Faculty of Aerospace Engineering

Behaviour Trees for Evolutionary Robotics

Reducing the Reality Gap

K.Y.W. Scheper

June 18, 2014

Behaviour Trees for Evolutionary Robotics

Reducing the Reality Gap

Master of Science Thesis

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

K.Y.W. Scheper

June 18, 2014

Faculty of Aerospace Engineering · Delft University of Technology

Delft University of Technology

Delft University Of Technology
Department Of
Control and Simulation

The undersigned hereby certify that they have read and recommend to the Faculty of Aerospace Engineering for acceptance a thesis entitled **"Behaviour Trees for Evolutionary Robotics"** by **K.Y.W. Scheper** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: June 18, 2014

Readers:

Prof. Dr. ir. J.A. Mulder

Dr. G.C.H.E. de Croon

Dr. ir. C.C. de Visser

Dr. ir. E. van Kampen

Luís F. Simões, M.Sc

Abstract

Designing effective behaviour with limited computational and sensory capabilities of small robotic platforms can be quite challenging for a human user. Evolutionary Robotics is a promising method to address this problem by allowing the robotic platform to autonomously learn effective behaviour. Automated learning often results in unexpected solutions to tasks utilising advanced sensory-motor coordination. This allows small and *limited* platforms to perform complex tasks.

Evolutionary Robotics typically involves the optimization of artificial neural networks in simulation to a solve a specific task. The advantage of such networks is that they provide distributed, parallel solutions, however, analysing and understanding evolved networks requires considerable effort. Additionally, as simulation always differs to some degree from reality, simulation based learning typically results in a reality gap between behaviour expressed in simulation and that in the real world. This thesis aims to show that the Behaviour Tree framework can be used to effectively express automatically developed robotic behaviour in a readily comprehensible manner. We also show that this improved understanding of the underlying behaviour can be used to reduce the reality gap when moving form simulation to reality. In this paper we answer the research question:

How can a Behaviour Tree framework be used to develop an effective automatically generated Artificial Intelligence UAV control system to reduce the reality gap of simulation trained systems?

The DelFly flapping wing UAV was selected to investigate the Behaviour Tree approach. The DelFly is tasked to fully autonomously navigate a room in search of a window which it must then fly through using onboard capabilities only. This is the most complex task yet attempted by the DelFly platform. The reality gap reduction is tested by first developing the behaviour tree to solve this task automatically using Evolutionary Learning techniques in simulation. This behaviour will then be applied to a real world DelFly and the user will be tasked with adapting the behaviour to reduce the eventual reality gap. A user-defined behaviour is used as a benchmark to compare the performance of genetically optimised behaviour.

The genetically optimised behaviour tree eventually contained only 8 behaviour nodes. The behaviour resulted in a simulation based success rate of 88%, slightly better than the 22 node user-defined behaviour at 82%. When moving the behaviour to the real platform, a large reality gap was observed as the success rate dropped to almost nil. After user adaptation the genetically optimised behaviour had a success rate of 54%. Although this leaves room for improvement, it is higher than 46% from a tuned user-defined controller.

In this thesis we show that it is feasible to evolve a behaviour tree in simulation and implement that evolved behaviour on a real world platform. We also show that the improved intelligibility of the Behaviour Tree behavioural encoding framework provides the user with tools to effectively identify and reduce the resultant reality gap. Additionally, the genetically optimised behaviour obtains a slightly better performance than a user-defined behaviour, both in simulation and on the real platform.

This work has two main contributions, namely: a Behaviour Tree was implemented on a airborne robotic platform to perform a window search and fly-through task all on the basis of onboard sensors and processing and the ability to reduce the reality gap of robotic behaviour using Behaviour Trees was effectively demonstrated. In essence, Behaviour Trees seem well suited to represent behaviours of increasing complexity. Future research will tell whether it can bring more complex tasks within reach of small and extremely limited robotic platforms.

Acronyms

AFMS	Advanced Flight Management System
AI	Artificial Intelligence
ANN	Artificial Neural Network
BBR	Behaviour-Based Robotics
BT	Behaviour Tree
CSP	Constraints Satisfaction Problem
DAG	Directed Acyclic Graph
DTD	Document Type Definition
DUT	Delft University of Technology
EL	Evolutionary Learning
ENAC	École Nationale de l'Aviation Civile
ER	Evolutionary Robotics
FCS	Flight Control Software
FDM	Flight Dynamics Model
FSM	Finite State Machine
GA	Genetic Algorithms
GCS	Ground Control Station
GNC	Guidance Navigation and Control
GOAP	Goal Oriented Action Planner
GP	Genetic Programming
GPS	Global Positioning System
GUI	Graphical User Interface
HFSM	Hierarchical Finite State Machine
HTN	Hierarchical Task Network
IR	Infrared
LISP	LISt Processing
MAVLink	Micro Air Vehicle Communication Protocol

NEAT	NeuroEvolution of Augmenting Topologies
NPC	Non-Player Character
PDDL	Planning Domain Definition Language
RL	Reinforcement Learning
STRIPS	Stanford Research Institute Problem Solver
UAV	Unmanned Aerial Vehicle
UML	Unified Modelling Language

Contents

List of Figures

Chapter 1

Introduction

Designing effective behaviour to complete complex tasks for small robotic platforms is a major challenge. Small vehicles with limited computational and sensory capabilities are becoming more common place due to their ability to swarm and collaboratively achieve a task, this however makes the task of a behavioural designer even harder. A promising method to address this problem is found in Evolutionary Robotics (ER), a methodology in which a robot's controller, and possibly its body, is optimised using Evolutionary Learning (EL) techniques (Nolfi & Floreano, 2000; Bongard, 2013). This approach often results in unconventional methods which exploit sensory-motor coordination to achieve complex tasks (Nolfi, 2002).

Early investigations into ER performed EL optimisation on real robotic platforms but this process is time consuming (Floreano & Mondada, 1994; Nolfi et al., 1994). With the ever improving computing technologies, simulation based learning has become the predominant method to evaluate ER, this however has some drawbacks of its own. Simulated environments always differ to some degree from reality, artifacts from the simulation are sometimes exploited by the EL optimisation solution strategy (Floreano & Mondada, 1994). As a result the behaviour seen in simulation can often not be recreated on a real robotic platform resulting in a *reality gap* (Nolfi et al., 1994; Jakobi et al., 1995).

Many methods have been investigated to reduce this reality gap and can be separated into three main approaches (Bongard, 2013). The first approach investigates the influence of simulation fidelity on the EL, with investigation focusing on the influence of adding differing levels of noise to the robotic agent's inputs and outputs (Jakobi et al., 1995; Miglino et al., 1995; Meeden, 1998). It was shown that sufficient noise can deter the EL from exploiting artifacts in the simulation but that this approach is generally not scalable as more simulation runs are needed to distinguish between noise and true environmental features. The second group focuses on *co-evolution*, this approach simultaneously develops a robotic controller which is evaluated in simulation while the simulation model is updated using the performance error with a real world robotic platform (Bongard et al., 2006; Zagal & Solar, 2007). Alternatively, the error between the simulation and real world environment can be used to estimate the suitability of a learnt behaviour on the real robot in a multi-objective function to trade off simulated robotic performance and the transferability of the behaviour (Koos et al., 2013). The final approach performs adaptation of the real robot behaviour after the EL using relatively complex methods (Hartland & Bredèche, 2006).

One factor adding to the reality gap problem is that typically Artificial Neural Networks (ANNs) are used as the encoding framework for the robot behaviour (Nolfi & Floreano, 2000). Although analysis of the evolved ANN is possible, this black-box framework does not lend itself well to manual adaptation hence requiring complex retraining algorithms to bridge the gap. Encoding the EL optimised behaviour in a more intelligible framework would aid a user in understanding the solution strategy. It would also help to reduce the reality gap by facilitating manual parameter adaptation for use on the real system.

Traditionally, user-defined autonomous behaviours are described using Finite State Machines (FSMs) which has also been successfully used within ER (Petrovi, 2008; König et al., 2009; Pintér-Bartha et al., 2012). FSMs are very useful for simple action sequences but quickly become illegible as the tasks become more complex due to *state explosion* (Valmari, 1998; Millington & Funge, 2009). This complexity makes it difficult for developers to modify and maintain the behaviour of the autonomous agents. A more recently developed method to describe behaviour is the Behaviour Tree (BT). Initially developed as a method to formally define system design requirements the BT framework was adapted by the computer gaming industry to control Non-Player Characters (NPCs) (Dromey, 2003; Champandard, 2007). BTs do not consider states and transitions the way FSMs do, rather they consider self contained behaviour made up of a hierarchical network of actions (Champandard, 2007; Heckel et al., 2010). The rooted tree structure of the BT make the encapsulated behaviour readily intelligible for users given that the trees are not too large. Previous work on evolving BTs has been applied to computer game environments where the state is fully known to the BT and actions have deterministic outcomes (Lim et al., 2010; Perez et al., 2011). BTs have not yet been applied to a real world robotic task. Such a task involves complicating factors such as state and action uncertainty, delays, and other properties of a non-deterministic and not fully known environment.

In this thesis, we perform the first investigation into the use of Behaviour Trees in Evolutionary Robotics. The DelFly Explorer flapping wing robotic platform has been selected to demonstrate the efficacy of the BT approach to reduce the reality gap. The DelFly is tasked to navigate a square room in search for an open window which it must fly through using onboard systems only. This is the most complex autonomous task yet attempted with the $20g$ flight platform.

1-1 Research Questions

The main goal of this thesis can be expressed in the following research question:

How can a Behaviour Tree framework be used to develop an effective automatically generated Artificial Intelligence UAV control system to reduce the reality gap of simulation trained systems?

This research question can be answered by the sum of the answers to the following research questions:

RQ1 What Behaviour Tree nodes will be used to implement the guidance system for the fly-through-window task?

RQ2 What inputs and outputs will be used by BT to interface with the UAV for the fly-through-window task?

RQ3 How will the Evolutionary Leaning operators be applied to the Behaviour Tree?

RQ4 What operating parameters will be used for the genetic optimisation to converge on a solution?

RQ5 How will the mission management system performance be evaluated?

RQ6 What is the performance of the learned mission management system as compared to a human designer?

RQ7 Can the the reality gap be reduced using the BT framework?

1-2 Thesis Layout

The first part of this thesis will summarise the methodology, implementation and results of this work in a scientific paper format. The second part will go more into more detail about the implementation and theoretical background.

To answer these research questions we will first discuss some detailed background information of many of the conceptual components of this work. This will be followed by a description to the implementation of the Behaviour Tree on the DelFly platform will be given in Chapter 3. Next, Chapter 4 describes the implementation of the EL method followed by the results of the genetic optimisation in Chapter 5. Chapter 6 presents the implementation of the Behaviour Tree framework onboard the DelFly and the results of the real world flight tests respectively. The work done in this thesis is then summarised in a conclusion and some recommendations are presented.

Part I

Scientific Paper

Behaviour Trees for Evolutionary Robotics

K.Y.W. Scheper, S. Tijmons, C.C. de Visser, and G.C.H.E. de Croon

***Abstract*—Evolutionary Robotics allows robots with limited sensors and processing to tackle complex tasks by means of creative sensory-motor coordination. In this paper we show the first application of the Behaviour Tree framework to a real robotic platform. This framework is used to improve the intelligibility of the emergent robotic behaviour as compared to the traditional Neural Network formulation. As a result, the behaviour is easier to comprehend and manually adapt when crossing the reality gap from simulation to reality. This functionality is shown by performing real-world flight tests with the 20-gram DelFly Explorer flapping wing UAV equipped with a 4-gram onboard stereo vision system. The experiments show that the DelFly can fully autonomously search for and fly through a window with only its onboard sensors and processing. The simulation success rate is 88%, this resulted in a real-world performance of 54% after necessary user adaptation. Although this leaves room for improvement, it is higher than 46% from a tuned user-defined controller.**

***Index Terms*—Behaviour Tree, Evolutionary Robotics, Reality Gap, UAV**

I. Introduction

DESIGNING effective behaviour to complete complex tasks for small robotic platforms is a major challenge. Small vehicles with limited computational and sensory capabilities are becoming more common place due to their ability to swarm and collaboratively achieve a task, this however makes the task of a behavioural designer even harder. A promising method to address this problem is found in Evolutionary Robotics (ER) is a methodology in which a robot's controller, and possibly its body, is optimised using Evolutionary Learning (EL) techniques [1], [2]. This approach often results in unexpected solutions which exploit sensory-motor coordination to achieve complex tasks [3].

Early investigations into ER used online EL but this process is time consuming [4], [5]. With the ever improving computing technologies, simulation based learning has become the predominant method to evaluate ER, this however has some drawbacks of its own. Simulated environments always differ to some degree from reality, artifacts from the simulation are sometimes exploited by the EL optimisation solution strategy [4]. As a result the behaviour seen in simulation can often not be recreated on a real robotic platform resulting in a *reality gap* [5], [6].

Many methods have been investigated to reduce this reality gap and can be separated into three main approaches [2]. The first approach investigates the influence of simulation fidelity on the EL, with investigation focusing on the influence of adding differing levels of noise to the robotic agent's inputs and outputs [6]–[8]. It was shown that sufficient noise can deter the EL from exploiting artifacts in the simulation but that this approach is generally not scalable as more simulation runs are needed to distinguish between noise and true environmental features. The second group focuses on *co-evolution*, this approach simultaneously develops a robotic controller which is evaluated in simulation while the simulation model is updated using the performance error with a real world robotic platform [9], [10]. Alternatively, the error between the simulation and real world environment can be used to estimate the suitability of a learnt behaviour on the real robot in a multi-objective function to trade off simulated robotic performance and the transferability of the behaviour [11]. The final approach performs adaptation of the real robot behaviour after the EL using relatively complex methods [12].

One factor adding to the reality gap problem is that typically Artificial Neural Networks (ANNs) are used as the encoding framework for the robot behaviour [1]. Although analysis of the evolved ANN is possible, this black-box framework does not lend itself well to manual adaptation hence requiring complex retraining algorithms to bridge the gap. Encoding the EL optimised behaviour in a more intelligible framework would aid a user in understanding the solution strategy. It would also help to reduce the reality gap by facilitating manual parameter adaptation for use on the real system.

Traditionally, user-defined autonomous behaviours are described using Finite State Machines (FSMs) which has also been successfully used within ER [13]–[15]. FSMs are very useful for simple action sequences but quickly become illegible as the tasks become more complex due to *state explosion* [16], [17]. This complexity makes it difficult for developers to modify and maintain the behaviour of the autonomous agents. A more recently developed method to describe behaviour is the Behaviour Tree (BT). Initially developed as a method to formally define system design requirements the BT framework was adapted by the computer gaming industry to control Non-Player Characters (NPCs) [18], [19]. BTs do not consider states and transitions the way FSMs do, rather they consider self contained behaviour made up of a hierarchical network of actions [19], [20]. The rooted tree structure of the BT make the encapsulated behaviour readily intelligible for users given that the trees are not too large. Previous work on evolving BTs has been applied to computer game environments where the state is fully known to the BT and actions have deterministic outcomes [21], [22]. BTs have not yet been applied to a real world robotic task. Such a task involves complicating factors such as state and action uncertainty, delays, and other properties of a non-deterministic and not fully known environment.

In this paper, we perform the first investigation into the use of Behaviour Trees in Evolutionary Robotics. We will first give a description of the DelFly Explorer flapping wing robotic platform selected to demonstrate our approach. This is followed by a description how offline EL techniques are used

All authors are with the Faculty of Aerospace, Delft University, 2629 HS Delft, The Netherlands. g.c.h.e.decroon@tudelft.nl

to automatically develop BTs. We will then show the efficacy of this automatically generated behaviour by comparing it to one designed by a human user. The implementation of both behaviours on the real world DelFly Explorer flight platform will be described to investigate if the reality gap can indeed be actively reduced by a user as a result of the legible behaviour expressed using the proposed method.

II. DelFly Fly-Through-Window

The limited computational and sensory capabilities of the DelFly Explorer makes it difficult to design even the most simple behaviour. This makes it an ideal candidate for the implementation of ER. In this paper, the DelFly Explorer is tasked to navigate a square room in search for an open window which it must fly through using onboard systems only. This is the most complex autonomous task yet attempted with the $20g$ flight platform.

Other applications with flapping wing flight platforms include using the H^{2}Bird $13g$ flapping wing Unmanned Aerial Vehicle (UAV) for a fly-through-window task [23]. Unlike the DelFly Explorer, the H^{2}Bird used a ground based camera and off-board image processing to generate heading set-points. Developing the algorithms to safely avoid crashing into the walls and other obstacles while searching and attempting to fly through a window is a non-trivial task. In-fact, the fly-through-window task is the most complex task to date for the DelFly Explorer.

A. DelFly Explorer

The DelFly is an insect-inspired flapping-wing UAV developed at the Delft University of Technology (DUT). The main feature of its design is its biplane-wing configuration which flap in anti-phase [24]. The DelFly Explorer is a recent iteration of this micro ornithopter design [25]. In its typical configuration, the DelFly Explorer is $20g$ and has a wing span of $28cm$. In addition to its 9 minute flight time, the DelFly Explorer has a large flight envelope ranging from maximum forward flight speed of $7m/s$, hover, and a maximum backward flight speed of $1m/s$. A photo of the DelFly Explorer can be seen below in Figure 1.

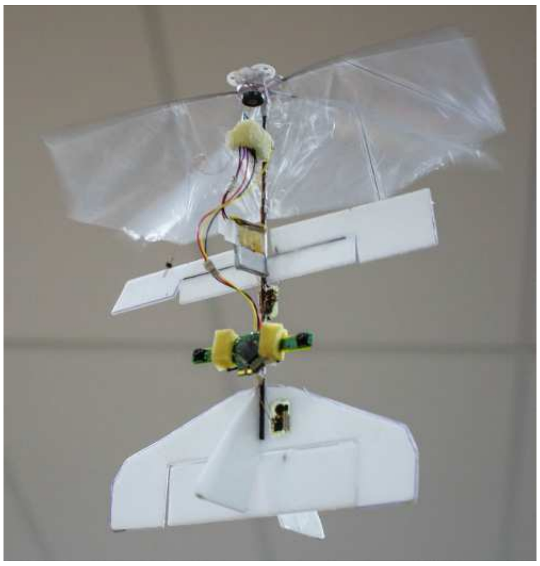


Fig. 1. DelFly Explorer in flight showing dual camera payload

The main payload of the DelFly Explorer is a pair of light weight cameras used to perform onboard vision based navigation as shown in Figure 1. Each camera is set to a resolution of 128×96 pixels with a field of view of $60° \times 45°$ respectively. The cameras are spaced $7cm$ apart facilitating stereo-optic vision. Using computer vision techniques these images can be used to generate depth perception with a method called Stereo Vision [26]. This makes the DelFly Explorer the first flapping wing UAV that can perform active obstacle avoidance using onboard sensors facilitating fully autonomous flight in unknown environments [25].

B. Vision Systems

1) LongSeq Stereo Vision: The DelFly Explorer uses a Stereo Vision algorithm called *LongSeq* to extract depth information of the environment from its two onboard optical cameras [25]. The main principle in computer vision based stereo vision is to determine which pixel corresponds to the same physical object in two or more images. The apparent shift in location of the the pixels is referred to as the disparity. The stereo vision algorithm produces a disparity map of all pixels in the images [26].

LongSeq is a localised line based search stereo vision algorithm. This is one candidate which results of the trade-off between computational complexity and image performance made by all image processing algorithms. The relatively low computational and memory requirements of LongSeq makes it a good candidate for application on the limited computational hardware onboard the DelFly Explorer.

2) Window Detection: An Integral Image window detection algorithm is used to aid the UAV in the fly-through-window task. Integral image detection is a high speed pattern recognition algorithm which can be used to identify features in a pixel intensity map [27], [28]. The integral image ($II(x, y)$) is computed as

$$II(x,y) = \sum_{x' \le x, y' \le y} I(x', y') \tag{1}$$

where x and y are pixel locations in the image I. As each point of the integral image is a summation of all pixels above and to the left of it, the sum of any rectangular subsection is simplified to the following computation

$$\begin{aligned} rect(x,y,w,h) =& II(x+w, y+h) + II(x,y) \\ & - II(x+w,h) - II(x, y+h) \end{aligned} \tag{2}$$

This method has been used to identify a dark window in a light environment by using cascaded classifiers [29]. This application was designed specifically to operate when approaching a building in the day time on a light day. A more generalised method is to apply the same technique described above to the disparity map rather than the original camera images. The disparity map would show a window as an area of low disparity (dark) in an environment of higher disparity (light).

C. SmartUAV Simulation Platform

SmartUAV is a Flight Control Software (FCS) and simulation platform developed in house at the DUT [30]. It is used primarily with small and micro sized aerial vehicles and it notably includes a detailed 3D representation of the simulation environment which is used to test vision based algorithms. It can be used as a ground station to control and monitor a single UAV or swarms of many UAVs. As a simulation platform to test advanced Guidance Navigation and Control (GNC) techniques. As a tool developed in-house, designers have freedom to adapt or change the operating computer code at will, making it very suitable for use in research projects.

SmartUAV contains a large visual simulation suite which actively renders the 3D environment around the vehicle. OpenGL libraries are used to generate images on the PC's GPU increasing SmartUAV's simulation fidelity without significant computational complexity. As a result high fidelity 3D environments can be used allowing the vision based algorithms to be tested with a high level of realism.

In terms of the larger SmartUAV simulation, the vision based calculations are the most computationally intensive portion making it the limiting factor for the speed of operation of the wider decision process. The higher the decision loop frequency relative to the flight dynamics the longer a single simulation will take. This must be balanced by the frequency at which the DelFly is given control instructions, where generally higher is better. Considering this trade-off the decision loop was set to run at $5Hz$ relative to the flight dynamics loop. This is a conservative estimate of the actual performance of the vision systems onboard the real DelFly Explorer.

D. Simplified DelFly Model

The DelFly Explorer is a newly developed platform and its flight dynamics have not yet been investigated in depth. As a result an existing model of the DelFly II previously implemented based on the intuition of the DelFly designers will be used in this work. This model is not a fully accurate representation of the true DelFly II dynamics but was sufficient for most vision based simulations previously carried out. In this work, the inaccuracy of the model will intentionally create a reality gap between the simulated dynamics of the DelFly and reality. We will briefly summarise the dynamics used in simulation below.

The DelFly II has three control inputs, namely: Elevator (δ_e), Rudder (δ_r) and Thrust (δ_t). The elevator and rudder simply set the control surface deflection and the thrust sets the flapping speed. The actuator dynamics of the DelFly for the rudder actuator were implemented using a time delay, defined as:

$$\dot{\delta_r} = \delta_{r_{max}}(u_r - \delta_r); \quad u_r : [-1, 1] \tag{3}$$

where u_r is the rudder input and $\delta_{r_{max}}$ is the maximum rudder deflection. This results in a rudder transfer function with a rise time of $2.2s$ and settling time of $3.9s$ The elevator deflection and thrust setting are simply mapped directly from the control input with the following equation:

$$\begin{bmatrix} \delta_e \\ \delta_t \end{bmatrix} = \begin{bmatrix} \delta_{e_{max}} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} u_e \\ u_t + 1 \end{bmatrix}; \quad u_e, u_t : [-1, 1] \tag{4}$$

where u_e and u_t are the elevator and thrust input respectively and $\delta_{e_{max}}$ is the maximum elevator deflection. The pitch (θ), yaw (ψ) and flight velocity (V) dynamics of the DelFly are updated as:

$$\begin{aligned} \begin{bmatrix} \dot{\theta} \\ \dot{\psi} \\ \dot{V} \end{bmatrix} &= \begin{bmatrix} -\frac{1}{72} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \theta \\ \psi \\ V \end{bmatrix} \\ &+ \begin{bmatrix} -\frac{1}{24} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \delta_e \\ \delta_r \\ \delta_t \end{bmatrix} \end{aligned} \tag{5}$$

Things to note are there is no coupling in the flight modes of the simulated DelFly. For this research, the throttle setting was constant at trim position resulting in a flight velocity of $0.5m/s$. Additionally, the maximum turn rate was set to $0.4rad/s$ resulting in a minimum turn radius of $1.25m$. The roll angle (ϕ) of the DelFly is simply a function of the rudder setting as in (6).

$$\phi = \frac{1}{2}\delta_r \tag{6}$$

There are some notable differences between the DelFly II and DelFly Explorer, firstly the Explorer replaces the rudder with a pair of ailerons to roll the DelFly without inducing yaw, this helps to stabilise the captured images. Additionally, the DelFly Explorer is $4g$ heavier and has a slightly higher wing flapping speed. The flight dynamics described above are very simple and not fully representative of the complex flight dynamics involved with the flapping wing flight of the DelFly platform. As a result a large reality gap is expected when moving from the simulation to reality.

III. Behaviour Tree Implementation

BTs are depth-first, ordered Directed Acyclic Graphs (DAGs) used to represent a decision process. DAGs are composed of a number of nodes with directed edges. Each edge connects one node to another such that starting at the root there is no way to follow a sequence of edges to return to the root. Unlike FSMs, BTs consider achieving a goal by recursively simplifying the goal into tasks similar to that seen in the Hierarchical Task Network (HTN). This hierarchy and recursive action make the BT a powerful way to describe complex behaviour.

A. Syntax and Semantics

A BT is syntactically represented as a rooted tree structure, constructed from a variety of nodes each with its individual internal function but all nodes have the same external interface. Each node in a BT has a return status. Generally, the return statuses are either *Success* or *Failure*. The status Success is the status of a node that has been successfully executed. Inversely, the status Failure is used when a node has failed during execution. This however does not define the condition

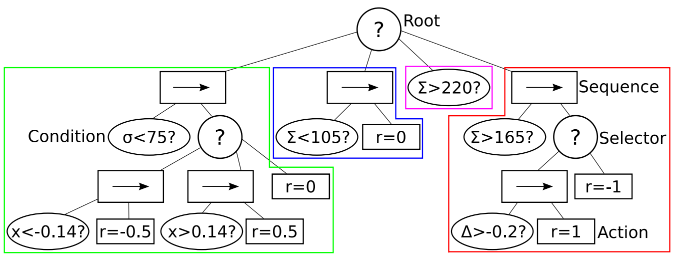


Fig. 2. Graphical depiction of user-defined BT for the fly-through-window task. Colours highlight different phases of the flight. x is the position of the centre of the window in frame, σ is window response value, Σ is sum of disparity and Δ is the horizontal difference in disparity

under which the node failed, some implementations include an *Exception* or *Error* status to provide this information.

Basic BTs are made up of three kinds of nodes: *Conditions*, *Actions* and *Composites* [19]. Conditions and Actions make up the leaf nodes of the BT whilst the branches consist of Composite nodes. Conditions test some property of the environment returning Success if the conditions are met and returns Failure otherwise. The agent acts on its environment through Action nodes. Leaf nodes must be individually developed to perform specific tasks but can be reused readily in the tree as required. Composite nodes are not platform dependent and can be reused in any BT. All the nodes in the BT use a similar interface so that arbitrary combination of these nodes is possible in the BT without knowledge of any other part of the BT making BTs more modular and reusable. A sample BT highlighting the graphical representation of the different nodes can be seen in Figure 2.

As the branches of the BT, Composite nodes determine how the BT is executed. Unlike Conditions and Actions, not many types of Composite nodes are needed as combinations of these simple nodes can achieve very complex behaviour. Although many different types can be used, we will only consider *Sequences* and *Selectors* in this paper. The *Root* node of a BT is typically a Selector node that has no parent.

Selectors return Success when one of its children return Success and Failure when all of its children return Failure. Conversely, Sequences will return Failure when one of its children fails and Success if all of its children return Success. Both of these nodes evaluate their children in order graphically represented from left to right.

The execution of the behaviour tree is referred to as a *tick*. A tick starts from the root node which is typically a selector node and evaluates down the tree starting at the left most node. An execution is complete when a branch of the root node returns success or all of its branches return failure.

B. DelFly Implementation

Aside from the generic Sequence and Selector Composite nodes, two condition nodes and one action node were developed for the DelFly, namely: *greater_than*; *less_than* and; *turnRate*. These behaviour nodes are accompanied by a *Blackboard* which was developed to share information with the BT

The Blackboard architecture implemented for the DelFly to add data to the BT, containing five entries: *window x location* (x), *window response* (σ), *sum of disparity* (Σ), *horizontal disparity difference* (Δ) and *turn rate* (r). This variable map is available to all nodes in the BT, the first four are condition variables which are set at the input of the BT and the last item is used to set the BT output.

The conditions have two internal constants which are set on their initialisation: one sets which Blackboard variable is to be evaluated and the other is the threshold to be tested. As the names suggest, *greater_than* and *less_than* have boolean returns as to whether a variable input is greater than or less than the threshold respectively. The Action node *turnRate* sets the DelFly rudder input directly.

C. User Designed Behaviour Tree

A human designed behaviour was designed which would be used as a control in comparison to the genetically optimised solution. The designed tree had 22 nodes and the structure of the BT as shown in Figure 2. The behaviour is made up of four main parts:

- 🟢 window tracking based on window response and location in frame - try to keep the window in the centre of the frame
- 🔵 go straight default action if disparity very low - helps when looking directly through window into next room
- 🔴 wall avoidance when high disparity - bidirectional turns to avoid collisions with walls, also helps to search for window
- 🟣 action hold when disparity very high - ensures the chosen action is not changed when already evading a wall

After validation of the behaviour, it was observed that for 250 random initialisations in the simulated environment, the behaviour had a resultant success rate of 82%. This behaviour is good but suffers from one main flaw which was observed during the validation. The bidirectional wall avoidance in a square room results that the DelFly can be caught in corners. There are many methods to correct for this behaviour but as this is typical conceptual feature typical with human designed systems we will use this behaviour as a baseline for the comparison later. Figure 3 shows the path of successful and unsuccessful flight initialisations of DelFly with the user-defined behaviour.

IV. Evolutionary Learning Incorporation

EL is a metaheuristic global optimisation method which imitates nature's mechanism of evolution [31]–[33]. Feasible solutions for a particular problem are equivalent to members of a particular species, where the fitness of each individual is measured by some user-defined, problem specific, objective function. For each generation of a Genetic Algorithms (GA), the current population consists of a set of trial solutions currently under consideration. Successful individuals are selected to share their genes to create the next population using the genetic recombination method *crossover*. This combination of genes from the fittest parents is a form of exploitation where

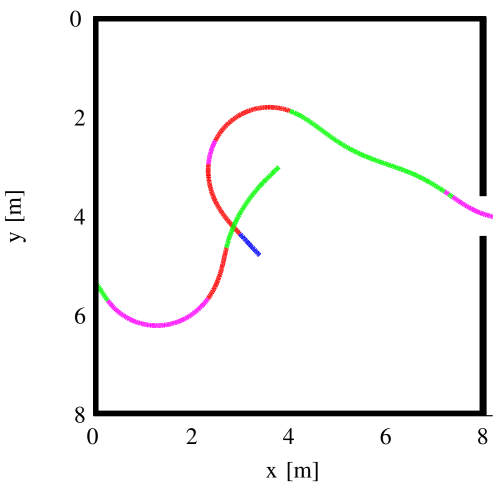


Fig. 3. Path of successful and unsuccessful flight initialisations of DelFly with the user-defined behaviour (top-down view). Colours denote different decision modes: Green - window tracking; Blue - default action in low disparity; Red - wall avoidance; Magenta - action hold

the emerging policy tends towards the best policy. Each child may also be subject to *mutation* where individual parts of their genes are changed. This introduces variation in the population which results in greater exploration of search space [34].

There are many implementations on EL each with a unique method to encode the genetic material in the individuals. GAs use binary strings [31], [32], Genetic Programming (GP) use LISt Processing (LISP) in a graph-like representation [35] and Neuroevolution techniques use ANNs [36]. In this paper we will use the EL to optimise the behaviour for a task using the BT framework. The custom implementation of EL for BTs used in this work is described below.

A. Genetic Operators

a) Initialisation: The initial population of M individuals is generated using the *grow* method [35]. This results in variable length trees where every Composite node is initialised with its maximum number of children and the tree is limited by some maximum tree depth. This provides an initial population of very different tree shapes with diverse genetic material to improve the chance of a good EL search.

b) Selection: A custom implementation of Tournament Selection is used in this paper [37]. This is implemented by first randomly selecting a subgroup of s individuals from the population. This subgroup is then sorted in order of their fitness. If two individuals have the same fitness they are then ranked based on tree size, where smaller is better. The best individual is typically returned unless the second individual is smaller, in which case the second individual is returned. This was done to introduce a constant pressure on reducing the size of the BTs.

c) Crossover: As the tournament selection returns one individual, two tournaments are needed to produce two parents needed to perform crossover. The percentage of the new population formed by Crossover is defined by the Crossover Rate P_c. Crossover is implemented as the exchange of one

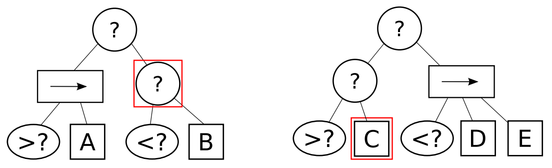


Fig. 4. Sample parent trees with selected nodes for crossover highlighted

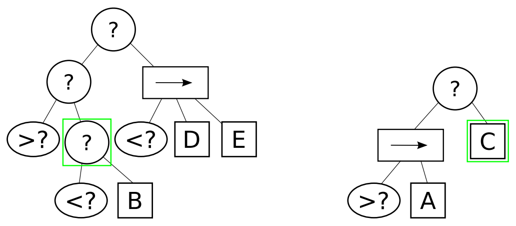


Fig. 5. Children of crossover of parents in Figure 4

randomly selected node from two parents to produce two children. The node is selected totally at random independent of its type or its location in the tree. Figure 4 and Figure 5 graphically shows this process.

d) Mutation: Mutation is implemented with two methods, namely: micro-mutation and; Macro-mutation also referred to as Headless Chicken Crossover [38]. Micro-mutation only affects leaf nodes and is implemented as a reinitialisation of the node with new operating parameters. Macro-mutation is implemented by replacing a selected node by a randomly generated tree which is limited in depth by the maximum tree depth. The probability that mutation is applied to a node in the BT is given by the mutation rate P_m. once a node has been selected for mutation the probability that macro-mutation will be applied rather than micro-mutation is given by the headless-chicken crossover rate P_{hcc}.

e) Stopping Rule: An important parameter in EL is when to stop the evolutionary process. Evolutionary computing is typically computationally intensive due to the large number of simulations required to evaluate the performance of the population of individuals. Placing a limit on the maximum number of generations can help avoid unnecessary long computational time. Additionally, like many learning methods, genetic optimisation can be affected by overtraining.

For these reasons, the genetic optimisation has a maximum number of generations (G) at which the optimisation will be stopped. Additionally, when the trees are sufficiently small to be intelligible, the process can be stopped by the user.

B. Fitness Function

The two main performance metrics used to evaluate the DelFly in the fly-through-window task are: Success Rate and Tree Size. Some secondary parameters that are not vital to the performance of the DelFly that define the suitability of its behaviour are: Angle of Window Entry, Time to Success and Distance from Centre of Window at Fly-Through.

The fitness function was chosen to encourage the EL to converge on a population that flies through the window as often as possible. After trying several differing forms of fitness

Fig. 6. Patterns used to increase the realism and texture of test environment. From left to right: multi-coloured stone for walls, dark grey textured carpet for floors and light grey concrete texture for ceiling

functions a discontinuous function was chosen such that a maximum score is received if the UAV flies through the window and a score inversely proportional to its distance to the window if not successful. The fitness F is defined as:

$$F = \begin{cases} 1 & if\ success \\ \frac{1}{1+3|\mathbf{e}|} & else \end{cases} \quad (7)$$

where success is defined as flying through the window and $\mathbf{e}$ is the vector from the centre of the window to the location of the UAV at the end of the simulation.

V. DelFly Task Optimisation

A. Simulated 3D Environment

The environment chosen to train the UAV in simulation was an $8 \times 8 \times 3m$ room with textured walls, floor and ceiling. A $0.8 \times 0.8m$ window was placed in the centre of one wall. Another identical room was placed on the other side of the windowed wall to ensure the stereo algorithm had sufficient texture to generate matches for the disparity map when looking through the window.

As it is not the focus of this research to focus on the vision systems to allow the stereo vision algorithm some texture, a multi-coloured stone texture pattern was used for the walls, a wood pattern was used for the floor and a concrete pattern used for the ceiling as shown in Figure 6. The identically textured walls ensure that the behaviour must identify the window and not any other features to aid in its task.

B. Experimental Setup

As the DelFly must fly through the window as often as possible, to evaluate this we will simulate the DelFly multiple times per generation to better estimate its generalised performance. Each generation has k simulation runs to evaluate the performance of each individual. Each run is characterised by a randomly initiated location in the room and a random initial pointing direction. Individual initial conditions are held over multiple generations until the elite members of the population (characterised by P_e) are all successful after which the initial condition in question is replaced by a new random initialisation. Each simulation run is terminated when the DelFly crashes, flies through the window or exceeded a maximum simulation time of $100s$.

The characteristic parameters are shown in Table II. These values were chosen after observing the effect of the parameters after several runs of the EL.

TABLE I
Table showing parameter values for the Evolutionary Learning optimisation run

Parameter	Value
Max Number of Generations (G)	150
Population size (M)	100
Tournament selection size (s)	6%
Elitism rate (P_e)	4%
Crossover rate (P_c)	80%
Mutation rate (P_m)	20%
Headless-Chicken Crossover rate (P_{hcc})	20%
Maximum tree depth (D_d)	6
Maximum children (D_c)	6
No. of simulation runs per generation (k)	6

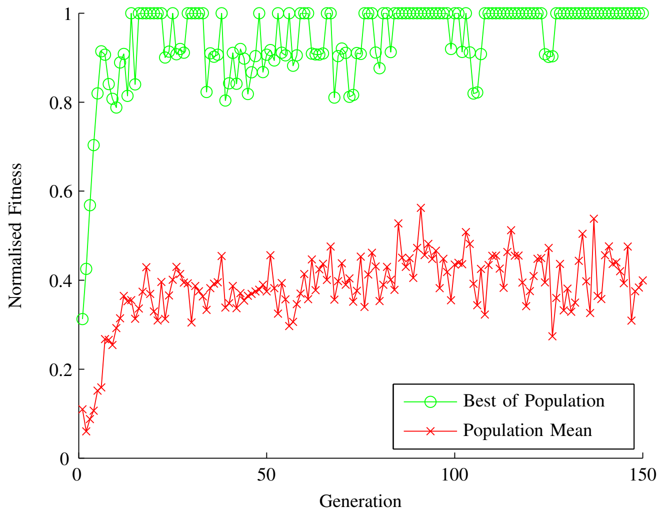


Fig. 7. Progression of the fitness score of the best individual and the mean of the population throughout the genetic optimisation

C. Optimisation Results

The main parameter which dictates the progress of the genetic optimisation is the mean fitness of the individuals in the population. Figure 7 shows the population mean fitness as well as the mean fitness of the best individual in each generation. It can be seen in Figure 7 that at least one member of the population is quickly bred to fly through the window quite often. Additionally, as the the generations progress and new initialisations are introduced the trees have to adjust their behaviour to be more generalised. The mean fitness also improves initially then settles out at around the 0.4 mark, the fact that this value doesn't continue to increase suggests that the genetic diversity in the pool is sufficient.

The other main parameter which defines the proficiency of the BTs is the tree size. The mean tree size of the population as well as the tree size of the best individual from each generation is shown below in Figure 8.

This figure shows that the average tree size began at about 5000 nodes and initially increases to 7000 before steadily dropping to around 1000 nodes around generation 50. The trees size then slowly continues to reduce in size and eventually drops below 150 nodes. The best individual in the population oscillated around this mean value. The best individual after 150 generations had 32 nodes. Pruning this final BT,

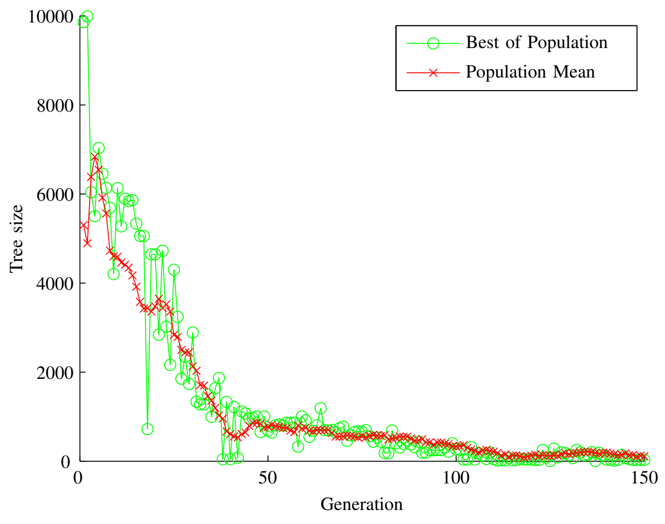


Fig. 8. Progression of the number of nodes in the best individual and the mean of the population

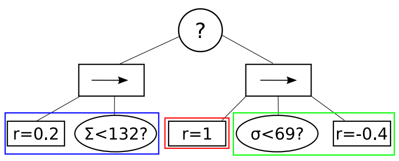


Fig. 9. Graphical depiction of genetically optimised BT. Colours highlight different phases of the flight x is the position of the centre of the window in frame, σ is window response value, Σ is sum of disparity and Δ is the horizontal difference in disparity

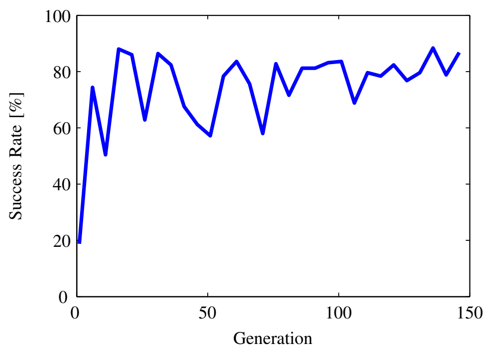


Fig. 10. Progression of validation score of the best individual of each generation

removing redundant nodes that have no effect on the final behaviour, resulted in a tree with 8 nodes. The structure of the tree can be seen graphically in Figure 9.

The optimised BT was put through the same validation set as used with the user-defined resulting in a success rate of 88%. Figure 10 shows the progression of the validation success rate for the best individual of each generation. It can be seen that the score quickly increases and oscillates around about 80% success. In early generations the variation of success rate from one generation to the next is larger than later generations.

Figure 8 and Figure 10 suggest that that the population quickly converges to a viable solution and then continues to rearrange the tree structure to result in ever smaller trees.

TABLE II
SUMMARY OF VALIDATION RESULTS

Parameter	user-defined	genetically optimised
Success Rate	82%	88%
Tree size	26	8
Mean flight time $[s]$	32	40
Mean approach angle $[°]$	21	34
Mean distance to centre $[m]$	0.08	0.15

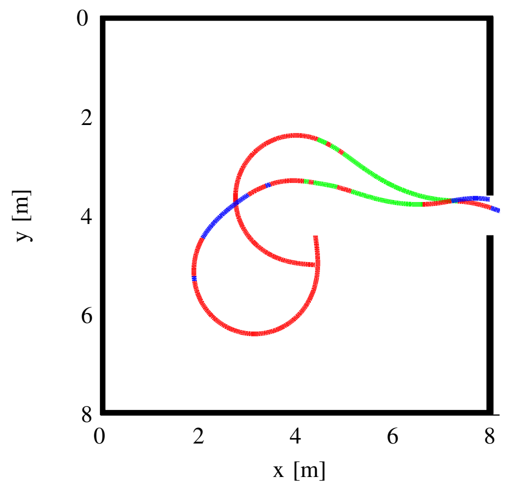


Fig. 11. Path of successful and unsuccessful flight initialisations of DelFly with the genetically optimised behaviour (top-down view). Colours denote different decision modes: Green - window tracking; Blue - default action in low disparity; Red - wall avoidance

The fact that the best individual of each population does not improve much above the 80% mark possibly identifies that the method used to expose the population to a generalised environment is not sufficient. A method to make the initial conditions more ”difficult” is by employing co-evolution of the initialisation as well as the robotic behaviour. This predator-prey type co-evolution may improve results. Alternatively, the fact that the behaviour does not continue to improve may also indicate that the sensory inputs used b the DelFly are not sufficient.

The performance characteristics of the best individual from the optimisation as compared to those from the user-defined BT is summarised below in Table II. The optimised BT has slightly higher success rate than the user-defined BT but with significantly less nodes. The mean fitness is also slightly higher with the optimised BT.

The successful flight shown in Figure 11 shows that the behaviour correctly avoids collision with the wall, makes its way to the centre of the room and then tracks into the window. Analysing the BT from Figure 9, the logic to fly through the window is very simple, the tree can be separated into three phases:

- 🔵 slight right turn default action when disparity low
- 🔴 max right turn to evade walls if disparity high (unidirectional avoidance)
- 🟢 if window detected make a moderate left turn

This very simple behaviour seems to have very good success however Figure 11 also highlights one pitfall of this solution.

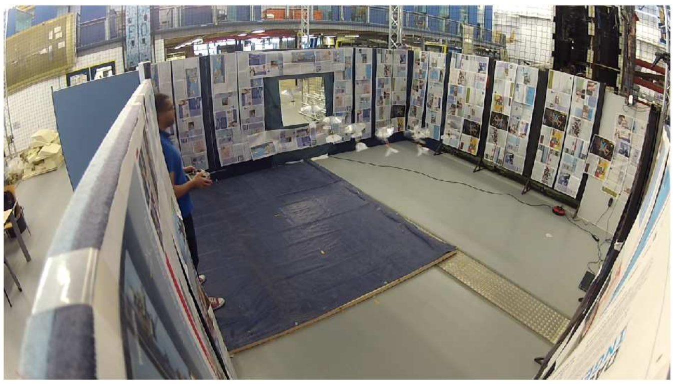


Fig. 12. Photograph showing the room environment used to test the DelFly Explorer for the fly-through-window task. Inset is collage of DelFly as it approaches and flies through window

As the behaviour doesn't use the location of the window in the frame for its guidance it is possible to drift off centre and lose the window in frame and enter a wall avoidance turn quite close to the wall resulting in a collision.

These results show that based on the given fitness function and optimisation parameters the genetic optimisation was very successful. The resultant BT was both smaller and better performing than the user-defined tree.

VI. DelFly Onboard Flight Testing

The BT was implemented on the camera module of the DelFly Explorer which is equipped with a STM32F405 processor operating at $168MHz$ with $192kB$ RAM. This processor is programmed in the `C` programming language as opposed to the `C++` implementation used in simulation. As a result a slightly simplified version of the BT system was implemented onboard.

The BT node is placed in series with the stereo vision and window detection algorithms and was found to run at $\sim 12Hz$. The commands were sent from the camera module to the DelFly Explorer flight control computer using serial communication. The DelFly flight control computer implements these commands in a control system operating at $100Hz$.

A. Test 3D Environment

The environment designed to test the UAV was a $5 \times 5 \times 2$ m room with textured walls. The floor was simply a concrete floor and the ceiling was left open. A 0.8×0.8 m window was placed in the centre of one wall. The area behind the window was a regular textured area. As the focus of this work is on investigating the development behaviour of the DelFly and not on the vision systems themselves, we added artificial texture to the environment to ensure we had good stereo images from the DelFly Explorer onboard systems. This texture was in the form of newspapers draped over the walls at random intervals. Sample photographs of the room can be seen below in Figure 12.

B. Experiment Set-up

At the beginning of each run, the DelFly will initially be flown manually to ensure it is correctly trimmed for flight. It will then be flown to a random initial position and pointing direction in the room. At this point the DelFly will be commanded to go autonomous where the DelFly flight computer implements the commands received from the BT. The flight will continue until the DelFly either succeeds in flying through the window, crashes or the test takes longer than $60s$. As the BT controls the horizontal dynamics only, the altitude is actively controlled by the user during flight, this was maintained around the height of the centre of the window.

All flights are recorded by video camera as well as an Optitrack vision based motion tracking system [39]. The motion tracking system was used to track the DelFly as it approached and flew through the window to determine some of the same metrics of performance that were used in simulation. As a result, information on the success rate, flight time, angle of approach and offset to the centre of the window can be determined.

VII. Flight Test Results

The flight speed of the DelFly was set to $\sim 0.5m/s$, the same as was used in simulation apart from this however there were significant differences observed between the DelFly simulated in SmartUAV and the DelFly used in the flight tests. The most significant was that the maximum turn radius was $\sim 0.5m$ much smaller than the $1.25m$ as in simulation. Additionally, the ailerons on the DelFly Explorer had a faster response rate than the low pass filter set on the rudder dynamics in simulation. It was also observed that the aileron deflection was not symmetrical, the DelFly would turn more effectively to the right than it did to the left. Aileron actuation would also result in a reduction in thrust meaning that active altitude control was required from the user throughout all flights. It was also observed that there were light wind drafts observed around the window which affected the DelFly's flight path. This draft would typically slow the DelFly forward speed slightly and push the DelFly to one side of the window.

With these significant differences between the model used to train the BTs and the real DelFly there as a clear reality gap present. Initially both behaviours were not successful in flying through the window, the behaviour thresholds had to be adjusted to ensure that the DelFly would behave similar to that in simulation. This was done first for the user-defined behaviour, the updated behaviour can be seen in Figure 13. It was required to adjust the turn rate set points to try to achieve a more symmetrical wall avoidance behaviour. Due to the different environment in reality the window response at which the DelFly started its window tracking was also raised slightly. The threshold at which the DelFly started its wall avoidance was also changed to ensure the DelFly could evade walls. These changes helped push the behaviour in reality towards that observed in simulation. In total, it took about 8 flights of about 3 minutes each to tune the parameters of the behaviour.

A similar process was done for the genetically optimised behaviour. As the parameters for the wall avoidance was the same for both behaviours, the changes to this could be done before any flight tests. As a result, only the window tracking turn rate, default turn rate and the window response values

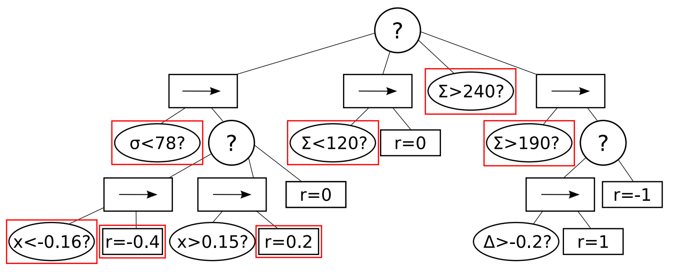


Fig. 13. Graphical depiction of user-defined BT after modification for real world flight. Red boxes highlight updated nodes. x is the position of the centre of the window in frame, σ is window response value, Σ is sum of disparity and Δ is the horizontal difference in disparity

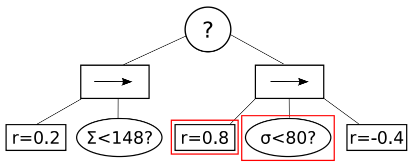


Fig. 14. Graphical depiction of genetically optimised BT after modification for real world flight. Red boxes highlight updated nodes. x is the position of the centre of the window in frame, σ is window response value, Σ is sum of disparity and Δ is the horizontal difference in disparity

TABLE III
SUMMARY OF FLIGHT TEST RESULTS

Parameter	user-defined	genetically optimised
Success Rate	46%	54%
Mean flight time $[s]$	12	16
Mean approach angle $[°]$	16	37
Mean distance to window centre $[m]$	0.12	0.12

had to be tuned. These parameters took about 3 flights to tune to result in behaviour similar to that seen in simulation. The updated behaviour can be seen in Figure 14.

After an initial training session where the thresholds were re-optimised to real flight, 26 test flights were conducted for both the user-defined behaviour as well as the genetically optimised BT. The results of the tests are summarised below in Table III.

It can be seen that the success rate of both behaviours is reduced success rate but the other performance parameters are similar to that seen in simulation. The relative performance of the behaviours is also similar to that seen in simulation. The mean flight time of the behaviours was reduced but notably the relative flight times of the behaviours is the same as seen in simulation. The reduction in the time to success can be explained by the reduced room size and increased turn rate of the DelFly seen in reality as opposed to that in simulation.

The mean angle of window entry is also similar to that observed in simulation. The mean distance to the centre of the window was higher for the user-defined behaviour than seen in simulation. This can be as a result of the drafts seen around the window pushing the DelFly to the edges of the window in the last phase of the flight when the window was too close to be in view.

The user-defined behaviour showed similar failure as seen in simulation characterised by being caught in corners, this happened 4/26 flights for the user-defined behaviour but not

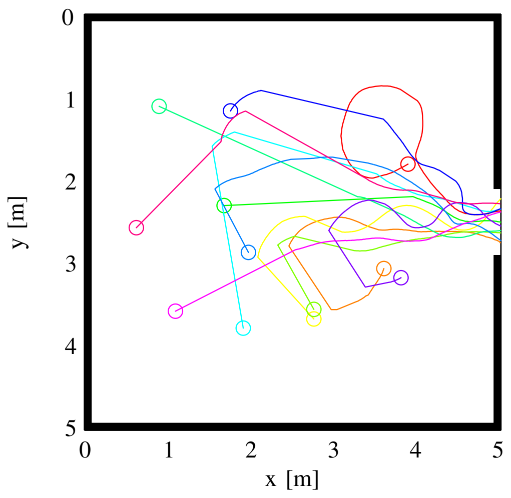


Fig. 15. Flight path tracks of the last 7s of all successful flights for the user-defined behaviour. Circle represents start of path

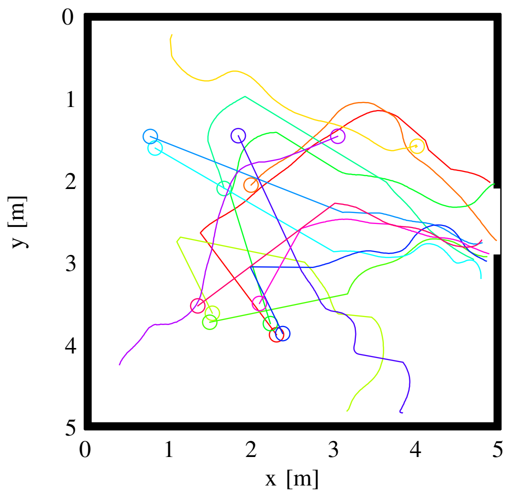


Fig. 16. Flight path tracks of the last 7s of all unsuccessful flights for the user-defined behaviour. Circle represents start of path

once in the genetically optimised behaviour. Figure 15 and Figure 16 show the last $7s$ of the user-defined behaviour for all flights grouped in successful and unsuccessful tests respectively. The Optitrack flight tracking system did not successfully track the DelFly in all portions of the room resulting in some dead areas but did accurately capture the final segment of the window approach.

These plots show that the DelFly tried to approach and fly through the window from various areas of the room at various approach angles. Approaches from areas of high approach angle typically resulted in a failed flight as the DelFly would hit the edge of the window. Additionally, the crashes in the wall due to being caught in corners can also be seen. Figure 17 shows one full successful and unsuccessful flight of the DelFly user-defined behaviour.

Similarly, Figure 18 and Figure 19 show the successful and unsuccessful flights of the genetically optimised behaviour as captured from the optitrack system. In these figures it can be seen that the flight tracks of genetically optimised behaviour

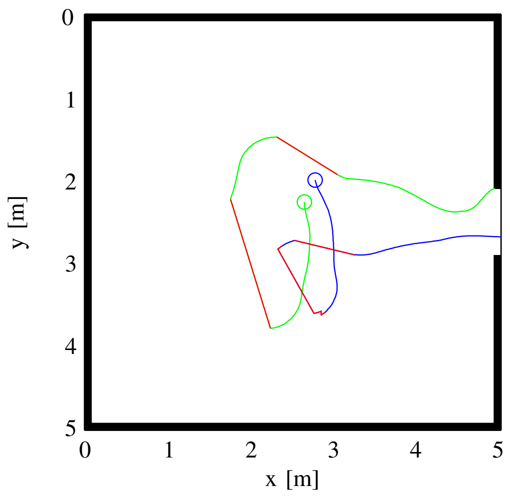


Fig. 17. Flight path tracks showing one complete successful (blue) and unsuccessful (green) flight for the genetically optimised behaviour. Circle represents start location of test. Red track shows area where tracking system lost lock of the DelFly

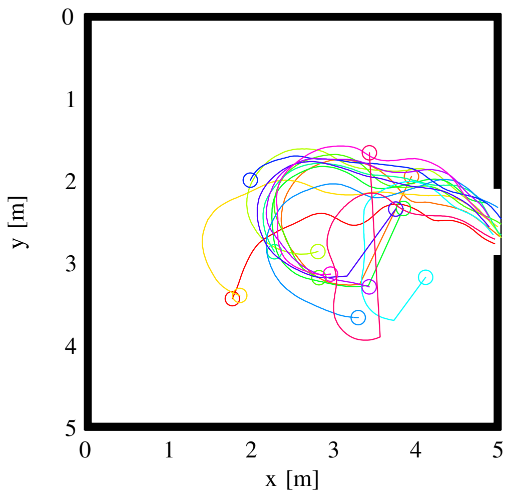


Fig. 18. Flight path tracks of the lest 7s of all successful flights for the genetically optimised behaviour. Circle represents start of path

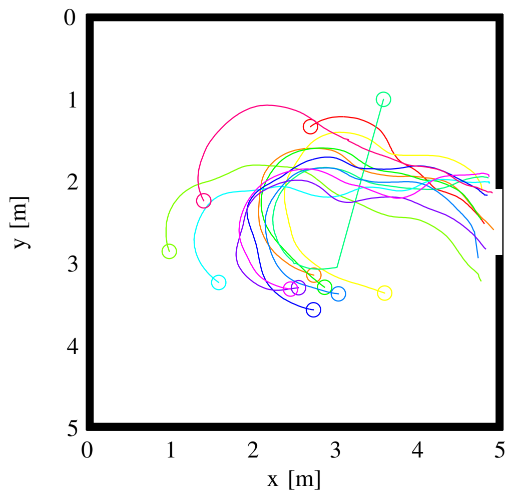


Fig. 19. Flight path tracks of the lest 7s of all unsuccessful flights for the genetically optimised behaviour. Circle represents start of path

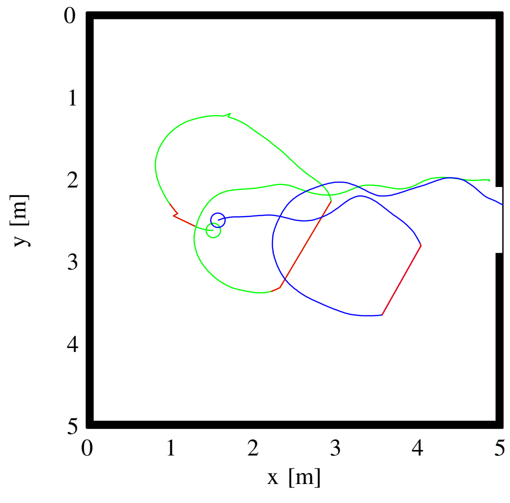


Fig. 20. Flight path tracks showing one complete successful (blue) and unsuccessful (green) flight for the genetically optimised behaviour. Circle represents start location of test. Red track shows area where tracking system lost lock of the DelFly

are tightly grouped with the same behaviour repeated over multiple flights. The DelFly always approaches from about the centre of the room with a coordinated left-right turn described earlier. It can be seen that some of the unsuccessful flights occur when the DelFly makes an approach from farther way than normal so the coordination of the left-right turning is out of sync causing the DelFly to drift off course and hit the window edge. Figure 20 shows one entire successful and unsuccessful flight of the genetically optimised behaviour in more detail. The typical failure mode was turning into the edge of the window in the final phase of the flight.

The failure mode of hitting into the window edge for both behaviours can be in part the result of the drafts observed around the window or in part due to the lack of detailed texture around the window. These external factors would affect the two behaviours equally so would not affect the comparison of behaviours.

The fact that the behaviours where not initially able to fly through the window and were able to fly through more than 50% of the time after user optimisation shows that the reality gap was actively reduced by the user. These results show that it is feasible to automatically evolve behaviour on a robotic platform in simulation using the BT description language. This method gives the user a high level of understanding of the underlying behaviour and the tools to adapt the behaviour to improve performance and reduce the reality gap. Using this technique an automated behaviour was shown to be as effective as a user-defined system in simulation with similar performance on a real world test platform.

VIII. Conclusion

We conclude that the increased intelligibility of the Behaviour Tree framework does give a designer increased understanding of the automatically developed behaviour. The low computational requirements of evaluating the Behaviour Tree framework makes it suitable to operate onboard platforms with limited capabilities as it was demonstrated on the $20g$ DelFly Explorer flapping wing UAV. It was also demonstrated that the Behaviour Tree framework provides a designer with the tools to identify and adapt the learnt behaviour on a real platform to reduce the reality gap when moving from simulation to reality.

Future work will investigate further into optimising the parameters of the Evolutionary Learning used in this paper. Multi-objective fitness functions and co-evolution of behaviour and simulated environment are interesting directions to investigate. Additionally, work will be done on investigating how Behaviour Trees scale within Evolutionary Learning, both in terms of Behaviour node types but also in task complexity.

ACKNOWLEDGMENT

The authors would like to thank Christophe de Wagter for his assistance with the SmartUAV simulation platform and Erik-Jan van Kampen and Luís Simões for useful feedback on this paper.

REFERENCES

[1] S. Nolfi and D. Floreano, *Evolutionary Robotics: The Biology, Intelligence and Technology*. Cambridge, MA, USA: MIT Press, 2000.
[2] J. C. Bongard, "Evolutionary Robotics," *Communications of the ACM*, vol. 56, no. 8, pp. 74–83, Aug. 2013.
[3] S. Nolfi, "Power and the Limits of Reactive Agents," *Neurocomputing*, vol. 42, no. 1-4, pp. 119–145, Jan. 2002.
[4] D. Floreano and F. Mondada, "Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural-Network Driven robot," *From Animals to Animats*, 1994.
[5] S. Nolfi, D. Floreano, O. Miglino, and F. Mondada, "How to Evolve Autonomous Robots: Different Approaches in Evolutionary Robotics," in *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, R. A. Brooks and P. Mates, Eds. Cambridge, MA, USA: MIT Press, 1994, pp. 190–197.
[6] N. Jakobi, P. Husbands, and I. Harvey, "Noise and the Reality Gap: The Use of Simulation in Evolutionary Robotics," in *Advances in Artificial Life*, F. Morán, A. Moreno, J. J. Merelo, and P. Chacón, Eds. Granada, Spain: Springer Berlin Heidelberg, Jun. 1995, pp. 704–720.
[7] O. Miglino, H. H. Lund, and S. Nolfi, "Evolving Mobile Robots in Simulated and Real Environments," *Artificial life*, vol. 2, no. 4, pp. 417–34, Jan. 1995.
[8] L. Meeden, "Bridging the Gap Between Robot Simulations and Reality with Improved Models of Sensor Noise," in *Genetic Programming*. Madison, WI, USA: Morgan Kaufmann, Jul. 1998, pp. 824–831.
[9] J. C. Bongard, V. Zykov, and H. Lipson, "Resilient Machines through Continuous Self-Modeling," *Science*, vol. 314, no. 5802, pp. 1118–21, Nov. 2006.
[10] J. C. Zagal and J. Ruiz-del Solar, "Combining Simulation and Reality in Evolutionary Robotics," *Journal of Intelligent and Robotic Systems*, vol. 50, no. 1, pp. 19–39, Mar. 2007.
[11] S. Koos, J.-B. Mouret, and S. Doncieux, "The Transferability Approach: Crossing the Reality Gap in Evolutionary Robotics," *Transactions on Evolutionary Computation*, vol. 17, no. 1, pp. 122–145, Feb. 2013.
[12] C. Hartland and N. Bredèche, "Evolutionary Robotics, Anticipation and the Reality gap," in *Robotics and Biomimetics*. Kunming, China: IEEE, Dec. 2006, pp. 1640–1645.
[13] P. Petrovi, "Evolving Behavior Coordination for Mobile Robots using Distributed Finite-State Automata," in *Frontiers in Evolutionary Robotics*, H. Iba, Ed. Intech, 2008, no. April, ch. 23, pp. 413–438.
[14] L. König, S. Mostaghim, and H. Schmeck, "Decentralized Evolution of Robotic Behavior using Finite State Machines," *International Journal of Intellegent Computing and Cybernetics*, 2009.
[15] A. Pintér-Bartha, A. Sobe, and W. Elmenreich, "Towards the Light - Comparing Evolved Neural Network Controllers and Finite State Machine Controllers," in *10th International Workshop on Intellegent Solutions in Embedded Systems*. Klagenfurt, Austira: IEEE, Jul. 2012, pp. 83–87.
[16] A. Valmari, "The State Explosion Problem," in *Lectures on Petri Nets I: Basic Models*, W. Reisig and G. Rozenberg, Eds. Springer Berlin Heidelberg, 1998, pp. 429–528.
[17] I. Millington and J. Funge, *Artificial Intelligence for Games*, 2nd ed. Morgan Kaufmann, 2009.
[18] R. G. Dromey, "From Requirements to Design: Formalizing the Key Steps," in *First International Conference on Software Engineering and Formal Methods*. Brisbane, Australia: IEEE, Sep. 2003, pp. 2–11.
[19] A. J. Champandard, "Behavior Trees for Next-Gen Game AI," 2007. [Online]. Available: http://aigamedev.com/open/articles/behavior-trees-part1/
[20] F. W. P. Heckel, G. M. Youngblood, and N. S. Ketkar, "Representational Complexity of Reactive Agents," in *Computational Intelligence and Games*. IEEE, Aug. 2010, pp. 257–264.
[21] C. Lim, R. Baumgarten, and S. Colton, "Evolving Behaviour Trees for the Commercial Game DEFCON," in *Applications of Evolutionary Computation*. Springer Berlin Heidelberg, 2010, pp. 100–110.
[22] D. Perez, M. Nicolau, M. O'Neill, and A. Brabazon, "Reactiveness and Navigation in Computer Games: Different Needs, Different Approaches," in *IEEE Conference on Computational Intelligence and Games (CIG'11)*. Seoul, South Korea: IEEE, Aug. 2011, pp. 273–280.
[23] R. C. Julian, C. J. Rose, H. Hu, and R. S. Fearing, "Cooperative Control and Modeling for Narrow Passage Traversal with an Ornithopter MAV and Lightweight Ground Station," in *International conference on Autonomous agents and multi-agent systems*. St. Paul, Minnesota, USA: IFAAMAS, May 2013, pp. 103–110.
[24] G. C. H. E. de Croon, K. M. E. de Clercq, R. Ruijsink, B. D. W. Remes, and C. de Wagter, "Design, Aerodynamics, and Vision-Based Control of the DelFly," *International Journal of Micro Air Vehicles*, vol. 1, no. 2, pp. 71–98, 2009.
[25] C. de Wagter, S. Tijmons, B. D. W. Remes, and G. C. H. E. de Croon, "Autonomous Flight of a 20-gram Flapping Wing MAV with a 4-gram Onboard Stereo Vision System," in *IEEE International Conference on Robotics and Automation*, 2014.
[26] D. Scharstein and R. Szeliski, "A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms," *International journal of computer vision*, vol. 47, no. 1, pp. 7–42, 2002.
[27] F. C. Crow, "Summed-Area Tables for Texture Mapping," *SIGGRAPH Computer Graphics*, vol. 18, no. 3, pp. 207–212, Jul. 1984.
[28] P. Viola and M. Jones, "Robust Real-Time Object Detection," *International Journal of Computer Vision*, 2001.
[29] C. de Wagter, A. A. Proctor, and E. N. Johnson, "Vision-Only Aircraft Flight Control," in *Digital Avionics Systems Conference*, vol. 2. Indianapolis, IN, USA: IEEE, Oct. 2003.
[30] M. Amelink, M. Mulder, and M. M. van Paassen, "Designing for Human-Automation Interaction: Abstraction-Sophistication Analysis for UAV Control," in *International MultiConference of Engineers and Computer Scientists*, vol. I. Hong Kong: IAE, Mar. 2008.
[31] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.
[32] J. R. Koza, "Genetic Algorithms and Genetic Programming," 2003. [Online]. Available: http://www.smi.stanford.edu/people/koza/
[33] F. S. Hiller and G. J. Lieberman, *Introduction to Operations Research*, 9th ed. McGraw-Hill, 2010.
[34] M. Melanie and M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press, 1998.
[35] J. R. Koza, "Genetic Programming as a Means for Programming Computers by Natural Selection," *Statistics and Computing*, vol. 4, no. 2, pp. 87–112, Jun. 1994.
[36] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: from Architectures to Learning," *Evolutionary Intelligence*, vol. 1, no. 1, pp. 47–62, Jan. 2008.
[37] B. L. Miller and D. E. Goldberg, "Genetic Algorithms, Tournament Selection, and the Effects of Noise," *Complex Systems*, vol. 9, no. 95006, pp. 193–212, 1995.
[38] P. J. Angeline, "Subtree Crossover: Building Block Engine or Macromutation," in *Genetic Programming*, Stanford University, CA, USA, Jul. 1997, pp. 9–17.
[39] NaturalPoint, Inc, "Optitrack," 2014. [Online]. Available: http://www.naturalpoint.com/optitrack/

Part II

Thesis

Chapter 2

Literature Review

In this chapter, we will present some detailed background into BTs as well as some popular EL techniques which can be used to automatically develop these trees to solve specific tasks. We will also shed some light on previous work done on developing autonomous behaviour on previous research on flapping wing Unmanned Aerial Vehicles (UAVs). Additional literature covering an overview of forms of Artificial Intelligence, learning and decision making techniques, planning methods and popular Flight Control System frameworks can be found in Appendix A.

2-1 Behaviour Trees

In the early 2000's, computer game designers developing behaviour for the NPCs within their games were asking for a behaviour management system with capabilities that no single method could deliver (Champandard, 2007). They found their solution in a method developed originally by Dromey (2003) to describe complex system requirements called the Behaviour Tree. BTs combine many of the strengths of other methods and as a result quickly became popular in the gaming industry, most notably in Halo 2 (Isla, 2005). Some of the notable properties of BTs are:

- Use of hierarchy in Hierarchical Finite State Machine (HFSM) facilitates reusability of code base and expandability of character behaviour
- Expressing behaviour in terms of a set tasks as in Hierarchical Task Network (HTN) allows for easier decomposition of behaviour
- The simplicity of decision trees makes recursive decomposition easy to implement
- Graphical state charts found in FSM give users a good overview of the NPC behaviour reducing representational complexity

BTs are depth-first, ordered Directed Acyclic Graphs (DAGs) used to represent a decision process. DAGs are composed of a number of nodes with directed edges. Each edge connects one node to another such that starting at the root there is no way to follow a sequence of edges to return to the root. This behaviour allows for more compact behavioural representation than the decision tree. Unlike FSMs, BTs consider achieving a goal by recursively simplifying the goal into tasks similar to that seen in the HTN. This hierarchy and recursive action make the BT a powerful way to describe complex behaviour. The mechanics of the BT are described below.

2-1-1 Syntax & Semantics

A BT is syntactically represented as a rooted tree structure, constructed from a variety of nodes each with its individual internal function but all nodes have the same external interface. Each node in a BT has a return status, generally either *Success* or *Failure*. The status Success is the status of a node that has been successfully executed. Inversely, the status Failure is used when a node has failed during execution. This however does not define the condition under which the node failed, some implementations include an *Exception* or *Error* status to provide this information.

Basic BTs are made up of three kinds of nodes: *Conditions*, *Actions* and *Composites* (Champandard, 2007). Conditions and Actions make up the leaf nodes of the BT whilst the branches consist of Composite nodes. Conditions test some property of the environment returning Success if the conditions are met and returns Failure otherwise. The agent acts on its environment through Action nodes. Actions can be blocking (node only returns when action is successfully completed) or non-blocking but typically always return Success. As a result, Actions usually follow a Condition node to determine if the Action is applicable. Alternatively, if some fixed precondition needs to be applied to a node, the condition can be checked within the node itself returning Failure is the condition is not meet or if there is some internal time-out.

Leaf nodes must be individually developed to perform specific tasks but can be reused readily in the tree as required. Composite nodes however are not typically platform dependant and can be reused in any BT. All the nodes in the BT use a similar interface so that arbitrary combination of these nodes is possible in the BT without knowledge of any other part of the BT making BTs modular and reusable. A sample BT highlighting the graphical representation of the different nodes can be seen in Figure 2-1.

As the branches of the BT, Composite nodes determine how the BT is executed. We will only consider *Sequences* and *Selectors* in this paper although many others are used in practice. The most popular Composite nodes shown below. The *Root* node of a BT is typically a Selector node that has no parent.

Selectors return Success when one of its children return Success and Failure when all of its children return Failure. The Selector typically tests its child nodes sequentially along a certain priority, graphically, this priority is defined in the ordering from left to right as shown in Figure 2-2a. Selectors have a sort of inherent fail-safe as the node will test its children in sequence until it finds an action that is applicable to the current state of the environment.

A Sequence will return Failure when one of its children fails and Success if all of its children return Success. This is the operational opposite of a Selector. Again, a sequence will run its

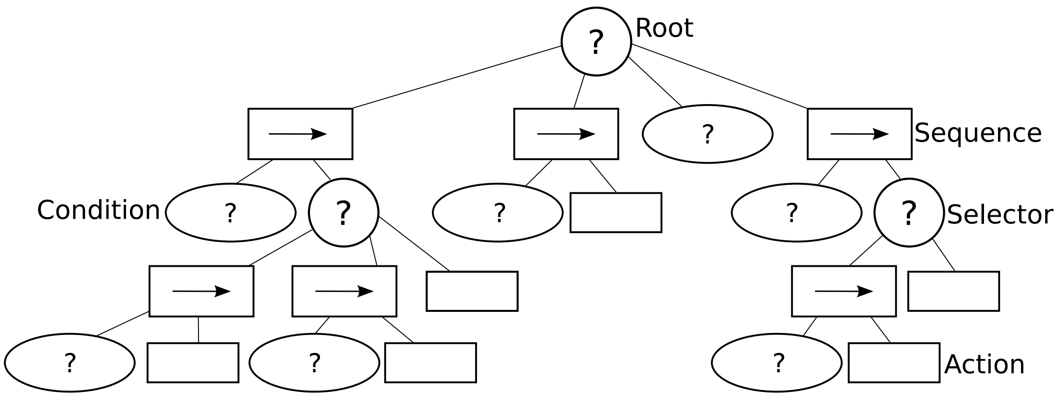
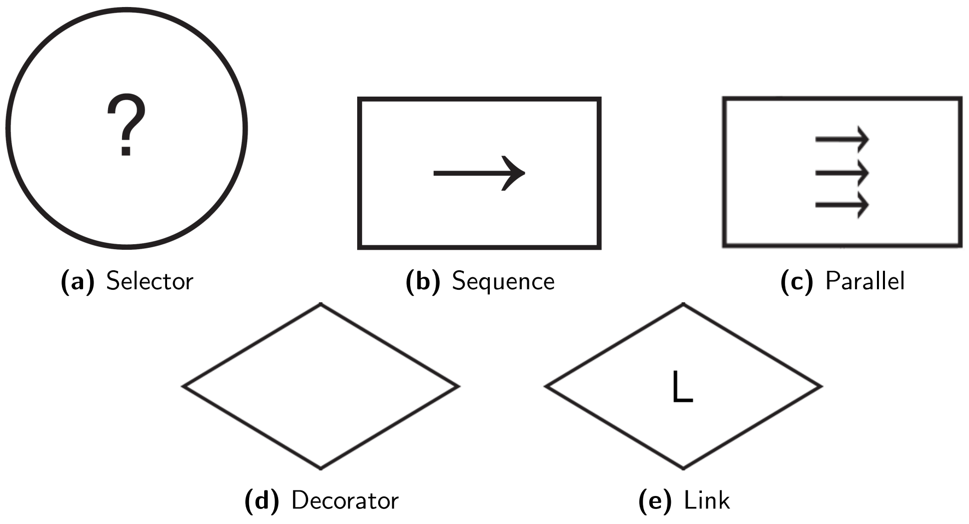


Figure 2-1: Graphical depiction of a Behaviour Tree highlighting node types



(a) Selector **(b)** Sequence **(c)** Parallel

(d) Decorator **(e)** Link

Figure 2-2: Graphical representation of popular BT Composite nodes

children in a prioritised manner, graphically represented left to right as shown in Figure 2-2b. Sequences are useful to perform combinations of tasks that are interdependent.

The Parallel differs from other Composites in that it runs all of its children concurrently as if each were executed in its own thread. The return conditions of the Parallel can be altered depending on the required behaviour. It could be made to act as a Selector or as a Sequence or some hybrid where some number of children must return either Success or Failure to trigger a return of the parent Parallel. Parallels are used to run concurrent branches of the BT but are also commonly used to check whether the conditions for an action are continually met while the action is running. Its node symbol can be seen in Figure 2-2c.

Decorator nodes typically have only one child and are used to enforce a certain return state or to implement timers to restrict how often the child will run in a given amount of time or how often it can be executed without a pause. The decorator node is shown in Figure 2-2d.

The Link node, is used in a BT to reference and execute another BT in place of the node. The name of the tree executed is written beneath the node. The success and failure of a Link node is based on the referenced BT. This node adds the ability to make modular BTs which supports reuse of behaviour. This node can be considered as a special type of decorator and its symbol is shown in Figure 2-2e.

2-1-2 Execution

The execution of the behaviour tree is referred to as a *tick*. A tick starts from the root node which is typically a selector node and evaluates down the tree starting at the left most node. An execution is complete when a branch of the root node returns success or all of its branches return failure. Let us evaluate an example from Millington & Funge (2009), seen in Figure 2-3, to illustrate the execution of a tick. This example shows how a BT can be used to open a door under differing conditions. This example shows how the modularity of the BT can be used to good effect. The only actions implemented are *Move* and *Barge Door* but *Move* is used three times in the tree with a different set point to move to. The nodes were not required to be changed at all depending on their location in the tree.

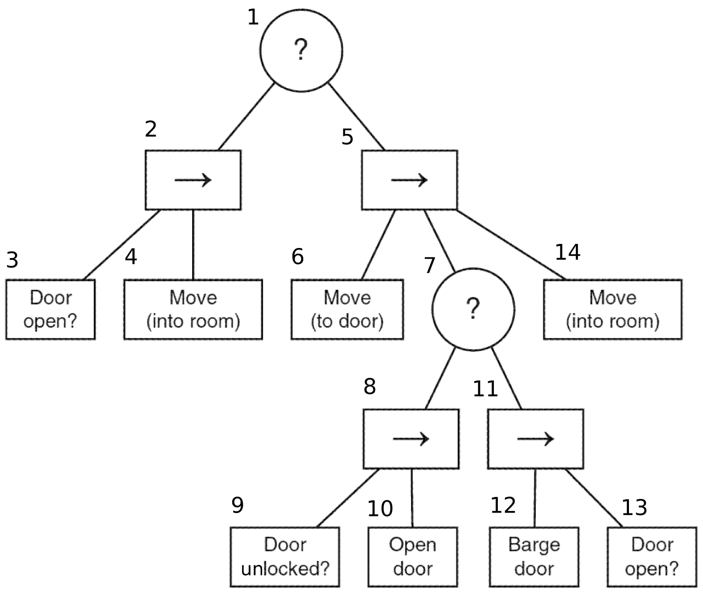


Figure 2-3: An example of BT used to define the behaviour to open a door Millington & Funge (2009) (modified for clarity)

Let us first assume that the Action nodes in the tree are blocking. Executing the tree with the situation that the door is open we start at the Root node and move to the first node on the left. We then move to (3) which will evaluate Success and return this value to its parent. The Sequence will then continue to its next child which is an Action which will return Success to its parent upon completion. The Sequence node has no more children and all of its children where successful so it returns Success to its parent ending the tick of the tree.

Evaluating another situation, let us assume the door is closed but unlocked. We again start at the Root node and evaluate down to (3) which will return Failure to its parent Sequence which will also return Failure to the Root node. The Root node will then evaluate its next child, moving down through (5) to (6) which evaluates Success once the agent has moved to the door. The Sequence (5) moves to its second child which is a Selector node which in turn evaluates its child a Sequence moving onto (9). As the door is open,0. (9) will return Success and the Sequence (8) will evaluate (10). Once the door is open (10) returns Success to (8) which returns Success to (7) which in turn immediately returns Success to (5) which then evaluates (14). Upon conclusion Success is passed up the tree to the Root ending the tick.

In the event that the door is closed and locked a similar execution will occur except (9) will return Failure and the execution will move to (12). If the barge action successfully opened the door the execution will proceed back up the tree to (14). Once the action is complete the execution returns to the Root node with Success.

This execution procedure is the main difference between BTs and Decision Trees. Decision Trees typically traverse from some root node to a leaf node by a series of binary decisions never traversing back up the tree. BTs contain decision nodes that perform compound decisions and the traversal the decision process can travel back up the tree and down a different branch. This traversal allows for the same behaviour to be expressed with little repetition.

2-1-3 Advantages

BTs consider achieving a goal based on a recursive decomposition of that goal into tasks. This design ethos inherently facilitates the generation of goal based behaviour. This is different to the approach as seen with FSM where the focus is based on the mapping of the state-action set and the designer has to mould this mapping to lead the agent to some desired state. This also prevents BTs from being subject to the state explosion phenomenon which affects FSMs.

The representational complexity that makes the use of FSM difficult to use for complex behaviour can be addressed in practice by introducing hierarchy resulting in a HFSM. While this approach is effective in reducing the representational complexity of the State Machine it is still not a very reusable framework. The transitions from any state to another still have to be explicitly described in the HFSM meaning that reusing particular state in another area of the behaviour requires an explicit description of its transitions. BTs implicitly describe its behaviour in the structure of the tree rather than in node-wise state transitions as in FSM or HFSM. This makes the BT approach more modular and reusable resulting in a more easily scalable design framework.

As the BT evaluates its branches in a prioritised manner which is defined by the structure of the tree, BTs inherently facilitate multiple methods to achieve the same behaviour. If one behaviour is not successful simple move to the next suitable behaviour in the priorities network, this creates a form of contingency planning.

BTs represent a very simple form of planning but as they do not explicitly consider the future effects of current actions but rather rely on a preset collection of behaviours, they fall under the definition of *reactive planners*. Reactive planners, or Dynamics Planners as they are also known, are not planners in the traditional sense as standard reactive systems do not contain a model of the environment and hence do not consider the future effects of a set of actions. Instead they are based on a predefined set of actions to perform based on the current environment state and the goal to be achieve. As BTs do not explicitly consider the effects of actions but rather only observe the current state, they do not require an internal model of the environment and are therefore very simple and are relatively computationally efficient to evaluate. BTs are interesting to be used on small robotic platforms where computational power is low a

Work has shown that agents with this form of reactive architecture can exhibit complex behaviours in challenging environments (Nolfi, 2002). Arkin (1998); Nolfi & Floreano (2000); Floreano & Mondada (1994) have also shown that combination of actions and in a prioritised or hierarchical network can be used to develop complex agent behaviour. This type of planning has very low computational requirements and can be run very quickly to operate in highly dynamic environments. A more detailed literature review of planning methods can be found in Appendix A-2.

2-1-4 Limitations

Although all FSM behaviour can be encoded in a BT framework, one disadvantage of the BTs framework is that handling discrete events is not inherently implemented in the structural framework as is the case with an event driven FSM. One way to reduce this problem is to include an internal cache to store these events for the BT to handle at some later tick the same way it would any other sensory input. Alternately, an event handler can be designed to call a BT to handle an event concurrently to the main behaviour.

2-1-5 Recent Advances

Two independent research groups have used different practical implementations of EL to evolve BTs to complete complex tasks of an agent in a computer game (Lim et al., 2010; Perez et al., 2011). The earliest was the work of Lim et al. (2010) which used this architecture to develop an AI agent to compete in the strategy game DEFCON. In this paper, the Genetic Programming (GP) search optimisation was applied directly to the BT framework, as the BT structure resembles that of the decision tree by design GP can be applied to BTs using existing theory with minimal changes. Crossover operations were implemented as an interchange of a tree branch and mutations were implemented as a random change of one or multiple nodes in the tree.

Using this framework Lim et al. (2010) were able to develop an agent that could defeat a user-defined agent more than 50% of the time. This paper also showed that with their configuration a plateau was seen in the performance of the agent after a small number of generations and that other optimisation techniques may be needed to aid performance improvement. It also showed that the situations used to train the agent significantly steer its final generalised performance, a board selection of test cases are needed to prevent over-fitting (Lim, 2009).

In contrast to this direct implementation Perez et al. (2011) uses a variation on Genetic Algorithms (GA) was used to evolve the BT called Grammatical Evolution. This method is more similar to standard GA in that is uses a binary string structure but unlike standard GAs, GE uses a variable length string. The authors used a function to parse the BT as a character string and then used GP to optimise the strings. This method was used to develop an agent for the Mario AI Benchmark.

To address some of the performance limitations seen by Lim et al. (2010) the reactive behaviour of BTs was augmented by an A* path planning algorithm implemented simply as a node in the BT with a low priority. The authors also limited the BT to evolve with an *and-or* tree structure. This restriction reduced the search space of the EL but also forces the BT to evolve in a fixed structure limiting the learning capabilities of the final solution.

This previous work on evolving BTs was applied to simulated games where the game world is inherently discrete and certain. Real robotic platforms operate in a continuous and uncertain environments where behaviour learnt in simulation is based on a limited model of reality. Extending this behaviour to real world platforms is therefore a non-trivial investigation.

In the Aerospace arena, Ögren (2012) was the first to propose that the modularity, reusability and complexity of UAVs may be improved by implementing the decision making systems using a BT framework. He went on to describe how such a system would be implemented, a figure showing a proposed BT can be seen below in Figure 2-4.

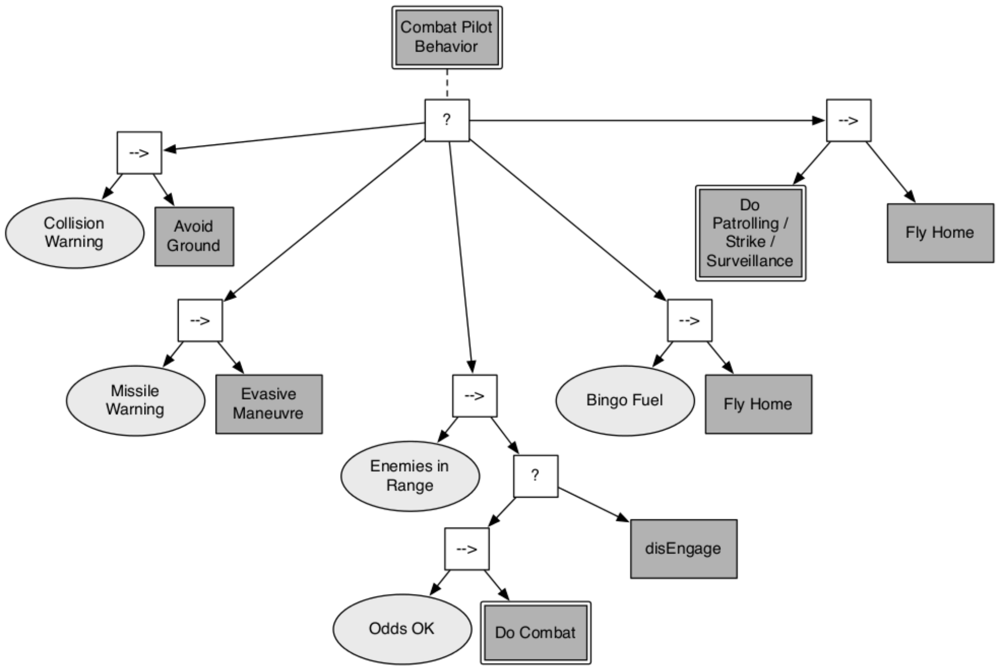


Figure 2-4: Example BT defining the behaviour for combat UAV (Ögren, 2012)

This approach is in contrast the the standard currently utilised in the Aerospace industry where FSM are commonly used to describe UAV behaviour. Some of the most popular UAV software packages such as Paparazzi and ArduPilot both use forms of FSM to implement their mission management (Paparazzi Community, 2013a; ArduPilot Community, 2013c,a).

Building on the work proposed by Ögren (2012), Klöckner (2013a) went on to discuss in more detail the possible advantages of using BTs in mission management and proposes some research area needed to implement such a management system. Klöckner (2013b) describes a method to parse a behaviour tree into logical statements using the description logic Attributive Language with Complements and concrete Domains (ALC(D)). These logic statements can then be analysed and the BT can be verified to be safe to operate based on a predefined set of logical rules.

BTs are also extensively used to model complex systems, bottlenecks in production systems or unreachable requirements in a product development can be identified (Dromey, 2003, 2004). The hierarchical formulation of BTs combined with the implementation of sequences and selectors make it a logical method to model a production cycle. The most notable implementation of this is with the Raytheon Company who uses this modelling technique to identify possible errors delays in the production of there missile systems (Dromey, 2003).

2-2 Evolutionary Learning Techniques

EL is a metaheuristic global optimisation method which imitates nature's mechanism of evolution (Hiller & Lieberman, 2010). This section aims to give a more in depth look into the operation of EL and provide an overview of the some of the most popular evolutionary techniques used today.

2-2-1 Genetic Algorithms

GAs are modelled after the theory of evolution as postulated by Charles Darwin in the mid 19^{th} century. Darwin observed that individuals in an environment gained a survival advantage through adaptation to the environment. This is commonly referred to as *survival of the fittest.* GAs are the digital application of the Darwinian principle of natural selection to artificial systems (Goldberg, 1989). Koza (2003) defines the GA as follows:

The Genetic Algorithm is a probabilistic search algorithm that iteratively transforms a set (called a population) of mathematical objects (typically fixed-length binary character strings), each with an associated fitness value, into a new population of offspring objects.

Feasible solutions for a particular problem are equivalent to members of a particular species, where the fitness of each individual is measured by some user-defined, problem specific, objective function. For each iteration or generation of a GA, the current population consists of a set of trial solutions currently under consideration. Some of the youngest members of the generation survive to adulthood based on their fitness and become parents who are paired at random to create the next generation. The resulting children have genes from both parents, the combination of which is determined through the process of crossover. This combination of genes from the fittest parents is a form of exploitation where the emerging policy tends towards the best policy. Each child may also be subject to mutation where individual parts of their genes are changed, this introduces variation in the population which results in greater exploration of search space. Mutation helps to search through the entire search space potentially moving the search out of local optima (Melanie & Mitchell, 1998; Chande & Sinha, 2008).

Unlike other heuristic methods, GAs analyse an entire population of solutions at each iteration (Hiller & Lieberman, 2010). Many derivative methods have branched from GAs, tree-like representations are explored in genetic programming and graph-form representations are explored in evolutionary programming; a mix of both linear chromosomes and trees is explored in gene expression programming.

The process of the GA can be summarised in the following steps:

1. **Initialisation** - Start with an initial population of feasible solutions.

2. **Evaluation** - In each iteration the fitness of each individual of the population is determined.
3. **Selection and Procreation** - Parents of the next generation are selected based on some heuristic fitness function. Parents are paired create children though Crossover where genes are combined in a random manner. Mutation is then applied to the children based on a mutation rate.

The steps 2 and 3 are iterated until some stop condition is met. The GA is characterised by 5 variables:

- **Population size** - Total number of individuals in the GA
- **Selection of Parents** - The heuristic used to determine which individuals of the current generation will become parents
- **Passage of Features** - How will procreation be implemented?
- **Genetic Operator Rates** - The rate at which the genetic operators will affect the population
- **Stopping Rule** - The GA is stopped based on some rule such as a maximum number of iterations, a maximum CPU time or a maximum number of iterations without an improvement of the best individual fitness value

Selection There are many techniques used to select individuals to become parents and contribute genetic material to the next generation. *Selection pressure* is defined as the degree to which the better performing individuals are preferred for mating than lower performing individuals (Miller & Goldberg, 1995). The selection pressure drives the GA to improve the average fitness of the generation and typically has large influence on the convergence rate of the GA, the higher the selection pressure the faster the convergence rate becomes. GAs can typically converge to the optimal or near-optimal solution for a large range of selective pressure but if it is too low the GA may take unnecessarily long time to converge and if the selection pressure is too high the GA may converge prematurely to a sub-optimum solution. We will mention a few of the more popular selection methods below.

Fitness Proportionate Selection was popular during the early development of GA with the Roulette Wheel a popular implementation method. Each slot in the virtual roulette wheel represents an individual in the population and the size of the slot is proportional to its fitness. The higher the fitness of the individual the more likely it is that more copies of that individual will be made for the next generation (Goldberg, 1989). This selection method is not very robust and breaks down in two situations: when all individuals obtain similar fitness, selective reproduction causes random search with genetic drift or; when only a few individuals have a much higher score than the remainder of the population, a large number of the offspring will be copies of these few individuals causing premature convergence of the GA (Nolfi & Floreano, 2000).

Another popular method is Rank Selection where individuals are ranked from best to worst and their probability of reproduction is proportional to their rank. A variation of this method is Truncation Selection which was used to good effect in (Nolfi & Floreano, 2000). This method involves ranking the individuals of the population and selecting the first M individuals and make O copies of their chromosomes given that $M \times O = N$.

A robust and easy to implement method used extensively is Tournament Selection (Miller & Goldberg, 1995). This method typically involves choosing s individuals at random from the population, sorting this sub-population based on their fitness and selecting the n^{th} member of the group based on the probability P given by

$$P = p \cdot (1 - p)^n; \quad n := [0, s - 1] \tag{2-1}$$

where p is the selection probability. Based on Equation (2-1), when $(p = 1)$ the best member of the selection sub-population is always selected and in the case where $(s = 1)$ selection is random. In practice this is implemented as follows for a selection population of $(s = 2)$. Two individuals are selected at random and then check if a randomly generated number in the range $[0, 1]$ is larger than P, if so then the individual with the higher fitness is selected for reproduction, else the individual with the lower fitness is selected. Selection pressure can be adjusted by changing the tournament size, the more individuals chosen in the tournament pool, the less likely weaker individuals will be selected to procreate thereby increasing the selection pressure. This method is often combined with Elitism to ensure that the best performing individuals are not lost. Elitism consists of choosing a number individuals with the highest fitness and copying them into the new generation

Procreation Varying techniques are used to ensure that the GA will converge in finite time to a near-optimal solution, we will consider a few here. Firstly, Reproduction is a process which typically involves making an exact copy of a member of the population chosen by some selection method and moving the copy into the population of the new generation. This is similar to elitism but with a more general selection method applied.

Crossover is the main operator in GAs, it is typically applied to the population of selected individuals for the new population with a uniform probability referred to as the Crossover rate. Crossover involves choosing a random point along each chromosome and exchanging genetic material between the chromosomes. This is referred to as One Point Crossover, alternatively if multiple points are used for crossover this is called multi-point crossover. Although reproduction methods that are based on the use of two parents are more biology inspired, some research suggests that more than two parents generate higher quality chromosomes (Eiben et al., 1994). A crossover rate that is too high may lead to premature convergence of the genetic algorithm in a local optima.

Mutation In GAs, mutation is typically applied to each node in the character string with a given uniform probability selection defined by the Mutation Rate. Mutation typically consists of some small localised change of a gene, for example, with binary representation it may be applied as a bit switch. A mutation rate that is too high may lead to loss of good solutions, this can be mediated by the inclusion of elitist selection.

2-2-2 Genetic Programming

In the field of machine learning and artificial intelligence a variation of the GA algorithm is commonly used called GP. This method addresses the limiting factors of the GA namely the use of a fixed length binary string, not all problems can be reduced to this representation. GPs were developed specifically to automatically develop computer programs to solve a high-level problem statement (Koza, 1992, 1994). The idea was postulated by Arthur Samuel in 1959 who stated the question:

"How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what is needed to be done, without being told exactly how to do it?"

Koza (1992) describes in detail how GPs can be represented in binary trees using LISt Processing (LISP) type expressions. LISP is a form of functional computer programming language based on nested lists, this is a very popular in Artificial Intelligence (AI). These structures are used in combination with GPs to automatically generate complex mathematical functions for problem solving or provide a mapping of inputs and outputs of a robotic platform.

Crossover is applied in a similar manner to GAs where a link in the binary tree is chosen at random and used to exchange tree branched between parents. Mutation however is applied by replacing a randomly selected node in the binary tree with a randomly generated sub-tree (Koza, 1992). This method can be seen to be similar to a crossover operation with a randomly generated second parent, this is therefore sometimes called *macromutation* or *headless chicken crossover* (Angeline, 1997). Due to the similarity in the mechanisms, mutation is not often used in GP and is often replaced by using a larger population size.

2-2-3 Neuroevolution

Unlike previous methods, this form of learning uses evolutionary algorithms to train an ANN. ANNs are computational models inspired by naturally occurring biological nervous systems (Floreano et al., 2008). As the name suggests ANN are composed of a network of interconnected *neurons* which have a number of *inputs* and *outputs*. Each neuron and the *synapses* connecting them together affect the signal passing along them with varying methods. The response of the ANN can be changed by varying the parameters of these neural and synaptic weights.ANNs can be represented as shown in Figure 2-5. In neuroevolution, these weights are optimised using evolutionary operators.

In the summary provided by Floreano et al. (2008), one unique design choice that has to be made when setting up the evolutionary process with neuroevolution is the selection of the representation method used, these can be divided into three groups: direct, developmental and implicit.

Direct representation is a one-to-one mapping of parameter values in the ANN and the genes in the genetic string. This is usually used to evolve the weights of ANN with fixed network topologies. One method for the encoding and evolution of both the architecture and neural weights of ANN is NeuroEvolution of Augmenting Topologies (NEAT) (Stanley & Miikkulainen, 2002).

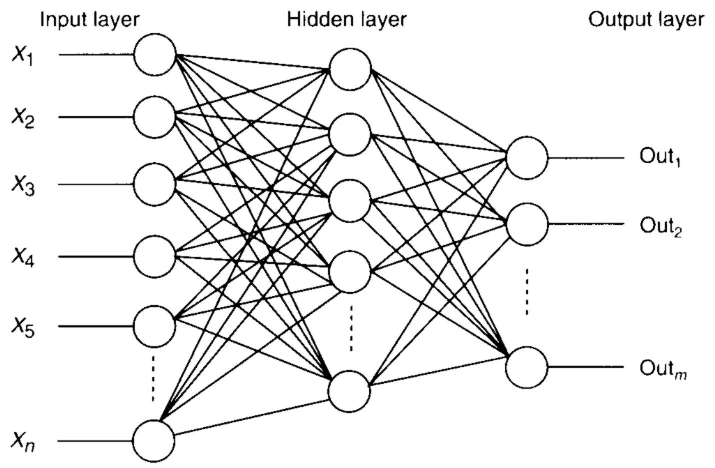


Figure 2-5: Graphical depiction of the Artificial Neural Network framework

Floreano et al. (2008) states that developmental representations were introduced by some researchers to evolve large networks. In this method, the specification of the neural net developmental process is encoded which in turn is used to generate the ANN. This representation results in compact genotypes which can generate complex ANN and developed sub-networks can be reused throughout the ANN. A popular method using this representational method is *Cellular Encoding.*

Finally, implicit encoding more closely mimics biological gene regulatory networks. As seen in biological neural networks, the neuron behaviour is not explicitly encoded in the neuron but is dependant on the environment it is immersed in. An implementation of this method is *Analog Genetic Encoding*, this can be applied to ANNs by introducing a level of abstraction where the synaptic weights connecting the neurons is encoded in an *interaction map.*

In the the field of Robotics, a notable application of neuroevolution is Evolutionary Robotics. This methodology builds upon the idea that complex behaviour can be expressed from a combination of simple behaviours as used in Behaviour-Based Robotics (BBR) (Arkin, 1998). BBR automatically encodes simple behaviours in discrete blocks through iterative decomposition of tasks and a coordinator chooses or mixes behaviours most applicable based on the environment. This idea has been used to good effect on simulated and real platforms.

Nolfi (1998) claim that the task of determining the decomposition and integration of behaviours should be incorporated in the adaptation process and not a separate process. Based on this idea, ER combines the determination and the selection of a network of simple behaviours into one step. This network of behaviours is automatically learned using neuroevolutionary processes (Nolfi & Floreano, 2000; Floreano & Mondada, 1994). ER is essentially an automatic technique for generating solutions for a particular robotic task, based on artificial evolution (Trianni, 2008).

One pitfall of this method is that the evolution of the ANN is typically based on the performance of the agent in a simulated environment. Research has shown that there are always significant differences between simulation and reality, this is referred to as the *reality-gap* (Zagal & Solar, 2007). Many methods have been investigated to reduce this reality gap and can be separated into three main approaches (Bongard, 2013). The first approach investigates the influence of simulation fidelity on the EL, with investigation focusing on the influence of adding differing levels of noise to the robotic agent's inputs and outputs (Jakobi et al., 1995; Miglino et al., 1995; Meeden, 1998). It was shown that sufficient noise can deter the EL from exploiting artifacts in the simulation but that this approach is generally not scalable as more simulation runs are needed to distinguish between noise and true observations. The second group focuses on *co-evolution* where the robotic controller is evaluated in and the simulation model is updated using the performance error with a real world robotic platform simultaneously (Bongard et al., 2006; Zagal & Solar, 2007). Alternatively, the error between the simulation and real world environment can be used to estimate of the suitability of a learnt behaviour on the real robot in a multi-objective function to trade-off simulated robotic performance and the transferability of the behaviour (Koos et al., 2013). The final approach performs online adaptation of the behaviour after the EL using relatively complex methods (Hartland & Bredèche, 2006).

Although these approaches have varying levels of success the problem of possible damage to the robotic platform during the learning process is also not addressed. Most tests with online learning or co-evolution are done with a robotic platform that has relatively slow dynamics and low mass so risk of damage is low. Additionally the path of the robot is observed by an external source. Applying online learning techniques with more complex platforms in an unsupervised arena is still a challenge.

2-3 Behavioural Development on Flapping Wing Platforms

Some previous work has been done with the aim of extending the functionality of the autonomous behaviour of the Delfly flight platform. This work has till now been focused on obstacle avoidance in cluttered environments. de Croon et al. (2012) proposed the framework expressed in the state machine showed in Figure 2-6. The DelFly was simply required to perform a bidirectional turn for a fixed time if it encountered an obstacle.

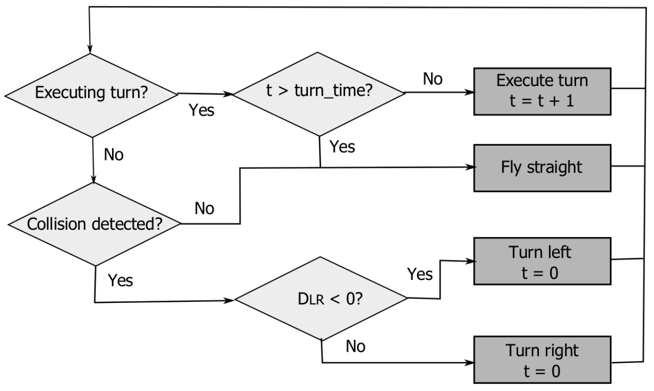


Figure 2-6: DelFly obstacle avoidance FSM implementation as defined by de Croon et al. (2012)

The work of Tijmons (2012) extended on this work and proposed that the DelFly should only perform unidirectional turns to ensure that it would not get caught in corners. The behaviour is expressed in the state machine below in Figure 2-7. Limited to the obstacle avoidance problem only, both of these solutions are relatively quite simple but already require many parameters to be tuned to operate successfully. Let us take the work of Tijmons (2012) for example, this decision process is determined using 4 parameters, each has to be tuned for a particular flight speed and flight configuration. This problem further advocates the necessity for future robotic system s to automatically learn required behaviour to achieve tasks.

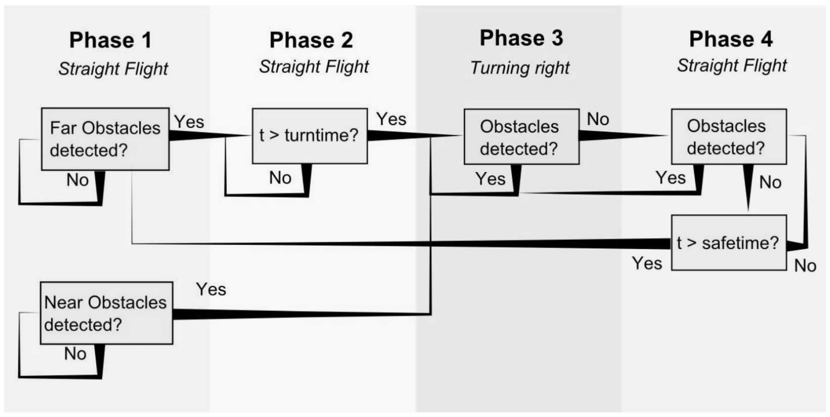


Figure 2-7: DelFly obstacle avoidance FSM implementation as defined by Tijmons (2012)

Other applications with flapping wing flight platforms include the work of Julian et al. (2013) using the H^{2}Bird 13g flapping wing UAV for a fly-through-window task. The experiment set-up an be seen in Figure 2-8. Unlike the self contained sensory and control system onboard the DelFly Explorer, this work used a ground based camera and off-board image processing to generate the heading set-point for the flight platform. Developing the algorithms to safely avoid crashing into the walls and other obstacles while searching and attempting to fly through a window is a non-trivial task. The fly-through-window task is a real world problem that is pertinent to current research and is therefore chosen as the target behaviour to be learned as part of this research.

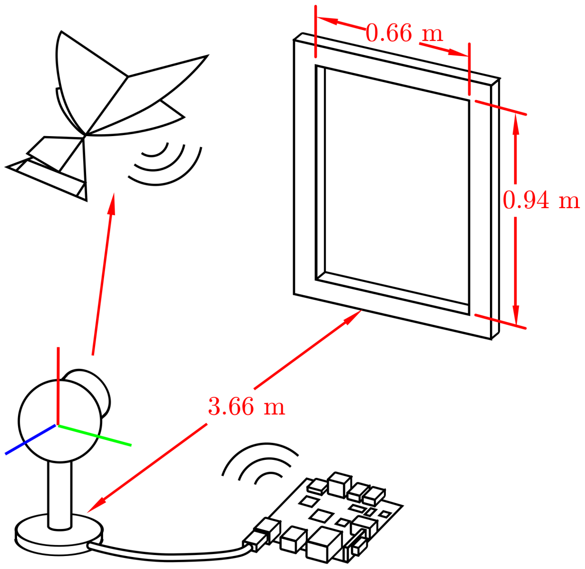


Figure 2-8: Berkeley fly-through-window set-up (Julian et al., 2013)

Chapter 3

DelFly Behaviour Tree Implementation

The task selected for the DelFly Explorer is to fly around a room to search for and identify a window. Once identified the DelFly should fly through the window into an adjoined room. This task should be completed using onboard sensors and computational capabilities only. For simplicity, this paper will only investigate the BT will only control the horizontal control of the DelFly. The flight velocity will also be fixed for all flights.

In this chapter we will first present the DelFly Explorer flight platform in Section 3-1 as well as discuss the onboard vision based algorithms used for the fly-through-window task in Section 3-2. Section 3-3 contains some information on the SmartUAV simulation environment, this is followed by a description of the simplified DelFly flight model used presented in Section 3-4. The design of the the mission management module is then presented in Section 3-5. The chapter is then concluded by a detailed discussion of the BT implementation in Section 3-6. The work in this chapter builds on a preliminary investigation performed on the Khepera wheeled robot implementation described in Appendix B.

3-1 DelFly Explorer

The DelFly is an insect-inspired flapping-wing UAV developed at the Delft University of Technology (DUT), the main feature of its design is its biplane-wing configuration which flap in anti-phase (de Croon et al., 2009). The DelFly Explorer is a recent iteration of this micro ornithopter design (de Wagter et al., 2014). In its typical configuration, the DelFly Explorer is $20g$ and has a wing span of $28cm$. In addition to its 9 minute flight time, the DelFly Explorer has a large flight envelope ranging from maximum forward flight speed of $7m/s$, hover, and a maximum backward flight speed of $1m/s$. A photo of the DelFly Explorer can be seen below in Figure 3-1.

The main payload of the DelFly Explorer is a pair of light weight cameras used to perform onboard vision based navigation as shown in Figure 3-1. Each camera is set to a resolution of 128×96 pixels with a field of view of about $60° \times 45°$ respectively. The cameras are separated

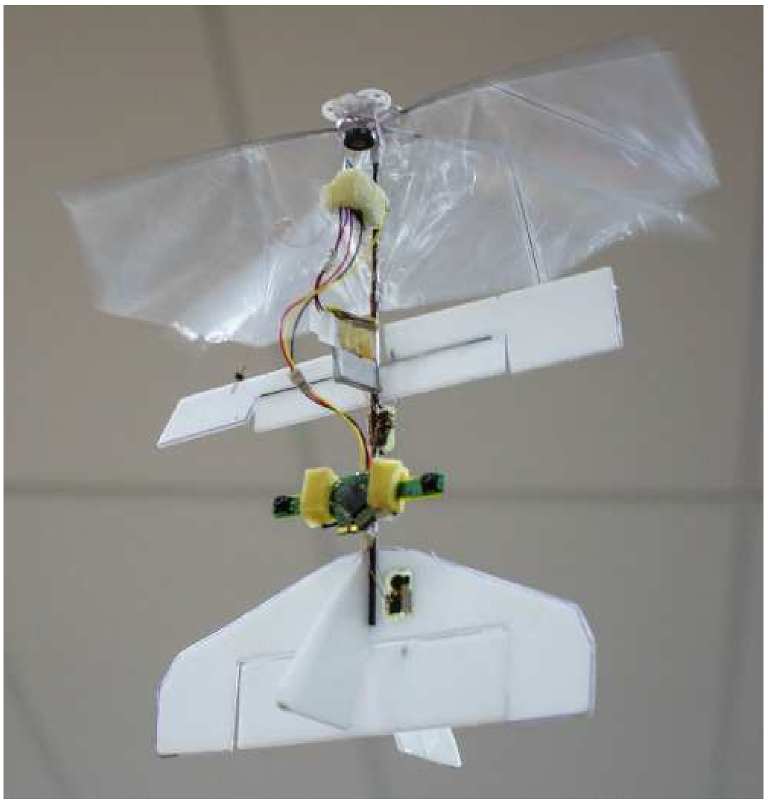


Figure 3-1: DelFly Explorer in flight with the camera module in view

by *7cm*, this separation can be used to gain some information about the 3D environment the DelFly is operating in.

Currently, *Stereo Vision* is the main vision based processing algorithm used on the DelFly Explorer. In computer vision, this is the method used to extract 3D information from digital images similar to the way Stereopsis is used by binocular vision in animals (Scharstein & Szeliski, 2002). This information gives the UAV depth perception which is used in obstacle avoidance algorithms onboard the DelFly Explorer making it the first flapping wing UAV that can perform fully autonomous flight in unknown environments. More information on the vision based algorithms is presented in the next section. Additionally, the DelFly Explorer also carries a barometer to measure its pressure altitude and a set of three gyroscopes to determine its rotational rates.

3-2 Vision systems

Two vision based algorithms have been previously implemented on the DelFly Explorer which will be used for the fly-through-window task, namely: LongSeq Stereo Vision and Window Detection. We will briefly describe these systems below.

3-2-1 LongSeq Stereo Vision

The DelFly Explorer uses a Stereo Vision algorithm called *LongSeq* which can be used to extract 3D information of the environment from its two onboard optical cameras (de Wagter et al., 2014). The main principle in computer vision based stereo vision is to determine which pixel corresponds to the same physical object in two or more images. The apparent shift in location of the corresponding pixels in the images, also referred to as the pixel disparity, with the known geometry of the cameras can be used to reconstruct the 3D environment (Szeliski & Zabih, 2000).

Some stereo vision methods use (semi-)global matching where a pixel is matched in a global area in the companion image (Szeliski & Zabih, 2000). This method is typically computationally and memory intensive. Generally, the more localised the search faster the algorithm runs. Localised search however is typically more susceptible to the amount of texture in the image as well as scene lighting and camera measurement noise. Considering the limited computational hardware onboard the DelFly Explorer a trade-off has to to be made as to what stereo vision method can be used.

LongSeq performs optimization along one image line at a time reducing the degrading effect of bad pixels as compared to other methods as pixel mismatching is constrained to that line only (de Wagter et al., 2014). Additionally, since only one line is processed at a time the memory requirements of the algorithm are lower than other conventional systems. This computation efficiency and relatively good performance when considering texture and video noise makes LongSeq a suitable candidate for the small DelFly Explorer platform.

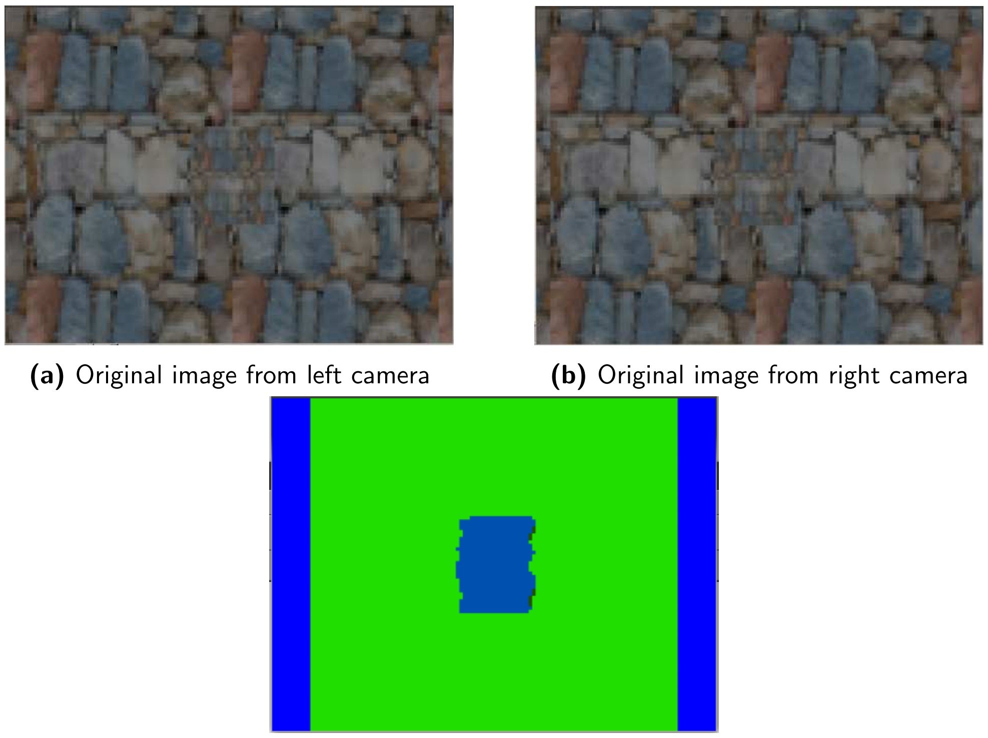
A version of LongSeq was implemented for this research, a detailed description of the workings of the algorithm see the work of de Wagter et al. (2014). The algorithm results in a pixel disparity map which can be used to determine the distance to an object in the environment. For the purpose of this research, the disparity map will be used primarily for obstacle avoidance and window detection.

LongSeq has two main input parameters namely: maximum disparity which limits the algorithm search length and defines the minimum distance at which a feature can be matched and; the cost function threshold which defines what is a good pixel match. The maximum disparity chosen for this paper was 12 which, in combination with the resolution and separation of the cameras, corresponds to distinguishing objects at a minimum distance of about $70cm$ away. This is sufficient to identify the window till it no longer fits in the camera frame and avoid most possible obstacles.

An example of the output can be seen below in Figure 3-2, this shows a disparity map along with the original optical image of the sample room facing a window in a wall at $4m$ distance. To the naked eye the window is clearly visible in the disparity image whilst not so in the original camera image.

3-2-2 Window Detection

An Integral Image window detection algorithm is used to aid the UAV in the fly-through-window task. Integral image detection is a high speed pattern recognition al-


(a) Original image from left camera (b) Original image from right camera

(c) Processed Stereo Disparity map

Figure 3-2: Original images from left and right camera as well as the resultant disparity map stereo image produced when DelFly aimed at a window in a wall from $4m$ distance

gorithm which can be used to identify features in a pixel intensity map (Crow, 1984; Viola & Jones, 2001). The integral image ($II(x, y)$) is computed as

$$II(x,y) = \sum_{x' \leq x, y' \leq y} I(x', y') \tag{3-1}$$

where x and y are pixel locations in the image $I(x, y)$. As each point of the integral image is a summation of all pixels above and to the left of it, the sum of any rectangular subsection is simplified to the following computation

$$rect(x, y, w, h) = II(x + w, y + h) + II(x, y) - II(x + w, h) - II(x, y + h) \tag{3-2}$$

This makes it computationally efficient to use these summations to identify a feature in the image. de Wagter et al. (2003) applied this technique to camera images to identify a dark window in a light environment using two cascaded classifiers. The first identified likely window location candidates by comparing the average intensity within a feature space to that of the boarder surrounding the feature reducing the search space significantly. The second classifier refines these likely locations by checking each side of the feature to the average inner disparity. This two step identification further increases the computational efficiency of this window detection method.

The work of de Wagter et al. (2003) was designed specifically to operate when approaching a building in the day time on a bight day. A more generalised method may be to apply the same technique described above to the disparity map rather than the original camera images. The disparity map shows closer objects with larger disparity and as a result a window will appear to be an area of low disparity (dark) in an environment of higher disparity (light). As a result the same classifiers can be used as described by de Wagter et al. (2003). These classifiers help to reduce the computational complexity of the window identification by only vigorously investigating probable solutions.

3-3 SmartUAV Simulation Platform

SmartUAV is a Flight Control Software (FCS) and simulation platform developed in-house at the DUT. It is used primarily with small and micro sized aerial vehicles and notably includes a detailed 3D representation of the simulation environment which is used to test vision based algorithms. It can be used as a ground station to control and monitor a single UAV or swarms of many UAVs or as a simulation platform to test advanced Guidance Navigation and Control (GNC) techniques (de Wagter & Amelink, 2007; Amelink et al., 2008). As a tool developed in-house, designers have freedom to adapt or change the operating computer code at will, making it very suitable for use in research projects.

SmartUAV contains a large visual simulation suite which actively renders the 3D environment around the vehicle. OpenGL libraries are used to generate images on the PC's GPU increasing SmartUAV's simulation fidelity without significant computational complexity. As a result high fidelity 3D environments can be used allowing the vision based algorithms to be tested with a high level of realism.

Additionally, SmartUAV was designed with a fully modular underlying framework. It is made up of a collection of separate modules which operate independently but can communicate with each other using dedicated communication channels. These modules are implemented on separate program threads on the processor making them truly modular. This modular framework makes this software platform very flexible and adaptable to many use cases. New models can be added or removed easily as required for each situation. In simulation mode, the vehicle and sensor dynamics are contained in a module which can be easily replaced with data from a real vehicle. Modules can be grouped together into Advanced Flight Management System (AFMS) charts which uses a Matlab Simulink like module flow diagram to give the user the ability to visually connect modules together simplifying the design of complex systems.

SmartUAV updates the flight dynamics of all the simulated vehicles at $100Hz$ but the sensor information is made available to other modules at $50Hz$ update rate. The user-defined AFMS can be updated at a variable rate. The 3D environment is rendered at a fixed user-defined rate.

3-4 Simplified DelFly Model

The DelFly Explorer is a newly developed platform and its flight dynamics have not yet been investigated in depth. As a result an existing model of the DelFly II previously implemented based on the intuition of the DelFly designers will be used in this work. This model is not a fully accurate representation of the true DelFly II dynamics but was sufficient for most vision based simulations previously carried out. In this work, the inaccuracy of the model will intentionally create a reality gap between the simulated dynamics of the DelFly and reality. We will briefly summarise the dynamics used in simulation below.

The DelFly II has three control inputs, namely: Elevator (δ_e), Rudder (δ_r) and Thrust (δ_t). All inputs are normalised to the range $[-1, 1]$ which are mapped onto the actuator limits of the DelFly. The elevator and rudder simply set the control surface deflection and the thrust sets the flapping speed. The actuator dynamics of the DelFly for the rudder actuator were implemented using a time delay and can be seen below in (3-3).

$$\dot{\delta_r} = \delta_{r_{max}}(u_r - \delta_r); \quad u_r : [-1, 1] \tag{3-3}$$

where u_r is the rudder input and $\delta_{r_{max}}$ is the maximum rudder deflection. This results in a rudder transfer function with a rise time of $2.2s$ and settling time of $3.9s$ The elevator deflection and thrust setting are simply mapped directly from the control input.

$$\begin{bmatrix} \delta_e \\ \delta_t \end{bmatrix} = \begin{bmatrix} \delta_{e_{max}} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} u_e \\ u_t + 1 \end{bmatrix}; \quad u_e, u_t : [-1, 1] \tag{3-4}$$

where u_e and u_t are the elevator and thrust input respectively and $\delta_{e_{max}}$ is the maximum elevator deflection. The pitch (θ), yaw (ψ) and flight velocity (V) dynamics of the DelFly can be seen in (3-5). Things to note are there is no coupling in the flight modes of the simulated DelFly. For this research, the throttle setting was constant at trim position resulting in a flight velocity of $0.5m/s$. Additionally, the maximum turn rate was set to $0.4rad/s$ resulting in a minimum turn radius of $1.25m$.

$$\begin{bmatrix} \dot{\theta} \\ \dot{\psi} \\ \dot{V} \end{bmatrix} = \begin{bmatrix} -\frac{1}{72} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \theta \\ \psi \\ V \end{bmatrix} + \begin{bmatrix} -\frac{1}{24} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \delta_e \\ \delta_r \\ \delta_t \end{bmatrix} \tag{3-5}$$

The roll angle (ϕ) of the DelFly is simply a function of the rudder setting as in (3-6).

$$\phi = \frac{1}{2}\delta_r \tag{3-6}$$

Although the DelFly is operated in an indoor environment due to its low mass, local air flow does affect its flight dynamics, for this reason gusts are introduced to the simulation. These gusts can be turned on or off depending on the flight settings. As implemented, these gusts affect the yaw angle and the vehicle directional velocity. The gust is defined using directional parameters in the earth fixed frame using separate gust strength parameters for each axis. The gust strengths are randomly generated at random time intervals during flight and are bounded by a maximum gust (w_{max}). The randomised changes in the gusts help to mimic the non-deterministic behaviour of indoor small scale air flow.

$$\dot{\psi} \leftarrow \dot{\psi} + 0.2\left(-w_x sin\psi + w_y cos\psi\right) \tag{3-7}$$

Including the gusts, the translational equations of motion become

$$\begin{aligned} \dot{x} &= Vcos\psi + w_x \\ \dot{y} &= Vsin\psi + w_y \end{aligned} \tag{3-8}$$

There are some notable differences between the DelFly II and DelFly Explorer, firstly the Explorer replaces the rudder with a pair of ailerons to roll the DelFly without inducing yaw, this helps to stabilise the captured images. Additionally, the DelFly Explorer is $4g$ heavier and has a slightly higher wing flapping speed. The flight dynamics described above are very simple and not fully representative of the complex flight dynamics involved with the flapping wing flight of the DelFly platform. As a result a large reality gap is expected when moving from the simulation to reality.

3-5 Mission Management Module

Based on the previous description of the sensors onboard the DelFly and the algorithms available the BT module has three inputs: the disparity map produced by the stereo vision; the certainty of a window detection and; the window centre (x, y) location in the image frame normalised in the range $[-1, 1]$ from the integral image window detector.

Internally, the module computes the sum of disparity of the disparity map which gives an indication of proximity to a large object such as a wall. Additionally, the difference of disparity sums from the left and right half of the map are also computed providing the DelFly with some information or relative proximity to object in the horizontal plane. The BT module outputs the turn rate set point. This set points is sent to the DelFly FCS to set the servo controller of the DelFly rudder. The module can generate values on the mapping $[-1, 1]$ in increments of 0.1.

In SmartUAV, the BT node is placed in series with the stereo vision module in an AFMS as shown graphically in Figure 3-3. In terms of the larger SmartUAV simulation, the vision based calculations are the most computationally intensive portion making it the limiting factor for the speed of operation of the wider decision process. The higher the decision loop frequency relative to the flight dynamics the longer a single simulation will take. This must be balanced by the frequency at which the DelFly is given control instructions, where generally higher is better. Considering this trade-off the AFMS loop was set to run at $5Hz$ relative to the flight dynamics loop. This is a conservative estimate of the actual performance of the vision systems onboard the real DelFly Explorer.

3-6 DelFly Behaviour Trees

A Unified Modelling Language (UML) chart with the node class structure for the DelFly can be seen below in Figure 3-4. This shows that two composite nodes were used: *Sequence* and; *Selector*. Two condition nodes and one action nodes were developed, namely: *greater_than*; *less_than* and; *turnRate*. These behaviour nodes are accompanied by a *Blackboard* which was developed to share information with the BT.

The Blackboard architecture implemented for the DelFly to add data to the BT, containing five entries: *window x location* (x), *window response* (σ), *sum of disparity* (Σ), *horizontal disparity difference* (Δ) and *turn rate* (r). This variable map is available to all nodes in the BT, the first four are condition variables which are set at the input of the BT and the last item is used to set the BT output.

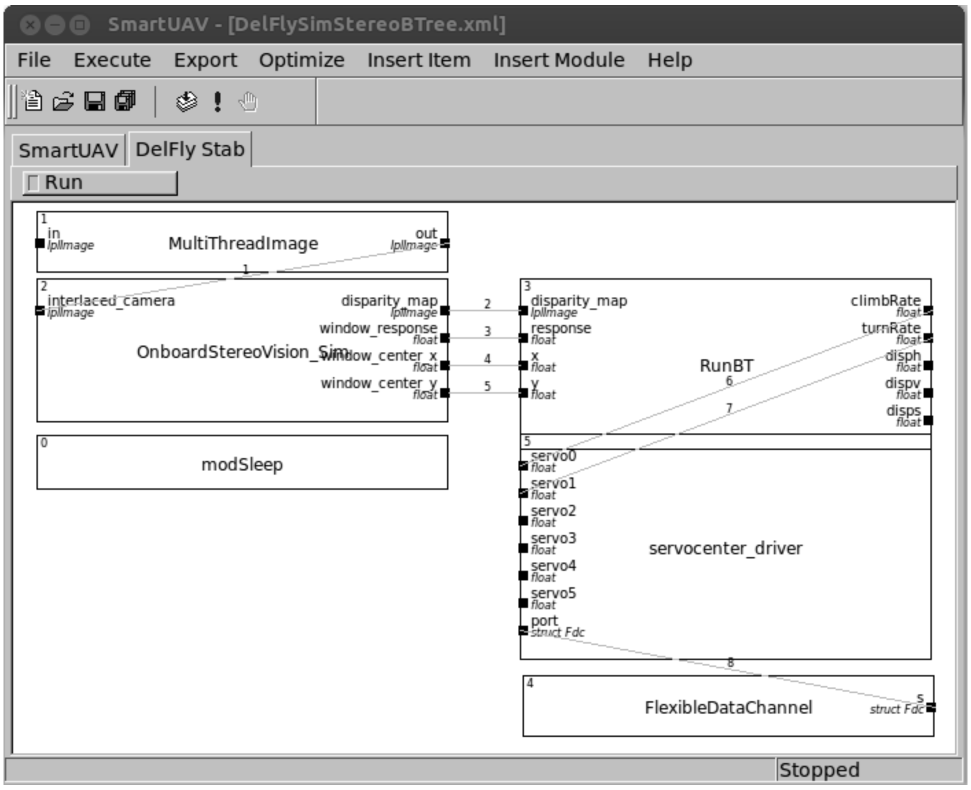


Figure 3-3: SmartUAV GUI showing an Advanced Flight Management System interaction overview chart for the DelFly fly-through-window task

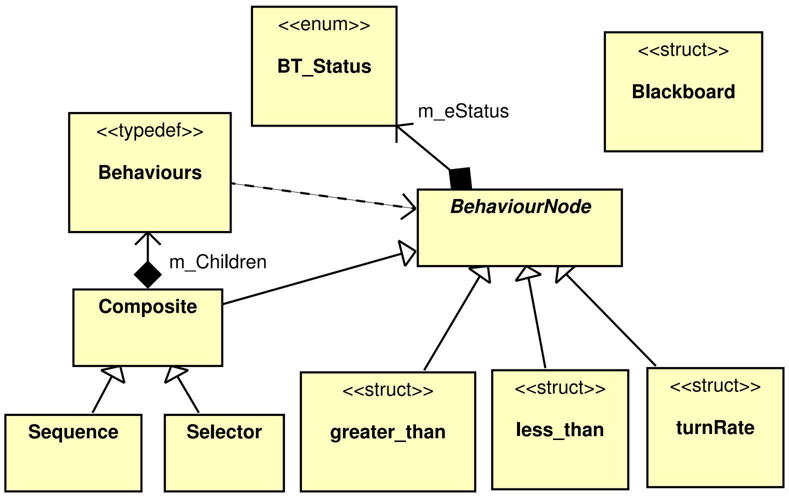


Figure 3-4: UML class diagram of the Behaviour Tree framework for the DelFly

The conditions have two internal constants which are set on their initialisation: one sets which Blackboard variable is to be evaluated and the other is the threshold to be tested. As the names suggest, *greater_than* and *less_than* have boolean returns as to whether a variable input is greater than or less than the threshold respectively. The Action node *turnRate* sets the DelFly rudder input directly.

Chapter 4

Evolutionary Learning Incorporation

The following chapter describes how EL was implemented to optimise the behaviour for the DelFly fly-through-window task. Section 4-1 presents the implementation of the various genetic operators. This is followed by a list of the metrics used to evaluate desired behaviour in Section 4-2 and how these metrics were implemented in a fitness function in Section 4-3.

First let us describe how SmartUAV was changed to make it suitable for the EL process. SmartUAV was designed primarily to simulate algorithms in realtime but these simulations would cause the genetic optimisation to take a long time to converge. To decrease the total simulation time some changes were made to the SmartUAV code base to increase the simulation frequency. After some program optimisation it was possible to run SmartUAV with all the necessary modules at $\times 40$ realtime, ie a $60s$ simulation takes $1.5s$ to execute.

4-1 Genetic Operators

The main elements which make up the genetic program are: how the population of solutions is initialised; how individuals are chosen for procreation; how individuals from one generation are combined to make the next generation; how individuals are mutated and; when do you stop the optimisation. These are individually described below.

Initialisation The initial population of M individuals is generated using the *grow* method (Koza, 1994). This results in variable length trees where every Composite node is initialised with its maximum number of children and the tree is limited by some maximum tree depth. This provides an initial population of very different tree shapes with diverse genetic material to improve the chance of a good EL search.

Selection A custom implementation of Tournament Selection is used in this thesis derived from Miller & Goldberg (1995). This is implemented by first randomly selecting a subgroup of s individuals from the population. This subgroup is then sorted in order of their fitness. If two individuals have the same fitness they are then ranked based on tree size, where smaller is better. The best individual is typically returned unless the second individual is smaller, in which case the second individual is returned. This was done to introduce a constant pressure on reducing the size of the BTs.

Crossover As the tournament selection returns one individual, two tournaments are needed to produce two parents needed to perform crossover. The percentage of the new population formed by Crossover is defined by the Crossover Rate P_c. Crossover is implemented as the exchange of one randomly selected node from two parents to produce two children. The node is selected totally at random independent of its type or its location in the tree. Figure 4-1 and Figure 4-2 graphically show this process.

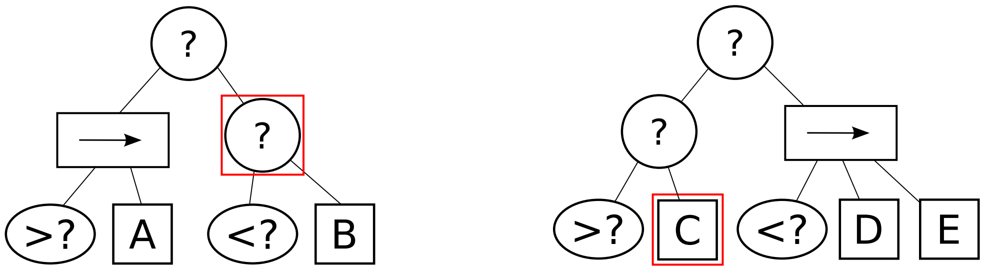


Figure 4-1: Sample parent Behaviour Trees with selected nodes for crossover highlighted with a box

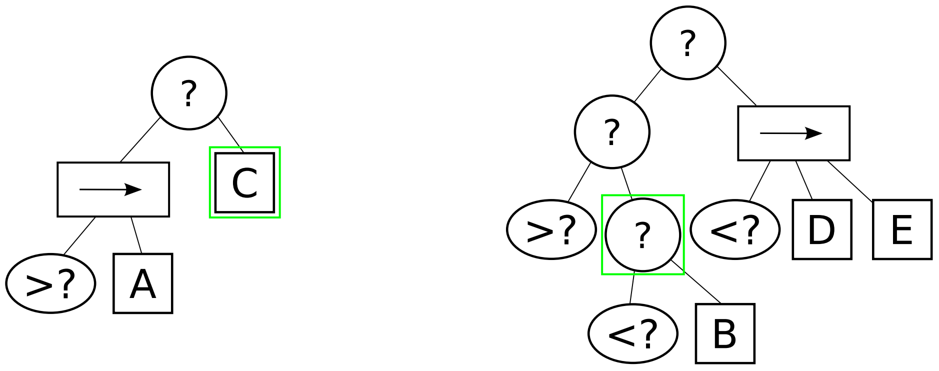


Figure 4-2: Resulting children from crossover operation of parents shown in Figure 4-1

Mutation Mutation is implemented with two methods, namely: micro-mutation and; Macro-mutation. Micro-mutation only affects leaf nodes and is implemented as a reinitialisation of the node with new operating parameters. Macro-mutation, also called Headless-Chicken Crossover, is implemented by replacing a selected node by a randomly generated tree which is limited in depth by the maximum tree depth (Angeline, 1997). The probability that mutation is applied to a node in the BT is given by the mutation rate P_m. Once a node has been selected for mutation the probability that macro-mutation will be applied rather than micro-mutation is given by the headless-chicken crossover rate P_{hcc}.

Stopping Rule An important parameter in EL is when to stop the evolutionary process. Evolutionary computing is typically computationally intensive due to the large number of simulations required to evaluate the performance of the population of individuals. Placing a limit on the maximum number of generations can help avoid unnecessary long computational time. Additionally, like many learning methods, genetic optimisation can be affected by overtraining. Typically in genetic optimisation a limited number of training initialisations are used to determine the individuals performance in one generation. This initialisation is randomly changed each generation to promote the behaviour is generalised for the task. Overtraining of the population can appear as the entire population being optimised to perform well in the generalised situation and not one individual. This should appear as a decrease in the generalised performance of the best performing individual.

For these reasons, the genetic optimisation has a maximum number of generations (G) at which the optimisation will be stopped. Additionally, when the trees are sufficiently small to be intelligible, the process can be stopped by the user.

Overview A flow chart of the implementation of the EL can be seen below in Figure 4-3.

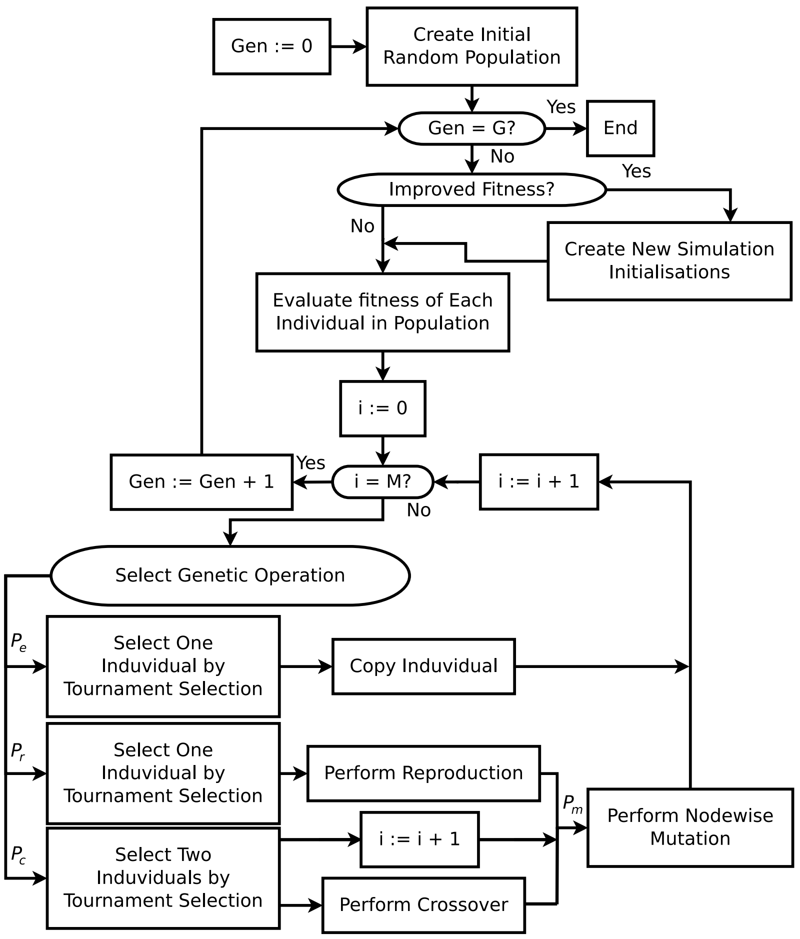

Gen := 0
Create Initial Random Population
Gen = G?
Yes
End
No
Improved Fitness?
Yes
No
Create New Simulation Initialisations
Evaluate fitness of Each Individual in Population
i := 0
Gen := Gen + 1
Yes
i = M?
i := i + 1
No
Select Genetic Operation
P_e
Select One Induvidual by Tournament Selection
Copy Induvidual
P_r
Select One Induvidual by Tournament Selection
Perform Reproduction
P_m
Perform Nodewise Mutation
P_c
Select Two Induviduals by Tournament Selection
i := i + 1
Perform Crossover

Figure 4-3: Flow diagram showing implementation of the Evolutionary Learning architecture for Behaviour Tree evolution

4-2 Performance Parameters

To evaluate the effectiveness of DelFly in the fly-through-window task we must define some performance metrics. These parameters are:

- **Success rate** - How often does the DelFly successfully fly through the window?
  This is the main performance parameter as it is a measure of how well the goal is met. The more often the DelFly is successful the better.
- **Tree size** - How large is the BT?
  As BTs are meant to increase the user awareness of what the automatically evolved individual is doing the evolved trees have to be of reasonable size. The smaller the BT the better.
- **Angle of Window Entry** - At what angle did the DelFly enter the window?
  It is expected that a the DelFly should enter the window near to perpendicular, the closer to perpendicular the better the flight path.
- **Time to Success** - How long does the DelFly take to fly though the window?
  The faster the DelFly flies through the window the better.
- **Distance from Centre of Window at Fly-Through** - How far away from the centre of the window was the DelFly during a successful flight? The closer to the centre the better.

4-3 Fitness Function

The purpose of the fitness function is to rate the performance of the individuals in the population and is an objective function that the EL is trying to optimise. This function is very important to determine the eventual behaviour exhibited by population. There are two main parameters that can be used for this function to promote successful fly-through behaviour: a boolean success parameter and a function inversely proportional to the distance to the window. Both of these have their benefits and pitfalls.

Using the boolean approach, whilst individuals that are successful will be rewarded there is no measure for how good behaviour is when not successful. This results in the population not being actively directed towards the goal but relying more on random sorting to realise a solution. This approach does not reliably find a solution.

Alternatively, the approach of the distance to the window is also flawed in that if you take the minimum distance to the window centre during a flight it is possible that a behaviour that flew close to the window but not through it would do better than one that crashed very near the window while attempting to fly through. If we use the distance to the window centre at the end of a simulation run we can see that the fitness of an individual that hits the edge of a window is not much lower than an individual that just flies through. As the goal is to fly through the window and not just towards it, this function is not sufficient.

Eventually after some experimentation, a combination of these two approaches was used to encourage the EL to converge on a population that flies through the window as often as possible. A discontinuous function was chosen such that a maximum score is received if the UAV flies through the window and a score inversely proportional to its distance to the window if not successful. The fitness F is described below in (4-1).

$$F = \begin{cases} 1 & if\ Success \\ \frac{1}{1+3|\mathbf{e}|} & else \end{cases} \tag{4-1}$$

where success is defined as flying through the window and $\mathbf{e}$ is the vector from the centre of the window to the location of the UAV at the end of the simulation. A plot of this fitness function in the $x - y - fitness$ domain can be seen below in Figure 4-4.

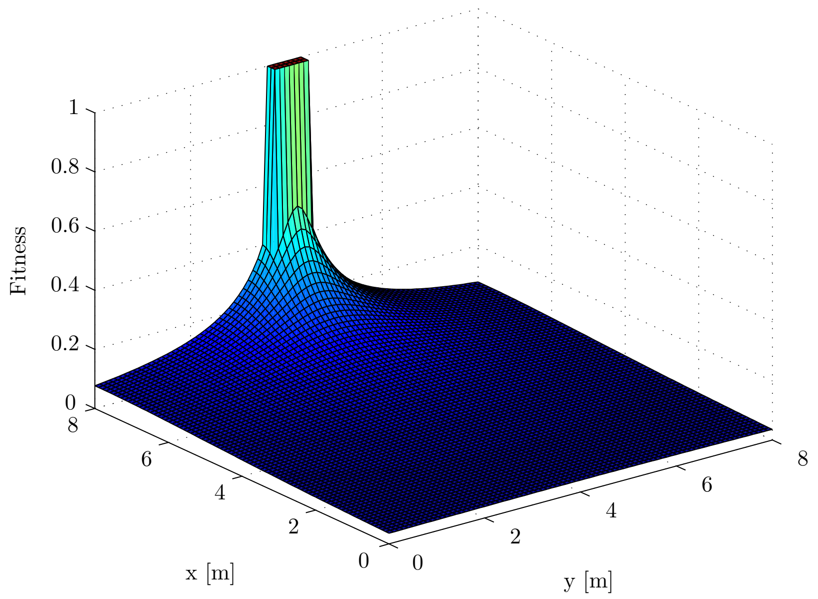


Figure 4-4: Fitness function used to train the population in the Evolutionary Learning system

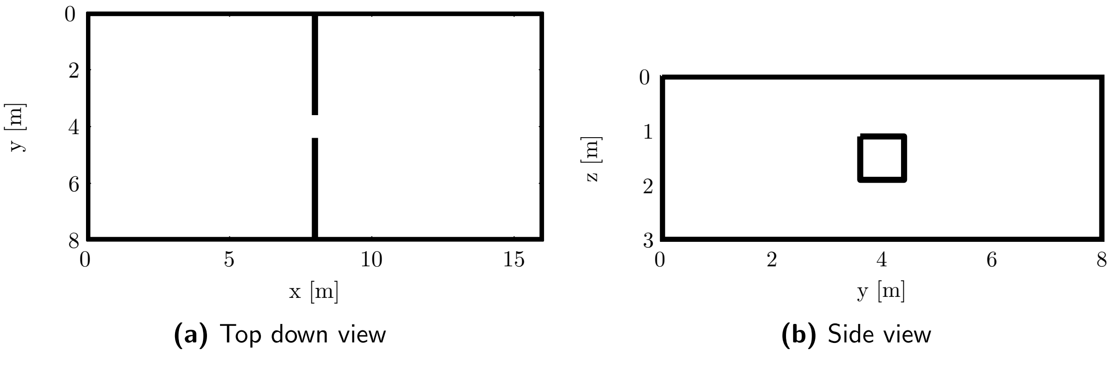
Chapter 5

DelFly Task Optimisation

This chapter aims to present and discuss the results of the genetic optimisation for the simulated DelFly fly-through-window task. We will first describe the simulated environment used to train the automated behaviour in Section 5-1. This will be followed by presenting the user-defined behaviour that will be used to compare the performance of the automated system in Section 5-2. Next, Section 5-3 the parameters used in the genetic optimisation will be stated. Finally, the results of the optimisation will be presented and discussed in Section 5-4.

5-1 Simulated 3D Environment

The environment chosen to train the UAV in simulation was an $8 \times 8 \times 3m$ room with textured walls, floor and ceiling. A $0.8 \times 0.8m$ window was placed in the centre of one wall. Another identical room was placed on the other side of the windowed wall to ensure the stereo algorithm had sufficient texture to generate matches for the disparity map when looking through the window. A schematic of the room can be seen in Figure 5-1 below.



(a) Top down view **(b)** Side view

Figure 5-1: Schematic of test environment for the fly-through-window task

As it is not the focus of this research to focus on the vision systems to provide the stereo vision algorithm some texture, a multi-coloured stone texture pattern was used for the four walls, a wood pattern was used for the floor and a concrete pattern used for the ceiling as shown in Figure 5-2. The identically textured walls ensure that the behaviour must identify the window and not any other features to aid in its task. Figure 5-3 below shows a picture of the room from the centre of the coordinate axis in the room.

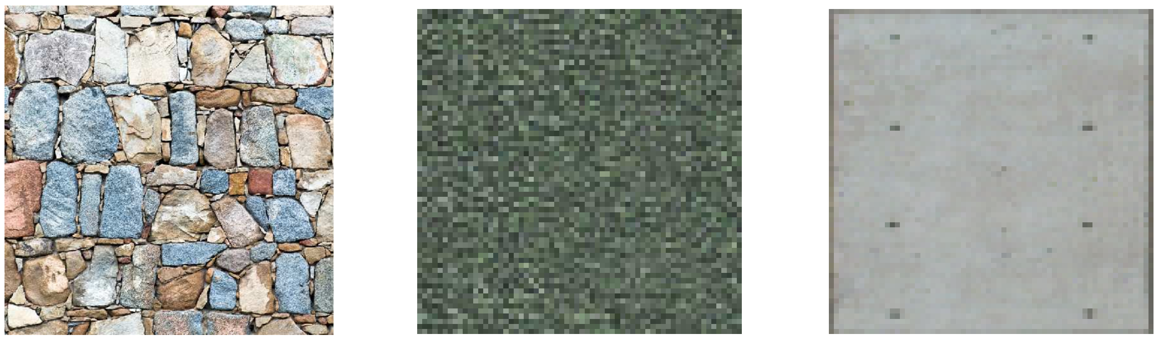

(a) Stone pattern used on walls **(b)** Carpet pattern used on floor **(c)** Concrete pattern used on ceiling

Figure 5-2: Texture used to decorate simulated flight test environment

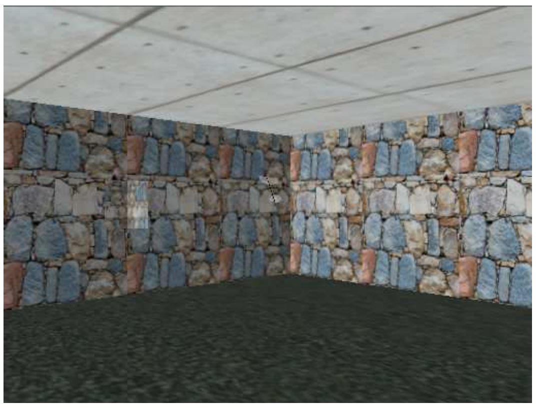


Figure 5-3: Image of room from the origin with the window and DelFly and target window in view

5-2 User-Defined Behaviour Tree

A human made behaviour was required to be designed which would be used as a control in comparison to the genetically optimised solution. The designed tree had 22 nodes and the structure of the BT can be seen graphically in Figure 5-4. The behaviour is made up of four main parts namely:

- 🟢 window tracking based on window response and location in frame - try to keep the window in the centre of the frame
- 🔵 default go straight when disparity very low - helps when looking directly through window into next room

- 🔴 wall avoidance when high disparity - bidirectional turns to avoid collisions with walls, also helps to search for window
- ● action hold when disparity very high - ensures the chosen action is not changed when already evading a wall

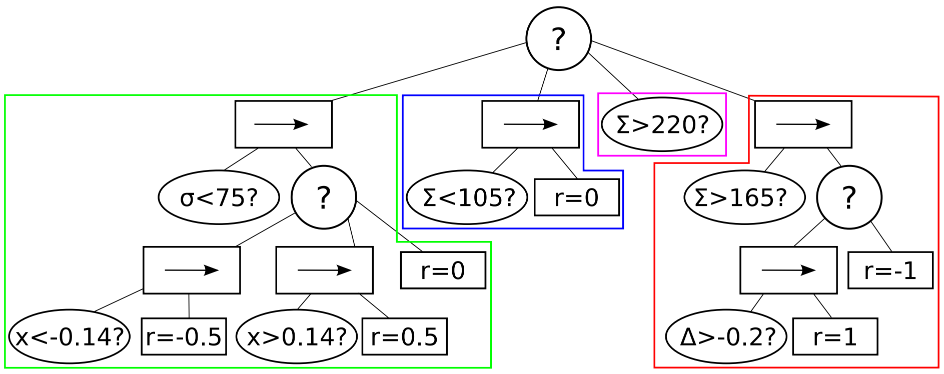
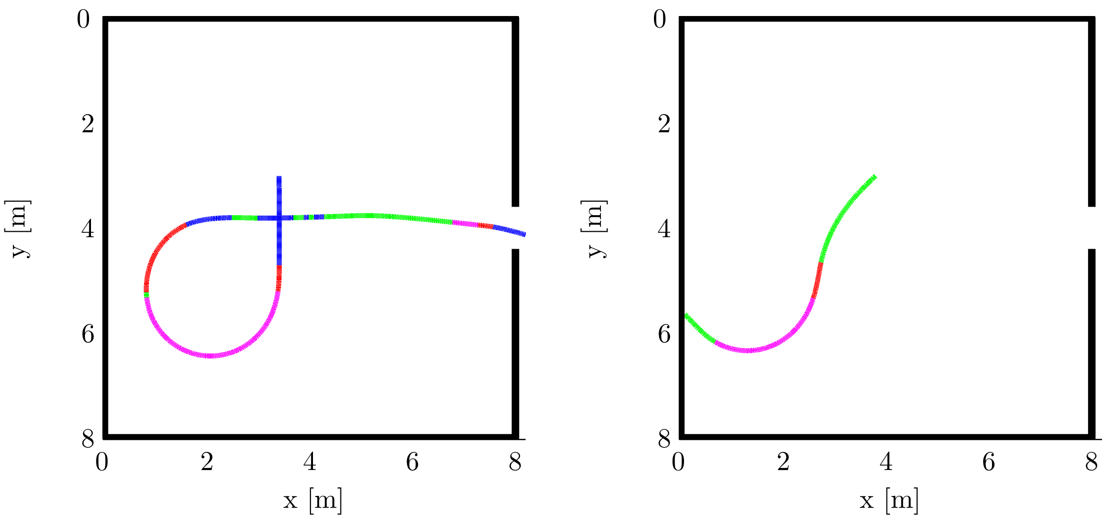


Figure 5-4: Graphical depiction of user-defined BT. Colours highlight different phases of the flight. x is the position of the centre of the window in frame, σ is window response value, Σ is sum of disparity and Δ is the horizontal difference in disparity

After validation of the behaviour, it was observed that for 250 random initialisations in the simulated environment, the behaviour successfully flew through the window in 205 of the runs, which translates to a success rate of 82%. Figure 5-5 shows a plot of the path of the DelFly with the user-defined behaviour in two random initialisations, one which resulted in a successful flight and the other a failure.



(a) Path of a successful flight **(b)** Path of an unsuccessful flight

Figure 5-5: Path of two flight initialisations of DelFly with the user-defined behaviour (top-down view). Colours denote different decision modes: Green - window tracking; Blue - default action in low disparity; Red - wall avoidance; Magenta - action hold

It can be seen in Figure 5-5a that the DelFly does actively avoid walls and tracks the window well resulting in a near perpendicular window entry from the centre of the room. This is quite favourable behaviour which quickly identifies and tracks the window as desired. Figure 5-5b however highlights one the pitfall of bidirectional wall avoidance in a square room with simple reactive behaviour. This particular initialisation saw the DelFly evade the wall to its left only to then react to the wall on its right resulting in a final collision with the original wall. This type of behaviour can be rectified using different techniques but we will keep the behaviour unchanged for further analysis as it is representative of true human designed behaviour.

5-3 Experimental Setup

As the DelFly must fly through the window as often as possible, to evaluate this we will simulate the DelFly multiple times per generation to better estimate its generalised performance. Each generation has k simulation runs to evaluate the performance of each individual. Each run is characterised by a randomly initiated location in the room and a random initial pointing direction. Individual initial conditions are held over multiple generations until the elite members of the population (characterised by P_e) are all successful, in which case the initial condition in question is replaced by a new random initialisation. Each simulation run is terminated when the DelFly crashes, flies through the window or exceeded a maximum simulation time of $100s$.

The characteristic parameters are defined below in Table 5-2. These values were chosen after observing the effect of the parameters after several runs of the EL.

Table 5-1: Table showing parameter values for the EL run

Parameter	Value
Max Number of Generations (G)	150
Population size (M)	100
Tournament selection size (s)	6%
Elitism rate (P_e)	4%
Crossover rate (P_c)	80%
Mutation rate (P_m)	20%
Headless-Chicken Crossover rate (P_{hcc})	20%
Maximum tree depth (D_d)	6
Maximum children (D_c)	6
No. of simulation runs per generation (k)	6

5-4 Optimisation Results

The main parameter which dictates the progress of the genetic optimisation is the mean fitness of the individuals in the population. Figure 5-6 shows the population mean fitness as well as the mean fitness of the best individual in each generation. It can be seen in Figure 5-6 that at least one member of the population is quickly bred to fly through the window quite often. Additionally, as the the generations progress and new initialisations are introduced the trees have to adjust their behaviour to be more generalised. The mean fitness also improves initially then settles out at around the 0.4 mark, the fact that this value doesn't continue to increase suggests that the genetic diversity in the pool is sufficient.

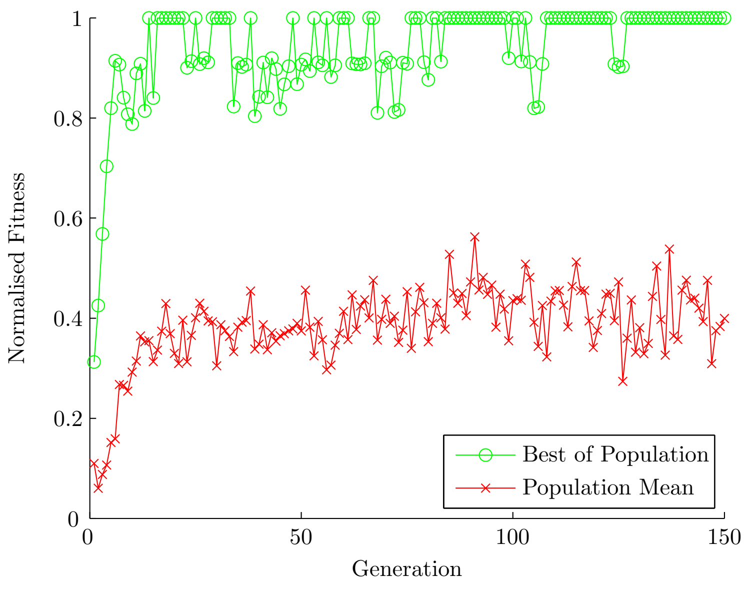


Figure 5-6: Progression of the fitness score of the best individual and the mean of the population

The other main parameter which defines the proficiency of the BTs is the tree size. The mean tree size of the population as well as the tree size of the best individual from each generation is shown below in Figure 5-7.

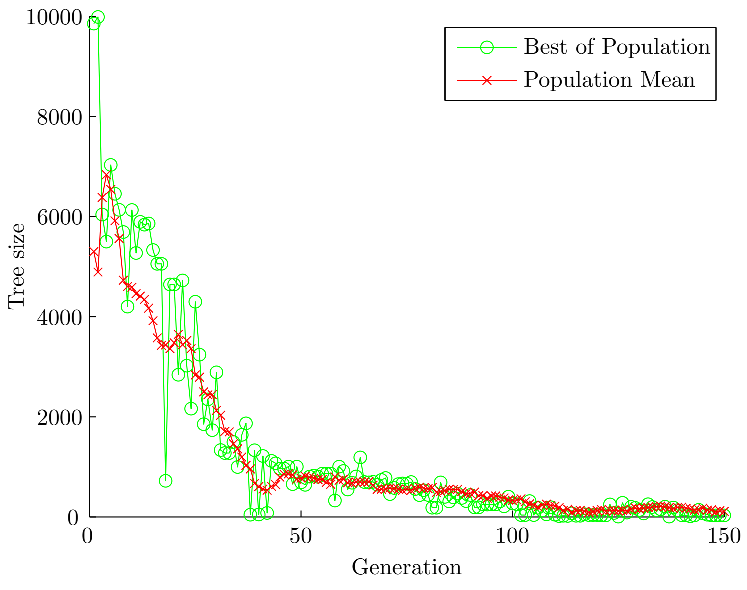


Figure 5-7: Progression of the number of nodes in the best performing tree and the mean of the population

This figure shows that the average tree size began at about 5000 nodes and initially increases to 7000 before steadily dropping to around 1000 nodes around generation 50. The trees size then slowly continues to reduce in size and eventually drops below 150 nodes. The best individual in the population oscillated around this mean value. The best individual after 150 generations had 32 nodes. Pruning this final BT resulted in a tree with 8 nodes. This process involves removing redundant nodes that have no effect on the final behaviour. A detailed description of this pruning process can be found in Appendix C. The resultant BT structure of the tree can be seen graphically in Figure 5-8.

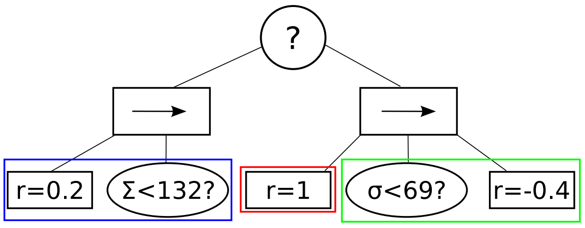


Figure 5-8: Graphical depiction of genetically optimised BT. Colours highlight different phases of the flight. x is the position of the centre of the window in frame, σ is window response value, Σ is sum of disparity and Δ is the horizontal difference in disparity

The optimised BT was put through the same validation set as used with the user-defined tree to compare their performance. The genetically optimised behaviour had a performance score of 88%. Figure 5-9 shows the progression of the validation success rate for the best individual of each generation. It can be seen that the score quickly increases and oscillates around about 80% success. In early generations the variation of success rate from one generation to the next is larger than later generations.

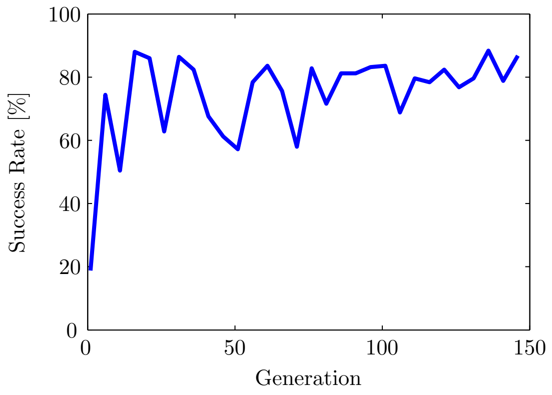


Figure 5-9: Progression of validation score of the best individual of each generation

Figure 5-7 and Figure 5-9 suggest that that the population quickly converges to a viable solution and then continues to rearrange the tree structure to result in ever smaller trees. The fact that the best individual of each population does not improve much above the 80% mark possibly identifies that the method used to expose the population to a generalised environment is not sufficient. A method to make the initial conditions more "difficult" is by employing co-evolution of the initialisation as well as the robotic behaviour. This predator-prey type co-evolution may improve results. Alternatively, the fact that the behaviour does not continue to improve may also indicate that the sensory inputs used b the DelFly are not sufficient.

The performance characteristics of the best individual from the optimisation as compared to those from the user-defined BT is summarised below in Table 5-2. The optimised BT has slightly higher success rate than the user-defined BT but with significantly less nodes. The mean fitness is also better with the optimised BT, this difference can be more clearly seen in Figure 5-10 which shows a histogram of the fitness of the two BTs.

Table 5-2: Summary of validation results

Parameter	user-defined	genetically optimised
Success Rate [%]	82	88
Tree size	26	8
Mean flight time [s]	32	40
Mean approach angle [°]	21	34
Mean distance to window centre [m]	0.08	0.15

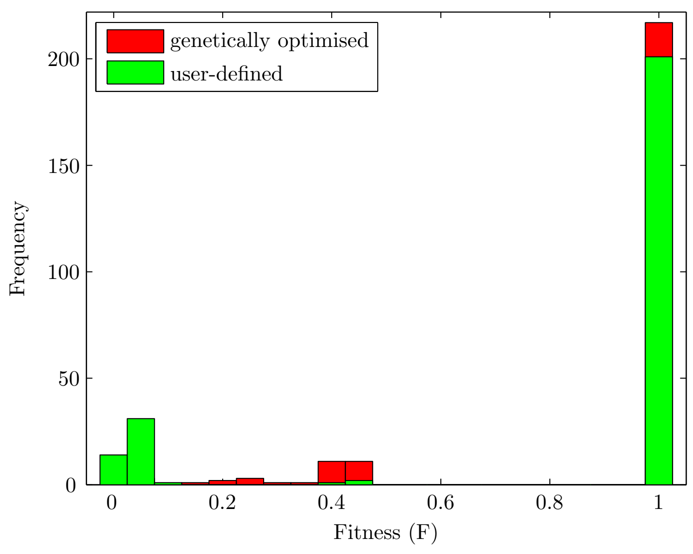
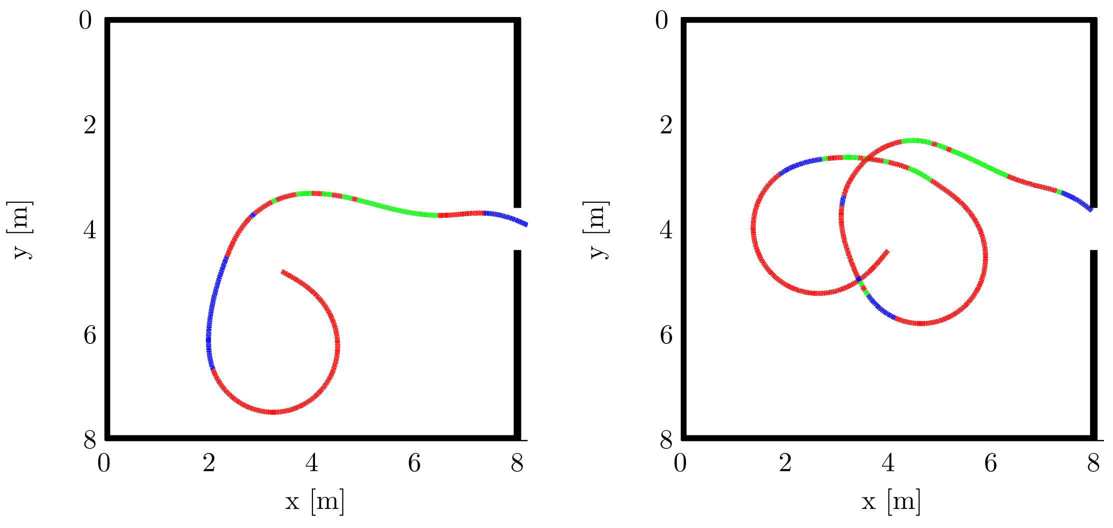


Figure 5-10: Histogram showing the distribution of fitness performance of the genetically optimised and user-defined behaviour for all initialisations of the validation

Figure 5-10 shows that when the user-defined tree is unsuccessful it typically has a very low fitness unlike the optimised system which typically has a relatively high fitness when not successful. This suggests that the user-defined tree has more failures in the wall avoidance phase of the flight and crashes into the wall far away from the window. The optimised system however seems to crash into the wall near to the window. To visualise this failure mode, Figure 5-11 shows a plot of a successful and an unsuccessful initialisations of the optimised BT.

Figure 5-11a shows that the behaviour correctly avoids collision with the wall, makes its way to the centre of the room and then tracks into the window. Analysing the BT itself, the logic to fly through the window is very simple, the tree can be separated into three phases:

- max right turn to evade walls if disparity is too high (unidirectional avoidance)



(a) Path of a successful flight **(b)** Path of an unsuccessful flight

Figure 5-11: Path of two flight initialisations of DelFly with the genetically optimised behaviour (top-down view). Colours denote different decision modes: Green - window tracking; Blue - default action in low disparity; Red - wall avoidance

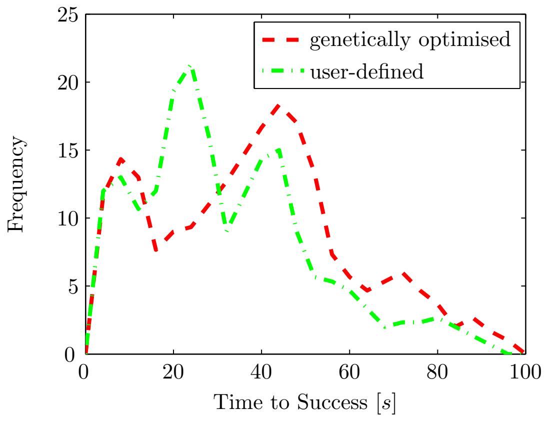
- if disparity relatively low slight right turn
- when window detected make a moderate left turn

This very simple behaviour seems to have very good success however Figure 5-12c highlights one pitfall of this solution. As the behaviour doesn't use the location of the window in the frame for its guidance it is possible to drift off centre and lose the window in frame and enter a wall avoidance turn quite close to the wall resulting in a collision.

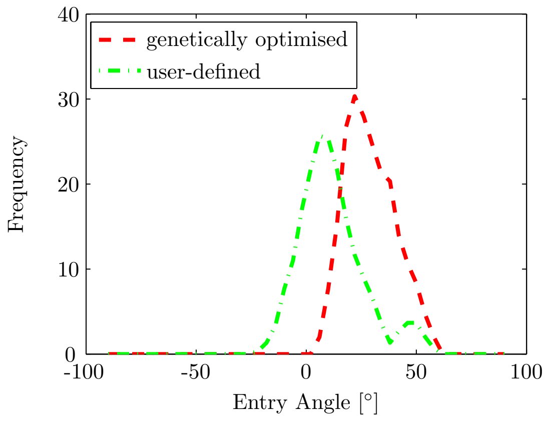
Figure 5-12 below shows that this behaviour generally has a longer time to success as a result of the DelFly making more loops before it is in a good position to make its approach. Additionally, the DelFly always approaches from the left of the window turning towards the window from an angle, this results in the DelFly entering the window at a relatively sharp angle and typically more towards the left side of the window than the user-defined behaviour.

These results show that based on the given fitness function and optimisation parameters the genetic optimisation was very successful. The resultant BT was both smaller and better performing than the user-defined tree. However, Figure 5-12 shows that the user-defined behaviour performs better when looking at the secondary parameters that were not explicitly described in the fitness function. This result highlights that the genetically optimised behaviour will optimise the fitness function so any parameters that the user wants to be optimised need to be included in that fitness function.

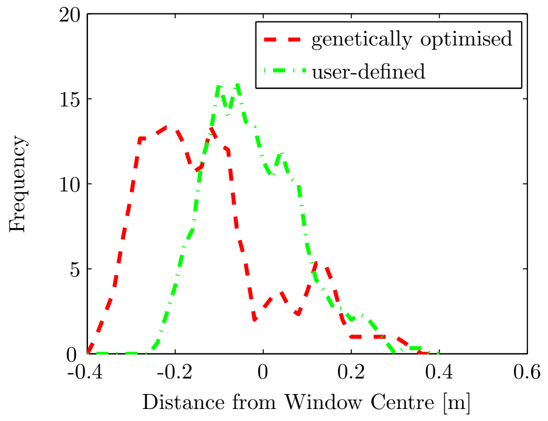
Combining many differing and possibly contradictory performance parameters into one fitness function can be difficult. A popular solution is to implement a multi-objective optimisation such as the Pareto approach (Sbalzarini et al., 2000; Tan et al., 2001; Fehervari et al., 2013). This approach often allows the user to optimise for multiple objective parameters simultaneously. This method could be used to implement a better trade-off scheme of generalised



(a) Distribution of flight time to success



(b) Distribution of mean window entry angle



(c) Distribution of flight distance to window centre

Figure 5-12: Distribution of secondary performance parameters for validation run

BT performance and tree size. The secondary parameters may also be included to promote behaviour preferred by the user.

5-5 Extended Analysis

The results of the optimisation described earlier raises some interesting questions: how robust is the optimised behaviour as compared to the larger user-defined behaviour to variation in the environment?; What would happen if we let the optimisation run with drafts introduced in the training runs?; what would result from genetically optimising the user-defined behaviour structure?. We will try to briefly investigate these questions in this section.

5-5-1 Alternate Validation

Genetically optimised behaviour is often suitable only for the behaviour that it was exposed to during its optimisation, in this section we investigate how the optimised behaviour described earlier performs when subject to changes in the operating environment. Three tests where done where validation was conducted with the original behaviours under different environmental or algorithmic changes. The first test involved validation with air drafts on, the second involved a doubling of the run frequency of the BT decision making module and the final test was done in a larger rectangular room in stead of the original square room.

The results of the test with random environmental drafts with maximum strength of $0.14m/s$ are shown below in Table 5-3. It can be seen that the success rate of both behaviours is reduced with the genetically optimised behaviour more significantly impacted by the drafts.

Table 5-3: Table showing performance parameter values for test with drafts

Parameter	user-defined	genetically optimised
Success Rate	76%	68%
Mean flight time $[s]$	37	43
Mean approach angle [°]	24	30
Mean distance to window centre $[m]$	0.13	0.16

Table 5-4 shows the performance parameters for a validation run with the BT module running at $10Hz$ increased from the original $5Hz$. The user-defined behaviour showed a marginal increase in performance suggesting that the original run frequency was sufficient for the given task while highlighting that a higher run frequency is typically better. The genetically optimised behaviour performance decreased slightly highlighting the fact that the optimised behaviour is coupled to the decision speed as well as the environmental and vehicular dynamics.

Table 5-4: Table showing performance parameter values for test with double decision frequency

Parameter	user-defined	genetically optimised
Success Rate	87%	75%
Mean flight time $[s]$	27	36
Mean approach angle [°]	27	36
Mean distance to window centre $[m]$	0.08	0.13

To test the performance in a different room we changed the dimensions of the room to $8\times16\times3$ and ran the validation process again. The results of the validation is summarised in Table 5-5. These results show that the user-defined behaviour is significantly degraded and is actually only successful in 21/250 simulation runs. This shows that the solution method selected by the user is suitable for rooms of different geometry, in this case an augmented solution would be required. It would require significant time for the user to redesign the behaviour for every room geometry the DelFly is exposed to. This highlights the need for automated development of behaviour. A benefit of EL is that the solution can be generalised for the room geometry as well as other parameters such as visual texture and environmental and vehicular dynamics simultaneously.

In general, these results highlight the fact that genetic optimisation will learn behaviour for the environment it has been exposed to. To promote generalised robust behaviour the optimisation must be presented with a wide range of differing environments that may be encountered in reality.

Table 5-5: Table showing performance parameter values for test with larger rectangular room

Parameter	user-defined	genetically optimised
Success Rate	8%	31%
Mean flight time $[s]$	22	35
Mean approach angle $[°]$	23	32
Mean distance to window centre $[m]$	0.37	0.15

5-5-2 Genetic Optimisation with Drafts

In reality the DelFly will be subject to drafts and other localised air disturbances. We saw earlier that the performance of the optimised behaviour drops slightly when exposed to drafts, it is therefore interesting to investigate how the behaviour would have to change to cope with this change in the environment.

To this end, the genetic optimisation was repeated for the fly-through-window with the drafts activated for all simulations with a maximum draft velocity of $0.14m/s$. The results of the optimisation can bee seen below in the following figures.

Figure 5-13 and Figure 5-14 show a similar trends in a quick optimisation of performance and tree size decay as seen with the original optimisation but the optimisation converges much earlier. Taking taking a closer look at Figure 5-15, we can see the effect of overtraining is more evident in this optimisation. More research should be done to investigate methods to reduce this effect, possibly into an implementation of the pareto multi-objective fitness optimisation.

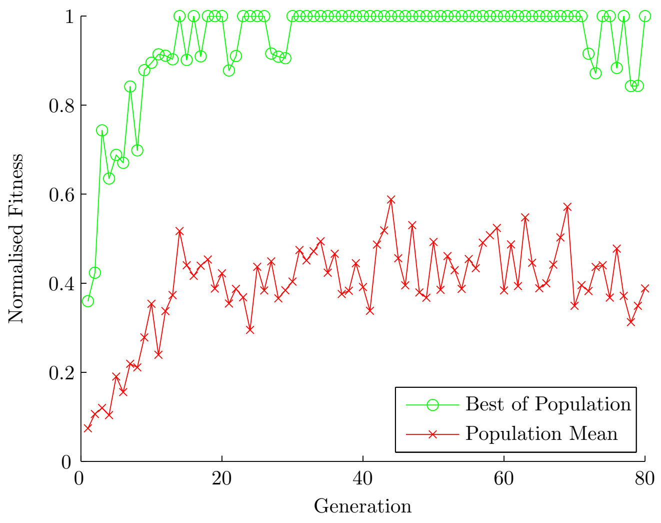


Figure 5-13: Progression of the fitness score of the best individual and the mean of the population throughout the genetic optimisation

5-5-3 Genetically Optimised User-Defined Behaviour

The user-defined BT was designed based on the desired behaviour of the DelFly but certain deficiencies were identified in this behaviour such as being caught in corners. It is of course

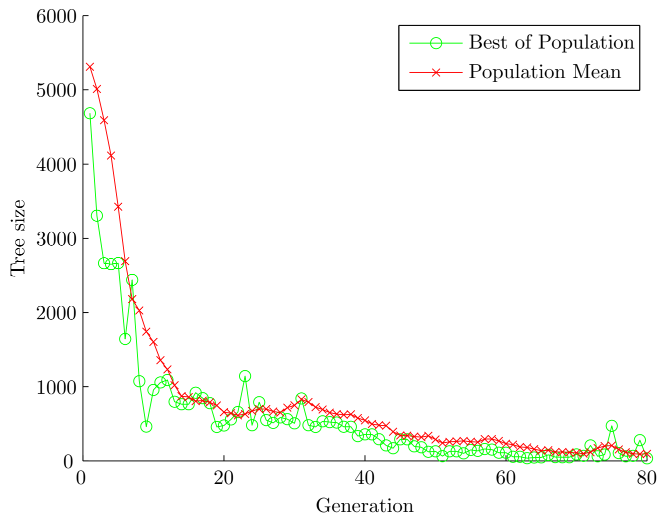


Figure 5-14: Progression of the number of nodes in the best tree and the mean of the population

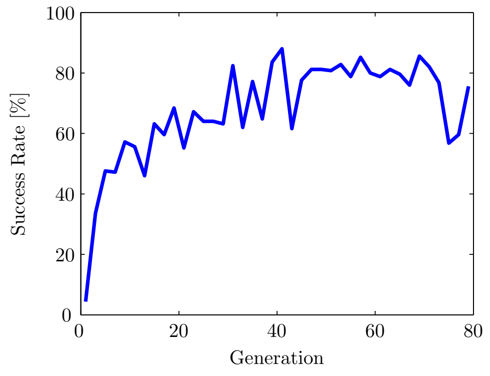


Figure 5-15: Progression of validation score of the best individual of each generation

possible to optimise the user-defined tree structure using the same EL techniques used to optimise the random population of individuals. The tree structure will be kept fixed but the values within the individual nodes are optimised applying micro-mutation only evolution. The characteristic parameters used for the optimisation are defined below in Table 5-6.

The optimisation resulted in a BT with improved performance as can be seen in Table 5-7. The success rate improved by 8% achieving almost perfect success record however the optimised behaviour was quite different than the original behaviour designed by the user. Figure 5-16 shows that the distribution of the performance parameter values for for the validation is quite different for the optimised BT than for the original in particular the window entry angle is noticeably sharper and more tightly grouped.

Looking at the BT as shown in Listing 5.1, some changes to the effective behaviour can be seen. The default action was to turn max right instead of straight flight as originally described. The section of code used to hold the decision making of the DelFly when too close to obstacles was set to such a high disparity that it would never be encountered in flight and therefore was equivalently removed. The optimised tree did interestingly keep the bidirectional wall avoidance behavioural feature as described in the original BT. With these changes the tree

Table 5-6: Table showing parameter values for the GP run

Parameter	Value
Max Number of Generations (G)	10
Population size (M)	100
Tournament selection size (s)	6%
Elitism rate (p_e)	4%
Crossover rate (p_c)	0%
Mutation rate (p_m)	20%
Headless-Chicken Crossover rate (p_{hcc})	0%
Maximum tree depth (D_d)	6
Maximum children (D_c)	6
No. of simulation runs per generation (k)	6

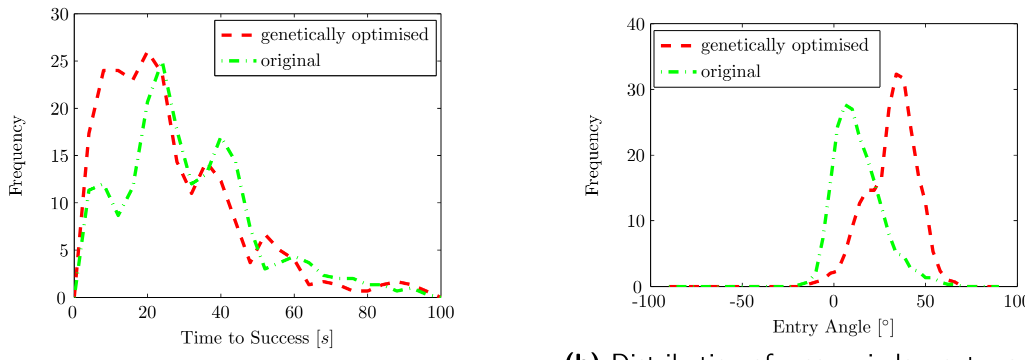
Table 5-7: Table showing performance parameter values for original and genetically optimised versions of the user-defined behaviour

Parameter	original	genetically optimised
Success Rate	82%	94%
Mean flight time $[s]$	33	27
Mean approach angle [°]	16	34
Mean distance to window centre $[m]$	0.09	0.1

could be equivalently pruned to 18 nodes instead of the original 22.

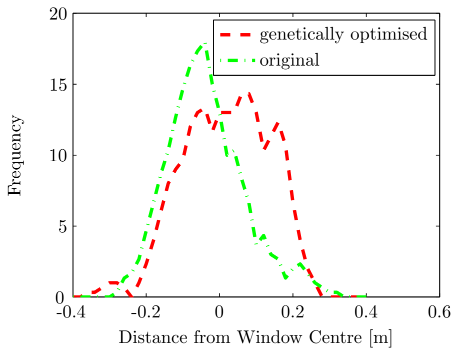
```
<BTtype>Composite<function>Selector<vars><name>Selector<endl>
   <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
      <BTtype>Condition<function>less_than<vars>75,1<name>DelFly<endl>
      <BTtype>Composite<function>Selector<vars><name>Selector<endl>
         <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
            <BTtype>Condition<function>less_than<vars>-0.1,0<name>DelFly<endl>
            <BTtype>Action<function>turnRate<vars>-0.5<name>DelFly<endl>
         <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
            <BTtype>Condition<function>greater_than<vars>0.66,0<name>DelFly<endl>
            <BTtype>Action<function>turnRate<vars>1<name>DelFly<endl>
         <BTtype>Action<function>turnRate<vars>0.01<name>DelFly<endl>
   <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
      <BTtype>Condition<function>less_than<vars>104,2<name>DelFly<endl>
      <BTtype>Action<function>turnRate<vars>1<name>DelFly<endl>
   <BTtype>Condition<function>greater_than<vars>376,2<name>DelFly<endl>
   <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
      <BTtype>Condition<function>greater_than<vars>60,2<name>DelFly<endl>
      <BTtype>Composite<function>Selector<vars><name>Selector<endl>
         <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
            <BTtype>Condition<function>greater_than<vars>-0.2,3<name>DelFly<endl>
            <BTtype>Action<function>turnRate<vars>1<name>DelFly<endl>
         <BTtype>Action<function>turnRate<vars>-1<name>DelFly<endl>
```

Listing 5.1: DelFly genetically optimised user-defined BT for fly-through-window task. Lines highlighted in red have been changed in the optimisation



(a) Distribution of flight time to success

(b) Distribution of mean window entry angle



(c) Distribution of flight distance to window centre

Figure 5-16: Distribution of performance parameters for original and genetically optimised versions of the user-defined behaviour

Chapter 6

DelFly Onboard Flight Testing

To investigate the impact of the BT to reduce the reality gap a real world test must be carried out. This chapter describes the DelFly Explorer fly-through-window flight test set-up and results. First, we will describe some of the differences between the simulation based implementation of the BT framework and that used onboard in Section 6-1. This is followed by a description of the 3D environment used for the flight test in Section 6-2. Section 6-3 presents the experimental set-up used in the flight tests. Finally the results of the flight tests are presented and discussed in Section 6-4.

6-1 Onboard Behaviour Tree Implementation

The BT was implemented on the camera module of the DelFly Explorer which is equipped with a STM32F405 processor operating at $168MHz$ with $192kB$ RAM. This processor is programmed in the `C` programming language as opposed to the `C++` implementation used in simulation. As a result a slightly simplified version of the BT system was implemented onboard.

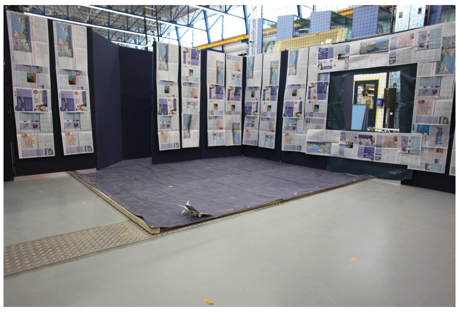
The same node types and functionality was implemented as described in the previous section. The main difference is seen in that the STM32F405 processor does not utilise a floating point unit so all comparisons and value returns were changed to integer math. Tthe output of the window detection algorithm is the pixel number of the centre of the window so the mapping of [-1,1] in simulation is converted to $[0, 128]$ on the DelFly. The sum and differences in disparity are now scaled to 16 bit integers. The action node returned values mapped onto the space $[0, 255]$ where 0 was full left turn and 255 full right turn.

The BT node is placed in series with the stereo vision and window detection algorithms and was found to run at ~$12Hz$. The commands were sent from the camera module to the DelFly Explorer flight control computer using serial communication. The DelFly flight control computer reads this command at about $100Hz$ and scales the command onto the limits of the aileron actuators.

6-2 Test 3D Environment

The environment designed to test the UAV was a $5 \times 5 \times 2m$ room with textured walls. The floor was simply a concrete floor and the ceiling was left open. A 0.8×0.8 m window was placed in the centre of one wall. The area behind the window was a regular textured area. As the focus of this work is on investigating the development behaviour of the DelFly and not on the vision systems themselves, we added artificial texture to the environment to ensure we had good stereo images from the DelFly Explorer onboard systems. This texture was in the form of newspapers draped over the walls at random intervals. Sample photographs of the room can be seen below in Figure 6-1.

(a) Photograph taken from $(0, 0, 1)$


(b) Photograph taken from $(0, 5, 1)$

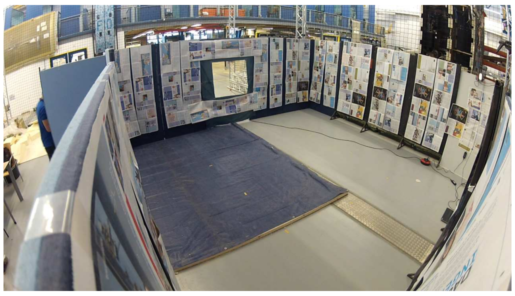

(c) Photograph taken from $(0, 0, 2)$

Figure 6-1: Photographs showing the room environment used to test the DelFly Explorer for the fly-through-window task

6-3 Experiment Set-up

At the beginning of each run, the DelFly will initially be flown manually to ensure it is correctly trimmed for flight. It will then be flown to a random initial position and pointing direction in the room. At this point the DelFly will be commanded to go autonomous where the DelFly flight computer implements the commands received from the BT. The flight will continue until the DelFly either succeeds in flying through the window, crashes or the test takes longer than $60s$. As the BT controls the horizontal dynamics only, the altitude is actively controlled by the user during flight, this was maintained around the height of the centre of the window.

All flights are recorded by video camera as well as an Optitrack vision based motion tracking system (NaturalPoint, Inc, 2014). The motion tracking system was used to track the DelFly as it approached and flew through the window to determine some of the same metrics of performance that were used in simulation. As a result, information on the success rate, flight time, angle of approach and offset to the centre of the window can be determined.

6-4 Flight Test Results

The flight speed of the DelFly was set to ~$0.5m/s$, the same as was used in simulation apart from this however there were significant differences observed between the DelFly simulated in SmartUAV and the DelFly used in the flight tests. The most significant was that the maximum turn radius was ~$0.5m$ much smaller than the $1.25m$ as in simulation. Additionally, the ailerons on the DelFly Explorer had a faster response rate than the low pass filter set on the rudder dynamics in simulation. It was also observed that the aileron deflection was not symmetrical, the DelFly would turn more effectively to the right than it did to the left. Aileron actuation would also result in a reduction in thrust meaning that active altitude control was required from the user throughout all flights. It was also observed that there were light wind drafts observed around the window which affected the DelFly's flight path. This draft would typically slow the DelFly forward speed slightly and push the DelFly to one side of the window.

With these significant differences between the model used to train the BTs and the real DelFly there as a clear reality gap present. Initially both behaviours were not successful in flying through the window, the behaviour thresholds had to be adjusted to ensure that the DelFly would behave similar to that in simulation. This was done first for the user-defined behaviour, the updated behaviour can be seen in Figure 6-2. It was required to adjust the turn rate set points to try to achieve a more symmetrical wall avoidance behaviour. Due to the different environment in reality the window response at which the DelFly started its window tracking was also raised slightly. The threshold at which the DelFly started its wall avoidance was also changed to ensure the DelFly could evade walls. These changes helped push the behaviour in reality towards that observed in simulation. In total, it took about 8 flights of about 3 minutes each to tune the parameters of the behaviour.

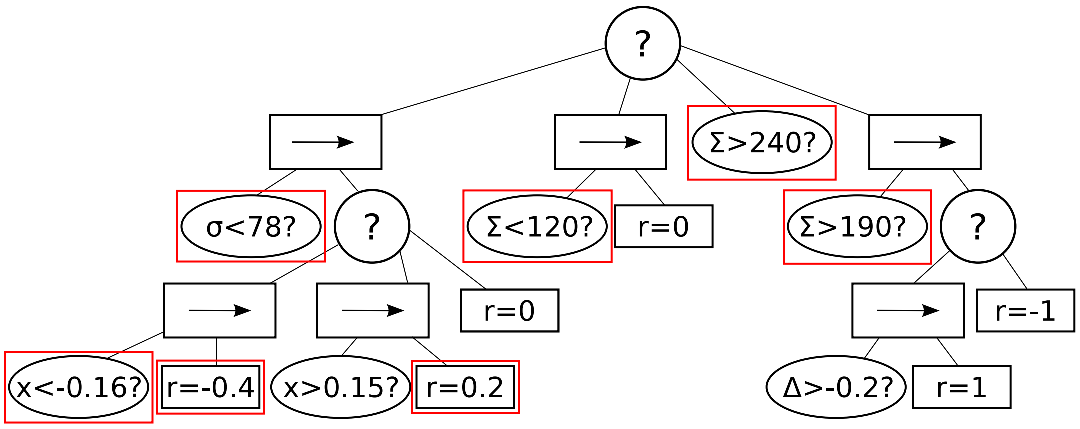


Figure 6-2: Graphical depiction of user-defined BT after modification for real world flight. Red boxes highlight updated nodes. x is the position of the centre of the window in frame, σ is window response value, Σ is sum of disparity and Δ is the horizontal difference in disparity

A similar process was done for the genetically optimised behaviour. As the parameters for the wall avoidance was the same for both behaviours, the changes to this could be done before any flight tests. As a result, only the window tracking turn rate, default turn rate and the window response values had to be tuned. These parameters took about 3 flights to tune to result in behaviour similar to that seen in simulation. The updated behaviour can be seen in Figure 6-3.

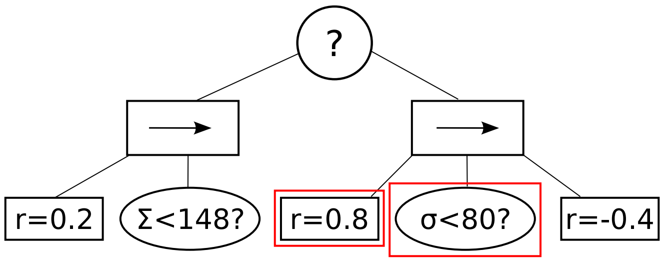


Figure 6-3: Graphical depiction of genetically optimised BT after modification for real world flight. Red boxes highlight updated nodes. x is the position of the centre of the window in frame, σ is window response value, Σ is sum of disparity and Δ is the horizontal difference in disparity

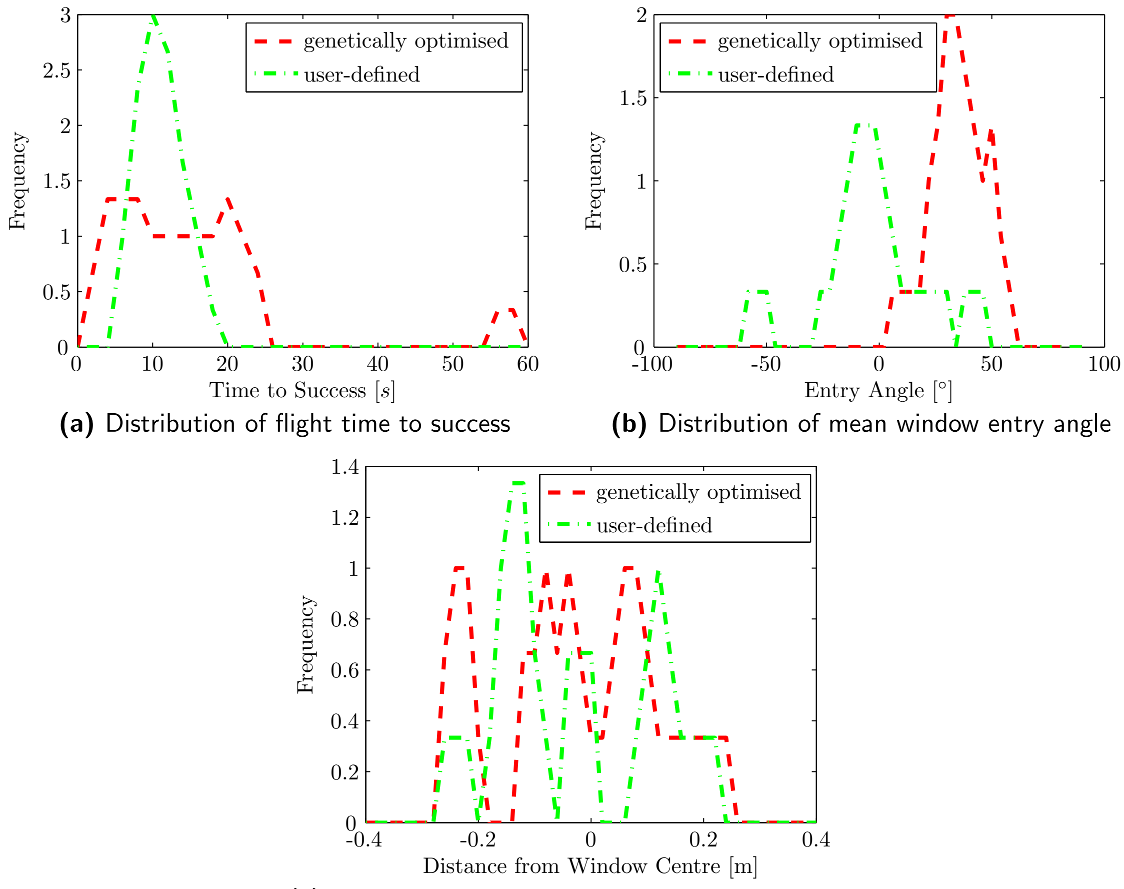
After an initial training session where the thresholds were re-optimised to real flight, 26 test flights were conducted for both the user-defined behaviour as well as the genetically optimised BT. The results of the tests are summarised below in Table 6-1.

Table 6-1: Summary of flight test results

Parameter	user-defined	genetically optimised
Success Rate [%]	46	54
Mean flight time [s]	12	16
Mean approach angle [°]	16	37
Mean distance to window centre [m]	0.12	0.12

It can be seen that the success rate of both behaviours is reduced success rate but the other performance parameters are similar to that seen in simulation. The relative performance of the behaviours is also similar to that seen in simulation. The mean flight time of the behaviours was reduced but notably the relative flight times of the behaviours is the same as seen in simulation. The reduction in the time to success can be explained by the reduced room size and increased turn rate of the DelFly seen in reality as opposed to that in simulation.

The mean angle of window entry is also similar to that observed in simulation. The mean distance to the centre of the window was higher for the user-defined behaviour than seen in simulation. This can be as a result of the drafts seen around the window pushing the DelFly to the edges of the window in the last phase of the flight when the window was too close to be in view. This conclusion is reiterated in Figure 6-4 which shows the distribution of the performance parameters for the successful flights of both behaviours. Figure 6-4c shows a more randomly distributed entry location than that seen in simulation. Figure 6-4a shows a more tightly grouped time to success than in simulation which is as a result of the smaller test area. Figure 6-4b shows that the angel of entry of the genetically optimised behaviour is grouped quite similarly to that seen in simulation while the distribution of the user-defined behaviour is more distributed. The increased distribution seen in Figure 6-4b and Figure 6-4c



(a) Distribution of flight time to success

(b) Distribution of mean window entry angle

(c) Distribution of flight distance to window centre

Figure 6-4: Distribution of secondary performance parameters from flight test

could also be as a result of the significantly smaller statistical data set used for the real flight test than that as a result of the simulation.

The failed flights of both behaviours can be characterised into 3 types: *hit right side of window*; *hit left side of window*; *hit wall in other part of room.* Table 6-2 below shows the frequency of the failure modes for each behaviour. Notably, the user-defined behaviour showed similar failure as seen in simulation characterised by being caught in corners, this happened 4/26 flights for the user-defined behaviour but not once in the genetically optimised behaviour.

Table 6-2: Summary of flight test failure cases

Failure Case	user-defined	genetically optimised
Hit right side of window	8	5
Hit left side of window	2	7
Hit wall	4	0

The Optitrack flight tracking system did not successfully track the DelFly in all portions of the room resulting in some dead areas but did accurately capture the final segment of the window approach. Figure 6-5 shows the last $7s$ of the user-defined behaviour for all flights grouped in successful and unsuccessful tests. These plots show that the DelFly tried to approach and fly through the window from various areas of the room at various approach angles. Approaches from areas of high approach angle typically resulted in a failed flight as the DelFly would hit the edge of the window. Additionally, the crashes in the wall due to being caught in corners can also be seen. Figure 6-6 show typical flight paths of successful and unsuccessful flights of the user-defined behaviour in more detail.

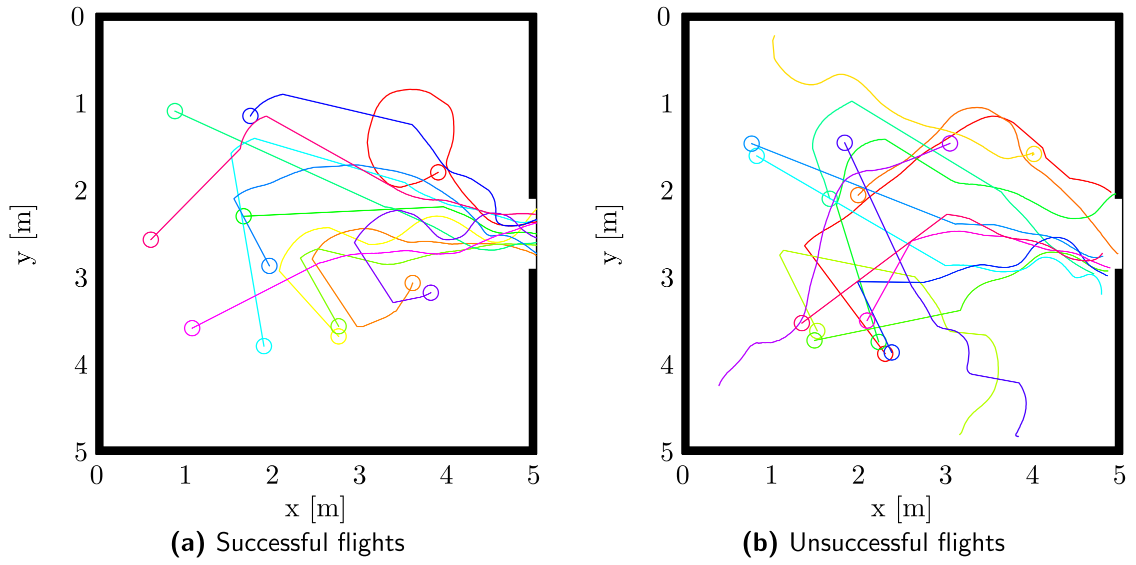


Figure 6-5: Flight path tracks of the last 7s of all flights for the user-defined behaviour

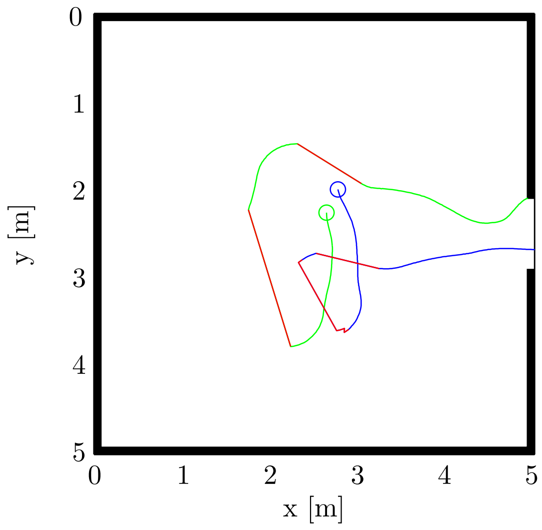


Figure 6-6: Flight path tracks showing a sample failure and success flight case for the user defined behaviour. Red track shows area where tracking system lost lock of the DelFly

Similarly, Figure 6-7 shows the successful and unsuccessful flights of the genetically optimised behaviour as captured from the Optitrack system. In these figures it can be seen that the flight tracks of genetically optimised behaviour are tightly grouped with the same behaviour repeated over multiple flights. The DelFly always approaches from about the centre of the room with a coordinated left-right turn described earlier. It can be seen that some of the unsuccessful flights occur when the DelFly makes an approach from farther way than normal so the coordination of the left-right turning is out of sync causing the DelFly to drift off course and hit the window edge. Figure 6-8 show typical flight paths of the genetically optimised behaviour in more detail. The typical failure mode was turning into the edge of the window in the final phase of the flight.

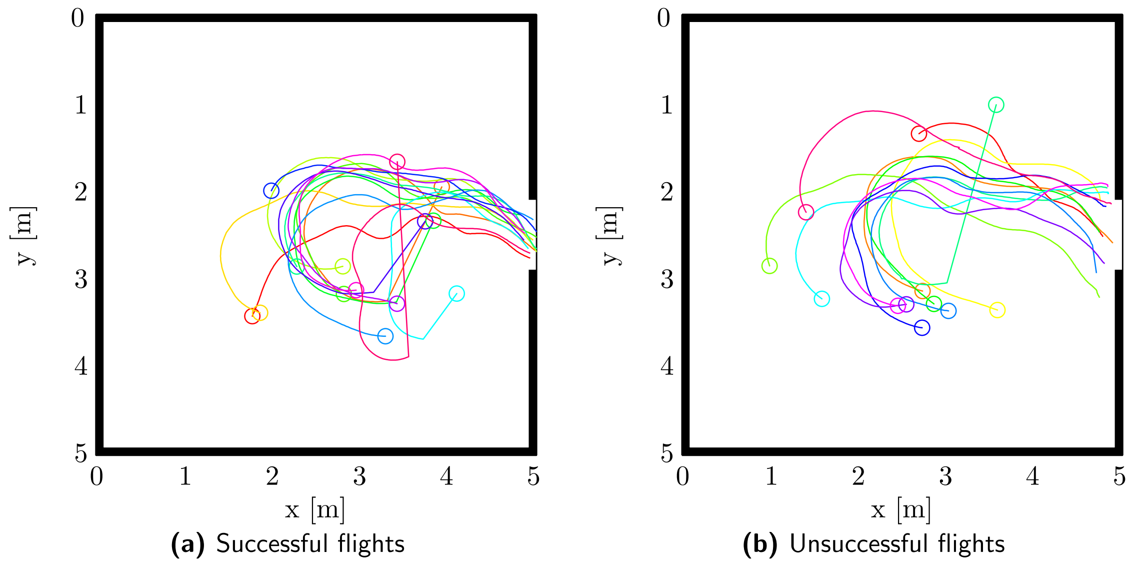


Figure 6-7: Flight path tracks of the last 7s of all flights for the genetically optimised behaviour

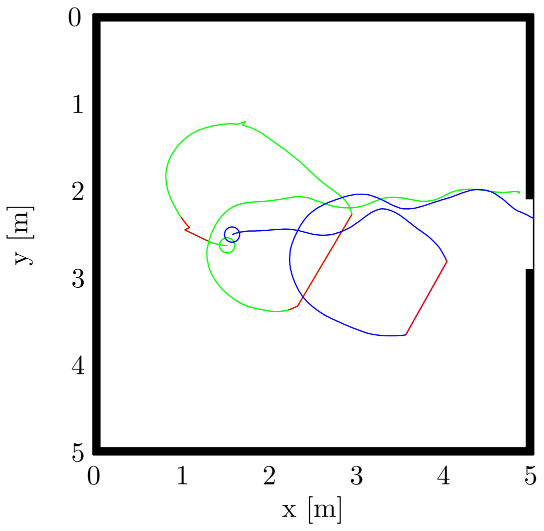


Figure 6-8: Flight path tracks showing sample failure and success flight case for the genetically optimised behaviour. Red track shows area where tracking system lost lock of the DelFly

The failure mode of hitting into the window edge for both behaviours can be in part the result of the drafts observed around the window or in part due to the lack of detailed texture around the window. These external factors would affect the two behaviours equally so would not affect the comparison of behaviours.

The fact that the behaviours where not initially able to fly through the window and were able to fly through more than 50% of the time after user optimisation shows that the reality gap was actively reduced by the user. These results show that it is feasible to automatically evolve behaviour on a robotic platform in simulation using the BT description language. This method gives the user a high level of understanding of the underlying behaviour and the tools to adapt the behaviour to improve performance and reduce the reality gap. Using this technique an automated behaviour was shown to be as effective as a user-defined system in simulation with similar performance on a real world test platform.

Chapter 7

Conclusion

In this thesis, we performed the first investigation into using Behaviour Trees for Evolutionary Robotics. A particular focus was whether the BT would help reduce the reality gap when transferring the controller evolved in simulation to a real robot. The presented experimental results show that the genetically optimised behaviour tree had a simulation based success rate of 88%, slightly better than user-defined behaviour at 82%. When moving the behaviour to the real platform, a large reality gap was observed as the success rate dropped to almost nil. After user adaptation, the genetically optimised behaviour had a success rate of 54%. Although this leaves room for improvement, it is higher than 46% from a tuned user-defined controller.

These results show that it is feasible to automatically evolve behaviour on a robotic platform in simulation using the BT encoding language. This method gives the user a high level of understanding of the underlying behaviour and the tools to adapt the behaviour to improve performance and reduce the reality gap.

We aimed to answer the research question: *How can a Behaviour Tree framework be used to develop an effective automatically generated Artificial Intelligence UAV control system to reduce the reality gap of simulation trained systems?*

This work has two main contributions, namely: a Behaviour Tree was implemented on a airborne robotic platform to perform a window search and fly-through task all on the basis of onboard sensors and processing; the ability to reduce the reality gap of robotic behaviour using Behaviour Trees was effectively demonstrated.

Future work will attempt to improve on the final success rate of 54%. The use of an onboard turn rate controller and further investigation into the Evolutionary Learning techniques used should help to improve the real world performance. Additionally, although this method was applied to a singular, arguably rather simple task it is not limited to such tasks. The inherent hierarchy of the BT framework lends itself naturally to building more complex behaviour from a combination of simple behaviours. The simple behaviours, as well, as the combination of simple behaviours can be determined using the same EL technique. This hierarchical combination of tasks can be utilised to reduce the search space optimised by the learning algorithm.

The use of BTs as a behaviour encoding framework gives the user increased flexibility and scalability of robotic behaviour facilitating further development. This tool can be very useful in future robotic behaviour development.

Chapter 8

Recommendations

General The modular design and make-up of behaviour trees as a directed graph lends itself naturally to the use of a visual editor to create and edit the tree design. Future research should go into a Graphical User Interface (GUI) to provide users with an intuitive method to understand and interactively design BTs, this will add many possible advances to a user's fault detection and comprehension as well as adding advanced debugging tools to the BT implementation.

The main aim of this paper was to show that the BT encoding framework can be a useful tool for a user to reduce the reality gap of a simulation trained robotic behaviour. Although this was successful, it was also shown that the solution behaviour for a particular task is strongly coupled to the dynamics of the vehicle and environment. Based on this observation, it may be possible that the reality gap between the optimised solution and reality requires a different solution behaviour. This observation validates other research done on reducing the reality gap which states that the simulation should accurately reflect reality in order to have a valid solution emerge from the optimisation.

The BT is comprised of many nodes with parameters that have to be tuned from simulation to the real world platform to reduce the reality gap. Some of these parameters are more important to the final effective behaviour of the platform than others. To help highlight to the user which nodes are critical to the behaviour a sensitivity analysis should be performed. This analysis will also identify parts of the behaviour that may be affected by additional parameter noise that may occur on the real platform.

Future research should investigate how this system can be implemented in a safety critical manner either by investigating the flow of the tree itself as described by (Klöckner, 2013b) or by having some safe fall back behaviour for the guidance system to default to.

BT Implementation The behaviours evolved in this paper were strictly reactive in that no internal state was explicitly stored in the tree, this however could be easily added to the tree using the shared BlackBoard. Simple state parameters such as timers and memory could be added to the tree. This could be used to improve the agent's ability to handle discrete events or to some precept history. Additionally, the action nodes were limited to lateral only control, it would be interesting to investigate the effect of adding altitude control to this as well.

One parameter which heavily influenced the reality gap was the effective turn rate of the DelFly. The BT should control parameters that are more robust to environmental changes. It may be more effective to control the turn rate operating within a control loop rather than the aileron setting directly.

Evolutionary Learning Implementation This paper showed only a limited investigation into the effect of the genetic parameters on the learning process. As the recombination of BTs is quite complex, more detailed analysis is required to determine what the true effect of each parameter on the learning process. It is also interesting to investigate adaptive evolutionary parameters, where the optimisation starts with crossover intensive operation which gradually reduces as the general structure converges and gradually switches to mutation intensive operations.

The current approach to the BT genetic combination in this paper has seen that little domain specific knowledge of the task or the BTs themselves is used. These operators were applied to random nodes in the BT regardless of the node type, these operation will therefore inevitably result in some nodes in the BT being redundant or not effective to the final effective behaviour exhibited by the robotic platform.

The intention was that by not adding task specific knowledge to the genetic operators the converged solution is truly emergent and not steered by the human designers. This does however result in bloat making trees more complex. This can be remedied by pruning the BTs. Pruning was applied to the evolved trees to make them more legible to the user. This pruning process can be automated to apply simple rules to remove redundant nodes from the BTs. The influence of pruning the trees during the evolution process can also be investigated.

The variables changed in the optimisation were limited only to initial position and orientation, additional parameters may be useful to promote more general behaviour from the EL. Rooms with different shapes and environmental textures as well as changes to environmental and vehicle dynamics which are more representative of reality should aid the development of effective behaviour.

Simulation runs of each individual in a single generation are not dependent on any other individual in that population which makes the simulation of individuals, lending itself inherently to parallelism. Spreading the computational load of the simulations over multiple platforms would reduce the total time to optimise the BTs significantly.

It was observed in the EL that is important to promote generalised behaviour and to do that a wide variety of environmental parameters must be varied. In this work the initial position and pointing angle were varied and randomly changed as the population evolved to succeed. Rather than a strictly random mechanism to select changes in the environment for testing a more directed method may be useful. One method is co-evolution of the behaviour and the environment in a predator prey type of construct. The agent behaviour is evolving to increase the performance within the environment whilst the environment is evolving to make it more difficult for the agent to succeed.

DelFly During flight testing it was observed that the DelFly expressed asymmetrical control effectiveness when actuating the ailerons. This resulted in the DelFly turning faster to one side than the other with the same actuator deflection. This problem could be reduced by implementing a rate control loop onboard the DelFly and have the BT mission manager send rate set point commands to the DelFly instead of directly setting the actuator deflection angle. This rate control loop would also have the added advantage of increasing the lateral stability of the DelFly. Additionally, a height loop should also be added to the DelFly to maintain the flight altitude during testing. Future management systems could then set a climb rate set point for the DelFly to track.

The implementation of the window detection algorithm produces an estimate of the size of the window in the image. This parameter may be useful information for the DelFly to base its behaviour. It could be used to refine an estimate for the distance from the window or determine if the window is large enough for the DelFly to fit though.

The final success rate of 54% is not very high and can be the result of many factors which should be investigated. Some of these include the sensors and algorithms used as well as the implemented control strategy. Additionally, future flight tests should be conducted in a more deterministic setting where environmental variables, such as drafts, are controlled or at least measured for comparison.

SmartUAV Initially developed to investigate the real time interaction of software modules and human-in-the-loop interfaces, SmartUAV was not intended to be used for fast time simulations. Some changes would be required to make the software more suitable for this purpose in the future. It is proposed that all modules should be timed to a virtual clock that is independently updated by a user-defined frequency. This would ensure that all modules scale in speed at the same ratio.

Another issue with the multi threaded framework of SmartUAV is that there is no guarantee that each module is loaded and running together. It is possible that one module is loaded and starts running before the others are. This delay is typically quite small but in fast time the delay could be problematic. There should be a guarantee that modules are only allowed to progress once all required modules are loaded.

Appendix A

Extended Literature Review

Much of the recent research GNC has been in an effort to improve the automation of flight platforms. These tools must be combined and controlled by a flight management system which makes automated decisions as to what actions are to be taken by the vehicle. Advanced tools are needed to facilitate the development of mission management and guidance systems in the future. This chapter aims to present a detailed overview of the current state of the art currently implemented and in development across many fields with the general theme of AI. The results of this literature review will be used to define the final research objective of this thesis research.

We start by giving an overview of the different forms of AI in Section A-1. An overview of planning techniques used by AI is given in Section A-2. Section A-3 goes on to discuss the most popular implementations of AI in machine control. Finally an overview of popular flight management systems is detailed in Section A-4.

A-1 Artificial Intelligence

This section presents some background information of AI systems mainly adapted from Russell & Norvig (2009). The term AI has been used to refer to many types of automated systems but in academia it can be separated into four main definitions, as defined by Russell & Norvig (2009), AI is a system that:

- thinks like humans
- acts like humans
- thinks rationally
- acts rationally

In this paper, we will refer to an AI agent with the last definition, i.e an agent that acts rationally. An agent is simply something that acts, it perceives its environment with the use of sensors and acts on its environment through actuators. Percept is defined as the agents perceptual inputs at any given time. A perceptual sequence is the complete history of everything the agent has ever perceived. A generalised representation of an agent can be seen in Figure A-1.

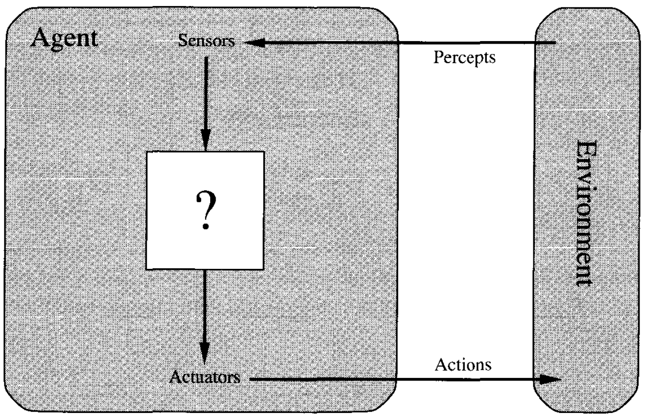


Figure A-1: Generalised representation of an agent (Russell & Norvig, 2009)

A rational agent is one that acts so as to achieve the best outcome. Rationality in time is based on the performance measure that defines the criteria for success: the agent's knowledge of its environment; the actions it can perform and; the agent's percept sequence.

A-1-1 Types of AI

There are different forms of agents which have differing levels of rationality, these are listed and described below.

Simple reflex agents These agents act only on the basis of the current percept, ignoring the rest of the percept history. The agent function is based on the condition-action rule: *if* condition *then* action. These type of agents are typically used in fully observable environments to avoid the possible precept ambiguity caused by input aliasing. Recent research has shown however, that sensor pattern aliasing is not as disadvantageous as previously considered. Nolfi (2002) shows that agents can use their own interaction with the environment to extract additional information to remove the ambiguity of the sensor input pattern referred to as *sensory-motor coordination.* Performing actions which make it easier to distinguish sensory patterns is referred to as *active perception.*

Model-based reflex agents One way to handle the problems simple reflex agents operating in partially observable environments is for the agent to have some concept of its internal state. This state is dependent on the percept history that can be used to infer part of the unobserved aspects of the current state. Updating the internal state of the agent typically requires information of how the environment is evolving and the effect of the agent on the world, this implies that some form of model of the world is required.

Goal-based agents This differs from the simple reflex agent in that it considers the future in two ways: what will be the results of the agent's actions and what actions will help achieve the agent's goal. This gives the agent a form of reasoning and therefore does not require direct state-action mapping that limits the reflex agents. These agents choose the most appropriate action which will allow it to achieve a defined goal by use of a model describing its environment. Search and planning are the fields of artificial intelligence devoted to finding action sequences that achieve the agent's goals. A general goal-based agent can be seen in Figure A-2.

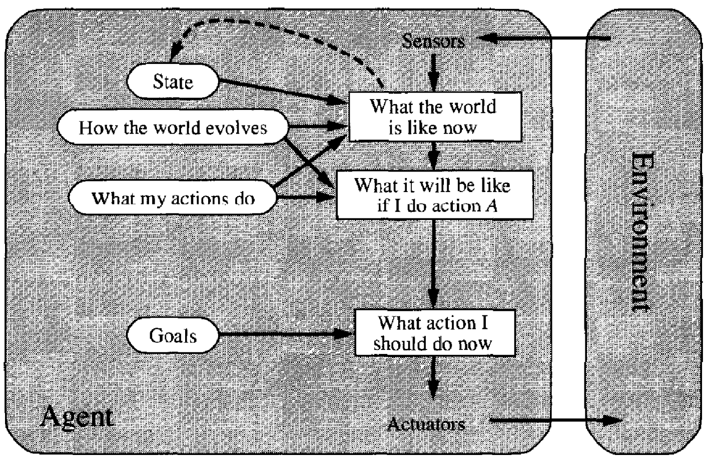

Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

Figure A-2: General model of a goal-based agent (Russell & Norvig, 2009)

Goal-based agent are less efficient than a model-based reflex agent as the most rational action must be determined by considering more variables but it is typically more flexible because the knowledge that supports its decisions is represented explicitly and can be modified.

Utility-based agents To generate high quality behaviour there must be some measure of how desirable a certain state is with reference to achieving the final goal of the agent. This can be represented by a utility function which maps a state to a measure of the utility of the state, where utility is the quality of being useful. A rational utility-based agent selects the action that maximizes the expected utility of the action outcomes, given the probabilities and utilities of each outcome. Figure A-3 shows a general utility-based agent.

This type of agent is very useful when the agent has to consider multiple, possibly conflicting, goals each with a given success certainty. Any true rational agent must behave as if it has a utility function, whether it is explicitly defined or not.

Learning agents Manually defining the behaviour of complex agents may be infeasible for programmers and it can be desirable that the agent learn behaviour as it interacts with its environment. A learning agent can be separated into four conceptual components which can also be seen in Figure A-4:

- **Critic** - Determines how the performance of the agent should be changed to improve its utility and provides this information as feedback to the learning agent

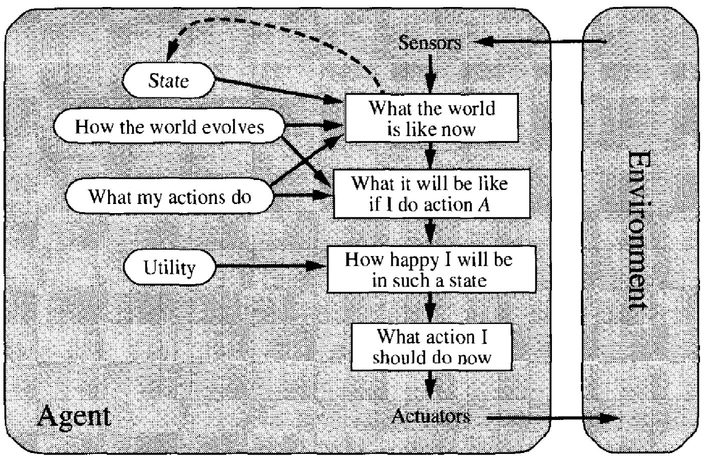

Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action A
How happy I will be in such a state
What action I should do now
Actuators
Agent
Environment

Figure A-3: General model of a utility-based agent (Russell & Norvig, 2009)

- **Learning agent** - Responsible for making improvements
- **Performance element** - Responsible for selecting external actions.
- **Problem generator** - Suggests actions that lead to a new and informative experiences.

The learning element is dependant on the performance element used and is therefore typically designed afterwards. The critic gives feedback to the learning element how well it is performing with regard to some utility function. This is important as the precepts do not typically provide enough information to the performance of the agent. Problem generators suggest exploratory actions that mat perhaps result in optimal actions in the short term but may result in better actions long term.

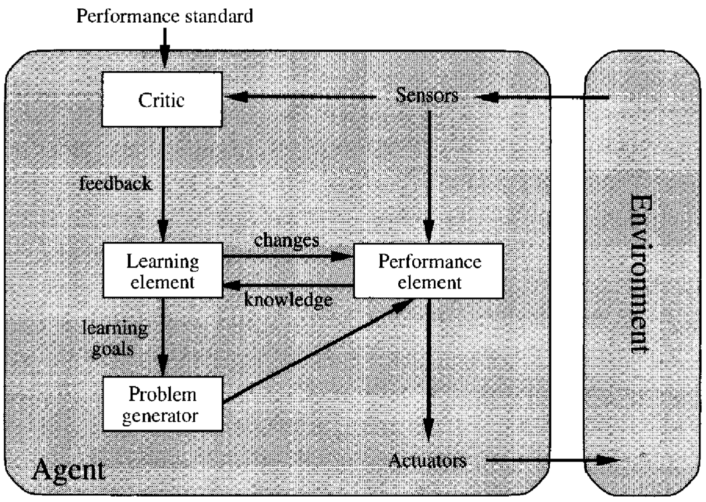


Figure A-4: General model of a learning agent (Russell & Norvig, 2009)

As learning agents are interesting to the study of advanced AI, the next section will investigate these type of AI agents further.

A-1-2 AI Learning

Manually encoding a complex multifunctional AI agent can be a difficult and very time consuming process and sometimes may just not be possible due to the limited knowledge of the system in which the AI will be operating in. As the tasks performed by robotic agents increase in complexity it is becoming ever more useful that AI can learn to improve its performance automatically. The ability for an AI agent to automatically adapt its behaviour is also very useful when operating in a changing environment. This section aims to summarise some background information behind machine learning techniques and discuss some of the methods in which they are implemented.

Firstly, we would like to define the difference between adaptation and learning as made by Arkin (1998). Adaptation is generally a process by with the agent gradually changes its behaviour to accommodate its environment improving its performance without substantial changes to the main behaviour structure. They go on to define three main categories of adaptation:

- **Behavioural Adaptation** - An agent's individual behaviours are adjusted within an individual
- **Evolutionary Adaptation** - Descendants change over long time scales based on the success or failure of their ancestors in their environment
- **Sensor Adaptation** - An agent's perceptual system becomes more attuned to its environment

Arkin (1998) goes on to define learning as more of a fundamental change to the knowledge base or representational structure of the behaviour of the AI. For example this may involve introducing new knowledge into the system, characterising a system or generalizing concepts from multiple exams. Learning methods can be differentiated on a scale of level and source of information contained in the feedback of the critic. This feedback is a form of *Supervision* and learning can be placed on a gliding scale from Supervised Learning to Unsupervised Learning.

In unsupervised learning, the information available to the agent for learning is typically restricted to the input data only and no indication of what the optimal goal state is. The performance of the agent is determined by a critic and this performance information is then feed back to the agent. This type of learning is sometimes used to find trends or patterns in data. In supervised learning, not only is the input and the output data of the agent available to it but the optimal or required output is also defined. All learning systems fall somewhere on the spectrum between these two extremes.

Arkin (1998) identified the following mechanisms which can be used to learn behaviour in AI:

- **Reinforcement Learning** - Rewards are given to the AI to adjust numeric values in a controller through trial and error based on a value function which evaluates the agents utility.
- **Evolutionary Learning** - AI controllers are derived deductively by alterations to an initial population of program code using genetic operators based on the utility of the population computed by use of a fitness function.

- **Learning by Imitation** - A form of learning similar to the way animals learn by being shown how to achieve a task.

We will investigate a little further into these methods to gain some insight into how learning can be implemented.

Reinforcement Learning

Reinforcement Learning (RL) is built on the fundamental idea of Thorndike's "Law of Effect" where the application of a reward or punishment after an action will increase the probability of its recurrence reinforcing causative behaviour. The reward is applied by a component called a critic which evaluates the agent based in its performance measured by some metric. The agent tries to modify its behaviour in order to maximise its reward. An overview of the RL framework can be seen in Figure A-5.

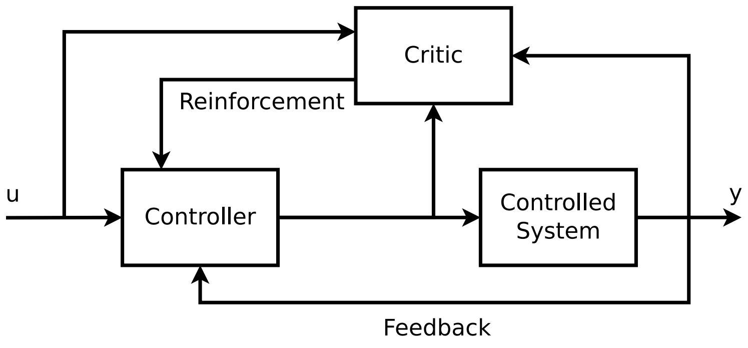


Figure A-5: Graphical representation of Reinforcement Learning framework

RL is a very useful tool used extensively to adapt control system responses but it comes with some pitfalls. One constant difficulty is how the critic is defined when evaluating multiple objectives. As the critic gives one feedback value based on the overall performance of the agent, it is difficult for the agent to optimise for each objective individually but rather optimises itself based on its overall performance so the definition of the critic is critical to the performance of the agent. There are many implementations of RL, however there are three fundamental methods used to solve RL problems: Dynamic Programming; Mone-Carlo Methods; and Temporal-Difference Learning (Sutton & Barto, 1998).

Evolutionary Learning

EL techniques are a form of heuristic search where the search space is defined as a population of feasible solutions which are slowly evolved by use of genetic operators. Heuristic methods are likely to converge towards a good feasible solution but is not necessarily the optimal solution (Hiller & Lieberman, 2010). These optimization techniques are typically applied to problems which are are sufficiently complex that it is infeasible to be tackled by non-probabilistic algorithms so a near optimal solution is generally sufficient. An overview of metaheuristic methods can be seen in Figure A-6.

[1]`http://en.wikipedia.org/wiki/File:Metaheuristics_classification.svg`, obtained on: 27-06-13

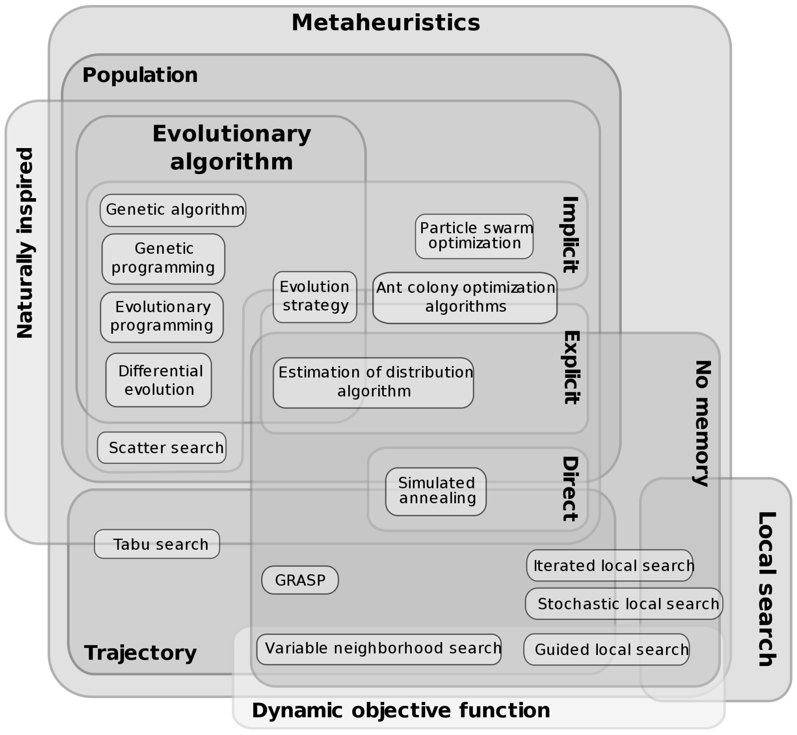


Figure A-6: Summary of metaheuristic search methods [1]

A-2 Task Planning

Planning can be loosely defined as the process of determining the sequence of actions required to achieve a goal. There are many different ways to implement this planning and differing ways of representing the planning domain. In this subsection we will first give a brief introduction to classical planning and an overview of the most popular planning techniques, this elaboration is primarily adapted from Ghallab et al. (2004) and Nau (2007).

Classical Planning Framework

Planning problems are traditionally described using classical planning system, which uses a state-transition or discrete-event system models. These systems are defined as a 4-tuple $\sum(S, A, E, \gamma)$ where $S = \{s_0, s_1, s_2, \ldots\}$ is a set of discrete states, $A = \{a_1, a_2, \ldots\}$ is a set of actions, $E = \{e_1, e_2, \ldots\}$ is a set of events and $\gamma : S \times (A \cup E) \to 2^S$ is a state-transition function. In the classical planning domain, these systems are subject to the following set of assumptions:

Finite $\sum$ - The system has a finite set of states

Fully observable $\sum$ - The agent has complete knowledge about the state of $\sum$

Deterministic $\sum$ - every state s and event action u , $|\gamma(s, u) \leq 1$

Static $\sum$ set of events E is empty so the system has no internal dynamics

Attainment Goals - The only type of goal is an attainment goal, which is an explicit goal state or set of states S_g. This assumption excludes, for example, states to be avoided, utility functions or constraints

Sequential Plans - A solution to the planning problem is a linearly ordered finite sequence of actions

Implicit Time - Actions and events have no duration

Offline planning - The system does not consider changes to the system $\sum$ while the system is planning ignoring any current dynamics.

Classical planners consider static systems only so the system can be reduced to a 3-tuple system $\sum(S, A, \gamma)$ where $\gamma : S \times A \rightarrow S^2$.

The assumptions in this planning domain are quite restrictive and most systems require a relaxation of some of these assumptions to fully represent and solve a planning problem. Some of the techniques used to solve planning problems are discussed here.

Planning Domains

Automated planning systems can be classified into categories based on the domain in which they operate, namely: domain-specific planners, domain-independent planners, and domain-configurable planners. Domain-specific planners are specially designed for use in a given planning domain and are unlikely to work in other domains unless significant changes are made to the planning system. In domain-independent planning systems, the only input is a description of a planning problem to solve, and the planning engine is general enough to work in any planning domain that satisfies some set of simplifying assumptions. Domain-configurable planners are planning systems in which the planning engine is domain-independent but the input to the planner includes domain-specific knowledge to constrain the planner's search so that the planner searches only a small part of the search space.

Whilst all of these planners have large theoretical benefits, the assumptions made in the domain-independent planner are typically too restrictive to be useful in practice. The domain-configurable planners are used more often in practice than domain-specific planners as they are easier to configure and maintain as standard tools can be used to build the planner.

Planning Methods

Plans can be synthesised by using one of three main types of search algorithms: State-Space Planner; Plan-Space Planner or; Planning Graph. These methods will be summarised below.

State-Space Planner

The simplest planning technique is State-Space Search. This planning problem consists of nodes which represent a given state of the environment, arcs connecting these nodes are considered possible state transitions and a plan is a path through the state-space. The search space is therefore a set of the state space. This representation is used in state machines and many other systems where there are discrete transitions between states in the environment. As the number of states and transitions increase, this representation can become quite complex and must be combined with advanced planning techniques to be useful.

There are many forms of search algorithms that are used to find a planning solution, one of the simplest is the Forward Search. As the name suggests, it is a search algorithm that searches the state space for a plan that starts at the current state and ends at the goal state. This method is non-deterministic with input $P(O, s_0, g)$ of planning problem $\mathcal{P}$, where O is a set of planning operators consistent of the action set A and g are preconditions of the goal state. Forward search algorithms are sound, i.e any plan π returned is a guaranteed solution to the planning problem.

The opposite approach is the Backward Search, this algorithm starts at the goal state and applies inverses of the planning operators to produce goals, stopping if a set of goals are produced which satisfied by the initial state. The inputs are the same as for the Forward Search and is also sound and complete.

Another algorithm worth noting is the Stanford Research Institute Problem Solver (STRIPS), this is an algorithm which uses a standard Planning Domain Definition Language (PDDL) to describe the planning domain. It is similar to Backward search but differs in that it only evaluates goals eligible based on the preconditions of the last operator added to the plan. This reduces the search space hence improving the efficiency of the search. Additionally, if the current state in the STRIPS plan meets the preconditions of all the operators, it commits to this plan and does not backtrack. This commitment makes STRIPS an incomplete planner, i.e. no guarantee to find solutions for solvable systems, solutions often not optimal.

Plan-Space Planner

This method is typically a more general planning technique than the state-space planner. The planning space consists of nodes which are partially specified plans and arcs are refinement operations used to further complete a partial plan. The planner begins with an empty plan node and builds on this till a path from the current state to the goal state is found and returns a partially ordered plan which solves the planning problem. Main idea of plan space planning is the least commitment principle, i.e refine one flaw at a time assigning only ordering and binging constraints to solve that flaw.

This technique differers mainly from the state-space search in that instead of simply making a sequence of actions, plan-space planners separate the planning domain into two distinct choices: the action to be applied, and the ordering of the actions to achieve the goal. As only one flaw is fixed a time, the interaction between actions and goals can be addressed by taking into account the causal links in the system.

The plan-space planner is a generalisation of the state-space planner where the number of states in the plan-space are infinite and intermediate states in the state-space are explicit. Plan-space planners are very powerful but are typically very computationally intensive due to the high level of complexity.

Planning Graph

The middle ground between state-space and plan-space planning is the Planning Graph, this technique returns a sequence of partially ordered sets of actions. This is therefore more general than a sequence of actions but less general than a partially ordered plan.The planning graph approach relies mainly on reachability analysis and disjunctive refinement.

Planning graphs take strong commitments while planning actions are considered fully instantiated at specific steps rely on reachability analysis and disjunctive refinement. Reachability analysis is a state reachable from some given state s_0 ($\hat{\Gamma}(s_0)$). Disjunctive refinement addresses one or several flaws through disjunction of resolvers.

In short, the planning graph operates recursively where a planning graph is created incrementally increasing the number of levels until a solution is found. A backwards state-space search is performed from the goal to try to find a solution plan, but the search is restricted to include only the actions in the planning graph. The Backward Search has dramatically increased computational efficiency as it is restricted to operate within the Planning Graph, the resulting solution can be computed in polynomial time. As a result, Planning Graph techniques, like GraphPlan, significantly faster than plan-space planning algorithms.

A-3 AI Implementation Techniques

Every agent must analyse its precepts and through some process determine the rational action that needs to be taken. The agent's input is the knowledge of the environment developed though its precept history and the output is an action request. This process can be formulated in many different ways, this section will describe a few of the more popular methods along with their respective benefits and drawbacks.

A-3-1 Decision Trees

The mapping between a given knowledge input and the corresponding output of an agent can be quite complex, one way to simplify this is to break the mapping into a matrix of simple choices which can be combined in such a way as to describe a complex system. An example of a Decision Tree can be seen in Figure A-7.

Decision Trees originate at a root node and propagate down the tree along links evaluating decisions until no choices remain. Nodes that have no downward links are called leaf nodes and are attached to actions. Most decision nodes make a binary decision between two nodes which makes evaluating the choices computationally light. It is possible to combine decisions in such a way to mimic more complex behaviour such as apparent boolean logic.

The simplification of the overall decision making into a series of simple choices makes decision trees a fast, easily implemented and simple to understand modelling technique. The use of standard nodes makes Decision Trees very modular and easy to create. This simple decomposition of choices in a tree however, also results in the fact that the tree will contain many more decisions than are actually needed to determine the appropriate action, this can be seen when you propagate down the tree only one branch will be used while the tree has

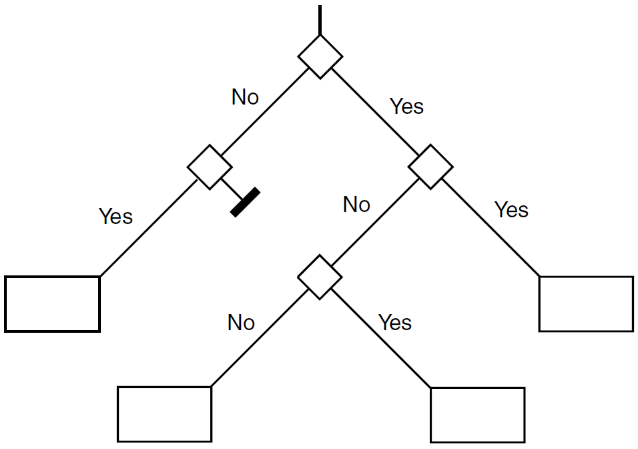


Figure A-7: Typical decision tree diagram (Millington & Funge, 2009)

many more branches not currently used requiring in a repetition of behaviour within the tree structure.

A *balanced tree* is one that has an equal number of decision nodes between the root node and the leaf nodes. A balanced tree will propagate with $O(log_2 n)$ decisions, where n is the number of decision nodes in tree. It can be seen that a very unbalanced tree will result in a degrading of this to $O(n)$.

A fuzzy implementation of decision trees has been used to model the behaviour of a UAV as showed by Smith & Nguyen (2007). This shows that this method is indeed practical for real flight platform implementation.

A-3-2 Finite State Machine

This decision making technique is built on the idea that the agent will occupy one particular state at any given time where each state is attached to an accompanying action, the agent will perform this action until an event or condition occurs to change the state of the agent. The connections between states are called *transitions*, each transition connects one state to another. When a transition is activated by an event it said to be *triggered*, once a transition has occurred it has been *fired*. Separating the transition and the firing of a transition allows transitions to have their own actions making them useful tools. An FSM is therefore completely defined by a list of its states, transitions and the triggering condition for each transition.

Unlike Decision Trees, State Machines not only take into account the environment but also their internal state. State Machines are depicted using a UML state chart diagram format (Harel, 1987). A typical FSM state chart can be seen in Figure A-8.

FSM are studied extensively in Automata Theory, in theoretical computer science this is the study of mathematical objects called abstract machines or automata and the computational problems that can be solved using them (Hopcroft et al., 2000). This means that there has been extensive research done into optimising the operation of FSM and there are many tools widely available to aid in their design.

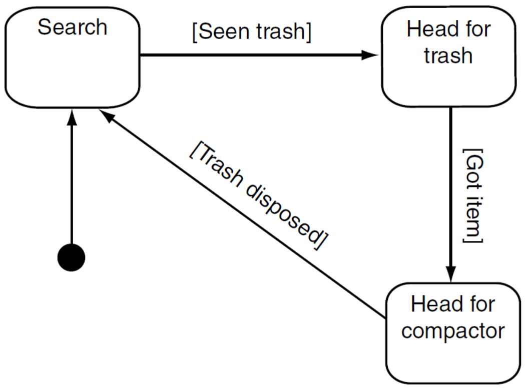


Figure A-8: Example of typical finite state machine (Millington & Funge, 2009)

There are many ways to implement an FSM depending on its final use, these implementations are typically quite easy to design and implement. All of these implementations however suffer from the same inherent problem, *state explosion*. As the number of states increase the apparent complexity of the system increases disproportionately. To illustrate this, we will use the example as described in Millington & Funge (2009) as shown in Figure A-9.

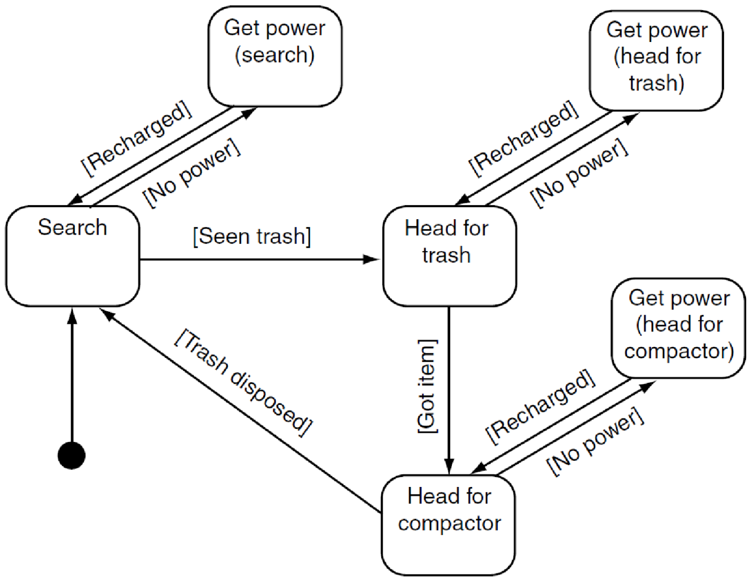


Figure A-9: Example of state explosion in FSM (Millington & Funge, 2009)

This state explosion means that maintaining large FSMs is very difficult due to the high level of complexity. Additionally this method is not very flexible or modular as when a state is added of removed all the transitions to and from the nodes typically must be recompiled which can be cumbersome for complex agents. Methods can be used to dynamically link state machine nodes at run-time but development of these methods can also be quite time consuming. As a result FSM are typically only practically useful for agents which have a limited set of actions required.

A-3-3 Hierarchical Finite State Machine

The problem of state explosion can be addressed by introducing a hierarchy to the definition of the FSM. States can nested in other states in a hierarchical order such that the number of state transitions in the FSM can be reduced significantly. Again we illustrate this from an example from Millington & Funge (2009) where Figure A-10 shows that the apparent complexity of the robot problem shown in Figure A-9 an be significantly reduced.

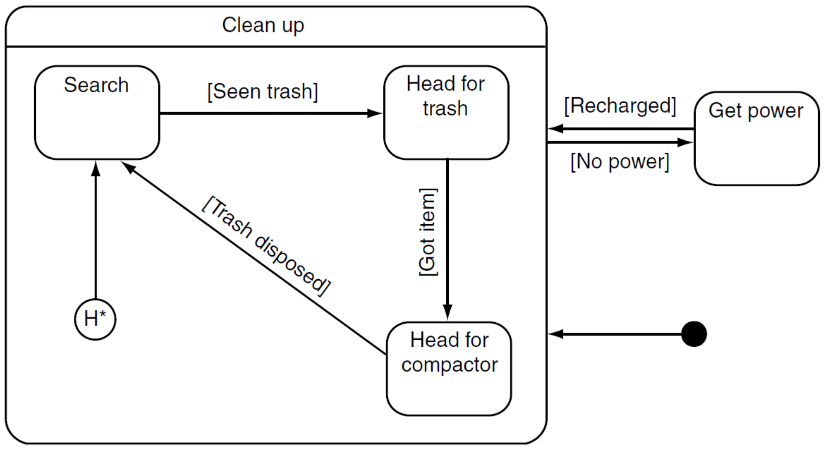


Figure A-10: Example of how HFSM can be used to address state explosion in FSM (Millington & Funge, 2009)

HFSMs are quite efficient and operate with performance $O(nt)$ where t is the number of transitions per state. Now, although the HFSM is useful in reducing the complexity of the agent as compared to the FSM it is still not very modular or flexible as the transitions between states must still be hard-coded making advanced agents difficult to manage.

A-3-4 Hierarchical Task Network

This is an implementation method of automated planner used to generate a sequenced network of actions to achieve some goal. This is interesting here as it uses hierarchy in a different way than the HFSM. HTNs uses the concept of hierarchical decomposition to recursively decompose a goal till a set of primitive operators are found which achieve the goal. Primitive operators represent actions that are executable and can appear in the final plan. Non-primitive operators represent goals that require further decomposition to be executed. Also, HTN differ from HFSM in that the planner's objective is described not directly as a set of goal states but instead as a collection of tasks to be performed (Brooks, 1987). Figure A-11 shows an example of the decomposition involved in an HTN.

To guide the decomposition process, the planner uses a collection of methods that give ways of decomposing tasks into sub-tasks. This is typically done by using domain-specific knowledge to create a set of rules for pruning the search space, thereby increasing the operational efficiency of the planner (Nau, 2007).

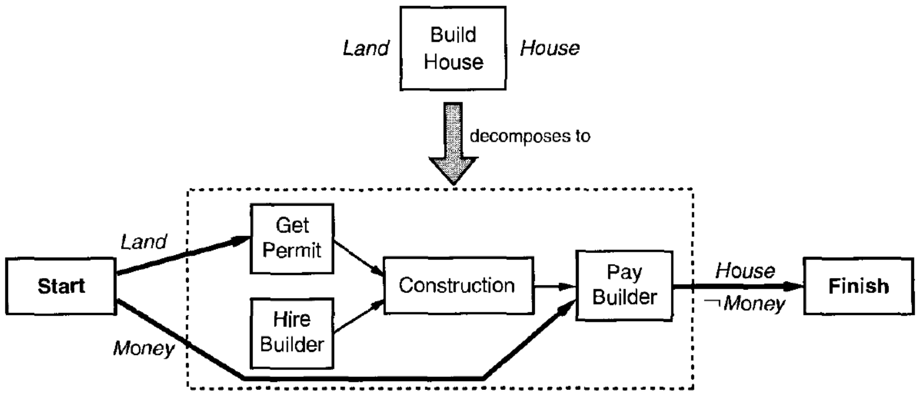


Figure A-11: A possible HTN task decomposition for building a house (Nau, 2007)

A-3-5 GraphHTN

This is a hybrid planning algorithm that performs HTN planning using a combination of HTN-style problem reduction and planning graph generation using GraphPlan (Lotem & Nau, 2000). The planning tree is an AND/OR tree that expresses all the possible ways to decompose the initial task network using the HTN methods.

Like the GraphPlan algorithm, it incrementally increases the length of the plan being searched. In each iteration of the algorithm, the planning tree is expanded by one layer by performing only task decompositions that might generate actions for the current time step. Only the actions that have been generated so far in that process are used for extending the planning graph. When certain conditions hold, the algorithm starts to search for a solution within the planning tree and the planning graph. This process is iterated until a solution is found. GraphHTN produces plans in the GraphPlan format, where several actions may occur at the same time if they do not interfere with each other. Like GraphPlan, the GraphHTN algorithm is sound and complete, and is guaranteed to report the plan with the shortest parallel length if a plan exists.

A-3-6 Goal Oriented Action Planner

The previous methods all address how to achieve a singular goal but if multiple goals are present, it becomes more difficult for the agent decide what action will be the most appropriate as some actions, although appropriate for one goal, may be detrimental to achieving other goals. One solution to this problem is the Goal Oriented Action Planner (GOAP) which is a type of utility-based agent where the utility of each action is determined using some heuristic and the action giving the best overall utility is typically executed.

A model of the world is required to determine the consequence of an action to determine the utility of an action or action set. Propagating this model for every available action or action series can be computationally intensive for complex agents. Performance is $O(nm^k)$ in time, where n is the number of goals, m is the number of possible actions and k is the number of time steps considered. To increase the efficiency of the GOAP algorithm it is usually augmented with a search algorithm like $A*$ and some selection heuristics to reduce the number of applicable actions prior to determining the utility of the actions.

A-4 UAV Flight Control System Design

To fully understand the requirements of an automated mission manager, we must also investigate how a mission guidance agent fits into the overall flight control system of a UAV. This section will look into three UAV flight control systems: Paparazzi; ArduPilot and; SmartUAV. We will discuss how the decision making system is implemented in these platforms.

A-4-1 Paparazzi

Paparazzi is a free and open-source hardware and software project aimed at developing autopilot systems for fixed-wing and multicopter aircraft (Brisset et al., 2006). All hardware and software is open-source and freely available to anyone under the GNU licensing agreement. Since its inception in 2003, development has been lead by the École Nationale de l'Aviation Civile (ENAC) and the DUT. It has also attracted some commercial contributors such as Transitioning Robotics (Paparazzi Community, 2013b).

Paparazzi provides a framework in which fully autonomous flight platforms can be developed and contains functionality for: onboard flight control software; ground station visualisation and control; mission planning and control. The framework used in Paparazzi can be seen in Figure A-12.

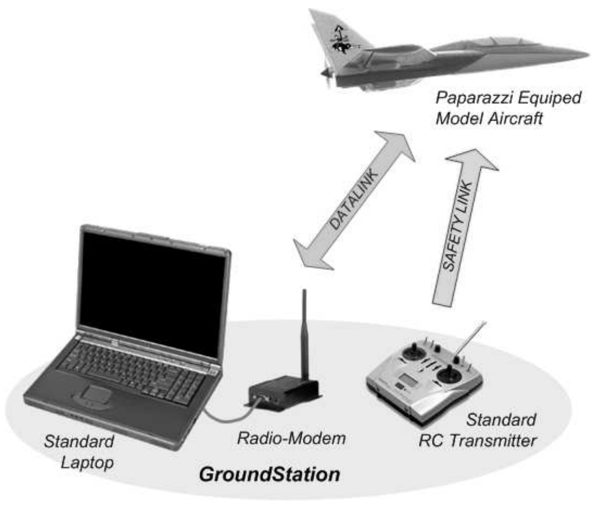


Figure A-12: Overview of the paparazzi FCS (Paparazzi Community, 2013b)

Mission Planning and Control Flight plans are implemented in an *XML* file format using a custom Document Type Definition (DTD). This flight plan is compiled at runtime and implemented onboard the UAV as a FSM. This flight plan can be changed online during flight using the Ground Control Station (GCS).

Future work is currently being carried out on advanced mission planning systems, where the planning problem is formulated as a Constraints Satisfaction Problem (CSP) which will be solved with the FaCiLe constraints library (Paparazzi Community, 2013a). This framework requires plans to be recomputed online to take into account the uncertainties during the performance of the flight. To ensure that a mission is completed in finite time, a classical reactive architecture has been implemented in Paparazzi to operate until the planner is developed.

Ground Control Station

Paparazzi also includes a GCS to allow the user to visualise the current state and settings of the aircraft in real-time and provides the user with a platform to communicate with the UAV. The GCS typically communicates with the UAV using a bidirectional wireless modem which supports both telemetry (downlink) and telecontrol (uplink).

Among others, the GCS software platform provides the following features:

- Compilation tools to produce the airborne software from the configurations and source code
- GUI to control and interact with the UAV(s) during flight as well as mission planning and simulation
- Basic simulator to ease the development of flight plans and verify some airborne code behaviour
- Data logging system with plot-based viewer for real-time and post-flight analysis
- Utilities for communicating with the UAV and other agents
- Control panel GUI for configuration and program management

The GCS is made up of multiple modules which operate independently and simultaneously, communicating to each other over a network. Each module contains code to interact with the sensory inputs and action outputs of the vehicle or to perform some computation. The required modules can be included when needed for a specific flight platform or mission requirement. This modular framework makes it easy to add or remove modules or to integrate custom modules. This also makes it easy to transition between simulated and real flight. The modules to be run can be configured in the PaparazziCenter.

Simulation

Simulations can be run from the GCS but instead of connecting to the real UAV in flight Paparazzi simulates the aircraft data and telecommunications link. Paparazzi currently has 3 different simulators with varying degrees of realism and intended purpose:

- SIM - The basic fixed-wing simulator
- JSBSim - More advanced fixed-wing simulator using JSBSim for the Flight Dynamic Model
- NPS - An advanced rotorcraft simulator with sensor models also uses JSBSim

JSBSim is an open source Flight Dynamics Model (FDM) which is essentially the mathematical model that defines the movement of a vehicle operating in a defined environment (Berndt, 2011). As JSBSim has no native graphical interface, it is typically used as a dynamics module in a larger simulation environment, some notable examples are the open source simulator *FlightGear*, motion-base research simulators at the University of Naples, Italy, and in the Institute of Flight System Dynamics and Institute of Aeronautics and Astronautics at RWTH Aachen University in Germany.

JSBSim contains features to include fully configurable flight control system, aerodynamics, propulsion, landing gear arrangement, etc. through *XML*-based text file format. It can also consider rotational earth effects on the equations of motion. It can also be augmented with the GAIA module which adds some environmental parameters such as wind, infrared contrast, Global Positioning System (GPS) quality, and time scale reference.

A-4-2 ArduPilot

ArduPilot, previously known as ArduPilot Mega (APM), is a open source, community maintained, autopilot system developed to be used to control and interact with many forms of aerial and land based vehicles (ArduPilot Community, 2013b). ArduPilot contains a code base which allows developers to interface with the vehicle hardware and automate the control of the platform to achieve some task. Similar to Paparazzi, ArduPilot provides users with a mission planning and control centre as well as a ground station to communicate with the UAV in flight. Just like Paparazzi, ArduPilot does not have an embedded simulation platform but rather it relies on an external program Flightgear to perform hardware-in-the-loop simulations.

ArduPilot provides two methods to implement mission planning, namely: *MissionPlanner* and; *APM Planner 2.0* (ArduPilot Community, 2013c,a). We will discuss these two systems in brief below.

MissionPlanner

The MissionPlanner is a ground station and vehicle mission planing platform developed by Michael Oborne to interface with ArduPilot based flight platforms (ArduPilot Community, 2013c). This software gives the user the ability to provide the flight platform with the following features:

- Point-and-click waypoint entry using Google Maps
- Select mission commands from drop-down menus
- Download mission log files and analyse them
- Configure settings for the selected flight platform
- See the output from APMs serial terminal

Mission management is described in a mission file which allows the user to program flight waypoints and commands such as: Loiter; Return-to-Launch; Land; Take-off, Change_Speed;

ect. The user can also apply some conditional logic such as Conditional_delay; Conditional_distance; Conditional_Change_Alt and; Do_Jump which allow the user to control when the flight platform executes a command from the flight plan. This line-based formulation for the mission planning is similar to many line-based programming languages such as *Basic*.

APM Planner 2.0

APM Planner 2.0 is an multi-platform open-source ground station application for UAV using the Micro Air Vehicle Communication Protocol (MAVLink) which includes: PX4, PIXHAWK, APM and Parrot AR.Drone (Meier, 2013). This recently updated platform builds on the MissionPlanner platform and incorporates some features from another ground station called *QGroundControl* (Bonney, 2013). This software allows the user to easily interface with some recently developed flight platforms. The simulation and mission planning capabilities of the APM Planer 2.0 are very similar to the MissionPlanner above.

Appendix B

Preliminary Work

This chapter presents the preliminary work done to show that EL based learning can be effectively used to automatically generate BTs that exhibit rational behaviour. First, a description of how the BT structure is implemented will be stated in Section B-1. This is followed by how the EL and its genetic operators have been implemented Section B-2. The test problem will then be described in Section B-3 and some preliminary results will be shown in Section B-4.

B-1 Behaviour Tree Implementation

The BT framework was developed in a `C++` programming language similar to the standard BT framework as described in (Champandard, 2012). A standard Behaviour class called a *Behaviour* was defined from which all other node types inherit from, this maintains the reusability and expandability that makes BTs so useful.

Basic nodes only contain information about themselves as well as some string identifiers to individualise each node. The core behaviour of each node is contained within a wrapper function which handles the initialisation, execution of the node behaviour and termination of the node. When a node is called or *ticked*, it is first initialised then its behaviour is called, upon completion of the behaviour it returns its state (Success, Failure, ect.) to its parent.

Actions and *Conditions* directly inherit from this class structure, where each action or condition contains different behaviour. As Composite nodes have children, their structure must be expanded to also contain information about their children. They contain a vector of pointers to their children and an iterator pointing to the current child. Additionally there are helper functions to set/get, add/delete or replace child nodes of a composite.

Only two composite nodes were implemented, namely the: *Sequence* and; *Selector*. These functions were implemented as described earlier. The Action and Condition nodes are dependent of the sensor input and action output of the specific platform and will be described later.

To retain the generic framework of the BT nodes, data cannot explicitly be passed to each node but rather the nodes can access a centralised set of data managed by a *Blackboard*. A Blackboard is used to store the required data and manages the reading and writing of data from any requester. A custom Blackboard was developed for this project. A Blackboard must be initiated and passed to the root node of the BT when it is ticked, explicitly passing the Blackboard retains the flexibility of changing or reinitialising the Blackboard during runtime or as it is passed down the tree.

B-2 Evolutionary Learning Implementation

The main elements which make up EL are: how the population of solutions is initialised; how individuals are chosen for procreation; how individuals from one generation are combined to make the next generation; how individuals are mutated. These points will be briefly described below.

Initialisation The initial population of M individuals is generated using the *grow* method as defined by Koza (1994). This results in variable length trees where every Composite node is initialised with its maximum number of children and the tree is limited by some maximum tree depth, this provides an initial population of very different tree shapes with good genetic material to ensure good EL search.

Selection Tournament selection is used and is implemented by selecting a number of random individuals from the population. The size of the group is defined by the tournament selection size parameter s. This subgroup is then sorted in order of their fitness, if two individuals have the same fitness they are then ranked based on tree size, where smaller is better. The best individual is then selected and returned for procreation, thus the probability of selection from subgroup is unity, $p = 1$.

Crossover As the tournament selection returns one individual, two tournaments are needed to produce two parents needed to perform crossover. The percentage of the new population formed by Crossover is defined by the Crossover Rate P_c. Crossover is implemented as the exchange of one randomly selected node from two parents to produce two children. The node is selected totally at random independent of its type or its location in the tree.

Mutation Mutation is implemented with two methods, namely: micro-mutation and; Macro-mutation. Micro-mutation only affects leaf nodes and is implemented as a reinitialisation of the node with new operating parameters. Macro-mutation, also known as Headless-Chicken Crossover, is implemented by replacing a selected node by a randomly generated tree which is limited in depth by the maximum tree depth. The probability that mutation is applied to a node in the BT is given by the mutation rate P_m, once a node has been selected for mutation the probability that macro-mutation will be applied rather than micro-mutation is given by the headless-chicken crossover rate P_{hcc}.

B-3 Khepera Test Problem

An initial test set-up was devised to test all the developed software and investigate the effect of the operating parameters of the learning system. The chosen problem was the application to the Khepera wheeled robot as shown in Figure B-1. This problem is often used in automated mission management development problems (Koza & Rice, 1992; Floreano & Mondada, 1994; Quinn et al., 2002; Nolfi, 2002; Trianni, 2008; Nelson et al., 2009). The vehicle consists of two wheels 55mm apart with eight Infrared (IR) sensors distributed around the vehicle.

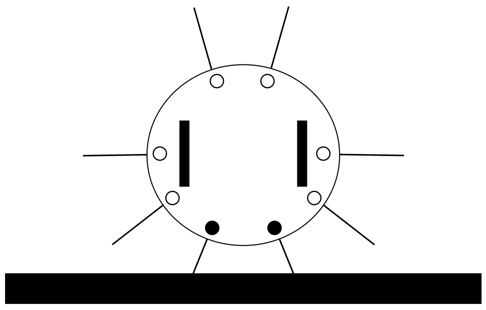


Figure B-1: Khepera vehicle

In a passive lighting situation the IR sensors output is a saturating linear function proportional to the distance from the wall, the maximum range is 60mm and the output will saturate at 25mm. Although in reality the IR sensors have a cone shaped viewing angle of about 70° the viewing angle was modelled as line to simplify the simulation. The wheel speeds can be set independently to control the motion with top vehicle speed of 80cm/s.

The Action node implemented in the BT was *SetWheelSpeed*, when this node is called it will set the wheel speeds of both wheels to some value determined at random when the node was initialised. The *LessThan* operator was implemented as the only condition to keep the eventual BT simple. Each *LessThan* node is initialise with a random sensor to check and a random value to compare against, when called it will return if the current IR sensor value is less then the threshold. A UML class diagram for the implementation of the BT can be found below in Figure B-2. The EL is evaluated with a fitness function and this has a great influence of the final vehicle behaviour. The fitness function was implemented as follows:

$$F = V \cdot (1 - \sqrt{\Delta V}) \cdot (1 - i); \quad V := [0,1], \Delta V := [0,1], i := [0,1] \tag{B-1}$$

where V is the normalised sum of the wheel speeds, ΔV is the difference of the wheel speeds and i is the normalised distance of the IR sensor closest to any wall. The three components of the fitness function will ensure that the Khepera will go quickly forward (V) is as straight a line as possible (ΔV) while avoiding walls (i).

Each generation had k simulation runs to evaluate the performance of each individual. Each run is characterised by a randomly initiated location in the room and a random initial pointing direction. Each run was terminated when the Khepera hit the wall or reached a maximum simulation time of 100s.

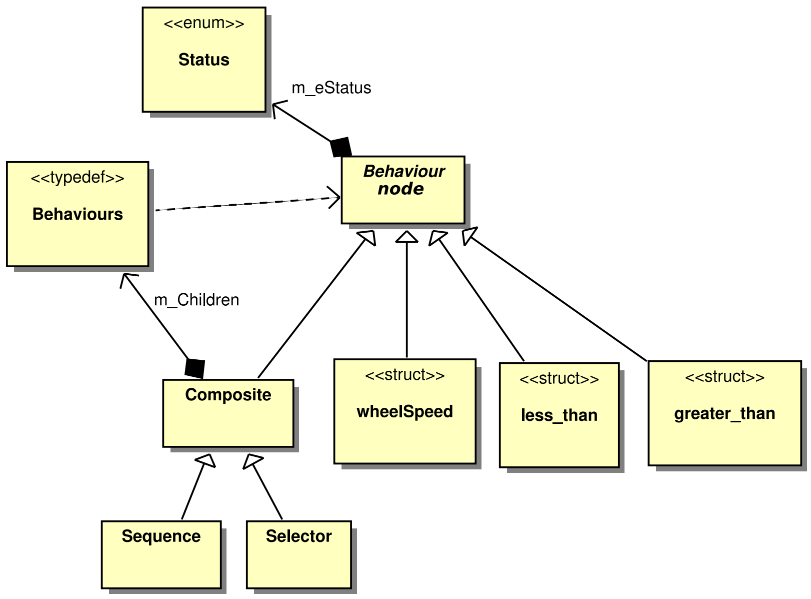


Figure B-2: UML class diagram of the Behaviour Tree framework for the wheeled robotic vehicle Khepera

B-4 Test Results

This section will briefly present some sample results from the test problem described above. The following parameters were used for the EL:

Table B-1: Table showing parameter values for the EL run

Parameter	Value
Number of Generations (G)	50
Population size (M)	1000
Tournament selection size (s)	1%
Elitism rate (p_e)	1%
Crossover rate (p_c)	89%
Mutation rate (p_m)	20%
Headless-Chicken Crossover rate (p_{hcc}	20%
Maximum tree depth (D_d)	6
Maximum children (D_c)	6
No. of simulation runs per generation (k)	3

Figure B-3 is a plot of the path navigated by the best individual in the last generation, this shows the eventual behaviour learned by the system to navigate an irregular room. In this particular run, the Khepera starts in the North-East corner and proceeds to navigate the room making sharp right turns. It can be seen that the Khepera acts as would be expected with the selected value function, it goes as straight and fast as possible while avoiding walls. It was observed that the Khepera typically converges to always turn in one direction. This is an interesting result which will ensure that the robot will never be caught in a corner by possible chattering between *turn right* and *turn left* behaviours, this would therefore increase the reliability and generality of the converged behaviour.

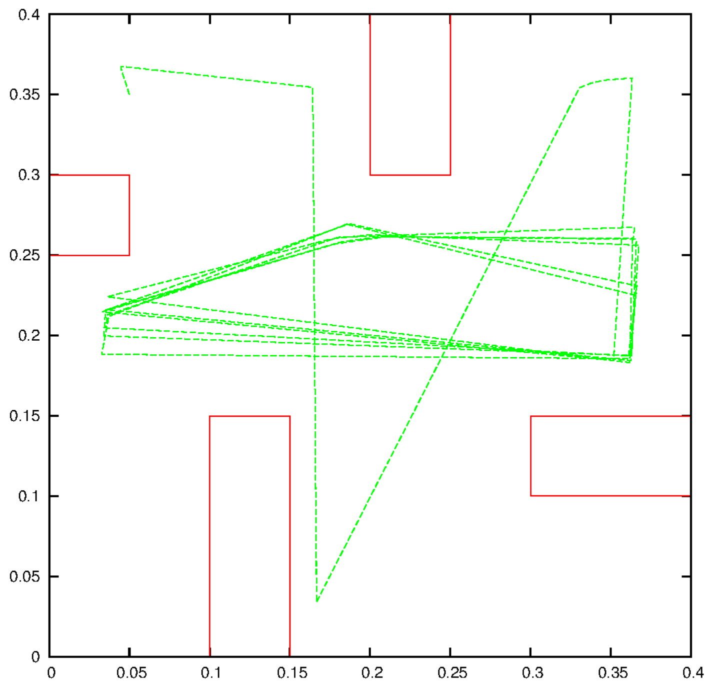


Figure B-3: Path of the best individual in the last generation of the EL in an irregular room

The progression of the EL can be seen in Figure B-4. This plot shows that the best individuals in the EL quickly converge and the fitness function has little variation for multiple simulation runs. The average score of the entire population initially improves but quickly levels off and has a large standard deviation. This indicates a diverse population which is good for exploration in the EL. The mean of the general population is also significantly lower than the best performers which indicates that the population has a significant number of poor performers which can also indicate good exploration of the solution space of the EL.

Finally, Figure B-5 shows the size of the BTs in the EL as the generations progress. It can be seen that the initial population has very large trees which ensures much genetic material to begin the EL search. The trees reduce in size as the number of generations progresses but still remain large by the end of the run, much larger than would be generally easily comprehensible to a human. As the ability for the user to comprehend the final BT is beneficial to the final usage of the behaviour in practice the reduction of the tree size should be explicitly considered in the EL. A more direct method for selection in the EL based on tree size could be possibly implemented with a multi-objective optimisation such as the Pareto approach (Sbalzarini et al., 2000; Tan et al., 2001). The EL would then be used to optimise the population for both the fitness function and the tree size.

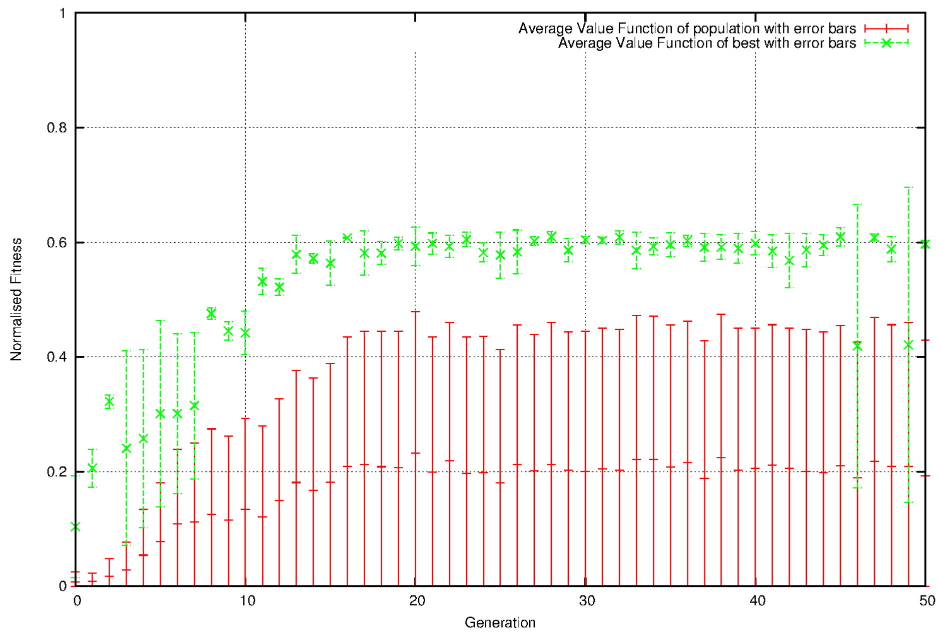


Figure B-4: Average fitness and standard deviation of the best individual and the entire population

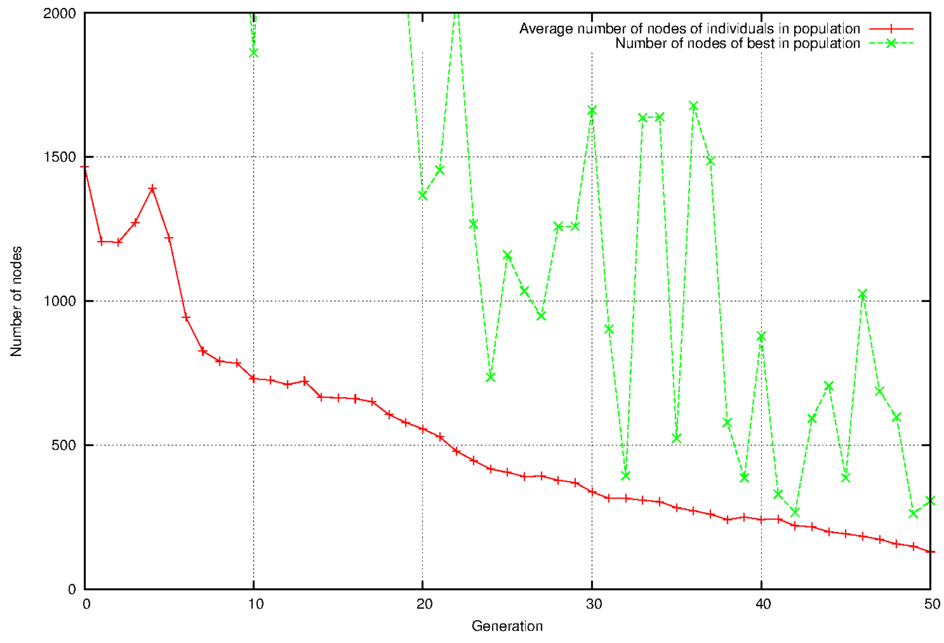


Figure B-5: Total number of nodes in the best individual and the average of the population

Appendix C

Behaviour Tree Pruning

This chapter describes the method used to prune the genetically optimised BT in more detail. As the GP optimisation used in this paper uses no information of the behaviour tree in its genetic operators other than the geometrical limits of the tree, some genetic operations result in nodes in the tree which have no purpose. Pruning is simply the process of removing nodes from the behaviour tree that have no resultant effect on the exhibited behaviour.

Pruning is dependant on the tree structure but also on the possible outputs of each node in the tree. In the current implementation, conditions have two possible status returns (SUCCESS or FAILURE) whilst actions have only one (SUCCESS). Additionally, condition nodes are implemented with only SUCCESS and FAILURE status returns. Due to this information, some simple examples of localised node combinations that result in unnecessary nodes are listed below:

- nodes after an Action node in a Selector will never be evaluated
- multiple Action nodes of the same type in series within a Sequence will overwrite each other so only the final action will be effective
- a composite with only one child can be replaced by its child node

Below, Listing C.1 shows the original result of the genetic optimisation. The BT contains 32 behaviour nodes with maximum depth 6 and maximum number of children of 7.

```
1  <BTtype>Composite<function>Selector<vars><name>Selector<endl>
2     <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
3        <BTtype>Action<function>turnRate<vars>0.2<name>DelFly<endl>
4        <BTtype>Composite<function>Selector<vars><name>Selector<endl>
5           <BTtype>Condition<function>greater_than<vars>308,2<name>DelFly<endl>
6           <BTtype>Action<function>turnRate<vars>-0.7<name>DelFly<endl>
7           <BTtype>Condition<function>less_than<vars>-0.84,3<name>DelFly<endl>
8           <BTtype>Action<function>turnRate<vars>0<name>DelFly<endl>
9           <BTtype>Condition<function>less_than<vars>-0.32,3<name>DelFly<endl>
10          <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
11             <BTtype>Composite<function>Selector<vars><name>Selector<endl>
12                <BTtype>Condition<function>less_than<vars>0.28,0<name>DelFly<endl>
13                <BTtype>Composite<function>Selector<vars><name>Selector<endl>
14                   <BTtype>Condition<function>less_than<vars>0.9,0<name>DelFly<endl>
15                   <BTtype>Action<function>turnRate<vars>-0.6<name>DelFly<endl>
16                   <BTtype>Condition<function>greater_than<vars>112,2<name>DelFly<endl>
17                <BTtype>Action<function>turnRate<vars>0.9<name>DelFly<endl>
18                <BTtype>Action<function>turnRate<vars>0.4<name>DelFly<endl>
19                <BTtype>Action<function>turnRate<vars>0.3<name>DelFly<endl>
20                <BTtype>Action<function>turnRate<vars>-0.9<name>DelFly<endl>
21          <BTtype>Condition<function>less_than<vars>94,1<name>DelFly<endl>
22       <BTtype>Action<function>turnRate<vars>-0.5<name>DelFly<endl>
23       <BTtype>Action<function>turnRate<vars>-0.5<name>DelFly<endl>
24       <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
25          <BTtype>Action<function>turnRate<vars>-0.5<name>DelFly<endl>
26       <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
27          <BTtype>Action<function>turnRate<vars>0.2<name>DelFly<endl>
28          <BTtype>Condition<function>less_than<vars>132,2<name>DelFly<endl>
29    <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
30       <BTtype>Action<function>turnRate<vars>1<name>DelFly<endl>
31       <BTtype>Condition<function>less_than<vars>69,1<name>DelFly<endl>
32       <BTtype>Action<function>turnRate<vars>-0.4<name>DelFly<endl>
```

Listing C.1: Original genetically optimised Behaviour Tree for fly-through-window task. Lines highlighted in red can be pruned away

Applying the previously mentioned rules we can reduce the tree to 14 children as shown below in Listing C.2. Applying very simple pruning rules have amounted to a reduction of about a 2/3 of the original tree size.

```
1  <BTtype>Composite<function>Selector<vars><name>Selector<endl>
2     <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
3        <BTtype>Action<function>turnRate<vars>0.2<name>DelFly<endl>
4        <BTtype>Composite<function>Selector<vars><name>Selector<endl>
5           <BTtype>Condition<function>greater_than<vars>308,2<name>DelFly<endl>
6           <BTtype>Action<function>turnRate<vars>-0.7<name>DelFly<endl>
7        <BTtype>Action<function>turnRate<vars>-0.5<name>DelFly<endl>
8        <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
9           <BTtype>Action<function>turnRate<vars>0.2<name>DelFly<endl>
10          <BTtype>Condition<function>less_than<vars>132,2<name>DelFly<endl>
11    <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
12       <BTtype>Action<function>turnRate<vars>1<name>DelFly<endl>
13       <BTtype>Condition<function>less_than<vars>69,1<name>DelFly<endl>
14       <BTtype>Action<function>turnRate<vars>-0.4<name>DelFly<endl>
```

Listing C.2: Genetically optimised Behaviour Tree for fly-through-window task after application of localised pruning. Lines highlighted in red can be pruned away

By applying more global information of the node combinations can lead to further reduction in size. An example of this global optimisation can be seen in Listing C.2. If we evaluate the BT to the Selector node on line 4, evaluating its children we see that the Condition node has two possible outcomes, let us first assume the outcome will be SUCCESS, the selector will then return SUCCESS to its parent which in this case is a Sequence which continues to evaluate the tree. In this case the Selector on line 4 has no effective impact on the output of the tree. Now, evaluating the tree again and assuming the outcome of the Condition node on line 5 is FAILURE, the Action node on line 6 will be evaluated and return SUCCESS to the Selector which returns SUCCESS to its parent a Sequence which then evaluates its next child which is an Action which overwrites the Action node on line 6. In this case again, the Selector and its children have no effective impact on the output of the BT. This selector can therefore be removed along with the Action node on line 3. A similar logic can be used to remove the Sequence on line 8 and therefore the turnRate on line 7. These actions result in our final pruned BT as shown in Listing C.3 which contains 8 nodes.

```
1 <BTtype>Composite<function>Selector<vars><name>Selector<endl>
2    <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
3       <BTtype>Action<function>turnRate<vars>0.2<name>DelFly<endl>
4       <BTtype>Condition<function>less_than<vars>132,2<name>DelFly<endl>
5    <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
6       <BTtype>Action<function>turnRate<vars>1<name>DelFly<endl>
7       <BTtype>Condition<function>less_than<vars>69,1<name>DelFly<endl>
8       <BTtype>Action<function>turnRate<vars>-0.4<name>DelFly<endl>
```

Listing C.3: Pruned genetically optimised Behaviour Tree for fly-through-window task

If we were to continue our in-depth investigation, it becomes clear that we can reduce the system to only 7 nodes if we compress the two Sequences together if we inverse the logic of node 4 as shown in Listing C.4. This is a quite invasive alteration so was not used in the thesis analysis but is possible due to the intelligence of the BT encoding framework.

```
1 <BTtype>Composite<function>Selector<vars><name>Selector<endl>
2    <BTtype>Composite<function>Sequence<vars><name>Sequence<endl>
3       <BTtype>Action<function>turnRate<vars>0.2<name>DelFly<endl>
4       <BTtype>Condition<function>greater_than<vars>131,2<name>DelFly<endl>
5       <BTtype>Action<function>turnRate<vars>1<name>DelFly<endl>
6       <BTtype>Condition<function>less_than<vars>69,1<name>DelFly<endl>
7       <BTtype>Action<function>turnRate<vars>-0.4<name>DelFly<endl>
```

Listing C.4: Pruned genetically optimised Behaviour Tree for fly-through-window task

The pruning applied to get from Listing C.1 to Listing C.2 only uses information about the node and its immediate neighbours in the tree, this can be simply automated using rules. The second pruning step we described here requires knowledge of how each node in the tree interacts with other nodes and would therefore need to be implemented using a more advanced method, perhaps with some form of recursive logical analysis.

Bibliography

Amelink, M., Mulder, M., & van Paassen, M. M. (2008, March). Designing for Human-Automation Interaction: Abstraction-Sophistication Analysis for UAV Control. In *International multiconference of engineers and computer scientists* (Vol. I). Hong Kong: IAE.

Angeline, P. J. (1997, July). Subtree Crossover: Building Block Engine or Macromutation. In *Genetic programming* (pp. 9–17). Stanford University, CA, USA.

ArduPilot Community. (2013a). *Apm planner 2.0.* Retrieved Online; accessed 13-01-2013, from `http://planner2.ardupilot.com/`

ArduPilot Community. (2013b). *ArduPilot/APM Development Site.* Retrieved Online; accessed 13-01-2013, from `http://dev.ardupilot.com/`

ArduPilot Community. (2013c). *Mission Planner.* Retrieved Online; accessed 13-01-2013, from `http://planner.ardupilot.com/`

Arkin, R. C. (1998). *Behaviour-Based Robotics.* MIT Press.

Berndt, J. S. (2011). *JSBSim: An Open Source, Platform-Independent, Flight Dynamics Model in C++.*

Bongard, J. C. (2013, August). Evolutionary Robotics. *Communications of the ACM*, *56*(8), 74–83.

Bongard, J. C., Zykov, V., & Lipson, H. (2006, November). Resilient Machines through Continuous Self-Modeling. *Science*, *314*(5802), 1118–21.

Bonney, B. (2013). *Announcing APM Planner 2.0 RC1.* Retrieved Online; accessed 13-01-2013, from `http://ardupilot.com/forum/viewtopic.php?f=82&t=5281&p=7889#p7887`

Brisset, P., Drouin, A., & Gorraz, M. (2006, September). The Paparazzi Solution. In *2nd us-european competition and workshop on micro air vehicles* (pp. 1–15). Sandestin, FL, USA.

Brooks, R. A. (1987). Planning is Just a Way of Avoiding Figuring Out What to do Next. *Massachusetts Institute of Technology, Artificial Intellegence Laboratory*.

Champandard, A. J. (2007). *Behavior Trees for Next-Gen Game AI.* Retrieved 27-05-2013, from `http://aigamedev.com/open/articles/behavior-trees-part1/`

Champandard, A. J. (2012). *Understanding the Second-Generation of Behavior Trees.* Retrieved Online; accessed 27-05-2013, from `http://aigamedev.com/insider/tutorial/second-generation-bt/`

Chande, S. V., & Sinha, M. (2008). Genetic Algorithm: A Versatile Optimization Tool. *International Journal of Information Technology*, *1*(1), 7–13.

Crow, F. C. (1984, July). Summed-Area Tables for Texture Mapping. *SIGGRAPH Computer Graphics*, *18*(3), 207–212.

de Croon, G. C. H. E., de Clercq, K. M. E., Ruijsink, R., Remes, B. D. W., & de Wagter, C. (2009). Design, Aerodynamics, and Vision-Based Control of the DelFly. *International Journal of Micro Air Vehicles*, *1*(2), 71–98.

de Croon, G. C. H. E., Groen, M., de Wagter, C., Remes, B., Ruijsink, R., & van Oudheusden, B. W. (2012, June). Design, Aerodynamics and Autonomy of the DelFly. *Journal of Bioinspiration and Biomimetics*, *7*(2), 1–36.

de Wagter, C., & Amelink, M. (2007, September). Holiday50AV Technical Paper. In *3rd us-european competetion and workshop on micro air vehicles & 7th european micro air vehicle conference and flight competetion (mav07)* (pp. 1–18). Toulouse, France: ISAE.

de Wagter, C., Proctor, A. A., & Johnson, E. N. (2003, October). Vision-Only Aircraft Flight Control. In *Digital avionics systems conference* (Vol. 2). Indianapolis, IN, USA: IEEE.

de Wagter, C., Tijmons, S., Remes, B. D. W., & de Croon, G. C. H. E. (2014). Autonomous Flight of a 20-gram Flapping Wing MAV with a 4-gram Onboard Stereo Vision System. In *Ieee international conference on robotics and automation.*

Dromey, R. G. (2003, September). From Requirements to Design: Formalizing the Key Steps. In *First international conference on software engineering and formal methods* (pp. 2–11). Brisbane, Australia: IEEE.

Dromey, R. G. (2004, September). From Requirements Change to Design Change: a Formal Path. In *International conference on software engineering and formal methods* (pp. 104–113). Beijing, China: IEEE.

Eiben, A. E., Raue, P. E., & Ruttkay, Z. (1994, October). Genetic Algorithms with Multi-Parent Recombination. In Y. Davidor, H.-P. Schwefel, & R. Männer (Eds.), *Parallel problem solving from nature* (pp. 78–87). Jerusalem, Israel: Springer Berlin Heidelberg.

Fehervari, I., Trianni, V., & Elmenreich, W. (2013, June). On the Effects of the Robot Configuration on Evolving Coordinated Motion Behaviors. In *Congress on evolutionary computation* (pp. 1209–1216). Cancún, Mexico: IEEE.

Floreano, D., Dürr, P., & Mattiussi, C. (2008, January). Neuroevolution: from Architectures to Learning. *Evolutionary Intelligence*, *1*(1), 47–62.

Floreano, D., & Mondada, F. (1994). Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural-Network Driven robot. *From Animals to Animats*.

Ghallab, M., Nau, D. S., & Traverso. (2004). *Automated Planning: Theory and Practice.* Morgan Kaufmann Publishers.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning* (1st ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer programming 8*, 231-274.

Hartland, C., & Bredèche, N. (2006, December). Evolutionary Robotics, Anticipation and the Reality gap. In *Robotics and biomimetics* (pp. 1640–1645). Kunming, China: IEEE.

Heckel, F. W. P., Youngblood, G. M., & Ketkar, N. S. (2010, August). Representational Complexity of Reactive Agents. In *Computational intelligence and games* (pp. 257–264). IEEE.

Hiller, F. S., & Lieberman, G. J. (2010). *Introduction to Operations Research* (9th ed.). McGraw-Hill.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2000). *Introduction to Automata Theory, Languages, and Computation* (2nd ed.). Pearson Education.

Isla, D. (2005, March). Handling Complexity in the Halo 2 AI. In *Game developers conference.* San Francisco, CA, USA.

Jakobi, N., Husbands, P., & Harvey, I. (1995, June). Noise and the Reality Gap: The Use of Simulation in Evolutionary Robotics. In F. Morán, A. Moreno, J. J. Merelo, & P. Chacón (Eds.), *Advances in artificial life* (pp. 704–720). Granada, Spain: Springer Berlin Heidelberg.

Julian, R. C., Rose, C. J., Hu, H., & Fearing, R. S. (2013, May). Cooperative Control and Modeling for Narrow Passage Traversal with an Ornithopter MAV and Lightweight Ground Station. In *International conference on autonomous agents and multi-agent systems* (pp. 103–110). St. Paul, Minnesota, USA: IFAAMAS.

Klöckner, A. (2013a, September). Behavior Trees for UAV Mission Management. In M. Horbach (Ed.), *Informatik 2013: informatik angepasst an mensch, organisation and umwelt* (Vol. P-220, pp. 57–68). Koblenz, Germany: K öllen Druck + Verlag GmbH , Bonn.

Klöckner, A. (2013b, August). Interfacing Behavior Trees with the World Using Description Logic. In *Aiaa guidance, navigation, and control conference.* Boston, MA: American Institute of Aeronautics and Astronautics.

König, L., Mostaghim, S., & Schmeck, H. (2009). Decentralized Evolution of Robotic Behavior using Finite State Machines. *International Journal of Intellegent Computing and Cybernetics*.

Koos, S., Mouret, J.-B., & Doncieux, S. (2013, February). The Transferability Approach: Crossing the Reality Gap in Evolutionary Robotics. *Transactions on Evolutionary Computation*, *17*(1), 122–145.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press.

Koza, J. R. (1994, June). Genetic Programming as a Means for Programming Computers by Natural Selection. *Statistics and Computing*, *4*(2), 87–112.

Koza, J. R. (2003). *Genetic Algorithms and Genetic Programming.* Retrieved Online; accessed 10-05-2013, from `http://www.smi.stanford.edu/people/koza/`

Koza, J. R., & Rice, J. P. (1992, July). Automatic Programming of Robots using Genetic Programming. In *Tenth national conference on artificial intelligence* (Vol. 2, p. 9). San Jose, CA, USA: AAAI Press.

Lim, C. (2009). *An AI Player for DEFCON: an Evolutionary Approach Using Behaviour Trees.* Beng final report, Imperial College London.

Lim, C., Baumgarten, R., & Colton, S. (2010). Evolving Behaviour Trees for the Commercial Game DEFCON. In *Applications of evolutionary computation* (pp. 100–110). Springer Berlin Heidelberg.

Lotem, A., & Nau, D. S. (2000, April). New Advances in GraphHTN: Identifying Independent Subproblems in Large HTN Domains. In *The fith international conference on artificial intellegence planning and scheduling (aips).* Breckenridge, CO, USA: AAAI Press.

Meeden, L. (1998, July). Bridging the Gap Between Robot Simulations and Reality with Improved Models of Sensor Noise. In *Genetic programming* (pp. 824–831). Madison, WI, USA: Morgan Kaufmann.

Meier, L. (2013). *MAVLink Micro Air Vehicle Communication Protocol.* Retrieved Online; accessed 13-01-2013, from `http://qgroundcontrol.org/mavlink/start`

Melanie, M., & Mitchell, M. (1998). *An Introduction to Genetic Algorithms.* Cambridge, MA: MIT Press.

Miglino, O., Lund, H. H., & Nolfi, S. (1995, January). Evolving Mobile Robots in Simulated and Real Environments. *Artificial life*, *2*(4), 417–34.

Miller, B. L., & Goldberg, D. E. (1995). Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex Systems*, *9*(95006), 193–212.

Millington, I., & Funge, J. (2009). *Artificial Intelligence for Games* (2nd ed.). Morgan Kaufmann.

NaturalPoint, Inc. (2014). *Optitrack.* Retrieved Online; accessed 23-04-2014, from `http://www.naturalpoint.com/optitrack/`

Nau, D. S. (2007). Current Trends in Automated Planning. *AI magazine*, *28*.

Nelson, A. L., Barlow, G. J., & Doitsidis, L. (2009, April). Fitness Functions in Evolutionary Robotics: A Survey and Analysis. *Robotics and Autonomous Systems*, *57*(4), 345–370.

Nolfi, S. (1998, May). Adaptation as a More Powerful Tool than Decomposition and Integration: Experimental Evidences from Evolutionary Robotics. In *Ieee international conference on fuzzy systems* (Vol. 1, pp. 141–146). Ancorage, AK, USA: IEEE.

Nolfi, S. (2002, January). Power and the Limits of Reactive Agents. *Neurocomputing*, *42*(1-4), 119–145.

Nolfi, S., & Floreano, D. (2000). *Evolutionary Robotics: The Biology, Intelligence and Technology.* Cambridge, MA, USA: MIT Press.

Nolfi, S., Floreano, D., Miglino, O., & Mondada, F. (1994). How to Evolve Autonomous Robots: Different Approaches in Evolutionary Robotics. In R. A. Brooks & P. Mates (Eds.), *Artificial life iv: Proceedings of the fourth international workshop on the synthesis and simulation of living systems* (pp. 190–197). Cambridge, MA, USA: MIT Press.

Ögren, P. (2012, August). Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees. In *Aiaa guidance, navigation, and control conference* (pp. 1–8). Minneapolis, MN, USA: AIAA.

Paparazzi Community. (2013a). *Mission Planning.* Retrieved Online; accessed 17-04-2013, from `http://paparazzi.enac.fr/wiki/Mission_planning`

Paparazzi Community. (2013b). *Welcome To Paparazzi.* Retrieved 17-04-2013, from `http://paparazzi.enac.fr/wiki/Main_Page`

Perez, D., Nicolau, M., O'Neill, M., & Brabazon, A. (2011, August). Reactiveness and Navigation in Computer Games: Different Needs, Different Approaches. In *Ieee conference on computational intelligence and games (cig'11)* (pp. 273–280). Seoul, South Korea: IEEE.

Petrovi, P. (2008). Evolving Behavior Coordination for Mobile Robots using Distributed Finite-State Automata. In H. Iba (Ed.), *Frontiers in evolutionary robotics* (pp. 413–438). Intech.

Pintér-Bartha, A., Sobe, A., & Elmenreich, W. (2012, July). Towards the Light - Comparing Evolved Neural Network Controllers and Finite State Machine Controllers. In *10th international workshop on intellegent solutions in embedded systems* (pp. 83–87). Klagenfurt, Austira: IEEE.

Quinn, M., Smith, L., Mayley, G., & Husbands, P. (2002, August). Evolving Team Behaviour for Real Robots. In *Artificial life viii.* Cambridge, MA, USA: MIT Press.

Russell, S. J., & Norvig, P. (2009). *Artificial Intellegence: A Modern Approach* (3rd ed.). Prentice Hall.

Sbalzarini, I. F., Müller, S., & Koumoutsakos, P. (2000). Multiobjective Optimization using Evolutionary Algorithms. In *Summer program.* Center for turbulance research, NASA.

Scharstein, D., & Szeliski, R. (2002). A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms. *International journal of computer vision*, *47*(1), 7–42.

Smith, J. F., III, & Nguyen, T. V. H. (2007). Fuzzy Decision Trees for Planning and Autonomous Control of a Coordinated Team of UAVs. *Signal Processing, sensor fusion and target recognition XVI*, *6567 656708-1*.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving Neural Networks through Augmenting Topologies. *Evolutionary computation*, *10*(2), 99–127.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction* (Vol. 9) (No. 5). Cambridge, MA, USA: MIT Press.

Szeliski, R., & Zabih, R. (2000). An Experimental Comparison of Stereo Algorithms. In B. Triggs, A. Zisserman, & R. Szeliski (Eds.), *Vision algorithms '99* (pp. 1–19). Springer Berlin Heidelberg.

Tan, K. C., Lee, T. H., & Khor, E. F. (2001, May). Evolutionary Algorithms for Multi-Objective Optimization: Performance Assessments and Comparisons. In *Congress on evolutionary computation* (Vol. 2, pp. 979–986). Seoul, South Korea: Ieee.

Tijmons, S. (2012). *Stereo Vision for Flapping Wing MAVs.* Unpublished doctoral dissertation, Delft University of Technology.

Trianni, V. (2008). Evolutionary Robotics for Self-Organising Behaviours. In *Evolutionary swarm robotics* (Vol. 59, pp. 47–59). Springer Berlin Heidelberg.

Valmari, A. (1998). The State Explosion Problem. In W. Reisig & G. Rozenberg (Eds.), *Lectures on petri nets i: Basic models* (pp. 429–528). Springer Berlin Heidelberg.

Viola, P., & Jones, M. (2001). Robust Real-Time Object Detection. *International Journal of Computer Vision*.

Zagal, J. C., & Solar, J. Ruiz-del. (2007, March). Combining Simulation and Reality in Evolutionary Robotics. *Journal of Intelligent and Robotic Systems*, *50*(1), 19–39.