Behavior Trees for Evolutionary Robotics†

Kirk Y. W. Scheper*,**,*** Sjoerd Tijmons**
Cornelis C. de Visser**
Guido C. H. E. de Croon**
Delft University of Technology

Abstract Evolutionary Robotics allows robots with limited sensors and processing to tackle complex tasks by means of sensory-motor coordination. In this article we show the first application of the Behavior Tree framework on a real robotic platform using the evolutionary robotics methodology. This framework is used to improve the intelligibility of the emergent robotic behavior over that of the traditional neural network formulation. As a result, the behavior 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-g DelFly Explorer flapping wing micro air vehicle equipped with a 4-g 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 success rate of the optimized behavior in simulation is 88%, and the corresponding real-world performance is 54% after user adaptation. Although this leaves room for improvement, it is higher than the 46% success rate from a tuned user-defined controller.

Keywords Behavior tree, evolutionary robotics, reality gap, micro air vehicle

1 Introduction

Small robots with limited computational and sensory capabilities are becoming more commonplace. Designing effective behavior for these small robotic platforms to complete complex tasks is a major challenge. This design problem becomes even more complex when the robots are expected to collaboratively achieve a task as a swarm. A promising methodology to address this problem is found in evolutionary robotics (ER), in which a robot’s controller, and possibly its body, is optimized using evolutionary algorithms (EAs) [4, 25, 43]. This approach satisfies given computational constraints, while often resulting in unexpected solutions that exploit sensory-motor coordination to achieve complex tasks [41].

Early investigations into ER used online EAs, in which behaviors generated by evolution were evaluated on real robotic hardware. However, this process can be time consuming [17, 44] and is not widely used, although notably there has been renewed interest in online evolution with swarms of small robots [14]. With the ever improving computing technologies, offline EAs based on simulation...
have become the predominant method used in ER. However, this method has some drawbacks of its own. Simulated environments always differ to some degree from reality. The resultant artefacts from the simulation are sometimes exploited by the evolved solution strategy [17]. As a result the behavior seen in simulation can often not be reproduced on a real robotic platform. This problem has been termed the reality gap [27, 44].

The many methods that have been investigated to reduce this reality gap can be separated into three main approaches [4]. The first approach investigates the influence of simulation fidelity on the EA, with investigation focusing on the influence of adding differing levels of noise to the robot’s inputs and outputs [27, 35, 37]. It was shown that sufficient noise can deter the EA from exploiting artefacts in the simulation, but that this approach is generally not scalable, as more simulation runs are needed to distinguish between noise and true features. A notable exception to this is the work of Jakobi, who discusses the idea of combining limited but varying noise with differing levels of simulation fidelity in what he calls minimal simulations [26]. This approach requires the designer to make choices as to which aspects of the environment the robot will use before evolution even begins, limiting the solution space of the EA. Additionally, selecting the type and magnitude of the noise applied requires some foreknowledge of the environmental model mismatch, which is not always available.

The second approach focuses on coevolution. 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 [5, 54]. Alternatively, the error between the simulation and the real-world environment can be used to estimate the suitability of a learned behavior of the real robot. A multi-objective function is used to trade off simulated robotic performance and the transferability of the behavior [32].

The third approach performs adaptation of the real robot behavior after first being optimized by the EA. This can be achieved using many methods, which are differentiated by their level of supervision and how the fitness of the behavior is determined. One approach involves the use of unsupervised learning, where the neural structure and ontogenetic learning rules are optimized using evolution and are used to generate a population of adaptive individuals [18, 42, 45]. Alternatively, semi-supervised methods such as Reinforcement Learning can be used to retrain the neural nets after evolution [16]. This work shows that systems that adapt to their environments are typically more robust to the reality gap. A typical downside of this approach, however, is that the time needed for the online learning to converge may be significant. This is especially problematic for robotic platforms performing complex tasks and operating in an unforgiving environment.

One factor adding to the reality gap problem is that typically artificial neural networks (ANNs) are used to encode the robot behavior [43]. Although analysis of the evolved ANNs is possible, they do not lend themselves well to manual adaptation, hence requiring retraining algorithms to bridge the gap. Encoding the optimized behavior 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 when moving to the real robotic platform.

Traditionally, user-defined autonomous behaviors are described using finite state machines (FSMs), which have also been successfully used within ER [19, 31, 47, 48]. FSMs are very useful for simple action sequences, but quickly become illegible as the tasks get more complex, due to state explosion [39, 51]. This complexity makes it difficult for developers to modify and maintain the behavior of the autonomous agents.

A more recently developed method to describe behavior is the behavior 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 [7, 13]. BTs do not consider states and transitions the way FSMs do, but rather they consider a hierarchical network of actions and conditions [7, 23]. The rooted tree structure of the BT makes the encapsulated behavior readily intelligible for users.

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 [34, 46]. The evolution
of BTs has not yet been applied to a real-world robotic task. Operating in the real world introduces complicating factors such as state and action uncertainty, delays, and other properties of a non-deterministic and not fully known environment. There is a growing body of research into proving the operation of BTs through logic and statistical analysis, which goes a long way to improving the safety of using BTs on real vehicles [8, 30].

In this article, we perform the first investigation into the use of behavior trees in evolutionary robotics. Section 2 will describe the DelFly Explorer [53], the flapping wing robotic platform selected to demonstrate our approach as well as the fly-through-window task it had to perform. This is followed by a detailed description of the BT framework used in Section 3. Section 4 goes on to describe how offline EA techniques are used to automatically develop BTs. The results of the optimization are presented in Section 5. Additionally, the performance of the best individual from the EA is compared with a human-user-designed BT to show the efficacy of this automatically generated behavior. The implementation of both behaviors on the real-world DelFly Explorer is described in Section 6. The method used to bridge the reality gap is described in Section 7. This is followed by the real-world test results in Section 8. Finally, in Section 9, we provide a short discussion of the results and talk about how this technique can be scaled to more complex systems and used in other applications.

2 DelFly Fly-through-Window Task

The limited computational and sensory capabilities of the DelFly Explorer make it a challenge to design even the simplest behavior. As a result, the DelFly Explorer is an ideal candidate for the implementation of ER. We will give a brief description of this platform and its capabilities.

2.1 DelFly Explorer

The DelFly is a bio-inspired flapping wing micro air vehicle (MAV) developed at the Delft University of Technology (TU Delft). The main feature of its design is its biplane configuration, whose wings flap in anti-phase [9]. The DelFly Explorer is a recent iteration of this micro ornithopter design [53]. In its typical configuration, the DelFly Explorer has mass 20 g and wing span 28 cm. In addition to its 9-min flight time, the DelFly Explorer has a large flight envelope, ranging from a maximum forward flight speed of 7 m/s, to hovering, to a maximum backward flight speed of 1 m/s. A photograph of the DelFly Explorer can be seen in Figure 1. The main payload of the DelFly Explorer is a pair of lightweight cameras used to perform onboard-vision-based navigation as shown in Figure 1. Each camera is set to a resolution of $128 \times 96$ pixels with a field of view of $60^\circ \times 45^\circ$, respectively. The cameras are spaced 6 cm apart, facilitating stereo-optic vision. Using computer vision techniques, these images can be used to generate depth perception with a method called stereo vision [49]. This makes the DelFly Explorer the first flapping wing MAV that can perform active obstacle avoidance using onboard sensors, facilitating fully autonomous flight in unknown environments [53].

2.2 Fly-through-Window Task

In this article, the DelFly Explorer is tasked to navigate a square room in a search for an open window, which it must fly through, using onboard systems only. This is the most complex autonomous task yet attempted with such a lightweight flapping wing platform. Due to the complexity of finding and flying through a window, we currently limit the task to directional control: Height control can be added in future work.

Other work on the fly-through-window task includes the H²Bird 13-g flapping wing MAV [29]. Unlike the DelFly Explorer, the H²Bird used a ground-based camera and off-board image processing.
to generate heading setpoints. In this work the DelFly must perform all tasks using only onboard computation and sensing, making the task much more complex than that of the H²Bird.

2.3 Vision Systems
In the light of the task, the following vision algorithms will be running onboard the DelFly Explorer:

2.3.1 LongSeq Stereo Vision
The DelFly Explorer uses a stereo vision algorithm called LongSeq to extract depth information from the environment using its two onboard optical cameras [53]. The main principle in artificial 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 pixels is referred to as the disparity. This can be applied to entire features, to groups of pixels, or to individual pixels. The stereo vision algorithm produces a disparity map of all pixels in the images [49].

LongSeq is a localized stereo vision algorithm using line-based search. This is one candidate resulting from the tradeoff between computational complexity and image performance made by all image processing algorithms. The relatively low computational and memory requirements of LongSeq make it a good candidate for application on the limited computational hardware onboard the DelFly Explorer.

2.3.2 Window Detection
An integral image window detection algorithm is used to aid the MAV in the fly-through-window task. Integral image detection is a high-speed pattern recognition algorithm that can be used to identify features in a pixel intensity map [11, 28]. The integral image $\Pi(x, y)$ is computed as

$$\Pi(x, y) = \sum_{x' \leq x, y' \leq y} I(x', y')$$

(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 over any rectangular subsection is simplified to the following computation:

$$\text{rect}(x, y, w, h) = II(x + w, y + h) + II(x, y) - II(x + w, h) - II(x, y + h).$$

(2)

This method has been previously used to identify a dark window in a light environment by using cascaded classifiers [52]. That algorithm was designed specifically to operate when approaching a building in the daytime on a light day. Naturally, a more generalized 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).

2.4 SmartUAV Simulation Platform

SmartUAV is a flight control software (FCS) and simulation platform developed in-house at the TU Delft [1]. It is used primarily with small and micro-size 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 MAV or swarms of many MAVs. As SmartUAV is 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. In this article we will only utilize the simulation capabilities.

The BT will be placed in series, following the LongSec disparity map generation and the window detection algorithm. Within the larger SmartUAV simulation, the vision-based calculations are the most computation-intensive portion, making them 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 tradeoff, the decision loop was set to run at 10 Hz. This is a conservative estimate of the actual performance of the vision systems onboard the real DelFly Explorer.

2.5 Simplified DelFly Model

The modeling of flapping wing MAV dynamics is an active research area driven by the largely unknown micro-scale aerodynamic effects [3, 6, 9]. Due to the lack of accurate models, an existing model of the DelFly II based on the intuition of the DelFly designers will be used in this work. This model is not an accurate representation of the true DelFly II dynamics, but was sufficient for most vision based simulations previously carried out.

The DelFly II has three control inputs, namely: elevator ($\delta_e$), rudder ($\delta_r$), and thrust ($\delta_t$). The elevator and rudder simply set the control surface deflection, and the thrust sets the flapping speed. The actuator dynamics of the DelFly rudder actuator is implemented using a low-pass filter with a rise time of 2.2 s and a settling time of 3.9 s. The elevator deflection and flapping speed have no simulated dynamics and are directly set to the setpoint.

For the simulated flights in this article, the throttle setting and elevator deflection were held constant at a trim position resulting in a flight speed of 0.5 m/s and no vertical speed. Additionally, the rudder deflection was limited to a resultant maximum turn rate of 0.4 rad/s, resulting in a minimum turn radius of 1.25 m. The simulated dynamics had no coupling in the flight modes of the simulated DelFly, which is a significant simplification of real-world flight.

Now, there are some notable differences between the DelFly II and DelFly Explorer. Firstly, the Explorer replaces the rudder with a pair of ailerons, which allows the DelFly Explorer to turn
without the camera rotating around the view axis. Additionally, the DelFly Explorer is 4 g heavier and has a slightly higher wing flapping frequency. It is expected that the DelFly model mismatch will exaggerate the resultant reality gap.

3 Behavior Tree Implementation

BTs are depth-first, ordered directed acyclic graphs (DAGs) used to represent a decision process [14]. DAGs are composed of a number of nodes with directed edges. Each edge connects one node to another in such a way that, starting at the root, there is no way to follow a sequence of edges to return to the root. Unlike FSMs, BTs pursue a goal by recursively simplifying the goal into subtasks, similarly to the hierarchical task network (HTN) [15]. This hierarchy and recursive action make the BT a powerful way to describe complex behavior.

3.1 Syntax and Semantics

A BT is syntactically represented as a rooted tree structure, constructed from a variety of nodes. Each node has its individual internal function while all nodes have the same external interface, making the structure very modular. When evaluated, each node in a BT has a return status, which dictates how the tree will be traversed. In its simplest form, the return statuses are either Success or Failure. As the terms suggest, Success is returned on the successful evaluation of the node, and Failure when unsuccessful. As this does not provide much information as to the condition under which the node failed, some implementations have augmented states such as Exception or Error to provide this information.

Figure 2 shows a typical BT and node types used in this article. Basic BTs are made up of three kinds of nodes: conditions, actions, and composites [7]. Conditions test some property of the environment, while actions allow the agent to act on its environment. Conditions and actions make up the leaf nodes of the BT while the branches consist of composite nodes. Naturally, leaf nodes are developed for specific robotic platforms, depending on the available sensors and actuators.

Composite nodes, however, are not platform dependent and can be reused in any BT. Each node requires no information about its location in the tree. Only composite nodes need to know who their children are, in order to direct the flow of execution down the tree. This structure makes BTs inherently modular and reusable.
The tree execution can also be seen in Figure 2. This demonstrates how the composite nodes determine the execution path of the tree, depending on the return value of their children. To understand this flow structure we must first describe the composite node in more detail. Although many different types of composite nodes exist, we will only consider the most basic nodes in this article: selectors and sequences.

Composites evaluate their children in a fixed order, graphically represented from left to right. A selector will break execution and return Success when one of its children returns Success, or Failure when all of its children return Failure. In contrast, a sequence will break execution and return Failure when one of its children fails, or Success if all of its children return Success. The first node in the tree is called the root node, and is typically a selector with no parent. The execution of the behavior tree is referred to as a tick.

This execution framework means that not all nodes are evaluated in every tick. The leftmost nodes are evaluated first and determine the flow through the tree, implementing a sort of prioritized execution.

### 3.2 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 set_rudder. These behavior nodes are accompanied by a blackboard, which was developed to share information with the BT.

The blackboard architecture implemented for the DelFly contains five entries: window x-location ($x$), window response ($\alpha$), total disparity ($\Sigma$), horizontal disparity difference ($\Delta$), and rudder deflection ($r$). The first four are condition variables, and the last is used to set the BT action output. The condition variables are set before the BT is ticked, and the outputs are passed to the DelFly FCS after the tick is complete. Note that this implementation of a BT has no explicit concept of memory or time.

The condition nodes check if some environmental variable is greater than or less than a given threshold. This means that each condition node has two internal settings: the environmental parameter to be checked, and the threshold. The action node set_rudder sets the DelFly rudder input and therefore only has one internal setting. Actions were defined to always return Success.

### 3.3 User-Designed Behavior Tree

A human-designed behavior was used as a baseline to judge the performance of the genetically optimized solution. The designed tree has 22 nodes and the structure of the BT shown in Figure 3. The behavior is made up of four main sub-behaviors:

- Window tracking, based on window response and location in the frame: Try to keep the window at the center of the frame.
- Go straight when disparity is very low: default action; also helps when looking directly through window into next room.
• Wall avoidance when disparity is high: bidirectional turns to avoid collisions with walls; also helps search for window.
• Action hold when disparity is very high: ensures the chosen action is not changed when already evading a wall.

After validation of this BT, it was observed that for 250 random initializations in the simulated environment, 82% of flights were successful. This behavior is good, but suffers from one main flaw, which was observed during the validation: The bidirectional wall avoidance in a square room can result in the DelFly getting caught in and crashing into corners. There are available methods to correct for this behavior [50, 55], but as this is a conceptual error typical of human-designed systems, we will keep this behavior as is. Figure 4 shows the path of successful and failed flight realizations of DelFly with the user-defined behavior.

4 Evolutionary Algorithm

Evolutionary algorithms are a population-based metaheuristic global optimization method inspired by Darwinian evolution [20, 24]. A population of feasible solutions for a particular problem is made up of a number of individuals. The fitness of each individual is measured by some user-defined, problem-specific objective function. The fitness of the individuals is evaluated each generation. Successful individuals are selected to generate the next generation, using the genetic recombination method crossover. Each generated child may also be subject to mutation, where individual parts of their genes are altered. These operations allow the EA to effectively explore and exploit the available search space [36].

There are many implementations of EAs, each with a different method to encode the genetic material in the individuals [16, 20, 33]. In this article we will use an EA to optimize the behavior for a task using the BT framework. The custom EA for BTs used in this work is described in the following subsections.

4.1 Genetic Operators

(a) Initialization: The initial population of $M$ individuals is generated using the grow method [33]. Nodes are selected at random to fill the tree with composite, action, and condition nodes.

![Figure 4. Path of successful (x) and unsuccessful (o) flight initializations of DelFly with the user-defined behavior (top-down view). Line types denote different decision modes: solid—window tracking; dash—default action in low disparity; dot-dash—wall avoidance; dot—action hold.](image-url)
with equal probability. Once a composite node is selected, there is equal probability for a sequence or selector. This was done because more leaf nodes than branch nodes are typically needed in trees.

The grow method results in variable-length trees where every composite node is initialized 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 EA search.

(b) Selection: A custom implementation of tournament selection is used in this article [38]. This is implemented by first randomly selecting a subgroup of \( s \) individuals from the population. This subgroup is then sorted in order of the individuals’ 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 for reducing the size of the BTs.

(c) Crossover: Crossover is an operation where the composition of two or more parents is recombined to produce offspring. In this article we use two-parent crossover to produce two children. Each parent is selected from a different tournament selection. The percentage of the new population formed by crossover is defined by the crossover rate \( P_c \). The point in the BT used to recombine the parents is selected at random. This selection is independent of its type or its location in the tree. Crossover can be applied to any node location till the maximum tree depth, after which nodes are ignored. Figures 5 and 6 graphically show this process.

(d) Mutation: Mutation is implemented using two methods, namely: micromutation and macromutation (also referred to as headless-chicken crossover [2]). Micromutation only affects leaf nodes and is implemented as a reinitialization of the node with new operating parameters. Macromutation is implemented by replacing a selected node with a randomly generated tree that is limited in depth by the maximum tree depth. This is functionally identical to crossover with a randomly generated BT. The probability that mutation is applied to a node is given by the mutation rate \( P_m \). Once a node has been selected for mutation, the probability that macromutation will be applied rather than micromutation is given by the headless-chicken crossover rate \( P_{hc} \).
Stopping rule: Like many optimization methods, EAs can be affected by overfitting. As a result, an important parameter in an EA is when to stop the evolutionary process. Additionally, due to the large number of simulations required to evaluate the performance of the population of individuals, placing an upper limit on the number of generations can help avoid unnecessarily long computational time.

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

### 4.2 Fitness Function

The two main performance metrics used to evaluate the DelFly in the fly-through-window task are the success rate and the tree size. The fitness function was chosen to encourage the EA to converge on a population that flies through the window as often as possible. After trying several different forms of fitness functions, a discontinuous function was chosen such that a maximum score is received if the MAV flies through the window, and a score inversely proportional to its distance to the window if not. The fitness \(F\) is defined as

\[
F = \begin{cases} 
1 & \text{if success} \\
\frac{1}{1 + 3|e|} & \text{else}
\end{cases}
\]  

(3)

where success is defined as flying through the window and \(e\) is the vector from the center of the window to the location of the MAV at the end of the simulation. This particular form of fitness function was selected to encourage the DelFly to try to get close to the window, with a maximum score if it flies through. The values selected are not very sensitive and were chosen at the discretion of the designer. Changing the gain of the error term affects the selection pressure of the EA.

Although not incorporated in the fitness function, we will also analyze some secondary parameters that are not vital to the performance of the DelFly. These define the suitability of its behavior from a user point of view and define the characteristics of a given fly-through-window behavior. These parameters are the angle of window entry, time to success, and distance from the center of the window at fly-through.

### 5 DelFly Task Optimization

#### 5.1 Simulated 3D Environment

The environment chosen to evaluate the DelFly in simulation was an 8 × 8 × 3-m room with textured walls, floor, and ceiling. A 0.8 × 0.8-m window was placed at the center of one wall. Another identical room was placed on the other side of the windowed wall to ensure that the stereo algorithm had sufficient texture to generate matches for the disparity map when looking through the window.

As it is not the purpose of this research to focus on the vision systems, the environment was rather abundantly textured. A multicolored stone texture pattern was used for the walls, a wood pattern was used for the floor, and a concrete pattern was used for the ceiling, as shown in Figure 7. The identically textured walls ensure that the behavior must identify the window and not any other features to aid in its task.

#### 5.2 Experimental Setup

The evolved DelFly behavior should be robust and therefore must fly through the window as often as possible. To evaluate this, each individual behavior must be simulated multiple times in each generation defined by the parameter \(k\). Each run is characterized by a randomly initiated location in the room and a random initial heading.
Initially, it was observed that randomly changing the initializations in every generation made it difficult for evolution to determine whether the behavior in subsequent generations improved due to the preceding behavior or due to the initialization. To remedy this, initial conditions are held over multiple generations until the elite members of the population (characterized by $P_e$) are all successful. Once all the elite members are successful in a given initialization run, the initial condition in question is replaced by a new random initialization. Each simulation run is terminated when the DelFly crashes, flies through the window, or exceeds a maximum simulation time of 100 s.

For the EA to converge to a near-optimum solution, the crossover rate must be high enough to push the optimization to exploit the local maxima. Additionally, the mutation rate must be high enough to explore the state space, though not too high to prematurely exit current solutions. The characteristic parameters for optimization shown in this article are shown in Table 1. The parameter combination selected is naturally only one realization among many possibilities. The relatively large number of runs per individual selected should promote the development of robust flight behavior. This however increases the total simulation time needed to evaluate each generation, hence affecting the choice of population size.

The maximum tree depth is measured with the root node as depth 0. The maximum tree size can be determined by $\maxchildren^{\maxdepth}$. So a tree depth of 6 with at most 6 children per composite was used, resulting in an upper limiting tree size of over 46,000 nodes. This is however not likely, as the node type selected in the trees is chosen at random over composite, condition, and action.

### 5.3 Optimization Results

The main parameter that dictates the progress of the genetic optimization is the mean fitness of the population. Figure 8 shows the population mean fitness as well as the mean fitness of the best individual in each generation. It can be seen in Figure 8 that at least one member of the population is quickly bred to fly through the window quite often. Additionally, as the generations progress and new initializations are introduced, the trees have to adjust their behavior to be more generalized. The mean fitness also improves initially and 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 to avoid premature conversion of the EA.

The other main parameter that 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...
in Figure 9. This figure shows that the average tree size began at about 5000 nodes and initially increased to 7000 before steadily dropping to around 1000 nodes at generation 50. The trees then slowly continued to reduce in size and eventually drop 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, removing redundant nodes that have no effect on the final behavior, resulted in a tree with 8 nodes. The structure of the tree can be seen graphically in Figure 10.

Figure 11 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 then oscillates around about 80% success. In early generations the variation of success rate from one generation to the next is larger than in later generations.

Figures 9 and 11 suggest 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 indicates that the selected initial conditions used for training are in fact not representative for the full set of initial conditions. One method to make the initial conditions more difficult is to adapt the environment to actively challenge the EA in a sort of predator-prey optimization. Alternatively, the fact that the behavior does not continue to improve over the 80% mark may indicate that the sensory inputs used by the DelFly are not sufficient.

The optimized BT was put through the same validation set as used with the user-defined behavior, resulting in a success rate of 88%. The performance characteristics of the best individual from the optimization, compared to those from the user-defined BT, are summarized in Table 2.
optimized BT has a slightly higher success rate than the user-defined BT, but with significantly less nodes. The results for the secondary parameters suggest that the genetically optimized behavior typically has a sharper window entry angle and enters the window closer to the edge than the user-defined behavior. It also has a longer time to window fly-through, as it circles the room more often than the user-defined behavior. This result highlights the fact that EAs typically only optimize the task explicitly described in the fitness function, sometimes at the cost of what the user might think is beneficial.

The successful flight shown in Figure 12 shows that the behavior correctly avoids collision with the wall, makes its way to the center of the room, and then tracks into the window. Analyzing the BT from Figure 10, the logic to fly through the window can be separated into three sub-behaviors:

- Slight right turn (default action) if disparity is low.
- Maximum right turn to evade walls if disparity is high (unidirectional avoidance).
- Moderate left turn if window is detected.

Although this very simple behavior seems to be very successful, Figure 12 also highlights one pitfall of this solution. As the behavior does not use the location of the window in the frame for its guidance, it is possible to drift off center, lose the window in the frame, and enter a wall avoidance turn quite close to the wall, resulting in a collision.

Table 2. Summary of validation results.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>User-defined</th>
<th>Genetically optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success rate</td>
<td>82%</td>
<td>88%</td>
</tr>
<tr>
<td>Tree size</td>
<td>26</td>
<td>8</td>
</tr>
<tr>
<td>Mean flight time [s]</td>
<td>32</td>
<td>40</td>
</tr>
<tr>
<td>Mean approach angle [deg]</td>
<td>21</td>
<td>34</td>
</tr>
<tr>
<td>Mean distance to center [m]</td>
<td>0.08</td>
<td>0.15</td>
</tr>
</tbody>
</table>
These results show that, for the given fitness function and optimization parameters, the genetic optimization was very successful. The resultant BT was both smaller and better performing than the user-defined tree.

6 DelFly Onboard Flight Testing

The BT was implemented on the camera module of the DelFly Explorer, which is equipped with an STM32F405 processor operating at 168 MHz with 192-kB RAM. The BT is placed in series with the stereo vision and window detection algorithms, as was done in simulation, and was found to run at \( \approx 12 \) Hz. 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 100 Hz.

6.1 Test 3D Environment

The environment designed to test the MAV was a 5 × 5 × 2-m room with textured walls. A 0.8 × 0.8-m window was placed at the center of one wall. The area behind the window had a regular texture. Artificial texture was added 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. A sample photograph of the room can be seen in Figure 13.

6.2 Experiment Setup

At the beginning of each run, the DelFly was initially flown manually and correctly trimmed for flight. It was then flown to a random initial position and pointing direction in the room. At this point the DelFly was set to autonomous mode, where the DelFly flight computer implements the commands received from the BT. The flight continued until either the DelFly succeeded in flying through the window, it crashed, or the test took longer than 60 s. As the BT controls the horizontal dynamics only, the altitude was actively controlled by the user during flight; it was maintained around the height of the center of the window.

All flights were recorded by video camera as well as an Optitrack vision-based motion tracking system [40]. Optitrack 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 from the center of the window can be determined.

7 Crossing the Reality Gap

The flight speed of the DelFly was set to $\approx 0.5 \text{ m/s}$, the same as was used in simulation. However, there were significant differences observed between the system simulated in SmartUAV and that in the flight tests. The most significant observations are summarized in Table 3. In short, the turn radius was smaller and the actuator response was faster and asymmetric. Additionally, aileron actuation would 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 around the window, which affected the DelFly’s flight path. These drafts would typically slow down the DelFly’s forward speed and push it to one side of the window.

With these significant differences between the model used to train the BTs and the real DelFly, there was a clear reality gap present. Initially both behaviors were unsuccessful in flying through the window. To adjust the behavior to improve the performance, we first considered the notion of

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Simulated</th>
<th>Reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flight speed [m/s]</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Minimum turn radius [m]</td>
<td>1.25</td>
<td>0.5</td>
</tr>
<tr>
<td>Actuator response time [s]</td>
<td>2.2</td>
<td>$&lt;1$</td>
</tr>
<tr>
<td>Decision loop speed [Hz]</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>Actuator deflection</td>
<td>Symmetric</td>
<td>Asymmetric</td>
</tr>
<tr>
<td>Environmental</td>
<td>No disturbances</td>
<td>Drafts</td>
</tr>
</tbody>
</table>

Table 3. Summary of the reality gap.
success as defined by Jakobi [26]. In his article he suggested that the performance of the robotic system should be judged on a subjective measure of how reliably the robot performs the task in reality, with no consideration of how the behavior achieves it. In the case of this article, that would simply be how often the DelFly flies through the window.

We initially tried to directly adjust the behavior in reality without comparing it with the behavior seen in simulation. To improve the fly-through-window performance, we mainly considered the final portion of the flight, but this proved ineffective. This results from the fact that the embodied agent’s success is tightly coupled with interaction of the robot’s sub-behaviors during the entire flight. For example, the way the DelFly wall avoidance sub-behavior performed determined its approach to the window in such a way that the window approach sub-behavior would be successful. This suggests then that to achieve a task reliably in reality the robot’s behavior must be similar to that observed in simulation for all sub-behaviors.

The insight into what parameters to change, and how, comes from the user’s understanding of the BT. From this the user can identify individual sub-behaviors. The technique of grouping nodes into sub-behaviors can be seen in Figures 3 and 10. This segmentation of the behavior helps to identify individual gaps, simplifying the behavior update process.

To demonstrate this, let us first look at the evolved behavior tree shown in Figure 10, which can be considered as made up of three sub-behaviors. Let us first look at the window detection sub-behavior. We flew the DelFly around our test room and observed that the window response value was never achieved with the certainty value 69 (a lower value represents higher certainty that a window is in the frame). We increased the threshold of node 7 till the node would be activated by the window but false positives from other locations would not be likely.

Let us now investigate the wall avoidance sub-behavior. This mode is entered when the total disparity is larger than a threshold set by node 3. Observing the behavior in Figure 12, we see that the DelFly tries to circle in around the center of the room, entering the wall avoidance mode at ≈4 m from the wall in the 8 × 8-m room. This would suggest that the real DelFly should enter this mode at ≈2.5 m in the real 5 × 5-m room, so the threshold in node 3 should be changed accordingly.

It should be noted that it appears that evolution has optimized the DelFly behavior to fly through windows in square rooms. The approach of avoiding walls at a fixed distance to line the DelFly up for the window entry would be more difficult if the window were not at the center of the wall or if the room size changed. This reiterates the strong coupling between optimized behavior and the environment that is characteristic of ER. It is therefore essential to vary the environment sufficiently to encourage the EA to converge to solutions robust to changes in the environment.

Last but not least, applying this consideration to the wall avoidance action, the simulated DelFly had a minimum turn radius of 1.25 m, which was much larger than in reality. A scaling factor was applied to increase the turn radius to that seen in simulation.

Using this approach, tuning these parameters took about three flights of about 3 min each to result in behavior similar to that seen in simulation. The updated behavior can be seen in Figure 14.

This same approach was used with the user-defined BT with significantly more nodes, and it took a total of eight flights of about 3 min each to tune the parameters to mimic the behavior observed in simulation. The updated behavior can be seen in Figure 15.

---

**Figure 14.** Graphical depiction of genetically optimized BT after modification for real-world flight. Encapsulating boxes highlight updated nodes. $x$ is the position of the center of the window in the frame, $\alpha$ is the window response value, $\Sigma$ is the total disparity, $\Delta$ is the horizontal difference in disparity, and $r$ is the aileron deflection setting for the DelFly Explorer.
Flight Test Results

26 test flights were conducted for both the user-defined behavior and the genetically optimized BT. The results of the tests are summarized in Table 4. It can be seen that the success rate of both behaviors is reduced, but notably, the relative difference of the two behaviors is maintained. Additionally, the other performance parameters, which are the characteristic behavior descriptors, are similar to those seen in simulation. This suggests that the user adaptation of the real behavior to emulate the simulated behavior was successful.

The mean distance to the center of the window was larger for the user-defined behavior than observed in simulation. This may be the result of the draft around the window pushing the DelFly to the edges of the window. This draft would also push the approaching DelFly into the window edge on some approaches. The time to success was shorter for both behaviors than the values observed in simulation. This is mainly due to the smaller room size used in reality.

Notably, the user-defined behavior showed the characteristic failure of being caught in corners. This happened in 4/26 flights for the user-defined behavior, but not once in the genetically optimized behavior. This is representative of the observations of the behavior in simulation, a fundamental deficiency of the bidirectional wall avoidance in a room with corners. This observation additionally suggests that the behavior seen in simulation is effectively transferred to the real DelFly. Figures 16 and 17 show the last 7 s of the user-defined behavior for all flights grouped into successful and unsuccessful tests, respectively. The Optitrack flight tracking system did not successfully track the DelFly in all portions of the room, so that there were some dead areas, but did accurately capture the final segment of the window approach.

Table 4. Summary of flight test results.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>User-defined</th>
<th>Genetically optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success rate</td>
<td>46%</td>
<td>54%</td>
</tr>
<tr>
<td>Mean flight time [s]</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>Mean approach angle [deg]</td>
<td>16</td>
<td>37</td>
</tr>
<tr>
<td>Mean distance to window center [m]</td>
<td>0.12</td>
<td>0.12</td>
</tr>
</tbody>
</table>

8 Flight Test Results

An accompanying video with some of the test flights can be found at https://www.youtube.com/watch?v=CBJ0O2dH4&feature=youtu.be.
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 on the wall due to being caught in corners can also be seen. Figure 18 shows one full successful and one unsuccessful flight of the DelFly user-defined behavior.

Similarly, Figures 19 and 20 show the successful and unsuccessful flights of the genetically optimized behavior as captured from the Optitrack system. In these figures it can be seen that the flight tracks of genetically optimized behavior are tightly grouped, with the same behavior repeated over multiple flights. The DelFly always approaches from about the center of the room with a coordinated left-right turn as described earlier. It can be seen that some of the unsuccessful flights

![Figure 16. Flight path tracks of the last 7 s of all successful flights for the user-defined behavior. o represents start location of each flight.]

![Figure 17. Flight path tracks of the last 7 s of all unsuccessful flights for the user-defined behavior. o represents start location of each flight.]

Artificial Life  Volume 22, Number 1  41
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 21 shows one entire successful and one unsuccessful flight of the genetically optimized
behavior in more detail. The typical failure mode was turning into the edge of the window in the
final phase of the flight. This is likely mainly due to the drafts around the window. Additionally, the
faster decision rate of the BT in reality, combined with the faster dynamics of the vehicle, may play a
role here.

The fact that the real-world test was conducted in a different-size room than tested in simulation
must have an effect on the success rate. In the future it would be interesting to observe the con-
verged behavior if the simulated room is not kept constant during evolution. It is expected that this
will result in behavior more robust to changes in the environment.

Figure 19. Flight path tracks of the last 7 s of all successful flights for the genetically optimized behavior. o represents start
location of each flight.

Figure 18. Flight path tracks showing one complete successful (dash-dot) and one unsuccessful (solid) flight for the user-
defined behavior. o represents start location of test. Dotted path shows where tracking system lost lock of the DelFly.
The failure mode of hitting the window edge for both behaviors may be in part due to 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 behaviors equally, so they would not affect the comparison of behaviors.

The fact that both the user-defined and genetically optimized behaviors were initially not able to fly through the window but after user adaptation were able to fly through more than 50% of the time shows that the reality gap was actively reduced by the user. These results show that it is feasible to automatically evolve behavior on a robotic platform in simulation using the BT description language. This method gives the user a high level of understanding of the underlying behavior and the tools to adapt the behavior to improve the performance and reduce the reality gap. Using

Figure 20. Flight path tracks of the last 7 s of all unsuccessful flights for the genetically optimized behavior. o represents start location of each flight.

Figure 21. Flight path tracks showing one complete successful (dash-dot) and one unsuccessful (solid) flight for the genetically optimized behavior. o represents start location of test. Dotted path shows where tracking system lost lock of the DelFly.
this technique, an automated behavior was shown to be at least as effective as, if not better than, a user-defined system in simulation, with similar performance on a real-world test platform.

9 Discussion

9.1 Abstraction of Input-Output Behavior
In this article we use the descriptive and user-legible framework of the BT to improve the user’s understanding of the solution strategy optimized through evolution. With this insight, the user can identify and reduce the resultant reality gap when moving from simulation to reality. This approach therefore necessitates that the elements of the tree also be conceptually tangible for the user, as such a higher level of abstraction was used for the sensory inputs. Unlike standard approaches, which use ANNs where raw sensor data is used as input, we first preprocess the data into a form that a user can understand. The only question then is how we determine what is the best set of inputs to the robotic platform that will facilitate a robust and effective solution to be optimized by evolution.

Now, compared to typical ER approaches, preprocessing the inputs may affect the level of emergence of the EA, as seen in Harvey et al. [22]. That article has demonstrated a robot completing an object detection task that was simplified by an EA to the correlation of just a few image pixels. This level of optimization may not be possible if the inputs are preprocessed. However, preprocessing the input data typically reduces its dimensionality, thereby reducing the search space of the EA. This reduction in search space is crucial as we implement this technique on more complex tasks and robotic platforms.

Robotic actions are typically not robust, as they are susceptible to unmodeled simulator dynamics and changes in the operating environment. For example, in this article we set the output of the BT to be the rudder deflection, which in hindsight is not a very robust parameter to control. It might have been more effective to control the turn rate and have a closed loop control system controlling the actuator deflection. The closed loop controller would reduce the BT’s reliance on the flight model in simulation. This would make the behavior more robust on the real robot, inherently reducing the reality gap. The concept of using nested loops to bound control systems in order to improve robustness has long been used in control theory.

Considering the reality gap, recent work suggests that limiting the EA to a set of predefined modules can actually improve the optimized behavior with respect to the eventual reality gap [19]. In this work, Francesca et al. compare an optimized FSM using a limited set of predefined modules with a traditional system using an ANN. The two systems performed similarly in simulation, but the ANN performed significantly worse in reality, while the FSM maintained its performance. Francesca et al. present their work in the context of the bias-variance tradeoff; they suggest that the introduction of the appropriate amount of bias will reduce the variance of the optimized system, thereby improving its generality. Bias can be introduced into an optimization problem by limiting the representational power of the system, which in this case is achieved by limiting the options of the optimization to a limited input-output state space [12]. This idea can also be considered in the context of this article, where the limitation of the state space is not a hindrance or a limitation of the system, but is in fact a benefit of this approach.

The abstraction of the behavior from the low-level sensor inputs and actuator output importantly not only introduces a bias, but additionally shields the behavior from the simulation mismatch causing the reality gap. The improved intelligibility, in combination with the improved generalizability and robustness to the reality gap, should ultimately make the approach presented in this article more suitable than conventional ER approaches for extensive use in real robots attempting complex tasks.

9.2 Scalability
The task completed in this article is more complex than other ER tasks typically cited in literature. Yet on the larger scale of autonomous navigation this task is just a start. To facilitate growing task
complexity we will recommend some points for future research. Firstly, it is interesting to investigate the implementation of memory and time to the BT. Memory could be implemented as elements of the blackboard that are not outputs of the BT to the platform but rather just internal variables. Time could be added by including a running state to the nodes, where they would hold till the action is completed. Alternatively, an explicit timer node could be added that would run for a given number of ticks.

Another point worth consideration is the addition of a link node to the BT framework. This node creates a symbolic link to a static BT branch outside the tree. Evolution could select branches of its own behavior that could be linked and reused in other parts of the tree. This should help the optimization to reuse already developed behavior effectively throughout the tree. It would provide the EA with not only the raw materials to build the behavior, but the ability to save combinations of these raw materials in a blueprint that can be reused at will.

With that said, the technique described in this article is dependent on the user’s understanding of the underlying robotic behavior, so how does this change with the growing task complexity? We showed in this article that the BT can be broken down into sub-behaviors, which helps the user to understand the global behavior. The prioritized selection of behaviors, based on their location in the tree, creates an inherent hierarchical structure. This structure will automatically group the nodes of a sub-behavior spatially in the tree. This makes the identification of the sub-behaviors straightforward. Tuning of the sub-behaviors would be accomplished using a divide and conquer approach, one sub-behavior at a time.

9.3 Evolution of Behavior Trees for Behavioral Modeling

The BT framework could also be used to model existing behavior or cognition of robots or animals [21]. This would be in a similar vein to a recent ER study on odor source identification, in which the insight into the evolved neural controller’s strategy was verified by constructing an equivalent FSM controller [10]. Instead of manually designing such a controller, EAs could be used to optimize a BT to best approximate the behavior of a robot or animal. The BT, optimized to mimic reality, would give researchers increased insight into the underlying system dynamics. To mention a few examples, this approach can be applied to self-organization, swarming, emergence, and predator-prey interaction.

10 Conclusion

We conclude that the increased intelligibility of the behavior tree framework does give a designer increased understanding of the automatically developed behavior. The low computational requirement of evaluating the behavior tree framework makes it suitable for operating onboard platforms with limited capabilities, as was demonstrated on the 20-g DelFly Explorer flapping wing MAV. It was also demonstrated that the behavior tree framework provides a designer with the tools to identify and adapt the learnt behavior on a real platform to reduce the reality gap when moving from simulation to reality.

Future work will also investigate further into optimizing the parameters of the evolutionary algorithm used in this article. Multi-objective fitness functions and adaptive simulated environments are possible avenues to improve the generality of the optimized behavior. Additionally, work will be done on investigating how behavior trees scale within evolutionary optimization, in terms of behavior node types but also in terms of task complexity. Regarding the DelFly, the most immediate improvement would be extending the automated control to include height, facilitating extended fully autonomous flight.

References


