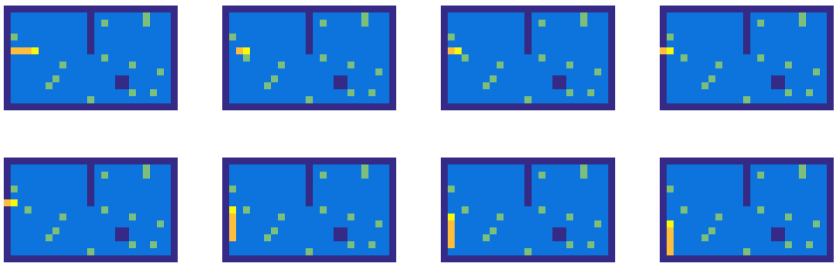


Figure 5-8: A motion example of UAV with optimized BT as action selection policy. Agent shows goal driven behavior when trash is observed. Dark blue indicate walls, light blue indicate free space, green indicate trash, yellow indicate the UAV's position. Read from left to right, top to bottom.

motion example that demonstrates collect trash behavior is shown in Figure 5-12. It shows the UAV that follows the wall and decides to turn when trash is observed.

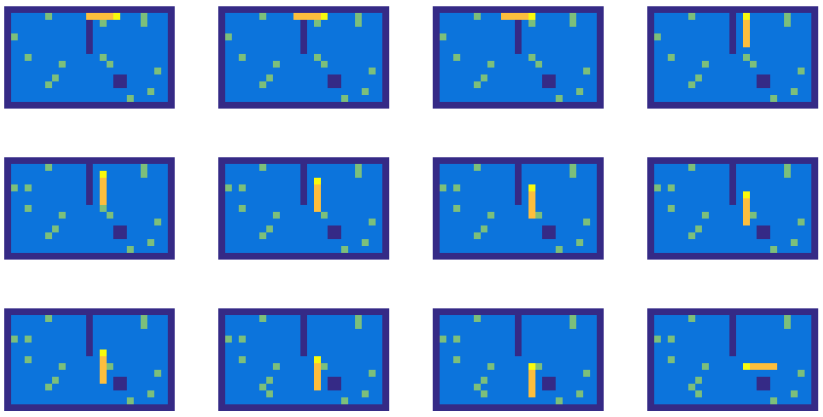


Figure 5-9: A motion example of UAV with the optimized BT as action selection policy. Agent shows goal driven behavior when trash is observed. Dark blue indicate walls, light blue indicate free space, green indicate trash, yellow indicates the UAV's position. Read from left to right, top to bottom.

It is verified using Figure 5-7 that the UAV will not collect trash on its right.

Search trash
Search trash behavior is not developed by the RL controller. The RL controller did learn actions to increase its chance to encounter trash. This behavior to increase the chance of encounter trash is not evolved in the BT.

The evolved tree captured *one* important subtask, collision avoidance. The evolved tree represents this subtask using 4 nodes which are easy to verify. Verify 4 nodes is less work compared to have to adjust all state-action values for states involving an object the Q-table.

5-1-3 Evaluation Fitness Function

Three subtasks that are identified for the RL policy that are not present in the evolved BT: collect trash on right, collect trash in front first and increase the chance to encounter trash. However, an advantage of the BT framework is the ease to make adjustments to the tree and add behaviors so these behaviors could be added.

Although already a partial success is achieved, this fitness function limits wider use of this method. If the same results can be obtained with a generic fitness function for the optimization, the method would increase its applicability to other problems. This led to the development and results presented in Part I.

5-2 Acting policy in the DTMC

The actions identified by the BT during the optimization were used to transition through the DTMC. An other option is to select the actions that the RL policy identified. To investigate the difference in optimization, a preliminary analysis is performed. One of the recommendation presented in Part I is to investigate the influence of selecting the *acting policy* in the DTMC during optimization. The results, with the RL policy as acting policy in the DTMC, present a direction for future research.

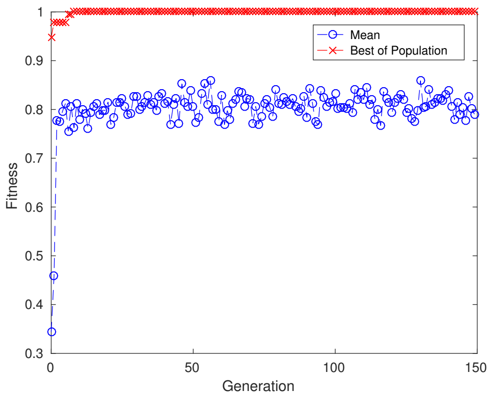
When the BT is selected as active policy in the DTMC, the state that are evaluated are the states that are more likely to be actually encountered by the tree in simulation. However, for wrong action identification, the tree moves to an other state compared to the RL policy and that state is less likely to be visited by the RL policy. It is interesting to see that the optimization has difficulty to evolve a BT that captures the characteristics of a RL controller.

This chapter present a preliminary analysis on selecting the RL policy as active policy to transition through the DTMC.

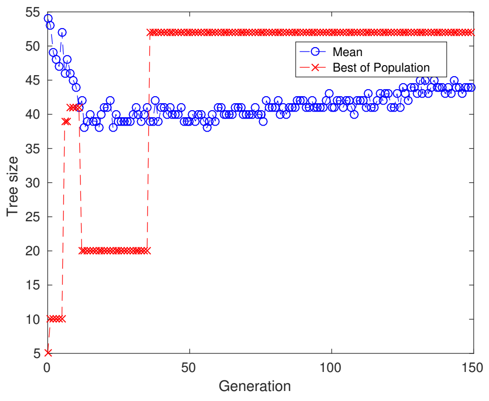
5-2-1 RL as Active Policy

To investigate the influence on the active policy in the DTMC, the optimization is run by transitioning in the DTMC using the action selected by the RL policy instead of the BT as done in the previous sections. The GAs settings used are the same as presented in the scientific paper. The only with difference is the acting policy. The preliminary analysis for the RL policy as active policy in the DTMC are discussed.

The optimization is run also used the same fitness function as was specified in the scientific paper in Part I.



(a) Progression of the fitness score of the best individual and the mean of the population throughout the genetic optimization. The highest fitness of the best scoring individual is evolved after 8 generations.



(b) Progression of the number of nodes of best individual of the population and the mean of the population throughout the genetic optimization.

Figure 5-10: Genetic Optimization results for run with RL policy as active policy.

The evolved BT which is optimized using the RL action selection in the DTMC is presented in Figure 5-11. An overview of the results of the evolved BT are presented in Table 5-7.

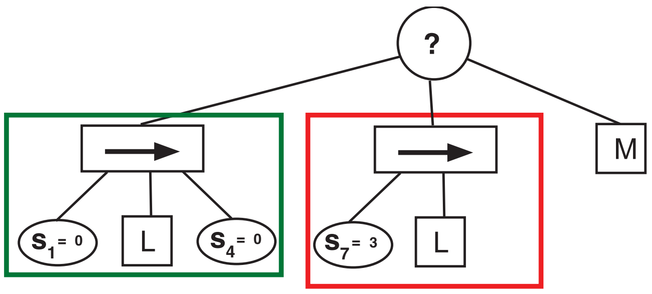


Figure 5-11: Graphical depiction of optimized BT using GAs. Transitions in the DTMC for the optimization based on the action selection of the RL policy. Green box indicates behavior to turn before a corner. Red box indicates the behavior to collect trash on its left.

Collision avoidance
By evaluating the tree, it is clear that the tree misses the nodes for a wall check *in front*. However, when the UAV moves towards a wall the check on sensor *s1* and *s4* ensure that the UAV will turn left instead of hitting the wall indicated with the green box. However, the results indicate that the UAV collides with the wall, resulting in an average reward of -484.

The motion plot in Figure 5-12, shows the UAV with the BT as action selector hit the obstacle block in the environment.

Table 5-7: Results optimized BT with GA for 150 generations using the RL policy to transition through the DTMC.

State	Q-learning	Optimized BT with Q-values
Action-selection for all states	100 %	82 %
Action-selection for simulation	100 %	49 %
Average reward	260	-484
Representation	3123 entries	9 nodes

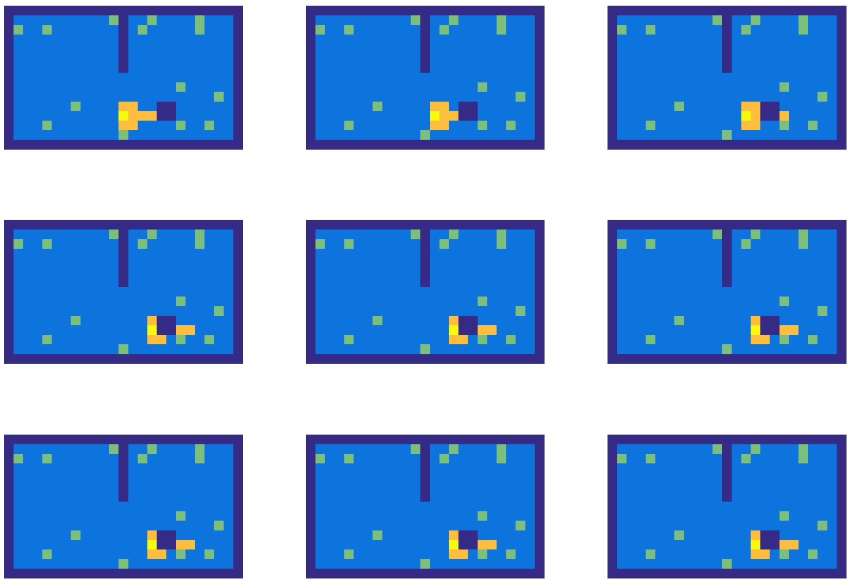


Figure 5-12: A motion example of UAV with the optimized BT as action selection policy. Agent shows goal driven behavior when trash is observed. Dark blue indicate walls, light blue indicate free space, green indicate trash, yellow indicates the UAV's position. Read from left to right, top to bottom.

The result of a tree without the check on a wall in front is seen in **??**. This explains the reward Table 5-7. To provide more information, the reward per run are shown in Table 5-8. This show that if 4 runs in the simulation were performed, this tree would have a high average score compared to its average score on 10 runs. It shows that the obstacle in the environment is not encountered often, which also explains the missing behavior nodes to select the right action in this state.

The simulation run without encountering the obstacle in the environment results in an successful action identification of 86% and a average reward of 119. The BT framework does provide the ability to easily extend the tree with this check.

Collect trash
The red box around the 3 nodes in Figure 5-11 show that the tree will select to turn left when a piece of trash is observed on its left sensor. However, when trash is observed on the right of the UAV, the tree does not identify the action to collect it.

Search trash

Table 5-8: Reward for 10 runs with the UAV running the evolved tree.

Run:	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Reward:	107	145	147	38	-1212	-1034	-1034	-1014	-974	159
Simulation Success rate	88%	85%	87%	79%	16%	25%	25%	24%	24%	85%

The tree evolved the characteristic behavior of turning before the corner. This way there is an increase chance to encounter trash. During the evaluation of the RL controller, this behavior was labeled as typical and was not optimized when the BT was selected as active policy. The green box in Figure 5-11 shows this characteristic behavior.

5-2-2 Evaluation Active Policy

A challenge when the RL action identification is used as policy to move through the DTMC is immediately observed. Without extra conditions on the fitness function, the optimization is most likely to transition in a subset of the states in Q-table and DTMC, resulting in missing important transitions.

When an action identified by the tree is wrong, the RL action selection will still move on from this state to the correct new state and this allows the tree to obtain high fitness for the rest of the run. This BT, with high fitness, will however not perform well in simulation if important states, as collision avoidance, are missed and the UAV will get stuck when facing this state in the actual simulation.

For this optimization the same (generic) fitness function is used. To implement small adjustment to this fitness function, or to dynamically switch between the active policies during the simulation might improve the optimization results.

Despite the fact that the evolved tree misses a branch to avoid collisions when an obstacle is observed in front of the UAV, the evolved BT *did* evolve one of the characteristic behavior of the RL controller. This behavior was viewed as the most creative one developed by the RL controller. Without the information to search for trash, it developed a behavior to increase the chance to encounter it. This genetic optimization was the first to evolve this behavior. Setting expectations that changing the active policy to the actions identified by the RL controller might produces better optimizations.

Chapter 6

Conclusions

This thesis presented a method to automatically generate a BT, using GAs, that approximates a RL policy, normally captured in in an difficult to interpret Q-table. A particular focus was whether the BT would help to understand the developed behavior of the RL controller and present the behavior in an intelligible way and allow for adjustments. The method was tested on a UAV simulation with a guidance task in an unknown environment.

The results show that the genetically optimized BT successfully identifies the correct action for 93% of the states encountered in the simulation run. When the action identification of the evolved BT is evaluated for all states in the Q-table, the tree identifies 63% of the actions correctly. After manually extending the tree, the optimized BT has a success rate of 86% for all states in the Q-table and a success rate of 96% for the states encountered in a simulation run. Although high success rates are obtained for the run in simulation, not all characteristic behaviors of the RL policy are captured by the tree.

A graphical depiction of the evolved BT allows easy verification for the subtask *collision avoidance*. The graphical depiction of the evolved BT allows for easy verification for the collision avoidance subtask with only 4 nodes, compared to the verification of 1464 state-action values in the Q-table. This suggests that a BT provides easy verification of subtasks for the designer of RL learned behavior.

The approximation is performed using GAs in a DTMC. This DTMC is constructed during the learning of the guidance task by the UAV. The DTMC successfully captures the transition probability of the UAV in the environment without the need to simulate the environment, which speed up computation for the optimization. However, transitioning to rare states pose a challenge during the optimization in the DTMC. But the DTMC helps to identify where insight would be valuable and hence act as both a guide and stimulant to further research.

The aim of this research was to answer the research question: *How can the policy of a Reinforcement Learning controller be made more intelligible by an automatically generated Behavior Tree?*

The results show that it is feasible to learn control actions using the RL Q-learning algorithm and automatically approximate this control policy with the BT framework. This representation with the BT results in a high level of understanding of the developed behavior, provides easy verification and the tools to extend the behavior to improve performance.

This work presented two main contribution: The approximation of a developed RL policy using a BT and this approximation is performed in a DTMC of the environment instead of the simulation environment; the ability to manually adjust the evolved BT and as such allow easy verification of subtask of the developed RL policy.

This thesis work also provided more insight in the policy structure of the RL controller. It is observed that the characteristic policy behavior for a RL agent is strongly coupled to successful action identification for rare states. It suggests that the success of the approximation of a RL policy depends on action identification for these rare states. Based on this observation future investigation should perform a sensitivity analysis identify and involve these states in the approximation.

This method is applied to a dual task with simple dynamics for the UAV system, but is not limited to such tasks. RL controllers are challenged with more difficult tasks with more difficult dynamics for hierarchical RL problems. The hierarchical combination of task can be utilized with the BT framework. The combination of hierarchical RL with the BT framework is interesting to investigate to present the RL policy in subtasks in order of priority

The use of the BT framework as a policy approximation provides the user with increased flexibility, manual adaptation of the developed RL policy and scalability for further development. The BT framework is a promising tool in future development for robotic behavior.

Chapter 7

Recommendations

Behavior Trees The BT framework is formed by hierarchically organizing behavior sub-trees consisting of ordered nodes Directed Acyclic Graphs (DAGs). DAGs consist of a number of nodes that are connected with directed edges which could be used by a visual editor to visualize and even edit the BT. Future research can design a Graphical User Interface (GUI) to be able to interactively design BTs. This would increase the validation of the automatically generated behavior since the designer could detect faults more easily.

Reinforcement Learning The variables changed during the learning of the RL controller were limited only to initial position and orientation of the UAV, additional parameters may be useful to promote more general behavior to learn the guidance task more effectively. Rooms with different shapes and environmental objects in as well as changes to vehicle dynamics which are more representative of reality should aid the development of more effective RL controller.

This method was applied to a simple dual-task problem with limited observability for the agent. Future research will investigate implementing a battery indication on the UAV and a charge location in the environment. larger discrete state-space RL problems, or even continuous state-spaces to test this method.

Genetic Optimization By not adding task specific knowledge to the genetic operators the converged solution is not steered by the human designers. This led to an optimization with large of redundant branches and nodes in the BTs. This is solved after optimization by pruning the BTs. It would be interesting to automate this pruning by applying the simple rules of the BT framework to the evolutionary optimization process. The influence of of pruning on the genetic diversity during the evolution process can also be investigated.

This research presented no investigation into the effect of genetic parameters of the GAs. It was used as a tool which was developed by Scheper et al. (2016) and adjusted for this research. As the genetic operators on BTs can result in quite complex BTs, a more detailed analysis should be performed to adapt it to this research. Different parameter settings were used during this research, but not enough to establish trends or perform analysis on the GAs settings for the optimized solutions.

Simulation runs in the DTMC of each BT in a generation are not dependent on other individuals in the population. This lends the simulation of individuals to run in parallelism. To spread the computational load of the simulations over multiple platforms would reduce the total time to optimism the BTs.

DTMC The observed behavior to free up its sensors is not approximated with the BT. The state for which this action needs to be identified also has a low probability state of transition too in the DTMC. The approximation did not successful identified this behavior, but it is viewed as characteristic for the RL controller. Future research will investigate to involve low probability states better in the optimization without increasing computation time significantly.

Another interesting direction for future research is to investigate which active policy must be followed in the DTMC. Future research will investigate the sensitivity of the acting policy in the DTMC. When the optimization uses the RL policy as active policy to step through the DTMC, an unsuccessful action selection by the tree will still transition to an state that will actually be encountered more often by the RL policy.

A first indication on how this effects the approximation is presented in Chapter 5. In Chapter 5 the active policy for state transitions is the RL policy.

Finally, future research should investigate mapping continuous state-space problems transitions into a DTMC. Providing a framework to approximate a RL without the use of computationally expensive simulation environments.

Appendix A

Pruning

The approximation with GAs let the BTs grow during the process of evolution. During the optimization process only the size of the BT is used in the fitness function. This leaves the chance for the evolution to evolve BTs with nodes in the BT that have no influence on the behavior. These BTs are pruned. Pruning is simply removing nodes from the BT that will have no effect on the exhibited behavior.

Pruning is possible by following the rules of the BT structure and the customized leaf nodes. For the current customization, conditions nodes have two possible return statuses, Success or Failure, where action nodes only return Success. With these leaf nodes and the standard BT framework, the following nodes can be removed:

- nodes after an Action node in a Selector will not be evaluated, see Listing A.1
- multiple Action nodes within a Sequence result in only the last action, see Listing A.2
- a composite with only one child can be replaced by its child node, see Listing A.3

Listing A.1: nodes in green is an Action node in a Selector will not be evaluated

```
<BTtype>composite<function>selector<vars><name>selector<endl>
    <BTtype>composite<function>sequence<vars><name>sequence<endl>
        <BTtype>composite<function>selector<vars><name>selector<endl>
            <BTtype>action<function>turn_left<vars>1<name>khepera<endl>
        <BTtype>condition<function>equal_to<vars>7,3<name>khepera<endl>
    <BTtype>condition<function>equal_to<vars>3,0<name>khepera<endl>
    <BTtype>composite<function>selector<vars><name>selector<endl>
        <BTtype>composite<function>sequence<vars><name>sequence<endl>
            <BTtype>composite<function>sequence<vars><name>sequence<endl>
                <BTtype>condition<function>equal_to<vars>3,0<name>khepera<endl>
                <BTtype>condition<function>equal_to<vars>1,0<name>khepera<endl>
                <BTtype>action<function>turn_right<vars>0<name>khepera<endl>
                <BTtype>action<function>move<vars>2<name>khepera<endl>
            <BTtype>composite<function>sequence<vars><name>sequence<endl>
                <BTtype>condition<function>equal_to<vars>6,0<name>khepera<endl>
                <BTtype>condition<function>equal_to<vars>1,3<name>khepera<endl>
            <BTtype>action<function>turn_right<vars>0<name>khepera<endl>
            <BTtype>action<function>turn_right<vars>0<name>khepera<endl>
    <BTtype>action<function>move<vars>2<name>khepera<endl>
    <BTtype>action<function>turn_left<vars>1<name>khepera<endl>
    <BTtype>condition<function>equal_to<vars>6,3<name>khepera<endl>
```

Listing A.2: Multiple Action nodes within a Sequence result in only the last action node so node indicated in green can be pruned away

```
<BTtype>composite<function>selector<vars><name>selector<endl>
    <BTtype>composite<function>sequence<vars><name>sequence<endl>
        <BTtype>composite<function>selector<vars><name>selector<endl>
            <BTtype>action<function>turn_left<vars>1<name>khepera<endl>
        <BTtype>condition<function>equal_to<vars>7,3<name>khepera<endl>
    <BTtype>condition<function>equal_to<vars>3,0<name>khepera<endl>
    <BTtype>composite<function>selector<vars><name>selector<endl>
        <BTtype>composite<function>sequence<vars><name>sequence<endl>
            <BTtype>composite<function>sequence<vars><name>sequence<endl>
                <BTtype>condition<function>equal_to<vars>3,0<name>khepera<endl>
                <BTtype>condition<function>equal_to<vars>1,0<name>khepera<endl>
                <BTtype>action<function>turn_right<vars>0<name>khepera<endl>
                <BTtype>action<function>move<vars>2<name>khepera<endl>
            <BTtype>composite<function>sequence<vars><name>sequence<endl>
                <BTtype>condition<function>equal_to<vars>6,0<name>khepera<endl>
                <BTtype>condition<function>equal_to<vars>1,3<name>khepera<endl>
            <BTtype>action<function>turn_right<vars>0<name>khepera<endl>
    <BTtype>action<function>move<vars>2<name>khepera<endl>
```

Listing A.3: A sequence with one child can be replaced by its child node

```
<BTtype>composite<function>selector<vars><name>selector<endl>
    <BTtype>composite<function>sequence<vars><name>sequence<endl>
        <BTtype>action<function>turn_left<vars>1<name>khepera<endl>
        <BTtype>condition<function>equal_to<vars>7,3<name>khepera<endl>
    <BTtype>condition<function>equal_to<vars>3,0<name>khepera<endl>
    <BTtype>composite<function>selector<vars><name>selector<endl>
        <BTtype>composite<function>sequence<vars><name>sequence<endl>
            <BTtype>composite<function>sequence<vars><name>sequence<endl>
                <BTtype>condition<function>equal_to<vars>3,0<name>khepera<endl>
                <BTtype>condition<function>equal_to<vars>1,0<name>khepera<endl>
                <BTtype>action<function>move<vars>2<name>khepera<endl>
            <BTtype>composite<function>sequence<vars><name>sequence<endl>
                <BTtype>condition<function>equal_to<vars>6,0<name>khepera<endl>
                <BTtype>condition<function>equal_to<vars>1,3<name>khepera<endl>
            <BTtype>action<function>turn_right<vars>0<name>khepera<endl>
    <BTtype>action<function>move<vars>2<name>khepera<endl>
```

Applying global information of the node combinations leads to further reduction in size. An example of this global optimization can be seen in Listing A.4

If we evaluate the BT to the Condition node on line 5 and the branch of nodes on line 9,10,11,12,13 under Sequence node at line 7. Evaluating the trees execution we see that the Condition node has two possible outcomes. If the outcome of the Condition node on line 5 is Success, the Sequence node node on line 7 will never be evaluated. In this case the branch on line 9,10,11,12,13 will not be evaluated. If the outcome of the Condition node on line 5 is Failure, then the Sequence node will be evaluated. Its first branch start with a checking the same condition as Condition node in line 5. Returning Failure to its parent, such that the Sequence will not evaluate its children further. In this case again, the Selector and its children have no effective impact on the output of the BT

Listing A.4: A sequence with one child can be replaced by its child node

```
 1  <BTtype>composite<function>selector<vars><name>selector<endl>
 2      <BTtype>composite<function>sequence<vars><name>sequence<endl>
 3          <BTtype>action<function>turn_left<vars>1<name>khepera<endl>
 4          <BTtype>condition<function>equal_to<vars>7,3<name>khepera<endl>
 5      <BTtype>condition<function>equal_to<vars>3,0<name>khepera<endl>
 6      <BTtype>composite<function>selector<vars><name>selector<endl>
 7          <BTtype>composite<function>sequence<vars><name>sequence<endl>
 8              <BTtype>composite<function>sequence<vars><name>sequence<endl>
 9                  <BTtype>condition<function>equal_to<vars>3,0<name>khepera<endl>
10                  <BTtype>condition<function>equal_to<vars>1,0<name>khepera<endl>
11                  <BTtype>action<function>move<vars>2<name>khepera<endl>
12              <BTtype>composite<function>sequence<vars><name>sequence<endl>
13                  <BTtype>condition<function>equal_to<vars>6,0<name>khepera<endl>
14                  <BTtype>condition<function>equal_to<vars>1,3<name>khepera<endl>
15              <BTtype>action<function>turn_right<vars>0<name>khepera<endl>
16      <BTtype>action<function>move<vars>2<name>khepera<endl>
```

Appendix B

Code structure

This chapter present the implementation of the BT framework. It explain how the framework was set-up. This appendix describes the implementation of the UAV simulation using Unified Modelling Language (UML) charts.

B-1 Simulation

For this research the simulations are programmed in the C++programming languages. The C++programming language allows to interaction with low level hardware and allows for real time performance which is often needed for developing UAVs and MAVs. The writer of this paper had no prior experience with compile languages. Programming in the C++languages turned out to be a challenge compared to the scripting language MATLAB. It took more time than expected to implement the same functionalities. However, as control engineer for robotic systems it the languages that is probably the most common languages used in robotics.

This thesis is programmed following the setup of Object-Oriented Programming. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated. In OOP, classes are designed by making them out of objects that interact with one. The UML chart with the class structure for the UAV simulation is shown in Figure B-1.

The UAV class includes all the simple dynamics of the UAV system. The UAV class selects a Solution class, either the Q-learning class or the BT class. Both Solution classes communicate with the UAV through a Blackboard Struct. A Blackboard is used to store the required data and manages the reading and writing of data from any requester, UAV Class or Solution Class.

This DTMC class is constructed during the learning of the guidance task by the UAV. When the learning stops, the transition probability matrix is stored in the DTMC class. The Solution Class can now be evaluated for the UAV in simulation, but also in the DTMC Class.

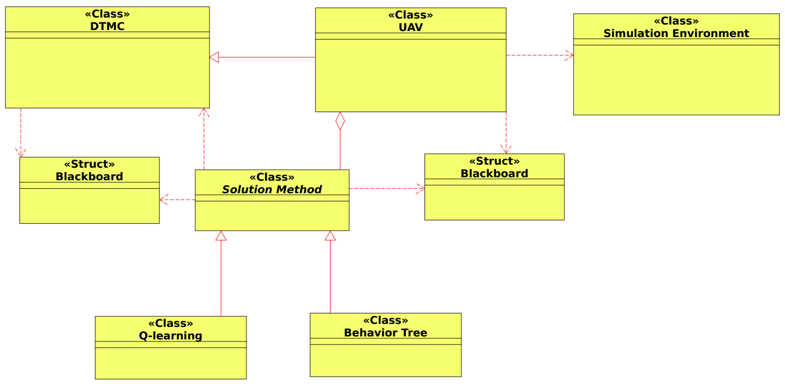


Figure B-1: Static class structure of the modeled UAV simulation.

B-2 BT implementation

For the Evolutionary Optimization (EO) on BTs the framework of Scheper et al. (2016) is implemented to use the GA and adjusted for this research. The notation of the BT framework is adapted from (Scheper et al., 2016; Champandard, 2007)

The standard node class, called behavior, is defined from which all other types of nodes inherit from. This standard class is independent from the implementation for this research. This set up the code such that it maintains the re usability and expandability which make the BT framework so useful.

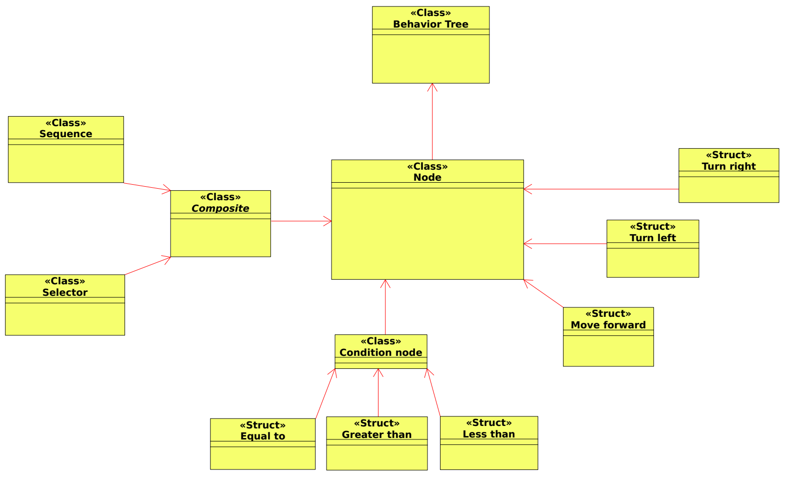
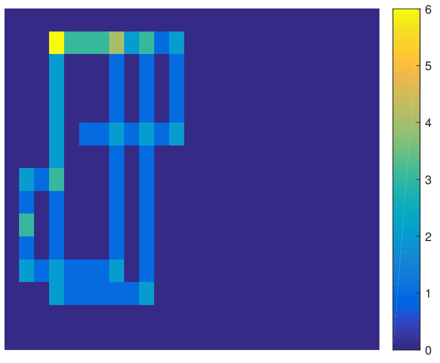


Figure B-2: The class structure of the BT class.

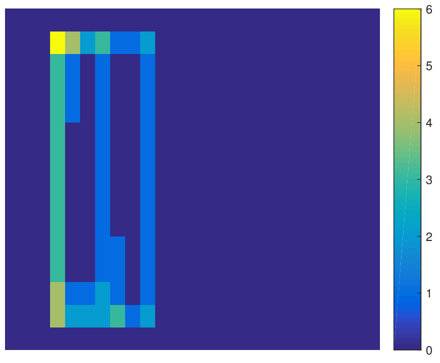
The Blackboard architecture implemented for the UAV to add data to the BT, containing 9 entries: for each 8 sensors a variable s_x, and an action a. The sensor values are set as the input for the BT and the action is set as as output by the BT.

Appendix C

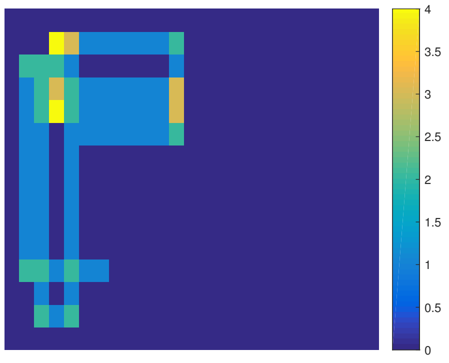
Reinforcement Learning



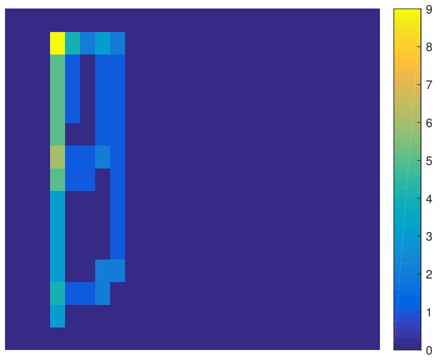
(a) Exploration UAV reward 109.



(b) Exploration UAV reward 105.

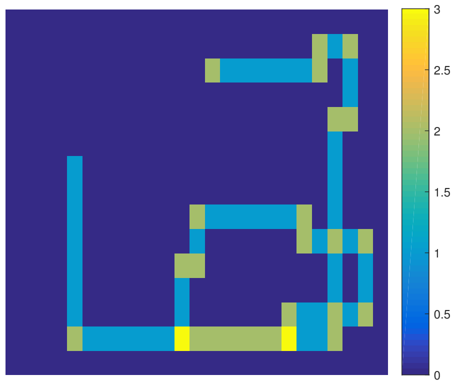


(c) Exploration UAV reward 109.

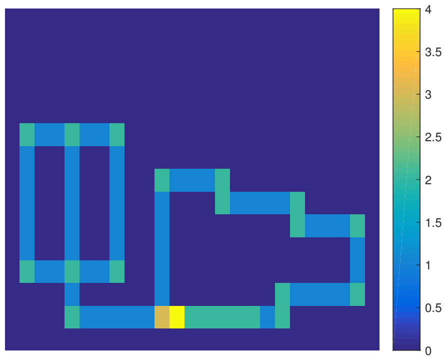


(d) Exploration UAV reward 105.

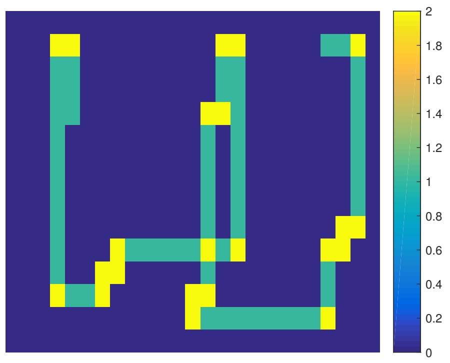
Figure C-1: Exploration maps for low reward episodes.



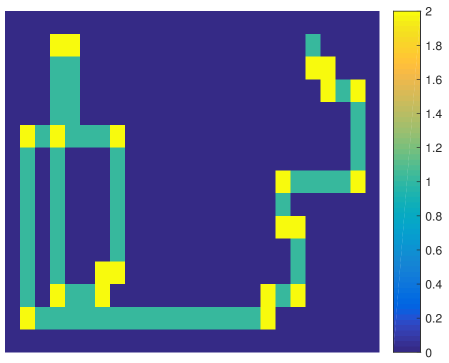
(a) Exploration UAV reward 262.



(b) Exploration UAV reward 262.



(c) Exploration UAV reward 279.



(d) Exploration UAV reward 240.

Figure C-2: Exploration maps for high reward episodes.

Bibliography

Al-emran, M. (2015). Hierarchical Reinforcement Learning: A Survey. *International Journal of Computing and Digital Systems*, *2*(2).

Andre, D., & Russell, S. J. (2001). Programmable reinforcement learning agents. In (pp. 1019–1025). MIT Press.

Barto, A. G., & Dietterich, T. G. (2004). Reinforcement learning and its relationship to supervised learning. In *Handbook of learning and approximate dynamic programming* (pp. 47–64). Wiley - IEEE Press. doi: 10.1002/9780470544785.ch2

Barto, A. G., & Mahadevan, S. (2003). Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, *13*(1/2), 41–77. doi: 10.1023/a:1022140919877

Bishop, C. (1994). Novelty detection and neural network validation. *IEE Proceedings - Vision, Image, and Signal Processing*, *141*(4), 217.

Busoniu, L., Schutter, B. D., Babuska, R., & Ernst, D. (2010, may). Using prior knowledge to accelerate online least-squares policy iteration. In *2010 IEEE international conference on automation, quality and testing, robotics (AQTR).* Institute of Electrical & Electronics Engineers (IEEE).

Champandard, A. J. (2007). *Packaging - google chrome.* `https://aigamedev.com/insider/presentations/behavior-trees/`. ((Accessed on 11/03/2016))

Colledanchise, M., & Ogren, P. (2014, sep). How behavior trees modularize robustness and safety in hybrid systems. In *2014 IEEE/RSJ international conference on intelligent robots and systems.* Institute of Electrical & Electronics Engineers (IEEE). doi: 10.1109/iros.2014.6942752

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

Dey, R., & Child, C. (2013, aug). QL-BT: Enhancing behaviour tree design and implementation with q-learning. In *2013 IEEE conference on computational inteligence in games (CIG).* Institute of Electrical & Electronics Engineers (IEEE). doi: 10.1109/cig.2013.6633623

Dietterich, T. G. (2000). Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research*, *13*, 227–303. doi: 10.1613/jair.639

Doya, K. (2000). Reinforcement learning in continuous time and space. *Neural Computation*, *12*(1), 219–245. doi: 10.1162/089976600300015961

Dromey, R. (2003). From requirements to design: formalizing the key steps. In *First international conference onSoftware engineering and formal methods, 2003.proceedings.* Institute of Electrical & Electronics Engineers (IEEE). doi: 10.1109/sefm.2003.1236202

He, P., & Jagannathan, S. (2007, apr). Reinforcement learning neural-network-based controller for nonlinear discrete-time systems with input constraints. *IEEE Trans. Syst., Man, Cybern. B*, *37*(2), 425–436.

Isla, D. (2005). Handling Complexity in the Halo 2 AI. In *Gdc 2005 proceeding.*

Klöckner, A. (2013). Behavior Trees for UAV Mission Management. *Informatik 2013: informatik angepasst an Mensch, Organisation and Umwelt*, *P-220*(September), 57–68.

Klöckner, A. (2013, aug). Interfacing behavior trees with the world using description logic. In *AIAA guidance, navigation, and control (GNC) conference.* American Institute of Aeronautics and Astronautics (AIAA). doi: 10.2514/6.2013-4636

Kober, J., Bagnell, J. A., & Peters, J. (2013, aug). Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, *32*(11), 1238–1274. doi: 10.1177/0278364913495721

Lazaric, A., Restelli, M., & Bonarini, A. (2007). Reinforcement learning in continuous action spaces through sequential monte carlo methods. In *Advances in neural information processing systems* (p. 8).

Lindsay, P. A. (2010, Sept). Behavior trees: From systems engineering to software engineering. In *2010 8th IEEE international conference on software engineering and formal methods.* Institute of Electrical & Electronics Engineers (IEEE). doi: 10.1109/sefm.2010.11

Millán, J. D. R., Posenato, D., & Dedieu, E. (2002). Continuous-action Q-learning. *Machine Learning*, *49*(2-3), 247–265. doi: 10.1023/A:1017988514716

Millington, I., & Funge, J. (2009). *Artificial intelligence for games, second edition* (2nd ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Nedić, A., & Bertsekas, D. P. (2003).
Discrete Event Dynamic Systems, *13*(1/2), 79–110.

Nolfi, S. (2002, Jan). Power and the limits of reactive agents. *Neurocomputing*, *42*(1-4), 119–145. doi: 10.1016/s0925-2312(01)00598-7

Ögren, P. (2012, Aug). Increasing modularity of UAV control systems using computer game behavior trees. In *AIAA guidance, navigation, and control conference.* American Institute of Aeronautics and Astronautics (AIAA). doi: 10.2514/6.2012-4458

Palanisamy, M., Modares, H., Lewis, F. L., & Aurangzeb, M. (2015, Feb). Continuous-time q-learning for infinite-horizon discounted cost linear quadratic regulator problems. *IEEE Trans. Cybern.*, *45*(2), 165–176. doi: 10.1109/tcyb.2014.2322116

Parr, R. E. (1998). *Hierarchical control and learning for markov decision processes* (Unpublished doctoral dissertation). University of California at Berkeley.

Perez, D., Nicolau, M., O'Neill, M., & Brabazon, A. (2011). Evolving behaviour trees for the mario ai competition using grammatical evolution. In *Applications of evolutionary computation* (pp. 123–132). Springer Science Business Media. doi: 10.1007/978-3-642-20525-5_13

Qiao, J., Fan, R., Han, H., & Ruan, X. (2009). Q-learning based on dynamical structure neural network for robot navigation in unknown environment. In *Advances in neural networks ISNN 2009* (pp. 188–196). Springer Science Business Media. doi: 10.1007/978-3-642-01513-7_21

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

Sutton, R., & Barto, A. (1998, Sep). Reinforcement learning: An introduction. *IEEE Trans. Neural Netw.*, *9*(5), 1054. doi: 10.1109/tnn.1998.712192

Sutton, R. S., Precup, D., & Singh, S. (1999, Aug). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*(1-2), 181–211. doi: 10.1016/s0004-3702(99)00052-1

Tesauro, G. (1994, Mar). TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, *6*(2), 215–219. doi: 10.1162/neco.1994.6.2.215

Thorndike, E. L. (1911.) *Animal intelligence; experimental studies, by edward l. thorndike.* New York,The Macmillan company,.

Watkins, C. J. C. H., & Dayan, P. (1992, May). Q-learning. *Machine Learning*, *8*(3-4), 279–292. doi: 10.1007/bf00992698

Xiaoqin, D., Qinghua, L., & Jianjun, H. (2009, Aug). Applying hierarchical reinforcement learning to computer games. In *2009 IEEE international conference on automation and logistics.* Institute of Electrical & Electronics Engineers (IEEE). doi: 10.1109/ical.2009.5262787

Yamasaki, T., Sakaida, H., Enomoto, K., Takano, H., & Baba, Y. (2007). Robust trajectory-tracking method for UAV guidance using proportional navigation. In *2007 international conference on control, automation and systems.* Institute of Electrical & Electronics Engineers (IEEE). doi: 10.1109/iccas.2007.4406558

Yan, Q., Liu, Q., & Hu, D. (2010). A hierarchical reinforcement learning algorithm based on heuristic reward function. In *2010 2nd international conference on advanced computer control.* Institute of Electrical & Electronics Engineers (IEEE). doi: 10.1109/icacc.2010.5486837

Yoshikawa, M., Kihira, T., & Terai, H. (2008). Q-learning based on hierarchical evolutionary mechanism. *WSEAS Transactions on Systems and Control*, *3*(3), 219–228.